The magic of Algorithms!

Lectures on some algorithmic pearls

Paolo Ferragina, Università di Pisa

These notes should be an advise for programmers and software engineers: no matter how much smart you are, the so called *"5-minutes thinking"* is not enough to get a reasonable solution for your real problem, unless it is a toy one! Real problems have reached such a large size, machines got so complicated, and algorithmic tools became so sophisticated that you cannot improvise to be an *algorithm designer*: you should be trained to be one of them!

These lectures provide a witness for this issue by introducing challenging problems together with elegant and efficient algorithmic techniques to solve them. In selecting their topics I was driven by a twofold goal: from the one hand, provide the reader with an *algorithm engineering toolbox* that will help him/her in attacking programming problems over massive datasets; and, from the other hand, I wished to collect the stuff that I would have liked to see when I was a master/phd student!

The style and content of these lectures is the result of many hours of highlighting and, sometime hard and fatiguing, discussions with many fellow researchers and students. Actually some of these lectures composed the courses in Information Retrieval and/or Advanced Algorithms that I taught at the University of Pisa and in various International PhD Schools, since year 2004. In particular, a preliminary draft of these notes were prepared by the students of the *"Algorithm Engineering"* course in the Master Degree of Computer Science and Networking in Sept-Dec 2009, done in collaboration between the University of Pisa and Scuola Superiore S. Anna. Some other notes were prepared by the Phd students attending the course on *"Advanced Algorithms for Massive DataSets"* that I taught at the BISS International School on Computer Science and Engineering, held in March 2010 (Bertinoro, Italy). I used these drafts as a seed for some of the following chapters.

My ultimate hope is that reading these notes you'll be pervaded by the same pleasure and excitement that filled my mood when I met these algorithmic solutions for the first time. If this will be the case, please read more about Algorithms to find inspiration for your work. It is still the time that *programming is an Art*, but you need the good *tools* to make itself express at the highest beauty!

P.F.

Contents

1

Prologo

"This is a rocket science but you
don't need to be a rocket
scientist to use it"
The Economist, September 2007

The main actor of this book is *the Algorithm* so, in order to dig into the beauty and challenges that pertain with its ideation and design, we need to start from one of its many possible definitions. The OXFORD ENGLISH DICTIONARY reports that an algorithm is, informally, *"a process, or set of rules, usually one expressed in algebraic notation, now used esp. in computing, machine translation and linguistics"*. The modern meaning for Algorithm is quite similar to that of *recipe, method, procedure, routine* except that the word Algorithm in Computer Science connotes something more precisely described. In fact many authoritative researchers have tried to pin down the term over the last 200 years [3] by proposing definitions which became more complicated and detailed nonetheless, hopefully in the minds of their proponents, more precise and elegant. As algorithm designers and engineers we will follow the definition provided by Donald Knuth at the end of the 60s [7, pag 4]: an Algorithm is *a finite, definite, effective procedure, with some output.* Although these five features may be intuitively clear and are widely accepted as requirements for a sequence-of-steps to be an Algorithm, they are so dense of significance that we need to look into them with some more detail, even because this investigation will surprisingly lead us to the scenario and challenges posed nowadays by algorithm design and engineering, and to the motivation underling these lectures.

Finite: *"An algorithm must always terminate after a finite number of steps ... a very finite number, a reasonable number."* Clearly, the term "reasonable" is related to the *efficiency* of the algorithm: Knuth [7, pag. 7] states that *"In practice, we not only want algorithms, we want good algorithms"*. The "goodness" of an algorithm is related to the use that the algorithm makes of some precious *computational resources* such as: time, space, communication, I/Os, energy, or just simplicity and elegance which both impact onto the coding, debugging and maintenance costs!

Definite: *"Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case"*. Knuth made an effort in this direction by detailing what he called the "machine language" for his *"mythical MIX...the world's first polyunsaturated computer"*. Today we know of many other programming languages such as `C/C++`, `Java`, `Python`, etc. etc. All of them specify a set of instructions that the programmer may use to describe the procedure underlying his/her algorithm in an unambiguous way: "unambiguity" here is granted by the formal semantics that researchers have attached to each of these instructions. This eventually means that anyone reading the algorithm's description will interpret it in a precise way: nothing will be left to personal mood!

Effective: *"... all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using paper and pencil".* Therefore the notion of "step" invoked in the previous item implies that one has to dig into a complete and deep understanding of the problem to be solved, and then into logical well-definite structuring of a step-by-step solution.

Procedure: *"... the sequence of specific steps arranged in a logical order".*

Input: *"... quantities which are given to it initially before the algorithm begins. These inputs are taken from specified sets of objects".*

Output: *"... quantities which have a specified relation to the inputs".*

In this booklet we will not use a *formal* approach to algorithm description, because we wish to concentrate on the theoretically elegant and practically efficient ideas which underlie the algorithmic solution of some interesting problems, without being lost in the maze of programming technicalities. So in every lecture we will take an interesting *problem* coming out from a *real/useful application* and then propose *deeper and deeper* solutions of increasing sophistication and improved efficiency, taking care that this will not necessarily correspond to increasing the complexity of algorithm's description. Actually, problems were selected to admit *surprisingly* elegant solutions that can be described in few lines of code! So we will opt for the current practice of algorithm design and describe our algorithms either *colloquially* or by using *pseudo-code* that mimics the most famous `C` and `Java` languages. In any case we will not renounce to be as much rigorous as it needs an algorithm description to match the five features above.

Elegance will not be the only feature of our algorithm design, of course, we will also aim for *efficiency* which commonly relates to the *time/space complexity* of the algorithm. Traditionally time complexity has been evaluated as a function of the input size n by counting the (maximum) number of steps, say $T(n)$, an algorithm takes to complete its computation over an input of n items. Since the maximum is taken over all inputs of that size, the time complexity is named *worst case* because it concerns with the input that induces the worst behavior in time for the algorithm. Of course, the larger is n the larger is $T(n)$, which is therefore non decreasing and positive. In a similar way we can define the (worst-case) *space complexity* of an algorithm, as the maximum number of memory cells it uses for its computation over an input of size n. This approach to the *design* and *analysis* of algorithms assumes a very simple model of computation, known as *model of Von Neumann* (aka Random Access Machine, *RAM* model). This model consists of a CPU and a memory of infinite size and constant-time access to each one of its cells. Here we argue that every step takes a fixed amount of time on a PC, which is the same for any operation: being it arithmetic, logical, or just a memory access (read/write). Here one postulates that it is enough to *count* the number of steps executed by the algorithm in order to have an "accurate" estimate of its execution time on a real PC. Two algorithms can then be *compared* according to the *asymptotic behavior* of their time-complexity functions as $n \longrightarrow +\infty$, the faster is growing the time complexity over inputs of larger and larger size, the worse is its corresponding algorithm. The robustness of this approach has been debated for a long time but, eventually, the RAM model dominated the algorithmic scene for decades (and is still dominating it!) because of its simplicity, which impacts on algorithm design and evaluation, and its ability to estimate the algorithm performance "quite accurately" on (old) PCs. Therefore it is not surprising that most introductory books on Algorithms take the RAM model as a reference.

But in the last ten years things have changed significantly, thus highlighting the need for a *shift* in algorithm design and analysis! Two main changes occurred: the architecture of modern PCs became more and more sophisticated (not just one CPU and one monolithic memory!), and input data have exploded in size ("$n \longrightarrow +\infty$" does not live only in the theory world!) because they are abundantly generated by many sources: such as DNA sequencing, bank transactions, mobile communications, Web navigation and searches, auctions, etc. etc.. The first change turned the RAM model into an unsatisfactory abstraction of modern PCs; whereas the second change made the design

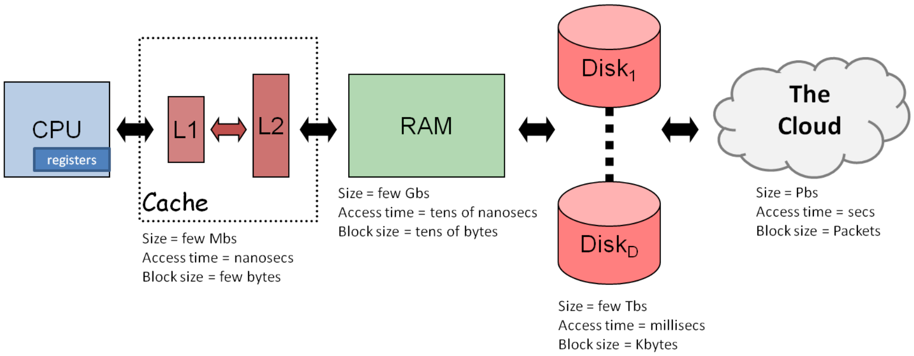


FIGURE 1.1: An example of memory hierarchy of a modern PC.

of *asymptotically good* algorithms ubiquitous and fruitful not only for dome-headed mathematicians but also for a much larger audience because of their impact on business [2], society [1] and science in general [4]. The net consequence was a revamped scientific interest in algorithmics and the spreading of the word "Algorithm" to even colloquial speeches!

In order to make algorithms effective in this new scenario, researchers needed new models of computation able to abstract in a better way the features of modern computers and applications and, in turn, to derive more accurate estimates of algorithm performance from their complexity analysis. Nowadays a modern PC consists of one or more CPUs (multi-core?) and a very complex hierarchy of memory levels, all with their own technological specialties (Figure 1.1): L1 and L2 caches, internal memory, one or more mechanical or SSDisks, and possibly other (hierarchical-)memories of multiple hosts distributed over a (possibly geographic) network, the so called "Cloud". Each of these memory levels has its own cost, capacity, latency, bandwidth and access method. The closer a memory level is to the CPU, the smaller, the faster and the more expensive it is. Currently nanoseconds suffice to access the caches, whereas milliseconds are yet needed to fetch data from disks (aka I/O). This is the so called *I/O-bottleneck* which amounts to the astonishing factor of $10^5 - 10^6$, nicely illustrated by Tomas H. Cormen with his quote:

> *"The difference in speed between modern CPU and (mechanical) disk technologies is analogous to the difference in speed in sharpening a pencil using a sharpener on one's desk or by taking an airplane to the other side of the world and using a sharpener on someone else's desk".*

Engineering research is trying nowadays to improve the input/output subsystem to reduce the impact of the I/O-bottleneck onto the efficiency of applications managing large datasets; but, on the other hand, we are aware that the improvements achievable by means of a good algorithm design abundantly surpass the best expected technology advancements. Let us see the why with a simple example![1]

Assume to take three algorithms having increasing I/O-complexity: $C_1(n) = n$, $C_2(n) = n^2$ and $C_3(n) = 2^n$. Here $C_i(n)$ denotes the number of disk accesses executed by the ith algorithm to process n input data (stored e.g. in n/B disk pages). Notice that the first two algorithms execute a *polynomial* number of I/Os (in the input size), whereas the last one executes an *exponential* number of I/Os. Moreover we note that the above complexities have a very simple (and thus unnatural) mathematical form because we want to simplify the calculations without impairing our final conclusions. Let us

[1] This is paraphrased from [8], now we talk about I/Os instead of steps.

now ask for how many data each of these algorithms is able to process in a fixed time-interval of size t, given that each I/O takes c time. The answer is obtained by solving the equation $C_i(n) \times c = t$ with respect to n: so we get t/c data are processed by the first algorithm in t time, $\sqrt{t/c}$ data are processed by the second algorithm, and only $\log_2(t/c)$ data are processed by the third algorithm. These values are already impressive by themselves, and provide a robust understanding of why polynomial-time algorithms are called *efficient*, whereas exponential-time algorithms are called *inefficient*: a large change in the length t of the time-interval induces just a tiny change in the amount of data that exponential-time algorithms can process. Of course, this distinction admits many exceptions when the problem instances have limited size or have distributions which favor efficient executions (think e.g. to the simplex algorithm). But, on the other hand, these examples are quite rare, and the much more stringent bounds on execution time satisfied by polynomial-time algorithms make them considered *provably* efficient and the preferred way to solve problems. Algorithmically speaking, most exponential-time algorithms are merely implementations of the approach based on *exhaustive* search, whereas polynomial-time algorithms are generally made possible only through the gain of some deeper insight into the structure of a problem. So polynomial-time algorithms are the *right* choice from many points of view.

Let us now assume to run the above algorithms with a better I/O-subsystem, say one that is k times faster, and ask: How many data can be managed by this new PC? To address this question we solve the previous equations with the time-interval set to the length $k \times t$, thus implicitly assuming to run the algorithms over the old PC but providing itself with k times more time. We get that the first algorithm perfectly scales by a factor k, the second algorithm scales by a factor $\sqrt{k}$, whereas the last algorithm scales *only of an additional term* $\log_2 k$. Noticeably the improvement induced by a k-times more powerful PC for an exponential-time algorithm is totally negligible even in the presence of impressive (and thus unnatural) technology advancements! Super-linear algorithms, like the second one, are positively affected by technology advancements but their performance improvement decreases as the degree of the polynomial-time complexity grows: more precisely, if $C(n) = n^\alpha$ then a k-times more powerful PC induces a speed-up of a factor $\sqrt[\alpha]{k}$. Overall, it is not hazardous to state that the impact of a good algorithm is far beyond any optimistic forecasting for the performance of future (mechanical or SSD) disks.[2]

Given this *appetizer* about the "Power of Algorithms", let us now turn back to the problem of analyzing the performance of algorithms in modern PCs by considering the following simple example: Compute the sum of the integers stored in an array $A[1, n]$. The simplest idea is to scan A and accumulate in a temporary variable the sum of the scanned integers. This algorithm executes n sums, accesses each integer in A once, and thus takes n steps. Let us now generalize this approach by considering a family of algorithms, denoted by $\mathcal{A}_{s,b}$, which differentiate themselves according to the pattern of accesses to A's elements, as driven by the parameters s and b. In particular $\mathcal{A}_{s,b}$ looks at array A as logically divided into blocks of b elements each, say $A_j = A[j * b + 1, (j+1) * b]$ for $j = 0, 1, 2, \ldots, n/b - 1$.[3] Then it sums all items in one block before moving to the next block that is s blocks apart on the right. Array A is considered cyclic so that, when the next block lies out of A, the algorithm wraps around it starting again from its beginning. Clearly not all values of s allow to take into account all of A's blocks (and thus sum all of A's integers). Nevertheless we know that if s is co-prime with n/b then $[s \times i \mod (n/b)]$ generates a permutation of the integers $\{0, 1, \ldots, n/b - 1\}$, and thus $\mathcal{A}_{s,b}$ touches all blocks in A and hence sums all of its integers. But the specialty of this parametrization is that by varying s and b we can sum according to different patterns of memory accesses: from the sequential scan indicated above (setting $s = b = 1$), to a block-wise

[2] See [11] for an extended treatment of this subject.

[3] For the sake of presentation we assume that n and b are powers of two, so b divides n.

access (set a large b) and/or a random-wise access (set a large s). Of course, all algorithms $\mathcal{A}_{s,b}$ are equivalent from a computational point of view, since they read exactly n integers and thus take n steps; but from a practical point of view, they have different time performance which becomes more and more significant as the array size n grows. The reason is that, for a growing n, data will be spread over more and more memory levels, each one with its own capacity, latency, bandwidth and access method. So the "equivalence in efficiency" derived by adopting the RAM model, and counting the number-of-steps executed by $\mathcal{A}_{s,b}$, is not an accurate estimate of the real time required by that algorithms to sum A's elements.

We need a different model that grasps the essence of real computers and is simple enough to not jeopardize algorithm design and analysis. In a previous example we already argued that the number of I/Os is a good estimator for the time complexity of an algorithm, given the large gap existing between disk- and internal-memory accesses. This is indeed what is captured by the so called *2-level memory model* (aka. disk-model, or external-memory model [11]) which abstracts the computer as composed by only *two memory levels*: the internal memory of size M, and the (unbounded) disk memory which operates by reading/writing data via blocks of size B (called *disk pages*). Sometimes the model consists of D disks, each of unbounded size, so that each I/O reads or writes a total of $D \times B$ items coming from D pages, each one residing on a different disk. For the sake of clarity we remark that the *two-level view* must not suggest to the reader that this model is restricted to abstracts disk-based computations; in fact, we are actually free to choose any two levels of the memory hierarchy, with their M and B parameters properly set. The algorithm performance is evaluated in this model by counting: (a) the number of accesses to disk pages (hereafter *I/O*s), (b) the internal running time (CPU time), and (c) the number of disk pages used by the algorithm as its working space. This suggests *two golden rules* for the design of "good" algorithms operating on large datasets: they must exploit *spatial locality* and *temporal locality*. The former imposes a data organization onto the disk(s) that makes each accessed disk-page as much useful as possible; the latter imposes to execute as much useful work as possible onto data fetched in internal memory, before they are written back to disk.

In the light of this new model, let us re-analyze the time complexity of algorithms $\mathcal{A}_{s,b}$ by taking into account I/Os, given that the CPU time is still n and the space occupancy is n/B pages. We start from the simplest settings for s and b in order to gain some intuitions about the general formulas. The case $s = 1$ is obvious, algorithms $\mathcal{A}_{1,b}$ scan A rightward by taking n/B I/Os, independently of the value of b. As s and b change the situation complicates, but by not much. Fix $s = 2$ and pick some $b < B$ that, for simplicity, is assumed to divide the block-size B. Every block of size B consists of B/b smaller (logical) blocks of size b, and the algorithm $\mathcal{A}_{2,b}$ examines only half of them because of the jump $s = 2$. This actually means that each B-sized page is half utilized in the summing process, thus inducing a total of $2n/B$ I/Os. It is then not difficult to generalize this formula by writing a cost of $\min\{s, B/b\} \times (n/B)$ I/Os, which correctly gives n/b for the case of large jumps over array A. This formula provides a better approximation of the real time complexity of the algorithm, although it does not capture all features of the disk. In fact, it considers all I/Os as *equal*, independently of their distribution. This is clearly unprecise because on real disks the *sequential* I/Os are faster than the *random* I/Os.[4] Referring to the previous example, the algorithms $\mathcal{A}_{s,B}$ have still I/O-complexity n/B, independently of s, although their behavior is rather different if executed on a (mechanical) disk because of the disk seeks induced by larger and larger s. As a result, we can conclude that even the 2-level memory model is an approximation of the behavior of

[4] Conversely, this difference will be almost negligible in an (electronic) memory, such as the DRAM or the modern Solid-State disks, where the distribution of the memory accesses does not significantly impact onto the throughput of the memory/SSD.

algorithms on real computers, although it results sufficiently good that it has been widely adopted in the literature to evaluate their performance on massive datasets. So that, in order to be as much precise as possible, we will evaluate in these notes our algorithms by specifying not only the number of executed I/Os but also characterizing their *distribution* (random vs contiguous) over the disk.

At this point one could object that given the impressive technological advancements of the last years, the internal-memory size M is so large that most of the working set of an algorithm (roughly speaking, the set of pages it will reference in the near future) can be fit into it, thus reducing significantly the case of an I/O-fault. We will argue that an even small portion of data resident to disk makes the algorithm slower than expected, and so, data organization cannot be neglected even in these extremely favorable situations.

Let us see why, by means of a "back of the envelope" calculation! Assume that the input size $n = (1 + \epsilon)M$ is larger than the internal-memory size of a factor $\epsilon > 0$. The question is how much ϵ impacts onto the average cost of an algorithm step, given that it may access a datum located either in internal memory or on disk. To simplify our analysis, without renouncing to a meaningful conclusion, we assume that $p(\epsilon)$ is the probability of an I/O-fault. As an example, if $p(\epsilon) = 1$ then the algorithm always accesses its data on disk (i.e. one of the ϵM items); if $p(\epsilon) = \frac{\epsilon}{1+\epsilon}$ then the algorithm has a fully-random behavior in accessing its input data (since, from above, we can rewrite $\frac{\epsilon}{1+\epsilon} = \frac{\epsilon M}{(1+\epsilon)M} = \frac{\epsilon M}{n}$); finally, if $p(\epsilon) = 0$ then the algorithm has a working set smaller than the internal memory size, and thus it does not execute any I/Os. Overall $p(\epsilon)$ measures the *un-locality* of the memory references of the analyzed algorithm.

To complete the notation, let us indicate with c the time needed for one I/O wrt one internal-memory access— we have $c \approx 10^5 - 10^6$, see above— and we set a to be the fraction of steps that induce a memory access in the running algorithm (this is typically 30% – 40%, according to [6]). Now we are ready to estimate the *average cost of the step* for an algorithm working in this scenario:

$$1 \times \mathcal{P}(\text{computation step}) + t_m \times \mathcal{P}(\text{memory-access step}),$$

where t_m is the average cost of a memory access. To compute t_m we have to distinguish two cases: an in-memory access (occurring with probability $1 - p(\epsilon)$) or a disk access (occurring with probability $p(\epsilon)$). So we have $t_m = 1 \times (1 - p(\epsilon)) + c \times p(\epsilon)$. Observing that $\mathcal{P}(\text{memory-access step}) + \mathcal{P}(\text{computation step}) = 1$, and plugging the fraction a of memory accesses into $\mathcal{P}(\text{memory-access step})$, we derive the final formula:

$$(1 - a) \times 1 + a \times [1 \times (1 - p(\epsilon)) + c \times p(\epsilon)] = 1 + a \times (c - 1) \times p(\epsilon) \geq 3 \times 10^4 \times p(\epsilon).$$

This formula clearly shows that, even for algorithms exploiting locality of references (i.e. a small $p(\epsilon)$), the slowdown may be significant and actually it turns out to be four order of magnitudes larger than what might be expected (i.e. $p(\epsilon)$). Just as an example, take an algorithm that exploits locality of references in its memory accesses, say 1 out of 1000 memory accesses is on disk (i.e. $p(\epsilon) = 0.001$). Then, its performance on a massive dataset that is stored on disk would be slowed down by a factor > 30 with respect to a computation executed completely in internal memory.

It goes without saying that this is just the tip of the iceberg, because the larger is the amount of data to be processed by an algorithm, the higher is the number of memory levels involved in the storage of these data and, hence, the more variegate are the types of "memory faults" (say cache-faults, memory-faults, etc.) to cope with for achieving efficiency. The overall message is that neglecting questions pertaining to the cost of memory references in a hierarchical-memory system may *prevent* the use of an algorithm on large input data.

Motivated by these premises, these notes will provide few examples of challenging problems which admit elegant algorithmic solutions whose efficiency is crucial to manage the large datasets that occur in many real-world applications. Algorithm design will be accompanied by several comments on the difficulties that underlie the *engineering* of those algorithms: how to turn a “theoretically efficient” algorithm into a “practically efficient” code. Too many times, as a theoretician, I got the observation that “your algorithm is far from being amenable to an efficient implementation!”. By following the recent surge of investigations in *Algorithm Engineering* [10] (to be not confused with the “practice of Algorithms”), we will also dig into the deep computational features of some algorithms by resorting few other successful models of computations— mainly the streaming model [9] and the cache-oblivious model [5]. These models will allow us to capture and highlight some interesting issues of the underlying computation: such as disk passes (streaming model), and universal scalability (cache-oblivious model). We will try our best to describe all these issues in their simplest terms but, nonetheless to say, we will be unsuccessful in turning this “rocket science for non-boffins” into a “science for dummies” [2]. In fact lots of many more things have to fall into place for algorithms to work: top-IT companies (like Google, Yahoo, Microsoft, IBM, Oracle, Facebook, Twitter, etc.) are perfectly aware of the difficulty to find people with the right skills for developing and refining “good” algorithms. This booklet will scratch just the surface of Algorithm Design and Engineering, with the main goal of spurring inspiration into your daily job as software designer or engineer.

References

[1] Person of the Year. *Time Magazine*, 168(27–28), December 2006.

[2] Business by numbers. *The Economist*, September 2007.

[3] Wikipedia’s entry: “Algorithm characterizations”, 2009. At `http://en.wikipedia.org/wiki/Algorithm_characterizations`

[4] Declan Butler. *2020 computing: Everything, everywhere*, volume 440, chapter 3, pages 402–405. Nature Publishing Group, March 2006.

[5] Rolf Fagerberg. Cache-oblivious model. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann, September 2006.

[7] Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, 1973.

[8] Fabrizio Luccio. *La struttura degli algoritmi*. Boringhieri, 1982.

[9] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[10] Peter Sanders. Algorithm engineering - an attempt at a definition. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2009.

[11] Jeffrey S. Vitter. External memory algorithms and data structures. *ACM Computing Surveys*, 33(2):209–271, 2001.

2

A warm-up!

"Everything should be made as simple as possible, but not simpler."
Albert Einstein

Let us consider the following problem, surprisingly simple in its statement but not that much for what concerns the design of its optimal solution.

> **Problem.** *We are given the performance of a stock at* NYSE *expressed as a sequence of* day-by-day differences *of its quotations. We wish to determine the best buy-&-sell strategy for that stock, namely the pair of days $\langle b, s\rangle$ that would have maximized our revenues if we would have bought the stock at (the beginning of) day b and sold it at (the end of) day s.*

The specialty of this problem is that it has a simple formulation, which finds many other useful variations and applications. We will comment on them at the end of this lecture, now we content ourselves by mentioning that we are interested in this problem because it admits a sequence of algorithmic solutions of increasing sophistication and elegance, which imply a significant reduction in their time complexity. The ultimate result will be a linear-time algorithm, i.e. linear in the number n of stock quotations. This algorithm is *optimal* in terms of the number of executed steps, because all day-by-day differences must be looked at in order to determine if they must be included or not in the optimal solution, actually, one single difference could provide a one-day period worth of investment! Surprisingly, the optimal algorithm will exhibit the simplest pattern of memory accesses— it will execute a single scan of the available stock quotations— and thus it will offer a *streaming behavior*, particularly useful in a scenario in which the granularity of the buy-&-sell actions is not restricted to full-days and we must possibly compute the optimal time-window *on-the-fly* as quotations oscillate. More than this, as we commented in the previous lecture, this algorithmic scheme is optimal in terms of I/Os and *uniformly* over all levels of the memory hierarchy. In fact, because of its streaming behavior, it will execute n/B I/Os independently of the disk-page size B, which may be thus unknown to the underlying algorithm. This is the typical feature of the so called *cache-oblivious algorithms* [4], which we will therefore introduce at the right point of this lecture.

This lecture will be the prototype of what you will find in the next pages: a simple problem to state, with few elegant solutions and challenging techniques to teach and learn, together with several intriguing extensions that can be posed as exercises to the students or as puzzles to tempt your mathematical skills!

Let us now dig into the technicalities, and consider the following example. Take the case of 11 days of exchange for a given stock, and assume that $D[1, 11] = [+4, -6, +3, +1, +3, -2, +3, -4, +1,$

−9, +6] denotes the day-by-day differences of quotations of that stock. It is not difficult to convince yourself that the gain of buying the stock at day x and selling it at day y is equal to the sum of the values in the sub-array $D[x, y]$, namely the sum of all its fluctuations. As an example, take $x = 1$ and $y = 2$, the gain is $+4 - 6 = -2$, and indeed we would loose 2 dollars in buying the morning of the first day and selling the stock at the end of the second day. Notice that the starting value of the stock is not crucial for determining the best time-interval of our investment, what is important are its variations. In other words, the problem stated above boils down to determine the sub-array of $D[1, n]$ which maximizes the sum of its elements. In the literature this problem is indeed known as the *maximum sub-array sum* problem.

Problem Abstraction. *Given an array* $D[1, n]$ *of positive and negative numbers, we want to find the sub-array* $D[b, s]$ *which maximizes the sum of its elements.*

It is clear that if all numbers are positive, then the optimal sub-array is the entire D: this is the case of an always increasing stock price, and indeed there is no reason to sell it before the last day! Conversely, if all numbers are negative, then we can pick the one-element window containing the largest (negative) value: if you are imposed to buy this poor stock, then do it in the day it looses the smallest value and sell it soon! In all other cases, it is not at all clear where the optimum sub-array is located. In the example, the optimum spans $D[3, 7] = [+3, +1, +3, -2, +3]$ and has gain +8 dollars. This shows that the optimum neither includes the best exploit of the stock (i.e. +6) nor it consists of positive values only. The *structure* of the optimum sub-array is not simple but, surprisingly enough, not very complicated as we will show in Section 2.3.

2.1 A cubic-time algorithm

We start by considering an inefficient solution which translates in pseudo-code the formulation of the problem given above. This algorithm is detailed in Figure 2.1, where the pair of variables $<b_o, s_o>$ identifies the current sub-array of maximum sum, whose value is stored in `MaxSum`. Initially `MaxSum` is set to the dummy value $-\infty$, so that it is immediately changed whenever the algorithm executes Step 8 for the first time. The core of the algorithm examines all possible sub-arrays $D[b, s]$ (Steps 2-3) computing for each of them the sum of their elements (Steps 4-7). If a sum larger than the current maximal value is found (Steps 8-9), then `TmpSum` and its corresponding sub-array are stored in `MaxSum` and $<b_o, s_o>$, respectively.

Algorithm 2.1 The cubic-time algorithm

```
1:  $MaxSum = -\infty$
2:  for ($b = 1$; $b \leq n$; $b$++) do
3:      for ($s = b$; $s \leq n$; $s$++) do
4:          $TmpSum = 0$
5:          for ($i = b$; $i \leq s$; $i$++) do
6:              $TmpSum+ = D[i]$;
7:          end for
8:          if ($MaxSum < TmpSum$) then
9:              $MaxSum = TmpSum$; $b_o = b$; $s_o = s$;
10:         end if
11:     end for
12: end for
13: return $\langle MaxSum, b_o, s_o \rangle$;
```

The correctness of the algorithm is immediate, since it checks all possible sub-arrays of $D[1, n]$ and selects the one whose sum of its elements is the largest (Step 8). The time complexity is cubic, i.e. $\Theta(n^3)$, and can be evaluated as follows. Clearly the time complexity is upper bounded by $O(n^3)$ because we can form no more than $\frac{n^2}{2}$ pairs `<b,s>` out of n elements,[1] and n is an upper-bound to the cost of computing the sum of each sub-array. Let us now show that the time cost is also $\Omega(n^3)$, so concluding that the time complexity is strictly cubic. To show this lower bound, we observe that $D[1, n]$ contains $(n - L + 1)$ sub-arrays of length L, and thus the cost of computing the sum for all of their elements is $(n - L + 1) \times L$. Summing over all values of L, we would get the exact time complexity. But here we are interested in a lower bound, so we can evaluate that cost just for the subset of sub-arrays whose length L is in the range $[n/4, n/2]$. For each such L, we have that $n - L + 1 > n/2$ and $L \geq n/4$, so the cost above is $(n - L + 1) \times L > n^2/8$. Since we have $\frac{n}{2} - \frac{n}{4} + 1 > n/4$ of those Ls, the total cost for analysing that subset of sub-arrays is lower bounded by $n^3/32 = \Omega(n^3)$.

It is natural now to ask ourselves how much fast in practice is the designed algorithm. We implemented it in Java and tested on a commodity PC. As n grows, its time performance reflects in practice its cubic time complexity, evaluated in the RAM model. More precisely, it takes about 20 seconds to solve the problem for $n = 10^3$ elements, and about 30 hours for $n = 10^5$ elements. Too much indeed if we wish to scale to very large sequences (of quotations), as we are aiming for in these lectures.

2.2 A quadratic-time algorithm

The key inefficiency of the cubic-time algorithm resides in the execution of Steps 4-7 of the pseudo-code in Figure 2.1. These steps re-compute from scratch the sum of the sub-array $D[b, s]$ each time its extremes change in Steps 2-3. But if we look carefully at the **for**-cycle of Step 3 we notice that the size s is incremented by one unit at a time from the value b (one element sub-array) to the value n (the longest possible sub-array that starts at b). Therefore, from one execution to the next one of Step 3, the sub-array to be summed changes from $D[b, s]$ to $D[b, s + 1]$. It is thus immediate to conclude that the new sum for $D[b, s + 1]$ does not need to be recomputed from scratch, but can be computed *incrementally* by just adding the value of the new element $D[s + 1]$ to the current value of `TmpSum` (which inductively stores the sum of $D[b, s]$). This is exactly what the pseudo-code of Figure 2.2 implements: its two main changes with respect to the cubic algorithm of Figure 2.1 are in Step 3, that nulls `TmpSum` every time b is changed (because the sub-array starts again from length 1, namely $D[b, b]$), and in Step 5, that implements the incremental update of the current sum as commented above. Such small changes are worth of a saving of $\Theta(n)$ additions per execution of Step 2, thus turning the new algorithm to have quadratic-time complexity, namely $\Theta(n^2)$.

More precisely, let us concentrate on counting the number of additions executed by the algorithm of Figure 2.2; this is the prominent operation of this algorithm so that its evaluation will give us an estimate of its total number of steps. This number is[2]

$$\sum_{b=1}^{n}(1 + \sum_{s=b}^{n} 1) = \sum_{b=1}^{n}(1 + (n - b + 1)) = n \times (n + 2) - \sum_{b=1}^{n} b = n^2 + 2n - \frac{n(n-1)}{2} = O(n^2).$$

This improvement is effective also in practice. Take the same experimental scenario of the previous section, this new algorithm requires less than 1 second to solve the problem for $n = 10^3$

[1] For each pair $< b, s >$, with $b \leq s$, $D[b, s]$ is a possible sub-array, but $D[s, b]$ is not.

[2] We use below the famous formula, discovered by the young Gauss, to compute the sum of the first n integers.

Algorithm 2.2 The quadratic-time algorithm

```
$MaxSum = -\infty$;
for ($b = 1$; $b \leq n$; $b{+}{+}$) do
    $TmpSum = 0$;
    for ($s = b$; $s \leq n$; $s{+}{+}$) do
        $TmpSum \mathrel{+}= D[s]$;
        if ($MaxSum < TmpSum$) then
            $MaxSum = TmpSum$; $b_o = b$; $s_o = s$;
        end if
    end for
end for
return $\langle MaxSum, b_o, s_o \rangle$;
```

elements, and about 28 minutes to manage 10^6 elements. This means that the new algorithm is able to manage more elements in "reasonable" time. Clearly, these timings and these numbers could change if we use a different programming language (Java, in the present example), operating system (Windows, in our example), and processor (the old Pentium IV, in our example). Nevertheless we believe that they are interesting anyway because they provide a concrete picture of what it does mean a theoretical improvement like the one we showed in the above paragraphs on a real situation. It goes without saying that the life of a coder is typically not so easy because theoretically-good algorithms many times hide so many details that their engineering is difficult and big-O notation often turn out to be not much "realistic". Do not worry, we will have time in these lectures to look at these issues in more detail.

2.3 A linear-time algorithm

The final step of this lecture is to show that the maximum sub-array sum problem admits an elegant algorithm that processes the elements of $D[1, n]$ is a streaming fashion and takes the *optimal* $O(n)$ time. We could not aim for more!

To design this algorithm we need to dig into the structural properties of the optimal sub-array. For the purpose of clarity, we refer the reader to Figure 2.1 below, where the optimal sub-array is assumed to be located at two positions $b_o \leq s_o$ in the range $[1, n]$.

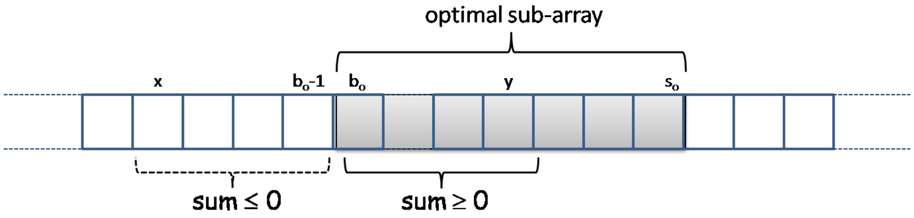


FIGURE 2.1: An illustrative example of Properties 1 and 2.

Let us now take a sub-array that starts before b_o and ends at position $b_o - 1$, say $D[x, b_o - 1]$. The sum of the elements in this sub-array cannot be positive because, otherwise, we could merge it with the (adjacent) optimal sub-array and thus get the longer sub-array $D[x, s_o]$ having sum *larger than* the one obtained with the (claimed) optimal $D[b_o, s_o]$. So we can state the following:

Property 1. The sum of the elements in a sub-array $D[x, b_o - 1]$, with $x < b_o$, cannot be (strictly) positive.

Via a similar argument, we can consider a sub-array that is a prefix of the optimal $D[b_o, s_o]$, say $D[b_o, y]$ with $y \leq s_o$. This sub-array cannot have negative sum because, otherwise, we could drop it from the optimal solution and get a shorter array, namely $D[y + 1, s_o]$ having sum larger than the one obtained by the (claimed) optimal $D[b_o, s_o]$. So we can state the following other property:

Property 2. The sum of the elements in a sub-array $D[b_o, y]$, with $y \leq s_o$, cannot be (strictly) negative.

We remark that any one of the sub-arrays considered in the above two properties might have sum equal to zero. This would not affect the optimality of $D[b_o, s_o]$, it could only introduce other optimal solutions being either longer or shorter than $D[b_o, s_o]$.

Let us illustrate these two properties on the array $D[1, 11] = [+4, -6, +3, +1, +3, -2, +3, -4, +1, -9, +6]$. Here the optimum sub-array is $D[3, 7] = [+3, +1, +3, -2, +3]$. We note that $D[x, 2]$ is always negative (Prop. 1), in fact for $x = 1$ the sum is $+4 - 6 = -2$ and for $x = 2$ the sum is -6. On the other hand the sum of all elements in $D[3, y]$ is positive for all prefixes of the optimum sub-array (Prop. 2), namely $y \leq 7$. We also point out that the sum of $D[3, y]$ is positive even for some $y > 7$, take for example $D[3, 8]$ for which the sum is 4 and $D[3, 9]$ for which the sum is 5. Of course, this does not contradict Prop. 2.

Algorithm 2.3 The linear-time algorithm

```
 1: $MaxSum = -\infty$
 2: $TmpSum = 0$; $b = 1$;
 3: for ($s = 1$; $s \leq n$; $s$++) do
 4:     $TmpSum$ += $D[s]$;
 5:     if ($MaxSum < TmpSum$) then
 6:         $MaxSum = TmpSum$; $b_o = b$; $s_o = s$;
 7:     end if
 8:     if ($TmpSum < 0$) then
 9:         $TmpSum = 0$; $b = s + 1$;
10:     end if
11: end for
12: return $\langle MaxSum, b_o, s_o \rangle$;
```

The two properties above lead to the simple Algorithm 2.3. It consists of one unique **for**-cycle (Step 3) which keeps in `TmpSum` the sum of a sub-array ending in the currently examined position s and starting at some position $b \leq s$. At any step of the **for**-cycle, the candidate sub-array is extended one position to the right (i.e. s++), and its sum `TmpSum` is increased by the value of the current element $D[s]$ (Step 4). Since the current sub-array is a candidate to be the optimal one, its sum is compared with the current optimal value (Step 5). Then, according to Prop. 1, if the sub-array sum is negative, the current sub-array is discarded and the process "restarts" with a new sub-array beginning at the next position $s + 1$ (Steps 8-9). Otherwise, the current sub-array is extended to the right, by incrementing s. The tricky issue here is to show that the optimal sub-array is checked in Step 5, and thus stored in $< b_o, s_o >$. This is not intuitive at all because the algorithm is checking n sub-arrays out of the $\Theta(n^2)$ possible ones, and we want to show that this (minimal) subset of candidates actually contains the optimal solution. This subset is *minimal* because these sub-arrays form a *partition* of $D[1,n]$ so that every element belongs to one, and only one checked sub-array. Moreover, since every element must be analyzed, we cannot discard any sub-array of this partition without checking its sum!

Before digging into the formal proof of correctness, let us follow the execution of the algorithm over the array $D[1,11] = [+4,-6,+3,+1,+3,-2,+3,-4,+1,-9,+6]$. Remember that the optimum sub-array is $D[3,7] = [+3,+1,+3,-2,+3]$. We note that $D[x,2]$ is negative for $x = 1,2$, so the algorithm surely zeroes the variable TmpSum when $s = 2$ in Steps 8-9. At that time, b is set to 3 and TmpSum is set to 0. The subsequent scanning of the elements $s = 3,\ldots,7$ will add their values to TmpSum which is always positive (see above). When $s = 7$, the examined sub-array coincides with the optimal one, we thus have TmpSum = 8, so Step 5 stores the optimum location in $< b_o, s_o >$. It is interesting to notice that, in this example, the algorithm does not re-start the value of TmpSum at the next position $s = 8$ because it is still positive (namely, TmpSum = 4); this means that the algorithm will examine sub-arrays longer than the optimal one, but all having a smaller sum, of course. The next re-starting will occur at position $s = 10$ where TmpSum = -4.

It is easy to realize that the time complexity of the algorithm is $O(n)$ because every element is examined just once. More tricky is to show that the algorithm is correct, which actually means that Steps 4 and 5 eventually compute and check the optimal sub-array sum. To show this, it suffices to prove the following two facts: (i) when $s = b_o - 1$, Step 8 resets b to b_o; (ii) for all subsequent positions $s = b_o,\ldots,s_o$, Step 8 never resets b so that it will eventually compute in TmpSum the sum of all elements in $D[b_o,s_o]$, whenever $s = s_o$. It is not difficult to see that Fact (i) derives from Property 1, and Fact (ii) from Property 2.

This algorithm is very fast in the same experimental scenario mentioned in the previous sections, it takes less than 1 second to process millions of quotations. A truly scalable algorithm, indeed, with many nice features that make it appealing also in a hierarchical-memory setting. In fact, this algorithm scans the array D from left to right and examines each of its elements just once. If D is stored on disk, these elements are fetched in internal memory one page at a time. Hence the algorithm executes n/B I/Os, which is *optimal*. It is interesting also to note that the design of the algorithm does not depend on B (which indeed does not appear in the pseudo-code), but we can evaluate its I/O-complexity in terms of B. Hence the algorithm takes n/B optimal I/Os independently of the the page size B, and thus subtly on the hierarchical-memory levels interested by the algorithm execution. Decoupling the use of the parameter B between algorithm design and algorithm analysis is the key issue of the so called *cache-oblivious algorithms*, which are a hot topic of algorithmic investigation nowadays. This feature is achieved here in a basic (trivial) way by just adopting a scan-based approach. The literature [4] offers more sophisticated results regarding the design of cache-oblivious algorithms and data structures.

2.4 Another linear-time algorithm

There exists another optimal solution to the maximum sub-array sum problem which hinges on a different algorithm design. For simplicity of exposition, let us denote by $Sum_D[y', y'']$ the sum of the elements in the sub-array $D[y', y'']$. Take now a selling time s and consider all sub-arrays that end at s: namely we are interested in sub-arrays having the form $D[x,s]$, with $x \leq s$. The value $Sum_D[x,s]$ can be expressed as the difference between $Sum_D[1,s]$ and $Sum_D[1,x-1]$. Both of these sums are indeed *prefix*-sums over the array D and can be computed in linear time. As a result, we can rephrase our maximization problem as follows:

$$\max_s \max_{b\leq s} \mathrm{Sum}_D[b,s] = \max_s \max_{b\leq s} (\mathrm{Sum}_D[1,s] - \mathrm{Sum}_D[1,b-1]).$$

We notice that if $b = 1$ the second term refers to the empty sub-array $D[1, 0]$; so we can assume that $Sum_D[1, 0] = 0$. This is the case in which $D[1, s]$ is the sub-array of maximum sum among all the sub-arrays ending at s (so no prefix sub-array $D[1, b-1]$ is dropped from it).

The next step is to pre-compute all prefix sums $P[i] = Sum_D[1, i]$ in $O(n)$ time and $O(n)$ space via a scan of the array D: Just notice that $P[i] = P[i-1] + D[i]$, where we set $P[0] = 0$ in order to manage the special case above. Hence we can rewrite the maximization problem in terms of the array P, rather than Sum_D: $\max_{b \leq s}(P[s] - P[b-1])$. The cute observation now is that we can decompose the max-computation into a max-min calculation over the two variables b and s

$$\max_s \max_{b \leq s}(P[s] - P[b-1]) = \max_s (P[s] - \min_{b \leq s} P[b-1]).$$

The key idea is that we can move $P[s]$ outside the inner max-calculation because it does not depend on the variable b, and then change a max into a min because of the negative sign. The final step is then to pre-compute the minimum $\min_{b \leq s} P[b-1]$ for all positions s, and store it in an array $M[0, n-1]$. We notice that, also in this case, the computation of $M[i]$ can be performed via a single scan of P in $O(n)$ time and space: set $M[0] = 0$ and then derive $M[i]$ as $\min\{M[i-1], P[i]\}$. Given M, we can rewrite the formula above as $\max_s(P[s] - M[s-1])$ which can be clearly computed in $O(n)$ time given the two arrays P and M. Overall this new approach takes $O(n)$ time and $O(n)$ extra space.

As an illustrative example, consider again the array $D[1, 11] = [+4, -6, +3, +1, +3, -2, +3, -4, +1, -9, +6]$. We have that $P[0, 11] = [0, +4, -2, +1, +2, +5, +3, +6, +2, +3, -6, 0]$ and $M[0, 10] = [0, 0, -2, -2, -2, -2, -2, -2, -2, -2, -6]$. If we compute the difference $P[s] - M[s-1]$ for all $s = 1, \ldots, n$, we obtain the sequence of values $[+4, -2, +3, +4, +7, +5, +8, +4, +5, -4, +6]$, whose maximum (sum) is $+8$ that occurs (correctly) at the (final) position $s = 7$. It is interesting to note that the left-extreme b_o of the optimal sub-array could be derived by finding the position $b_o - 1$ where $P[b_o - 1]$ is minimum: in the example, $P[2] = -2$ and thus $b_o = 3$.

Algorithm 2.4 Another linear-time algorithm

```
$MaxSum = -\infty$; $b_o = 1$;
$TmpSum = 0$; $MinTmpSum = 0$;
for ($s = 1$; $s \leq n$; $s$++) do
    $TmpSum$ += $D[s]$;
    if ($MaxSum < TmpSum - MinTmpSum$) then
        $MaxSum = TmpSum - MinTmpSum$; $s_o = s$; $b_o = b_{tmp}$;
    end if
    if ($TmpSum < MinTmpSum$) then
        $MinTmpSum = TmpSum$; $b_{tmp} = s + 1$;
    end if
end for
return $\langle MaxSum, b_o, s_o \rangle$;
```

We conclude this section by noticing that the proposed algorithm executes three passes over the array D, rather than the single pass of Algorithm 2.3. It is not difficult to turn this algorithm to make *one-pass* too. It suffices to deploy the associativity of the min/max functions, and use two variables that inductively keep the values of $P[s]$ and $M[s-1]$ as the array D is scanned from left to right. Algorithm 2.4 implements this idea by using the variable `TmpSum` to store $P[s]$ and the variable `MinTmpSum` to store $M[s-1]$. This way the formula $\max_s(P[s] - M[s-1])$ is evaluated incrementally for $s = 1, \ldots, n$, thus avoiding the two passes for pre-calculating the arrays P and

M and the extra-space needed to store them. One pass over D is then enough, and so we have re-established the nice algorithmic properties of Algorithm 2.3 but with a completely different design!

2.5 Few interesting variants$^{\infty}$

As we promised at the beginning of this lecture, we discuss now few interesting variants of the maximum sub-array sum problem. For further algorithmic details and formulations we refer the interested reader to [1, 2]. Note that this is a challenging section, because it proposes an algorithm whose design and analysis are sophisticated!

Sometimes in the bio-informatics literature the term "sub-array" is substituted by "segment", and the problem takes the name of "maximum-sum segment problem". In the bio-context the goal is to identify segments which occur inside DNA sequences (i.e. strings of four letters A, T, G, C) and are *rich* of G or C nucleotides. Biologists believe that these segments are biologically significant since they predominantly contain genes. The mapping from DNA sequences to *arrays of numbers*, and thus to our problem abstraction, can be obtained in several ways depending on the objective function that models the *GC-richness* of a segment. Two interesting mappings are the following ones:

- Assign a penalty $-p$ to the nucleotides A and T of the sequence, and a reward $1-p$ to the nucleotides C and G. Given this assignment, the sum of a segment of length l containing x occurrences of C+G is equal to $x - p \times l$. Biologists think that this function is a good measure for the CG-richness of that segment. Interestingly enough, all algorithms described in the previous sections can be used to identify the CG-rich segments of a DNA sequence in linear time, according to this objective function. Often, however, biologists prefer to define a cutoff range on the length of the segments for which the maximum sum must be searched, in order to avoid the reporting of extremely short or extremely long segments. In this new scenario the algorithms of the previous sections cannot be applied, but yet linear-time optimal solutions are known for them (see e.g. [2]).
- Assign a value 0 to the nucleotides A and T of the sequence, and a value 1 to the nucleotides C and G. Given this assignment, the density of C+G nucleotides in a segment of length l containing x occurrences of C and G is x/l. Clearly $0 \leq x/l \leq 1$ and every single occurrence of a nucleotide C or G provides a segment with maximum density 1. Biologists consider this as an interesting measure of CG-richness for a segment, provided that a cutoff range on the length of the searched segments is imposed. This problem is more difficult than the one stated in the previous item, nevertheless it posses optimal (quasi-)linear time solutions which are much sophisticated and for which we refer the interested reader to the pertinent bibliography (e.g. [1, 3, 5]).

These examples are useful to highlight a *dangerous trap* that often occurs when abstracting a real problem: apparently small changes in the problem formulation lead to big jumps in the complexity of designing efficient algorithms for them. Think for example to the density function above, we needed to introduce a cutoff lower-bound to the segment length in order to avoid the trivial solution consisting of *single* nucleotides C or G! With this "small" change, the problem results more challenging and its solutions sophisticated.

Other subtle traps are more difficult to be discovered. Assume that we decide to circumvent the single-nucleotide outcome by searching for the the *longest* segment whose density is *larger than* a fixed value d. This is, in some sense, a complementary formulation of the problem stated in the second item above, because maximization is here on the segment length and a (lower) cut-off is imposed on the density value. Surprisingly it is possible to *reduce* this density-based problem to a sum-based problem, in the spirit of the one stated in the first item above, and solved in the previous sections. Algorithmic reductions are often employed by researchers to re-use known solutions and thus do not re-discover again and again the “hot water”. To prove this reduction it is enough to notice that:

$$\frac{\text{Sum}_D[x,y]}{y-x+1} = \sum_{k=x}^{y} \frac{D[k]}{y-x+1} \geq t \iff \sum_{k=x}^{y}(D[k]-t) \geq 0.$$

Therefore, subtracting to all elements in D the density-threshold t, we can turn the problem stated in the second item above into the one that asks for the *longest segment that has sum larger or equal than* 0. Be careful that if you change the request from the *longest segment* to the *shortest one* whose density is larger than a threshold t, then the problem becomes trivial again: Just take the single occurrence of a nucleotide C or G. Similarly, if we fix an upper bound S to the segment’s sum (instead of a lower bound), then we can change the sign to all D’s elements and thus turn the problem again into a problem with a lower bound $t = -S$. So let us stick on the following general formulation:

Problem. *Given an array* $D[1,n]$ *of positive and negative numbers, we want to find the* longest *segment in D whose sum of its elements is* larger or equal than *a fixed threshold t.*

We notice that this formulation is in some sense a complement of the one given in the first item above. Here we maximize the segment length and pose a lower-bound to the sum of its elements; there, we maximized the sum of the segment provided that its length was within a given range. It is nice to observe that the structure of the algorithmic solution for both problems is similar, so we detail only the former one and refer the reader to the literature for the latter.

The algorithm proceeds inductively by assuming that, at step $i = 1, 2, \ldots, n$, it has computed the longest sub-array having sum larger than t and occurring within $D[1, i-1]$. Let us denote the solution available at the beginning of step i with $D[l_{i-1}, r_{i-1}]$. Initially we have $i = 1$ and thus the inductive solution is the empty one, hence having length equal to 0. To move from step i to step $i+1$, we need to compute $D[l_i, r_i]$ by possibly taking advantage of the currently known solution.

It is clear that the new segment is either inside $D[1, i-1]$ (namely $r_i < i$) or it ends at position $D[i]$ (namely $r_i = i$). The former case admits as solution the one of the previous iteration, namely $D[l_{i-1}, r_{i-1}]$, and so nothing has to be done: just set $r_i = r_{i-1}$ and $l_i = l_{i-1}$. The latter case is more involved and requires the use of some special data structures and a tricky analysis to show that the total complexity of the solution proposed is $O(n)$ in space and time, thus turning to be optimal!

We start by making a simple, yet effective, observation:

FACT 2.1

If $r_i = i$ then the segment $D[l_i, r_i]$ must be strictly longer than the segment $D[l_{i-1}, r_{i-1}]$. This means in particular that l_i occurs to the left of position $L_i = i - (r_{i-1} - l_{i-1})$.

The proof of this fact follows immediately by the observation that, if $r_i = i$, then the current step i has found a segment that improves the previously known one. Here “improved” means “longer” because the other constraint imposed by the problem formulation is boolean since it refers to a lower-bound on the segment’s sum. This is the reason why we can discard all positions within the range $[L_i, i]$, in fact they originate intervals of length shorter or equal than the previous solution $D[l_{i-1}, r_{i-1}]$.

Reformulated Problem. *Given an array* $D[1,n]$ *of positive and negative numbers, we want to find at every step the* smallest *index* $l_i \in [1, L_i)$ *such that* $\text{Sum}_D[l_i, i] \geq t$.

We point out that there could be many such indexes l_i, here we wish to find the *smallest* one because we aim at determining the *longest* segment.

At this point it is useful to recall that $\text{Sum}_D[l_i, i]$ can be re-written in terms of prefix-sums of array D, namely $\text{Sum}_D[1, i] - \text{Sum}_D[1, l_i - 1] = P[i] - P[l_i - 1]$ where the array P was introduced in Section 2.4. So we need to find the smallest index $l_i \in [1, L_i)$ such that $P[i] - P[l_i - 1] \geq t$. The array P can be pre-computed in linear time and space.

It is worth to observe that the computation of l_i could be done by scanning $P[1, L_i - 1]$ and searching for the *leftmost* index x such that $P[i] - P[x] \geq t$. We could then set $l_i = x + 1$ and have been done. Unfortunately, this is inefficient because it leads to scan over and over again the same positions of P as i increases, thus leading to a quadratic-time algorithm! Since we aim for a linear-time algorithm, we need to spend constant time "on average" per step i. We used the quotes because there is no *stochastic* argument here to compute the average, we wish only to capture syntactically the idea that, since we want to spend $O(n)$ time in total, our algorithm has to take constant time *amortized* per steps. In order to achieve this performance we first need to show that we can avoid the scanning of the whole prefix $P[1, L_i - 1]$ by identifying a *subset* of *candidate positions* for x. Call $C_{i,j}$ the candidate positions for iteration i, where $j = 0, 1, \ldots$. They are defined as follows: $C_{i,0} = L_i$ (it is a dummy value), and $C_{i,j}$ is defined inductively as the *leftmost minimum* of the sub-array $P[1, C_{i,j-1} - 1]$ (i.e. the sub-array to the left of the current minimum and/or to the left of L_i). We denote by $c(i)$ the number of these candidate positions for the step i, where clearly $c(i) \leq L_i$ (equality holds when $P[1, L_i]$ is decreasing).

For an illustrative example look at Figure 2.2, where $c(i) = 3$ and the candidate positions are connected via leftward arrows.

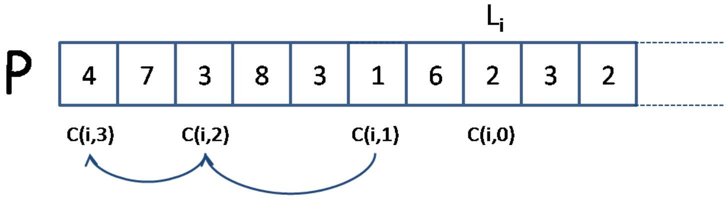


FIGURE 2.2: An illustrative example for the candidate positions $C_{i,j}$, given an array P of prefix sums. The picture is generic and reports only L_i for simplicity.

Looking at Figure 2.2 we derive three key properties whose proof is left to the reader because it immediately comes from the definition of $C_{i,j}$:

Property a. The sequence of candidate positions $C_{i,j}$ occurs within $[1, L_i)$ and moves leftward, namely $C_{i,j} < C_{i,j-1} < \ldots < C_{i,1} < C_{i,0} = L_i$.

Property b. At each iteration i, the sequence of candidate values $P[C_{i,j}]$ is increasing with $j = 1, 2, \ldots, c(i)$. More precisely, we have $P[C_{i,j}] > P[C_{i,j-1}] > \ldots > P[C_{i,1}]$ where the indices move leftward according to Property (a).

Property c. The value $P[C_{i,j}]$ is smaller than any other value on its left in P, because it is the leftmost minimum of the prefix $P[1, C_{i,j-1} - 1]$.

It is crucial now to show that the index we are searching for, namely l_i, can be derived by looking only at these candidate positions. In particular we can prove the following:

FACT 2.2
At each iteration i, the largest index j^* such that $Sum_D[C_{i,j^*} + 1, i] \geq t$ (if any) provides us with the longest segment we are searching for.

By Fact 2.1 we are interested in segments having the form $D[l_i, i]$ with $l_i < L_i$, and by properties of prefix-sums, we know that $\mathrm{Sum}_D[C_{i,j} + 1, i]$ can be re-written as $P[i] - P[C_{i,j}]$. Given this and Property (c), we can conclude that all segments $D[z, i]$, with $z < C_{i,j}$, have a sum *smaller* than $\mathrm{Sum}_D[C_{i,j} + 1, i]$. Consequently, if we find that $\mathrm{Sum}_D[C_{i,j} + 1, i] < t$ for some j, then we can discard all positions z to the left of $C_{i,j} + 1$ in the search for l_i. Therefore the index j^* characterized in Fact 2.2 is the one giving correctly $l_i = C_{i,j^*} + 1$.

There are two main problems in deploying the candidate positions for the efficient computation of l_i: (1) How do we compute the $C_{i,j}$s as i increases, (2) How do we search for the index j^*. To address issue (1) we notice that the computation of $C_{i,j}$ depends only on the position of the previous $C_{i,j-1}$ and *not* on the indices i or j. So we can define an auxiliary array $LMin[1, n]$ such that $LMin[i]$ is the leftmost position of the minimum within $P[1, i - 1]$. It is not difficult to see that $C_{i,1} = LMin[L_i]$, and that according to the definition of C it is $C_{i,2} = LMin[\, LMin[L_i]\,] = LMin^2[L_i]$. In general, it is $C_{i,k} = LMin^k[L_i]$. This allows an incremental computation:

$$LMin[x] = \begin{cases} 0 & \text{if } x = 0 \\ x - 1 & \text{if } P[x-1] < P[\, LMin[x-1]\,] \\ LMin[x-1] & \text{otherwise} \end{cases}$$

The formula above has an easy explanation. We know inductively $LMin[x - 1]$ as the leftmost minimum in the array $P[1, x - 2]$: initially we set $LMin[0]$ to the dummy value 0. To compute $LMin[x]$ we need to determine the leftmost minimum in $P[1, x - 1]$: this is located either in $x - 1$ (with value $P[x - 1]$) or it is the one determined for $P[1, x - 2]$ of position $LMin[x - 1]$ (with value $P[LMin[x - 1]]$). Therefore, by comparing these two values we can compute $LMin[x]$ in constant time. Hence the computation of all candidate positions $LMin[1, n]$ takes $O(n)$ time.

We are left with the problem of determining j^* efficiently. We will not be able to compute j^* in constant time at each iteration i but we will show that, if at step i we execute $s_i > 1$ steps, then we are advancing in the construction of the longest solution. Specifically, we are extending the length of that solution by $\Theta(s_i)$ units. Given that the longest segment cannot be longer than n, the sum of these extra-costs cannot be larger than $O(n)$, and thus we are done! This is called *amortized argument* because we are, in some sense, charging the cost of the expensive iterations to the cheapest ones. The computation of j^* at iteration i requires the check of the positions $LMin^k[L_i]$ for $k = 1, 2, \ldots$ until the condition in Fact 2.2 is satisfied; in fact, we know that all the other $j > j^*$ do not satisfy Fact 2.2. This search takes j^* steps and finds a new segment whose length is *increased* by at least j^* units, given Property (a) above. This means that either an iteration i takes constant time, because the check fails immediately at $LMin[L_i]$ (so the current solution is not better than the one computed at the previous iteration $i - 1$), or the iteration takes $O(j^*)$ time but the new segment $D[L_i, r_i]$ has been extended by j^* units. Since a segment cannot be longer than the entire sequence $D[1, n]$, we can conclude that the total extra-time cannot be larger than $O(n)$.

We leave to the diligent reader to work out the details of the pseudo-code of this algorithm, the techniques underlying its elegant design and analysis should be clear enough to approach it without any difficulties.

References

[1] Kun-Mao Chao. Maximum-density segment. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[2] Kun-Mao Chao. Maximum-scoring segment with length restrictions. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[3] Chih-Huai Cheng, Hsiao-Fei Liu, and Kun-Mao Chao. Optimal algorithms for the average-constrained maximum-sum segment problem. *Information Processing Letters*, 109(3):171–174, 2009.

[4] Rolf Fagerberg. Cache-oblivious model. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[5] Takeshi Fukuda, Yasuhiko Morimoto, Shinichi Morishita, and Takeshi Tokuyama. Mining optimized association rules for numeric attributes. *Journal of Computer System Sciences*, 58(1):1–12, 1999.

3

Random Sampling

"So much of life, it seems to me, is determined by pure randomness."
Sidney Poitier

This lecture attacks a simple-to-state problem which is the backbone of many randomized algorithms, and admits solutions which are algorithmically challenging to design and analyze.

> **Problem.** *Given a sequence of items $S = (i_1, i_2, \dots, i_n)$ and a positive integer $m \leq n$, the goal is to select a subset of m items from S uniformly at random.*

Uniformity here means that any item in S has to be sampled with probability $1/n$. Items can be numbers, strings or general objects either stored in a file located on disk or streaming through a channel. In the former scenario, the input size n is known and items occupy n/B pages, in the latter scenario, the input size may be even unknown yet the *uniformity* of the sampling process must be guaranteed. In this lecture we will address both scenarios aiming at efficiency in terms of I/Os, extra-space required for the computation (in addition to the input), and amount of randomness deployed (expressed as number of randomly generated integers). Hereafter, we will make use of a built-in procedure `Rand(a,b)` that randomly selects a number within the range $[a, b]$. The number, being either real or integer, will be clear from the context. The design of a good `Rand`-function is a challenging task, however we will not go into its details because we wish to concentrate in this lecture on the sampling process rather than on the generation of random numbers; though, the interested reader can refer to the wide literature about (pseudo-)random number generators.

Finally we notice that it is desirable to have the positions of the sampled items in *sorted* order because this speeds up their extraction from S both in the disk setting (less seek time) and in the stream-based setting (less passes over the data). Moreover it reduces the working space because it allows to extract the items efficiently via a scan, rather than using an auxiliary array of pointers to items. We do not want to detail further the sorting issue here, which gets complicated whenever $m > M$ and thus these positions cannot fit into internal memory. In this case we need a disk-based sorter, which is indeed an advanced topic of a subsequent lecture. If instead $m \leq M$ we could deploy the fact that positions are integers in a fixed range and thus use radix sort or any other faster routine available in the literature.

3.1 Disk model and known sequence length

We start by assuming that the input size n is known and that $S[1, n]$ is stored in a file on disk which cannot be modified because it may be the input of a more complicated problem that includes the current one as a sub-task. The first algorithm we propose is very simple, and allows us to arise some issues that will be attacked in the subsequent solutions.

Algorithm 3.1 Drawing from all un-sampled positions

```
1: Initialize the auxiliary array $S'[1,n] = S[1,n]$;
2: for $s = 0, 1, \dots, m-1$ do
3:     $p = \texttt{Rand}(1, n-s)$;
4:     select the item (pointed by) $S'[p]$;
5:     swap $S'[p]$ with $S'[n-s]$.
6: end for
```

At each step s the algorithm maintains the following invariant: *the subarray $S'[n-s+1,n]$ contains the items that have been already sampled, the rest of the items of S are contained in $S'[1,n-s]$*. Initially (i.e. $s = 0$) this invariant holds because $S'[n-s+1,n] = S'[n+1,n]$ is empty. At a generic step s, the algorithm selects randomly one item from $S'[1,n-s]$, and replaces it with the last item of that sequence (namely, $S'[n-s]$). This preserves the invariant for $s+1$. At the end (i.e. $s = m$), the sampled items are contained in $S'[n-m+1,n]$. We point out that S' cannot be a *pure* copy of S but it must be implemented as an *array of pointers* to S's items. The reason is that these items may have variable length (e.g. strings) so their retrieval in constant time could not be obtained via arithmetic operations, as well as the replacement step might be impossible due to difference in length between the item at $S'[p]$ and the item at $S'[n-s]$. Pointers avoid these issues but occupy $\Theta(n \log n)$ bits of space, which might be a non negligible space when n gets large and might turn out even larger than S if the average length of S's objects is shorter than $\log n$.[1] Another drawback of this approach is given by its pattern of memory accesses, which acts over $O(n)$ cells in purely random way, taking $\Theta(m)$ I/Os. This may be slow when $m \approx n$, so in this case we would like to obtain $O(n/B)$ I/Os which is the cost of scanning the whole S.

Let us attack these issues by proposing a series of algorithms that incrementally improve the I/Os and the space resources of the previous solution, up to the final result that will achieve $O(m)$ extra space, $O(m)$ average time and $O(\min\{m, n/B\})$ I/Os. We start by observing that the swap of the items in Step 5 of `Algorithm 3.1` guarantees that every step generates one distinct item, but forces to duplicate S and need $\Omega(m)$ I/Os whichever is the value of m. The following `Algorithm 3.2` improves the I/O- and space-complexities by avoiding the item-swapping via the use of an auxiliary data structure that keeps track of the selected positions in sorted order and needs only $O(m)$ space.

Algorithm 3.2 Dictionary of sampled positions

```
Initialize the dictionary $\mathcal{D} = \emptyset$;
while $(|\mathcal{D}| < m)$ do
    $p = \texttt{Rand}(1, n)$;
    if $p \notin \mathcal{D}$ insert it;
end while
```

[1] This may occur only if S contains duplicate items, otherwise a classic combinatorial argument applies.

`Algorithm 3.2` stops when $\mathcal{D}$ contains m (distinct) integers which constitute the positions of the items to be sampled. According to our observation made at the beginning of the lecture, $\mathcal{D}$ may be sorted before S is accessed on disk to reduce the seek time. In any case, the efficiency of the algorithm mainly depends on the implementation of the dictionary $\mathcal{D}$, which allows to detect the presence of duplicate items. The literature offers many data structures that efficiently support membership and insert operations, based either on hashing or on trees. Here we consider only an hash-based solution which consists of implementing $\mathcal{D}$ via a hash table of size $\Theta(m)$ with collisions managed via chaining and a universal hash function for table access [1]. This way each membership query and insertion operation over $\mathcal{D}$ takes $O(1)$ time on average (the load factor of this table is $O(1)$), and total space $O(m)$. Time complexity could be forced to be worst case by using more sophisticated data structures, such as dynamic perfect hashing, but the final time bounds would always be *in expectation* because of the underlying re-sampling process.

However this algorithm may generate *duplicate* positions, which must be discarded and *re-sampled*. Controlling the cost of the re-sampling process is the main drawback of this approach, but this induces a constant-factor slowdown on average, thus making this solution much appealing in practice. In fact, the probability of having extracted an item already present in $\mathcal{D}$ is $|\mathcal{D}|/n \leq m/n < 1/2$ because, without loss of generality, we can assume that $m < n/2$ otherwise we can solve the *complement* of the current problem and thus randomly select the positions of the items that are *not* sampled from S. So we need an average of $O(1)$ re-samplings in order to obtain a new item for $\mathcal{D}$, and thus advancing in our selection process. Overall we have proved the following:

FACT 3.1 `Algorithm 3.2` *based on hashing with chaining requires $O(m)$ average time and takes $O(m)$ additional space to select uniformly at random m positions in $[1, n]$. The average depends both on the use of hashing and the cost of re-sampling. An additional sorting-cost is needed if we wish to extract the sampled items of S in a streaming-like fashion. In this case the overall sampling process takes $O(\min\{m, n/B\})$ I/Os.*

If we substitute hashing with a (balanced) search tree and assume to work in the RAM model (hence we assume $m < M$), then we can avoid the sorting step by performing an *in-visit* of the search tree in $O(m)$ time. However, `Algorithm 3.2` would still require $O(m \log m)$ time because each insertion/membership operation would take $O(\log m)$ time. We could do better by deploying an *integer*-based dictionary data structure, such as a van Emde-Boas tree, and thus take $O(\log \log n)$ time for each dictionary operation. The two bounds would be incomparable, depending on the relative magnitudes of m and n. Many other trade-offs are possible by changing the underlying dictionary data structure, so we leave to the reader this exercise.

The next step is to avoid a dictionary data structure and use *sorting* as a basic block of our solution. This could be particularly useful in practice because comparison-based sorters, such as `qsort`, are built-in in many programming languages. The following analysis will have also another side-effect which consists of providing a more clean evaluation of the average time performance of `Algorithm 3.2`, rather than just saying re-sample each item at most $O(1)$ times on average.

The cost of `Algorithm 3.3` depends on the number of times the sorting step is repeated and thus do exist duplicates in the sampled items. We argue that a small number of re-samplings is needed. So let us compute the probability that `Algorithm 3.3` executes just one iteration: this means that the m sampled items are all distinct. This analysis is well known in the literature and goes under the name of *birthday problem*: how many people do we need in a room in order to have a probability larger than $1/2$ that at least two of them have the same birthday. In our context we have that people = items and birthday = position in S. By mimicking the analysis done for the birthday problem, we

Algorithm 3.3 Sorting

```
$\mathcal{D} = \emptyset$;
while ($|\mathcal{D}| < m$) do
    $\mathcal{X}$ = randomly draw $m$ positions from $[1, n]$;
    Sort $\mathcal{X}$ and eliminate the duplicates;
    Set $\mathcal{D}$ as the resulting $\mathcal{X}$;
end while
```

can estimate the probability that a duplicate among m randomly-drawn items does not occur as:

$$\frac{m!\binom{n}{m}}{n^m} = \frac{n(n-1)(n-2)\cdots(n-m+1)}{n^m} = 1 \times (1 - \frac{1}{n}) \times (1 - \frac{2}{n}) \times \cdots (1 - \frac{m-1}{n})$$

Given that $e^x \geq 1 + x$, we can upper bound the above formula as:

$$e^0 \times e^{-1/n} \times e^{-2/n} \times \cdots e^{-(m-1)/n} = e^{-(1+2+\cdots+(m-1))/n} = e^{-m(m-1)/2n}$$

So the probability that a duplicate *does not* occur is at most $e^{-m(m-1)/2n}$ and, in the case of the birthday paradox in which $n = 365$, this is slightly smaller than one-half already for $m = 23$. In general we have that $m = \sqrt{n}$ elements suffices to make the probability of a duplicate at least $1 - \frac{1}{\sqrt{e}} \approx 0.4$, thus making our algorithm need re-sampling. On the positive side, we notice that if $m \ll \sqrt{n}$ then e^x can be well approximated with $1 + x$, so $e^{-m(m-1)/2n}$ is not only an upper-bound but also a reasonable estimate of the collision probability and it could be used to estimate the number of re-samplings needed to complete `Algorithm 3.3`.

FACT 3.2 `Algorithm 3.3` *requires a constant number of sorting steps on average, and $O(m)$ additional space, to select uniformly at random m items from the sequence $S[1, n]$. This is $O(m \log m)$ average time and* $\min\{m, n/B\}$ *worst-case I/Os if $m \leq M$ is small enough to keep the sampled positions in internal memory. Otherwise an external-memory sorter is needed. The average depends on the re-sampling, integers are returned in sorted order for streaming-like access to the sequence S.*

Sorting could be speeded up by deploying the specialty of our problem, namely, that items to be sorted are m random integers in a fixed range $[1, n]$. Hence we could use either radix-sort or, even better for its simplicity, bucket sort. In the latter case, we can use an array of m slots each identifying a range of n/m positions in $[1, n]$. If item i_j is randomly selected, then it is inserted in slot $\lceil jm/n \rceil$. Since the m items are randomly sampled, each slot will contain $O(1)$ items on average, so that we can sort them in constant time per bucket by using insertion sort or the built-in `qsort`.

FACT 3.3 `Algorithm 3.3` *based on bucket-sort requires $O(m)$ average time and $O(m)$ additional space, whenever $m \leq M$. The average depends on the re-sampling. Integers are returned in sorted order for streaming-like access to the sequence S.*

We conclude this section by noticing that all the proposed algorithms, except `Algorithm 3.1`, generate the set of sampled positions using $O(m)$ space. If $m \leq M$ the random generation can occur within main memory without incurring in any I/Os. Sometimes this is useful because the randomized algorithm that invokes the random-sampling subroutine does not need the corresponding items, but rather their positions.

3.2 Streaming model and known sequence length

We next turn to the case in which S is flowing through a channel and the input size n is known and big (e.g. Internet traffic or query logs). We will turn to the more general case in which n is unknown at the end of the lecture, in the next section. This stream-based model imposes that no preprocessing is possible (as instead done above where items' positions were re-sampled and/or sorted), every item of S is considered once and the algorithm must immediately and irrevocably take a decision whether or not that item must be included or not in the set of sampled items. Possibly future items may kick out that one from the sampled set, but no item can be re-considered again in the future. Even in this case the algorithms are simple in their design but their probabilistic analysis is a little bit more involved than before. The algorithms of the previous section offer an *average* time complexity because they are faced with the re-sampling problem: possibly some samples have to be eliminated because duplicated. In order to avoid re-sampling, we need to ensure that each item is not considered more than once. So the algorithms that follow implement this idea in the simplest possible way, namely, they scan the input sequence S and consider each item once for all. This approach brings with itself two main difficulties which are related with the guarantee of both conditions: uniform sample from the range $[1, n]$ and sample of size m.

We start by designing an algorithm that draws just one item from S (hence $m = 1$), and then we generalize it to the case of a subset of $m > 1$ items. This algorithm proceeds by selecting the item $S[j]$ with probability $\mathcal{P}(j)$ which is properly defined in order to guarantee both two properties above.[2] In particular we set $\mathcal{P}(1) = 1/n, \mathcal{P}(2) = 1/(n-1), \mathcal{P}(3) = 1/(n-2)$ etc. etc., so the algorithm selects the item j with probability $\mathcal{P}(j) = \frac{1}{n-j+1}$, and if this occurs it stops. Eventually item $S[n]$ is selected because its drawing probability is $\mathcal{P}(n) = 1$. So the proposed algorithm guarantees the condition on the sample size $m = 1$, but more subtle is to prove that the probability of sampling $S[j]$ is $1/n$, independently of j, given that we defined $\mathcal{P}(j) = 1/(n - j + 1)$. The reason derives from a simple probabilistic argument because $n - j + 1$ is the number of remaining elements in the sequence and all of them have to be drawn uniformly at random. By induction, the first $j - 1$ items of the sequence have uniform probability $1/n$ to be sampled; so it is $1 - \frac{j-1}{n}$ the probability of not selecting anyone of them. As a result,

$$\mathcal{P}(\text{Sample } i_j) = \mathcal{P}(\text{Not sampling } i_1 \cdots i_{j-1}) \times \mathcal{P}(\text{Picking } i_j) = (1 - \frac{j-1}{n}) \times \frac{1}{n-j+1} = 1/n$$

Algorithm 3.4 Scanning and selecting

```
$s = 0$;
for ($j = 1$; $j \le n$; $j$++) do
    $p$ = Rand(0, 1);
    if ($p \le \frac{m-s}{n-j+1}$) then
        select $S[j]$;
        $s$++;
    end if
end for
```

[2] In order to draw an item with probability p, it suffices to draw a random real $r \in [0, 1]$ and then compare it against p. If $r \le p$ then the item is selected, otherwise it is not.

`Algorithm 3.4` works for an arbitrarily large sample $m \geq 1$. The difference with the previous algorithm lies in the probability of sampling $S[j]$ which is now set to $\mathcal{P}(j) = \frac{m-s}{n-j+1}$ where s is the number of items already selected before $S[j]$. Notice that if we already got all samples, it is $s = m$ and thus $\mathcal{P}(j) = 0$, which correctly means that `Algorithm 3.4` does not generate more than m samples. On the other hand, it is easy to convince ourselves that `Algorithm 3.4` cannot generate less than m items, say y, given that the last $m - y$ items of S would have probability 1 to be selected and thus they would be surely included in the final sample (according to Step 4). As far as the uniformity of the sample is concerned, we show that $\mathcal{P}(j)$ equals the probability that $S[j]$ is included in a random sample of size m given that s samples lie within $S[1, j-1]$. We can rephrase this as the probability that $S[j]$ is included in a random sample of size $m - s$ taken from $S[j, n]$, and thus from $n - j + 1$ items. This probability is obtained by counting how many such combinations include $S[j]$, i.e. $\binom{n-j}{m-s-1}$, and dividing by the number of all combinations that include or not $S[j]$, i.e. $\binom{n-j+1}{m-s}$. Substituting to $\binom{b}{a} = \frac{b!}{a!\,(b-a)!}$ we get the formula for $\mathcal{P}(j)$.

FACT 3.4 `Algorithm 3.4` *takes $O(n/B)$ I/Os, $O(n)$ time, n random samples, and $O(m)$ additional space to sample uniformly m items from the sequence $S[1, n]$ in a streaming-like way.*

We conclude this section by pointing out a sophisticated solution proposed by Jeff Vitter [2] that reduces the amount of randomly-generated numbers from n to m, and thus speeds up the solution to $O(m)$ time and I/Os. This solution could be also fit into the framework of the previous section (random access to input data), and in that case its specialty would be the avoidance of re-sampling. Its key idea is not to generate random *indicators*, which specify whether or not an item $S[j]$ has to be selected, but rather generate random *jumps* that count the number of items to skip over before selecting the next item of S. Vitter introduces a random variable $G(v, V)$ where v is the number of items remaining to be selected, and V is the total number of items left to be examined in S. According to our previous notation, we have that $v = m - s$ and $V = n - j + 1$. The item $S[G(v, V) + 1]$ is the next one selected to form the uniform sample. It goes without saying that this approach avoids the generation of duplicate samples, but yet it incurs in an average bound because of the cost of generating the jumps according to the following distribution:

$$\mathcal{P}(G = g) = \binom{V - g - 1}{v - 1} / \binom{V}{v}$$

In fact the key problem here is that we cannot tabulate (and store) the values of all binomial coefficients in advance, because this would need space exponential in $V = \Theta(n)$. Surprisingly Vitter solved this problem in $O(1)$ average time, by adapting in an elegant way the von Neumann's rejection-acceptance method to the discrete case induced by G's jumps. We refer the reader to [2] for further details.

3.3 Streaming model and unknown sequence length

It goes without saying that the knowledge of n was crucial to compute $\mathcal{P}(j)$ in `Algorithm 3.4`. If n is unknown we need to proceed differently, and indeed the rest of this lecture is dedicated to detail two possible approaches.

The first one is pretty much simple and deploys a min-heap $\mathcal{H}$ of size m plus a real-number random generator, say `Rand(0,1)`. The key idea underlying this algorithm is to associate a random key to each item of S and then use the heap $\mathcal{H}$ to select the items corresponding to the top-m keys. The pseudo-code below implements this idea, we notice that $\mathcal{H}$ is a min-heap so it takes $O(1)$ time

Algorithm 3.5 Heap and random keys

```
Initialize the heap $\mathcal{H}$ with $m$ dummy pairs $\langle -\infty, \emptyset \rangle$;
for each item $S[j]$ do
    $r_j$ = Rand(0, 1);
    $m$ = the minimum key in $\mathcal{H}$;
    if ($r_j > m$) then
        extract the minimum key;
        insert $\langle r_j, S[j] \rangle$ in $\mathcal{H}$;
    end if
end for
return $\mathcal{H}$
```

to detect the minimum key among the current top-m ones. This is the key compared with r_j in order to establish whether or not $S[j]$ must enter the top-m set.

Since the heap has size m, the final sample will consists of m items. Each item takes $O(\log m)$ time to be inserted in the heap. So we have proved the following:

FACT 3.5 `Algorithm 3.5` *takes $O(n/B)$ I/Os, $O(n \log m)$ time, generates n random numbers, and uses $O(m)$ additional space to sample uniformly at random m items from the sequence $S[1, n]$ in a streaming-like way and without the knowledge of n.*

We conclude the lecture by introducing the elegant *reservoir sampling* algorithm, designed by Knuth in 1997, which improves `Algorithm 3.5` both in time and space complexity. The idea is similar to the one adopted for `Algorithm 3.4` and consists of properly defining the probability with which an item is selected. The key issue here is that we cannot take an irrevocable decision on $S[j]$ because we do not know how long the sequence S is, so we need some freedom to *change* what we have decided so far as the scanning of S goes on.

Algorithm 3.6 Reservoir sampling

```
Initialize array $R[1, m] = S[1, m]$;
for each next item $S[j]$ do
    $h$ = Rand(1, $j$);
    if $h \leq m$ then
        set $R[h] = S[j]$;
    end if
end for
return array $R$;
```

The pseudo-code of `Algorithm 3.6` uses a "reservoir" array $R[1, m]$ to keep the candidate samples. Initially R is set to contain the first m items of the input sequence. At any subsequent step j, the algorithm makes a choice whether $S[j]$ has to be included or not in the current sample. This choice occurs with probability $\mathcal{P}(j) = m/j$, in which case some previously selected item has to be kicked out from R. This item is chosen at random, hence with probability $1/m$. This double-choice is implemented in `Algorithm 3.6` by choosing an integer h in the range $[1, j]$, and making the substitution only if $h \leq m$. This event has probability m/j: exactly what we wished to set for $\mathcal{P}(j)$.

For the correctness, it is clear that `Algorithm 3.6` selects m items, it is less clear that these items are drawn uniformly at random from S, which actually means with probability m/n. Let's see why by assuming inductively that this property holds for a sequence of length $n-1$. The base case in which $n = m$ is obvious, every item has to be selected with probability $m/n = 1$, and indeed this is what Step 1 does by selecting all $S[1,m]$ items. To prove the inductive step (from $n-1$ to n items), we notice that the uniform-sampling property holds for $S[n]$ since by definition that item is inserted in R with probability $\mathcal{P}(n) = m/n$ (Step 4). Computing the probability of being sampled for the other items in $S[1,n-1]$ is more difficult to see. An item belongs to the reservoir R at the n-th step of `Algorithm 3.6` iff it was in the reservoir at the $(n-1)$-th step and it is not kicked out at the n-th step. This latter may occur either if $S[n]$ is not picked (and thus R is untouched) or if $S[n]$ is picked and $S[j]$ is not kicked out from R (being these two events independent of each other). In formulas,

$$\begin{aligned}\mathcal{P}(\text{item } S[j] \in R \text{ after } n \text{ items}) &= \mathcal{P}(S[j] \in R \text{ after } n-1 \text{ items}) \times [\mathcal{P}(S[n] \text{ is not picked})\\ &\quad + \mathcal{P}(S[n] \text{ is picked}) \times \mathcal{P}(S[j] \text{ is not removed from } R)]\end{aligned}$$

Now, each of these items has probability $m/(n-1)$ of being in the reservoir R, by the inductive hypothesis, before that $S[n]$ is processed. Item $S[j]$ remains in the reservoir if either $S[n]$ is not picked (which occurs with probability $1 - \frac{m}{n}$) or if it is not kicked out by the picked $S[n]$ (which occurs with probability $\frac{m-1}{m}$). Summing up these terms we get

$$\mathcal{P}(\text{item } S[j] \in R \text{ after } n \text{ items}) = \frac{m}{n-1} \times [(1 - \frac{m}{n}) + (\frac{m}{n} \times \frac{m-1}{m})] = \frac{m}{n-1} \times \frac{n-1}{n} = \frac{m}{n}$$

To understand this formula assume that we have a reservoir of 1000 items, so the first 1000 items of S are inserted in R by Step 1. Then the item 1001 is inserted in the reservoir with probability 1000/1001, the item 1002 with probability 1000/1002, and so on. Each time an item is inserted in the reservoir, a random element is kicked out from it, hence with probability 1/1000. After n steps the reservoir R contains 1000 items, each sampled from S with probability $1000/n$.

FACT 3.6 `Algorithm 3.6` *takes $O(n/B)$ I/Os, $O(n)$ time, n random numbers, and exactly m additional space, to sample uniformly at random m items from the sequence $S[1,n]$ in a streaming-like way and without the knowledge of n. Hence it is time, space and I/O-optimal in this model of computation.*

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. Chapter 11: "Hashing", The MIT press, third edition, 2009.

[2] Jeffrey Scott Vitter. Faster methods for random sampling. *ACM Computing Surveys*, 27(7):703–718, 1984.

4

List Ranking

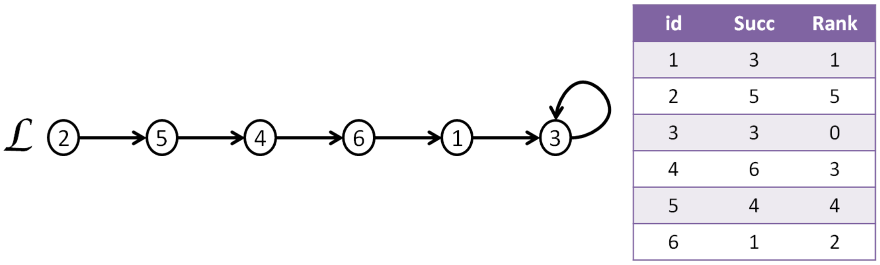
“Pointers are dangerous in disks!”

This lecture attacks a simple problem over lists, the basic data structure underlying the design of many algorithms which manage interconnected items. We start with an easy to state, but inefficient solution derived from the optimal one known for the RAM model; and then discuss more and more sophisticated solutions that are elegant, efficient/optimal but still simple enough to be coded with few lines. The treatment of this problem will allow also us to highlight a subtle relation between *parallel computation* and *external-memory computation*, which can be deployed to derive efficient disk-aware algorithms from efficient parallel algorithms.

Problem. *Given a (mono-directional) list $\mathcal{L}$ of n items, the goal is to compute the distance of each of those items from the tail of $\mathcal{L}$.*

Items are represented via their ids, which are integers from 1 to n. The list is encoded by means of an array $\texttt{Succ}[1, n]$ which stores in entry $\texttt{Succ}[i]$ the id j if item *i points to* item j. If t is the id of the tail of the list $\mathcal{L}$, then we have $\texttt{Succ}[t] = t$, and thus the link outgoing from t forms a *self-loop*. The following picture exemplifies these ideas by showing a graphical representation of a list (left), its encoding via the array Succ (right), and the output required by the list-ranking problem, hereafter encoded in the array $\texttt{Rank}[1, n]$.

This problem can be solved easily in the RAM model by exploiting the constant-time access to its internal memory. We can foresee three solutions. The first one scans the list from its head and computes the number n of its items, then re-scans the list by assigning to its head the rank $n-1$ and to



id	Succ	Rank
1	3	1
2	5	5
3	3	0
4	6	3
5	4	4
6	1	2

FIGURE 4.1: An example of input and output for the List Ranking problem.

every subsequent element in the list a value decremented by one at every step. The second solution computes the array of predecessors as Pred[Succ[i]] = i; and then scans the list backward, starting from its tail t, setting Rank[t] = 0, and then incrementing the Rank's value of each item as the percolated distance from t. The third way to solve the problem is *recursively*, without needing (an explicit) additional working space, by defining the function ListRank(i) which works as follows: Rank[i] = 0 if Succ[i] = i (and hence $i = t$), else it sets Rank[i] = ListRank(Succ[i]) + 1; at the end the function returns the value Rank[i]. The time complexity of both algorithms is $O(n)$, and obviously it is optimal since all list's items must be visited to set their n Rank's values.

If we execute this algorithm over a list stored on disk (via its array Succ), then it could elicit $\Theta(n)$ I/Os because of the arbitrary distribution of links which might induce an irregular pattern of disk accesses to the entries of arrays Rank and Succ. This I/O-cost is significantly far from the lower-bound $\Omega(n/B)$ which can be derived by the same argument we used above for the RAM model. Although this lower-bound seems very low, we will come in this lecture very close to it by introducing a bunch of sophisticated techniques that are general enough to find applications in many other, apparently dissimilar, contexts.

The moral of this lecture is that, in order to achieve I/O-efficiency on *linked* data structures, you need to avoid the *percolation* of pointers as much as possible; and possibly dig into the wide parallel-algorithms literature (see e.g. [2]) because efficient parallelism can be turned surprisingly into I/O-efficiency.

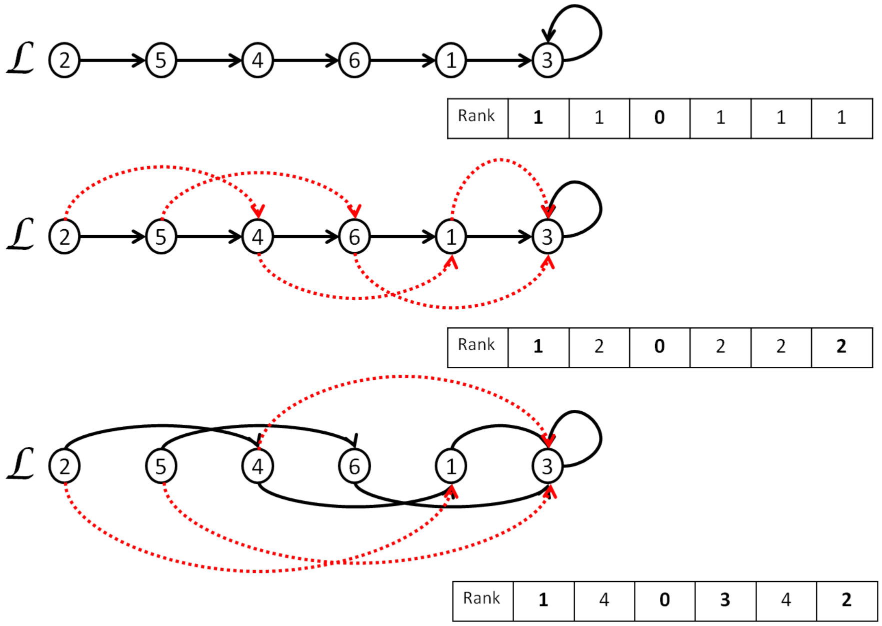
4.1 The pointer-jumping technique

There exists a well-known technique to solve the list-ranking problem in the parallel setting, based on the so called *pointer jumping* technique. The algorithmic idea is pretty much simple, it takes n processors, each dealing with one item of $\mathcal{L}$. Processor i initializes Rank[i] = 0 if $i = t$, otherwise it sets Rank[i] = 1. Then executes the following two instructions: Rank[i] += Rank[Succ[i]], Succ[i] = Succ[Succ[i]]. This update actually maintains the following invariant: Rank[i] *measures the distance (number of items) between i and the current* Succ[i] *in the original list.* We skip the formal proof that can be derived by induction, and refer the reader to the illustrative example in Figure 4.2.

In that Figure the red-dashed arrows indicate the new links computed by one pointer-jumping step, and the table on the right of each list specifies the values of array Rank[$1, n$] as they are recomputed after this step. The values in bold are the final/correct values. We notice that distances do not grow linearly (i.e. $1, 2, 3, \ldots$) but they grow as a power of two (i.e. $1, 2, 4, \ldots$), up to the step in which the next jump leads to reach t, the tail of the list. This means that the total number of times the parallel algorithm executes the two steps above is $O(\log n)$, thus resulting an exponential improvement with respect to the time required by the sequential algorithm. Given that n processors are involved, pointer-jumping executes a total of $O(n \log n)$ operations, which is inefficient if we compare it to the number $O(n)$ operations executed by the optimal RAM algorithm.

LEMMA 4.1 The parallel algorithm, using n processors and the pointer-jumping technique, takes $O(\log n)$ time and $O(n \log n)$ operations to solve the list-ranking problem.

Optimizations are possible to further improve the previous result and come close the optimal number of operations; for example, by *turning off* processors, as their corresponding items reach the end of the list, could be an idea but we will not dig into these details (see e.g. [2]) because they pertain to a course on parallel algorithms. Here we are interested in *simulating* the pointer-jumping technique in our setting which consists of one single processor and a 2-level memory, and show that deriving an I/O-efficient algorithm is very simple whenever an efficient parallel algorithm is



Rank	**1**	1	**0**	1	1	1

Rank	**1**	2	**0**	2	2	**2**

Rank	**1**	4	**0**	**3**	4	**2**

FIGURE 4.2: An example of pointer jumping applied to the list $\mathcal{L}$ of Figure 4.1. The dotted arrows indicate one pointer-jumping step applied onto the solid arrows, which represent the current configuration of the list.

available. The simplicity hinges onto an algorithmic scheme which deploys two basic primitives— Scan and Sort a set of triples— nowadays available in almost every distributed platforms, such as Apache Hadoop.

4.2 Parallel algorithm simulation in a 2-level memory

The key difficulty in using the pointer-jumping technique within the 2-level memory framework is the arbitrary layout of the list on disk, and the consequent arbitrary pattern of memory accesses to update Succ-pointers and Rank-values, which might induce many I/Os. To circumvent this problem we will describe how the two key steps of the pointer-jumping approach can be simulated via a constant number of Sort and Scan primitives over n triples of integers. Sorting is a basic primitive which is very much complicated to be implemented I/O-efficiently, and indeed will be the subject of the entire Chapter 5. For the sake of presentation, we will indicate its I/O-complexity as $\widetilde{O}(n/B)$ which means that we have hidden a logarithmic factor depending on the main parameters of the model, namely M, n, B. This factor is negligible in practice, since we can safely upper bound it with 4 or less, and so we prefer now to *hide it* in order to avoid jeopardizing the reading of this chapter. On the other hand, Scan is easy and takes $O(n/B)$ I/Os to process a contiguous disk portion occupied by the n triples.

We can identify a common algorithmic structure in the two steps of the pointer-jumping technique: each of them consists of an operation (either copy or sum) between two entries of an array (either Succ or Rank). For the sake of presentation we will refer to a generic array A, and model the parallel operation to be simulated on disk as follows:

Assume that a parallel step has the following form: $A[a_i]$ `op` $A[b_i]$*, where* `op` *is the operation executed in parallel over the two array entries* $A[a_i]$ *and* $A[b_i]$ *by all processors* $i = 1, 2, \ldots, n$ *which actually read* $A[b_i]$ *and use this value to update the content of* $A[a_i]$.

The operation `op` is a sum and an assignment for the update of the `Rank`-array (here $A =$ `Rank`), it is a copy for the update of the `Succ`-array (here $A =$ `Succ`). As far as the array indices are concerned they are, for both steps, $a_i = i$ and $b_i =$ `Succ`$[i]$. The key issue is to show that $A[a_i]$ `op` $A[b_i]$ can be implemented, simultaneously over all $i = 1, 2, 3, \ldots, n$, by using a constant number of `Sort` and `Scan` primitives, thus taking a total of $\widetilde{O}(n/B)$ I/Os. The simulation consists of 5 steps:

1. `Scan` the disk and create a sequence of triples having the form $\langle a_i, b_i, 0\rangle$. Every triple brings information about the source address of the array-entry involved in `op` (b_i), its destination address (a_i), and the value that we are moving (the third component, initialized to 0).
2. `Sort` the triples according to their second component (i.e. b_i). This way, we are ”aligning” the triple $\langle a_i, b_i, 0\rangle$ with the memory cell $A[b_i]$.
3. `Scan` the triples and the array A to create the new triples $\langle a_i, b_i, A[b_i]\rangle$. Notice that not all memory cells of A are referred as second component of any triple, nevertheless their coordinated order allows to copy $A[b_i]$ into the triple for b_i via a coordinated scan.
4. `Sort` the triples according to their first component (i.e. a_i). This way, we are aligning the triple $\langle a_i, b_i, A[b_i]\rangle$ with the memory cell $A[a_i]$.
5. `Scan` the triples and the array A and, for every triple $\langle a_i, b_i, A[b_i]\rangle$, update the content of the memory cell $A[a_i]$ according to the semantics of `op` and the value $A[b_i]$.

I/O-complexity is easy to derive since the previous algorithm is executing 2 `Sort` and 3 `Scan` involving n triples. Therefore we can state the following:

THEOREM 4.1 *The parallel execution of n operations* $A[a_i]$ `op` $A[b_i]$ *can be simulated in a* 2*-level memory model by using a constant number of* `Sort` *and* `Scan` *primitives, thus taking a total of* $\widetilde{O}(n/B)$ *I/Os.*

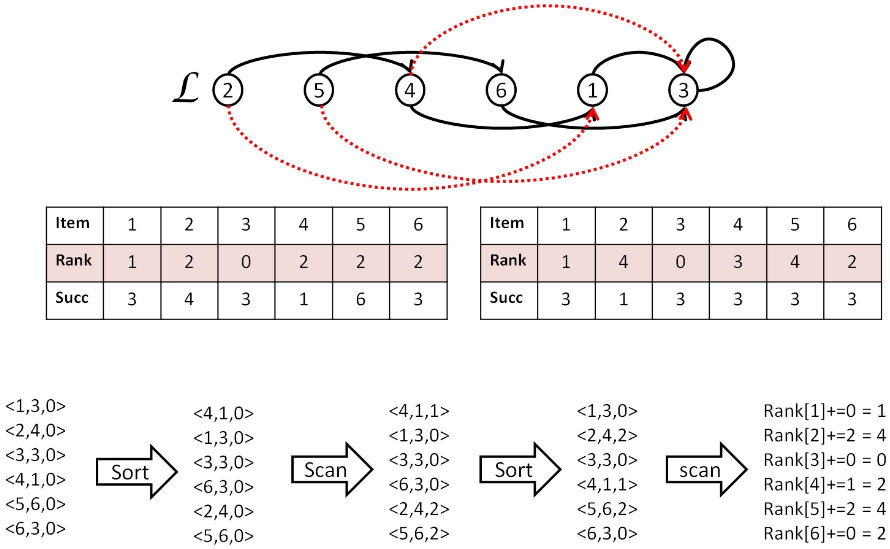
In the case of the parallel pointer-jumping algorithm, this parallel assignment is executed for $O(\log n)$ steps, so we have:

THEOREM 4.2 *The parallel pointer-jumping algorithm can be simulated in a* 2*-level memory model taking* $\widetilde{O}((n/B)\log n)$ *I/Os.*

This bound turns to be $o(n)$, and thus better than the direct execution of the sequential algorithm on disk, whenever $B = \omega(\log n)$. This condition is trivially satisfied in practice because $B \approx 10^4$ bytes and $\log n \leq 80$ for any real dataset size (being 2^{80} the number of atoms in the Universe[1]).

Figure 4.3 reports a running example of this simulation over the list at the top of the Figure 4.3. Table on the left indicates the content of the arrays `Rank` and `Succ` encoding the list; table on the right indicates the content of these two arrays after one step of pointer-jumping. The five columns of triples correspond to the application of the five `Scan`/`Sort` phases. This simulation is related to the update of the array `Rank`, array `Succ` can be recomputed similarly. Actually, the update

[1] See e.g. `http://en.wikipedia.org/wiki/Large_numbers`



Item	1	2	3	4	5	6
Rank	1	2	0	2	2	2
Succ	3	4	3	1	6	3

Item	1	2	3	4	5	6
Rank	1	4	0	3	4	2
Succ	3	1	3	3	3	3

FIGURE 4.3: An example of simulation of the basic parallel step via Scan and Sort primitives, relative to the computation of the array Rank, with the configuration specified in the picture and tables above.

can be done simultaneously by using a quadruple instead of a triple which brings both the values of Rank[Succ[i]] and the value of Succ[Succ[i]], thus deploying the fact that both values use the same source and destination address (namely, i and Succ[i]).

The first column of triples is created as $\langle i, \texttt{Succ}[i], 0\rangle$, since $a_i = i$ and $b_i = \texttt{Succ}[i]$. The third column of triples is sorted by the second component, namely Succ[i], and so its third component is obtained by Scanning the array Rank and creating $\langle i, \texttt{Succ}[i], \texttt{Rank}[\texttt{Succ}[i]]\rangle$. The fourth column of triples is ordered by their first component, namely i, so that the final Scan-step can read in parallel the array Rank and the third component of those triples, and thus compute correctly Rank[i] as Rank[i] + Rank[Succ[i]] = 1 + Rank[Succ[i]].

The simulation scheme introduced in this section can be actually generalized to every parallel algorithm thus leading to the following important, and useful, result (see [1]):

THEOREM 4.3 *Every parallel algorithm using n processors and taking T steps can be simulated in a 2-level memory by a disk-aware sequential algorithm taking $\widetilde{O}((n/B)\ T)$ I/Os and $O(n)$ space.*

This simulation is advantageous whenever $T = o(B)$, which implies a sub-linear number of I/Os $o(n)$. This occurs in all cases in which the parallel algorithm takes a low *poly-logarithmic* time-complexity. This is exactly the situation of parallel algorithms developed over the so called P-RAM model of computation which assumes that all processors work independently of each other and they can access in constant time an unbounded shared memory. This is an ideal model which was very famous in the '80s-'90s and led to the design of many powerful parallel techniques, which have been then applied to distributed as well as disk-aware algorithms. Its main limit was to do not account for conflicts among the many processors accessing the shared memory, and a simplified communication among them. Nevertheless this simplified model allowed researchers to concentrate onto the algorithmic aspects of parallel computation and thus design precious parallel schemes as the ones described below.

4.3 A Divide&Conquer approach

The goal of this section is to show that the list-ranking problem can be solved more efficiently than pointer-jumping on a list. The algorithmic solution we describe in this section relies on an interesting application of the *Divide&Conquer paradigm*, here specialized to work on a (mono-directional) list of items.

Before going into the technicalities related to this application, let us briefly recall the main ideas underlying the design of an algorithm, say $\mathcal{A}_{dc}$, based on the Divide&Conquer technique which solves a problem $\mathcal{P}$, formulated on n input data. $\mathcal{A}_{dc}$ consists of three main phases:

Divide. $\mathcal{A}_{dc}$ creates a set of k *subproblems*, say $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k$, having sizes $n_1, n_2, \ldots, n_k$, respectively. They are identical to the original problem $\mathcal{P}$ but are formulated on smaller inputs, namely $n_i < n$.

Conquer. $\mathcal{A}_{dc}$ is invoked *recursively* on the subproblems $\mathcal{P}_i$, thus getting the solution s_i.

Recombine. $\mathcal{A}_{dc}$ recombines the solutions s_i to obtain the solution s for the original problem $\mathcal{P}$. s is *returned* as output of the algorithm.

It is clear that the Divide&Conquer technique originates a *recursive* algorithm $\mathcal{A}_{dc}$, which needs a *base case* to terminate. Typically, the base case consists of stopping $\mathcal{A}_{dc}$ whenever the input consists of few items, e.g. $n \leq 1$. In these small-input cases the solution can be computed easily and directly, possibly by enumeration.

The time complexity $T(n)$ of $\mathcal{A}_{dc}$ can be described as a recurrence relation, in which the base condition is $T(n) = O(1)$ for $n \leq 1$, and for the other cases it is:

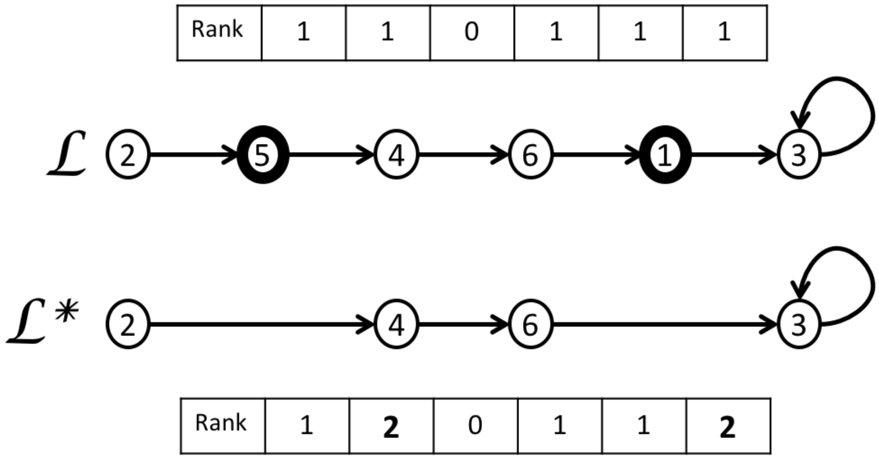
$$T(n) = D(n) + R(n) + \sum_{i=1,\ldots,k} T(n_i)$$

where $D(n)$ is the cost of the Divide step, $R(n)$ is the cost of the Recombination step, and the last term accounts for the cost of all recursive calls. These observations are enough for these notes; we refer the reader to Chapter 4 in [3] for a deeper and clean discussion about the Divide&Conquer technique and the Master Theorem that provides a mathematical solution to recurrence relations, such as the one above.

We are ready now to specialize the Divide&Conquer technique over the List-Ranking problem. The algorithm we propose is pretty simple and starts by assigning to each item i the value $\texttt{Rank}[i] = 0$ for $i = t$, otherwise $\texttt{Rank}[i] = 1$. Then it executes three main steps:

Divide. We identify a set of items $I = \{i_1, i_2, \ldots, i_h\}$ drawn from the input list $\mathcal{L}$. Set I must be an *independent set*, which means that the successor of each item in I does not belong to I. This condition clearly guarantees that $|I| \leq n/2$, because at most one item out of two consecutive items may be selected. The algorithm will guarantee also that $|I| \geq n/c$, where $c > 2$, in order to make the approach effective.

Conquer. Form the list $\mathcal{L}^* = \mathcal{L} - I$, by pointer-jumping only on the predecessors of the removed items I: namely, for every $\texttt{Succ}[x] \in I$ we set $\texttt{Rank}[x] += \texttt{Rank}[\texttt{Succ}[x]]$ and $\texttt{Succ}[x] = \texttt{Succ}[\texttt{Succ}[x]]$. This way, at any recursive call, $\texttt{Rank}[x]$ accounts for the number of items of the original input list that lie between x and the current $\texttt{Succ}[x]$. Then solve recursively the list-ranking problem over $\mathcal{L}^*$. Notice that $n/2 \leq |\mathcal{L}^*| \leq (1 - 1/c)n$, so that the recursion acts on a list which is a fractional part of $\mathcal{L}$. This is crucial for the efficiency of the recursive calls.



Rank	1	1	0	1	1	1

Rank	1	**2**	0	1	1	**2**

FIGURE 4.4: An example of reduction of a list due to the removal of the items in an Independent Set, here specified by the bold nodes. The new list on the bottom is the one resulting from the removal, the Rank-array is recomputed accordingly to reflect the missing items. Notice that the Rank-values are 1, 0 for the tail, because we assume that $\mathcal{L}$ is the initial list.

Recombine. At this point we can assume that the recursive call has computed correctly the list-ranking of all items in $\mathcal{L}^*$. So, in this phase, we derive the rank of each item $x \in I$ as $\texttt{Rank}[x] = \texttt{Rank}[x] + \texttt{Rank}[\texttt{Succ}[x]]$, by adopting an update rule which reminds the one used in pointer jumping. The correctness of Rank-computation is given by two facts: (i) the independent-set property about I ensures that $\texttt{Succ}[x] \notin I$, thus $\texttt{Succ}[x] \in \mathcal{L}^*$ and so its Rank is available; (ii) by induction, $\texttt{Rank}[\texttt{Succ}[x]]$ accounts for the distance of $\texttt{Succ}[x]$ from the tail of $\mathcal{L}$ and $\texttt{Rank}[x]$ accounts for the number of items between x and $\texttt{Succ}[x]$ in the original input list (as observed in Conquer's step). In fact, the removal of x (because of its selection in I, may occur at any recursive step so that x may be far from the current $\texttt{Succ}[x]$ when considered the original list; this means that it might be the case of $\texttt{Rank}[x] \gg 1$, which the previous summation-step will take into account. As an example, Figure 4.4 depicts the starting situation in which all ranks are 1 except the tail's one, so the update is just $\texttt{Rank}[x] = 1 + \texttt{Rank}[\texttt{Succ}[x]]$. In a general recursive step, $\texttt{Rank}[x] \geq 1$ and so we have to take care of this when updating its value. As a result all items in $\mathcal{L}$ have their Rank-value correctly computed and, thus, induction is preserved and the algorithm may return to its invoking caller.

Figure 4.4 illustrates how an independent set (denoted by bold nodes) is removed from the list $\mathcal{L}$ and how the Succ-links are properly updated. Notice that we are indeed pointer-jumping only on the predecessors of the removed items (namely, the predecessors of the items in I), and that the other items leave untouched their Succ-pointers. It is clear that, if the next recursive step selects $I = \{6\}$, the final list will be constituted by three items $\mathcal{L} = (2, 4, 3)$ whose final ranks are $(6, 3, 0)$, respectively. The Recombination-step will re-insert 6 in $\mathcal{L} = (2, 4, 3)$, just after 4, and compute $\texttt{Rank}[6] = \texttt{Rank}[6] + \texttt{Rank}[3] = 2 + 0 = 2$ because $\texttt{Succ}[6] = 3$ in the current list. Conversely, if one would have not taken into account the fact that item 6 may be far from its current $\texttt{Succ}[6] = 3$, when referred to the original list, and summed 1 it would have made a wrong calculation for $\texttt{Rank}[6]$.

This algorithm makes clear that its I/O-efficiency depends onto the Divide-step. In fact, Conquer-step is recursive and thus can be estimated as $T((1 - \frac{1}{c})n)$ I/Os; Recombine-step executes all re-insertions simultaneously, given that the removed items are not contiguous (by definition of independent set), and can be implemented by Theorem 4.1 in $\widetilde{O}(n/B)$ I/Os.

THEOREM 4.4 *The list-ranking problem formulated over a list $\mathcal{L}$ of length n, can be solved via a Divide&Conquer approach taking $T(n) = I(n) + \widetilde{O}(n/B) + T((1 - \frac{1}{c})n)$ I/Os, where $I(n)$ is the I/O-cost of selecting an independent set from $\mathcal{L}$ of size at least n/c (and, of course, at most $n/2$).*

Deriving a large independent set is trivial if a scan of the list $\mathcal{L}$ is allowed, just pick one every two items. But in our disk-context the list scanning is I/O-inefficient and this is exactly what we want to avoid: otherwise we would have solved the list-ranking problem!

In what follows we will therefore concentrate on the problem of identifying a *large* independent set within the list $\mathcal{L}$. The solution must deploy only local information within the list, in order to avoid the execution of many I/Os. We will propose two solutions: one is simple and randomized, the other one is deterministic and more involved. It is surprising that the latter technique (called *deterministic coin tossing*) has found applications in many other contexts, such as data compression, text similarity, string-equality testing. It is a very general and powerful technique that, definitely, deserves some attention in these notes.

4.3.1 A randomized solution

The algorithmic idea, as anticipated above, is simple: toss a fair coin for each item in $\mathcal{L}$, and then select those items i such that $\texttt{coin}(i) = \texttt{H}$ but $\texttt{coin}(\texttt{Succ}[i]) = \texttt{T}$.[2]

The probability that the item i is selected is $\frac{1}{4}$, because this happens for one configuration (HT) out of the four possible configurations. So the average number of items selected for I is $n/4$. By using sophisticated probabilistic tools, such as Chernoff bounds, it is possible to prove that the number of selected items is strongly concentrated around $n/4$. This means that the algorithm can repeat the coin tossing until $|I| \geq n/c$, for some $c > 4$. The strong concentration guarantees that this repetition is executed a (small) constant number of times.

We finally notice that the check on the values of `coin`, for selecting I's items, can be simulated by Theorem 4.1 via few `Sort` and `Scan` primitives, thus taking $I(n) = \widetilde{O}(n/B)$ I/Os on average. So, by substituting this value in Theorem 4.4, we get the following recurrence relation for the I/O-complexity of the proposed algorithm: $T(n) = \widetilde{O}(n/B) + T(\frac{3n}{4})$. It can be shown by means of the Master Theorem (see Chapter 4 in [3]) that this recurrence relation has solution $\widetilde{O}(n/B)$.

THEOREM 4.5 *The list-ranking problem, formulated over a list $\mathcal{L}$ of length n, can be solved with a randomized algorithm in $\widetilde{O}(n/B)$ I/Os on average.*

4.3.2 Deterministic coin-tossing$^\infty$

The key property of the randomized process was the *locality* of I's construction which allowed to pick an item i by just looking at the results of the coins tossed for i itself and for its successor $\texttt{Succ}[i]$. In this section we try to simulate *deterministically* this process by introducing the so called *deterministic coin-tossing* strategy that, instead of assigning two coin values to each item (i.e. `H` and `T`), it starts by assigning n coin values (hereafter indicated with the integers $0, 1, \ldots, n-1$) and

[2] The algorithm works also in the case that we exchange the role of head (H) and tail (T); but it does not work if we choose the configurations HH or TT. Why?

eventually reduces them to *three* coin values (namely $0, 1, 2$). The final selection process for I will then pick the items whose coin value is minimum among their adjacent items in $\mathcal{L}$. Therefore, here, three possible values and three possible items to be compared, still a constant execution of `Sort` and `Scan` primitives.
The pseudo-code of the algorithm follows.

Initialization. Assign to each item i the value $\texttt{coin}(i) = i - 1$. This way all items take a different coin value, which is smaller than n. We represent these values in $b = \lceil \log n \rceil$ bits, and we denote by $\texttt{bit}_b(i)$ the binary representation of $\texttt{coin}(i)$ using b bits.

Get 6-coin values. Repeat the following steps until $\texttt{coin}(i) < 6$, for all i:

- Compute the position $\pi(i)$ where $\texttt{bit}_b(i)$ and $\texttt{bit}_b(\texttt{Succ}[i])$ differ, and denote by $z(i)$ the bit-value of $\texttt{bit}_b(i)$ at that position.
- Compute the new coin-value for i as $\texttt{coin}(i) = 2\pi(i) + z(i)$ and set the new binary-length representation as $b = \lceil \log b \rceil + 1$.

Get just 3-coin values. For each element i, such that $\texttt{coin}(i) \in \{3, 4, 5\}$, do $\texttt{coin}(i) = \{0, 1, 2\} - \{\texttt{coin}(\texttt{Succ}[i]), \texttt{coin}(\texttt{Pred}[i])\}$.

Select *I*. Pick those items i such that $\texttt{coin}(i)$ is a local minimum, namely it is smaller than $\texttt{coin}(\texttt{Pred}[i])$ and $\texttt{coin}(\texttt{Succ}[i])$.

Let us first discuss the correctness of the algorithm. At the beginning all coin values are distinct, and in the range $\{0, 1, \ldots, n-1\}$. By distinctness, the computation of $\pi(i)$ is sound and $2\pi(i) + z(i) \leq 2(b-1) + 1 = 2b - 1$ since $\texttt{coin}(i)$ was represented with b bits and hence $\pi(i) \leq b - 1$ (counting from 0). Therefore, the new value $\texttt{coin}(i)$ can be represented with $\lceil \log b \rceil + 1$ bits, and thus the update of b is correct too.

A key observation is that the new value of $\texttt{coin}(i)$ is still different of the coin value of its adjacent items in $\mathcal{L}$, namely $\texttt{coin}(\texttt{Succ}[i])$ and $\texttt{coin}(\texttt{Pred}[i])$. We prove it by contradiction. Let us assume that $\texttt{coin}(i) = \texttt{coin}(\texttt{Succ}[i])$ (the other case is similar), then $2\pi(i) + z(i) = 2\pi(\texttt{Succ}[i]) + z(\texttt{Succ}[i])$. Since z denotes a bit value, the two coin-values are equal iff it is both $\pi(i) = \pi(\texttt{Succ}[i])$ and $z(i) = z(\texttt{Succ}[i])$. But if this condition holds then the two bit sequences $\texttt{bit}_b(i)$ and $\texttt{bit}_b(\texttt{Succ}[i])$ cannot differ at bit-position $\pi(i)$.

Easily it follows the correctness of the step which allows to go from 6-coin values to 3-coin values, as well as it is immediate the proof that the selected items form an independent set because of the minimality of $\texttt{coin}(i)$ and distinctness of adjacent coin values.

As far as the I/O-complexity is concerned, we start by introducing the function $\log^* n$ defined as $\min\{j \mid \log^{(j)} n \leq 1\}$, where $\log^{(j)} n$ is the repeated application of the logarithm function for j times to n. As an example[3] take $n = 16$ and compute $\log^{(0)} 16 = 16, \log^{(1)} 16 = 4, \log^{(2)} 16 = 2, \log^{(3)} 16 = 1$; thus $\log^* 16 = 3$. It is not difficult to convince yourselves that $\log^* n$ grows very much slowly, and indeed its value is 5 for $n = 2^{65536}$.

In order to estimate the I/O-complexity, we need to bound the number of iterations needed by the algorithm to reduce the coin-values to $\{0, 1, \ldots, 5\}$. This number is $\log^* n$, because at each step the reduction in the number of possible coin-values is logarithmic ($b = \lceil \log b \rceil + 1$). All single steps can be implemented by Theorem 4.1 via few `Sort` and `Scan` primitives, thus taking $\widetilde{O}(n/B)$ I/Os. So the construction of the independent set takes $I(n) = \widetilde{O}((n/B) \log^* n) = \widetilde{O}(n/B)$ I/Os, by definition of $\widetilde{O}()$. The size of I can be lower bounded as $|I| \geq n/4$ because the distance between two consecutive

[3] Recall that logarithms are in base 2 in these lectures.

selected items (local minima) is maximum when the coin-values form a *bitonic sequence* of the form $\ldots, 0, 1, 2, 1, 0, \ldots$.

By substituting this value in Theorem 4.4 we get the same recurrence relation of the randomized algorithm, with the exception that now the I/O-bound is worst case and deterministic: $T(n) = \widetilde{O}(n/B) + T(\frac{3n}{4})$.

THEOREM 4.6 *The list-ranking problem, formulated over a list $\mathcal{L}$ of length n, can be solved with a deterministic algorithm in $\widetilde{O}(n/B)$ worst-case I/Os.*

A comment is in order to conclude this chapter. The logarithmic term hidden in the $\widetilde{O}()$-notation has the form $(\log^* n)(\log_{M/B} n)$, which can be safely assumed to be smaller than 15 because, in practice, $\log_{M/B} n \leq 3$ and $\log^* n \leq 5$ for n up to 1 petabyte.

References

[1] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, Jeffrey Scott Vitter. External-Memory Graph Algorithms. *ACM-SIAM Symposium on Algorithms (SODA)*, 139-149, 1995.

[2] Joseph JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[3] Tomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[4] Jeffrey Scott Vitter. Faster methods for random sampling. *ACM Computing Surveys*, 27(7):703–718, 1984.

5

Sorting Atomic Items

This lecture will focus on the very-well known problem of sorting a set of *atomic* items, the case of *variable-length* items (aka strings) will be addressed in the following chapter. Atomic means that they occupy a constant-fixed number of memory cells, typically they are integers or reals represented with a fixed number of bytes, say 4 (32 bits) or 8 (64 bits) bytes each.

> **The sorting problem.** *Given a sequence of n atomic items $S[1, n]$ and a total ordering $\leq$ between each pair of them, sort S in increasing order.*

We will consider two complemental sorting paradigms: the *merge-based* paradigm, which underlies the design of Mergesort, and the *distribution-based* paradigm which underlies the design of Quicksort. We will adapt them to work in the 2-level memory model, analyze their I/O-complexities and propose some useful tools that can allow to speed up their execution in practice, such as the Snow Plow technique and Data compression. We will also demonstrate that these disk-based adaptations are *I/O-optimal* by proving a sophisticated lower-bound on the number of I/Os any external-memory sorter must execute to produce an ordered sequence. In this context we will relate the Sorting problem with the so called *Permuting* problem, typically neglected when dealing with sorting in the RAM model.

> **The permuting problem.** *Given a sequence of n atomic items $S[1, n]$ and a permutation $\pi[1, n]$ of the integers $\{1, 2, \ldots, n\}$, permute S according to π thus obtaining the new sequence $S[\pi[1]], S[\pi[2]], \ldots, S[\pi[n]]$.*

Clearly Sorting includes Permuting as a sub-task: to order the sequence S we need to determine its sorted permutation and then implement it (possibly these two phases are intricately intermingled). So Sorting should be more difficult than Permuting. And indeed in the RAM model we know that sorting n atomic items takes $\Theta(n \log n)$ time (via Mergesort or Heapsort) whereas permuting them takes $\Theta(n)$ time. The latter time bound can be obtained by just moving one item at a time according to what indicates the array π. Surprisingly we will show that this *complexity gap* does not exist in the disk model, because these two problems exhibit the same I/O-complexity under some reasonable conditions on the input and model parameters n, M, B. This elegant and deep result was obtained by Aggarwal and Vitter in 1998 [1], and it is surely the result that spurred the huge amount of algorithmic literature on the I/O-subject. Philosophically speaking, AV's result formally proves the intuition that *moving* items in the disk is the real *bottleneck*, rather than *finding* the sorted permutation. And indeed researchers and software engineers typically speak about the *I/O-bottleneck* to characterize this issue in their (slow) algorithms.

We will conclude this lecture by briefly mentioning at two solutions for the problem of sorting items on D-disks: the disk-striping technique, which is at the base of RAID systems and turns any efficient/optimal 1-disk algorithm into an efficient D-disk algorithm (typically loosing its optimality, if any), and the Greed-sort algorithm, which is specifically tailored for the sorting problem on D-disks and achieves I/O-optimality.

5.1 The merge-based sorting paradigm

We recall the main features of the external-memory model introduced in Chapter 1: it consists of an internal memory of size M and allows blocked-access to disk by reading/writing B items at once.

Algorithm 5.1 The binary merge-sort: MergeSort(S, i, j)

```
1: if ($i < j$) then
2:     $m = (i + j)/2$;
3:     MergeSort($S, i, m - 1$);
4:     MergeSort($S, m, j$);
5:     Merge($S, i, m, j$);
6: end if
```

Mergesort is based on the Divide&Conquer paradigm. Step 1 checks if the array to be sorted consists of at least two items, otherwise it is already ordered and nothing has to be done. If items are more than two, it splits the input array S into two halves, and then recurses on each part. As recursion ends, the two halves $S[i, m - 1]$ and $S[m, j]$ are ordered so that Step 5 fuses them in $S[i, j]$ by invoking procedure Merge. This merging step needs an auxiliary array of size n, so that MergeSort is not an *in-place* sorting algorithm (unlike Heapsort and Quicksort) but needs $O(n)$ extra working space. Given that at each recursive call we halve the size of the input array to be sorted, the total number of recursive calls is $O(\log n)$. The Merge-procedure can be implemented in $O(j-i+1)$ time by using two pointers, say x and y, that start at the beginning of the two halves $S[i, m - 1]$ and $S[m, j]$. Then $S[x]$ is compared with $S[y]$, the smaller is written out in the fused sequence, and its pointer is advanced. Given that each comparison advances one pointer, the total number of steps is bounded above by the total number of pointer's advancements, which is upper bounded by the length of $S[i, j]$. So the time complexity of MergeSort($S, 1, n$) can be modeled via the recurrence relation $T(n) = 2T(n/2) + O(n) = O(n \log n)$, as well known from any basic algorithm course.[1]

Let us assume now that $n > M$, so that S must be stored on disk and I/Os become the most important resource to be analyzed. In practice every I/O takes 5ms on average, so one could think that every item comparison takes one I/O and thus one could estimate the running time of Mergesort

[1] In all our lectures when the base of the logarithm is not indicated, it means 2.

on a massive S as: 5ms × $\Theta(n \log n)$. If n is of the order of few Gigabytes, say $n \approx 2^{30}$ which is actually not much massive for the current memory-size of commodity PCs, the previous time estimate would be of about $5 \times 2^{30} \times 30 > 10^8$ms, namely more than 1 day of computation. However, if we run Mergesort on a commodity PC it completes in few hours. This is not surprising because the previous evaluation totally neglected the existence of the internal memory, of size M, and the sequential pattern of memory-accesses induced by Mergesort. Let us therefore analyze the Mergesort algorithm in a more precise way within the disk model.

First of all we notice that $O(z/B)$ I/Os is the cost of merging two ordered sequences of z items in total. This holds if $M \geq 2B$, because the Merge-procedure in Algorithm 5.1 really keeps in internal memory the 2 pages that contain the two pointers scanning $S[i, j]$ where $z = j - i + 1$. Every time a pointer advances into another disk page, an I/O-fault occurs, the page is fetched in internal memory, and the fusion continues. Given that S is stored contiguously on disk, $S[i, j]$ occupies $O(z/B)$ pages and this is the I/O-bound for merging two sub-sequences of total size z. Similarly, the I/O-cost for writing the merged sequence is $O(z/B)$ because it occurs sequentially from the smallest to the largest item of $S[i, j]$ by using an auxiliary array. As a result the recurrent relation for the I/O-complexity of Mergesort can be written as $T(n) = 2T(n/2) + O(n/B) = O(\frac{n}{B} \log n)$ I/Os.

But this formula does not explain completely the good behavior of Mergesort in practice, because it does not account for the memory hierarchy yet. In fact as Mergesort recursively splits the sequence S, smaller and smaller sub-sequences are generated that have to be sorted. So when a subsequence of length z fits in internal memory, namely $z = O(M)$, then it is entirely cached by the underlying operating system using $O(z/B)$ I/Os and thus the subsequent sorting steps do not incur in any I/Os. The net result of this simple observation is that the I/O-cost of sorting a sub-sequence of $z = O(M)$ items is no longer $\Theta(\frac{z}{B} \log z)$, as accounted for in the previous recurrence relation, but it is $O(z/B)$ I/Os which accounts only the cost of loading the subsequence in internal memory. This saving applies to all S's subsequences of size $\Theta(M)$ on which Mergesort is recursively run, which are $\Theta(n/M)$ in total. So the overall saving is $\Theta(\frac{n}{B} \log M)$, which leads us to re-formulate the Mergesort's complexity as $\Theta(\frac{n}{B} \log \frac{n}{M})$ I/Os. This bound is particularly interesting because relates the I/O-complexity of Mergesort not only to the disk-page size B but also to the internal-memory size M, and thus to the *caching* available at the sorter. Moreover this bounds suggests three immediate optimizations to the classic pseudocode of Algorithm 5.1 that we discuss below.

5.1.1 Stopping recursion

The first optimization consists of introducing a threshold on the subsequence size, say $j - i < cM$, which triggers the stop of the recursion, the fetching of that subsequence entirely in internal-memory, and the application of an internal-memory sorter on this sub-sequence (see Figure 5.1). The value of the parameter c depends on the space-occupancy of the sorter, which must be guaranteed to work entirely in internal memory. As an example, c is 1 for in-place sorters such as Insertionsort and Heapsort, it is much close to 1 for Quicksort (because of its recursion), and it is less than 0.5 for Mergesort (because of the extra-array used by Merge). As a result, we should write cM instead of M into the I/O-bound above, because recursion is stopped at cM items: thus obtaining $\Theta(\frac{n}{B} \log \frac{n}{cM})$. This substitution is useless when dealing with asymptotic analysis, given that c is a constant, but it is important when considering the real performance of algorithms. In this setting it is desirable to make c as closer as possible to 1, in order to reduce the logarithmic factor in the I/O-complexity thus preferring in-place sorters such as Heapsort or Quicksort. We remark that Insertionsort could also be a good choice (and indeed it is) whenever M is small, as it occurs when considering the sorting of items over the 2-levels: L1 and L2 caches, and the internal memory. In this case M would be few Megabytes.

5.1.2 Snow Plow

Looking at the I/O-complexity of mergesort, i.e. $\Theta(\frac{n}{B} \log \frac{n}{M})$, is clear that the larger is M the smaller is the number of merge-passes over the data. These passes are clearly the bottleneck to the efficient execution of the algorithm especially in the presence of disks with low bandwidth. In order to circumvent this problem we can either buy a larger memory, or try to deploy as much as possible the one we have available. As algorithm engineer we opt for the second possibility and thus propose two techniques that can be combined together in order to enlarge (virtually) M.

The first technique is based on data compression and builds upon the observation that the runs are increasingly sorted. So, instead of representing items via a fixed-length coding (e.g. 4 or 8 bytes), we can use *integer compression* techniques that squeeze those items in fewer bits thus allowing us to pack more of them in internal memory. A following lecture will describe in detail several approaches to this problem (see Chapter 11), here we content ourselves mentioning the names of some of these approaches: γ-code, δ-code, Rice/Golomb-coding, etc. etc.. In addition, since the smaller is an integer the fewer bits are used for its encoding, we can enforce the presence of small integers in the sorted runs by encoding not just their absolute value but the *difference* between one integer and the previous one in the sorted run (the so called *delta*-coding). This difference is surely non negative (equals zero if the run contains equal items), and smaller than the item to be encoded. This is the typical approach to the encoding of integer sequences used in modern search engines, that we will discuss in a following lecture (see Chapter 11).

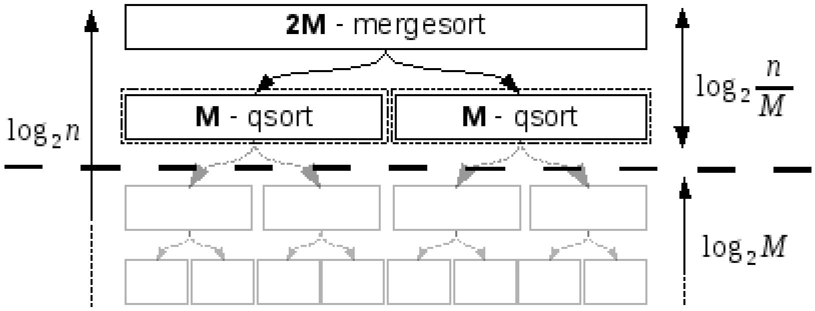


FIGURE 5.1: When a run fits in the internal memory of size M, we apply `qsort` over its items. In gray we depict the recursive calls that are executed in internal memory, and thus do not elicit I/Os. Above there are the calls based on classic Mergesort, only the call on $2M$ items is shown.

The second technique is based on an elegant idea, called the `Snow Plow` and due to D. Knuth [3], that allows to *virtually* increase the memory size of a factor 2 *on average*. This technique scans the input sequence S and generates sorted runs whose length has variable size longer than M and $2M$ on average. Its use needs to change the sorting scheme because it first creates these sorted runs, of variable length, and then applies repeatedly over the sorted runs the MERGE-procedure. Although runs will have different lengths, the MERGE will operate as usual requiring an optimal number of I/Os for their merging. Hence $O(n/B)$ I/Os will suffice to halve the number of runs, and thus a total of $O(\frac{n}{B} \log \frac{n}{2M})$ I/Os will be used on average to produce the totally ordered sequence. This corresponds to a saving of 1 pass over the data, which is non negligible if the sequence S is very long.

For ease of description, let us assume that items are transferred one at a time from disk to memory, instead that block-wise. Eventually, since the algorithm scans the input items it will be apparent that the number of I/Os required by this process is linear in their number (and thus optimal). The algorithm proceeds in phases, each phase generates a sorted run (see Figure 5.2 for an illustrative

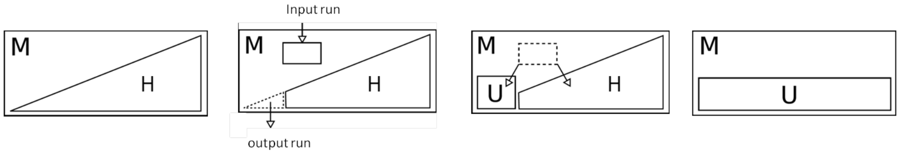


FIGURE 5.2: An illustration of four steps of a phase in Snow Plow. The leftmost picture shows the starting step in which $\mathcal{U}$ is heapified, then a picture shows the output of the minimum element in $\mathcal{H}$, hence the two possible cases for the insertion of the new item, and finally the stopping condition in which $\mathcal{H}$ is empty and $\mathcal{U}$ fills entirely the internal memory.

example). A phase starts with the internal-memory filled of M (unsorted) items, stored in a heap data structure called $\mathcal{H}$. Since the array-based implementation of heaps requires no additional space, in addition to the indexed items, we can fit in $\mathcal{H}$ as many items as we have memory cells available. The phase scans the input sequence S (which is unsorted) and at each step, it writes to the output the minimum item within $\mathcal{H}$, say `min`, and loads in memory the next item from S, say `next`. Since we want to generate a sorted output, we cannot store `next` in $\mathcal{H}$ if `next` < `min`, because it will be the new heap-minimum and thus it will be written out at the next step thus destroying the property of ordered run. So in that case `next` must be stored in an auxiliary array, called $\mathcal{U}$, which stays unsorted. Of course the total size of $\mathcal{H}$ and $\mathcal{U}$ is M over the whole execution of a phase. A phase stops whenever $\mathcal{H}$ is empty and thus $\mathcal{U}$ consists of M unsorted items, and the next phase can thus start (storing those items in a new heap $\mathcal{H}$ and emptying $\mathcal{U}$). Two observations are in order: (i) during the phase execution, the minimum of $\mathcal{H}$ is non decreasing and so it is non-decreasing also the output run, (ii) the items in $\mathcal{H}$ at the beginning of the phase will be eventually written to output which thus is longer than M. Observation (i) implies the correctness, observation (ii) implies that this approach is not less efficient than the classic Mergesort.

Algorithm 5.2 A phase of the Snow-Plow technique

```
Require: $\mathcal{U}$ is an unsorted array of $M$ items
1:  $\mathcal{H}$ = build a min-heap over $\mathcal{U}$'s items;
2:  Set $\mathcal{U} = \emptyset$;
3:  while ($\mathcal{H} \neq \emptyset$) do
4:      min = Extract minimum from $\mathcal{H}$;
5:      Write min to the output run;
6:      next = Read the next item from the input sequence;
7:      if (next < min) then
8:          write next in $\mathcal{U}$;
9:      else
10:         insert next in $\mathcal{H}$;
11:     end if
12: end while
```

Actually it is more efficient than that on average. Suppose that a phase reads τ items in total from S. By the while-guard in Step 3 and our comments above, we can derive that a phase ends when $\mathcal{H}$ is empty and $|\mathcal{U}| = M$. We know that the read items go in part in $\mathcal{H}$ and in part in $\mathcal{U}$. But since items are added to $\mathcal{U}$ and never removed during a phase, M of the τ items end-up in $\mathcal{U}$.

Consequently $(\tau - M)$ items are inserted in $\mathcal{H}$ and eventually written to the output (sorted) run. So the length of the sorted run at the end of the phase is $M + (\tau - M) = \tau$, where the first addendum accounts for the items in $\mathcal{H}$ at the beginning of a phase, whereas the second addendum accounts for the items read from S and inserted in $\mathcal{H}$ during the phase. The key issue now is to compute the average of τ. This is easy if we assume a random distribution of the input items. In this case we have probability $1/2$ that `next` is smaller than `min`, and thus we have equal probability that a read item is inserted either in $\mathcal{H}$ or in $\mathcal{U}$. Overall it follows that $\tau/2$ items go to $\mathcal{H}$ and $\tau/2$ items go to $\mathcal{U}$. But we already know that the items inserted in $\mathcal{U}$ are M, so we can set $M = \tau/2$ and thus we get $\tau = 2M$.

***FACT* 5.1** *Snow-Plow builds $O(n/M)$ sorted runs, each longer than M and actually of length $2M$ on average. Using Snow-Plow for the formation of sorted runs in a Merge-based sorting scheme, this achieves an I/O-complexity of $O(\frac{n}{B}\log_2\frac{n}{2M})$ on average.*

5.1.3 From binary to multi-way Mergesort

Previous optimizations deployed the internal-memory size M to reduce the number of recursion levels by increasing the size of the initial (sorted) runs. But then the merging was *binary* in that it fused two input runs at a time. This binary-merge impacted onto the base 2 of the logarithm of the I/O-complexity of Mergesort. Here we wish to increase that base to a much larger value, and in order to get this goal we need to deploy the memory M also in the merging phase by enlarging the number of runs that are fused at a time. In fact the merge of 2 runs uses only 3 blocks of the internal memory: 2 blocks are used to cache the current disk pages that contain the compared items, namely $S[x]$ and $S[y]$ from the notation above, and 1 block is used to cache the output items which are flushed when the block is full (so to allow a block-wise writing to disk of the merged run). But the internal memory contains a much larger number of blocks, i.e. $M/B \gg 3$, which remain unused over the whole merging process. The third optimization we propose, therefore consists of deploying all those blocks by designing a k-way merging scheme that fuses k runs at a time, with $k \gg 2$. Let us set $k = (M/B) - 1$, so that k blocks are available to read block-wise k input runs, and 1 block is reserved for a block-wise writing of the merged run to disk. This scheme poses a challenging merging problem because at each step we have to select the minimum among k candidates items and this cannot be obviously done brute-forcedly by iterating among them. We need a smarter solution that again hinges onto the use of a min-heap data structure, which contains k pairs (one per input run) each consisting of two components: one denoting an item and the other denoting the origin run. Initially the items are the minimum items of the k runs, and so the pairs have the form $\langle R_i[1], i\rangle$, where R_i denotes the ith input run and $i = 1, 2, \ldots, k$. At each step, we extract the pair containing the current smallest item in $\mathcal{H}$ (given by the first component of its pairs), write that item to output and insert in the heap the next item in its origin run. As an example, if the minimum pair is $\langle R_m[x], m\rangle$ then we write in output $R_m[x]$ and insert in $\mathcal{H}$ the new pair $\langle R_m[x+1], m\rangle$, provided that the mth run is not exhausted, in which case no pair replaces the extracted one. In the case that the disk page containing $R_m[x+1]$ is not cached in internal memory, an I/O-fault occurs and that page is fetched, thus guaranteeing that the next B reads from run R_m will not elicit any further I/O. It should be clear that this merging process takes $O(\log_2 k)$ time per item, and again $O(z/B)$ I/Os to merge k runs of total length z.

As a result the merging-scheme recalls a k-way tree with $O(n/M)$ leaves (runs) which can have been formed using any of the optimizations above (possibly via Snow Plow). Hence the total number of merging levels is now $O(\log_{M/B}\frac{n}{M})$ for a total volume of I/Os equal to $O(\frac{n}{B}\log_{M/B}\frac{n}{M})$. We observe that sometime we will also write the formula as $O(\frac{n}{B}\log_{M/B}\frac{n}{B})$, as it typically occurs in the literature, because $\log_{M/B} M$ can be written as $\log_{M/B}(B \times (M/B)) = (\log_{M/B} B) + 1 = \Theta(\log_{M/B} B)$. This makes no difference asymptotically given that $\log_{M/B} \frac{n}{M} = \Theta(\log_{M/B} \frac{n}{B})$.

THEOREM 5.1 *Multi-way Mergesort takes $O(\frac{n}{B} \log_{M/B} \frac{n}{M})$ I/Os and $O(n \log n)$ comparisons/-time to sort n atomic items in a two-level memory model in which the internal memory has size M and the disk page has size B. The use of Snow-Plow or integer compressors would virtually increase the value of M with a twofold advantage in the final I/O-complexity, because M occurs twice in the I/O-bound.*

In practice the number of merging levels will be very small: assuming a block size $B = 4$KB and a memory size $M = 4$GB, we get $M/B = 2^{32}/2^{12} = 2^{20}$ so that the number of passes is $1/20$th smaller than the ones needed by binary Mergesort. Probably more interesting is to observe that one pass is able to sort $n = M$ items, but two passes are able to sort M^2/B items, since we can merge M/B-runs each of size M. It goes without saying that in practice the internal-memory space which can be dedicated to sorting is smaller than the *physical* memory available (typically MBs versus GBs). Nevertheless it is evident that M^2/B is of the order of Terabytes already for $M = 128$MB and $B = 4$KB.

5.2 Lower bounds

At the beginning of this lecture we commented on the relation existing between the Sorting and the Permuting problems, concluding that the former one is more difficult than the latter in the RAM model. The gap in time complexity is given by a logarithmic factor. The question we address in this section is whether this gap does exist also when measuring I/Os. Surprisingly enough we will show that Sorting is *equivalent* to Permuting in terms of I/O-volume. This result is amazing because it can be read as saying that the I/O-cost for sorting is not in the *computation* of the sorted permutation but rather the *movement* of the data on the disk to realize it. This is the so called *I/O-bottleneck* that has in this result the mathematical proof and quantification.

Before digging into the proof of this lower bound, let us briefly show how a sorter can be used to permute a sequence of items $S[1, n]$ in accordance to a given permutation $\pi[1, n]$. This will allow us to derive an upper bound to the number of I/Os which suffice to solve the Permuting problem on any $\langle S, \pi \rangle$. Recall that this means to generate the sequence $S[\pi[1]], S[\pi[2]], \ldots, S[\pi[n]]$. In the RAM model we can jump among S's items according to permutation π and create the new sequence $S[\pi[i]]$, for $i = 1, 2, \ldots, n$, thus taking $O(n)$ optimal time. On disk we have actually two different algorithms which induce two incomparable I/O-bounds. The first algorithm consists of mimicking what is done in RAM, paying one I/O per moved item and thus taking $O(n)$ I/Os. The second algorithm consists of generating a proper set of tuples and then sort them. Precisely, the algorithm creates the sequence $\mathcal{P}$ of pairs $\langle i, \pi[i] \rangle$ where the first component indicates the position i where the item $S[\pi[i]]$ must be stored. Then it sorts these pairs according to the π-component, and via a parallel scan of S and $\mathcal{P}$ substitutes $\pi[i]$ with the item $S[\pi[i]]$, thus creating the new pairs $\langle i, S[\pi[i]] \rangle$. Finally another sort is executed according to the first component of these pairs, thus obtaining a sequence of items correctly permuted. The algorithm uses two scan of the data and two sorts, so it needs $O(\frac{n}{B} \log_{M/B} \frac{n}{M})$ I/Os.

THEOREM 5.2 *Permuting n items takes $O(\min\{n, \frac{n}{B} \log_{M/B} \frac{n}{M}\})$ I/Os in a two-level memory model in which the internal memory has size M and the disk page has size B.*

In what follows we will show that this algorithm, in its simplicity, is I/O-optimal. The two upper-bounds for Sorting and Permuting equal each other whenever $n = \Omega(\frac{n}{B} \log_{M/B} \frac{n}{M})$. This occurs when

$B > \log_{M/B} \frac{n}{M}$ that holds always in practice because that logarithm term is about 2 or 3 for values of n up to many Terabytes. So programmers should not be afraid to find sophisticated strategies for moving their data in the presence of a permutation, just sort them, you cannot do better!

	time complexity (RAM model)	I/O complexity (two-level memory model)
Permuting	$O(n)$	$O(\min\{n, \frac{n}{B} \log_{M/B} \frac{n}{M}\})$
Sorting	$O(n \log_2 n)$	$O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{M})$

TABLE 5.1 Time and I/O complexities of the Permuting and Sorting problems in a two-level memory model in which M is the internal-memory size, B is the disk-page size, and $D = 1$ is the number of available disks. The case of multi-disks presents the multiplicative term n/D in place of n.

5.2.1 A lower-bound for Sorting

There are some subtle issues here that we wish to do not investigate too much, so we hereafter give only the intuition which underlies the lower-bounds for both Sorting and Permuting.[2] We start by resorting the comparison-tree technique for proving comparison-based lower bounds in the RAM model. An algorithm corresponds to a family of such trees, one per input size (so infinite in number). Every node is a comparison between two items. The comparison has two possible results, so the fan-out of each internal node is two and the tree is binary. Each leaf of the tree corresponds to a solution of the underlying problem to be solved: so in the case of sorting, we have one leaf per permutation of the input. Every root-to-leaf path in the comparison-tree corresponds to a computation, so the longest path corresponds to the worst-case number of comparisons executed by the algorithm. In order to derive a lower bound, it is therefore enough to compute the depth of the shallowest binary tree having that number of leaves. The shallowest binary tree with ℓ leaves is the (quasi-)perfectly balanced tree, for which the height h is such that $2^h \geq \ell$. Hence $h \geq \log_2 \ell$. In the case of sorting $\ell = n!$ so the classic lower bound $h = \Omega(n \log_2 n)$ is easily derived by applying logarithms at both sides of the equation and using the Stirling's approximation for the factorial.

In the two-level memory model the use of comparison-trees is more sophisticated. Here we wish to account for I/Os, and exploit the fact that the information available in the internal memory can be used for free. As a result every node corresponds to one I/O, the number of leaves equals still to $n!$, but the fan-out of each internal node equals to the *number of comparison-results* that this single I/O can generate among the items it reads from disk (i.e. B) and the items available in internal memory (i.e. $M - B$). These B items can be distributed in at most $\binom{M}{B}$ ways among the other $M - B$ items present in internal memory, so one I/O can generate no more than $\binom{M}{B}$ different results for those comparisons. But this is an incomplete answer because we are not considering the permutations among those items! However, some of these permutations have been already counted by some previous I/O, and thus we have not to recount them. These permutations are the ones concerning with items that have already passed through internal memory, and thus have been fetched by some previous I/O. So we have to count only the permutations among the *new* items, namely the ones

[2] There are two assumptions that are typically introduced in those arguments. One concerns with *item indivisibility*, so items cannot be broken up into pieces (hence hashing is not allowed!), and the other concerns with the possibility to *only move items* and not create/destroy/copy them, which actually implies that exactly one copy of each item does exist during their sorting or permuting.

that have never been considered by a previous I/O. We have n/B input pages, and thus n/B I/Os accessing new items. So these I/Os generate $\binom{M}{B}(B!)$ results by comparing those new B items with the $M - B$ ones in internal memory.

Let us now consider a computation with t I/Os, and thus a path in the comparison-tree with t nodes. n/B of those nodes must access the input items, which must be surely read to generate the final permutation. The other $t - \frac{n}{B}$ nodes read pages containing already processed items. Any root-to-leaf path has this form, so we can look at the comparison tree as having the new-I/Os at the top and the other nodes at its bottom. Hence if the tree has depth t, its number of leaves is at least $\binom{M}{B}^t \times (B!)^{n/B}$. By imposing that this number is $\geq n!$, and applying logarithms to both members, we derive that $t = \Omega(\frac{n}{B} \log_{M/B} \frac{n}{M})$. It is not difficult to extend this argument to the case of D disks thus obtaining the following.

THEOREM 5.3 *In a two-level memory model with internal memory of size M, disk-page size B and D disks, a comparison-based sorting algorithm must execute* $\Omega(\frac{n}{DB} \log_{M/B} \frac{n}{DB})$ *I/Os.*

It is interesting to observe that the number of available disks D does not appear in the denominator of the base of the logarithm, although it appears in the denominator of all other terms. If this would be the case, instead, D would somewhat penalize the sorting algorithms because it would reduce the logarithm's base. In the light of Theorem 5.1, multi-way Mergesort is I/O and time optimal on one disk, so D linearly boosts its performance thus having more disks is *linearly* advantageous (at least from a theoretical point of view). But Mergesort is no longer optimal on multi-disks because the simultaneous merging of $k > 2$ runs, should take $O(n/DB)$ I/Os in order to be optimal. This means that the algorithm should be able to fetch D pages per I/O, hence one per disk. This cannot be guaranteed, at every step, by the current merging-scheme because whichever is the distribution of the k runs among the D disks, and even if we know which are the next DB items to be loaded in the heap $\mathcal{H}$, it could be the case that more than B of these items reside on the same disk thus requiring more than one I/O from that disk, hence preventing the parallelism in the read operation.

In the following Section 5.4 we will address this issue by proposing the *disk striping* technique, that comes close to the I/O-optimal bound via a simple data layout on disks, and the *Greedsort* algorithm that achieves full optimality by devising an elegant and sophisticated merging scheme.

5.2.2 A lower-bound for Permuting

Let us assume that at any time the memory of our model, hence the internal memory of size M and the unbounded disk, contains a permutation of the input items possibly interspersed by empty cells. No more than n blocks will be non empty during the execution of the algorithm, because n steps (and thus I/Os) is an obvious upper bound to the I/O-complexity of Permuting (obtained by mimicking on disk the Permuting algorithm for the RAM model). We denote by P_t the number of permutations generated by an algorithm with t I/Os, where $t \leq n$ and $P_0 = 1$ since at the beginning we have the input order as initial permutation. In what follows we estimate P_t and then set $P_t \geq n!$ in order to derive the minimum number of steps t needed to realize any possible permutation given in input. Permuting is different from Sorting because the permutation to be realized is provided in input, and thus we do not need any computation. So in this case we distinguish three types of I/Os, which contribute differently to the number of generated permutations:

Write I/O: This may increase P_t by a factor $O(n)$ because we have at most $n+1$ possible ways to write the output page among the at most n not-empty pages available on disk. Any written page is “touched”, and they are no more than n at any instant of the permuting process.

Read I/O on an untouched page: If the page was an input page never read before, the read operation imposes to account for the permutations among the read items, hence $B!$ in number, and to account also for the permutations that these B items can realize by distributing them among the $M - B$ items present in internal memory (similarly as done for Sorting). So this read I/O can increase P_t by a factor $O(\binom{M}{B}(B!))$. The number of input (hence "untouched") pages is n/B. After a read I/O, they become "touched".

Read I/O on a touched page: If the page was already read or written, we already accounted in P_t for the permutations among its items, so this read I/O can increase P_t only by a factor $O(\binom{M}{B})$ due to the shuffling of the B read items with the $M - B$ ones present in internal memory. The number of touched pages is at most n.

If t_r is the number of reads and t_w is the number of writes executed by a Permuting algorithm, where $t = t_r + t_w$, then we can bound P_t as follows (here big-Oh have been dropped to ease the reading of the formulas):

$$P_t \le (\frac{n}{B}\binom{M}{B}(B!))^{n/B} \times (n\binom{M}{B})^{t_r-n/B} \times n^{t_w} \le (n\binom{M}{B})^t(B!)^{\frac{n}{B}}$$

In order to generate every possible permutation of the n input items, we need that $P_t \geq n!$. We can thus derive a lower bound on t by imposing that $n! \le (n\binom{M}{B})^t\ (B!)^{\frac{n}{B}}$ and resolving with respect to t:

$$t = \Omega(\frac{n\log\frac{n}{B}}{B\log\frac{M}{B} + \log n})$$

We distinguish two cases. If $B\log\frac{M}{B} \le \log n$, then the above equation becomes $t = \Omega(\frac{n\log\frac{n}{B}}{\log n}) = \Omega(n)$; otherwise it is $t = \Omega(\frac{n\log\frac{n}{B}}{B\log\frac{M}{B}}) = \Omega(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{M})$. As for sorting, it is not difficult to extend this proof to the case of D disks.

THEOREM 5.4 *In a two-level memory model with internal memory of size M, disk-page size B and D disks, permuting n items needs $\Omega(\min\{\frac{n}{D}, \frac{n}{DB}\log_{M/B}\frac{n}{DB}\})$ I/Os.*

Theorems 5.2–5.4 prove that the I/O-bounds provided in Table 5.1 for the Sorting and Permuting problems are optimal. Comparing these bounds we notice that they are asymptotically different whenever $B\log\frac{M}{B} < \log n$. Given the current values for B and M, respectively few KBs and few GBs, this inequality holds if $n = \Omega(2^B)$ and hence when n is much more than Yottabytes ($= 2^{80}$). This is indeed an unreasonable situation to deal with one CPU and few disks. Probably in this context it would be more reasonable to use a *cloud* of PCs, and thus analyze the proposed algorithms via a *distributed* model of computation which takes into account many CPUs and more-than-2 memory levels. It is therefore not surprising that researchers typically assume `Sorting = Permuting` in the I/O-setting.

5.3 The distribution-based sorting paradigm

Like Mergesort, Quicksort is based on the divide&conquer paradigm, so it proceeds by dividing the array to be sorted into two pieces which are then sorted recursively. But unlike Mergesort, Quicksort does not explicitly allocate *extra*-working space, its *combine*-step is absent and its *divide*-step is sophisticated and impacts onto the overall efficiency of this sorting algorithm. Algorithm 5.3

Algorithm 5.3 The binary quick-sort: QuickSort(S, i, j)

```
1: if ($i < j$) then
2:     $r$ = pick the position of a "good pivot";
3:     swap $S[r]$ with $S[i]$;
4:     $p$ = Partition($S, i, j$);
5:     QuickSort($S, i, p - 1$);
6:     QuickSort($S, p + 1, j$);
7: end if
```

reports the pseudocode of Quicksort, this will be used to comment on its complexity and argue for some optimizations or tricky issues which arise when implementing it over hierarchical memories.

The key idea is to partition the input array $S[i, j]$ in two pieces such that one contains items which are *smaller (or equal)* than the items contained in the latter piece. This partition is order preserving because no subsequent steps are necessary to recombine the ordered pieces after the two recursive calls. Partitioning is typically obtained by selecting one input item as a *pivot*, and by distributing all the other input items into two sub-arrays according to whether they are smaller/greater than the pivot. Items equal to the pivot can be stored anywhere. In the pseudocode the pivot is forced to occur in the first position $S[i]$ of the array to be sorted (steps 2–3): this is obtained by swapping the real pivot $S[r]$ with $S[i]$ before that procedure Partition(S, i, j) is invoked. We notice that step 2 does not detail the selection of the pivot, because this will be the topic of a subsequent section.

There are two issues for achieving efficiency in the execution of Quicksort: one concerns with the implementation of Partition(S, i, j), and the other one with the ratio between the size of the two formed pieces because the more *balanced* they are, the more Quicksort comes closer to Mergesort and thus to the optimal time complexity of $O(n \log n)$. In the case of a totally unbalanced partition, in which one piece is possibly empty (i.e. $p = i$ or $p = j$), the time complexity of Quicksort is $O(n^2)$, thus recalling in its cost the Insertion sort. Let us comment these two issues in detail in the following subsections.

5.3.1 From two- to three-way partitioning

The goal of Partition(S, i, j) is to divide the input array into two pieces, one contains items which are smaller than the pivot, and the other contains items which are larger than the pivot. Items equal to the pivot can be arbitrarily distributed among the two pieces. The input array is therefore permuted so that the smaller items are located before the pivot, which in turn precedes the larger items. At the end of Partition(S, i, j), the pivot is located at $S[p]$, the smaller items are stored in $S[i, p - 1]$, the larger items are stored in $S[p + 1, j]$. This partition can be implemented in many ways, taking $O(n)$ optimal time, but each of them offers a different cache usage and thus different performance in practice. We present below a tricky algorithm which actually implements a *three-way* distribution and takes into account the presence of items equal to the pivot. They are detected and stored aside in a "special" sub-array which is located between the two smaller/larger pieces.

It is clear that the central sub-array, which contains items equal to the pivot, can be discarded from the subsequent recursive calls, similarly as we discard the pivot. This reduces the number of items to be sorted recursively, but needs a change in the (classic) pseudo-code of Algorithm 5.3, because Partition must now return the pair of indices which delimit the central sub-array instead of just the position p of the pivot. The following Algorithm 5.4 details an implementation for the three-way partitioning of $S[i, j]$ which uses three pointers that move rightward over this array and maintain the following invariant: P is the pivot driving the three-way distribution, $S[c]$ is the item currently compared against P, and $S[i, c - 1]$ is the part of the input array already scanned and three-way partitioned in its elements. In particular $S[i, c - 1]$ consists of three parts: $S[i, l - 1]$ contains items smaller than P, $S[l, r-1]$ contains items equal to P, and $S[r, c-1]$ contains items larger than P. It may be the case that anyone of these sub-arrays is empty.

Algorithm 5.4 The three-way partitioning: PARTITION(S, i, j)

```
1:  $P = S[i]$; $l = i$; $r = i + 1$;
2:  for ($c = r; c \leq j; c{+}{+}$) do
3:      if ($S[c] == P$) then
4:          swap $S[c]$ with $S[r]$;
5:          $r{+}{+}$;
6:      else if ($S[c] < P$) then
7:          swap $S[c]$ with $S[l]$;
8:          swap $S[c]$ with $S[r]$;
9:          $r{+}{+}$; $l{+}{+}$;
10:     end if
11: end for
12: return $\langle l, r-1 \rangle$;
```

Step 1 initializes P to the first item of the array to be partitioned (which is the pivot), l and r are set to guarantee that the smaller/greater pieces are empty, whereas the piece containing items equal to the pivot consists of the only item P. Next the algorithm scans $S[i+1, j]$ trying to maintain the invariant above. This is easy if $S[c] > P$, because it suffices to extend the part of the larger items by advancing r. In the other two cases (i.e. $S[c] \leq P$) we have to insert $S[c]$ in its correct position among the items of $S[i, r-1]$, in order to preserve the invariant on the three-way partition of $S[i, c]$. The cute idea is that this can be implemented in $O(1)$ time by means of at most two swaps, as described graphically in Figure 5.3.

The three-way partitioning algorithm takes $O(n)$ time and offers two positive properties: (i) stream-like access to the array S which allows the pre-fetching of the items to be read; (ii) the items equal to the pivot can then be eliminated from the following recursive calls. A last note concerns with the pair of indices $\langle l, r-1 \rangle$ returned by PARTITION(S, i, j): they delimit the part of S which consists of elements equal to P, and thus they can be dropped from the subsequent recursive calls, being them in the final correct positions.

5.3.2 Pivot selection

The selection of the pivot is crucial to get balanced partitions, reduce the number of recursive calls, and achieve optimal $O(n \log n)$ time complexity. The pseudo-code of Algorithm 5.3 does not detail the way the pivot is selected because this may occur in many different ways, each offering pros/cons. As an example, if we choose the pivot as the first item of the input array (namely $r = i$), the selection is fast but it is easy to instantiate the input array in order to induce un-balanced partitions: just take S to be an increasing or decreasing ordered sequence of items. Worse than this, it is the observation that any deterministic choice incurs in this drawback.

One way to circumvent bad inputs is to select the pivot *randomly* among the items in $S[i, j]$. This prevents the case that a given input is bad for Quicksort, but makes the behavior of the algorithm un-predictable in advance and dependant on the random selection of the pivot. We can show that the *average* time complexity is the optimal $O(n \log_2 n)$, with an hidden constant small and equal to 1.39. This fact, together with the in-place nature of Quicksort, makes this approach much appealing in practice (cfr `qsort` below).

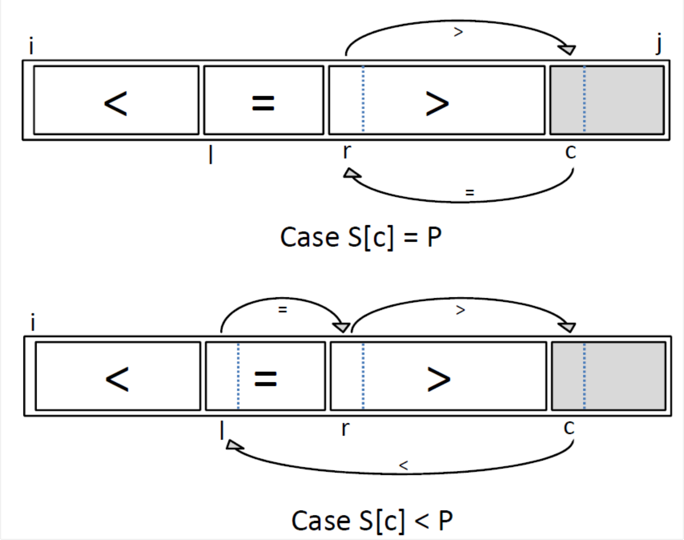


FIGURE 5.3: The two cases and the corresponding swapping. On the arrow we specify the value of the moved item with respect to the pivot.

THEOREM 5.5 *The random selection of the pivot drives Quicksort to compare no more than $2n \ln n$ items, on average.*

Proof The proof is deceptively simple if attacked from the correct angle. We wish to compute the number of comparisons executed by Partition over the sequence S. Let $X_{u,v}$ be the random binary variable which indicates whether $S[u]$ and $S[v]$ are compared by Partition, and denote by $p_{u,v}$ the probability that this event occurs. The average number of comparisons executed by Quicksort can then be computed as $E[\sum_{u,v} X_{u,v}] = \sum_u \sum_{v>u} 1 \times p_{u,v} + 0 \times (1 - p_{u,v}) = \sum_{u=1}^{n} \sum_{v=u+1}^{n} p_{u,v}$ by linearity of expectation.

To estimate $p_{u,v}$ we concentrate on the random choice of the pivot $S[r]$ because two items are compared by Partition only if one of them is the pivot. So we distinguish three cases. If $S[r]$ is smaller or larger than both $S[u]$ and $S[v]$, then the two items $S[u]$ and $S[v]$ are not compared to each other and they are passed to the same recursive call of Quicksort. So the problem presents itself again on a smaller subset of items containing both $S[u]$ and $S[v]$. This case is therefore not interesting for estimating $p_{u,v}$, because we cannot conclude anything at this recursive-point about the execution or not of the comparison between $S[u]$ and $S[v]$. In the case that either $S[u]$ or $S[v]$ is the pivot, then they are compared by Partition. In all other cases, the pivot is taken among the items of S whose value is between $S[u]$ and $S[v]$; so these two items go to two different partitions (hence two different recursive calls of Quicksort) and will never be compared.

As a result, to compute $p_{u,v}$ we have to consider as interesting pivot-selections the two last situations. Among them, two are the ”good” cases, but how many are all these interesting pivot-selections? We need to consider the sorted S, denoted by S'. There is an obvious bijection between pairs of items in S' and pairs of items in S. Let us assume that $S[u]$ is mapped to $S'[u']$ and $S[v]$ is mapped to $S'[v']$, then it is easy to derive the number of interesting pivot-selections as $v' - u' + 1$ which corresponds to the number of items (hence pivot candidates) whose value is between $S[u]$ and $S[v]$ (extremes included). So $p_{u,v} = 2/(v' - u' + 1)$.

This formula may appear complicate because we have on the left u, v and on the right u', v'. Given the bijection between S and S', We can rephrase the statement ”considering all pairs (u, v) in S” as ”considering all pairs (u', v') in S'”, and thus write:

$$E[\sum_{u,v} X_{u,v}] = \sum_{u=1}^{n} \sum_{v=u+1}^{n} p_{u,v} = \sum_{u'=1}^{n} \sum_{v'>u'}^{n} \frac{2}{v'-u'+1} = 2 \sum_{u'=1}^{n} \sum_{k=2}^{n-u'+1} \frac{1}{k} \leq 2 \sum_{u'=1}^{n} \sum_{k=2}^{n} \frac{1}{k} \leq 2n \ln n$$

where the last inequality comes from the properties of the n-th harmonic number, namely $\sum_{k=1}^{n} \frac{1}{k} \leq 1 + \ln n$. ■

The next question is how we can enforce the average behavior. The natural answer is to sample more than one pivot. Typically 3 pivots are randomly sampled from S and the central one (i.e. the median) is taken, thus requiring just two comparisons in $O(1)$ time. Taking more than 3 pivots makes the selection of a “good one” more robust, as proved in the following theorem [2].

THEOREM 5.6 *If Quicksort partitions around the median of* $2s+1$ *randomly selected elements, it sorts n distinct elements in* $\frac{2nH_n}{H_{2s+2}-H_{s+1}} + O(n)$ *expected comparisons, where* H_z *is the z-th harmonic number* $\sum_{i=1}^{z} \frac{1}{i}$.

By increasing s, we can push the expected number of comparisons close to $n \log n + O(n)$, however the selection of the median incurs a higher cost. In fact this can be implemented either by sorting the s samples in $O(s \log s)$ time and taking the one in the middle position $s + 1$ of the ordered sequence; or in $O(s)$ worst-case time via a sophisticated algorithm (not detailed here). Randomization helps in simplifying the selection still guaranteeing $O(s)$ time on average. We detail this approach here because its analysis is elegant and its structure general enough to be applied not only for the selection of the median of an unordered sequence, but also for selecting the item of any rank k.

Algorithm 5.5 Selecting the k-th ranked item: RandSelect(S, k)

```
$r$ = pick a random item from $S$;
$S_<$ = items of $S$ which are smaller than $S[r]$;
$S_>$ = items of $S$ which are larger than $S[r]$;
$n_< = |S_<|$;
$n_= = |S| - (|S_<| + |S_>|)$;
if ($k \leq n_<$) then
    return RandSelect($S_<, k$);
else if ($k \leq (n_< + n_=)$) then
    return $S[r]$;
else
    return RandSelect($S_>, k - n_< - n_=$);
end if
```

Algorithm 5.5 is randomized and selects the item of the unordered S having rank k. It is interesting to see that the algorithmic scheme mimics the one used in the Partitioning phase of Quicksort: here the selected item $S[r]$ plays the same role of the pivot in Quicksort, because it is used to partition the input sequence S in three parts consisting of items smaller/equal/larger than $S[r]$. But unlike Quicksort, RandSelect recurses only in one of these three parts, namely the one containing the k-th ranked item. This part can be determined by just looking at the sizes of those parts, as done in Steps 6 and 8. There are two specific issues that deserve a comment. We do not need to recurse on $S_=$ because it consists of items equal to $S[r]$. If recursion occurs on $S_>$, we need to update the rank k because we are dropping from the original sequence the items belonging to the set $S_< \cup S_=$. Correctness is therefore immediate, so we are left with computing the average time complexity of this algorithm which turns to be the optimal $O(n)$, given that S is unsorted and thus all of its n items have to be examined to find the one having rank k among them.

THEOREM 5.7 *Selecting the k-th ranked item in an unordered sequence of size n takes $O(n)$ average time in the RAM model, and $O(n/B)$ I/Os in the two-level memory model.*

Proof Let us call "good selection" the one that induces a partition in which $n_<$ and $n_>$ are not larger than $2n/3$. We do not care of the size of $S_=$ since, if it contains the searched item, that item is returned immediately as $S[r]$. It is not difficult to observe that $S[r]$ must have rank in the range $[n/3, 2n/3]$ in order to ensure that $n_< \leq 2n/3$ and $n_> \leq 2n/3$. This occurs with probability 1/3, given that $S[r]$ is drawn uniformly at random from S. So let us denote by $\hat{T}(n)$ the average time complexity of RandSelect when run on an array $S[1, n]$. We can write

$$\hat{T}(n) \leq O(n) + \frac{1}{3} \times \hat{T}(2n/3) + \frac{2}{3} \times \hat{T}(n),$$

where the first term accounts for the time complexity of Steps 2-5, the second term accounts for the average time complexity of a recursive call on a "good selection", and the third term is a crude upper bound to the average time complexity of a recursive call on a "bad selection" (that is actually assumed to recurse on the entire S again). This is not a classic recurrent relation because the term $\hat{T}(n)$ occurs on both sides; nevertheless, we observe that this term occurs with different constants in the front. Thus we can simplify the relation by subtracting those terms, so getting $\frac{1}{3}\hat{T}(n) \leq O(n) + \frac{1}{3}\hat{T}(2n/3)$, which gives $\hat{T}(n) = O(n) + \hat{T}(2n/3) = O(n)$. If this algorithm is executed in the two-level memory model, the equation becomes $\hat{T}(n) = O(n/B) + \hat{T}(2n/3) = O(n/B)$ given that the construction of the three subsets can be done via a single pass over the input items. ■

We can use RandSelect in many different ways within Quicksort. For example, we can select the pivot as the median of the entire array S (setting $k = n/2$) or the median among an over-sampled set of $2s + 1$ pivots (setting $k = s + 1$, where $s \ll n/2$), or finally, it could be subtly used to select a pivot that generates a balanced partition in which the two parts have different sizes both being a fraction of n, say αn and $(1 - \alpha)n$ with $\alpha < 0.5$. This last choice $k = \lfloor \alpha n \rfloor$ seems meaningless because the three-way partitioning still takes $O(n)$ time but increases the number of recursive calls from $\log_2 n$ to $\log_{\frac{1}{1-\alpha}} n$. But this observation neglects the sophistication of modern CPUs which are parallel, pipelined and superscalar. These CPUs execute instructions in parallel, but if there is an event that impacts on the instruction flow, their parallelism is *broken* and the computation slows down significantly. Particularly slow are *branch mispredictions*, which occur in the execution of Partition(S, i, j) whenever an item smaller than or equal to the pivot is encountered. If we reduce these cases, then we reduce the number of branch-mispredictions, and thus deploy the full parallelism of modern CPUs. Thus the goal is to properly set α in a way that the reduced number of mispredictions balances the increased number of recursive calls. The right value for α is clearly architecture dependent, recent results have shown that a reasonable value is 0.1.

5.3.3 Bounding the extra-working space

QuickSort is frequently named as an *in-place* sorter because it does not use extra-space for ordering the array S. This is true if we limit ourself to the pseudocode of Algorithm 5.3, but it is no longer true if we consider the cost of managing the recursive calls. In fact, at each recursive call, the OS must allocate space to save the local variables of the caller, in order to retrieve them whenever the recursive call ends. Each recursive call has a space cost of $\Theta(1)$ which has to be multiplied by the number of nested calls Quicksort can issue on an array $S[1, n]$. This number can be $\Omega(n)$ in the worst case, thus making the extra-working space $\Theta(n)$ on some bad inputs (such as the already sorted ones, pointed out above).

Algorithm 5.6 The binary quick-sort with bounded recursive-depth: BoundedQS(S, i, j)

```
 1: while ($j - i > n_0$) do
 2:     $r$ = pick the position of a "good pivot";
 3:     swap $S[r]$ with $S[i]$;
 4:     $p$ = Partition($S, i, j$);
 5:     if ($p \leq \frac{i+j}{2}$) then
 6:         BoundedQS($S, i, p - 1$);
 7:         $i = p + 1$;
 8:     else
 9:         BoundedQS($S, p + 1, j$);
10:         $j = p - 1$;
11:     end if
12: end while
13: InsertionSort($S, i, j$);
```

We can circumvent this behavior by restructuring the pseudocode of Algorithm 5.3 as specified in Algorithm 5.6. This algorithm is cryptic at a first glance, but the underlying design principle is pretty smart and elegant. First of all we notice that the while-body is executed only if the input array is longer than n_0, otherwise Insertion-sort is called in Step 13, thus deploying the well-known efficiency of this sorter over very small sequences. The value of n_0 is typically chosen of few tens of items. If the input array is longer than n_0, a modified version of the classic binary Quicksort is executed that mixes one single recursive call with an iterative while-loop. The ratio underlying this code re-factoring is that the correctness of classic Quicksort does not depend on the order of the two recursive calls, so we can reshuffle them in such a way that the first call is always executed on the smaller part of the two/three-way partition. This is exactly what the IF-statement in step 5 guarantees. In addition to that, the pseudo-code above drops the recursive call onto the larger part of the partition in favor of another execution of the body of the while loop in which we properly changed the parameters i and j to reflect the new extremes of that larger part. This "change" is well-known in the literature of compilers with the name of *elimination of tail recursion*. The net result is that the recursive call is executed on a sub-array whose size is no more than the half of the input array. This guarantees an upper bound of $O(\log_2 n)$ on the number of recursive calls, and thus on the size of the extra-space needed to manage them.

THEOREM 5.8 BoundedQS *sorts n atomic items in the RAM model taking $O(n \log n)$ average time, and using $O(\log n)$ additional working space.*

We conclude this section by observing that the C89 and C99 ANSI standards define a sorting algorithm, called `qsort`, whose implementation encapsulates most of the algorithmic tricks detailed above.[3] This witnesses further the efficiency of the distribution-based sorting scheme over the 2-levels: cache and DRAM.

5.3.4 From binary to multi-way Quicksort

Distribution-based sorting is the *dual* of merge-based sorting in that the first proceeds by splitting sequences according to pivots and then ordering them recursively, while the latter merges sequences which have been ordered recursively. Disk-efficiency was obtained in Multi-way Mergesort by managing (fusing) multiple sequences together. The same idea is applied to design the Multi-way Quicksort which splits the input sequence into $k = \Theta(M/B)$ sub-sequences by using $k-1$ pivots. Given that $k \gg 1$ the selection of those pivots is not a trivial task because it must ensure that the k partitions they form, are *balanced* and thus contain $\Theta(n/k)$ items each. Section 5.3.2 discussed the difficulties underlying the selection of one pivot, so the case of selecting many pivots is even more involved and needs a sophisticated analysis.

We start with denoting by $s_1, \ldots, s_{k-1}$ the pivots used by the algorithm to split the input sequence $S[1, n]$ in k parts, also called *buckets*. For the sake of clarity we introduce two dummy pivots $s_0 = -\infty$ and $s_k = +\infty$, and denote the i-th bucket by $B_i = \{S[j] : \ s_{i-1} < S[j] \leq s_i\}$. We wish to guarantee that $|B_i| = \Theta(n/k)$ for all the k buckets. This would ensure that $\log_k \frac{n}{M}$ partitioning phases are enough to get sub-sequences shorter than M, which can thus be sorted in internal-memory without any further I/Os. Each partitioning phase can be implemented in $O(n/B)$ I/Os by using a memory organization which is the dual of the one employed for Mergesort: namely, 1 input block (used to read from the input sequence to be partitioned) and k output blocks (used to write into the k partitions under formation). By imposing $k = \Theta(M/B)$, we derive that the number of partitioning phases is $\log_k \frac{n}{M} = \Theta(\log_{M/B} \frac{n}{M})$ so that the Multi-way Quicksort takes the optimal I/O-bound of $\Theta(\frac{n}{B} \log_{M/B} \frac{n}{M})$, provided that each partitioning step *distributes evenly* the input items among the k buckets.

To find efficiently k good pivots, we deploy a fast and simple randomized strategy based on *oversampling*, whose pseudocode is given in Algorithm 5.7 below. Parameter $a \geq 0$ controls the amount of oversampling and thus impacts onto the robustness of the selection process as well as on the cost of Step 2. The latter cost is $O((ak)\log(ak))$ if we adopt an optimal in-memory sorter, such as Heapsort or Mergesort, to sort the $\Theta(ak)$ sampled items.

Algorithm 5.7 Selection of $k-1$ good pivots via oversampling

```
1: Take $(a+1)k-1$ samples at random from the input sequence;
2: Sort them into an ordered sequence $A$;
3: For $i = 1, \ldots, k-1$, pick the pivot $s_i = A[(a+1)i]$;
4: **return** the pivots $s_i$;
```

The main idea is to select $\Theta(ak)$ candidate pivots from the input sequence and then pick k among them, namely the ones which are evenly spaced and thus $(a+1)$ far apart from each other. We are arguing that those $\Theta(ak)$ samples provide a faithful picture of the distribution of the items in the

[3] Actually `qsort` is based on a different two-way partitioning scheme that uses two iterators, one moves forward and the other one moves backward over S; a swap occurs whenever two un-sorted items are encountered. The asymptotic time complexity does not change, but practical efficiency can spur from the fact that the number of swaps is reduced since equal items are not moved.

entire input sequence, so that the balanced selection $s_i = A[(a+1)i]$ should provide us with "good pivots". The larger is a the closer to $\Theta(n/k)$ should be the size of all buckets, but the higher would be the cost of sorting the samples. At the extreme case of $a = n/k$, the samples could not be sorted in internal memory! On the other hand, the closer a is to zero the faster would be the pivot selection but more probable is to get unbalanced partitions. As we will see in the following Lemma 5.1, choosing $a = \Theta(\log k)$ is enough to obtain balanced partitions with a pivot-selection cost of $O(k \log^2 k)$ time. We notice that the buckets will be not perfectly balanced but quasi-balanced, since they include no more than $\frac{4n}{k} = O(n/k)$ items; the factor 4 will nonetheless leave unchanged the aimed asymptotic time complexity.

LEMMA 5.1 Let $k \geq 2$ and $a + 1 = 12 \ln k$. A sample of size $(a+1)k - 1$ suffices to ensure that all buckets receive less than $4n/k$ elements, with probability at least $1/2$.

Proof We provide an upper bound of $1/2$ to the probability of the complement event stated in the Lemma, namely that there exists one bucket whose size is larger than $4n/k$. This corresponds to a *failure* sampling, which induces an un-balanced partition. To get this probability estimate we will introduce a cascade of events that are implied by this one and thus have larger and larger probabilities to occur. For the last one in the sequence we will be able to fix an explicit upper-bound of $1/2$. Given the implications, this upper bound will also hold for the original event. And so we will be done.

Let us start by considering the sorted version of the input sequence S, which hereafter we denote by S'. We logically split S' in $k/2$ segments of length $2n/k$ each. The event we are interested in is that there exists a bucket B_i with at least $4n/k$ items assigned to it. As illustrated in Figure 5.4 this large bucket completely spans at least one segment, say t_2 in the Figure below, because the former contains $\geq 4n/k$ items whereas the latter contains $2n/k$ items.

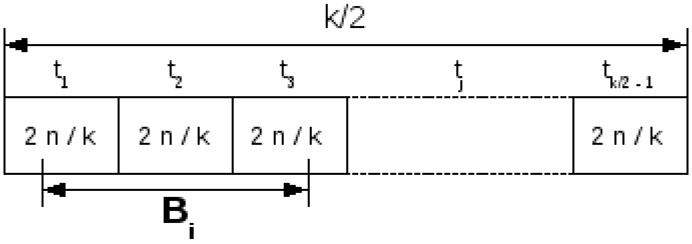


FIGURE 5.4: Splitting of the sorted sequence S' into segments.

By definition of the buckets, the pivots s_{i-1} and s_i which delimit B_i fall outside t_2. Hence, by Algorithm 5.7, less that $(a+1)$ samples fall in the segment overlapped by B_i. In the figure it is t_2, but it might be any segment of S'. So we have that:

$$\begin{aligned}\mathcal{P}(\exists B_i : \ |B_i| \geq 4n/k) &\leq \mathcal{P}(\exists t_j : \ t_j \text{ contains } < (a+1) \text{ samples}) \\ &\leq \frac{k}{2} \times \mathcal{P}(\text{a specific segment contains } < (a+1) \text{ samples}) \qquad (5.1)\end{aligned}$$

where the last inequality comes from the *union bound*, given that $k/2$ is the number of segments constituting S'. So we will hereafter concentrate on providing an upper bound to the last term.

The probability that one sample ends in a given segment is equal to $\frac{(2n/k)}{n} = \frac{2}{k}$ because they are assumed to be drawn uniformly at random from S (and thus from S'). So let us call X the number of those samples, we are interested in computing $\mathcal{P}(X < a+1)$. We start by observing that $E[X] = ((a+1)k - 1) \times \frac{2}{k} = 2(a+1) - \frac{2}{k}$. The Lemma assumes that $k \geq 2$, so $E[X] \geq 2(a+1) - 1$ which is $\geq \frac{3}{2}(a+1)$ for all $a \geq 1$. We can thus state that $a + 1 \leq (2/3)E[X] = (1 - \frac{1}{3})E[X]$. This form is useful to resort the Chernoff bound:

$$\mathcal{P}(X < (1-\delta)E[X]) \leq e^{-\frac{\delta^2}{2}\, E[X]}$$

By setting $\delta = 1/3$, we derive

$$\begin{aligned} \mathcal{P}(X < a+1) &\leq \mathcal{P}(X < (1-\frac{1}{3})E[X]) \leq e^{-(E[X]/2)(1/3)^2} = e^{-E[X]/18} \\ &\leq e^{-(3/2)(a+1)/18} = e^{-(a+1)/12} = e^{-\ln k} = \frac{1}{k} \end{aligned} \tag{5.2}$$

where we used the inequality $E[X] \geq (3/2)a + 1$ and the lemma's assumption that $a + 1 = 12 \ln k$. By plugging this value in Eqn 5.1, we get the statement of the Lemma. ∎

5.3.5 The Dual Pivot Quicksort

Very recently, in 2012, a new Quicksort variant due to Yaroslavskiy was chosen as the standard sorting method for Oracle's Java 7 runtime library.[4] The decision for the change was based on empirical studies showing that, on average, his new algorithm is faster than the formerly used classic Quicksort. The improvement was achieved by using a new three-way partition strategy based on a pair of pivots *properly moved* over the input sequence S. Surprisingly, this algorithmic scheme was considered not promising by several theoretical studies in the past. Instead, authors of [7] showed that this improvement is due to a reduction in the number of comparisons, about $1.9n \ln n$, at the expenses of an increase in the number of swaps, about $0.6n \ln n$. Despite this trade-off, the dual-pivot strategy results more than 10% faster, probably because branch mispredictions are more costly than memory accesses in modern PC architectures, as we commented earlier in this chapter.[5]

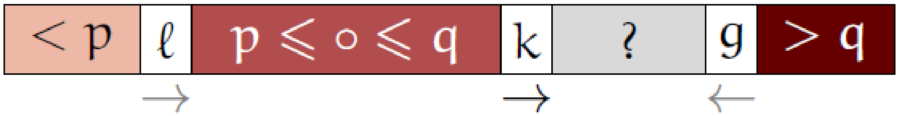


FIGURE 5.5: The invariant guaranteed by the Dual Pivot Quicksort

Figure 5.5 provides a pictorial representation of the Invariant maintained by the Dual Pivot Quicksort during the partitioning step. The algorithm uses two pivots– namely, p and q–, and three iterators– namely, ℓ, k and g– which partition the input sequence in four pieces.

[4] The discussion is archived at `http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628`

[5] Studies about the selection of the pivots and the reason for this improvement are yet underway.

- The leftmost piece, delimited by ℓ, includes all items which are smaller that p;
- the mid-left piece, delimited between positions ℓ and k, includes items that are larger than p and smaller than q;
- the mid-right piece, delimited between positions k and g, includes items that have still to be examined by the algorithm;
- the rightmost piece, after position g, includes items that are larger than q.

This invariant is similar to the invariant maintained in Section 5.3.1, but there the three-way partition was obtained by using one single pivot, treating in a special way the items equal to the pivot, and all iterators were moving towards the right; here the algorithm uses two pivots, items equal to these pivots are not treated separately, and two pivots move rightward (i.e. ℓ and k) whereas the third one (i.e. g) moves leftward. This way the items larger than q are correctly located at the end of the input sequence, and the items smaller than p are correctly located at the beginning of the input sequence.

The partitioning proceeds in rounds until $k \geq g$ occurs. Each round first compares $S[k] < p$ and, if this is not true, it compares $S[k] > q$. In the former case it swaps $S[k]$ with $S[\ell]$, and then advances the pointers. In the latter case, the algorithm enters in a loop that moves g leftward up to the position where $S[g] \leq q$; here, it swaps $S[k]$ with $S[g]$. So it is the comparison with $S[k]$ which drives the phase, possibly switching to a long g shifting to the left. This nesting of the comparisons is the key for the efficiency of the algorithm which is able to move towards "better" comparisons which reduce the branch mispredictions. We cannot go into the sophisticate details of the analysis, so we refer the reader to [7].

This example is illustrative of the fact that classic algorithms and problems, known for decades and considered antiquated, may be harbingers of innovation and deep/novel theoretical analysis. So do not ever leave the curiosity to explore and analyze new algorithmic schemes!

5.4 Sorting with multi-disks$^\infty$

The bottleneck in disk-based sorting is obviously the time needed to perform an I/O operation. In order to mitigate this problem, we can use D disks working in parallel so to transfer DB items per I/O. On the one hand this increases the bandwidth of the I/O subsystem, but on the other hand, it makes the design of I/O-efficient algorithms particularly difficult. Let's see why.

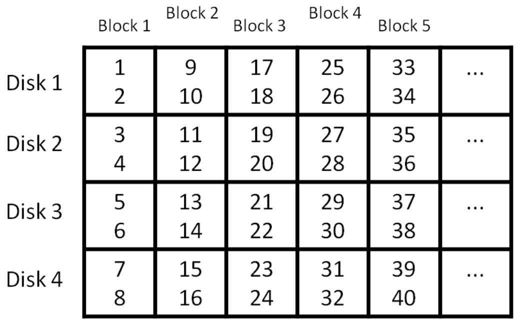
The simplest approach to manage parallel disks is called *disk striping* and consists of looking at the D disks as *one single* disk whose page size is $B' = DB$. This way we gain simplicity in algorithm design by just using *as-is* any algorithm designed for one disk, now with a disk-page of size B'. Unfortunately, this simple approach pays an un-negligible price in terms of I/O-complexity:

$$O(\frac{n}{B'} \log_{M/B'} \frac{n}{M}) = O(\frac{n}{DB} \log_{M/DB} \frac{n}{M})$$

This bound is not optimal because the base of the logarithm is D times smaller than what indicated by the lower bound proved in Theorem 5.3. The ratio between the bound achieved via disk-striping and the optimal bound is $1 - \log_{M/B} D$, which shows disk striping to be less and less efficient as the number of disks increases $D \longrightarrow M/B$. The problem resides in the fact that we are not deploying the *independency* among disks by using them as a monolithic sub-system.

On the other hand, deploying this independency is tricky and it took several years before designing fully-optimal algorithms running over multi-disks and achieving the bounds stated in Theorem 5.3. The key problem with the management of multi-disks is to guarantee that every time we access the disk sub-system, we are able to read or write D pages each one coming from or going to a different disk. This is to guarantee a throughput of DB items per I/O. In the case of sorting, such a difficulty arises both in the case of distributed-based and merge-based sorters, each with its specialties given the duality of those approaches.

Let us consider the multi-way Quicksort. In order to guarantee a D-way throughput in reading the input items, these must be distributed evenly among the D disks. For example they could be striped circularly as indicated in Figure 5.6. This would ensure that a scan of the input items takes $O(n/DB)$ optimal I/Os.



	Block 1	Block 2	Block 3	Block 4	Block 5	
Disk 1	1 2	9 10	17 18	25 26	33 34	...
Disk 2	3 4	11 12	19 20	27 28	35 36	...
Disk 3	5 6	13 14	21 22	29 30	37 38	...
Disk 4	7 8	15 16	23 24	31 32	39 40	...

FIGURE 5.6: An example of striping a sequence of items among $D = 4$ disks, with $B = 2$.

This way the subsequent distribution phase can read the input sequence at that I/O-speed. Nonetheless problems occur when writing the output sub-sequences produced by the partitioning process. In fact that writing should guarantee that each of these sub-sequences is circularly striped among the disks in order to maintain the invariant for the next distribution phase (to be executed independently over those sub-sequences). In the case of D disks, we have D output blocks that are filled by the partitioning phase. So when they are full these D blocks must be written to D distinct disks to ensure full I/O-parallelism, and thus one I/O. Given the striping of the runs, if all these output blocks belong to the same run, then they can be written in one I/O. But, in general, they belong to different runs so that conflicts may arise in the writing process because blocks of different runs could have to be written onto the same disks. An example is given in Figure 5.7 where we have illustrated a situation in which we have three runs under formation by the partitioning phase of Quicksort, and three disks. Runs are striped circularly among the 3 disks and shadowed blocks correspond to the prefixes of the runs that have been already written on those disks. Arrows point to the next free-blocks of each run where the partitioning phase of Quicksort can append the next distributed items. The figure depicts an extremely bad situation in which all these blocks are located on the same disk D_2, so that an I/O-conflict may arise if the next items to be output by the partitioning phase go to these runs. This practically means that the I/O-subsystem must *serialize* the write operation in $D = 3$ distinct I/Os, hence loosing all the I/O-parallelism of the D-disks. In order to avoid these difficulties, there are known *randomized* solutions that ensure optimal I/Os in the average case [6].

In what follows we sketch a deterministic multi-disk sorter, known as Greed Sort [5], which solves the difficulties above via an elegant merge-based approach which consists of two stages: first, items are *approximately* sorted via an I/O-efficient Multi-way Merger that deals with $R = \Theta(\sqrt{M/B})$ sorted runs in an independent way (thus deploying disks in parallel), and then it *completes* the sorting of the input sequence by using an algorithm (aka ColumnSort, due to T. Leighton in 1985) that takes a linear number of I/Os when executed over *short* sequences of length $O(M^{\frac{3}{2}})$. Correctness comes from the fact that the distance of the un-sorted items from their correct sorted position, after the first stage, is smaller than the size of the sequences manageable by Columsort. Hence the second

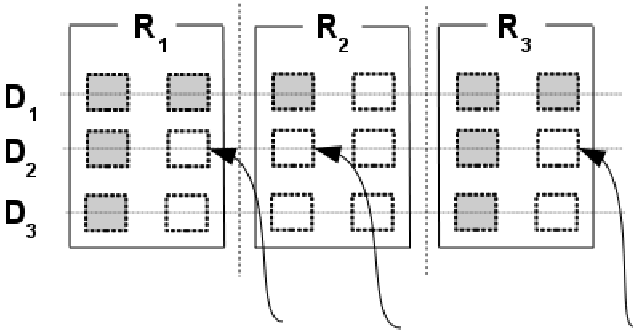


FIGURE 5.7: An example of an I/O-conflict in writing $D = 3$ blocks belonging to 3 distinct runs.

stage can correctly turn the approximately-sorted sequence into a totally-sorted sequence by a single pass.

How to get the approximately sorted runs in I/O-efficient way is the elegant algorithmic contribution of GreedSort. We sketch its main ideas here, and refer the interested reader to the corresponding paper [5] for further details. We assume that sorted runs are stored in a striped way among the D disks, so that reading D consecutive blocks from each of them takes one I/O. As we discussed for Quicksort, also in this Merge-based approach we could incur in I/O-conflicts when reading these runs. GreedSort avoids this problem by operating independently on each disk: in a parallel read operation, GreedSort fetches the two *best* available blocks from each disk. These two blocks are called "best" because they contain the *smallest minimum item*, say m_1, and the *smallest maximum item*, say m_2, currently present in blocks stored on that disk (possibly these two blocks are the same). It is evident that this selection can proceed independently over the D disks, and it needs a proper data structure that keeps track of minimum/maximum items in disk-blocks. Actually [5] shows that this data structure can fit in internal memory, thus not incurring any further I/Os for this selection operations.

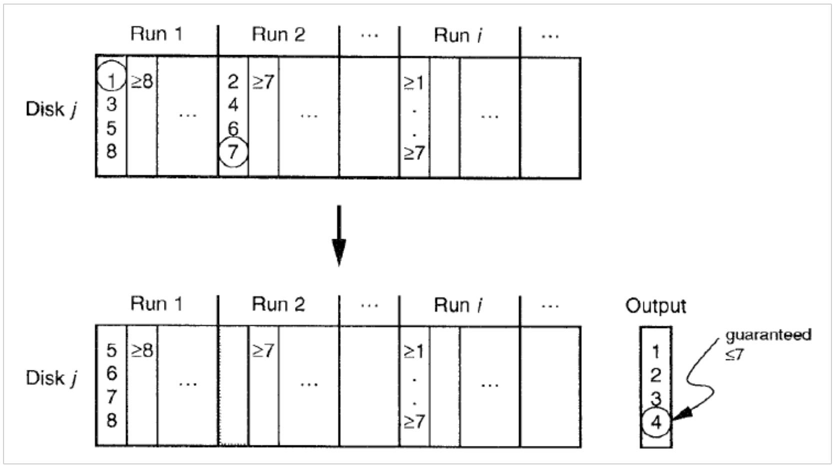


FIGURE 5.8: Example taken from the GreedSort's paper [5].

Figure 5.8 shows an example on disk j, which contains the blocks of several runs because of the striping-based storage. The figure assumes that run 1 contains the block with the smallest minimum item (i.e. 1) and run 2 contains the block with the smallest maximum item (i.e. 7). All the other blocks which come from run 1 contain items larger than 8 (i.e. the maximum of the first block), and all the other blocks which come from run 2 contain items larger than 7. All blocks coming form other runs have minimum larger than 1 and maximum larger than 7. Greedsort then merges these blocks creating two new sorted blocks: the first one is written to output (it contains the items $\{1, 2, 3, 4\}$), the second one is written back to the run of the smallest minimum m_1, namely run 1 (it contains the items $\{5, 6, 7, 8\}$). This last write back into run 1 does not disrupt that ordered subsequence, because the second block contains surely items smaller than the maximum of the block of m_1.

We notice that the items written in output are not necessarily the four smallest items of all runs. In fact it could exist a block in another run (different from runs 1 and 2) which contains a value within $[1, 4]$, say 2.5, and whose minimum is larger than 1 and whose maximum is larger than 7. So this block is compatible with the selection we did above from run 1 and 2, but it contains items that should be stored in the first block of the sorted sequence. So the selection of the "two-best blocks" proceeds independently over all disks until all runs have been examined and written in output. The final sequence produced by this merging process is *not* sorted, but if we read it in a striped-way along all D disks, then it results *approximately* sorted as stated in the following lemma (proved in [5]).

LEMMA 5.2 A sequence is called *L-regressive* if any pair of un-sorted records, say $\ldots y \ldots x \ldots$ with $y > x$, has distance less than L in the sequence. The previous sorting algorithm creates an output that is L-regressive, with $L = RDB = D\sqrt{MB}$.

The application of ColumnSort over the L-regressive sequence, by sliding a window of $2L$ items which moves L steps forward at each phase, allows to produce a merged sequence which is totally sorted. In fact $L = D\sqrt{MB} \leq DB\sqrt{M} \leq M^{3/2}$ and thus ColumnSort is effective in producing the entirely sorted sequence. We notice that at this point this sorted sequence is striped along all D disks, thus the invariant for the next merging phase is preserved and the merge can thus start over a number of runs that has been reduced by a factor R. The net result is that each merging takes $O(n/DB)$ I/Os, the total number of merging stages is $\log_R \frac{n}{M} = O(\log_{M/B} \frac{n}{M})$, and thus the optimal I/O-bound follows.

References

[1] Alok Aggarwal and Jeffrey S. Vitter. The Input/Output complexity of Sorting and Related Problems. *Communication of the ACM*, 31(9): 1116-1127, 1988.

[2] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Procs of the 8th ACM-SIAM Symposium on Discrete Algorithms*, 360–369, 1997.

[3] Donald E. Knuth. *The Art of Computer Programming: volume 3*. Addison-Wesley, 2nd Edition, 1998.

[4] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The basic toolbox.* Springer, 2009.

[5] Mark H. Nodine and Jeffrey S. Vitter. Greed Sort: Optimal Deterministic Sorting on Parallel Disks. *Journal of the ACM*, 42(4): 919-933, 1995.

[6] Jeffrey S. Vitter. External memory algorithms and data structures. *ACM Computing Surveys*, 33(2):209–271, 2001.

[7] Sebastian Wild and Markus E. Nebel Average case analysis of Java 7's Dual Pivot Quicksort. In *European Symposium on Algorithms (ESA)*, Lecture Notes in Computer Science 7501, Springer, 826–836, 2012.

6

Sorting Strings

In the previous chapter we dealt with sorting *atomic items*, namely items that either occupy $O(1)$ space or have to be managed in their entirety as atomic objects, and thus without possibly deploying their constituent parts. In the present chapter we will generalize those algorithms, and introduce new ones, to deal with the case of *variable-length items* (aka *strings*). More formally, we will be interested in solving efficiently the following problem:

The string-sorting problem. *Given a sequence $S[1, n]$ of strings, having total length N and drawn from an alphabet of size σ, sort these strings in increasing lexicographic order.*

The first idea to attack this problem consists of deploying the power of *comparison-based* sorting algorithms, such as QuickSort or MergeSort, and implementing a proper comparison function between pair of strings. The obvious way to do this is to compare the two strings from their beginning, character-by-character, find their mismatch and then use this character to derive their lexicographic order. Let $L = N/n$ be the average length of the strings in S, an optimal comparison-based sorter would take $O(L\, n \log n) = O(N \log n)$ average time on RAM, because every string comparison may involve $O(L)$ characters on average.

Apart from the time complexity, which is not optimal (see next), the key limitation of this approach in a memory-hierarchy is that S is typically implemented as an *array of pointers* to strings which are stored elsewhere, possibly on disk if N is very large or spread in the internal-memory of the computer. Figure 6.1 shows the two situations via a graphical example. Whichever is the allocation your program chooses to adopt, the sorter will *indirectly* order the strings of S by moving their pointers rather than their characters. This situation is typically neglected by programmers, with a consequent slowness of their sorter when executed on large string sets. The motivation is clear, every time a string comparison is executed between two items, say $S[i]$ and $S[j]$, these two pointers are resolved by accessing their corresponding strings, so that two cache misses or I/Os are elicited in order to fetch and then compare their characters. As a result, the algorithm might incur $\Theta(n \log n)$ I/Os. As we noticed in the first chapter of these notes, the Virtual Memory of the OS will provide an help by buffering the most recently compared strings, and thus possibly reducing the number of incurred I/Os. Nevertheless, two arrays are here competing for that buffering space— i.e. the array of pointers and the strings— and time is wasted by *re-scanning* over and over again string prefixes which have been already compared because of their brute-force comparison.

The rest of this lecture is devoted to propose algorithms which are optimal in the number of executed character comparisons, and possibly offer I/O-conscious patterns of memory accesses which make them efficient also in the presence of a memory hierarchy.

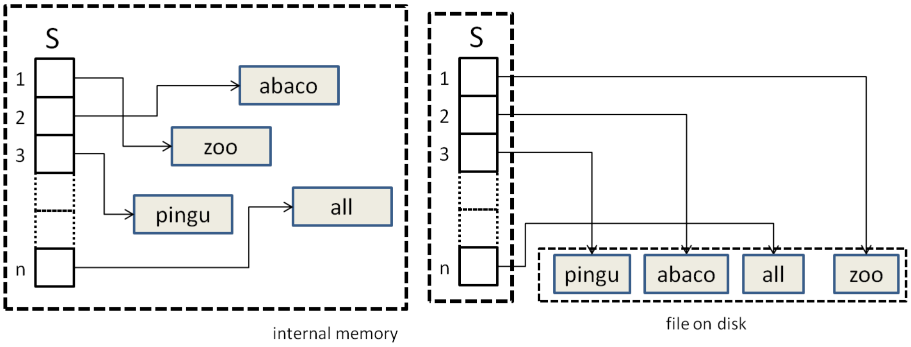


FIGURE 6.1: Two examples of string allocation, spread in the internal memory (left) and written contiguously in a file (right).

6.1 A lower bound

Let d_s be the length of the shortest prefix of the string $s \in S$ that distinguishes it from the other strings in the set. The sum $d = \sum_{s \in S} d_s$ is called the *distinguishing prefix* of the set S. Referring to Figure 6.1, and assuming that S consists of the 4 strings shown in the picture, the distinguishing prefix of the string `all` is `al` because this substring does not prefixes any other string in S, whereas `a` does.

It is evident that any string sorter must compare the initial d_s characters of s, otherwise it would be unable to distinguish s from the other strings in S. So $\Omega(d)$ is a term that must appear in the string-sorting lower bound. However, this term does not take into account the cost to compute the sorted order among the n strings, which is $\Omega(n \log n)$ string comparisons.

LEMMA 6.1 Any algorithm solving the string-sorting problem must execute $\Omega(d + n \log n)$ comparisons.

This formula deserves few comments. Assume that the n strings of S are binary, share the initial ℓ bits, and differ for the rest $\log n$ bits. So each string consists of $\ell + \log n$ bits, and thus $N = n(\ell + \log n)$ and $d = \Theta(N)$. The lower bound in this case is $\Omega(d + n \log n) = \Omega(N + n \log n) = \Omega(N)$. But string sorters based on Mergesort or Quicksort (as the ones detailed above) take $\Theta((\ell + \log n) n \log n) = \Theta(N \log n)$ time. Thus, for any ℓ, those algorithms may be far from optimality of a factor $\Theta(\log n)$.

One could wonder whether the upper-bound can be turned to be smaller than the input size N. This is possible because the string sorting could be implemented without looking at the entire content of strings, provided that $d < N$. And indeed, this is the reason why we introduced the parameter d, which allows a finer analysis of the following algorithms.

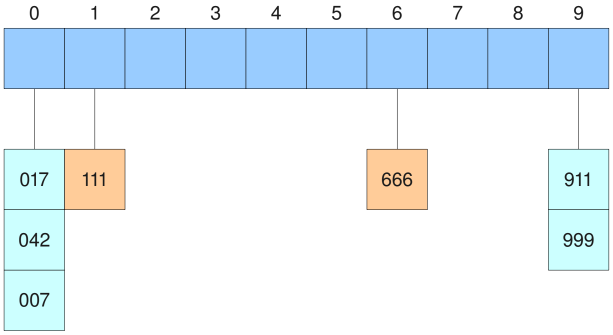


FIGURE 6.2: First distribution step in MSD-first RadixSort.

6.2 RadixSort

The first step to get a more competitive algorithm for string sorting is to look at strings as *sequence of characters* drawn from an integer alphabet $\{0, 1, 2, \ldots, \sigma - 1\}$ (aka *digits*). This last condition can be easily enforced by sorting in advance the characters occurring in S, and then assigning to each of them an integer (rank) in that range. This is typically called *naming* process and takes $O(N \log \sigma)$ time because we can use a binary-search tree built over the at most σ distinct characters occurring in S. After that, all strings can be sorted by considering them as sequence of σ-bounded digits.

Hereafter we assume that strings in S have been drawn from a integer alphabet of size σ and keep in mind that, if this is not the case, a term $O(N \log \sigma)$ has to be added to the time complexity. Moreover, we observe that each character can be encoded in $\lceil \log(\sigma + 1) \rceil$ bits; so that the input size is $\Theta(N \log \sigma)$ whenever it is measured in bits.

We can devise two main variants of RadixSort that differentiate each other according to the order in which the digits of the strings are processed: MSD-first processes the strings rightward starting from the Most Significant Digit, LSD-first processes the strings leftward starting from the Least Significant Digit.

6.2.1 MSD-first

This algorithm follows a divide&conquer approach which processes the strings character-by-character from their beginning, and distributes them into σ buckets. Figure 6.2 shows an example in which S consists of seven strings which are distributed according to their first (most-significant) digit in 10 buckets, because $\sigma = 10$. Since buckets `1` and `6` consist of one single string each, their ordering is known. Conversely, buckets `0` and `9` have to be sorted recursively according to the second digit of the strings contained into them. Figure 6.3 shows the result of the recursive sorting of those two buckets. Then, to get the ordered sequence of strings, all groups are concatenated in left-to-right order.

We point out that, the distribution of the strings among the buckets can be obtained via a comparison-based approach, namely binary search, or by the usage of CountingSort. In the former case the distribution cost is $O(\log \sigma)$ per string, in the latter case it is $O(1)$.

It is not difficult to notice that distribution-based approaches originate search trees. The classic Quicksort originates a binary search tree. The above MSD-first RadixSort originates a σ-ary tree because of the σ-ary partition executed at every distribution step. This tree takes in the literature

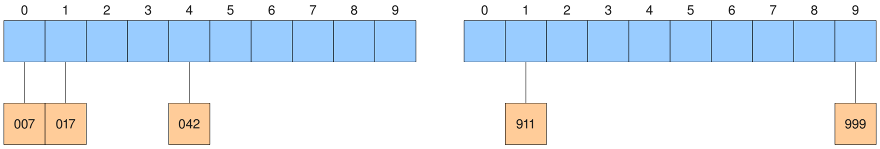


FIGURE 6.3: Recursive sort of bucket 0 (left) and bucket 9 (right) according to the second digit of their strings.

the name of *trie*, or *digital* search tree, and its main use is in string searching (as we will detail in the Chapter 9). An example of trie is given in Figure 6.4.

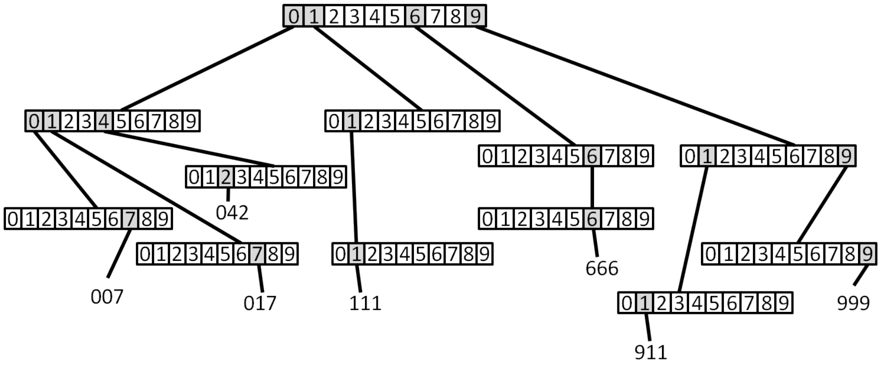


FIGURE 6.4: The trie-based view of MSD-first RadixSort for the strings of Figure 6.2

Every node is implemented as a σ-sized array, one entry per possible alphabet character. Strings are stored in the leaves of the trie, hence we have n leaves. Internal nodes are less than N, one per character occurring in the strings of S. Given a node u, the downward path from the root of the trie to u spells out a string, say $s[u]$, which is obtained by concatenating the characters encountered in the path traversal. For example, the path leading to the leaf 017 traverses three nodes, one per possible prefix of that string. Fixed a node u, all strings that descend from u share the same prefix $s[u]$. For example, $s[\text{root}]$ is the empty string, and indeed all strings of S do share no prefix. Take the leftmost child of the root, it spells the string 0 because it is reached form the root by traversing the edge spurring from the 0-entry of the array. Notice that the trie may contain *unary* nodes, namely nodes that have one single child. All the other internal nodes that have at least two children are called *branching* nodes. In the figure we have 9 unary nodes and 3 branching nodes, where $n = 7$ and $N = 21$. In general the trie can have no more than n branching nodes, and $O(N)$ unary nodes. Actually the unary nodes which have a descending branching node are at most $O(d)$. In fact, these unary nodes correspond to characters occurring in the distinguishing prefixes and the lowest descending branching nodes correspond to the characters that end the distinguishing prefixes; on the other hand, the unary paths which spur from the lowest branching nodes in the trie and lead to leaves correspond to the string suffixes which follow those distinguishing prefixes. In algorithmic terms, the unary nodes correspond to buckets formed by items all sharing the same compared-character in the distribution of MSD-first RadixSort, the branching nodes correspond to buckets formed by items with distinct compared-characters in the distribution of MSD-first RadixSort.

If edge labels are alphabetically sorted, as in Figure 6.4, reading the leaves according to the pre-order visit of the trie gets the sorted S. This suggests a simple trie-based string sorter. The idea is to start with an empty trie, and then insert one string after the other into it. Inserting a string $s \in S$ in the current trie consists of tracing a downward path until s's characters are matched with edge labels. As soon as the next character in s cannot be matched with any of the edges outgoing from the reached node u,[1] then we say that the mismatch for s is found. So a *special* node is appended to the trie at u with that branching character. This special node points to s. The specialty resides in the fact that we have dropped the not-yet-matched suffix of s, but the pointer to the string keeps implicitly track of it for the subsequent insertions. In fact, if inserting another string s' we encounter the special-node u, then we resort the string s (linked to it) and create a (unary) path for the other characters constituting the common prefix between s and s' which descends from u. The last node in this path branches to s and s', possibly dropping again the two paths corresponding to the not-yet-matched suffixes of these two strings, and introducing for each of them one special character.[2]

Every time a trie node is created, an array of size σ is allocated, thus taking $O(\sigma)$ time and space. So the following theorem can be proved.

THEOREM 6.1 *Sorting strings over an (integer) alphabet of size σ can be solved via a trie-based approach in $O(d\,\sigma)$ time and space.*

Proof Every string s spells out a path of length $O(d_s)$, before that the special node pointing to s is created. Each node of those paths allocates $O(\sigma)$ space and takes that amount of time to be allocated. Moreover $O(1)$ is the time cost for traversing a trie-node. Therefore $O(\sigma)$ is the time spent for each traversed/created node. The claim then follows by visiting the constructed trie and listing its leaves from left to right (given that they are lexicographically sorted, because the naming of characters is lexicographic and thus reflects their order). ■

The space occupancy is significant and should be reduced. An option is to replace the σ-sized array into each node u with an hash table (with chaining) of size proportional to the number of edges spurring out of u, say e_u. This guarantees $O(1)$ average time for searching and inserting one edge in each node. The construction time becomes in this case: $O(d)$ to insert all strings in the trie (here every node access takes constant time), $O(\sum_u e_u \log e_u) = O(\sum_u e_u \log \sigma) = O(d \log \sigma)$ for the sorting of the trie edges over all nodes, and $O(d)$ time to scan the trie leaves rightward via a pre-order visit of the trie, and thus get the dictionary strings in lexicographic order.

THEOREM 6.2 *Sorting strings drawn from an integer alphabet of size σ, by using the trie-based approach with hashing, takes $O(d \log \sigma)$ average time and $O(d)$ space.*

When σ is small we cannot conclude that this result is *better than the lower bound* provided in Lemma 6.1 because that applies to comparison-based algorithms and thus it does not apply to hashing or integer sorting.

[1] This actually means that the slot in the σ-sized array of u corresponding to the next character of s is `null`.

[2] We are assuming that allocating a σ-sized array cost $O(1)$ time.

The space allocated for the trie can be further reduced to $O(n)$, and the construction time to $O(d+n\log\sigma)$, by using *compacted* tries, namely tries in which the unary paths have been compacted into single edges whose length is equal to the length of the compacted unary path. The discussion of this data structure is deferred to Chapter 9.

6.2.2 LSD-first

The next sorter was discovered by Herman Hollerith more than a century ago and led to the implementation of a card-sorting machine for the 1890 U.S. Census. It is curious to find that he was the founder of a company that then became IBM [4]. The algorithm is counter-intuitive because it sorts strings digit-by-digit starting from the least-significant one and using a *stable* sorter as black-box for ordering the digits. We recall that a sorter is *stable* iff equal keys maintain in the final sorted order the one they had in the input. This sorter is typically the CountingSort (see e.g. [5]), and this is the one we use below to describe the LSD-first RadixSort. We assume that all strings have the same length L, otherwise they are *logically* padded to their front with a special digit which is assumed to be *smaller than* any other alphabet digit. The ratio is that, this way, the LSD-first RadixSort correctly obtains an ordered lexicographic sequence.

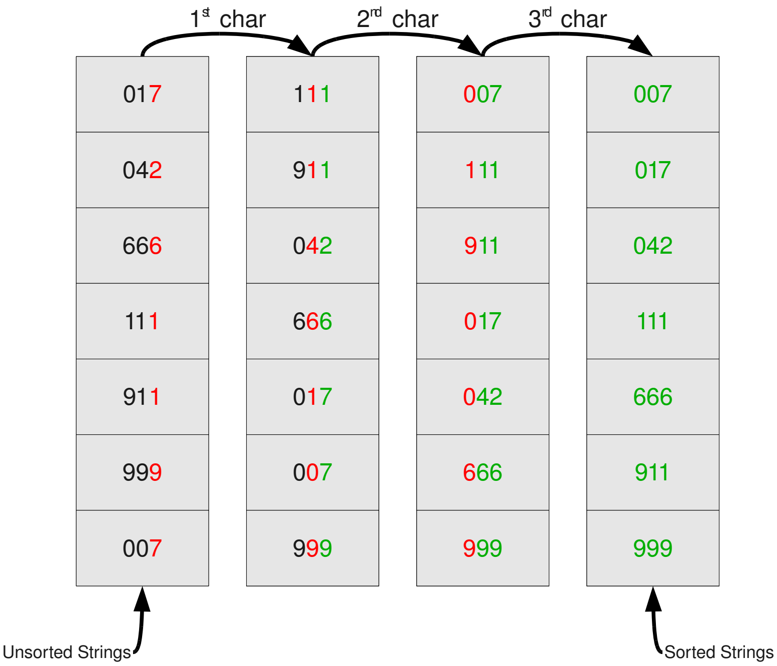


FIGURE 6.5: A running example for LSD-first RadixSort.

The LSD-first RadixSort consists of L phases, say $i = 1, 2, \ldots, L$, in each phase we stably sort all strings according to their i-th least significant digit. A running example of LSD-first RadixSort is given in Figure 6.5: the red digits (characters) are the ones that are going to be sorted in the current phase, whereas the green digits are the ones already sorted in the previous phases. Each phase produces a new sorted order which deploys the order in the input sequence, obtained from the previous phases, to resolve the ordering of the strings which show equal digits in the currently compared position i. As an example let us consider the strings `111` and `017` in the 2nd phase of Figure 6.5. These strings present the same second digit so their ordering in the second phase poses `111` before `017`, just because this was the ordering after the first sorting step. This is clearly a wrong order which will be nonetheless correctly adjusted after the last phase which operates on the third digit, i.e. `1` vs `0`. The time complexity can be easily estimated as L times the cost of executing CountingSort over n integer digits drawn from the range $[1, \sigma]$. Hence it is $O(L\,(n + \sigma))$. A nice property of this sorter is that it is in-place whenever the sorting black-box is in-place, namely $\sigma = O(1)$.

LEMMA 6.2 LSD-first Radixsort solves the string-sorting problem over an integer alphabet of size σ in $O(L\,(n+\sigma)) = O(N + L\sigma)$ time and $O(N)$ space. The sorter is in-place iff an in-place digit sorter is adopted.

Proof Time and space efficiency follow from the previous observations. The correctness is proved by deploying the stability of the Counting Sort. Let α and β be two strings of S, and assume that $\alpha < \beta$ according to the lexicographic order. Since we assumed that S's strings have the same length we can decompose these two strings into three parts: $\alpha = \gamma a \alpha_1$ and $\beta = \gamma b \beta_1$, where γ is the longest common prefix between α and β (possibly it is empty), $a < b$ are the first mismatch characters, α_1 and β_1 are the two remaining suffixes (which may be empty).

Let us now look at the history of comparisons between the digits of α and β. We can identify three stages, depending on the position of the compared digit within the three-way decomposition above. Since the algorithm starts from the least-significant digit, it starts comparing digits in α_1 and β_1. We do not care about the ordering after the first $|\alpha_1| = |\beta_1|$ phases, because at the immediately next phase, α and β are sorted in accordance to characters a and b. Since $a < b$ this sorting places α before β. All other $|\gamma|$ sorting steps will compare the digits falling in γ, which are equal in both strings, so their order will not change because of the stability of the digit-sorter. At the end we will correctly have $\alpha < \beta$. Since this holds for any pair of strings in S, the final sequence produced by LSD-first RadixSort will be lexicographically ordered. ■

The previous time bound deserves few comments. LSD-first RadixSort processes all digits of all strings, so it seems not appealing when $d \ll N$ with respect to MSD-first RadixSort. But the efficiency of LSD-first RadixSort hinges onto the observation that nobody prevents a phase to sort *groups of digits* rather than a single digit at a time. Clearly the longer is this group, the larger is the time complexity of a phase, but the smaller is the number of phases. We are in the presence of a trade-off that can be tuned by investigating deeply the relation that exists between these two parameters. Without loss of generality, we simplify our discussion by assuming that the strings in S are *binary* and have equal-length of b bits, so $N = bn$ and $\sigma = 2$. Of course, this is not a limitation in practice because any string is encoded in memory as a bit sequence; in theory, we can assume to pre-sort the alphabet-characters in $O(N \log \sigma)$ time and then encode them via bit-sequences of $\lceil \log \sigma \rceil$ bits reflecting the digit order.

LEMMA 6.3 LSD-first RadixSort takes $\Theta(\frac{b}{r}(n + 2^r))$ time and $O(nb) = O(N)$ space to sort n strings of b bits each. Here $r \leq b$ is a positive integer fixed in advance.

Proof We decompose each string in $g = \frac{b}{r}$ groups of r bits each. Each phase will order the strings according to a group of r bits. Hence CountingSort is asked to order n integers between 0 and $2^r - 1$ (extremes included), so it takes $O(n + 2^r)$ time. As there are $g = \frac{b}{r}$ phases, the total time is $O(g\,(n + 2^r)) = O((\frac{b}{r})(n + 2^r))$. ■

Given n and b, we need to choose a proper value for r such that the time complexity is minimized. We could derive this minimum via analytic calculus (i.e. first-order derivatives) but, instead, we argue for the minimum as follows. Since the CountingSort uses $O(n + 2^r)$ time to sort each group of r digits, it is useless to use groups shorter than $\log n$, given that $\Omega(n)$ time has to be payed in any case. So we have to choose r in the interval $[\log n, b]$. As r grows larger than $\log n$, the time complexity in Lemma 6.3 also increases because of the ratio $2^r/r$. So the best choice is $r = \Theta(\log n)$ for which the time complexity is $O(\frac{bn}{\log n})$.

THEOREM 6.3 *LSD-first Radixsort sorts n strings of b bits each in $O(\frac{bn}{\log n})$ time and $O(bn)$ space, by using CountingSort on groups of $\Theta(\log n)$ bits. The algorithm is not in-place because it needs $\Theta(n)$ space for the Counting Sort.*

We finally observe that bn is the total length in bits of the strings in S, so we can express that number as $N \log \sigma$ since each character takes $\log \sigma$ bits to be represented.

COROLLARY 6.1 LSD-first Radixsort solves the string-sorting problem on strings drawn from an arbitrary alphabet in $O(\frac{N \log \sigma}{\log n})$ time and $O(N \log \sigma)$ bits of space.

If $d = \Theta(N)$ and σ is a constant, the comparison-based lower-bound (Lemma 6.1) becomes $\Omega(d + n \log n) = \Omega(N)$. So LSD-first Radixsort equals or even beats that lower bound; but this is not surprising because this sorter operates on an *integer* alphabet and uses CountingSort, so it is *not* a comparison-based string sorter.

Comparing the trie-based construction (Theorems 6.1–6.2) and the LSD-first RadixSort algorithm we conclude that the former is always better than the latter for $d = O(\frac{N}{\log n})$, which is true for most practical cases. In fact LSD-first RadixSort needs to scan the whole string set whichever are the string compositions, whereas the trie-construction may skip some string suffixes whenever $d \ll N$. However the LSD-first approach avoids the dynamic memory allocation, incurred by the construction of the trie, and the extra-space due to the storage of the trie structure. This additional space and work is non negligible in practice and could impact unfavorably on the real performance of the trie-based sorter, or even prevent its use over large string sets because the internal memory has bounded size M.

6.3 Multi-key Quicksort

This is a variant of the well-known Quicksort algorithm extended to manage items of variable length. Moreover it is a comparison-based string sorter which matches the lower bound of $\Omega(d + n \log n)$ stated in Lemma 6.1. For a recap about Quicksort we refer the reader to the previous chapter. Here it is enough to recall that Quicksort hinges onto two main ingredients: the pivot-selection procedure and the algorithm to partition the input array according to the selected pivot. In Chapter 5 we discussed widely these issues, for the present section we fix ourselves to a pivot-selection based on a *random* choice and to a *three-way* partitioning of the input array. All other variants discussed in Chapter 5 can be easily adapted to work in the string setting too.

The key here is that items are not considered as atomic, but they are strings to be split into their constituent characters. Now the pivot is a character, and the partitioning of the input strings is done according to the single character that occupies a given position within them. Figure 6.1 details the pseudocode of Multi-key Quicksort, in which it is assumed that the input set R is *prefix free*, so no string in R prefixes any other string. This condition can be easily guaranteed by assuming that strings of R are distinct and logically padded with a dummy character that is smaller than any other character in the alphabet. This guarantees that any pair of strings in R admits a bounded longest-common-prefix (shortly, *lcp*), and that the mismatch character following the lcp does exist in both strings.

Algorithm 6.1 MULTIKEYQS(R, i)

```
 1: if $|R| \leq 1$ then
 2:     return R;
 3: else
 4:     choose a pivot-string $p \in R$;
 5:     $R_< = \{s \in R \mid s[i] < p[i]\}$;
 6:     $R_= = \{s \in R \mid s[i] = p[i]\}$;
 7:     $R_> = \{s \in R \mid s[i] > p[i]\}$;
 8:     $A = MultikeyQS(R_<, i)$;
 9:     $B = MultikeyQS(R_=, i + 1)$;
10:     $C = MultikeyQS(R_>, i)$;
11:     return the concatenated sequence $A, B, C$;
12: end if
```

The algorithm receives in input a sequence R of strings to be sorted and an integer parameter $i \geq 0$ which denotes the offset of the character driving the three-way partitioning of R. The pivot-character is $p[i]$ where p is randomly chosen string within R. The real implementation of this three-way partitioning can follow the PARTITION procedure of Chapter 5. MULTIKEYQS(R, i) assumes that the following pre-condition holds on its input parameters: All strings in R are lexicographically sorted up to their $(i - 1)$-long prefixes. So the sorting of a string sequence $R[1, n]$ is obtained by invoking MULTIKEYQS(R, 1), which ensures that the invariant trivially holds for the initial sequence R. Steps 5-7 partitions R in three subsets whose notation is explicative of their content. All these three subsets are recursively sorted and their ordered sequences are eventually concatenated in order to obtain the ordered R. The tricky issue here is the invocation of the three recursive calls:

- the sorting of the strings in $R_<$ and $R_>$ has still to reconsider the ith character, because we just checked that it is smaller/greater than $p[i]$ (and this is not sufficient to order those strings). So recursion does not advance i, but it hinges on the current validity of the invariant.
- the sorting of the strings in $R_=$ can advance i because, by the invariant, these strings are sorted up to their $(i - 1)$-long prefixes and, by construction of $R_=$, they share the i-th character. Actually this character is equal to $p[i]$, so $p \in R_=$ too.

These observations make correctness immediate. We are therefore left with the problem of computing the average time complexity of MULTIKEYQS(R, 1). Let us concentrate on a single string, say $s \in R$, and count the number of comparisons that involve one of its characters. There are two cases, either $s \in R_< \cup R_>$ or $s \in R_=$. In the first case, s is compared with the pivot-string p and then included in a smaller set $R_< \cup R_> \subset R$ with the offset i unchanged. In the other case s is compared with p but, since the i-th character is found equal, it is included in a smaller set *and* offset i is advanced. If the pivot selection is *good* (see Chapter 5), the three-way partitions are balanced and thus

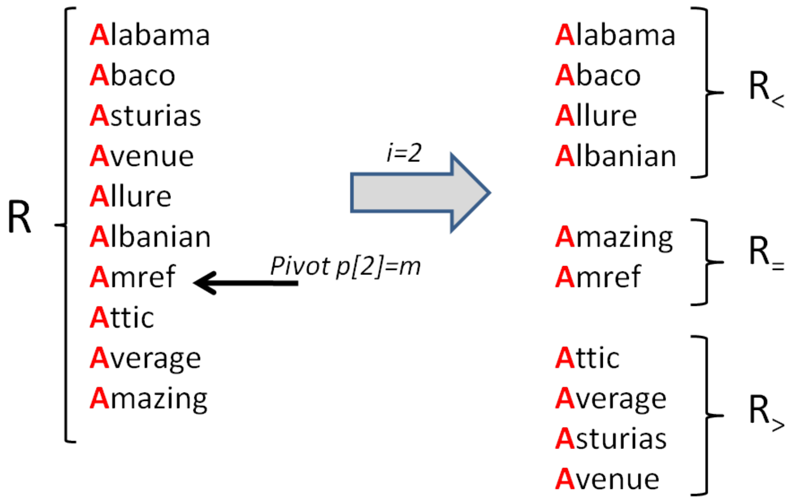


FIGURE 6.6: A running example for MULTIKEYQS(R, 2). In red we have the 1-long prefix shared by all strings in R.

$|R_< \cup R_>| \leq \alpha\, n$, for a proper constant $\alpha < 1$. As a result, both cases cost $O(1)$ time but one reduces the string set by a constant factor, while the other increases i. Since the initial set R has size n, and i is bounded above by the string length $|s|$, we have that the number of comparisons involving s is $O(|s| + \log n)$. Summing up over all strings in R we get the time bound $O(N + n \log n)$. A finer look at the second case shows that i can be bounded above by the number of characters that belong to s's distinguishing prefix, because these characters will led s to be located in a singleton set.

THEOREM 6.4 *Multi-key Quicksort solves the string-sorting problem by performing $O(d + n \log n)$ character comparisons on average. The bound can be turned into a worst-case bound by adopting a worst-case linear-time algorithm to select the pivot as the median of R. This is optimal.*

Comparing this result against what was obtained for the MSD-first RadixSort (Theorem 6.2) we observe that in that Theorem we got $\log \sigma$ instead of $\log n$, nevertheless the succinct space occupancy make Multi-key Quicksort very appealing in practice. Moreover, similarly as done for Quicksort, it is possible to prove that if the partition is done around the median of $2t + 1$ randomly selected pivots, Multi-key Quicksort needs at most $\frac{2nH_n}{H_{2t-2} - H_{t+1}} + O(d)$ average comparisons. By increasing the sample size t, one can reduce the time near to $n \log n + O(d)$. This bound is similar to the one obtained with the trie-based sorter (see Theorem 6.2, where the log-argument was σ instead of n), but the algorithm is much simpler, it does not use additional data structures (i.e. hash tables), and in fact it is the typical choice in practice.

We conclude this section by noticing an interesting parallel between Multikey Quicksort and *ternary* search trees, as discussed in [6]. These are search data structures in which each node contains a *split character* and pointers to low and high (or left and right) children. In some sense a ternary search tree is obtained from a trie by collapsing together the children whose leading edges are smaller/greater than the split character. If a given node splits on the character in position i, its low and high children also split on i-th character. Instead, the equal-child splits on the next $(i + 1)$-th character. Ternary search trees may be balanced by either inserting elements in random order

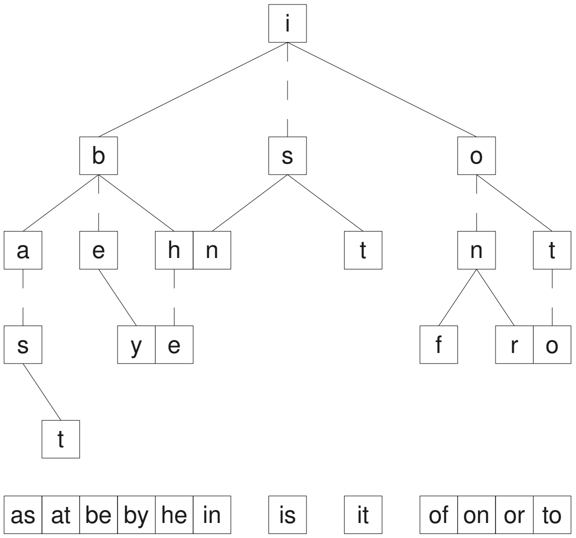


FIGURE 6.7: A ternary search tree for 12 two-letter strings. The low and high pointers are shown as solid lines, while the pointers to the equal-child are shown as dashed lines. The split character is indicated within the nodes.

or applying a variety of known schemes. Searching proceeds by following edges according to the split-characters of the encountered nodes. Figure 6.7 shows an example of a ternary search tree. The search for the word $P =$"ir" starts at the root, which is labeled with the character `i`, with the offset $x = 1$. Since $P[1] = \mathtt{i}$, the search proceeds down to the equal-child, increments x to 2, and thus reaches the node with split-character `s`. Here $P[2] = \mathtt{r} < \mathtt{s}$, so the search goes to the low/left child which is labeled `n`, and keeps x unchanged. At that node the search stops, because the split character is different and no (low or high) children do exist. So the search concludes that P does not belong to the string set indexed by the ternary search tree.

THEOREM 6.5 *A search for a pattern $P[1, p]$ in a perfectly-balanced ternary search tree representing n strings takes at most $\lfloor \log n \rfloor + p$ comparisons. This is optimal when P is drawn from a general alphabet (or, equivalently, for a comparison-based search algorithm).*

6.4 Some observations on the I/O-model$^{\infty}$

Sorting strings on disk is not nearly as simple as it is in internal memory, and a bunch of sophisticated string-sorting algorithms have been introduced in the literature which achieve I/O-efficiency (see e.g. [1, 3]). The difficulty is that strings have variable length and their brute-force comparisons over the sorting process may induce a lot of I/Os. In the following we will use the notation: n_s is the number of strings shorter than B, whose total length is N_s, n_l is the number of strings longer than B, whose total length is N_l. Clearly $n = n_s + n_l$ and $N = N_s + N_l$,

The known algorithms can be classified according to the way strings are managed in their sorting process. We can devise mainly three models of computations [1]:

***Model A*:** Strings are considered *indivisible* (i.e., they are moved in their entirety and cannot be broken into characters), except that long strings can be divided into blocks of size B.

***Model B*:** Relaxes the indivisibility assumption of Model A by allowing strings to be divided into single characters, but this may happen *only in internal memory*.

***Model C*:** Waives the indivisibility assumption by allowing division of strings *in both internal and external memory*.

Model A forces to use Mergesort-based sorters which achieve the following optimal bounds:

THEOREM 6.6 *In Model A, string sorting takes* $\Theta(\frac{N_s}{B}\log_{M/B}\frac{N_s}{B} + n_l\log_{M/B} n_l + \frac{N_s+N_l}{B})$ *I/Os.*

The first term in the bound is the cost of sorting the short strings, the second term is the cost of sorting the long strings, and the last term accounts for the cost of reading the whole input. The result shows that sorting short strings is as difficult as sorting their individual characters, which are N_s, while sorting long strings is as difficult as sorting their first B characters. The lower bound for small strings in Theorem 6.6 is proved by extending the technique used in Chapter 5 and considering the special case where all n_s small strings have the same length N_s/n_s. The lower bound for the long strings is proved by considering the n_l small strings obtained by looking at their first B characters. The upper bounds in Theorem 6.6 are obtained by using a special Multi-way MergeSort approach that takes advantage of a *lazy trie* stored in internal memory to guide the merge passes among the strings.

Model B presents a more complex situation, and leads to handle long and short strings separately.

THEOREM 6.7 *In Model B, sorting long strings takes* $\Theta(n_l\log_M n_l + \frac{N_l}{B})$ *I/Os, whereas sorting short strings takes* $O(\min\{n_s\log_M n_s, \frac{N_s}{B}\log_{M/B}\frac{N_s}{B}\})$ *I/Os.*

The first bound for long strings is optimal, the second for short strings is not. Comparing the optimal bound for long strings with the corresponding bound in Theorem 6.6, we notice that they differ in terms of the base of the logarithm: the base is M rather than M/B. This shows that breaking up long strings in internal memory is provably helpful for external string-sorting. The upper bound is obtained by combining the String B-tree data structure (described in Chapter 9) with a proper buffering technique. As far as short strings are concerned, we notice that the I/O-bound is the same as the cost of sorting all the characters in the strings when the average length N_s/n_s is $O(\frac{B}{\log_{M/B} M})$. For the (in practice) narrow range $\frac{B}{\log_{M/B} M} < \frac{N_s}{n_s} < B$, the cost of sorting short strings becomes $O(n_s\log_M n_s)$. In this range, the sorting complexity for Model B is lower than the one for Model A, which shows that breaking up short strings in internal memory is provably helpful.

Surprisingly enough, the best deterministic algorithm for Model C is derived from the one designed from Model B. However, since Model C allows to split strings on disk too, we can use randomization and hashing. The main idea is to shrink strings by hashing some of their pieces. Since hashing does not preserve the lexicographic order, these algorithms must orchestrate the selection of the string pieces to be hashed with a carefully designed sorting process so that the correct sorted order may be eventually computed. Recently [3] proved the following result (which can be extended to the more powerful cache-oblivious model):

THEOREM 6.8 *In Model C, the string-sorting problem can be solved by a randomized algorithm using* $O(\frac{n}{B}(\log^2_{M/B}\frac{N}{M})(\log n) + \frac{N}{B})$ *I/Os, with arbitrarily high probability.*

References

[1] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeff S. Vitter. On sorting strings in external memory. In *Procs of the ACM Symposium on Theory of Computing (STOC)*, pp. 540–548, 1997.

[2] Jon L. Bentley and Doug McIlroy. Engineering a sort function. *Software-Practice and Experience*, pages 1249–1265, 1993.

[3] Rolf Fagerberg, Anna Pagh, Rasmus Pagh. External String Sorting: Faster and Cache-Oblivious. In *Procs of the Symposium on Theoretical Aspects of Computer Science (STACS)*, LNCS 3884, Springer, pp. 68-79, 2006.

[4] Herman Hollerith. Wikipedia's entry at `http://en.wikipedia.org/wiki/Herman_Hollerith`.

[5] Tomas H. Cormen, Charles E. Leiserson, Ron L. Rivest and Cliff Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[6] Robert Sedgewick and Jon L. Bentley. Fast algorithms for sorting and searching strings. *Eight Annual ACM-SIAM Symposium on Discrete Algorithms*, 360-369, 1997.

7

The Dictionary Problem

In this lecture we present *randomized* and simple, yet smart, data structures that solve efficiently the classic **Dictionary Problem**. These solutions will allow us to propose *algorithmic fixes* to some *issues* that are typically left untouched or only addressed via ”hand waiving” in basic courses on algorithms.

Problem. *Let $\mathcal{D}$ be a set of n objects, called the* dictionary*, uniquely identified by* `keys` *drawn from a* universe *U. The dictionary problem consists of designing a data structure that efficiently supports the following three basic operations:*

- `Search(`k`):` *Check whether $\mathcal{D}$ contains an object o with key k =* `key`$[o]$*, and then return* `true` *or* `false`*, accordingly. In some cases, we will ask to return the object associated to this key, if any, otherwise return* `null`.
- `Insert(`x`):` *Insert in $\mathcal{D}$ the object x indexed by the key k =* `key`$[x]$*. Typically it is assumed that no object in $\mathcal{D}$ has key k, before the insertion takes place; condition which may easily be checked by executing a preliminary query* `Search(`k`)`.
- `Delete(`k`):` *Delete from $\mathcal{D}$ the object indexed by the key k, if any.*

In the case that all three operations have to be supported, the problem and the data structure are named *dynamic*; otherwise, if only the query operation has to be supported, the problem and the data structure are named *static*.

We point out that in several applications the structure of an object x typically consists of a pair $\langle k, d \rangle$, where $k \in U$ is the key indexing x in $\mathcal{D}$, and d is the so called *satellite data* featuring x. For the sake of presentation, in the rest of this chapter, we will drop the satellite data and the notation $\mathcal{D}$ in favor just of the key set $S \subseteq U$ which consists of all keys indexing objects in $\mathcal{D}$. This way we will simplify the discussion by considering dictionary search and update operations only on those

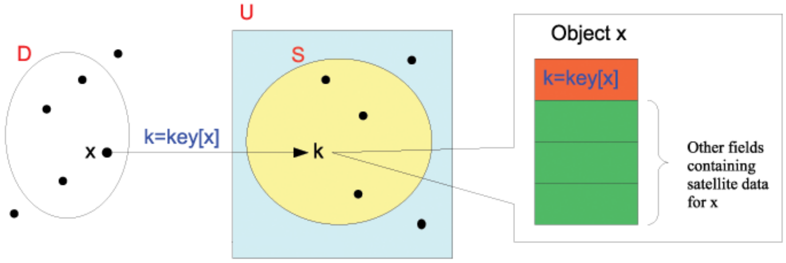


FIGURE 7.1: Illustrative example for U, S, $\mathcal{D}$ and an object x.

keys rather than on (full) objects. But if the context will require also satellite data, we will again talk about objects and their implementing pairs. See Figure 7 for a graphical representation.

Without any loss of generality, we can assume that keys are non-negative integers: $U = \{0, 1, 2, ...\}$. In fact keys are represented in our computer as binary strings, which can thus be interpreted as natural numbers.

In the following sections, we will analyze three main data structures: direct-address tables (or arrays), hash tables (and some of their sophisticated variants) and the Bloom Filter. The former are introduced for teaching purposes, because several times the dictionary problem can be solved very efficiently without resorting involved data structures. The subsequent discussion on hash tables will allow us, first, to fix some issues concerning with the design of a *good* hash function (typically flied over in basic algorithm courses), then, to design the so called *perfect* hash tables, that address *optimally and in the worst case* the static dictionary problem, and then move to the elegant *cuckoo* hash tables, that manage dictionary updates efficiently, still guaranteing constant query time in the *worst case*. The chapter concludes with the *Bloom Filter*, one of the most used data structures in the context of large dictionaries and Web/Networking applications. Its surprising feature is to guarantee query and update operations in constant time, and, more surprisingly, to take space depending on the number of keys n, but not on their lengths. The reason for this impressive *"compression"* is that keys are dropped and only a *fingerprint of few bits* for each of them is stored; the incurred cost is a *one-side error* when executing Search(k): namely, the data structure answers in a correct way when $k \in S$, but it may answer un-correctly if k is not in the dictionary, by returning answer *true* (a so called *false positive*). Despite that, we will show that the probability of this error can be mathematically bounded by a function which exponentially decreases with the space m reserved to the Bloom Filter or, equivalently, with the number of bits allocated per each key (i.e. its fingerprint). The nice thing of this formula is that it is enough to take m a constant-factor slightly more than n and reach a negligible probability of error. This makes the Bloom Filter much appealing in several interesting applications: crawlers in search engines, storage systems, P2P systems, etc..

7.1 Direct-address tables

The simplest data structure to support all dictionary operations is the one based on a binary table T, of size $u = |U|$ bits. There is a one-to-one mapping between keys and table's entries, so that entry $T[k]$ is set to 1 *iff* the key $k \in S$. If some satellite data for k has to be stored, then T is implemented as a table of *pointers to* these satellite data. In this case we have that $T[k] \neq \texttt{NULL}$ *iff* $k \in S$ and it points to the memory location where the satellite data for k are stored.

Dictionary operations are trivially implemented on T and can be performed in *constant (optimal) time* in the worst case. The main issue with this solution is that table's occupancy depends on the

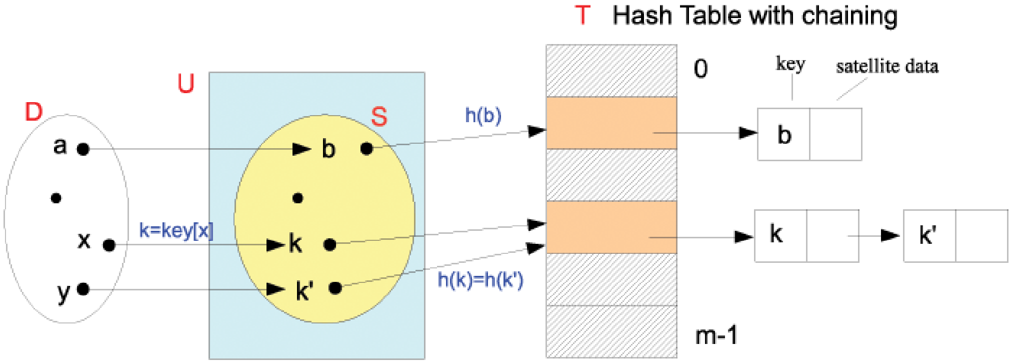


FIGURE 7.2: Hash table with chaining.

universe size u; so if $n = \Theta(u)$, then the approach is optimal. But if the dictionary is small compared to the universe, the approach wastes a lot of space and becomes unacceptable. Take the case of a university which stores the data of its students indexed by their IDs: there can be even million of students but if the IDs are encoded with integers (hence, 4 bytes) then the universe size is 2^32, and thus of the order of billions. Smarter solutions have been therefore designed to reduce the sparseness of the table still guaranteeing the efficiency of the dictionary operations: among all proposals, hash tables and their many variations provide an excellent choice!

7.2 Hash Tables

The simplest data structure for implementing a dictionary are arrays and lists. The former data structure offers constant-time access to its entries but linear-time updates; the latter offers opposite performance, namely linear-time to access its elements but constant-time updates whenever the position where they have to occur is given. Hash tables combine the best of these two approaches, their simplest implementation is the so called *hashing with chaining* which consists of an *array of lists*. The idea is pretty simple, the hash table consists of an array T of size m, whose entries are either NULL or they point to lists of dictionary items. The mapping of items to array entries is implemented via an *hash function* $h : U \rightarrow \{0, 1, 2 \ldots, m-1\}$. An item with key k is appended to the list pointed to by the entry $T[h(k)]$. Figure 7.2 shows a graphical example of an hash table with chaining; as mentioned above we will hereafter interchange the role of items and their indexing keys, to simplify the presentation, and imply the existence of some satellite data.

Forget for a moment the implementation of the function h, and assume just that its computation takes constant time. We will dedicate to this issue a significant part of this chapter, because the overall efficiency of the proposed scheme strongly depends on the efficiency and efficacy of h to *distribute* items evenly among the table slots.

Given a *good* hash function, dictionary operations are easy to implement over the hash table because they are just turned into operations on the array T and on the lists which spur out from its entries. Searching for an item with key k boils down to a search for this key in the list $T[h(k)]$. Inserting an item x consists of appending it at the front of the list pointed to by $T[h(\mathtt{key}[x])]$. Deleting an item with key k consists of first searching for it in the list $T[h(k)]$, and then removing the corresponding object from that list. The running time of dictionary operations is constant for Insert(x), provided that the computation of $h(k)$ takes constant time, and it is linear in the length of the list pointed to by $T[h(k)]$ for both the other operations, namely Search(k) and Delete(k).

Therefore, the efficiency of hashing with chaining depends on the ability of the hash function h to *evenly distribute* the dictionary items among the m entries of table T, the more evenly distributed they are the shorter is the list to scan. The worst situation is when all dictionary items are hashed to the same entry of T, thus creating a list of length n. In this case, the cost of searching is $\Theta(n)$ because, actually, the hash table boils down to a single linked list!

This is the reason why we are interested in *good* hash functions, namely ones that distribute items among table slots uniformly at random (aka *simple uniform hashing*). This means that, for such hash functions, every key $k \in S$ is *equally likely* to be hashed to everyone of the m slots in T, *independently* of where other keys are hashed. If h is such, then the following result can be easily proved.

THEOREM 7.1 *Under the hypotheses of simple uniform hashing, there exists a hash table with chaining, of size m, in which the operation* Search(k) *over a dictionary of n items takes* $\Theta(1+n/m)$ *time on average. The value* $\alpha = n/m$ *is often called the* load factor *of the hash table.*

Proof In case of unsuccessful search (i.e. $k \notin S$), the average time for operation Search(k) equals the time to perform a full scan of the list $T[h(k)]$, and thus it equals its length. Given the uniform distribution of the dictionary items by h, the average length of a generic list $T[i]$ is $\sum_{x \in S} p(h(\mathtt{key}[x]) = i) = |S| \times \frac{1}{m} = n/m = \alpha$. The "plus 1" in the time complexity comes from the constant-time computation of $h(k)$.

In case of successful search (i.e. $k \in S$), the proof is less straightforward. Assume x is the i-th item inserted in T, and let the insertion be executed at the tail of the list $\mathcal{L}(x) = T[h(\mathtt{key}[x])]$; we need just one additional pointer per list keep track of it. The number of elements examined during Search(key[x]) equals the number of items which were present in $\mathcal{L}(x)$ plus 1, i.e. x itself. The average length of $\mathcal{L}(x)$ can be estimated as $n_i = \frac{i-1}{m}$ (given that x is the i-th item to be inserted), so the cost of a successful search is obtained by averaging $n_i + 1$ over all n dictionary items. Namely,

$$\frac{1}{n}\sum_{i=1}^{n}\left(1 + \frac{i-1}{m}\right) = 1 + \frac{\alpha}{2} - \frac{1}{2m}$$

Therefore, the total time is $O(2 + \frac{\alpha}{2} - \frac{1}{2m}) = O(1 + \alpha)$. ■

The space taken by the hash table can be estimated very easily by observing that list pointers take $O(\log n)$ bits, because they have to index one out of n items, and the item keys take $O(\log u)$ bits, because they are drawn from a universe U of size u. It is interesting to note that the key storage can dominate the overall space occupancy of the table as the universe size increases (think e.g. to URL as keys). It might take even more space than what it is required by the list pointers and the table T (aka, the indexing part of the hash table). This is a simple but subtle observation which will be exploited when designing the Bloom Filter in Section 7.7. To be precise on the space occupancy, we state the following corollary.

COROLLARY 7.1 Hash table with chaining occupies $(m + n)\log_2 n + n \log_2 u$ bits.

It is evident that if the dictionary size n is known, the table can be designed to consists of $m = \Theta(n)$ cells, and thus obtain a constant-time performance over all dictionary operations, on average. If n is unknown, one can resize the table whenever the dictionary gets too small (many deletions), or too large (many insertions). The idea is to start with a table size $m = 2n_0$, where n_0 is the initial number of dictionary items. Then, we keep track of the current number n of dictionary items present in T.

If the dictionary gets too small, i.e. $n < n_0/2$, then we halve the table size and rebuild it; if the dictionary gets too large, i.e. $n > 2n_0$, then we double the table size and rebuild it. This scheme guarantees that, at any time, the table size m is proportional to the dictionary size n by a factor 2, thus implying that $\alpha = m/n = O(1)$. Table rebuilding consists of inserting the current dictionary items in a new table of proper size, and drop the old one. Since insertion takes $O(1)$ time per item, and the rebuilding affects $\Theta(n)$ items to be deleted and $\Theta(n)$ items to be inserted, the total rebuilding cost is $\Theta(n)$. But this cost is paid at least every $n_0/2 = \Omega(n)$ operations, the worst case being the one in which these operations consist of all insertions or all deletions; so the rebuilding cost can be spread over the operations of this sequence, thus adding a $O(1 + m/n) = O(1)$ *amortized cost* at the actual cost of each operation. Overall this means that

COROLLARY 7.2 Under the hypothesis of simple uniform hashing, there exists a *dynamic* hash table with chaining which takes constant time, expected and amortized, for all three dictionary operations, and uses $O(n)$ space.

7.2.1 How do we design a "good" hash function ?

Simple uniform hashing is difficult to guarantee, because one rarely knows the probability distribution according to which the keys are drawn and, in addition, it could be the case that the keys are not drawn independently. Let us dig into this latter feature. Since h maps keys from a universe of size u to a integer-range of size m, it induces a partition of those keys in m subsets $U_i = \{k \in U \ : \ h(k) = i\}$. By the *pigeon principle* it does exist at least one of these subsets whose size is larger than the average load factor u/m. Now, if we reasonably assume that the universe is sufficiently large to guarantee that $u/m = \Omega(n)$, then we can choose the dictionary S as that subset of keys and thus force the hash table to offer its worst behavior, by boiling down to a single linked list of length $\Omega(n)$.

This argument is independent of the hash function h, so we can conclude that no hash function is robust enough to guarantee *always* a "good" behavior. In practice heuristics are used to create hash functions that perform well sufficiently often: The design principle is to compute the hash value in a way that it is expected to be independent of any regular pattern that might exist among the keys in S. The two most famous and practical hashing schemes are based on division and multiplication, and are briefly recalled below (for more details we refer to any classic text in Algorithms, such as [3].

Hashing by division. The hash value is computed as the remainder of the division of k by the table size m, that is: $h(k) = k \bmod m$. This is quite fast and behaves well as long as $h(k)$ does not depend on few bits of k. So power-of-two values for m should be avoided, whereas prime numbers not too much close to a power-of-two should be chosen. For the selection of large prime numbers do exist either simple, but slow (exponential time) algorithms (such as the famous Sieve of Eratosthenes method); or fast algorithms based on some (randomized or deterministic) *primality test*.[1] In general, the cost of prime selection is $o(m)$; and thus turns out to be negligible with respect to the cost of table allocation.

Hashing by multiplication. The hash value is computed in two steps: First, the key k is multiplied by a constant A, with $0 < A < 1$; then, the fractionary part of kA is multiplied by m and the integral part of the result is taken as index into the hash table T. In formula: $h(k) = \lfloor m\,\mathtt{frac}(kA)\rfloor$. An

[1] The most famous, and randomized, primality test is the one by Miller and Rabin; more recently, a deterministic test has been proposed which allowed to prove that this problem is in $\mathcal{P}$. For some more details look at `http://en.wikipedia.org/wiki/Prime_number`.

advantage of this method is that the choice of m is not critical, and indeed it is usually chosen as a power of 2, thus simplifying the multiplication step. For the value of A, it is often suggested to take $A = (\sqrt{5} - 1)/2 \cong 0.618$.

It goes without saying that none of these practical hashing schemes surpasses the problem stated above: it is always possible to select a bad set of keys which makes the table T to boil down to a single linked list, e.g., just take multiples of m to disrupt the hashing-by-division method. In the next section, we propose an hashing scheme that is robust enough to guarantee a "good" behavior on average, whichever is the input dictionary.

7.3 Universal hashing

Let us first argue by a counting argument why the uniformity property, we required to *good* hash functions, is computationally hard to guarantee. Recall that we are interested in hash functions which map keys in U to integers in $\{0, 1, ..., m-1\}$. The total number of such hash functions is $m^{|U|}$, given that each key among the $|U|$ ones can be mapped into m slots of the hash table. In order to guarantee uniform distribution of the keys and independence among them, our hash function should be anyone of those ones. But, in this case, its representation would need $\Omega(\log_2 m^{|U|}) = \Omega(|U| \ \log_2 m)$ bits, which is really too much in terms of space occupancy and in the terms of computing time (i.e. it would take at least $\Omega(\frac{|U| \log_2 m}{\log_2 |U|})$ time to just read the hash encoding).

Practical hash functions, on the other hand, suffer of several *weaknesses* we mentioned above. In this section we introduce the powerful Universal Hashing scheme which overcomes these drawbacks by means of *randomization* proceeding similarly to what was done to make *more robust* the pivot selection in the Quicksort procedure (see Chapter 5). There, instead of taking the pivot from a fixed position, it was chosen *uniformly at random* from the underlying array to be sorted. This way no input was bad for the pivot-selection strategy, which being unfixed and randomized, allowed to spread the risk over the many pivot choices guaranteeing that most of them led to a good-balanced partitioning.

Universal hashing mimics this algorithmic approach into the context of hash functions. Informally, we do not set the hash function in advance (cfr. fix the pivot position), but we will choose the hash function *uniformly at random* from a *properly defined* set of hash functions (cfr. random pivot selection) which is defined in a way that it is very probable to pick a *good* hash for the current input set of keys S (cfr. the partitioning is balanced). Good function means one that minimizes the number of *collisions* among the keys in S, and can be computed in constant time. Because of the randomization, even if S is fixed, the algorithm will behave differently on various executions, but the nice property of Universal Hashing will be that, on average, the performance will be the expected one. It is now time to formalize these ideas.

DEFINITION 7.1

Let $\mathcal{H}$ be a finite collection of hash functions which map a given universe U of keys into integers in $\{0, 1, ..., m-1\}$. $\mathcal{H}$ is said to be **universal** if, and only if, for all pairs of distinct keys $x, y \in U$ it is:

$$|\{h \in \mathcal{H} \ : \ h(x) = h(y)\}| \leq \frac{|\mathcal{H}|}{m}$$

In other words, the class $\mathcal{H}$ is defined in such a way that a randomly-chosen hash function h from this set has a chance to make the distinct keys x and y to collide *no more than* $\frac{1}{m}$. This is exactly the basic property that we deployed when designing hashing with chaining (see the proof of Theorem 7.1). Figure 7.3 pictorially shows this concept.

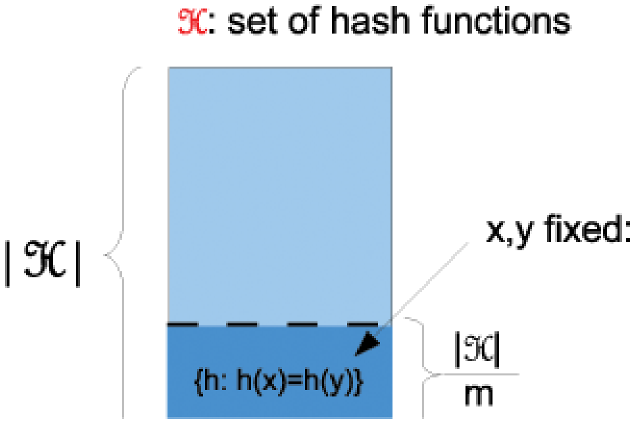


FIGURE 7.3: A schematic figure to illustrate the Universal Hash property.

It is interesting to observe that the definition of Universal Hashing can be extended with some slackness into the guarantee of probability of collision.

DEFINITION 7.2 Let c be a positive constant and $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into integers in $\{0, 1, ..., m-1\}$. $\mathcal{H}$ is said to be c**-universal** if, and only if, for all pairs of distinct keys $x, y \in U$ it is:

$$|\{h \in \mathcal{H} \ : \ h(x) = h(y)\}| \leq \frac{c|\mathcal{H}|}{m}$$

That is, for each pair of distinct keys, the number of hash functions for which there is a collision between this keys-pair is c times larger than what is guaranteed by universal hashing. The following theorem shows that we can use a universal class of hash functions to design a good hash table with chaining. This specifically means that Theorem 7.1 and its Corollaries 7.1–7.2 can be obtained by substituting the ideal Simple Uniform Hashing with Universal Hashing. This change will be effective in that, in the next section, we will define a real Universal class $\mathcal{H}$, thus making concrete all these mathematical ruminations.

THEOREM 7.2 *Let $T[0, m-1]$ be an hash table with chaining, and suppose that the hash function h is picked at random from a universal class $\mathcal{H}$. The expected length of the chaining lists in T, whichever is the input dictionary of keys S, is still no more than $1 + \alpha$, where α is the load factor n/m of the table T.*

Proof We note that the expectation here is over the choices of h in $\mathcal{H}$, and it does not depend on the distribution of the keys in S. For each pair of keys $x, y \in S$, define the indicator random variable I_{xy} which is 1 if these two keys collide according to a given h, namely $h(x) = h(y)$, otherwise it assumes the value 0. By definition of universal class, given the random choice of h, it is $P(I_{xy} = 1) = P(h(x) = h(y)) \leq 1/m$. Therefore we can derive $E[I_{xy}] = 1 \times P(I_{xy} = 1) + 0 \times P(I_{xy} = 0) = P(I_{xy} = 1) \leq 1/m$, where the average is computed over h's random choices.

Now we define, for each key $x \in S$, the random variable N_x that counts the number of keys other than x that hash to the slot $h(x)$, and thus collide with x. We can write N_x as $\sum_{\substack{y \in S \\ y \neq x}} I_{xy}$. By averaging, and applying the linearity of the expectation, we get $\sum_{\substack{y \in S \\ y \neq x}} E[I_{xy}] = (n-1)/m < \alpha$. By adding 1, because of x, the theorem follows. ■

We point out that the time bounds given for hashing with chaining are in expectation. This means that the average length of the lists in T is small, namely $O(\alpha)$, but there could be one or few lists which might be very long, possibly containing up to $\Theta(n)$ items. This satisfies Theorem 7.2 but is of course not a nice situation because it might occur sadly that the *distribution of the searches* privileges keys which belong to the very long lists, thus taking significantly more that the "average" time bound! In order to circumvent this problem, one should guarantee also a small upper bound on the length of the *longest list* in T. This can be achieved by putting some care when inserting items in the hash table.

THEOREM 7.3 *Let T be an hash table with chaining formed by m slots and picking an hash function from a universal class $\mathcal{H}$. Assume that we insert in T a dictionary S of $n = \Theta(m)$ keys, the expected length of the longest chain is $O(\frac{\log n}{\log \log n})$.*

Proof Let h be an hash function picked uniformly at random from $\mathcal{H}$, and let $Q(k)$ be the probability that exactly k keys of S are hashed by h to a particular slot of the table T. Given the universality of h, the probability that a key is assigned to a fixed slot is $\leq \frac{1}{m}$. There are $\binom{n}{k}$ ways to choose k keys from S, so the probability that a slot gets k keys is:

$$Q(k) = \binom{n}{k} \left(\frac{1}{m}\right)^k \left(\frac{m-1}{m}\right)^{n-k} < \frac{e^k}{k^k}$$

where the last inequality derives from Stirling's formula $k! > (k/e)^k$. We observe that there exists a constant $c < 1$ such that, fixed $m \geq 3$ and $k_0 = c \log m / \log \log m$, it holds $Q(k_0) < 1/m^3$.

Let us now introduce M as the length of the longest chain in T, and the random variable $N(i)$ denoting the number of keys that hash to slot i. Then we can write

$$\begin{aligned} P(M = k) &= P(\exists i : N(i) = k \text{ and } N(j) \leq k \text{ for } j \neq i) \\ &\leq P(\exists i : N(i) = k) \ \leq \ m\, Q(k) \end{aligned}$$

where the two last inequalities come, the first one, from the fact that probabilities are ≤ 1, and the second one, from the union bound applied to the m possible slots in T.

If $k \leq k_0$ we have $P(M = k_0) \leq mQ(k_0) < m\frac{1}{m^3} \leq 1/m^2$. If $k > k_0$, we can pick c large enough such that $k_0 > 3 > e$. In this case $e/k < 1$, and so $(e/k)^k$ decreases as k increases, tending to 0. Thus, we have $Q(k) < (e/k)^k \leq (e/k_0)^{k_0} < 1/m^3$, which implies again that $P(M = k) < 1/m^2$.
We are ready to evaluate the expectation of M:

$$E[M] = \sum_{k=0}^{n} k \times P(M = k) = \sum_{k=0}^{k_0} k \times P(M = k) + \sum_{k=k_0+1}^{n} k \times P(M = k) \tag{7.1}$$

$$\leq \sum_{k=0}^{k_0} k \times P(M = k) + \sum_{k=k_0+1}^{n} n \times P(M = k) \tag{7.2}$$

$$\leq k_0 \sum_{k=0}^{k_0} P(M = k) + n \sum_{k=k_0+1}^{n} P(M = k) \tag{7.3}$$

$$= k_0 \times P(M \leq k_0) + n \times P(M > k_0) \tag{7.4}$$

We note that $P(M \leq k_0) \leq 1$ and

$$Pr(M > k_0) = \sum_{k=k_0+1}^{n} P(M = k) < \sum_{k=k_0+1}^{n} (1/m^2) < n(1/n^2) = 1/n.$$

By combining together all these inequalities we can conclude that $E[M] \leq k_0 + n(1/n) = k_0 + 1 = O(\log m/ \log\log m)$, which is the thesis since we assumed $m = \Theta(n)$. ■

Two observations are in order at this point. The condition on $m = \Theta(n)$ can be easily guaranteed by applying the *doubling method* to the table T, as we showed in Theorem 7.2. The bound on the maximum chain length is on average, but it can be turned into worst case via a simple argument. We start by picking a random hash function $h \in \mathcal{H}$, hash every key of S into T, and see whether the condition on the length of the longest chain is at most twice the expected length $\log m/ \log\log m$. If so, we use T for the subsequent search operations, otherwise we pick a new function h and re-insert all items in T. A constant number of trials suffice to satisfy that bound[2], thus taking $O(n)$ construction time in expectation.

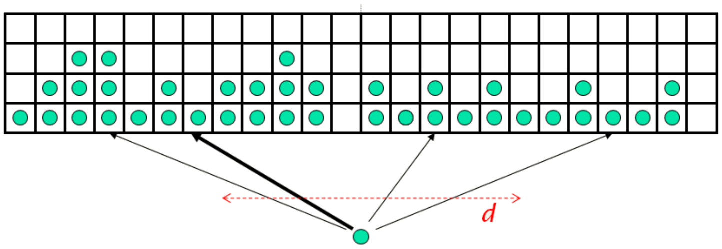


FIGURE 7.4: Example of d-left hashing with four subtables, four hash functions, and each table entry consisting of a bucket of a 4 slots.

Surprisingly enough, this result can be further improved by using two or more, say d, hash functions and d sub-tables $T_1, T_2, ..., T_d$ of the same size m/d, for a total space occupancy equal to the classic single hash table T. Each table T_i is indexed by a different hash function h_i ranging in $\{0, 1, \ldots, m/d - 1\}$. The specialty of this scheme resides in the implementation of the procedure `Insert(`k`)`: it tests the loading of the d slots $T_i[h_i(k)]$, and inserts k in the sparsest one. In the case of a tie, about slots' loading, the algorithm chooses the leftmost table, i.e. the one with minimum index i. For this reason this algorithmic scheme is also known as *d-left hashing*. The implementation of `Search(`k`)` follows consequently, we need to search all d lists $T_i[h_i(k)]$ because we do not know which were their loading when k was inserted, if any. The time cost for `Insert(`k`)` is $O(d)$ time, the time cost of `Search(`k`)` is given by the total length of the d searched lists. We can upper bound this length by d times the length of the longest list in T which, surprisingly, can be proved to be $O(\log\log n)$, when $d = 2$, and it is $\frac{\log\log n}{\log d} + O(1)$ for larger $d > 2$. This result has to be compared against the bound $O(\frac{\log n}{\log\log n})$ obtained for the case of a single hash function in Theorem 7.3. So, by just using one more hash function, we can get an exponential reduction in the search time: this sur-

[2] Just use the Markov bound to state that the longest list longer than twice the average may occur with probability $\leq 1/2$.

prising result is also known in the literature as *the power of two choices*, exactly because choosing between two slots the sparsest one allows to reduce exponentially the longest list.

As a corollary we notice that this result can be used to design a better hash table which does not use chaining-lists to manage collisions, thus saving the space of pointers and increasing locality of reference (hence less cache/IO misses). The idea is to allocate *small and fixed-size* buckets per each slot, as it is illustrated in Figure 7.4. We can use two hash functions and buckets of size $c \log\log n$, for some small $c > 1$. The important implication of this result is that even for just two hash functions there is a large reduction in the maximum list length, and thus search time.

7.3.1 Do universal hash functions exist?

The answer is positive and, surprisingly enough, universal hash functions can be easily constructed as we will show in this section for three classes of them. We assume, without loss of generality, that the table size m is a prime number and keys are integers represented as bit strings of $\log_2 |U|$ bits.[3] We let $r = \frac{\log_2 |U|}{\log_2 m}$ and assume that this is an integer. We decompose each key k in r parts, of $\log_2 m$ bits each, so $k = [k_0, k_1, ...k_{r-1}]$. Clearly each part k_i is an integer smaller than m, because it is represented in $\log_2 m$ bits. We do the same for a generic integer $a = [a_0, a_1, ..., a_{r-1}] \in [0, |U| - 1]$ used as the parameter that defines the universal class of hash functions $\mathcal{H}$ as follows: $h_a(k) = \sum_{i=0}^{r-1} a_i k_i \mod m$. The size of $\mathcal{H}$ is $|U|$, because we have one function per value of a.

THEOREM 7.4 *The class $\mathcal{H}$ that contains the following hash functions: $h_a(k) = \sum_{i=0}^{r-1} a_i k_i \mod m$, where m is prime and a is a positive integer smaller than $|U|$, is universal.*

Proof Suppose that x and y are two distinct keys which differ, hence, on at least one bit. For simplicity of exposition, we assume that a differing bit falls into the first part, so $x_0 \neq y_0$. According to Definition 7.1, we need to count how many hash functions make these two keys collide; or equivalently, how many a do exist for which $h_a(x) = h_a(y)$. Since $x_0 \neq y_0$, and we operate in arithmetic modulo a prime (i.e. m), the inverse $(x_0 - y_0)^{-1}$ must exist and it is an integer in the range $[1, |U| - 1]$, and so we can write:

$$\begin{aligned} h_a(x) = h_a(y) &\Leftrightarrow \sum_{i=0}^{r-1} a_i x_i \equiv \sum_{i=0}^{r-1} a_i y_i \pmod{m} \\ &\Leftrightarrow a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r-1} a_i(x_i - y_i) \pmod{m} \\ &\Leftrightarrow a_0 \equiv \left(-\sum_{i=1}^{r-1} a_i(x_i - y_i)\right)(x_0 - y_0)^{-1} \pmod{m} \end{aligned}$$

The last equation actually shows that, whatever is the choice for $[a_1, a_2, ..., a_{r-1}]$ (and they are m^{r-1}), there exists only one choice for a_0 (the one specified in the last line above) which causes x and y to collide. As a consequence, there are $m^{r-1} = |U|/m = |\mathcal{H}|/m$ choices for $[a_1, a_2, ..., a_{r-1}]$ that cause x and y to collide. So the Definition 7.1 of Universal Hash class is satisfied. ∎

[3] This is possible by pre-padding the key with 0, thus preserving its integer value.

It is possible to turn the previous definition holding for any table size m, thus not only for prime values. The key idea is to make a *double modular computation* by means of a large prime $p > |U|$, and a generic integer $m \ll |U|$ equal to the size of the hash table we wish to set up. We can then define an hash function parameterized in two values $a \geq 1$ and $b \geq 0$: $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$, and then define the family $\mathcal{H}_{p,m} = \bigcup_{a>0,b\geq 0}\{h_{a,b}\}$. It can be shown that $\mathcal{H}_{p,m}$ is a universal class of hash functions.

The above two definitions require r multiplications and r modulo operations. There are indeed other universal hashing which are faster to be computed because they rely only on operations involving power-of-two integers. As an example take $|U| = 2^h$, $m = 2^l < |U|$ and a be an odd integer smaller than $|U|$. Define the class $\mathcal{H}_{h,l}$ that contains the following hash functions: $h_a(k) = (ak \mod 2^h)$ $\texttt{div}$ 2^{h-l}. This class contains 2^{h-1} distinct functions because a is odd and smaller than $|U| = 2^h$. The following theorem presents the most important property of this class:

THEOREM 7.5 *The class $\mathcal{H}_{h,l} = \{\, h_a(k) = (ak \mod 2^h)\, \texttt{div}\, 2^{h-l}\}$, with $|U| = 2^h$ and $m = 2^l < |U|$ and a odd integer smaller than 2^h, is 2-universal because for any two distinct keys x and y, it is $P(h_a(x) = h_a(y)) \leq \dfrac{1}{2^{l-1}} = \dfrac{2}{m}$.*

Proof Without loss of generality let $x > y$ and define A as the set of possible values for a (i.e. a odd integer smaller than $2^h = |U|$). If there is a collision $h_a(x) = h_a(y)$, then we have:

$$\begin{aligned}
ax \mod 2^h\, \texttt{div}\, 2^{h-l} \;-\; ay \mod 2^h\, \texttt{div}\, 2^{h-l} &= 0\\
|ax \mod 2^h\, \texttt{div}\, 2^{h-l} \;-\; ay \mod 2^h\, \texttt{div}\, 2^{h-l}| &< 1\\
|ax \mod 2^h - ay \mod 2^h| &< 2^{h-l}\\
|a(x-y) \mod 2^h| &< 2^{h-l}
\end{aligned}$$

Set $z = x - y > 0$ (given that keys are distinct and $x > y$) and $z < |U| = 2^h$, it is $z \not\equiv 0(\mod 2^h)$ and $az \not\equiv 0(\mod 2^h)$ because a is odd, so we can write:

$$az \mod 2^h \in \{1, ..., 2^{h-l} - 1\} \cup \{2^h - 2^{h-l} + 1, ..., 2^h - 1\} \tag{7.5}$$

In order to estimate the number of $a \in A$ that satisfy this condition, we write z as $z'2^s$ with z' odd and $0 \leq s < h$. The odd numbers $a = 1, 3, 7, ..., 2^h - 1$ create a mapping $a \mapsto az' \mod 2^h$ that is a permutation of A because z' is odd. So, if we have the set $\{a2^s \mid a \in A\}$, a possible permutation is so defined: $a2^s \mapsto az'2^s \mod 2^h = az \mod 2^h$. Thus, the number of $a \in A$ that satisfy Eqn. 7.5 is the same as the number of $a \in A$ that satisfy:

$$a2^s \mod 2^h \in \{1, ..., 2^{h-l} - 1\} \cup \{2^h - 2^{h-l} + 1, ..., 2^h - 1\}$$

Now, $a2^s \mod 2^h$ is the number represented by the $h - s$ least significant bits of a, followed by s zeros. For example:

- If we take $a = 7$, $s = 1$ and $h = 3$:
  $7 * 2^1 \mod 2^3$ is in binary $111_2 * 10_2 \mod 1000_2$ that is equal to $1110_2 \mod 1000_2 = 110_2$. The result is represented by the $h - s = 3 - 1 = 2$ least significant bits of a, followed by $s = 1$ zeros.
- If we take $a = 7$, $s = 2$ and $h = 3$:
  $7*2^2 \mod 2^3$ is in binary $111_2*100_2 \mod 1000_2$ that is equal to $11100_2 \mod 1000_2 = 100_2$. The result is represented by the $h-s = 3-2 = 1$ least significant bits of a, followed by $s = 2$ zeros.

- If we take $a = 5$, $s = 2$ and $h = 3$:
  $5{*}2^2 \mod 2^3$ is in binary $101_2{*}100_2 \mod 1000_2$ that is equal to $10100_2 \mod 1000_2 = 100_2$. The result is represented by the $h-s = 3-2 = 1$ least significant bits of a, followed by $s = 2$ zeros.

So if $s \geq h - l$ there are no values of a that satisfy Eqn. 7.5, while for smaller s, the number of $a \in A$ satisfying that equation is at most 2^{h-l}. Consequently the probability of randomly choosing such a is at most $2^{h-l}/2^{h-1} = 1/2^{l-1}$. Finally, the universality of $\mathcal{H}_{h,l}$ follows immediately because $\frac{1}{2^{l-1}} < \frac{1}{m}$. ■

7.4 Perfect hashing, minimal, ordered!

The most known algorithmic scheme in the context of hashing is probably that of hashing with chaining, which sets $m = \Theta(n)$ in order to guarantee an average constant-time search; but that optimal time-bound is on average, as most students forget. This forgetfulness is not erroneous in absolute terms, because do indeed exist variants of hash tables that offer a constant-time *worst-case* bound, thus making hashing a competitive alternative to *tree-based* data structures and the *de facto* choice in practice. A crucial concept in this respect is *perfect hashing*, namely, a hash function which avoids collisions among the keys to be hashed (i.e. the keys of the dictionary to be indexed). Formally speaking,

DEFINITION 7.3 A hash function $h : U \to \{0, 1, \ldots, m-1\}$ is said to be *perfect* with respect to a dictionary S of keys if, and only if, for any pair of distinct keys $k', k'' \in S$, it is $h(k') \neq h(k'')$.

An obvious counting argument shows that it must be $m \geq |S| = n$ in order to make perfect-hashing possible. In the case that $m = n$, i.e. the minimum possible value, the perfect hash function is named *minimal* (shortly MPHF). A hash table T using an MPHF h guarantees $O(1)$ worst-case search time as well as no waste of storage space, because it has the size of the dictionary S (i.e. $m = n$) and keys can be directly stored in the table slots. Perfect hashing is thus a sort of "perfect" variant of direct-address tables (see Section 7.1), in the sense that it achieves constant search time (like those tables), but optimal linear space (unlike those tables).

A (minimal) perfect hash function is said to be *order preserving* (shortly OP(MP)HF) iff, $\forall k_i < k_j \in S$, it is $h(k_i) < h(k_j)$. Clearly, if h is also minimal, and thus $m = n$; then $h(k)$ returns the *rank* of the key in the ordered dictionary S. It goes without saying that, the property OP(MP)HF strictly depends onto the dictionary S upon which h has been built: by changing S we could destroy this property, so it is difficult, even if not impossible, to maintain this property under a *dynamic* scenario. In the rest of this section we will confine ourselves to the case of *static* dictionaries, and thus a fixed dictionary S.

The design of h is based upon three auxiliary functions h_1, h_2 and g, which are defined as follows:

- h_1 and h_2 are two universal hash functions from strings to $\{0, 1, \ldots, m'-1\}$ picked randomly from a universal class (see Section 7.3). They are not necessarily perfect, and so they might induce collisions among S's keys. The choice of m' impacts onto the efficiency of the construction, typically it is taken $m' = cn$, where $c > 1$, so spanning a range which is larger than the number of keys.
- g is a function that maps integers in the range $\{0, \ldots, m'-1\}$ to integers in the range $\{0, \ldots, n-1\}$. This mapping cannot be perfect, given that $m' \geq n$, and so some output

(a)	Term t	$h_1(t)$	$h_2(t)$	$h(t)$	(b)	x	$g(x)$
	body	1	6	0		0	0
	cat	7	2	1		1	5
	dog	5	7	2		2	0
	flower	4	6	3		3	7
	house	1	10	4		4	8
	mouse	0	1	5		5	1
	sun	8	11	6		6	4
	tree	11	9	7		7	1
	zoo	5	3	8		8	0
						9	1
						10	8
						11	6

TABLE 7.1 An example of an OPMPHF for a dictionary S of $n = 9$ strings which are in alphabetic order. Column $h(t)$ reports the lexicographic rank of each key; h is minimal because its values are in $\{0, \ldots, n-1\}$ and is built upon three functions: (a) two random hash functions $h_1(t)$ and $h_2(t)$, for $t \in S$; (b) a properly derived function $g(x)$, for $x \in \{0, 1, \ldots, m'\}$. Here $m' = 11 > 9 = n$.

values could be repeated. The function g is designed in a way that it *properly combines* the values of h_1 and h_2 in order to derive the OP(MP)HF h:

$$h(t) = (\, g(h_1(t)) + g(h_2(t))\,) \bmod n$$

The construction of g is obtained via an elegant randomized algorithm which deploys *paths in acyclic random graphs* (see below).

Examples for these three functions are given in the corresponding columns of Table 7.1. Although the values of h_1 and h_2 are randomly generated, the values of g are derived by a proper algorithm whose goal is to guarantee that the formula for $h(t)$ maps a string $t \in S$ to its lexicographic rank in S. It is clear that the evaluation of $h(t)$ takes constant time: we need to perform accesses to arrays h_1 and h_2 and g, plus two sums and one modular operation. The total required space is $O(m'+n) = O(n)$ whenever $m' = cn$. It remains to discuss the choice of c, which impacts onto the efficiency of the randomized construction of g and onto the overall space occupancy. It is suggested to take $c > 2$, which leads to obtain a successful construction in $\sqrt{\frac{m}{m-2n}}$ trials. This means about two trials by setting $c = 3$ (see [4]).

THEOREM 7.6 *An* OPMPHF *for a dictionary S of n keys can be constructed in $O(n)$ average time. The hash function can be evaluated in $O(1)$ time and uses $O(n)$ space (i.e. $O(n \log n)$ bits); both time and space bounds are worst case and optimal.*

Before digging into the technical details of this solution, let us make an important observation which highlights the power of OPMPHF. It is clear that we could assign the rank to each string by deploying a trie data structure (see Theorem 9.7), but this would incur two main limitations: (i) rank assignment would need a trie search operation which would incur $\Theta(s)$ I/Os, rather than $O(1)$; (ii) space occupancy would be linear in the total dictionary length, rather than in the total dictionary cardinality. The reason is that, the OPMPHF's machinery does not need the string storage thus allowing to pay space proportional to the number of strings rather than their length; but on the other hand, OPMPHF does not allow to compute the rank of strings *not* in the dictionary, an operation which is instead supported by tries via the so called *lexicographic search* (see Chapter 9).

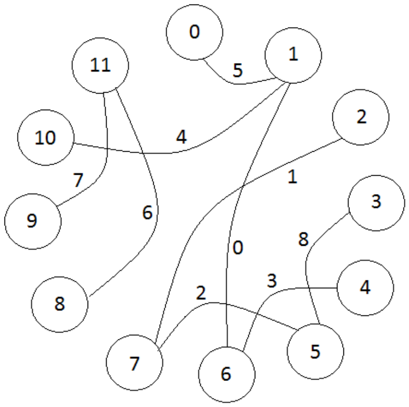


FIGURE 7.5: Graph corresponding to the dictionary of strings S and to the two functions h_1 and h_2 of Table 7.1.

We are left with detailing the algorithm that computes the function g, which will be actually implemented as an array of m' positions storing values bounded by n (see above). So g-array will occupy a total of $O(m' \log_2 n)$ bits. This array must have a quite peculiar feature: its entries have to be defined in a way that the computation

$$h(t) = (\, g(h_1(t)) + g(h_2(t))\,) \bmod n$$

returns the rank of term t in S. Surprisingly enough, the computation of g is quite simple and consists of building an undirected graph $G = (V, E)$ with m' nodes labeled $\{0, 1, \ldots, m' - 1\}$ (the same range as the co-domain of h_1 and h_2, and the domain of g) and n edges (as many as the keys in the dictionary) defined according to $(h_1(t), h_2(t))$ labeled with the *desired* value $h(t)$, for each dictionary string $t \in S$. It is evident that the topology of G depends only on h_1 and h_2, and it is exactly what it is called a *random graph*.[4]

Figure 7.5 shows an example of graph G constructed according to the setting of Table 7.1. Take the string t = body for which $h_1(t) = 1$ and $h_2(t) = 6$. The lexicographic rank of body in the dictionary S is $h(t) = 0$ (it is the first string), which is then the value labeling edge $(1, 6)$. We have to derive $g(t)$ so that 0 must be turn to be equal to $(\, g(1) + g(6)\,) \bmod 9$. Of course there are some correct values for entries $g(1)$ and $g(6)$ (e.g. $g(1) = 0$ and $g(6) = 0$), but these values should be correct for all terms t whose edges are incident onto the nodes 1 and 6. Because these edges refer to terms whose g's computation depends on $g(1)$ or $g(6)$. In this respect, the structure of G is useful to drive the instantiation of g's values, as detailed by the algorithm LabelAcyclicGraph(G).

The key algorithmic idea is that, if the graph originated by h_1 and h_2 is acyclic, then it can be decomposed into paths. If we assume that the first node, say v_0, of every path takes value 0, as indeed we execute LabelFrom(v_0, 0) in LabelAcyclicGraph(G), then all other nodes v_i subsequently traversed in this path will have value undef for $g(v_i)$ and thus this entry can be easily set by solving the following equation with respect to $g(v_i)$:

$$h(v_{i-1}, v_i) = (\, g(v_{i-1}) + g(v_i)\,) \bmod n$$

[4] The reader should be careful that the role of letters n and m' is exchanged here with respect to the traditional graph notation in which n refers to number of nodes and m' refers to number of edges.

Algorithm 7.1 Procedure `LabelAcyclicGraph`(G)

```
for $v \in V$ do
    $g[v]$ = undef
end for
for $v \in V$ do
    if $g[v]$ = undef then
        LabelFrom($v, 0$)
    end if
end for
```

which actually means to compute:

$$g(v_i) = (\,h(v_{i-1}, v_i) - g(v_{i-1})\,) \bmod n$$

Algorithm 7.2 Procedure `LabelFrom`(v, c)

```
if $g[v] \neq$ undef then
    if $g[v] \neq c$ then
        return - the graph is cyclic; STOP
    end if
end if
$g[v] = c$
for $u \in$ Adj$[v]$ do
    LabelFrom($u, h(v, u) - g[v] \mod n$)
end for
```

This is exactly what the algorithm `LabelFrom`(v, c) does. It is natural at this point to ask whether these algorithms always find a good assignment to function g or not. It is not difficult to convince ourselves that they return a solution only if the input graph G is acyclic, otherwise they stop. In this latter case, we have to rebuild the graph G, and thus the hash functions h_1 and h_2 by drawing them from the universal class.

What is the likelihood of building an acyclic graph ? This question is quite relevant since the technique is useful only if this *probability of success* is large enough to need few rebuilding steps. According to Random Graph Theory [4], if $m' \leq 2n$ this probability is almost equal to 0; otherwise, if $m' > 2n$ then it is about $\sqrt{\frac{m'-2n}{m'}}$, as we mentioned above. This means that the average number of graphs we will need to build before finding an acyclic one is: $\sqrt{\frac{m'}{m'-2n}}$; which is a constant number if we take $m' = \Theta(n)$. So, on average, the algorithm `LabelAcyclicGraph`(G) builds $\Theta(1)$ graphs of $m' + n = \Theta(n)$ nodes and edges, and thus it takes $\Theta(n)$ time to execute those computations. Space usage is $m' = \Theta(n)$.

In order to reduce the space requirements, we could resort multi-graphs (i.e. graphs with multiple edges between a pair of nodes) and thus use three hash functions, instead of two:

$$h\,(t) = (g\,(h_1\,(t)) + g\,(h_2\,(t)) + g\,(h_3\,(t))) \bmod n$$

We conclude this section by a running example that executes `LabelAcyclicGraph`(G) on the graph in Figure 7.5.

1. Select node 0, set $g(0) = 0$ and start to visit its neighbors, which in this case is just the node 1.
2. Set $g(1) = (h(0,1) - g(0))\texttt{mod}9 = (5 - 0)\texttt{mod}9 = 5$.
3. Take the unvisited neighbors of 1: 6 and 10, and visit them recursively.
4. Select node 6, set $g(6) = (h(1,6) - g(1))\texttt{mod}9 = (0 - 5)\texttt{mod}9 = 4$.
5. Take the unvisited neighbors of 6: 4 and 10, the latter is already in the list of nodes to be explored.
6. Select node 10, set $g(10) = (h(1,10) - g(1))\texttt{mod}9 = (4 - 5)\texttt{mod}9 = 8$.
7. No unvisited neighbors of 10 do exist.
8. Select node 4, set $g(4) = (h(4,6) - g(6))\texttt{mod}9 = (3 - 4)\texttt{mod}9 = 8$.
9. No unvisited neighbors of 4 do exist.
10. No node is left in the list of nodes to be explored.
11. Select a new starting node, for example 2, set $g(2) = 0$, and select its unvisited neighbor 7.
12. Set $g(7) = (h(2,7) - g(2))\texttt{mod}9 = (1 - 0)\texttt{mod}9 = 1$, and select its unvisited neighbor 5.
13. Set $g(5) = (h(7,5) - g(7))\texttt{mod}9 = (2 - 1)\texttt{mod}9 = 1$, and select its unvisited neighbor 3.
14. Set $g(3) = (h(3,5) - g(5))\texttt{mod}9 = (8 - 1)\texttt{mod}9 = 7$.
15. No node is left in the list of nodes to be explored.
16. Select a new starting node, for example 8, set $g(8) = 0$, and select its unvisited neighbor 11.
17. Set $g(11) = (h(8,11) - g(8))\texttt{mod}9 = (6 - 0)\texttt{mod}9 = 6$, and select its unvisited neighbor 9.
18. Set $g(9) = (h(11,9) - g(11))\texttt{mod}9 = (7 - 6)\texttt{mod}9 = 1$.
19. No node is left in the list of nodes to be explored.
20. Since all other nodes are isolated, their g's value is set to 0;

It goes without saying that, if S undergoes some insertions, we possibly have to rebuild $h(t)$. Therefore, all of this works for a static dictionary S.

7.5 A simple perfect hash table

If ordering and minimality (i.e. $h(t) < n$) is not required, then the design of a (static) perfect hash function is simpler. The key idea is to use a *two-level hashing scheme* with universal hashing functions at each level. The first level is essentially hashing with chaining, where the n keys are hashed into m slots using a universal hash function h; but, unlike chaining, every entry $T[j]$ points to a **secondary hash table** T_j which is addressed by another specific universal hash function h_j. By choosing h_j carefully, we can guarantee that there are no collisions at this secondary level. This way, the search for a key k will consist of two table accesses: one at T according to h, and one at some T_i according to h_i, given that $i = h(k)$. For a total of $O(1)$ time complexity in the worst case.

The question now is to guarantee that: (i) hash functions h_j are perfect and thus elicit no collisions among the keys mapped by h into $T[j]$, and (ii) the total space required by table T and all sub-tables T_j is the optimal $O(n)$. The following theorem is crucial for the following arguments.

THEOREM 7.7 *If we store q keys in a hash table of size $w = q^2$ using a universal hash function, then the expected number of collisions among those keys is less than* $1/2$. *Consequently, the probability to have a collision is less than* $1/2$.

Proof In a set of q elements there are $\binom{q}{2} < q^2/2$ pairs of keys that may collide; if we choose the function h from a universal class, we have that each pair collides with probability $1/w$. If we set $w = q^2$ the expected number of collisions is $\binom{q}{2}\frac{1}{w} < q^2/(2q^2) = \frac{1}{2}$. The probability to have at least one collision can be upper bounded by using the Markov inequality (i.e. $P(X \geq tE[X]) \leq 1/t$), with X expressing the number of collisions and by setting $t = 2$. ■

We use this theorem in two ways: we will guarantee (i) above by setting the size m_j of hash table T_j as n_j^2 (the square of the number of keys hashed to $T[j]$); we will guarantee (ii) above by setting $m = n$ for the size of table T. The former setting ensures that every hash function h_j is perfect, by just two re-samples on average; the latter setting ensures that the total space required by the sub-tables is $O(n)$ as the following theorem formally proves.

THEOREM 7.8 *If we store n keys in a hash table of size $m = n$ using a universal hash function h, then the expected size of all sub-tables T_j is less than $2n$: in formula, $E[\sum_{j=0}^{m-1} n_j^2] < 2n$ where n_j is the number of keys hashing to slot j and the average is over the choices of h in the universal class.*

Proof Let us consider the following identity: $a^2 = a + 2\binom{a}{2}$ which is true for any integer $a > 0$. We have:

$$\begin{aligned} E[\sum_{j=0}^{m-1} n_j^2] &= E[\sum_{j=0}^{m-1}(n_j + 2\binom{n_j}{2})] \\ &= E[\sum_{j=0}^{m-1} n_j] + 2E[\sum_{j=0}^{m-1}\binom{n_j}{2}] \\ &= n + 2E[\sum_{j=0}^{m-1}\binom{n_j}{2}] \end{aligned}$$

The former term comes from the fact that $\sum_{j=0}^{m-1} n_j$ equals the total number of items hashed in the secondary level, and thus it is the total number n of dictionary keys. For the latter term we notice that $\binom{n_j}{2}$ accounts for the number of collisions among the n_j keys mapped to $T[j]$, so that $\sum_{j=0}^{m-1}\binom{n_j}{2}$ equals the number of collisions induced by the primary-level hash function h. By repeating the argument adopted in the proof of Theorem 7.7 and using $m = n$, the expected value of this sum is at most $\binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2}$. Summing these two terms we derive that the total space required by this two-level hashing scheme is bounded by $n + 2\frac{n-1}{2} = 2n - 1$. ■

It is important to observe that every hash function h_j is independent on the others, so that if it generates some collisions among the n_j keys mapped to $T[j]$, it can be re-generated without influencing the other mappings. Analogously if at the first level h generates more than zn collisions, for some large constant z, then we can re-generate h. These theorems ensure that the average number of re-generations is a small constant per hash function. In the following subsections we detail insertion and searching with perfect hashing.

Create a perfect hash table

In order to create a perfect hash table from a dictionary S of n keys we proceed as indicated in the pseudocode of Figure 7.3. Theorem 7.8 ensures that the probability to extract a good function from the family $H_{p,m}$ is at least $1/2$, so we have to repeat the extraction process (at the first level) an average number of 2 times to guarantee a successful extraction, which actually means $L < 2n$. At the second level Theorem 7.7 and the setting $m_j = n_j^2$ ensure that the probability to have a collision in table T_j because of the hash function h_j is lower than $1/2$. SO, again, we need on average two extractions for h_j to construct a perfect hash table T_j.

Algorithm 7.3 Creating a Perfect Hash Table

```
 1: Choose a universal hash function $h$ from the family $H_{p,m}$;
 2: for $j = 0, 1, \ldots, m-1$ do
 3:     $n_j = 0, S_j = \emptyset$;
 4: end for
 5: for $k \in S$ do
 6:     $j = h(k)$;
 7:     Add $k$ to set $S_j$;
 8:     $n_j = n_j + 1$;
 9: end for
10: Compute $L = \sum_{j=0}^{m-1} (n_j)^2$;
11: if $L \geq 2n$ then
12:     Repeat the algorithm from Step 1;
13: end if
14: for $j = 0, 1, \ldots, m-1$ do
15:     Construct table $T_j$ of size $m_j = (n_j)^2$;
16:     Choose hash function $h_j$ from class $H_{p,m_j}$;
17:     for $k \in S_j$ do
18:         $i = h_j(k)$;
19:         if $T_j[i] \neq$ NULL then
20:             Destroy $T_j$ and repeat from step 15;
21:         end if
22:         $T_j[i] = k$;
23:     end for
24: end for
```

Searching keys with perfect hashing

Searching for a key k in a perfect hash table T takes just two memory accesses, as indicated in the pseudocode of Figure 7.4.

With reference to Figure 7.6, let us consider the successful search for the key $98 \in S$. It is $h(98) = ((2 \times 98 + 42) \bmod 101) \bmod 11 = 3$. Since T_3 has size $m_3 = 4$, we apply the second-level hash function $h_3(98) = ((4 \times 98 + 42) \bmod 101) \bmod 4 = 2$. Given that $T_3(2) = 98$, we have found the searched key.

Let us now consider the unsuccessful search for the key $k = 8 \notin S$. Since it is $h(8) = ((2 \times 8 + 42) \bmod 101) \bmod 11 = 3$, we have to look again at the table T_3 and compute $h_3(8) = ((4 \times 8 + 42) \bmod 101) \bmod 4 = 2$. Given that $T_3(2) = 19$, we conclude that the key $k = 8$ is not in the dictionary.

7.6 Cuckoo hashing

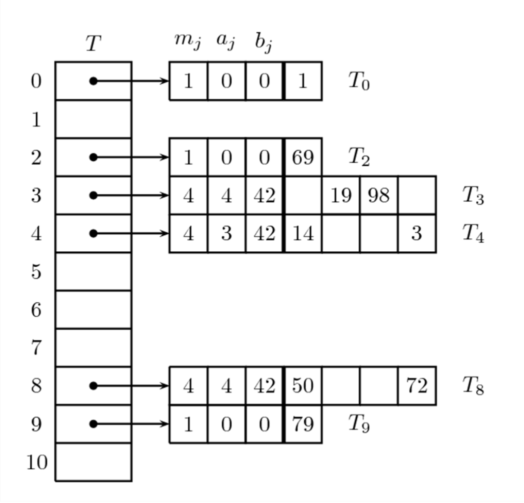


FIGURE 7.6: Creating a perfect hash table over the dictionary $S = \{98, 19, 14, 50, 1, 72, 79, 3, 69\}$, with $h(k) = ((2k + 42) \bmod 101) \mod 11$. Notice that $L = 1 + 1 + 4 + 4 + 4 + 1 = 15 < 2 \times 9$ and, at the second level, we use the same hash function changing only the table size for the modulo computation. There no collision occur.

When the dictionary is dynamic a different hashing scheme has to be devised, an efficient and elegant solution is the so called **cuckoo hashing**: it achieves constant time in updates, on average, and constant time in searches, in the worst case. The only drawback of this approach is that it makes use of two hash functions that are $O(\log n)$-independent (new results in the literature have significantly relaxed this requirement [1] but we stick on the original scheme for its simplicity). In pills, cuckoo hashing combines the multiple-choice approach of d-left hashing with the ability to move elements. In its simplest form, cuckoo hashing consists of two hash functions h_1 and h_2 and one table T of size m. Any key k is stored either at $T[h_1(k)]$ or at $T[h_2(k)]$, so that searching and deleting operations are trivial: we need to look for k in both those entries, and eventually remove it. Inserting a key is a little bit more tricky in that it can trigger a *cascade* of key moves in the table. Suppose a new key k has to be inserted in the dictionary, according to the Cuckoo scheme it has to be inserted either at position $h_1(k)$ or at position $h_2(k)$. If one of these locations in T is empty (if both are, $h_1(k)$ is chosen), the key is stored at that position and the insertion process is completed. Otherwise, both entries are occupied by other keys, so that we have to create room for k by evicting one of the two keys stored in those two table entries. Typically, the key y stored in $T[h_1(k)]$ is evicted and replaced with k. Then, y plays the role of k and the insertion process is repeated.

There is a warning to take into account at this point. The key y was stored in $T[h_1(k)]$, so that $T[h_i(y)] = T[h_1(k)]$ for either $i = 1$ or $i = 2$. This means that if both positions $T[h_1(y)]$ and $T[h_2(y)]$ are occupied, the key to be evicted cannot be chosen from the entry that was storing $T[h_1(k)]$ because it is k, so this would induce a trivial infinite cycle of evictions over this entry between keys k and y. The algorithm therefore is careful to always avoid to evict the previously inserted key. Nevertheless cycles may arise (e.g. consider the trivial case in which $\{h_1(k), h_2(k)\} = \{h_1(y), h_2(y)\}$) and they can be of arbitrary length, so that the algorithm must be careful in defining an *efficient escape condition* which detects those situations, in which case it re-sample the two hash functions and re-hash all dictionary keys. The key property, proved in Theorem 7.9, will be to show that cycles occurs with

Algorithm 7.4 Procedure `Search(`k`)` in a Perfect Hash Table T

```
Let $h$ and $h_j$ be the universal hash functions defining $T$;
Compute $j = h(k)$;
if $T_j$ is empty then
    Return false; // $k \notin S$
end if
Compute $i = h_j(k)$;
if $T_j[i] =$ NULL then
    Return false; // $k \notin S$
end if
if $T_j[i] \neq k$ then
    Return false; // $k \notin S$
end if
Return true; // $k \in S$
```

bounded probability, so that the $O(n)$ cost of re-hashing can be amortized, by charging $O(1)$ time per insertion (Corollary 7.4).

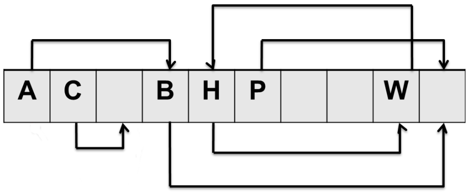


FIGURE 7.7: Graphical representation of cuckoo hashing.

In order to analyze this situation it is useful to introduce the so called *cuckoo graph* (see Figure 7.7), whose nodes are entries of table T and edges represent dictionary keys by connecting the two table entries where these keys can be stored. Edges are directed to keep into account where a key is stored (source), and where a key could be alternatively stored (destination). This way the cascade of evictions triggered by key k traverses the nodes (table entries) laying on a directed path that starts from either node (entry) $h_1(k)$ or node $h_2(k)$. Let us call this path the *bucket* of k. The bucket reminds the list associated to entry $T[h(k)]$ in hashing with chaining (Section 7.2), but it might have a more complicated structure because the cuckoo graph can have cycles, and thus this path can form loops as it occurs for the cycle formed by keys `W` and `H`.

For a more detailed example of insertion process, let us consider again the cuckoo graph depicted in Figure 7.7. Suppose to insert key `D` into our table, and assume that $h_1(D) = 4$ and $h_2(D) = 1$, so that D evicts either `A` or `B` (Figure 7.8). We put `D` in table entry 1, thereby evicting `A` which tries to be stored in entry 4 (according to the directed edge). In turn, `A` evicts `B`, stored in 4, which is moved to the last location of the table as its possible destination. Since such a location is free, `B` goes there and the insertion process is successful and completed. Let us now consider the insertion of key `F`, and assume two cases: $h_1(F) = 2$ and $h_2(F) = 5$ (Figure 7.9.a), or $h_1(F) = 4$ and $h_2(F) = 5$ (Figure 7.9.b). In the former case the insertion is successful: `F` causes `C` to be evicted, which in turn finds the third location empty. It is interesting to observe that, even if we check first $h_2(F)$, then the insertion is still successful: F causes H to be evicted, which causes W to be evicted, which in turn causes again F to be evicted. We found a cycle which nevertheless does not induces an infinite loop, in fact in this case the eviction of F leads to check its second possible location, namely $h_1(F) = 2$, which evicts C that is then stored in the third location (currently empty). Consequently the existence of a cycle does not imply an unsuccessful search; said this, in the following, we will compute the probability of the existence of a cycle as an upper bound to the probability of an unsuccessful search. Conversely, the case of an unsuccessful insertion occurs in Figure 7.9.b where there are two cycles F-H and A-B which gets glued together by the insertion of the key F. In this case the keys flow from one cycle to the other without stopping. As a consequence, the insertion algorithm must be designed in order to check whether the traversal of the cuckoo graph ended up in an infinite loop: this is done *approximately* by bounding the number of eviction steps.

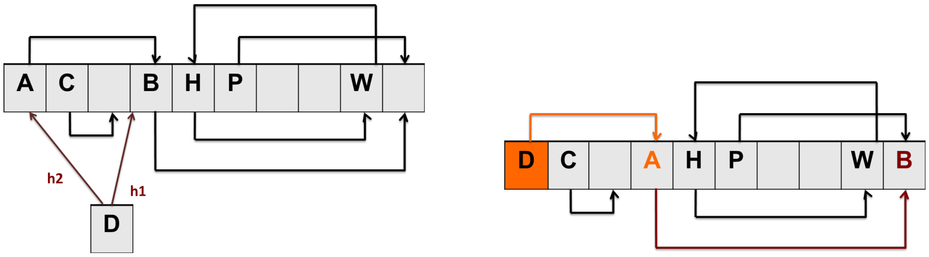


FIGURE 7.8: Inserting the key D: (left) the two entry options, (right) the final configuration.

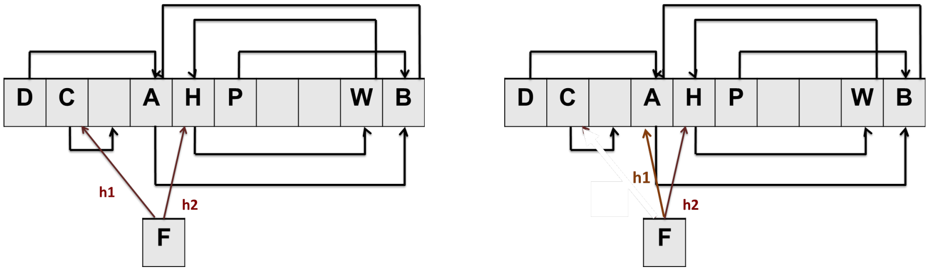


FIGURE 7.9: Inserting the key F: (left) successful insertion, (right) unsuccessful insertion.

7.6.1 An analysis

Querying and deleting a key k takes constant time, only two table entries have to be checked. The case of insertion is more complicated because it has necessarily to take into account the formation of paths in the (random) cuckoo graph. In the following we consider an *undirected* cuckoo graph, namely one in which edges are not oriented, and observe that a key y is in the bucket of another key x only if there is a path between one of the two positions (nodes) of x and one of the two positions (nodes) of y in the cuckoo graph. This relaxation allows to easy the bounding of the probability of the existence of these paths (recall that m is the table size and n is the dictionary size):

THEOREM 7.9 *For any entries i and j and any constant $c > 1$, if $m \geq 2cn$, then the probability that in the undirected cuckoo graph there exists a shortest path from i to j of length $L \geq 1$ is at most c^{-L}/m.*

Proof We proceed by induction on the path length L. The base case $L = 1$ corresponds to the existence of the undirected edge (i, j); now, every key can generate that edge with probability no more than $\leq 2/m^2$, because edge is undirected and $h_1(k)$ and $h_2(k)$ are uniformly random choices among the m table entries. Summing over all n dictionary keys, and recalling that $m \geq 2cn$, we get the bound $\sum_{k \in S} 2/m^2 = 2n/m^2 \leq c^{-1}/m$.

For the inductive step we must bound the probability that there exists a path of length $L > 1$, but no path of length less than L connects i to j (or vice versa). This occurs only if, for some table entry h, the following two conditions hold:

- there is a shortest path of length $L - 1$ from i to z (that clearly does not go through j);
- there is an edge from z to j.

By the inductive hypothesis, the probability that the first condition is true is bounded by $c^{-(L-1)}/m = c^{-L+1}/m$. The probability of the second condition has been already computed and it is at most $c^{-1}/m = 1/cm$. So the probability that there exists such a path (passing through z) is $(1/cm) * (c^{-L+1}/m) = c^{-L}/m^2$. Summing over all m possibilities for the table entry z, we get that the probability of a path of length L between i and j is at most c^{-L}/m. ■

In other words, this Theorem states that if the number m of nodes in the cuckoo graph is sufficiently large compared to the number n of edges (i.e. $m \geq 2cn$), there is a low probability that any two nodes i and j are connected by a path, thus fall in the same bucket, and hence participate in a cascade of evictions. Very significant is the case of a constant-length path $L = O(1)$, for which the probability of occurrence is $O(1/m)$. This means that, even for this restricted case, the probability of a large bucket is small and thus the probability of a not-constant number of evictions is small. We can related this probability to the *collision probability* in hashing with chaining. We have therefore proved the following:

THEOREM 7.10 *For any two distinct keys x and y, the probability that x hashes to the same bucket of y is $O(1/m)$.*

Proof If x and y are in the same bucket, then there is a path of some length L between one node in $\{h_1(x), h_2(x)\}$ and one node in $\{h_1(y), h_2(y)\}$. By Theorem 7.9, this occurs with probability at most $4 \sum_{L=1}^{\infty} c^{-L}/m = \frac{4}{c-1}/m = O(1/m)$, as desired. ■

What about rehashing? How often do we have to rebuild table T? Let us consider a sequence of operations involving ϵn insertions, where ϵ is a small constant, e.g. $\epsilon = 0.1$, and assume that the table size is sufficiently large to satisfy the conditions imposed in the previous theorems, namely $m \geq 2cn + 2c(\epsilon n) = 2cn(1 + \epsilon)$. Let S' be the final dictionary in which all ϵn insertions have been performed. Clearly, there is a re-hashing of T only if some key insertion induced an infinite loop in the cuckoo graph. In order to bound this probability we consider the final graph in which all keys S' have been inserted, and thus all cycles induced by their insertions are present. This graph consists of m nodes and $n(1 + \epsilon)$ keys. Since we assumed $m \geq 2cn(1 + \epsilon)$, according to the Theorem 7.9, any given position (node) is involved in a cycle (of any length) if it is involved in a path (of any length) that starts and end at that position: the probability is at most $\sum_{L=1}^{\infty} c^{-L}/m$. Thus, the probability that there is a cycle of any length involving any table entry can be bounded by summing over all m

table entries: namely, $m \sum_{L=1}^{\infty} c^{-L}/m = \frac{1}{c-1}$. As we observed previously, this is an upper bound to the probability of an unsuccessful insertion given that the presence of a cycle does not necessarily imply an infinite loop for an insertion.

COROLLARY 7.3 By setting $c = 3$, and taking a cuckoo table of size $m \geq 6n(1 + \epsilon)$, the probability for the existence of a cycle in the cuckoo graph of the final dictionary S' is at most $1/2$.

Therefore a constant number of re-hashes are enough to ensure the insertion of ϵn keys in a dictionary of size n. Given that the time for one rehashing is $O(n)$ (we just need to compute two hashes per key), the expected time for all rehashing is $O(n)$, which is $O(1/\epsilon)$ per insertion.

COROLLARY 7.4 By setting $c > 2$, and taking a cuckoo table of size $m \geq 2cn(1 + \epsilon)$, the cost for inserting ϵn keys in a dictionary of size n by cuckoo hashing is constant expected amortized. Namely, expected with respect to the random selection of the two universal hash functions driving the cuckoo hashing, and amortised over the $\Theta(n)$ insertions.

In order to make the algorithm works for every n and ϵ, we can adopt the same idea sketched for hashing with chaining and called *global rebuilding technique*. Whenever the size of the dictionary becomes too small compared to the size of the hash table, a new, smaller hash table is created; conversely, if the hash table fills up to its capacity, a new, larger hash table is created. To make this work efficiently, the size of the hash table is increased or decreased by a constant factor (larger than 1), e.g. doubled or halved.

The cost of rehashing can be further reduced by using a very small amount (i.e. constant) of extra-space, called a *stash*. Once a failure situation is detected during the insertion of a key k (i.e. k incurs in a loop), then this key is stored in the stash (without rehashing). This reduces the rehashing probability to $\Theta(1/n^{s+1})$, where s is the size of the stash. The choice of parameter s is related to some structural properties of the cuckoo graph and of the universal hash functions, which are too involved to be commented here (for details see [1] and refs therein).

7.7 Bloom filters

There are situations in which the universe of keys is very large and thus every key is long enough to take a lot of space to be stored. Sometimes it could even be the case that the storage of table pointers, taking $(n+m)\log n$ bits, is much smaller than the storage of the keys, taking $n \log_2 |U|$ bits. An example is given by the dictionary of URLs managed by crawlers in search engines; maintaining this dictionary in internal memory is crucial to ensure the fast operations over those URLs required by crawlers. However URLs are thousands of characters long, so that the size of the indexable dictionary in internal memory could be pretty much limited if whole URLs should have to be stored. And, in fact, crawlers do not use neither cuckoo hashing nor hashing with chaining but, rather, employ a simple and randomised, yet efficient, data structure named *Bloom filter*. The crucial property of Bloom filters is that keys are *not explicitly stored*, only a small fingerprint of them is, and this induces the data structure to make a *one-side error* in its answers to membership queries whenever the queried key is not in the currently indexed dictionary. The elegant solution proposed by Bloom filters is that those errors can be controlled, and indeed their probability *decreases exponentially* with the size of the fingerprints of the dictionary keys. Practically speaking tens of bits (hence, few bytes) per fingerprint are enough to guarantee tiny error probabilities[5] and succinct space occupancy, thus making this solution much appealing in a big-data context. It is useful at this point recall the *Bloom filter principle*: "Wherever a list or set is used, and space is a consideration, a Bloom filter should be considered. When using a Bloom filter, consider the potential effects of false positives".

Let $S = \{x_1, x_2, ..., x_n\}$ be a set of n keys and B a bit vector of length m. Initially, all bits in B are set to 0. Suppose we have r universal hash functions $h_i : U \longrightarrow \{0, ..., m-1\}$, for $i = 1, ..., r$. As anticipated above, every key k is not represented explicitly in B but, rather, it is fingerprinted by setting r bits of B to 1 as follows: $B[h_i(k)] = 1, \forall\, 1 \leq i \leq r$. Therefore, inserting a key in a Bloom filter requires $O(r)$ time, and sets at most r bits (possibly some hashes may collide). For searching, we claim that a key y is in S if $B[h_i(y)] = 1, \forall\, 1 \leq i \leq r$. Searching costs $O(r)$, as well as inserting. In the example of Figure 7.10, we can assert that $y \notin S$, since three bits are set to 1 but the last checked bit $B[h_4(y)]$ is zero.

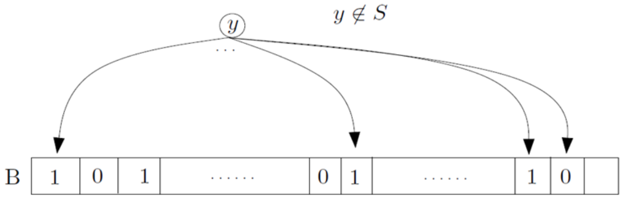


FIGURE 7.10: Searching key y in a Bloom filter.

Clearly, if $y \in S$ the Bloom filter correctly detects this; but it might be the case that $y \notin S$ and nonetheless all r bits checked are 1 because of the setting due to other hashes and keys. This is called *false positive* error, because it induces the Bloom filter to return a positive but erroneous answer to a membership query. It is therefore natural to ask for the probability of a false-positive error, which can be proved to be bounded above by a surprisingly simple formula.

The probability that the insertion of a key $k \in S$ has left null a bit-entry $B[j]$ equals the probability that the r independent hash functions $h_i(k)$ returned an entry different of j, which is $\left(\frac{m-1}{m}\right)^r \approx e^{-\frac{r}{m}}$. After the insertion of all n dictionary keys, the probability that $B[j]$ is still null can be then bounded by $p_0 \approx \left(e^{-\frac{r}{m}}\right)^n = e^{-\frac{rn}{m}}$ by assuming independencies among those hash functions.[6] Hence the probability of a false-positive error (or, equivalently, the false positive rate) is the probability that all r bits checked for a key not in the current dictionary are set to 1, that is:

$$p_{err} = (1 - p_0)^r \approx \left(1 - e^{-\frac{rn}{m}}\right)^r$$

Not surprisingly the error probability depends on the three parameters that define the Bloom filter's structure: the number r of hash functions, the number n of dictionary keys, and the number m of bits in the binary array B. It's interesting to notice that the fraction $f = m/n$ can be read as the average number of bits per dictionary key allocated in B, hence the fingerprint size f. The larger is

[5] One could object that, errors anyway might occur. But programmers counteract by admitting that these errors can be made smallers than hardware/network errors in data centers or PCs. So they can be neglected!

[6] A more precise analysis is possible, but much involved and without changing the final result, so that we prefer to stick on this simpler approach.

f the smaller is the error probability p_{err}, but the larger is the space allocated for B. We can optimize p_{err} according to m and n, by computing the first-order derivative and equalling it to zero: this gets $r = \frac{m}{n} \ln 2$. It is interesting to observe that for this value of r the probability a bit in B gets null value is $p_0 = 1/2$; which actually means that the array is half filled by 1s and half by 0s. And indeed this result could not be different: a larger r induces more 1s in B and thus a larger probability of positive errors, a lower r induces more 0s in B and thus a larger probability of correct answers: the correct choice of r falls in the middle! For this value of $r = \frac{m}{n} \ln 2$, we have $p_{err} = (0.6185)^{m/n}$ which decreases exponentially with the fingerprint size $f = m/n$. Figure 7.11 reports the false positive rate as a function of the number r of hashes for a Bloom-filter designed to use $m = 32n$ bits of space, hence a fingerprint of $f = 32$ bits per key. By using 22 hash functions we can minimize the false positive rate to less than $1.E - 6$. However, we also note that adding one more hash function does not significantly decreases the error rate when $r \geq 10$.

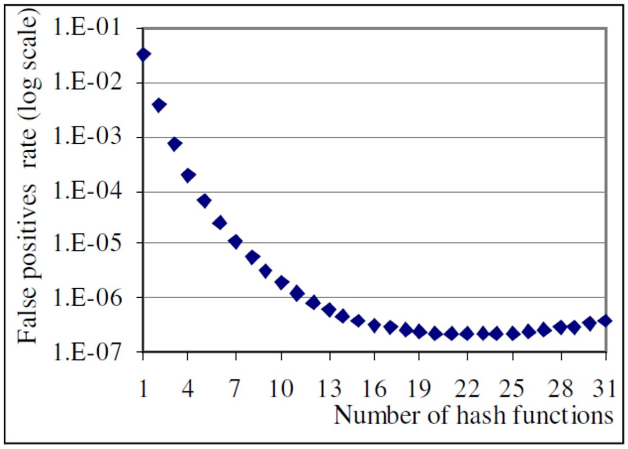


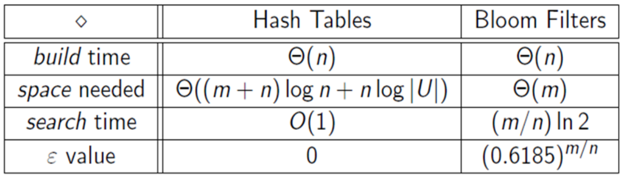
FIGURE 7.11: Plotting p_{err} as a function of the number of hashes.

It is natural now to derive the size m of the B-array whenever r is fixed to its optimal value $\frac{m}{n} \ln 2$, and n is the number of current keys in the dictionary. We obtain different values of p_{err} depending on the choice of m. For example, if $m = n$ then $p_{err} = 0.6185$, if $m = 2n$ then $p_{err} = 0.38$, and if $m = 5n$ we have $p_{err} = 0.09$. In practice $m = cn$ is a good choice, and for $c > 10$ the error rate is interestingly small and useful for practical applications. Figure 7.12 compares the performance of hashing (with chaining) to that of Bloom filters, assuming that the number r of used hash functions is the one which minimizes the false-positive error rate. In hashing with chaining, we need $\Theta(n(\log n + \log u))$ bits to store the pointers in the chaining lists, and $\Theta(m \log n$ is the space occupancy of the table, in bits. Conversely the Bloom filter does not need to store keys and thus it incurs only in the cost of storing the bit array $m = fn$, where f is pretty small in practice as we observed above.

7.7.1 A lower bound on space

The question is how small can be the bit array B in order to guarantee an given error rate ϵ for a dictionary of n keys drawn from a universe U of size u. This lower bound will allow us to prove that space-wise Bloom filters are within a factor of $\log_2 e \approx 1.44$ of the asymptotic lower bound.

The proof proceeds as follows. Let us represent any data structure solving the membership query



⋄	Hash Tables	Bloom Filters
build time	$\Theta(n)$	$\Theta(n)$
space needed	$\Theta((m+n)\log n + n\log\|U\|)$	$\Theta(m)$
search time	$O(1)$	$(m/n)\ln 2$
ε value	0	$(0.6185)^{m/n}$

FIGURE 7.12: Hashing vs Bloom filter

on a dictionary $X \subseteq U$ with those time/error bounds with a m-bit string $F(X)$. This data structure must work for every possible subset of n elements of U, they are $\binom{u}{n}$. We say that an m-bit string s accepts a key x if $s = F(X)$ for some X containing x, otherwise we say that s rejects x. Now, let us consider a specific dictionary X of n elements. Any string s that is used to represent X must accept each one of the n elements of X, since no false negatives are admitted, but it may also accept at most $\epsilon(u-n)$ other elements of the universe, thus guaranteeing a false positive rate smaller than ϵ. Each string s therefore accepts at most $n + \epsilon(u-n)$ elements, and can thus be used to represent any of the $\binom{n+\epsilon(u-n)}{n}$ subsets of size n of these elements, but it cannot be used to represent any other set. Since we are interested into data structures of m bits, they are 2^m, and we are asking ourselves whether they can represent all the $\binom{u}{n}$ possible dictionaries of U of n keys. Hence, we must have:

$$2^m \times \binom{n+\epsilon(u-n)}{n} \geq \binom{u}{n}$$

or, equivalently:

$$m \geq \log_2 \binom{u}{n} / \binom{n+\epsilon(u-n)}{n} \;\geq\; \log_2 \binom{u}{n} / \binom{\epsilon u}{n} \;\geq\; \log_2 \epsilon^{-n} = n \log_2(1/\epsilon)$$

where we used the inequalities $(\frac{a}{b})^b \leq \binom{a}{b} \leq (\frac{ae}{b})^b$ and the fact that in pratice it is $u \gg n$. If we consider a Bloom filter with the same configuration— namely, error rate ϵ and space occupancy m bits—, then we have $\epsilon = (1/2)^r \geq (1/2)^{m \ln 2/n}$ by setting r to the optimal number of hash functions. After some algebraic manipulations, we find that:

$$m \geq n \frac{\log_2(1/\epsilon)}{\ln 2} \approx 1.44\, n\ \log_2(1/\epsilon)$$

This means that Bloom filters are asymptotically optimal in space, the constant factor is 1.44 more than the minimum possible.

7.7.2 Compressed Bloom filters

In many Web applications, the Bloom filter is not just an object that resides in memory, but it is a data structure that must be transferred between proxies. In this context it is worth to investigate whether Bloom filters can be *compressed* to save bandwidth and transfer time [3]. Suppose that we optimize the false positive rate of the Bloom filter under the constraint that the number of bits to be sent after compression is $z \leq m$. As compression tool we can use Arithmetic coding (see Chapter 12), which well approximates the entropy of the string to be compressed: here simply expressed as $-(p(0)\log_2 p(0) + p(1)\log_2 p(1))$ where $p(b)$ is the frequency of bit b in the input string. Surprisingly enough it turns out that using a larger, but sparser, Bloom filter can yield the same false positive rate with a smaller number of transmitted bits. Said in other words, one can transmit the same number of bits but reduce the false positive rate. An example is given in Table 7.2, where the goal is to obtain small false positive rates by using less than 16 transmitted bits per element. Without compression, the optimal number of hash functions is 11, and the false positive rate is 0.000459. By making a sparse Bloom filter using 48 bits per element but only 3 hash functions, one can compress the result down to less than 16 bits per item (with high probability) and decrease the false positive rate by roughly a factor of 2.

Array bits per element	m/n	16	28	48
Transmission bits per element	z/n	16	15,846	15,829
Hash functions	k	11	4	3
False positive probability	f	0,000459	0,000314	0,000222

TABLE 7.2 Using at most sixteen bits per element after compression, a bigger but sparser Bloom filter can reduce the false positive rate.

Compressing a Bloom filter has benefits: (i) it uses a smaller number of hash functions, so that the lookups are more efficient; (ii) it may reduce the false positive rate for a desired compressed size, or reduce the transmited size for a fixed false positive rate. However the size m of the uncompressed Bloom filter increases the memory usage at running time, and comes at the computational cost of compressing/decompressing it. Nevertheless, some sophisticate approaches are possible which allow to access directly the compressed data without incurring in their decompression. An example was given by the FM-index in Chapter 14, which could built over the bit-array B.

7.7.3 Spectral Bloom filters

A *spectral bloom filter* (SBF) is an extension of the original Bloom filter to multi-sets, thus allowing the storage of multiplicities of elements, provided that they are below a given threshold (a *spectrum* indeed). SBF supports queries on the multiplicities of a key with a small error probability over its estimate, using memory only slightly larger than that of the original Bloom filter. SBF supports also insertions and deletions over the data set.

Let S be a multi-set consisting of n distinct keys from U and let f_x be the multiplicity of the element $x \in S$. In a SBF the bit vector B is replaced by an array of counters $C[0, m-1]$, where $C[i]$ is the sum of f_x-values for those elements $x \in S$ mapping to position i. For every element x, we add the value f_x to the counters $C[h_1(x)], C[h_2(x)], ..., C[h_r(x)]$. Due to possible conflicts among hashes for different elements, the $C[i]$s provide approximated values, specifically upper bounds (given that $f_x > 0$.

The multi-set S can be dynamic. When inserting a new item s, its frequency f_s increases by one, so that we increase by one the counters $C[h_1(s)], C[h_2(s)], \ldots, C[h_r(s)]$; deletion consists symmetrically in decreasing those counters. In order to search for the frequency of element x, we simply return the minimum value $m_x = \min_i C[h_i(x)]$. Of course, m_x is a biased estimator for f_x. In particular, since all $f_x \leq C[h_i]$ for all i, the case in which the estimate is wrong (i.e. $m_x < f_x$) corresponds to the event "all counters $C[h_i(x)]$ have a collision", which in turn corresponds to a "false positive" event in the classical Bloom filter. So, the probability of error in a SBF is the error rate probability for a Bloom filter with the same set of parameters m, n, r.

7.7.4 A simple application

Bloom filters can be used to approximate the intersection of two sets, say A and B, stored in two machines M_A and M_B. We wish to compute $A \cap B$ distributively, by exchanging a small number of bits. Typical applications of the problem are data replication check and distributed search engines. The problem can be efficiently solved by using Bloom filters $BF(A)$ and $BF(B)$ stored in M_A and

M_B, respectively. The algorithmic idea to compute $A \cap B$ is as follows:

1. M_A sends $BF(A)$ to M_B, using $r_{opt} = (m_A ln2)/|A|$ hash functions and a bit-array $m_A = \Theta(|A|)$;
2. M_B checks the existence of elements B into A by deploying $BF(A)$ and sends back explicitly the set of found elements, say Q. Note that, in general, $Q \supseteq A \cap B$ because of false positives;
3. M_A computes $Q \cap A$, and returns it.

Since Q contains $|A \cap B|$ keys plus the number of false positives (elements belonging only to A), we can conclude that $|Q| = |A \cap B| + |B|\epsilon$ where $\epsilon = 0.6185^{m_A/|A|}$ is the error rate for that design of $BF(A)$. Since we need $\log |U|$ bits to represent each key, the total number of exchanged bits is $\Theta(|A|) + (|A \cap B| + |B| 0.6185^{m_A/|A|}) \log |U|$ which is much smaller than $|A| \log |U|$ the number of bits to be exchanged by using a plain algorithm that sends the whole A's set to M_B.

References

[1] Martin Aumüller, Martin Dietzfelbinger, and Philipp Woelfel. Explicit and Efficient Hash Families Suffice for Cuckoo Hashing with a Stash. In Proceedings of European Symposium on Algorithms (ESA), Lecture Notes in Computer Science 7501, Springer, 108–120, 2012.

[2] Andrei Z. Broder and Michael Mitzenmacher. Survey: Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4): 485-509, 2003.

[3] Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networks*, 10(5): 604-612, 2002.

[4] Ian H. Witten, Alistair Moffat and Timothy C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition. Morgan Kaufmann, 1999.

8

Set Intersection

"Sharing is caring!"

This lecture attacks a simple problem over sets, it constitutes the backbone of every query resolver in a (Web) search engine. A search engine is a well-known tool designed to search for information in a collection of documents $\mathcal{D}$. In the present chapter we restrict our attention to search engines for *textual* documents, meaning with this the fact that a document $d_i \in \mathcal{D}$ is a book, a news, a tweet or any file containing a sequence of linguistic tokens (aka, *words*). Among many other auxiliary data structures, a search engine builds an *index* to answer efficiently the queries posed by users. The user query Q is commonly structured as a *bag of words*, say $w_1 w_2 \cdots w_k$, and the goal of the search engine is to retrieve the *most relevant* documents in $\mathcal{D}$ which contain all query words. The people skilled in this art know that this is a very simplistic definition, because modern search engines search for documents that contain possibly *most* of the words in Q, the verb *contain* may be fuzzy interpreted as *contain synonyms or related words*, and the notion of *relevance* is pretty subjective and time-varying so that it cannot be defined precisely. In any case, this is not a chapter of an Information Retrieval book, so we refer the interested reader to the Information Retrieval literature, such as [4, 8]. Here we content ourselves to attack the most generic algorithmic step specified above.

> **Problem.** *Given a sequence of words $Q = w_1 w_2 \cdots w_k$ and a document collection $\mathcal{D}$, find the documents in $\mathcal{D}$ that contain all words w_i.*

An obvious solution is to scan each document in $\mathcal{D}$ searching for all words specified by Q. This is simple but it would take time proportional to the whole length of the document collection, which is clearly too much even for a supercomputer or a data-center given the Web size! And, in fact, modern search engines build a very simple, but efficient, data structure called *inverted index* that helps in speeding up the flow of bi/million of daily user queries.

The inverted index consists of three main parts: the dictionary of words w, one list of occurrences per dictionary word (called *posting list*, below indicated with $\mathcal{L}[w]$), plus some additional information indicating the importance of each of these occurrences (to be deployed in the subsequent phases where the relevance of a document has to be established). The term "inverted" refers to the fact that word occurrences are not sorted according to their position in the document, but according to the alphabetic ordering of the words to which they refer. So inverted indexes remind the classic *glossary* present at the end of books, here extended to represent occurrences of *all* the words present into a collection of documents (and so, not just the most important words of them).

Each posting list $\mathcal{L}[w]$ is stored contiguously in a single array, eventually on disk. The names of the indexed documents (actually, their identifying URLs) are placed in another table and are succinctly identified by integers, called *docID*s, which we may assume to have been assigned arbitrarily by the search engine.[1] Also the dictionary is stored in a table which contains some satellite information plus the pointers to the posting lists. Figure 8.1 illustrates the main structure of an inverted index.

`Dictionary`	`Posting list`
...	...
`abaco`	50, 23, 10
`abiura`	131, 100, 90, 132
`ball`	20, 21, 90
`mathematics`	15, 1, 3, 23, 30, 7, 10, 18, 40, 70
`zoo`	5, 1000
...	...

FIGURE 8.1: An example of inverted (unsorted) index for a part of a dictionary.

Coming back to the problem stated above, let us assume that the query Q consists of two words `abaco mathematics`. Finding the documents in $\mathcal{D}$ that contain both two words of Q boils down to finding the docIDs shared by the two inverted lists pointed to by `abaco` and `mathematics`: namely, 10 and 23. It is easy to conclude that this means to solve a *set intersection* problem between the two sets represented by $\mathcal{L}$[`abaco`] and $\mathcal{L}$[`mathematics`], which is the key subject of this chapter.

Given that the integers of two posting lists are arbitrarily arranged, the computation of the intersection might be executed by comparing each docID $a \in \mathcal{L}$[`abaco`] with all docIDs $b \in \mathcal{L}$[`mathematics`]. If $a = b$ then a is inserted in the result set. If the two lists have length n and m, this brute-force algorithm takes $n \times m$ steps/comparisons. In the real case that n and m are of the order of millions, as it typically occurs for common words in the modern Web, then that number of steps/comparisons is of the order of $10^6 \times 10^6 = 10^{12}$. Even assuming that a PC is able to execute one billion comparisons per second (10^9 cmp/sec), this trivial algorithm takes 10^3 seconds to process a bi-word query (so about ten minutes), which is too much even for a patient user!

The bad news is that the docIDs occurring in the two posting lists cannot be arranged *arbitrarily*, but we must impose some proper structure over them in order to speed up the identification of the common integers. The key idea here is to *sort* the posting lists as shown in Figure 8.2.

It is therefore preferable, from a computational point of view, to reformulate the intersection problem onto two *sorted* sets $A = \mathcal{L}$[`abaco`] and $B = \mathcal{L}$[`mathematics`], as follows:

> **(Sorted) Set Intersection Problem.** *Given two sorted integer sequences $A = a_1 a_2 \cdots a_n$ and $B = b_1 b_2 \cdots b_m$, such that $a_i < a_{i+1}$ and $b_i < b_{i+1}$, compute the integers common to both sets.*

The *sortedness* of the two sequences allows to design an intersection algorithm that is deceptively simple, elegant and fast. It consists of scanning A and B from left to right by comparing at each step a pair of docIDs from the two lists. Say a_i and b_j are the two docIDs currently compared, initially $i = j = 1$. If $a_i < b_j$ the iterator i is incremented, if $a_i > b_j$ the iterator j is incremented, otherwise $a_i = b_j$ and thus a common docID is found and both iterators are incremented. At each

[1] To be precise, the docID assignment process is a crucial one to save space in the storage of those posting lists, but its solution is too much sophisticated to be discussed here and thus it is deferred to the scientific literature [7].

Dictionary	Posting list
...	...
abaco	10, 23, 50
abiura	90, 100, 131, 132
ball	20, 21, 90
mathematics	1, 3, 7, 10, 15, 18, 23, 30, 40, 70
zoo	5, 1000
...	...

FIGURE 8.2: An example of inverted (sorted) index for a part of a dictionary.

step the algorithm executes one comparison and advances at least one iterator. Given that $n = |A|$ and $m = |B|$ are the number of elements in the two sequences, we can deduct that i (resp. j) can advance at most n times (resp. m times), so we can conclude that this algorithm requires no more than $n + m$ comparisons/steps; we write *no more* because it could be the case that one sequence is exhausted much before the other one, so that many elements of the latter may be not compared. This time cost is significantly smaller than the one mentioned above for the unsorted sequences (namely $n \times m$), and its real advantage in practice is strikingly evident. In fact, by considering our running example with n and m of the order of 10^6 docIDs and a PC performing 10^9 comparisons per second, we derive that this new algorithm takes 10^{-3} seconds to compute $A \cap B$, which is in the order of milliseconds, exactly what occurs in modern search engines.

An attentive reader may have noticed this algorithm mimics the merge-procedure used in Merge-Sort, here adapted to fing the common elements of the two sets A and B rather than merging them.

FACT 8.1 *The intersection algorithm based on the merge-based paradigm solves the sorted set intersection problem in $O(m + n)$ time.*

In the case that $n = \Theta(m)$ this algorithm is optimal, and thus it cannot be improved; moreover it is based on the scan-based paradigm that it is optimal also in the disk model because it takes $O(n/B)$ I/Os. To be more precise, the scan-based paradigm is optimal whichever is the memory hierarchy underlying the computation (the so called *cache-oblivious model*). The next question is what we can do whenever m is much different of n, say $m \ll n$. This is the situation in which one word is much more selective than the other one; here, the classic *binary search* can be helpful, in the sense that we can design an algorithm that binary searches every element $b \in B$ (they are few) into the (many) sorted elements of A thus taking $O(m \log n)$ steps/comparisons. This time complexity is better than $O(n + m)$ if $m = o(n/\log n)$ which is actually less stringent that the condition $m \ll n$ we imposed above.

FACT 8.2 *The intersection algorithm based on the binary-search paradigm solves the sorted set intersection problem in $O(m \log n)$ time.*

The next question is whether an algorithm can be designed that combines the best of both merge-based and search-based approaches. In fact, there is an inefficacy in the binary-search approach which becomes apparent when m is of the order of n. When we search item b_i in A we possibly re-check over and over the same elements of A. Surely this is the case for its middle element, say $a_{n/2}$, which is the first one checked by any binary search. But if $b_i > a_{n/2}$ then it is useless to compare b_{i+1} with $a_{n/2}$ because for sure it is larger, since $b_{i+1} \geq b_i > a_{n/2}$. And the same holds for all subsequent elements of B. A similar argument applies possibly to other elements in A checked by the binary search; so the next challenge we address is how to avoid this *useless comparisons*.

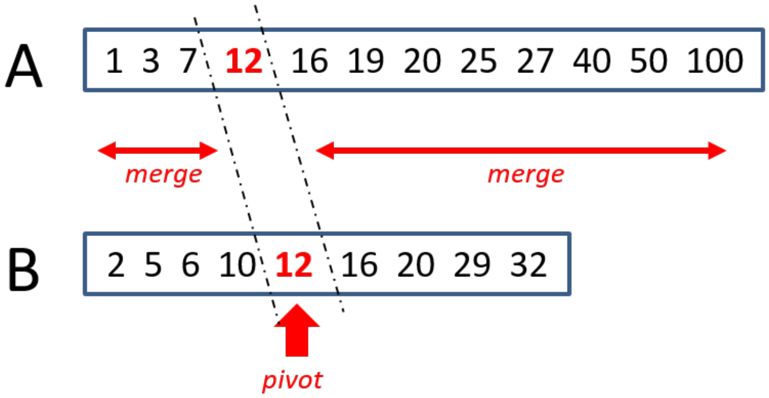


FIGURE 8.3: An example of the Intersection paradigm based on Mutual Partitioning: the pivot is 12, the median element of B.

This is achieved by adopting another classic algorithmic paradigm, called *partitioning*, which is the one we used to design the Quicksort, and here applied to split repeatedly and mutually two sequences. Formally, let us assume that $m \leq n$ and be both even numbers, we pick the *median* element $b_{m/2}$ of the shortest sequence B as a *pivot* and search for it into the longer sequence A. Two cases may occur: (i) $b_{m/2} \in A$, say $b_{m/2} = a_j$ for some j, and thus $b_{m/2}$ is returned as one of the elements of the intersection $A \cap B$; or (ii) $b_{m/2} \notin A$, say $a_j < b_{m/2} < a_{j+1}$ (where we assume that $a_0 = -\infty$ and $a_{n+1} = +\infty$). In both cases the intersection algorithm proceeds *recursively* in the two parts in which each sequence A and B has been split by the choice of the pivot, thus computing recursively $A[1, j] \cap B[1, m/2 - 1]$ and $A[j + 1, n] \cap B[m/2 + 1, n]$. A small optimization consists of discarding from the first recursive call the element $b_{m/2} = a_j$ (in case (i)). The pseudo-code is given in Figure 8.1, and a running example is illustrated in Figure 8.3. There the median element of B used as the pivot for the mutual partitioning of the two sequences is 12, and it splits A into two unbalanced parts (i.e. $A[1, 4]$ and $A[5, 12]$) and B into two almost-halves (i.e. $B[1, 5]$ and $B[6, 9]$) which are recursively intersected; since the pivot occurs both in A and B it is returned as an element of the intersection. Moreover we notice that the first part of A is shorter than the first part of B and thus in the recursive call their role will be exchanged.

In order to evaluate the time complexity we need to identify the worst case. Let us begin with the simplest situation in which the pivot falls outside A (i.e. $j = 0$ or $j = n$). This means that one of the two parts in A is empty and thus the corresponding halve of B can be discarded from the subsequent recursive calls. So one binary search over A, costing $O(\log n)$, has discarded an half of B. If this occurs at any recursive call, the total number of calls will be $O(\log m)$ thus inducing an overall cost for the algorithm equal to $O(\log m \ \log n)$. That is, an *unbalanced* partitioning of A induces indeed a very good behavior of the intersection algorithm; this is something opposite to what stated typically about recursive algorithms. On the other hand, let us assume that the pivot $b_{m/2}$ falls inside the sequence A and consider the case that it coincides with the median element of A, say $a_{n/2}$. In this specific situation the two partitions are balanced in both sequences we are intersecting, so the time complexity can be expressed via the following recurrent relation $T(n, m) = O(\log n) + 2T(n/2, m/2)$,

Algorithm 8.1 Intersection based on Mutual Partitioning

```
Let $m = |B| \leq n = |A|$, otherwise exchange the role of $A$ and $B$;
Pick the median element $p = b_{\lfloor m/2 \rfloor}$ of $B$;
Binary search for the position of $p$ in $A$, say $a_j \leq p < a_{j+1}$;
**if** $p = a_j$ **then**
    print $p$;
**end if**
Compute recursively the intersection $A[1, j] \cap B[1, m/2]$;
Compute recursively the intersection $A[j+1, n] \cap B[m/2+1, n]$.
```

with the base case of $T(n,m) = O(1)$ whenever $n, m \leq 1$. It can be proved that this recurrent relation has solution $T(n,m) = O(m(1 + \log \frac{n}{m}))$ for any $m \leq n$. It is interesting to observe that this time complexity subsumes the ones of the previous two algorithms (namely the one based on merging and the one based on binary searching). In fact, when $m = \Theta(n)$ it is $T(n,m) = O(n)$ (á la merging); when $m \ll n$ it is $T(n,m) = O(m \log n)$ (á la binary searching). As we will see in Chapter 12, about Statistical compression, the term $m \log \frac{n}{m}$ reminds an entropy cost of encoding m items within n items and thus induces to think about something that cannot be improved (for details see [1]).

FACT 8.3 *The intersection algorithm based on the mutual-partitioning paradigm solves the sorted set intersection problem in $O(m(1 + \log \frac{n}{m}))$ time.*

We point out that the bound $m \log \frac{n}{m}$ is optimal in the comparison model because it follows from the classic binary decision-tree argument. In fact, they do exist at least $\binom{n}{m}$ solutions to the set intersection problem (here we account only for the case in which $B \subseteq A$), and thus every comparison-based algorithm computing anyone of them must execute $\Omega(\log \binom{n}{m})$ steps, which is $\Omega(m \log \frac{n}{m})$ by definition of binomial coefficient.

Algorithm 8.2 Intersection based on Doubling Search

```
Let $m = |B| \leq n = |A|$, otherwise exchange the role of $A$ and $B$;
$i = 1$;
**for** $j = 1, 2, \ldots, m$ **do**
    $k = 0$;
    **while** $(i + 2^k \leq n)$ and $(B[j] > A[i + 2^k])$ **do**
        $k = k + 1$;
    **end while**
    $i'$ = Binary search $B[j]$ into $A[i + 1, \min\{i + 2^k, n\}]$;
    **if** $(a_{i'} = b_j)$ **then**
        print $b_j$;
    **end if**
    $i = i'$.
**end for**
```

Although this time complexity is appealing, the previous algorithm is heavily based on recursive calls and binary searching which are two paradigms that offer poor performance in a disk-based setting when sequences are long and thus the number of recursive calls can be large (i.e. many dynamic memory allocations) and large is the number of binary-search steps (i.e. random memory accesses). In order to partially compensate with these issues we introduce another approach to ordered set intersection which allows us to discuss another interesting algorithmic paradigm: the so called *doubling search* or *galloping search* or also *exponential search*. It is a mix of merging and binary searching, which is clearer to discuss by means of an inductive argument. Let us assume that we have already checked the first $j-1$ elements of B for their appearance in A, and assume that $a_i \leq b_{j-1} < a_{i+1}$. To check for the next element of B, namely b_j, it suffices to search it in $A[i+1, n]$. However, and this is the bright idea of this approach, instead of binary searching this sub-array, we execute a *galloping search* which consists of checking elements of $A[i+1, n]$ at distances which grow as a power of two. This means that we compare b_j against $A[i+2^k]$ for $k = 0, 1, \ldots$ until we find that either $b_j < A[i+2^k]$, for some k, or it is $i + 2^k > n$ and thus we jumped out of the array A. Finally we perform a binary search for b_j in $A[i+1, \min\{i+2^k, n\}]$, and we return b_j if the search is successful. In any case, we determine the position of b_j in that subarray, say $a_{i'} \leq b_j < a_{i'+1}$, so that the process can be repeated by discarding $A[1, i']$ from the subsequent search for the next element of B, i.e. b_{j+1}. Figure 8.4 shows a running example, whereas Figure 8.2 shows the pseudo-code of the doubling search algorithm.

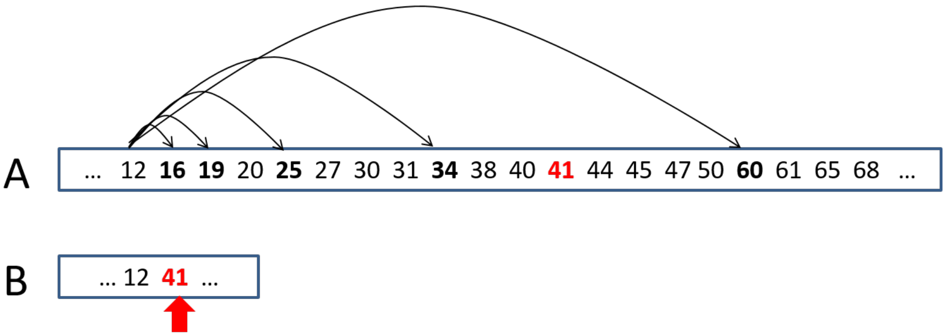


FIGURE 8.4: An example of the Doubling Search paradigm: the two sequences A and B are assumed to have been intersected up to the element 12. The next element in B, i.e. 41, is taken to be exponentially searched in the *suffix* of A following 12. This search checks A's elements at distances which are a power of two— namely $1, 2, 4, 8, 16$— until it finds the element 60 which is larger than 41 and thus delimits the portion of A within which the binary search for 41 can be confined. We notice that the searched sub-array has size 16, whereas the distance of 41 from 12 in A is 11 thus showing, on this example, that the binary search is executed on a sub-array whose size is smaller than twice the real distance of the searched element.

As far as the time complexity is concerned, we observe that the parameter k satisfies the property that $A[i+2^{k-1}] < b_j \leq A[i+2^k]$. So the position $i' - i$ of b_j in $A[i+1, \min\{i+2^k, n\}]$ is not much smaller than the size of this sub-array, because it is $2^{k-1} < i' - i \leq 2^k$ and so $2^k < 2(i' - i)$. Let us therefore denote with Δ_j the size of the sub-array where the binary search of b_j is executed, and let us denote with $i_j = i'$ as the position where b_j occurs in A. For the sake of presentation we set $i_0 = 0$. Clearly $i_j - i_{j-1} \leq \Delta_j \leq 2^k$ and thus, from before, we have $\Delta_j \leq 2(i_j - i_{j-1})$. These sub-arrays may be overlapping but by not much, as indeed we have $\sum_{j=1}^{m} \Delta_j \leq \sum_{j=1}^{m} 2(i_j - i_{j-1}) = 2n$ because this is a telescopic sum in which consecutive terms in the summation cancel out. For every j, the algorithm in Figure 8.2 executes $O(\log \Delta_j)$ steps because of the while-statement and because of the binary search. Summing for $j = 1, 2, \ldots, m$ we get a total time complexity of $O(\sum_{j=1}^{m} \log \Delta_j) = O(m \log \sum_{j=1}^{m} \frac{\Delta_j}{m}) = O(m \log \frac{n}{m})$.

FACT 8.4 *The intersection algorithm based on the doubling-search paradigm solves the sorted set intersection problem in $O(m(1 + \log \frac{n}{m}))$ time. This is the same time complexity of the intersection algorithm based on the mutual-partitioning paradigm but without incurring in the costs due to the recursive partitioning of the two sequences A and B. The time complexity is optimal in the comparison model.*

Although the previous approach avoids some of the pitfalls due to the recursive partitioning of the two sequences A and B, it still needs to jump over the array A because of the doubling scheme; and we know that this is inefficient when executed in a hierarchical memory. In order to avoid this issue, programmers adopt a *two-level organization of the data*, which is a very frequent scheme of efficient data structures for disk. The main idea of this storage scheme is to *logically* partition the sequence A into blocks A_i of size L each, and copy the first element of each block (i.e. $A_i[1] = A[iL + 1]$) into an auxiliary array A' of size $O(n/L)$. For the simplicity of exposition, let us assume that $n = hL$ so that the blocks A_i are h in number. The intersection algorithm then proceeds in two main phases. Phase 1 consists of merging the two sorted sequences A' and B, thus taking $O(n/L + m)$ time. As a result, the elements of B are interspersed among the element of A'. Let us denote by B_i the elements of B which fall between $A_i[1]$ and $A_{i+1}[1]$ and thus may occur in the block A_i. Phase 2 then consists of executing the merge-based paradigm of Fact 8.1 over all pairs of sorted sequences A_i and B_i which are non empty. Clearly, these pairs are no more than m. The cost of one of these merges is $O(|A_i| + |B_i|) = O(L + |B_i|)$ and they are at most m because this is the number of unempty blocks B_i. Moreover $B = \cup_i B_i$, consequently this intersection algorithm takes a total of $O(\frac{n}{L} + mL)$ time. For further details on this approach and its performance in practice the reader can look at [5].

FACT 8.5 *The intersection algorithm based on the two-level storage paradigm solves the sorted set intersection problem in $O(\frac{n}{L} + mL)$ time and $O(\frac{n}{LB} + \frac{mL}{B} + m)$ I/Os, because every merge of two sorted sequences A_i and B_i takes at least* 1 *I/O and they are no more than m.*

The two-level storage paradigm is suitable to adopt a compressed storage for the docIDs in order to save space. Let $a'_1, a'_2, \ldots, a'_L$ be the L docIDs stored ordered in some block A_i. These integers can be squeezed by adopting the so called Δ-scheme which consists of setting $a'_0 = 0$ and then representing a'_j as its difference with the preceding docID a'_{j-1} for $j = 1, 2, \ldots, L$. Then each of these differences can be stored somewhat compressed by using $\lceil \log_2 \max_i \{a'_i - a'_{i-1}\} \rceil$ bits, instead of the full-representation of four bytes!

This is a clear advantage whenever the differences are much smaller than the universe size from which the docIDs are taken. But this is not necessarily the case in practice, because the partitioning of the sequences is done according to L and not according to values of the docIDs. In general, given a sequence of n docIDs spread in a universe of $U = \{1, 2, \ldots, u\}$, the distribution which guarantees the smallest-maximum gap is the uniform one: for which it is $\max_i \{a'_i - a'_{i-1}\} \leq \frac{u}{n}$. In order to force this situation we preliminary shuffle the docIDs via a random permutation $\pi : U \longrightarrow U$, and assume that the possible sets over which the intersection problem can be invoked have been given in advance, and they are not larger than N. This is not a restriction in the Search Engine scenario because the dictionary and its posting lists are fixed in advance. Said this, we propose a solution which combines speed (as in the two-level scheme) and compressed space (unlike other solutions) by distinguishing between a preprocessing phase and a query phase.

In the preprocessing phase, we logically spit the universe U into M/L buckets of size uL/M each, denoted by U_i, where M is set as the size of the longest list. Then, we permute A according to the random permutation π and assign its permuted elements to the buckets U_i: namely, for each $x \in A$ we compute $\pi(x)$, take its $\ell = \lceil \log_2 \frac{uL}{M} \rceil$ most significant bits and denote by $\pi_\ell(x)$ their value. We then assign x to the bucket $U_{\pi_\ell(x)}$. We denote by A_i the sub-list of A's elements that have been mapped to U_i and are sorted according to their π-values. To implement the following query phase we need to have available π^{-1} so that we can retrieve the original element from its π-image.

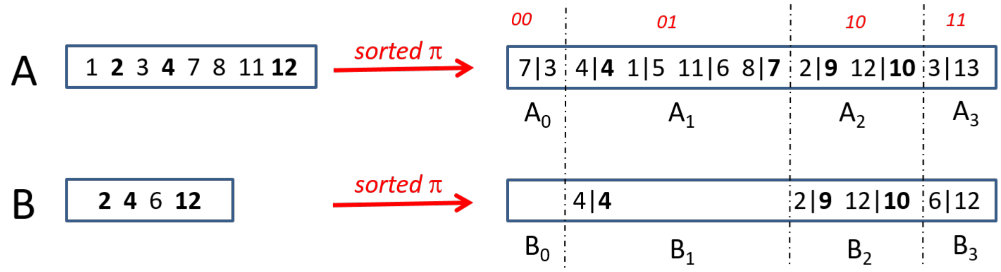


FIGURE 8.5: An example of the Random Permuting and Splitting paradigm. We assume universe $U = \{1, \ldots, 13\}$, set $L = 2$ and $M = 8$, and consider the permutation $\pi(x) = 1 + (4x \mod 13)$. So U is partitioned in $M/L = 4$ buckets identified by the MSB $\ell = \lceil \log_2 UL/M \rceil = \lceil \log_2 13 * 2/8 \rceil = 2$ bits of the π-image of each element. Recall that every π-image is represented in $\log_2 u = 4$ bits, so that $\pi(1) = 5 = (0101)_2$ and its 2 MSB are 01. The figure shows in bold the elements of $A \cap B$, moreover it depicts for the sake of exposition each docID as the pair $x \,|\, \pi(x)$ and, on top of every sublist, shows the 2 MSBs. In the example only three buckets of B are unempty, so we intersect only them with the corresponding ones of A, so that we drop the sublist A_0 without scanning it. The result is $\{4|4, 2|9, 12|10\}$, that gives $A \cap B$ by dropping the second π-component: namely, $\{2, 4, 12\}$.

In the query phase, let us assume that we wish to compute the intersection of two sets A and B which have been preprocessed above, and that $n = |A| > m = |B|$. The intuition is that, since the permutation π is the same for all sets, if element $z \in A \cap B$ then $\pi(z)$ will be routed to some sublists A_j and B_j with $j = \pi_\ell(z)$ according to its ℓ most significant bits. Therefore intersecting $A_j \cap B_j$ will correctly detect $\pi(z)$; moreover, since we have available π^{-1}, we can recover the original shared item z after having matched $\pi(z)$. Given these premises the intersection algorithm can be easily designed: for each B_j, we compute $B_j \cap A_j$ via the merge-based approach (Fact 8.1) and return the π^{-1}-image of the intersected elements. The average time complexity is therefore $O(m + \min\{n, mL\})$ because the number of unempty sublists B_j is at most m and each pair of intersected sets contains $O(L)$ elements on average. In fact the random permutation π maps A's elements in U's buckets of size $uL/M \leq uL/n$ so that, on average, each of those buckets contains $|A| * (L/M) \leq |A| * (L/n) = L$. This time complexity improves the one of the two-level storage scheme (Fact 8.5) whenever $m < \frac{n}{L}$, as it typically occurs in practice. A running example is shown in Figure 8.6.

As far as the space occupancy is concerned we notice that there are $\Theta(M/L)$ buckets, and the largest difference between two bucket entries can be bounded in two ways: the bucket width $O(uL/M)$ and the largest difference between any two consecutive list entries (after π-mapping). The latter quantity is well known from balls-and-bins problem: here having n balls and u bins. It can be shown that the largest difference is $O(\frac{u}{n} \log n)$ with high probability. The two bounds can be combined to a bound of $O(\log_2 \min\{\frac{uL}{M}, \frac{u}{n} \log n\}) = \log_2 \frac{u}{n} + \log_2 \min\{L, \log n\} + O(1)$ bits per list element (after π-mapping). The first term is unavoidable since it already shows up in the information theoretic lower bound. To be precise, an additional $O(n \frac{\log n}{L})$ bits have to be considered for A

in order to account for the cost of the $O(\log n)$-bits pointer to (the beginning of) each sublist of A. This term can be made smaller and smaller by increasing (not much!) the value of L.

FACT 8.6 *The intersection algorithm based on the random-permuting and splitting paradigm solves the sorted set intersection problem in $O(m + \min\{n, mL\})$ time and $O(m + \min\{\frac{n}{B}, \frac{mL}{B}\})$ I/Os, because every merge of two sublists A'_j and B'_j takes at least 1 I/O and they are no more than m. The space cost for storing a list of length n is $n(\log_2 \frac{u}{n} + \log_2 \min\{L, \log n\} + O(1 + \frac{\log n}{L}))$ bits with high probability.*

By analyzing the algorithmic structure of this last solution we notice few further advantages. First, we do not need to sort the original sequences, because the sorting is required only within the individual sublists which have average length L; this is much shorter than the lists' length so that we can use an internal-memory sorting algorithm over each π-permuted sublist. A second advantage is that we can avoid the checking of some sublists during the intersection process, without looking at them; this allows to drop the term $\frac{n}{L}$ occurring in Fact 8.5. Third, the choice of L can be done according to the hierarchical memory in which the algorithm is run; this means that if sublists are stored on disk, then $L = \Theta(B)$ can be the right choice.

The authors of [5, 3, 2] discuss some variants and improvements over all previous algorithms, some really sophisticate, we refer the interested reader to this literature. Here we report a picture taken from [5] that compares various algorithms with the following legenda: *zipper* is the merge-based algorithm (Fact 8.1), *skipper* is the two-level algorithm (Fact 8.5, with $L = 32$), *Baeza-Yates* is the mutual-intersection algorithm (Fact 8.3, 32 denotes the bucket size for which recursion is stopped), *lookup* is our last proposal (Fact 8.6, $L = 8$).

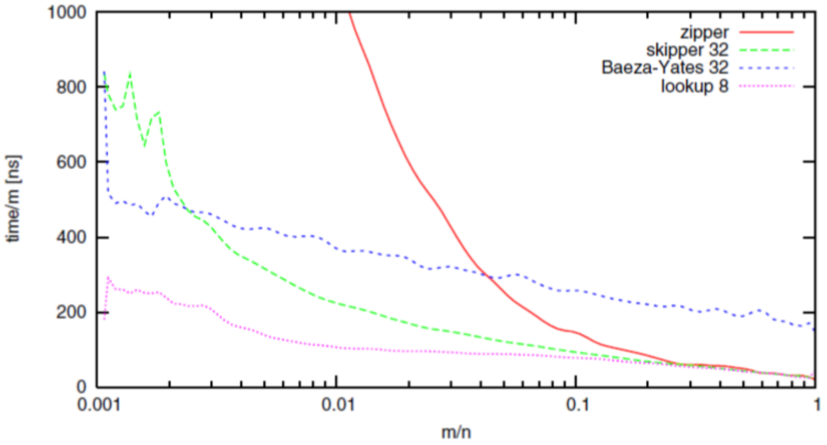


FIGURE 8.6: An experimental comparison among four sorted-set intersection algorithms.

We notice that *lookup* is the best algorithm up to a length ratio close to one. For lists of similar length all algorithms are very good. Still, it could be a good idea to implement a version of *lookup* optimized for lists of similar length. It is also interesting to notice that *skipper* improves *Baeza-Yates* for all but very small length ratios. For compressed lists and very different list lengths, we can claim that *lookup* is considerably faster over all other algorithms. Randomization allows interesting performance guarantees on both time and space performance. The experimented version of *skipper* uses a compressed first-level array; probably by dropping compression from the first-level would not increase much the space, but it would induce a significant speedup in time. The only clear looser is *Baeza-Yates*, for every list lengths there are other algorithms that improve it. It is pretty much clear that a good asymptotic complexity does not reflect onto a good time efficiency whenever recursion is involved.

References

[1] Ricardo Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Procs of Annual Symposium on Combinatorial Pattern Matching (CPM)*, Lecture Notes in Computer Science 3109, pp. 400-408, 2014.

[2] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, Alejandro Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[3] Bolin Ding, Arnd Christian König. Fast set intersection in memory. *PVLDB*, 4(4): 255-266, 2011.

[4] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[5] Peter Sanders, Frederik Transier. Intersection in integer inverted indices. In *Procs of Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[6] Hao Yan, Shuai Ding, Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Procs of WWW*, pp. 401-410, 2009.

[7] Ian H. Witten, Alistair Moffat, Timoty C. Bell. *Managing Gigabytes*. Morgan Kauffman, second edition, 1999.

9

Searching Strings by Prefix

This problem is experiencing renewed interest in the algorithmic community because of new applications spurring from Web search-engines. Think to the *auto-completion* feature currently offered by mayor search engines Google, Bing and Yahoo, in their search bars: It is a prefix search executed on-the-fly over millions of strings, using the query pattern typed by the user as the string to search. The dictionary typically consists of the most recent and the most frequent queries issued by other users. This problem is made challenging by the size of the dictionary and by the time constraints imposed by the patience of the users. In this chapter we will describe many different solutions to this problem of increasing sophistication and efficiency both in time, space and I/O complexities.

The prefix-search problem. *Given a dictionary $\mathcal{D}$ consisting of n strings of total length N, drawn from an alphabet of size σ, the problem consists of preprocessing $\mathcal{D}$ in order to retrieve (or just count) the strings of $\mathcal{D}$ that have P as a prefix.*

We mention two other typical string queries which are the *exact* search and the *substring* search within the dictionary strings in $\mathcal{D}$. The former is best addressed via hashing because of its simplicity and practical speed; Chapter 7 will detail several hashing solutions. The latter problem is more sophisticated and finds application in computational genomics and asian search engines, just to cite a few. It consists of finding all *positions* where the query-pattern P occurs as a substring of the dictionary strings. Surprisingly enough, it does exist a simple *algorithmic reduction* from substring search to prefix search over the set of *all suffixes of the dictionary strings*. This reduction will be commented in Chapter 10 where the Suffix Array and the Suffix Tree data structures will be introduced. As a result, we can conclude that the prefix search is the backbone of other important search problems on strings, so the data structures introduced in this chapter offer applications which go far beyond the simple ones discussed below.

9.1 Array of string pointers

We start with a simple, common solution to the prefix-search problem which consists of an array of pointers to strings stored in arbitrary locations on disk. Let us call $A[1, n]$ the array of pointers, which are *indirectly* sorted according to the strings pointed to by its entries. We assume that each pointer takes w bytes of memory, typically 4 bytes (32 bits) or 8 bytes (64 bits). Several other representations of pointers are possible, as e.g. variable-length representations, but this discussion is deferred to Chapter 11, where we will deal with the efficient encoding of integers.

Figure 9.1 provides a running example in which the dictionary strings are stored in an array S, according to an arbitrary order.

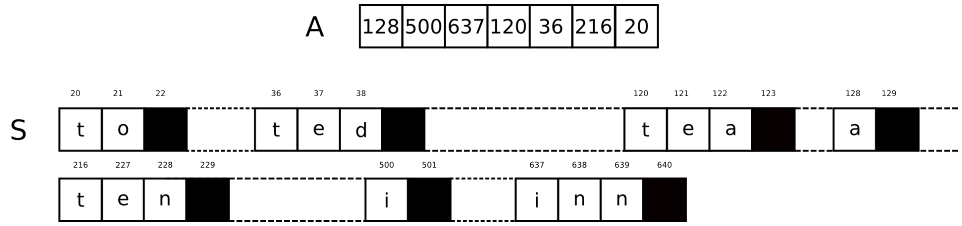


FIGURE 9.1: The array S of strings and the array A of (indirectly) sorted pointers to S's strings.

There are two crucial properties that the (sorted) array A satisfies:

- all dictionary strings prefixed by P occur contiguously if lexicographically sorted. So their pointers occupy a subarray, say $A[l, r]$, which may be possibly empty if P does not prefix any dictionary string.
- the string P is lexicographically located beyween $A[l - 1]$ and $A[l]$.

Since the prefix search returns either the number of dictionary strings prefixed by P, hence the value $r - l + 1$, or visualizes these strings, the key problem is to identify the two extremes l and r efficiently. To this aim, we reduce the prefix search problem to the *lexicographic search* of a pattern Q in $\mathcal{D}$: namely, the search of the lexicographic position of Q among $\mathcal{D}$'s strings. The formation of Q is simple: Q is either the pattern P or the pattern $P\#$, where # is larger than any other alphabet character. It is not difficult to convince yourself that $Q = P$ will precede the string $A[l]$ (see above), whereas $Q = P\#$ will follow the string $A[r]$. This means actually that two lexicographic searches for patterns of length no more than $p + 1$ are enough to solve the prefix-search problem.

The lexicographic search can be implemented by means of an (indirect) binary search over the array A. It consists of $O(\log n)$ steps, each one requiring a string comparison between Q and the string pointed by the entry tested in A. The comparison is lexicographic and thus takes $O(p)$ time and $O(p/B)$ I/Os, because it may require in the worst case the scan of all $\Theta(p)$ characters of Q. The poor time and I/O-complexities derive from the *indirection*, which forces no locality in the memory/string accesses of the binary search. The inefficiency is even more evident if we wish to retrieve all n_occ strings prefixed by P, and not just count them. After that the range $A[l, r]$ has been identified, each string visualization elicits at least one I/O because contiguity in A does not imply contiguity of the pointed strings in S.

THEOREM 9.1 *The complexity of a prefix search over the array of string pointers is $O(p \log n)$ time and $O(\frac{p}{B} \log n)$ I/Os, the total space is $N + (1 + w)n$ bytes. Retrieving the n_{occ} strings prefixed by P needs $\Omega(n_{occ})$ I/Os.*

Proof Time and I/O complexities derive from the previous observations. For the space occupancy, A needs n pointers, each taking a memory word w, and all dictionary strings occupy N bytes plus one-byte delimiter for each of them (commonly `\0` in C). ■

The bound $\Omega(n_{occ})$ may be a major bottleneck if the number of returned strings is large, as it typically occurs in queries that use the prefix search as a preliminary step to select a *candidate set of answers* that have then to be refined via a proper post-filtering process. An example is the solution to the problem of *searching with wild-cards* which involves the presence in P of many special symbols *. The wild-card * matches any substring. In this case if $P = \alpha * \beta * \cdots$, where $\alpha, \beta, \ldots$ are un-empty strings, then we can implement the wild-card search by first performing a prefix-search for α in $\mathcal{D}$ and then checking brute-forcedly whether P matches the returned strings given the presence of the wild-cards. Of course this approach may be very expensive, especially when α is not a selective prefix and thus many dictionary strings are returned as candidate matches. Nevertheless this puts in evidence how much slow may be in a disk environment a wild-card query if solved with a trivial approach.

9.1.1 Contiguous allocation of strings

A simple trick to circumvent some of the previous limitations is to store the dictionary strings sorted lexicographically and contiguously on disk. This way (pointers) contiguity in A reflects into (string) contiguity in S. This has two main advantages:

- when the binary search is confined to few strings, they will be closely stored both in A and S, so probably they have been buffered by the system in internal memory (*speed*);
- some compression can be applied to contiguous strings in S, because they typically share some prefix (*space*).

Given that S is stored on disk, we can deploy the first observation by blocking strings into groups of B characters each and then *store* a pointer to the first string of each group in A. The sampled strings are denoted by $\mathcal{D}_B \subseteq \mathcal{D}$, and their number n_B is upper bounded by $\frac{N}{B}$ because we pick at most one string per block. Since A has been squeezed to index at most $n_B \leq n$ strings, the search over A must be changed in order to reflect the *two-level structure* given by the array A and the blocks of strings in S. So the idea is to decompose the lexicographic search for Q in a two-stages process: in the first stage, we search for the lexicographic position of Q within the sampled strings of $\mathcal{D}_B$; in the second stage, this position is deployed to identify the block of strings where the lexicographic position of Q lies in, and then the strings of this block are scanned and compared with Q for prefix match. We recall that, in order to implement the prefix search, we have to repeat the above process for the two strings P and $P\#$, so we have proved the following:

THEOREM 9.2 *Prefix search over $\mathcal{D}$ takes $O(\frac{p}{B}\log\frac{N}{B})$ I/Os. Retrieving the strings prefixed by P needs $\frac{N_{occ}}{B}$ I/Os, where N_{occ} is their length. The total space is $N + (1 + w)n_B$ bytes.*

Proof Once the block of strings $A[i, j]$ prefixed by P has been identified, we can report all of them in $O(\frac{N_{occ}}{B})$ I/Os; scanning the contiguous portion of S that contains those strings. The space occupancy comes from the observation that pointers are stored only for the n_B sampled strings. ■

Typically strings are shorter than B, so $\frac{N}{B} \leq n$, hence this solution is faster than the previous one, in addition it can be effectively combined with the technique called *Front-Coding compression* to further lowering the space and I/O-complexities.

9.1.2 Front Coding

Given a sequence of sorted strings is probable that adjacent strings share a common prefix. If ℓ is the number of shared characters, then we can substitute them with a proper variable-length binary encoding of ℓ thus saving some bits with respect to the classic fixed-size encoding based on 4- or 8-bytes. A following chapter will detail some of those encoders, here we introduce a simple one to satisfy the curiosity of the reader. The encoder pads the binary representation of ℓ with 0 until an integer number of bytes is used. The first two bits of the padding (if any, otherwise one more byte is added), are used to encode the number of bytes used for the encoding.[1] This encoding is prefix-free and thus guarantees unique decoding properties; its byte-alignment also ensures speed in current processors.

More efficient encoders are available, anyway this simple proposal ensures to replace the initial $\Theta(\ell \log_2 \sigma)$ bits, representing ℓ characters of the shared prefix, with $O(\log \ell)$ bits of the integer encoding, so resulting advantageous in space. Obviously its final impact depends on the amount of shared characters which, in the case of a dictionary of URLs, can be up to 70%.

Front coding is a *delta*-compression algorithm, which can be easily defined in an incremental way: given a sequence of strings $(s_1, \ldots, s_n)$, it encodes the string s_i using the couple $(\ell_i, \hat{s}_i)$, where ℓ_i is the length of the longest common prefix between s_i and its predecessor s_{i-1} (0 if $i = 1$) and $\hat{s}_i = s_i[\ell_i + 1, |s_i|]$ is the "remaining suffix" of the string s_i. As an example consider the dictionary $\mathcal{D} = \{$ `alcatraz, alcool, alcyone, anacleto, ananas, aster, astral, astronomy` $\}$; its front-coded representation is (0, `alcatraz`), (3, `ool`), (3, `yone`), (1, `nacleto`), (3, `nas`), (1, `ster`), (3, `ral`), (4, `onomy`).

Decoding a string a pair $(\ell, \hat{s})$ is symmetric, we have to copy ℓ characters from the previous string in the sequence and then append the remaining suffix $\hat{s}$. This takes $O(|s|)$ optimal time and $O(1 + \frac{|s|}{B})$ I/Os, provided that the preceding string is available. In general, the reconstruction of a string s_i may require to scan back the input sequence up to the first string s_1, which is available in its entirety. So we may possibly need to scan $(\hat{s}_1, \ell_1), \ldots, (\hat{s}_{i-1}, \ell_{i-1})$ and reconstruct $s_1, \ldots, s_{i-1}$ in order to decode $(\ell_i, \hat{s}_i)$. Therefore, the time cost to decode s_i might be much higher than the optimal $O(|s_i|)$ cost.[2]

To overcome this drawback, it is typical to apply front-coding to block of strings thus resorting the two-level scheme we introduced in the previous subsection. The idea is to restart the front-coding at the beginning of every block, so the first string of each block is stored *uncompressed*. This has two immediate advantages onto the prefix-search problem: (1) these uncompressed strings are the ones participating in the binary-search process and thus they do not need to be decompressed when compared with Q; (2) each block is compressed individually and thus the scan of its strings for searching Q can be combined with the decompression of these strings without incurring in any slowdown. We call this storage scheme "Front-coding with bucketing", and shortly denote it by FC_B. Figure 9.2 provides a running example in which the strings "alcatraz", "alcyone", "ananas", and "astral" are stored explicitly because they are the first of each block.

As a positive side-effect, this approach reduces the number of sampled strings because it can potentially increase the number of strings stuffed in one disk page: we start from s_1 and we front-compress the strings of $\mathcal{D}$ in order; whenever the compression of a string s_i overflows the current block, it starts a new block where it is stored *uncompressed*. The number of sampled strings lowers from about $\frac{N}{B}$ to about $\frac{FC_B(\mathcal{D})}{B}$ strings, where $FC_B(\mathcal{D})$ is the space required by FC_B to store all the dictionary strings. This impacts positively onto the number of I/Os needed for a prefix search in a obvious manner, given that we execute a binary search over the sampled strings. However space

[1] We are assuming that ℓ can be binary encoded in 30 bits, namely $\ell < 2^{30}$.

[2] A smarter solution would be to reconstruct only the first ℓ characters of the previous strings $s_1, s_2, \ldots, s_{i-1}$ because these are the ones interesting for s_i's reconstruction.

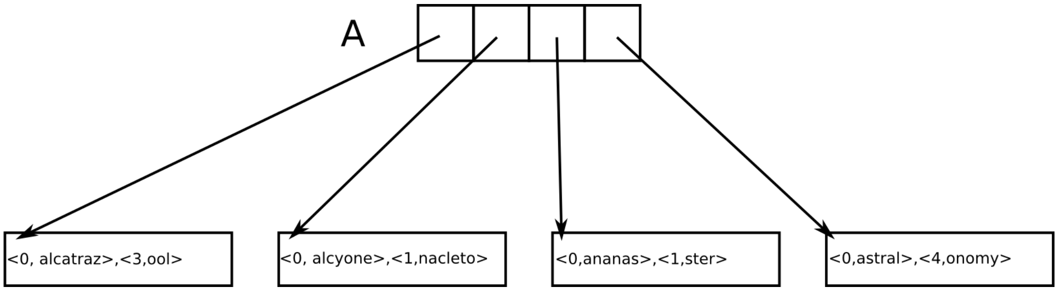


FIGURE 9.2: Two-level indexing of the set of strings $\mathcal{D}$ = { `alcatraz, alcool, alcyone, anacleto, ananas, aster, astral, astronomy`} are compressed with FC_B, where we assumed that each page is able to store two strings.

occupancy increases with respect to $FC(\mathcal{D})$ because $FC_B(\mathcal{D})$ forces the first string of each block to be stored uncompressed; nonetheless, we expect that this increase is negligible because $B \gg 1$.

THEOREM 9.3 *Prefix search over $\mathcal{D}$ takes $O(\frac{p}{B} \log \frac{FC_B(\mathcal{D})}{B}$ I/Os. Retrieving the strings prefixed by P needs $O(\frac{FC_B(\mathcal{D}_{occ})}{B})$ I/Os, where $\mathcal{D}_{occ} \subseteq \mathcal{D}$ is the set of strings in the answer set.*

So, in general, compressing the strings is a good idea because it lowers the space required for storing the strings, and it lowers the number of I/Os. However we must observe that FC-compression might increase the time complexity of the scan of a block from $O(B)$ to $O(B^2)$ because of the decompression of that block. In fact, take the sequence of strings $(a, aa, aaa, \ldots)$ which is front coded as $(0, a), (1, a), (2, a), (3, a), \ldots$. In one disk page we can stuff $\Theta(B)$ such pairs, which represent $\Theta(B)$ strings whose total length is $\sum_{i=0}^{B} \Theta(i) = \Theta(B^2)$ characters. Despite these pathological cases, in practice the space reduction consists of a *constant factor* so the time increase incurred by a block scan is negligible.

Overall this approach introduces a time/space trade-off driven by the block size B. As far as time is concerned we can observe that the longer is B, the better is the compression ratio but the slower is a prefix search because of a longer scan-phase; conversely, the shorter is B, the faster is the scan-phase but the worse is the compression ratio because of a larger number of fully-copied strings. As far as space is concerned, the longer is B, the smaller is the number of copied strings and thus the smaller is the storage space in internal memory needed to index their pointers; conversely, the shorter is B, the larger is the number of pointers thus making probably impossible to fit them in internal memory.

In order to overcome this trade-off we decouple search and compression issues as follows. We notice that the proposed data structure consists of *two* levels: the “upper” level contains references to the *sampled strings* $\mathcal{D}_B$, the “lower” level contains the strings themselves stored in a block-wise fashion. The choice of the algorithms and data structures used in the two levels are “orthogonal” to each other, and thus can be decided independently. It goes without saying that this 2-level scheme for searching-and-storing a dictionary of strings is suitable to be used in a hierarchy of two memory levels, such as the cache and the internal memory. This is typical in Web search, where $\mathcal{D}$ is the dictionary of terms to be searched by users and disk-accesses have to be avoided in order to support each search over $\mathcal{D}$ in few millisecs.

In the next three sections we propose three improvements to the 2-level solution above, two of them regard the first level of the sampled strings, one concerns with the compressed storage of all dictionary strings. Actually, these proposals have an interest in themselves and thus the reader should not confine their use to the one described in these notes.

9.2 Interpolation search

Until now, we have used binary search over the array A of string pointers. But if $\mathcal{D}_B$ satisfies some statistical properties, there are searching schemes which support faster searches, such as the well known *interpolation search*. In what follows we describe a variant of classic interpolation search which offers some interesting additional properties (details in [3]). For simplicity of presentation we describe the algorithm in terms of a dictionary of integers, knowing that if items are strings, we can still look at them as integers in base σ. So for the prefix-search problem we can pad logically all strings at their end, thus getting to the same length, by using a character that we assume to be smaller than any other alphabet character. Lexicographic order among strings is turned in classic ordering of integers, so that the search for P and $P\#$ can be turned into a search for two proper integers.

So without loss of generality, assume that $\mathcal{D}_B$ is an array of integers $X[1, m] = x_1 \ldots x_m$ with $x_i < x_{i+1}$ and $m = n_B$. We evenly subdivide the range $[x_1, x_m]$ into m bins $B_1, \ldots, B_m$ (so we consider as many bins as integers in X), each bin representing a contiguous range of integers having length $b = \frac{x_m - x_1 + 1}{m}$. Specifically $B_i = [x_1 + (i-1)b, x_1 + ib)$. In order to guarantee the constant-time access to these bins we need to keep an additional array, say $I[1, m]$, such that $I[i]$ points to the first and last item of B_i in X.

Figure 9.3 reports an example where $m = 12$, $x_1 = 1$ and $x_{12} = 36$ and thus the bin length is $b = 3$.

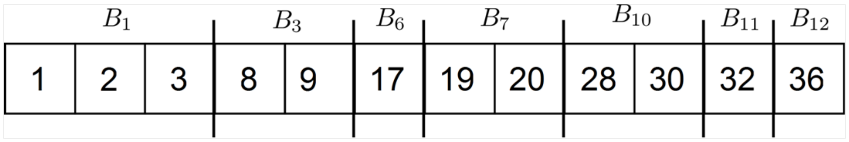


FIGURE 9.3: An example of use of interpolation search over an itemset of size 12. The bins are separated by bars; some bins, such as B_4 and B_8, are empty.

The algorithm searches for an integer y in two steps. In the first step it calculates j, the index of the candidate bin B_j where y could occur: $j = \lfloor \frac{y - x_1}{b} \rfloor + 1$. In the second step, it determines via $I[j]$ the sub-array of X which stores B_j and it does a binary search over it for y, thus taking $O(\log |B_i|) = O(\log b)$ time. The value of b depends on the magnitude of the integers present in the indexed dictionary. Surprisingly enough, we can get a better bound which takes into account the distribution of the integers of X in the range $[x_1, x_m]$.

THEOREM 9.4 *We can search for an integer in a dictionary of size m taking $O(\log \Delta)$ time in the worst case, where Δ is the ratio between the maximum and the minimum gap between two consecutive integers of the input dictionary. The extra space is $O(m)$.*

Proof Correctness is immediate. For the time complexity, we observe that the maximum of a series of integers is at least as large as their mean. Here we take as those integers the gaps $x_i - x_{i-1}$, and write:

$$\max_{i=2\ldots m}(x_i - x_{i-1}) \geq \frac{\sum_{i=2}^{m} x_i - x_{i-1}}{m-1} \geq \frac{x_m - x_1 + 1}{m} = b \tag{9.1}$$

The last inequality comes from the following arithmetic property: $\frac{a'}{a''} \geq \frac{a'+1}{a''+1}$ whenever $a' \geq a''$, which can be easily proved by solving it.

Another useful observation concerns with the maximum number of integers that can belong to any bin. Since integers of X are spaced apart by $s = \min_{i=2,\ldots,m}(x_i - x_{i-1})$ units, every bin contains no more than b/s integers.

Recalling the definition of Δ, and the two previous observations, we can thus write:

$$|B_i| \leq \frac{b}{s} \leq \frac{\max_{i=2\ldots m}(x_i - x_{i-1})}{\min_{i=2,\ldots,m}(x_i - x_{i-1})} = \Delta$$

.

So the theorem follows due to the binary search performed within B_i. Space occupancy is optimal and equal to $O(m)$ because of the arrays $X[1, m]$ and $I[1, m]$. ■

We note the following interesting properties of the proposed algorithm:

- The algorithm is oblivious to the value of Δ, although its complexity can be written in terms of this value.
- The worst-case search time is $O(\log m)$, when the whole X ends up in a single bin, and thus the precise bound should be $O(\log \min\{\Delta, m\})$. So it cannot be worst than the binary search.
- The space occupancy is $O(m)$ which is optimal asymptotically; however, it has to be notice that binary search is in-place, whereas interpolation search needs the extra array $I[1, m]$.
- The algorithm reproduces the $O(\log\log m)$ time performance of classic interpolation search on data drawn independently from the uniform distribution, as shown in the following lemma. We observe that, uniform distribution of the X's integers is uncommon in practice, nevertheless we can artificially enforce it by selecting a random permutation $\pi : U \longrightarrow U$ and shuffling X according to π before building the proposed data structure. Care must be taken at query time since we search not y but its permuted image $\pi(y)$ in $\pi(X)$. This way the query performance proved below holds with high probability whichever is the indexed set X. For the choice of π we refer the reader to [6].

LEMMA 9.1 If the m integers are drawn uniformly at random from $[1, U]$, the proposed algorithm takes $O(\lg\lg m)$ time with high probability.

Proof Say integers are uniformly distributed over $[0, U - 1]$. As in bucket sort, every bucket B_i contains $O(1)$ integers on average. But we wish to obtain bounds with high probability. So let us assume to partition the integers in $r = \frac{m}{2\log m}$ ranges. We have the probability $1/r$ that an integer belongs to a given range. The probability that a given range does not contain any integer is $(1 - \frac{1}{r})^m = (1 - \frac{2\log m}{m})^m = O(e^{-2\log m}) = O(1/m^2)$. So the probability that at least one range remains empty is smaller than $O(1/m)$; or, equivalently, with high probability every range contains at least one integer.

If this occurs with high probability, the maximum distance between two adjacent integers must be smaller than twice the range's length: namely $\max_i(x_i - x_{i-1}) \leq 2U/r = O(\frac{U \log m}{m})$.

Let us now take $r' = \Theta(m \log m)$ ranges, similarly as above we can prove that every adjacent pair of ranges contains at most one integer with high probability. Therefore if a range contains an integer, its two adjacent ranges (on the left and on the right) are empty with high probability. Thus we can lower bound the minimum gap with the length of one range: $\min_i(x_i - x_{i-1}) \geq U/r' = \Theta(\frac{U}{m \log m})$. Taking the ratio between the minimum and the maximum gap, we get the desired $\Delta = O(\log^2 m)$. ■

If this algorithm is applied to our string context, and strings are uniformly distributed, the number of I/Os required to prefix-search P in the dictionary $\mathcal{D}$ is $O(\frac{p}{B} \log\log \frac{N}{B})$. This is an exponential reduction in the search time performance according to the dictionary length.

9.3 Locality-preserving front coding

This is an elegant variant of front coding which provides a controlled trade-off between space occupancy and time to decode one string [2]. The key idea is simple, and thus easily implementable, but proving its guaranteed bounds is challenging. We can state the underlying algorithmic idea as follows: *a string is front-coded only if its decoding time is proportional to its length, otherwise it is written uncompressed.* The outcome in time complexity is clear: we compress only if decoding is optimal. But what appears surprising is that, even if we concentrated on the time-optimality of decoding, its “constant of proportionality” controls also the space occupancy of the compressed strings. It seems magic, indeed it is!

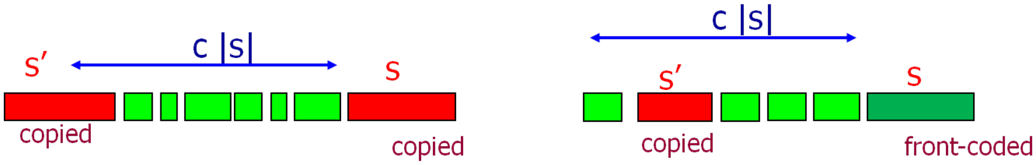


FIGURE 9.4: The two cases occurring in LPFC. Red rectangles are copied strings, green rectangles are front-coded strings.

Formally, suppose that we have front-coded the first $i-1$ strings $(s_1, \ldots, s_{i-1})$ into the compressed sequence $\mathcal{F} = (0, \hat{s}_1), (\ell_2, \hat{s}_2), \ldots, (\ell_{i-1}, \hat{s}_{i-1})$. We want to compress s_i so we scan backward at most $c|s_i|$ characters of $\mathcal{F}$ to check whether these characters are enough to reconstruct s_i. This actually means that an uncompressed string is included in those characters, because we have available the first character for s_i. If so, we front-compress s_i into $(\ell_i, \hat{s}_i)$; otherwise s_i is copied uncompressed in $\mathcal{F}$ outputting the pair $(0, s_i)$. The key difficulty here is to show that the strings which are left uncompressed, and were instead compressed by the classic front-coding scheme, have a length that can be controlled by means of the parameter c as the following theorem shows:

THEOREM 9.5 *Locality-preserving front coding takes at most* $(1+\epsilon)FC(\mathcal{D})$ *space, and supports the decoding of any dictionary string* s_i *in* $O(\frac{|s_i|}{\epsilon B})$ *optimal I/Os.*

Proof We call any uncompressed string s, a *copied* string, and denote the $c|s|$ characters explored during the backward check as the *left extent* of s. Notice that if s is a copied string, there can be no copied string preceding s and beginning in its left extent (otherwise it would have been front-coded). Moreover, the copied string that precedes S may *end* within s's left extent. For the sake of presentation we call *FC-characters* the ones belonging to the output suffix $\hat{s}$ of a front-coded string s.

Clearly the space occupied by the front-coded strings is upper bounded by $FC(\mathcal{D})$. We wish to show that the space occupied by the copied strings, which were possibly compressed by the classic front-coding but are left uncompressed here, sums up to $\epsilon\, FC(\mathcal{D})$, where ϵ is a parameter depending on c and to be determined below.

We consider two cases for the copied strings depending on the amount of FC-characters that lie between two consecutive occurrences of them. The first case is called *uncrowded* and occurs when that number of FC-characters is at least $\frac{c|s|}{2}$; the second case is called *crowded*, and occurs when that number of FC-characters is at most $\frac{c|s|}{2}$. Figure 9.5 provides an example which clearly shows that if the copied string s is crowded then $|s'| \geq c|s|/2$. In fact, s' starts before the left extent of s but ends within the last $c|s|/2$ characters of that extent. Since the extent is $c|s|$ characters long, the above observation follows. If s is uncrowded, then it is preceded by at least $c|s|/2$ characters of front-coded strings (FC-characters).

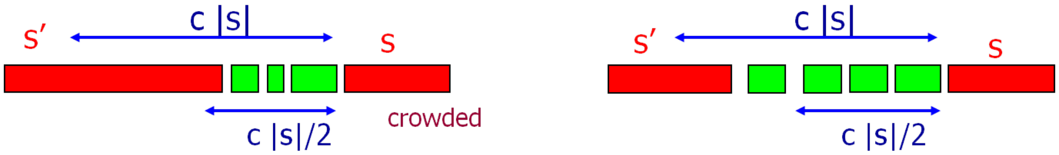


FIGURE 9.5: The two cases occurring in LPFC. The green rectangles denote the front-coded strings, and thus their FC-characters, the red rectangles denote the two consecutive copied strings.

We are now ready to bound the total length of copied strings. We partition them into chains composed by one uncrowded copied-string followed by the maximal sequence of crowded copied-strings. In what follows we prove that the total number of characters in each chain is proportional to the length of its first copied-string, namely the uncrowded one. Precisely, consider the chain $w_1 w_2 \cdots w_x$ of consecutive copied strings, where w_1 is uncrowded and the following w_is are crowded. Take any crowded w_i. By the observation above, we have that $|w_{i-1}| \geq c|w_i|/2$ or, equivalently, $|w_i| \leq 2|w_{i-1}|/c = \cdots = (2/c)^{i-1}|w_1|$. So if $c > 2$ the crowded copied strings shrink by a constant factor. We have $\sum_i |w_i| = |w_1| + \sum_{i>1} |w_i| \leq |w_1| + \sum_{i>1} (2/c)^{i-1}|w_1| = |w_1| \sum_{i\geq 0}(2/c)^i < \frac{c|w_1|}{c-2}$.

Finally, since w_1 is uncrowded, it is preceded by at least $c|w_1|/2$ FC-characters (see above). The total number of these FC-characters is bounded by $FC(\mathcal{D})$, so we can upper bound the total length of the uncrowded strings by $(2/c)FC(\mathcal{D})$. By plugging this into the previous bound on the total length of the chains, we get $\frac{c}{c-2} \times \frac{2FC(\mathcal{D})}{c} = \frac{2}{c-2}FC(\mathcal{D})$. The theorem follows by setting $\epsilon = \frac{2}{c-2}$. ■

So locality-preserving front coding (shortly LPFC) is a compressed storage scheme for strings that can substitute their plain storage without introducing any asymptotic slowdown in the accesses to the compressed strings. In this sense it can be considered as a sort of *space booster* for any string indexing technique.

The two-level indexing data-structure described in the previous sections can benefit of LPFC as follows. We can use A to point to the copied strings of LPFC (which are uncompressed). This way the buckets delimited by these strings have variable length, but any string can be decompressed in optimal time and I/Os (cfr. previous observation about the $\Theta(B^2)$ size of a bucket in classic FC_B). So the bounds are the ones stated in Theorem 9.3 but without the pathological cases commented next to its proof. This way the scanning of a bucket, identified by the binary-search step takes $O(1)$ I/O and time proportional to the returned strings, and hence it is optimal.

The remaining question is therefore how to speed-up the search over the array A. We foresee two main limitations: (i) the binary-search step has time complexity depending on N or n, (ii) if the pointed strings do not fit within the internal-memory space allocated by the programmer, or available in cache, then the binary-search step incurs many I/Os, and this might be expensive. In the next sections we propose a trie-based approach that takes full-advantage of LPFC by overcoming these limitations, resulting efficient in time, I/Os and space.

9.4 Compacted Trie

We already talked about tries in Chapter 6, here we dig further into their properties as efficient data structures for string searching. In our context, the trie is used for the indexing of the sampled strings $\mathcal{D}_B$ in internal memory. This induces a speed up in the first stage of the prefix search from $O(\log(N/B))$ to $O(p)$ time, thus resulting surprisingly independent of the dictionary size. The reason is the power of the RAM model which allows to manage and address memory-cells of $O(\log N)$ bits in constant time.

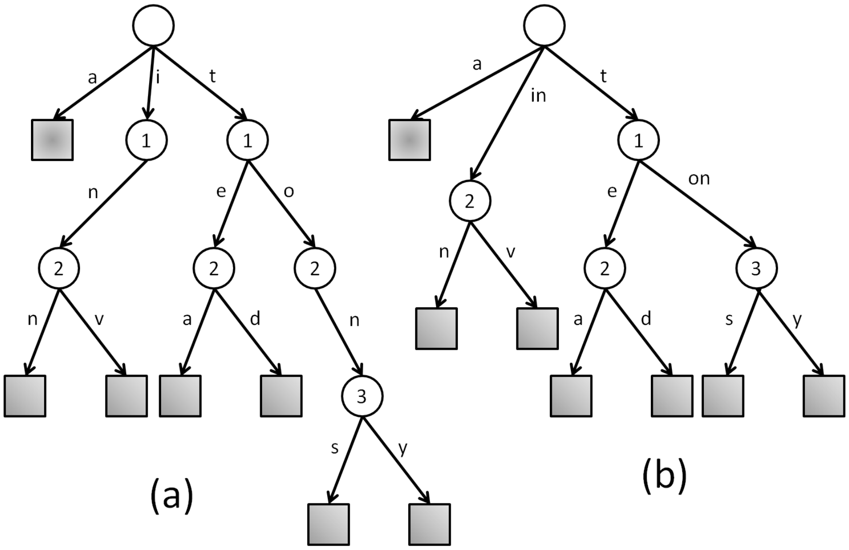


FIGURE 9.6: An example of uncompacted trie (a) and compacted trie (b) for $n = 7$ strings. The integer showed in each internal node u denotes the length of the string spelled out by u. In the case of uncompacted tries they are useless because they correspond to u's depth. Edge labels in compacted tries are substrings of variable length but they can be represented in $O(1)$ space with triples of integers: e.g. on could be encoded as $\langle 6, 2, 3\rangle$, since the 6-th string tons includes on from position 2 to position 3.

A trie is a multi-way tree whose edges are labeled by characters of the indexed strings. An internal node u is associated with a string $s[u]$ which is indeed a *prefix* of a dictionary string. String $s[u]$ is obtained by concatenating the characters found on the downward path that connects the trie's root with the node u. A leaf is associated with a dictionary string. All leaves which descend from a node u are prefixed by $s[u]$. The trie has n leaves and at most N nodes, one per string character.[3] Figure 9.6 provides an illustrative example of a trie built over 6 strings. This form of trie is commonly called *uncompacted* because it can have *unary paths*, such as the one leading to string `inn`.[4]

If we want to check if a string P prefixes some dictionary string, we have just to check if there is a downward path spelling out P. All leaves descending from the reached node provide the correct answer to our prefix search. So tries do not need the reduction to the lexicographic search operation, introduced for the binary-search approach.

A big issue is how to efficiently find the "edge to follow" during the downward traversal of the trie, because this impacts onto the overall efficiency of the pattern search. The efficiency of this step hinges on a proper storage of the edges (and their labeling characters) outgoing from a node. The simplest data structure that does the job is the *linked list*. Its space requirement is optimal, namely proportional to the number of outgoing edges, but it incurs in a $O(\sigma)$ cost per traversed node. The result would be a prefix search taking $O(p\ \sigma)$ time in the worst case, which is too much for large alphabets. If we store the branching characters (and their edges) into a sorted array, then we could binary search it taking $O(\log \sigma)$ time per node. A faster approach consists of using a full-sized array of σ entries, the un-empty entries (namely the ones for which the pointer in not null) are the entries corresponding to the existing branching characters. In this case the time to branch out of a node is $O(1)$ and thus $O(p)$ time is the cost for searching the pattern Q. But the space occupancy of the trie grows up to $O(N\sigma)$, which may be unacceptably high for large alphabets. The best approach consists of resorting a *perfect hash table*, which stores just the existing branching characters and their associated pointers. This guarantees $O(1)$ branching time in the worst-case and optimal space occupancy, thus combining the best of the two previous solutions. For details about perfect hashes we refer the reader to Chapter 7.

THEOREM 9.6 *The uncompacted trie solves the prefix-search problem in $O(p + n_{occ})$ time and $O(p + n_{occ}/B)$ I/Os, where n_{occ} is the number of strings prefixed by P. The retrieval of those strings prefixed by P takes $O(N_{occ})$ time, and it takes $O(N_{occ}/B)$ I/Os provided that leaves and strings are stored contiguously and alphabetically sorted on disk. The trie consists of at most N nodes, exactly n leaves, and thus takes $O(N)$ space. The retrieval of the result strings takes $O(N_{occ})$ time and $O(N_{occ}/B)$ I/Os, where N_{occ} is the total length of the retrieved strings.*

Proof Let u be the node such that $s[u] = P$. All strings descending from u are prefixed by P, and they can be visualized by visiting the subtree rooted in u. The I/O-complexity of the traversal is still $O(p)$ because of the jumps among trie nodes. The retrieval of the n_{occ} leaves descending from the node spelling Q takes optimal $O(n_{occ}/B)$ I/Os because we can assume that trie leaves are stored contiguously from left-to-right on disk. Notice that we have identified the strings (leaves) prefixed by Q but, in order to display them, we still need to retrieve them, this takes additional $O(N_{occ}/B)$ I/Os provided that the indexed strings are stored contiguously on disk. This is $O(n_{occ}/B)$ I/Os if we are interested only in the string pointers/IDs, provided that every internal node keeps a pointer to its leftmost descending leaf and all leaves are stored contiguously on disk. (These are the main reasons

[3] We say "at most" because some paths (prefixes) can be shared among several strings.

[4] The trie cannot index strings which are one the prefix of the other. In fact the former string would end up into an internal node. To avoid this case, each string is extended with a special character which is not present in the alphabet and is typically denoted by \$.

for keeping the pointers in the leaves of the uncompacted trie, which anyway stores the strings in its edges, and thus could allow to retrieve them but with more I/Os because of the difficulty to pack arbitrary trees on disk.) ■

A Trie can be wasteful in space if there are long strings with a short common prefix: this would induce a significant number of unary nodes. We can save space by *contracting* the unary paths into one single edge. This way edge labels become (possibly long) sub-strings rather than characters, and the resulting trie is named *compacted.* Figure 9.7 (left) shows an example of compacted trie. It is evident that each edge-label is a substring of a dictionary string, say $s[i, j]$, so it can be represented via a triple $\langle s, i, j\rangle$ (see also Figure 9.6). Given that each node is at least binary, the number of internal nodes and edges is $O(n)$. So the total space required by a compacted trie is $O(n)$ too.

Prefix searching is implemented similarly as done for uncompacted tries. The difference is that it alternates character-branches out of internal nodes, and sub-string matches with edge labels. If the edges spurring from the internal nodes are again implemented with perfect hast tables, we get:

THEOREM 9.7 *The compacted trie solves the prefix-search problem in $O(p + n_{occ})$ time and $O(p + n_{occ}/B)$ I/Os, where n_{occ} is the number of strings prefixed by P. The retrieval of those strings prefixed by P takes $O(N_{occ})$ time, and and it takes $O(N_{occ}/B)$ I/Os provided that leaves and strings are stored contiguously and alphabetically sorted on disk. The compacted trie consists of $O(n)$ nodes, and thus its storage takes $O(n)$ space. It goes without saying that the trie needs also the storage of the dictionary strings to resolve its edge labels, hence additional N space.*

At this point an attentive reader can realize that the compacted trie can be used also to search for the lexicographic position of a string Q among the indexed strings. It is enough to percolate a downward path spelling Q as much as possible until a mismatch character is encountered. This character can then be deployed to determine the lexicographic position of Q, depending on whether the percolation stopped in the middle of an edge or in a trie node. So the compacted trie is an interesting substitute for the array A in our two-level indexing structure and could be used to support the search for the candidate bucket where the string Q occurs in, taking $O(p)$ time in the worst case. Since each traversed edge can induce one I/O, to fetch its labeling substring to be compared with the corresponding one in Q, we point out that this approach is efficient if the trie and its indexed strings can be fit in internal memory. Otherwise it presents two main drawbacks: the linear dependance of the I/Os on the pattern length p, and the space dependance on the block-size B (influencing the the sampling) and the length of the sampled strings.

The *Patricia Trie* solves the former problem, whereas its combination with the LPFC solves both of them.

9.5 Patricia Trie

A Patricia Trie built on a string dictionary is a compacted Trie in which the edge labels consist just of their initial *single characters*, and the internal nodes are labeled with integers denoting the *lengths* of the associated strings. Figure 9.7 illustrates how to convert a Compacted Trie (left) into a Patricia Trie (right).

Even if the Patricia Trie strips out some information from the Compacted Trie, it is still able to support the search for the lexicographic position of a pattern P among a (sorted) sequence of strings, with the significant advantage (discussed below) that this search needs to access only one single string, and hence execute typically one I/O instead of the p I/Os potentially incurred by the edge-resolution step in compacted tries. This algorithm is called *blind search* in the literature [4].

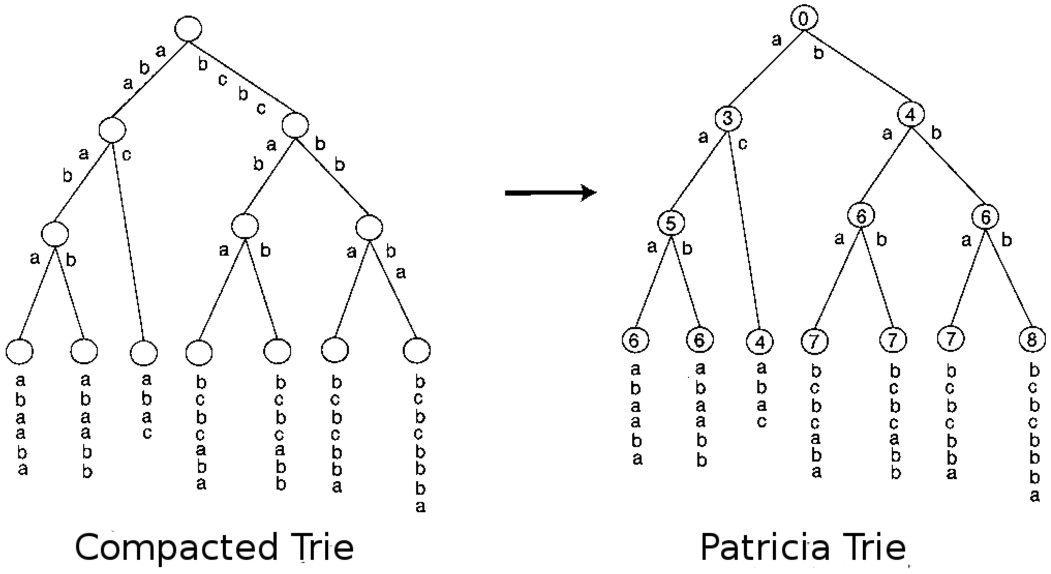


FIGURE 9.7: An example of Compacted Trie and the corresponding Patricia Trie.

It is a little bit more complicated than the prefix-search in classic tries, because of the presence of only one character per edge label, and in fact it consists of three stages:

- Trace a downward path in the Patricia Trie to locate a leaf l which points to an interesting string of the indexed dictionary. This string does not necessarily identify P's lexicographic position in the dictionary (which is our goal), but it provides *enough information* to find that position in the second stage. The retrieval of the interesting leaf l is done by traversing the Patricia Trie from the root and comparing the characters of P with the single characters which label the traversed edges until either a leaf is reached or no further branching is possible. In this last case, we choose l to be any descendant leaf from the last traversed node.
- Compare P against the string pointed by leaf l, in order to determine their longest common prefix. Let ℓ be the length of this shared prefix, then it is possible to prove that (see [4]) the leaf l stores one of the strings indexed by the Patricia Trie that shares the longest common prefix with P. Call s this pointed string. The length ℓ and the two mismatch characters $P[\ell + 1]$ and $s[\ell + 1]$ are then used to find the lexicographic position of P among the strings stored in the Patricia Trie.
- First the Patricia trie is traversed upward from l to determine the edge $e = (u, v)$ where the mismatch character $s[\ell + 1]$ lies; this is easy because each node on the upward path stores an integer that denotes the length of the corresponding prefix of s, so that we have $|s[u]| < \ell \leq |s[v]|$. If $s[\ell + 1]$ is a branching character (i.e. $\ell = |s[u]|$), then we determine the lexicographic position of $P[\ell + 1]$ among the branching characters of node u. Say this is the i-th child of u, the lexicographic position of P is therefore to the immediate left of the subtree descending from this child. Otherwise (i.e. $\ell > |s[u]|$), the character $s[\ell + 1]$ lies within e, so the lexicographic position of P is to the immediate right of the subtree descending from e, if $P[\ell + 1] > s[\ell + 1]$, otherwise it is to the immediate left of that subtree.

A running example is illustrated in Figure 9.8.

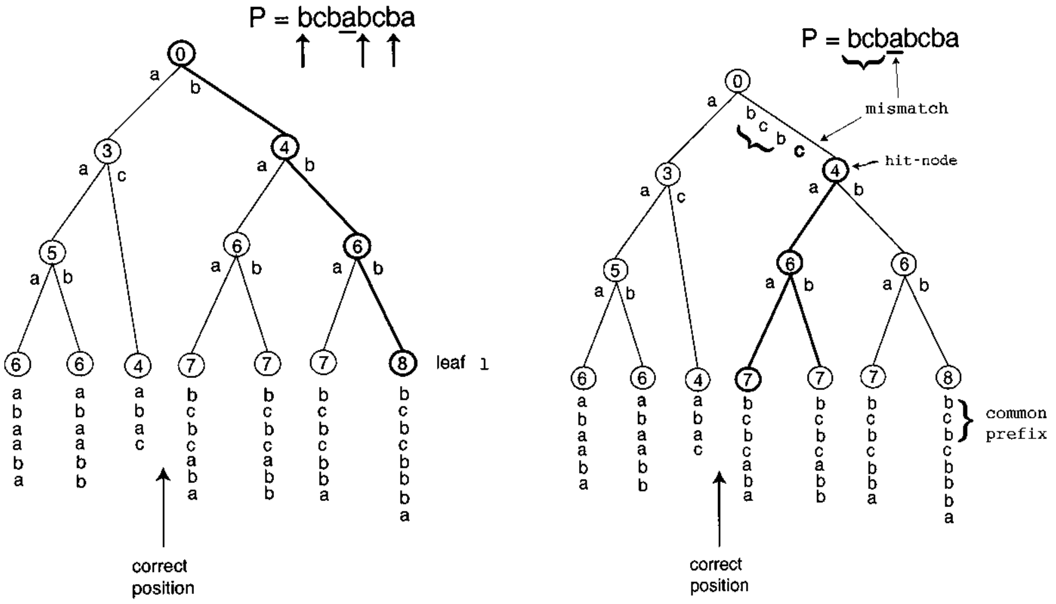


FIGURE 9.8: An example of the first (left) and second (right) stages of the blind search for P in a dictionary of 7 strings.

In order to understand why the algorithm is correct, let us take the path spelling out the string $P[1, \ell]$. We have two cases, either we reached an internal node u such that $|s[u]| = \ell$ or we are in the middle of an edge (u, v), where $|s[u]| < \ell < |s[v]|$. In the former case, all strings descending from u are the ones in the dictionary which share ℓ characters with the pattern, and this is the *lcp*. The correct lexicographic position therefore falls among them or is adjacent to them, and thus it can be found by looking at the branching characters of the edges outgoing from the node u. This is correctly done also by the blind search that surely stops at u, computes ℓ and finally determines the correct position of P by comparing u's branching characters against $P[\ell + 1]$.

In the latter case the blind search reaches v by skipping the mismatch character on (u, v), and possibly goes further down in the trie because of the possible match between branching characters and further characters of P. Eventually a leaf descending from v is taken, and thus ℓ is computed correctly given that all leaves descending from v share ℓ characters with P. So the backward traversal executed in the second stage of the Blind search reaches correctly the edge (u, v), which is above the selected leaf. There we deploy the mismatch character which allows to choose the correct lexicographic position of P which is either to the left of the leaves descending from v or to their right. Indeed all those leaves share $|s[v]| > \ell$ characters, and thus P falls adjacent to them, either to their left or to their right. The choice depends on the comparison between the two characters $P[\ell + 1]$ and $s[v][\ell + 1]$.

The blind search has excellent performance:

THEOREM 9.8 *A Patricia trie takes $O(n)$ space, hence $O(1)$ space per indexed string (in-*

dependent, therefore, of its length). The blind search for a pattern $P[1,p]$ requires $O(p)$ time to traverse the trie's structure (downward and upward), and $O(p/B)$ I/Os to compare the single string (possibly residing on disk) identified by the blind search. It returns the lexicographic position of P among the indexed strings. By searching for P and P#, as done in suffix arrays, the blind search determines the range of indexed strings prefixed by P, if any.

This theorem states that if $n < M$ then we can index in internal memory the whole dictionary, and thus build the Patricia trie over all dictionary strings and stuff it in the internal memory of our computer. The dictionary strings are stored on disk. The prefix search for a pattern P takes in $O(p)$ time and $O(p/B)$ I/Os. The total required space is the one needed to store the strings, and thus it is $O(N)$.

If we wish to compress the dictionary strings, then we need to resort front-coding. More precisely, we combine the Patricia Trie and LPFC as follows. We fit in the internal memory the Patricia trie of the dictionary $\mathcal{D}$, and store on disk the locality-preserving front coding of the dictionary strings. The two traversals of the Patricia trie take $O(p)$ time and no I/Os (Theorem 9.8), because use information stored in the Patricia trie and thus available in internal memory. Conversely the computation of the lcp takes $O(|s|/B + p/B)$ I/Os, because it needs to decode from its LPFC-representation (Theorem 9.5) the string s selected by the blind search and it also needs to compare s against P to compute their lcp. These information allow to identify the lexicographic position of P among the leaves of the Patricia trie.

THEOREM 9.9 *The data structure composed of the Patricia Trie as the index in internal memory ("upper level") and the LPFC for storing the strings on disk ("lower level") requires $O(n)$ space in memory and $O((1+\epsilon)FC(\mathcal{D}))$ space on disk. Furthermore, a prefix search for P requires $O(\frac{p}{B} + \frac{|s|}{B\epsilon})$ I/Os, where s is the "interesting string" determined in the first stage of the Blind search. The retrieval of the prefixed strings takes $O(\frac{(1+\epsilon)FC(\mathcal{D}_{occ})}{B})$ I/Os, where $\mathcal{D}_{occ} \subseteq \mathcal{D}$ is the set of returned strings.*

In the case that $n = \Omega(M)$, we cannot index in the internal-memory Patricia trie the whole dictionary, so we have to resort the bucketing strategy over the strings stored on disk and index in the Patricia trie only a sample of them. If $N/B = O(M)$ we can index in internal memory the first string of every bucket and thus be able to prefix-search P within the bounds stated in Theorem 9.9, by adding just one I/O due to the scanning of the bucket (i.e. disk page) containing the lexicographic position of P. The previous condition can be rewritten as $N = O(MB)$ which is pretty reasonable in practice, given the current values of $M \approx$ 4Gb and $B \approx$ 32Kb, which make $MB \approx$ 128Tb.

9.6 Managing Huge Dictionaries$^{\infty}$

The final question we address in this lecture is: What if $N = \Omega(MB)$? In this case the Patricia trie is too big to be fit in the internal memory of our computer. We can think to store the trie on disk without taking much care on the layout of its nodes among the disk pages. Unfortunately a pattern search could take $\Omega(p)$ I/Os in the two traversals performed by the Blind search. Alternatively, we could incrementally grow a root page and repeatedly add some node not already packed into that page, where the choice of that node might be driven by various criteria that either depend on some access probability or on the node's depth. When the root page contains B nodes, it is written onto disk and the algorithm recursively lays out the rest of the tree. Surprisingly enough, the obtained packing is far from optimality of a factor $\Omega(\frac{\log B}{\log\log B})$, but it is surely within a factor $O(\log B)$ from the optimal [1].

In what follows we describe two distinct optimal approaches to solve the prefix-search over dictionaries of huge size: the first solution is based on a data structure, called the *String B-Tree* [4], which boils down to a B-tree in which the routing table of each node is a Patricia tree; the second solution consists of applying proper *disk layouts of trees* onto the Patricia trie built over the entire dictionary.

9.6.1 String B-Tree

The key idea consists of dividing the big Patricia trie into a set of smaller Patricia tries, each fitting into one disk page. And then linking together all of them in a B-Tree structure. Below we outline a constructive definition of the String B-Tree, for details on this structure and the supported operations we refer the interested reader to the cited literature.

The dictionary strings are stored on disk contiguously and ordered. The pointers to these strings are partitioned into a set of smaller, equally sized chunks $\mathcal{D}_1, \ldots, \mathcal{D}_m$, each including $\Theta(B)$ strings independently of their length. This way, we can index each chunk $\mathcal{D}_i$ with a Patricia Trie that fits into one disk page and embed it into a leaf of the B-Tree. In order to search for P among those set of nodes, we take from each partition $\mathcal{D}_i$ its *first* and *last* (lexicographically speaking) strings s_{if} and s_{il}, defining the set $\mathcal{D}^1 = \left\{s_{1f}, s_{1l}, \ldots, s_{mf}, s_{ml}\right\}$.

Recall that the prefix search for P boils down to the lexicographic search of a pattern Q, properly defined from P. If we search Q within $\mathcal{D}^1$, we can discover one of the following three cases:

1. Q falls before the first or after the last string of $\mathcal{D}$, if $Q < s_{1f}$ or $Q > s_{ml}$.
2. Q falls among the strings of some $\mathcal{D}_i$, and indeed it is $s_{if} < Q < s_{il}$. So the search is continued in the Patricia trie that indexes $\mathcal{D}_i$;
3. Q falls between two chunks, say $\mathcal{D}_i$ and $\mathcal{D}_{i+1}$, and indeed it is $s_{il} < Q < s_{(i+1)f}$. So we found Q's lexicographic position in the whole $\mathcal{D}$, namely it is between these two adjacent chunks.

In order to establish which of the three cases occurs, we need to search efficiently in $\mathcal{D}^1$ for the lexicographic position of Q. Now, if $\mathcal{D}^1$ is small and can be fit in memory, we can build on it a Patricia trie ad we are done. Otherwise we repeat the partition process on $\mathcal{D}^1$ to build a smaller set $\mathcal{D}^2$, in which we sample, as before, two strings every B, so that $|\mathcal{D}^2| = \frac{2|\mathcal{D}^1|}{B}$. We continue this partitioning process for k steps, until it is $|\mathcal{D}^k| = O(B)$ and thus we can fit the Patricia trie built on $\mathcal{D}^k$ within one disk page[5].

We notice that each disk page gets an even number of strings when partitioning $\mathcal{D}^1, \ldots, \mathcal{D}^k$, and to each pair $\left(s_{if}, s_{il}\right)$ we associate a pointer to the block of strings which they delimit in the lower level of this partitioning process. The final result of the process is then a B-Tree over string pointers. The *arity* of the tree is $\Theta(B)$, because we index $\Theta(B)$ strings in each single node. The nodes of the String B-Tree are then stored on disk. The following Figure 9.9 provides an illustrative example for a String B-tree built over 7 strings.

A (prefix) search for the string P in a String B-Tree is simply the traversal of the B-Tree, which executes at each node a lexicographic search of the proper pattern Q in the Patricia trie of that node. This search discovers one of the three cases mentioned above, in particular:

- case 1 can only happen on the root node;
- case 2 implies that we have to follow the node pointer associated to the identified partition.

[5] Actually, we could stop as soon as $\left|\mathcal{D}^k\right| = O(M)$, but we prefer the former to get a standard B-Tree structure.

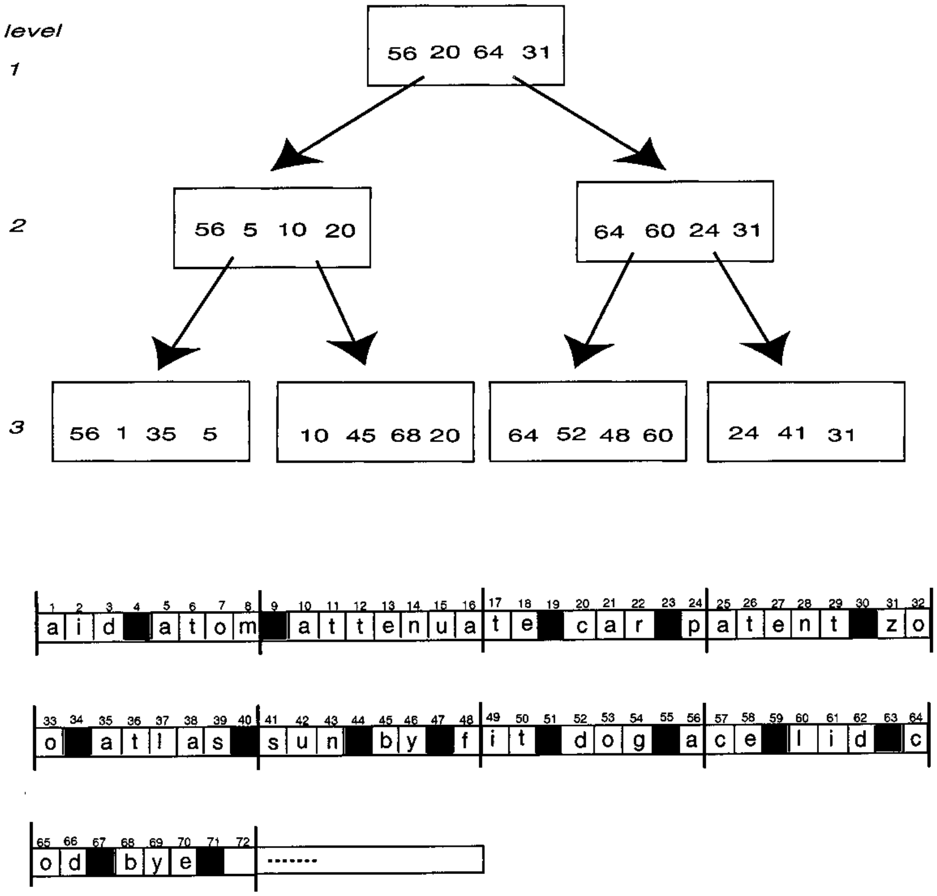


FIGURE 9.9: An example of an String B-tree on built on the suffixes of the strings in $\mathcal{D}$ = \{*'ace'*, *'aid'*, *'atlas'*, *'atom'*, *'attenuate'*, *'by'*, *'bye'*, *'car'*, *'cod'*, *'dog'*, *'fit'*, *'lid'*, *'patent'*, *'sun'*, *'zoo'*\}. The strings are stored in the B-tree leaves by means of their logical pointers 56, 1, 35, 5, 10, . . ., 31. Notice that strings are not sorted on disk, nevertheless sorting improves their I/O-scanning, and indeed our theorems assume an ordered $\mathcal{D}$ on disk.

- case 3 has found the lexicographic position of Q in the dictionary $\mathcal{D}$, so the search in the B-tree stops.

The I/O complexity of the data structure just defined is pretty good: since the arity of the B-Tree is $\Theta(B)$, we have $\Theta(\log_B n)$ levels, so a search traverses $\Theta(\log_B n)$ nodes. Since on each node we need to load the node's page into memory and perform a Blind search over its Patricia trie, we pay $O(1 + \frac{p}{B})$ I/Os, and thus $O(\frac{p}{B} \log_B n)$ I/Os for the overall prefix search of P in the dictionary $\mathcal{D}$.

THEOREM 9.10 *A prefix search in the String B-Tree built over the dictionary $\mathcal{D}$ takes $O(\frac{p}{B} \log_B n + \frac{N_{occ}}{B})$ I/Os, where N_{occ} is the total length of the dictionary strings which are prefixed by P. The data structure occupies $O(\frac{N}{B})$ disk pages and, indeed, strings are stored uncompressed on disk.*

This result is good but not yet *optimal*. The issue that we have to resolve to reach optimality is *pattern rescanning*: each time we do a Blind search, we compare Q and one of the strings stored in the currently visited B-Tree node starting from their first character. However, as we go down in the string B-tree we can capitalize on the characters of Q that we have already compared in the upper levels of the B-tree, and thus avoid the rescanning of these characters during the subsequent lcp-computations. So if f characters have been already matched in Q during some previous lcp-computation, the next lcp-computation can compare Q with a dictionary string starting from their $(f + 1)$-th character. The pro of this approach is that I/Os turn to be optimal, the cons is that strings have to be stored uncompressed in order to support the efficient access to that $(f + 1)$-th character. Working out all the details [4], one can show that:

THEOREM 9.11 *A prefix search in the String B-Tree built over the dictionary $\mathcal{D}$ takes $O(\frac{p+N_{occ}}{B} + \log_B n)$ optimal I/Os, where N_{occ} is the total length of the dictionary strings which are prefixed by P. The data structure occupies $O(\frac{N}{B})$ disk pages and, indeed, strings are stored uncompressed on disk.*

If we want to store the strings compressed on disk, we cannot just plug LPFC in the approach illustrated above, because the decoding of LPFC works only on full strings, and thus it does not support the efficient skip of some characters without wholly decoding the compared string. [2] discusses a sophisticated solution to this problem which gets the I/O-bounds in Theorem 9.11 but in the cache-oblivious model and guaranteeing LPFC-compressed space. We refer the interested reader to that paper for details.

9.6.2 Packing Trees on Disk

We point out that the advantage of finding a good layout for unbalanced trees among disk pages (of size B) may be unexpectedly large, and therefore, must not be underestimated when designing solutions that have to manage large trees on disk. In fact, while balanced trees save a factor $O(\log B)$ when mapped to disk (pack B-node balanced subtrees per page), the mapping of unbalanced trees grows with non uniformity and approaches, in the extreme case of a linear-height tree, a saving factor of $\Theta(B)$ over a naïve memory layout.

This problem is also known in the literature as the *Tree Packing* problem. Its goal is to find an allocation of tree nodes among the disk pages in such a way that the number of I/Os executed for a pattern search is minimized. Minimization may involve either the total number of loaded pages in internal memory (i.e. page faults) , or the number of distinct visited pages (i.e. working-set size). This way we model two extreme situations: the case of a one-page internal memory (i.e. a small buffer), or the case of an unbounded internal memory (i.e. an unbounded buffer). Surprisingly, the optimal solution to the tree packing problem is *independent* of the available buffer size because no disk page is visited twice when page faults are minimized or the working set is minimum. Moreover, the optimal solution shows a nice *decomposability property*: the optimal tree packing forms in turn a tree of disk pages. These two facts allow to restrict our attention to the page-fault minimization problem, and to the design of recursive approaches to the optimal tree decomposition among the disk pages.

In the rest of this section we present two solutions of increasing sophistication and addressing two different scenarios: one in which the goal is to *minimize the maximum number* of page faults executed during a downward root-to-leaf traversal; the other in which the goal is to *minimize the average number* of page faults by assuming an access distribution to the tree leaves, and thus to the possible tree traversals. We briefly mention that both solutions assume that B is known; the literature actually offers cache-oblivious solutions to the tree packing problem, but they are too much sophisticated to be reported in these notes. For details we refer the reader to [1, 5].

Min-Max Algorithm. This solution operates greedily and bottom up over the tree to be packed with the goal of minimizing the maximum number of page faults executed during a downward traversal which starts from the root of the tree. The tree is assumed to be binary, this is not a restriction for Patricia Tries because it is enough to encode the alphabet characters with binary strings. The algorithm assigns every leaf to its own disk page and the height of this page is set to 1. Working upward, Algorithm 9.1 is applied to each processed node until the root of the tree is reached.

Algorithm 9.1 Min-Max Algorithm over binary trees (general step).

```
Let $u$ be the currently visited node;
if If both children of $u$ have the same page height $d$ then
    if If the total number of nodes in both children's pages is $< B$ then
        Merge the two disk pages and add $u$;
        Set the height of this new page to $d$;
    else
        Close off the pages of $u$'s children;
        Create a new page for $u$ and set its height to $d + 1$;
    end if
else
    Close off the page of $u$'s child with the smaller height;
    If possible, merge the page of the other child with $u$ and leave its height unchanged;
    Otherwise, create a new page for $u$ with height $d + 1$ and close off the child's page;
end if
```

The final packing may induce a poor page-fill ratio, nonetheless several changes can alleviate this problem in real situations:

1. When a page is closed off, scan its children pages from the smallest to the largest and check whether they can be merged with their parent.
2. Design logical disk pages and pack many of them into one physical disk page; possibly ignore physical page boundaries when placing logical pages onto disk.

THEOREM 9.12 *The Min-Max Algorithm provides a disk-packing of a tree of n nodes and height H such that every root-to-leaf path traverses less than* $1 + \lceil \frac{H}{\sqrt{B}} \rceil + \lceil 2log_B n \rceil$ *pages.*

Distribution-aware Packing. We assume that it is known an access distribution to the Patricia trie leaves. Since this distribution is often skewed towards some leaves, that are then accessed more frequently than others, the Min-Max algorithm may be significantly inefficient. The following algorithm is based on a Dynamic-Programming scheme, and optimizes the *expected* number of I/Os incurred by any traversal of a root-to-leaf path.

We denote by τ this optimal tree packing (from tree nodes to disk pages), so $\tau(u)$ denotes the disk page to which the tree node u is mapped. Let $w(f)$ be the probability to access a leaf f, we derive a distribution over all other nodes u of the tree by summing up the access probabilities of its descending leaves. We can assume that the tree root r is always mapped to a fixed page $\tau(r) = R$. Consider now the set V of tree nodes that descend from R's nodes but are not themselves in R. We observe that the optimal packing τ induces a tree of disk pages and consequently, if τ is optimal for the current tree T, then τ is optimal for all subtrees T_v rooted in $v \in V$.

This result allows to state a recursive computation for τ that first determines which nodes reside in R, and then continues recursively with all subtrees T_v for which $v \in V$. Dynamic programming provides an efficient implementation of this idea, based on the following definition: An *i-confined* packing of a tree T is a packing in which the page R contains exactly i nodes (clearly $i \leq B$). Now, in the optimal packing τ, the root page R will contain i^* nodes from the left subtree $T_{left(r)}$ and $(B - i^* - 1)$ nodes from the right subtree $T_{right(r)}$, for some i^*. The consequence is that τ is both an optimal i^*-confined packing for $T_{left(r)}$ and an optimal $(B - i^* - 1)$-confined packing for $T_{right(r)}$. This property is at the basis of the Dynamic-Programming rule which computes $A[v, i]$, for a generic node v and integer $i \leq B$, as the cost of an optimal i-confined packing of the subtree T_v. In the paper [5] the authors showed that $A[v, i]$, for $i > 1$, can be computed as the access probability $w(v)$ plus the minimum among the following three quantities:

1. $A[left(v), i - 1] + w(right(v)) + A[right(v), B]$
2. $w(left(v)) + A[left(v), B] + A[right(v), i - 1]$
3. $\min_{1 \leq j < i-1}\{A[left(v), j] + A[right(v), i - j - 1]\}$

Rule (1) accounts for the (unbalanced) case in which the i-confined packing is obtained by storing $i - 1$ nodes from $T_{left(v)}$ into the v's page; Rule (2) is the symmetric of Rule (1); whereas Rule (3) accounts for the case in which j nodes from $T_{left(v)}$ and $i - j - 1$ nodes from $T_{right(v)}$ are stored into the page of v to form the optimal i-confined packing of T_v. The special case $i = 1$ is given by $A[v, 1] = w(T_v) + A[left(v), B] + A[right(v), B]$.

Algorithm 9.2 deploys these rules to compute the optimal tree packing in $O(nB^2)$ time and $O(nB)$ space.

Algorithm 9.2 Distribution-aware packing of trees on disk.

```
Initialize $A[v, i] = w(v)$, for all leaves $v$ and integers $i \leq B$;
while there exist an unmarked node $v$ do
    mark $v$;
    update $A[v, 1] = w(v) + A[left(v), B] + A[right(v), B]$;
    for $i = 2$ to $B$ do
        update $A[v, i]$ according to the dyn-prog rule specified in the text.
    end for
end while
```

THEOREM 9.13 *An optimal packing for a f-ary tree of n nodes can be computed in $O(nB^2 \log f)$ time and $O(B \log n)$ space. The packing maps the tree into at most $2\lfloor \frac{n}{B} \rfloor$ disk pages. Optimality is with respect to the* expected *number of I/Os incurred by any root-to-leaf traversal.*

References

[1] Stefan Alstrup, Michael A. Bender, Erik D. Demaine, Martin Farach-Colton, Jan I. Munro, Theis Rauhe, and M. Thorup. *Efficient Tree Layout in a Multilevel Memory Hierarchy*, 2003. Personal Communication, corrected version of a paper appeared in the *European Symposium on Algorithms 2002*.

[2] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string B-trees. In *Procs ACM Symposium on Principles of Database Systems*, pages 223–242, 2006.

[3] Erik D. Demaine, Thouis Jones, and Mihai Pǎtraşcu. Interpolation search for non-independent data. In *Procs ACM-SIAM Symposium on Discrete algorithms*, pages 529–530, 2004.

[4] Paolo Ferragina and Roberto Grossi. The String B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[5] Joseph Gil and Alon Itai. How to pack trees. *Journal of Algorithms*, 32(2):108–132, 1999.

[6] Michael Luby, Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17, 373–386, 1988.

10

Searching Strings by Substring

In this lecture we will be interested in solving the following problem, known as *full-text searching* or *substring searching*.

> **The substring-search problem.** *Given a text string $T[1, n]$, drawn from an alphabet of size σ, retrieve (or just count) all text positions where a query pattern $P[1, p]$ occurs as a substring of T.*

It is evident that this problem can be solved by brute-forcedly comparing P against every substring of T, thus taking $O(np)$ time in the worst case. But it is equivalently evident that this *scan*-based approach is unacceptably slow when applied to massive text collections subject to a massive number of queries, which is the scenario involving genomic databases or search engines. This suggests the usage of a so called *indexing* data structure which is built over T before that searches start. A setup cost is required for this construction, but this cost is amortized over the subsequent pattern searches, thus resulting convenient in a quasi-static environment in which T is changed very rarely.

In this lecture we will describe two main approaches to substring searching, one based on arrays and another one based on trees, that mimic what we have done for the prefix-search problem. The two approaches hinge on the use of two fundamental data structures: the *suffix array* (shortly *SA*) and the *suffix tree* (shortly *ST*). We will describe in much detail those data structures because their use goes far beyond the context of full-text search.

10.1 Notation and terminology

We assume that text T ends with a special character $T[n] = \$$, which is smaller than any other alphabet character. This ensures that text suffixes are prefix-free and thus no one is a prefix of another suffix. We use $suff_i$ to denote the i-th suffix of text T, namely the substring $T[i, n]$. The following observation is crucial:

> If $P = T[i, i + p - 1]$, then the pattern occurs at text position i and thus we can state that P is a prefix of the i-th text suffix, namely P is a prefix of the string $suff_i$.

As an example, if P ="siss" and T ="mississippi\$", then P occurs at text position 4 and indeed it prefixes the suffix $suff_4 = T[4, 12]$ ="sissippi\$". For simplicity of exposition, and for historical reasons, we will use this text as running example; nevertheless we point out that a text may be an arbitrary sequence of characters, hence not necessarily a single word.

Given the above observation, we can form with all text suffixes the dictionary $SUF(T)$ and state that *searching for P as a substring of T boils down to searching for P as a prefix of some string in* $SUF(T)$. In addition, since there is a bijective correspondence among the text suffixes prefixed by P and the pattern occurrences in T, then

1. the suffixes prefixed by P occur contiguously into the lexicographically sorted $SUF(T)$,
2. the lexicographic position of P in $SUF(T)$ immediately precedes the block of suffixes prefixed by P.

An attentive reader may have noticed that these are the properties we deployed to efficiently support prefix searches. And indeed the solutions known in the literature for efficiently solving the substring-search problem hinge either on array-based data structures (i.e. the Suffix Array) or on trie-based data structures (i.e. the Suffix Tree). So the use of these data structures in pattern searching is pretty immediate. What is challenging is the efficient construction of these data structures and their mapping onto disk to achieve efficient I/O-performance. These will be the main issues dealt with in this lecture.

Text suffixes	**Indexes**
`mississippi$`	1
`ississippi$`	2
`ssissippi$`	3
`sissippi$`	4
`issippi$`	5
`ssippi$`	6
`sippi$`	7
`ippi$`	8
`ppi$`	9
`pi$`	10
`i$`	11
`$`	12

Sorted Suffixes	*SA*	***Lcp***
`$`	12	0
`i$`	11	1
`ippi$`	8	1
`issippi$`	5	4
`ississippi$`	2	0
`mississippi$`	1	0
`pi$`	10	1
`ppi$`	9	0
`sippi$`	7	2
`sissippi$`	4	1
`ssippi$`	6	3
`ssissippi$`	3	-

FIGURE 10.1: *SA* and `lcp` array for the string T ="mississippi\$".

10.2 The Suffix Array

The suffix array for a text T is the array of pointers to all text suffixes ordered lexicographically. We use the notation $SA(T)$ to denote the suffix array built over T, or just SA if the indexed text is clear from the context. Because of the lexicographic ordering, $SA[i]$ is the i-th smallest text suffix, so we have that $suff_{SA[1]} < suff_{SA[2]} < \cdots < suff_{SA[n]}$, where $<$ is the lexicographical order between strings. For space reasons, each suffix is represented by its starting position in T (i.e. an integer). SA consists of n integers in the range $[1, n]$ and hence it occupies $O(n \log n)$ bits.

Another useful concept is the *longest common prefix* between two consecutive suffixes $suff_{SA[i]}$ and $suff_{SA[i+1]}$. We use `lcp` to denote the array of integers representing the lengths of those lcps. Array `lcp` consists of $n - 1$ entries containing values smaller than n. There is an optimal and non obvious linear-time algorithm to build the *lcp*-array which will be detailed in Section 10.2.3. The interest in `lcp` rests in its usefulness to design efficient/optimal algorithms to solve various search and mining problems over strings.

10.2.1 The substring-search problem

We observed that this problem can be reduced to a prefix search over the string dictionary $SUF(T)$, so it can be solved by means of a binary search for P over the array of text suffixes ordered lexicographically, hence $SA(T)$. Figure 10.1 shows the pseudo-code which coincides with the classic binary-search algorithm specialized to compare strings rather than numbers.

Algorithm 10.1 SubstringSearch($P, SA(T)$)

```
$L = 1, R = n$;
while ($L \neq R$) do
    $M = \lfloor (L + R)/2 \rfloor$;
    if (strncmp($P, suff_M, p$) $> 0$) then
        $L = M + 1$;
    else
        $R = M$;
    end if
end while
if (strncmp($P, suff_L, p$) $= 0$) then
    return $L$;
else
    return $-1$;
end if
```

A binary search in SA requires $O(\log n)$ string comparisons, each taking $O(p)$ time in the worst case.

LEMMA 10.1 Given the text $T[1, n]$ and its suffix array, we can count the occurrences of a pattern $P[1, p]$ in the text taking $O(p \log n)$ time and $O(\log n)$ memory accesses in the worst case. Retrieving the positions of these *occ* occurrences takes additional $O(occ)$ time. The total required space is $n(\log n + \log \sigma)$ bits, where the first term accounts for the suffix array and the second term for the text.

Figure 10.2 shows a running example, which highlights an interesting property: the comparison between P and $suff_M$ does not need to start from their initial character. In fact one could exploit the lexicographic sorting of the suffixes and skip the characters comparisons that have already been carried out in previous iterations. This can be done with the help of three arrays:

- the `lcp`$[1, n - 1]$ array;
- two other arrays $Llcp[1, n - 1]$ and $Rlcp[1, n - 1]$ which are defined for every triple (L, M, R) that may arise in the inner loop of a binary search. We define $Llcp[M] =$

$\texttt{lcp}(\mathit{suff}_{SA[L]}, \mathit{suff}_{SA[M]})$ and $Rlcp[M] = \texttt{lcp}(\mathit{suff}_{SA[M]}, \mathit{suff}_{SA[R]})$, namely $Llcp[M]$ accounts for the prefix shared by the leftmost suffix $\mathit{suff}_{SA[L]}$ and the middle suffix $\mathit{suff}_{SA[M]}$ of the range currently explored by the binary search; $Rlcp[M]$ accounts for the prefix shared by the rightmost suffix $\mathit{suff}_{SA[R]}$ and the middle suffix $\mathit{suff}_{SA[M]}$ of that range.

```
⟹    $                 $                 $                 $
|     i$                i$                i$                i$
|     ippi$             ippi$             ippi$             ippi$
|     issippi$          issippi$          issippi$          issippi$
|     ississippi$       ississippi$       ississippi$       ississippi$
|→    mississippi$      mississippi$      mississippi$      mississippi$
|     pi$          ⟹   pi$               pi$               pi$
|     ppi$         |    ppi$              ppi$              ppi$
|     sippi$       |    sippi$            sippi$            sippi$
|     sissippi$    |→   sissippi$         sissippi$         sissippi$
|     ssippi$      |    ssippi$      ⟹   ssippi$      ⟹   ssippi$
⟹    ssissippi$   ⟹   ssissippi$   ⟹   ssissippi$        ssissippi$
      Step (1)          Step (2)          Step (3)          Step (4)
```

FIGURE 10.2: Binary search steps for the lexicographic position of the pattern P ="ssi" in "mississippi$".

We notice that each triple (L, M, R) is uniquely identified by its midpoint M because the execution of a binary search defines actually a hierarchical partition of the array SA into smaller and smaller sub-arrays delimited by (L, R) and thus centered in M. Hence we have $O(n)$ triples overall, and these three arrays occupy $O(n)$ space in total.

We can build arrays $Llcp$ and $Rlcp$ in linear time by exploiting two different approaches. We can deploy the observation that the $\texttt{lcp}[i, j]$ between the two suffixes $\mathit{suff}_{SA[i]}$ and $\mathit{suff}_{SA[j]}$ can be computed as the minimum of a range of $\texttt{lcp}$-values, namely $\texttt{lcp}[i, j] = \min_{k=i,\ldots,j-1} \texttt{lcp}[k]$. By associativity of the min we can split the computation as $\texttt{lcp}[i, j] = \min\{\texttt{lcp}[i, k], \texttt{lcp}[k, j]\}$ where k is any index in the range $[i, j]$, so in particular we can set $\texttt{lcp}[L, R] = \min\{\texttt{lcp}[L, M], \texttt{lcp}[M, R]\}$. This implies that the arrays $Llcp$ and $Rlcp$ can be computed via a bottom-up traversal of the triplets (L, M, R) in $O(n)$ time. Another way to deploy the previous observation is to compute $\texttt{lcp}[i, j]$ on-the-fly via a Range-Minimum Data structure built over the array $\texttt{lcp}$ (see Section 10.4.1). All of these approaches take $O(n)$ time and space, and thus they are optimal.

We are left with showing how the binary search can be speeded up by using these arrays. Consider a binary-search iteration on the sub-array $SA[L, R]$, and let M be the midpoint of this range (hence $M = (L + R)/2$). A lexicographic comparison between P and $\mathit{suff}_{SA[M]}$ has to be made in order to choose the next search-range between $SA[L, M]$ and $SA[M, R]$. The goal is to compare P and $\mathit{suff}_{SA[M]}$ without starting necessarily from their first character, but taking advantage of the previous binary-search steps in order to infer, hopefully in constant time, their lexicographic comparison.

Surprisingly enough this is possible and requires to know, in addition to $Llcp$ and $Rlcp$, the values $l = \texttt{lcp}(P, \mathit{suff}_{SA[L]})$ and $r = \texttt{lcp}(P, \mathit{suff}_{SA[R]})$ which denote the number of characters the pattern P shares with the strings at the extremes of the range currently explored by the binary search. At the first step, in which $L = 1$ and $R = n$, these two values can be computed in $O(p)$ time by comparing character-by-character the involved strings. At a generic step, we assume that l and r are known inductively, and show below how the binary-search step can preserve their knowledge after that we move onto $SA[L, M]$ or $SA[M, R]$.

So let us detail the implementation of a generic binary-search step. We know that P lies between $\mathit{suff}_{SA[L]}$ and $\mathit{suff}_{SA[R]}$, so P surely shares $\texttt{lcp}[L, R]$ characters with these suffixes given that any string (and specifically, all suffixes) in this range must share this number of characters (given that they are lexicographically sorted). Therefore the two values l and r are larger (or equal) than k, as well as it is larger (or equal) to k the number of characters m that the pattern P shares with $\mathit{suff}_{SA[M]}$. We could then take advantage of this last inequality to compare P with $\mathit{suff}_{SA[M]}$ starting from their $(k + 1)$-th character. But actually we can do better because we know r and l, and these values can be significantly larger than k, thus more characters of P have been already involved in previous comparisons and so they are known.

We distinguish two main cases. If $l = r$ then all suffixes in the range $[L, R]$ share (at least) l characters (hence $\mathit{suff}_{SA[M]}$ too) which are equal to $P[1, l]$. So the comparison between P and $\mathit{suff}_{SA[M]}$ can start from their $(l+1)$-th character, which means that we are advancing in the scanning of P. Otherwise (i.e. $l \neq r$), a more complicated test has to be done. Consider the case where $l > r$ (the other being symmetric). Here P shares more characters (i.e. $l > r$) with $\mathit{suff}_{SA[L]}$ than with $\mathit{suff}_{SA[R]}$; moreover, it is surely $r = k$ because $P[1, r]$ is shared by both $\mathit{suff}_{SA[L]}$ and $\mathit{suff}_{SA[R]}$ and k is the longest shared prefix by these two suffixes, by definition.

So, given that $l > r$ and we do not want to re-scan characters of P that have been already seen (namely characters in $P[1, l]$), we define our algorithm in such a way that the order between P and $\mathit{suff}_{SA[M]}$ can be inferred either comparing characters in $P[l + 1, n]$, or comparing the values l and $\mathit{Llcp}[M]$ (which give us information about $P[1, l]$). We can thus distinguish three cases:

- If $l < \mathit{Llcp}[M]$, then P is greater that $\mathit{suff}_{SA[M]}$ and we can set $m = l$. In fact, by induction, $P > \mathit{suff}_{SA[L]}$ and their mismatch character lies at position $l+1$. By definition of $\mathit{Llcp}[M]$ and the hypothesis, we have that $\mathit{suff}_{SA[L]}$ shares more than l characters with $\mathit{suff}_{SA[M]}$. So the mismatch between P and $\mathit{suff}_{SA[M]}$ is the same as it is between P and $\mathit{suff}_{SA[L]}$, hence their comparison gives the same answer— i.e. $P > \mathit{suff}_{SA[M]}$— and the search can thus continue in the subrange $SA[M, R]$. We remark that this case does not induce any character comparison.
- If $l > \mathit{Llcp}[M]$, this case is similar as the previous one. We can conclude that P is smaller than $\mathit{suff}_{SA[M]}$ and it is $m = \mathit{Llcp}[M]$. So the search continues in the subrange $SA[L, M]$, without additional character comparisons.
- If $l = \mathit{Llcp}[M]$, then P shares l characters with $\mathit{suff}_{SA[L]}$ and $\mathit{suff}_{SA[M]}$. So the comparison between P and $\mathit{suff}_{SA[M]}$ can start from their $(l+1)$-th character. Eventually we determine m and their lexicographic order. Here some character comparisons are executed, but the *knowledge* about P's characters advanced too.

It is clear that every binary-search step either advances the comparison of P's characters, or it does not compare any character but halves the range $[L, R]$. The first case can occur at most p times, the second case can occur $O(\log n)$ times. We have therefore proved the following.

LEMMA 10.2 Given the three arrays $\texttt{lcp}$, Llcp and Rlcp built over a text $T[1, n]$, we can count the occurrences of a pattern $P[1, p]$ in the text taking $O(p + \log n)$ time in the worst case. Retrieving the positions of these occ occurrences takes additional $O(\mathit{occ})$ time. The total required space is $O(n)$.

Proof We remind that searching for all strings having the pattern P as a prefix requires two lexicographic searches: one for P and the other for $P\#$, where # is a special character larger than any other alphabet character. So $O(p + \log n)$ character comparisons are enough to delimit the range $SA[i, j]$ of suffixes having P as a prefix. It is then easy to count the pattern occurrences in constant time, as $occ = j - i + 1$, or print all of them in $O(occ)$ time. ■

10.2.2 The LCP-array and its construction$^{\infty}$

Surprisingly enough the longest common prefix array $\texttt{lcp}[1, n-1]$ can be derived from the input string T and its suffix array $SA[1, n]$ in optimal linear time.[1] This time bound cannot be obtained by the simple approach that compares character-by-character the $n - 1$ contiguous pairs of text suffixes in SA; as this takes $\Theta(n^2)$ time in the worst case. The optimal $O(n)$ time needs to avoid the re-scanning of the text characters, so some property of the input text has to be proved and deployed in the design of an algorithm that achieves this complexity. This is exactly what Kasai *et al* did in 2001 [8], their algorithm is elegant, deceptively simple, and optimal in time and space.

Sorted Suffixes	SA	SA `positions`
abc`def`...	$j-1$	$p-1$
abc`hi`...	$i-1$	p
.	.	.
.	.	.
.	.	.
bc`def`...	j	
.	.	.
.	.	.
.	.	.
bch...	k	$q-1$
bch`i`...	i	q

FIGURE 10.3: Relation between suffixes and `lcp` values in the Kasai's algorithm. Suffixes are shown only with their starting characters, the rest is indicated with ... for simplicity.

For the sake of presentation we will refer to Figure 10.3 which illustrates clearly the main algorithmic idea. Let us concentrate on two consecutive suffixes in the text T, say $suff_{i-1}$ and $suff_i$, which occur at positions p and q in the suffix array SA. And assume that we know inductively the value of $\texttt{lcp}[p-1]$, storing the longest common prefix between $SA[p-1] = suff_{j-1}$ and the next suffix $SA[p] = suff_{i-1}$ in the lexicographic order. Our goal is to show that $\texttt{lcp}[q-1]$ storing the longest common prefix between suffix $SA[q-1] = suff_k$ and the next ordered suffix $SA[q] = suff_i$, which interests us, can be computed without re-scanning these suffixes from their first character but can start where the comparison between $SA[p-1]$ and $SA[p]$ ended. This will ensure that re-scanning of text characters is avoided, precisely it is avoided the re-scanning of $suff_{i-1}$, and as a result we will get a linear time complexity.

We need the following property that we already mentioned when dealing with prefix search, and that we restate here in the context of suffix arrays.

[1] Recall that $\texttt{lcp}[i] = \texttt{lcp}(suff_{SA[i]}, suff_{SA[i+1]})$ for $i < n$.

FACT 10.1 *For any position $x < y$ it holds* $\texttt{lcp}(suff_{SA[y-1]}, suff_{SA[y]}) \geq \texttt{lcp}(suff_{SA[x]}, suff_{SA[y]})$.

Proof This property derives from the observation that suffixes in SA are ordered lexicographically, so that, as we go farther from $SA[y]$ we reduce the length of the shared prefix. ■

Let us now refer to Figure 10.3, concentrate on the pair of suffixes $suff_{j-1}$ and $suff_{i-1}$, and take their next suffixes $suff_j$ and $suff_i$ in T. There are two possible cases: Either they share some characters in their prefix, i.e. $\texttt{lcp}[p-1] > 0$, or they do not. In the former case we can conclude that, since lexicographically $suff_{j-1} < suff_{i-1}$, the next suffixes preserve that lexicographic order, so $suff_j < suff_i$ and moreover $\texttt{lcp}(suff_j, suff_i) = \texttt{lcp}[p-1] - 1$. In fact, the first shared character is dropped, given the step ahead from $j-1$ (resp. $i-1$) to j (resp. i) in the starting positions of the suffixes, but the next $\texttt{lcp}[p-1] - 1$ shared characters (possibly none) remain, as well as remain their mismatch characters that drives the lexicographic order. In the Figure above, we have $\texttt{lcp}[p-1] = 3$ and the shared prefix is abc, so when we consider the next suffixes their lcp is bc of length 2, their order is preserved (as indeed $suff_j$ occurs before $suff_i$), and now they lie not adjacent in SA.

FACT 10.2 *If* $\texttt{lcp}(suff_{SA[y-1]}, suff_{SA[y]}) > 0$ *then:*

$$\texttt{lcp}(suff_{SA[y-1]+1}, suff_{SA[y]+1}) = \texttt{lcp}(suff_{SA[y-1]}, suff_{SA[y]}) - 1$$

By Fact 10.1 and Fact 10.2, we can conclude the key property deployed by Kasai's algorithm: $\texttt{lcp}[q-1] \geq \max\{\texttt{lcp}[p-1]-1, 0\}$. This algorithmically shows that the computation of $\texttt{lcp}[q-1]$ can take full advantage of what we compared for the computation of $\texttt{lcp}[p-1]$. By adding to this the fact that we are processing the text suffixes rightward, we can conclude that the characters involved in the suffix comparisons move themselves rightward and, since re-scanning is avoided, their total number is $O(n)$. A sketch of the Kasai's algorithm is shown in Figure 10.2, where we make use of the inverse suffix array, denoted by SA^{-1}, which returns for every suffix its position in SA. Referring to Figure 10.3, we have that $SA^{-1}[i] = p$.

Algorithm 10.2 LCP-Build(char *T, int n, char **SA)

```
 1: $h = 0$;
 2: for ($i = 1$; $i \leq n$, $i$++) do
 3:     $q = SA^{-1}[i]$;
 4:     if ($q > 1$) then
 5:         $k = SA[q-1]$;
 6:         if ($h > 0$) then
 7:             $h--$;
 8:         end if
 9:         while ($T[k+h] == T[i+h]$) do
10:             $h$++;
11:         end while
12:         lcp[$q-1$] = $h$;
13:     end if
14: end for
```

Step 4 checks whether $suff_q$ occupies the first position of the suffix array, in which case the lcp with the previous suffix is undefined. The **for**-loop then scans the text suffixes $suff_i$ from left to right, and for each of them it first retrieves the position of $suff_i$ in SA, namely $i = SA[q]$, and its preceding suffix in SA, namely $k = SA[q-1]$. Then it extends their longest common prefix starting from the offset h determined for $suff_{i-1}$ via character-by-character comparison. This is the algorithmic application of the above observations.

As far as the time complexity is concerned, we notice that h is decreased at most n times (once per iteration of the for-loop), and it cannot move outside T (within each iteration of the for-loop), so $h \leq n$. This implies that h can be increased at most $2n$ times and this is the upper bound to the number of character comparisons executed by the Kasai's algorithm. The total time complexity is therefore $O(n)$.

We conclude this section by noticing that an I/O-efficient algorithm to compute the lcp-array is still missing in the literature, some heuristics are known to reduce the number of I/Os incurred by the above computation but an optimal $O(n/B)$ I/O-bound is yet to come, if possible.

10.2.3 Suffix-array construction

Given that the suffix array is a sorted sequence of items, the most intuitive way to construct SA is to use an efficient comparison-based sorting algorithm and specialize the comparison-function in such a way that it computes the lexicographic order between strings. Algorithm 10.3 implements this idea in C-style using the built-in procedure QSORT as sorter and a properly-defined subroutine Suffix_cmp for comparing suffixes:

```
Suffix_cmp(char **p, char **q){ return strcmp(*p, *q) };
```

Notice that the suffix array is initialized with the pointers to the real starting positions in memory of the suffixes to be sorted, and not the integer offsets from 1 to n as stated in the formal description of SA of the previous pages. The reason is that in this way Suffix_cmp does not need to know T's position in memory (which would have needed a global parameter) because its actual parameters passed during an invocation provide the starting positions in memory of the suffixes to be compared. Moreover, the suffix array SA has indexes starting from 0 as it is typical of C-language.

Algorithm 10.3 COMPARISON_BASED_CONSTRUCTION(char *T, int n, char **SA)

```
for ($i = 0; i < n; i$ ++) do
    $SA[i] = T + i$;
end for
QSORT($SA$, $n$, sizeof(char *), Suffix_cmp);
```

A major drawback of this simple approach is that it is not I/O-efficient for two main reasons: the optimal number $O(n \log n)$ of comparisons involves now variable-length strings which may consists of up to $\Theta(n)$ characters; locality in SA does not translate into locality in suffix comparisons because of the fact that sorting permutes the string pointers rather than their pointed strings. Both these issues elicit I/Os, and turn this simple algorithm into a slow one.

THEOREM 10.1 *In the worst case the use of a comparison-based sorter to construct the suffix array of a given string $T[1,n]$ requires $O((\frac{n}{B})n \log n)$ I/Os, and $O(n \log n)$ bits of working space.*

In Section 10.2.3 we describe a Divide-and-Conquer algorithm— the *Skew* algorithm proposed by Kärkkäinen and Sanders [7]— which is elegant, easy to code, and flexible enough to achieve the optimal I/O-bound in various models of computations. In Section 10.2.3 we describe another algorithm— the *Scan-based* algorithm proposed by BaezaYates, Gonnet and Sniders [6]— which is also simple, but incurs in a larger number of I/Os; we nonetheless introduce this algorithm because it offers the positive feature of processing the input data in passes (streaming-like) thus forces prefetching, allows compression and hence it turns to be suitable for slow disks.

The Skew Algorithm

In 2003 Kärkkäinen and Sanders [7] showed that the problem of constructing suffix-arrays can be *reduced* to the problem of sorting a set of triplets whose components are integers in the range $[1, O(n)]$. Surprisingly this reduction takes *linear time and space* thus turning the complexity of suffix-array construction into the complexity of sorting atomic items, a problem about which we discussed deeply in the previous chapters and for which we know optimal algorithms for hierarchical memories and multiple disks. More than this, since the items to be sorted are integers bounded in value by $O(n)$, the sorting of the triplets takes $O(n)$ time in the RAM model, so this is the optimal time complexity of suffix-array construction in RAM. Really impressive!

This algorithm is named *Skew* in the literature, and it works in every model of computation for which an efficient sorting primitive is available: disk, distributed, parallel. The algorithm hinges on a divide&conquer approach that executes a $\frac{2}{3} : \frac{1}{3}$ split, crucial to make the final merge-step easy to implement. Previous approaches used the more natural $\frac{1}{2} : \frac{1}{2}$ split (such as [2]) but were forced to use a more sophisticated merge-step which needed the use of the suffix-tree data structure.

For the sake of presentation we use $T[1, n] = t_1 t_2 \ldots t_n$ to denote the input string and we assume that the characters are drawn from an integer alphabet of size $\sigma = O(n)$. Otherwise we can sort the characters of T and rename them with integers in $O(n)$, taking overall $O(n \log \sigma)$ time in the worst-case. So T is a text of integers, taking $\Theta(\log n)$ bits each; this will be the case for all texts created during the suffix-array construction process. Furthermore we assume that $t_n = \$$, a special symbol smaller than any other alphabet character, and logically pad T with an infinite number of occurrences of \$.

Given this notation, we can sketch the three main steps of the Skew algorithm:

Step 1. Construct the suffix array $SA^{2,0}$ limited to the suffixes starting at positions $P_{2,0} = \{i : i \bmod 3 = 2, \text{ or } i \bmod 3 = 0\}$:

- Build a special string $T^{2,0}$ of length $(2/3)n$ which compactly encodes all suffixes of T starting at positions $P_{2,0}$.
- Build recursively the suffix-array SA' of $T^{2,0}$.
- Derive the suffix-array $SA^{2,0}$ from SA'.

Step 2 Construct the suffix array SA^1 of the remaining suffixes starting at positions $P_1 = \{i : i \bmod 3 = 1\}$:

- For every $i \in P_1$, represent suffix $T[i, n]$ with a pair $\langle T[i], \mathtt{pos}(i + 1)\rangle$, where it is $i + 1 \in P_{2,0}$.
- Assume to have pre-computed the array $\mathtt{pos}[i + 1]$ which provides the position of the $(i + 1)$-th text suffix $T[i + 1, n]$ in $SA^{2,0}$.
- Radix-sort the above $O(n)$ pairs.

Step 3. Merge the two suffix arrays into one:

- This is done by deploying the decomposition $\frac{2}{3} : \frac{1}{3}$ which ensures a constant-time lexicographic comparison between any pair of suffixes (see details below).

The execution of the algorithm is illustrated over the input string $T[1, 12]$ ="mississippi\$" whose suffix array is $SA = (12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3)$. In this example we have: $P_{2,0} = \{2, 3, 5, 6, 8, 9, 11, 12\}$ and $P_1 = \{1, 4, 7, 10\}$.

Step 1. The first step is the most involved one and constitutes the backbone of the entire recursive process. It lexicographically sorts the suffixes starting at the text positions $P_{2,0}$. The resulting array is denoted by $SA^{2,0}$ and represents a *sampled* version of the final suffix array SA because it is restricted to the suffixes starting at positions $P_{2,0}$.

To efficiently obtain $SA^{2,0}$, we reduce the problem to the construction of the suffix array for a string $T^{2,0}$ of length about $\frac{2n}{3}$. This text consists of "characters" which are integers smaller than $\approx \frac{2n}{3}$. Since we are *again* in the presence of a text of integers, of length proportionally smaller than n, we can construct its suffix array by invoking *recursively* the construction procedure.

The key difficulty is how to define $T^{2,0}$ so that its suffix array may be used to derive easily $SA^{2,0}$, namely the sorted sequence of text suffixes starting at positions in $P_{2,0}$. The elegant solution consists of considering the two text suffixes $T[2, n]$ and $T[3, n]$, pad them with the special symbol \$ in order to have multiple-of-three length, and then decompose the resulting strings into triplets of characters $T[2, \cdot] = [t_2, t_3, t_4][t_5, t_6, t_7][t_8, t_9, t_{10}] \dots$ and $T[3, \cdot] = [t_3, t_4, t_5][t_6, t_7, t_8][t_9, t_{10}, t_{11}] \dots$. The dot expresses the fact that we are considering the smallest integer, larger than n, that allows those strings to have length which is a multiple of three.
With reference to the previous example, we have:

$$T[2,\cdot] = [\mathtt{i}\,\underset{2}{\mathtt{s}}\,\mathtt{s}]\,[\mathtt{i}\,\underset{5}{\mathtt{s}}\,\mathtt{s}]\,[\mathtt{i}\,\underset{8}{\mathtt{p}}\,\mathtt{p}]\,[\mathtt{i}\,\underset{11}{\$}\,\$] \qquad T[3,\cdot] = [\,\underset{3}{\mathtt{s\,s\,i}}\,]\,[\,\underset{6}{\mathtt{s\,s\,i}}\,]\,[\,\underset{9}{\mathtt{p\,p\,i}}\,]\,[\,\underset{12}{\$\,\$\,\$}\,]$$

We then construct the string $R = T[2, \cdot] \bullet T[3, \cdot]$, and thus we obtain:

$$R = [\mathtt{i}\,\underset{2}{\mathtt{s}}\,\mathtt{s}]\,[\mathtt{i}\,\underset{5}{\mathtt{s}}\,\mathtt{s}]\,[\mathtt{i}\,\underset{8}{\mathtt{p}}\,\mathtt{p}]\,[\mathtt{i}\,\underset{11}{\$}\,\$][\,\underset{3}{\mathtt{s\,s\,i}}\,]\,[\,\underset{6}{\mathtt{s\,s\,i}}\,]\,[\,\underset{9}{\mathtt{p\,p\,i}}\,]\,[\,\underset{12}{\$\,\$\,\$}\,]$$

The key property on which the first step of the Skew algorithm hinges on, is the following:

Property 10.2 *Every suffix $T[i, n]$ starting at a position $i \in P_{2,0}$, can be put in correspondence with a suffix of R consisting of an integral sequence of triplets. Specifically, if $i \bmod 3 = 0$ then the text suffix coincides exactly with a suffix of R; if $i \bmod 3 = 2$, then the text suffix prefixes a suffix of R which nevertheless terminates with special symbol \$.*

The correctness of this property can be inferred easily by observing that any suffix $T[i, n]$ starting at a position in $P_{2,0}$ is clearly a suffix of either $T[2, \cdot]$ or $T[3, \cdot]$, given that $i > 0$, and $i \bmod 3$ is either 0 or 2. Moreover, since $i \in P_{2,0}$, it has the form $i = 3 + 3k$ or $i = 2 + 3k$, for some $k \geq 0$, and thus $T[i, n]$ occurs within R aligned to the beginning of some triplet.

By the previous running example, take $i = 6 = 0 \bmod 3$, the suffix $T[6, 12]$ = `ssippi`\$ occurs at the second triplet of $T[3, \cdot]$, which is the sixth triplet of R. Similarly, take $i = 8 = 2 \bmod 3$, the suffix $T[8, 12]$ = `ippi`\$ occurs at the third triplet of $T[2, \cdot]$, which is the third triplet of R. Notice that, even if $T[8, 12]$ is not a full suffix of R, we have that $T[8, 12]$ ends with two \$s, which will constitute sort of end-delimiters.

The final operation is then to encode those triplets via integers, and thus squeeze R into a string $T^{2,0}$ of $\frac{2n}{3}$ integer-symbols, thus realizing the reduction in length we were aiming for above. This encoding must be implemented in a way that the lexicographic comparison between two triplets can be obtained by comparing those integers. In the literature this is called *lexicographic naming* and can be easily obtained by *radix sorting* the triplets in R and associating to each distinct triplet its *rank* in the lexicographic order. Since we have $O(n)$ triplets, each consisting of symbols in a range $[0, n]$, their radix sort takes $O(n)$ time.

In our example, the sorted triplets are labeled with the following ranks:

[$ $ $]	[i $ $]	[i p p]	[i s s]	[i s s]	[p p i]	[s s i]	[s s i]	sorted triplets
0	1	2	3	3	4	5	5	sorted ranks

R = [i s s]	[i s s]	[i p p]	[i $ $]	[[s s i]	[s s i]	[p p i]	[$ $ $]	triplets
3	3	2	1	5	5	4	0	$T^{2,0}$ (string of ranks)

As a result of the naming of the triplets in R, we get the new text $T^{2,0} = 33215540$ whose length is $\frac{2n}{3}$. The crucial observation here is that we have a text $T^{2,0}$ which is again a text of integers as T, taking $O(\log n)$ bits per integer (as before), but $T^{2,0}$ has length shorter than T, so that we can invoke recursively the suffix-array construction procedure over it.

It is evident from the discussion above that, since the ranks are assigned in the same order as the lexicographic order of their triplets, the lexicographic comparison between suffixes of R (aligned to the triplets) equals the lexicographic comparison between suffixes of $T^{2,0}$.

Here Property 10.2 comes into play, because it defines a bijection between suffixes of R aligned to triplet beginnings, hence suffixes of $T^{2,0}$, with text suffixes starting in $P_{2,0}$. This correspondence is then deployed to derive $SA^{2,0}$ from the suffix array of $T^{2,0}$.

In our running example $T^{2,0} = 33215540$, the suffix-array construction algorithm is applied recursively thus deriving the suffix-array $(8, 4, 3, 2, 1, 7, 6, 5)$. We can turn this suffix array into $SA^{2,0}$ by turning the positions in $T^{2,0}$ into positions in T. This can be done via simple arithmetic operations, given the layout of the triplets in$T^{2,0}$, and obtains in our running example the suffix array $SA^{2,0} = (12, 11, 8, 5, 2, 9, 6, 3)$.

Before concluding the description of step 1, we add two notes. The first one is that, if all symbols in $T^{2,0}$ are different, then we do not need to recurse because suffixes can be sorted by looking just at their first characters. The second observation is for programmers that should be careful in turning the suffix-positions in $T^{2,0}$ into the suffix positions in T to get the final $SA^{2,0}$, because they must take into account the layout of the triplets of R.

Step 2. Once the suffix array $SA^{2,0}$ has been built (recursively), it is possible to sort lexicographically the remaining suffixes of T, namely the ones starting at the text positions $i \bmod 3 = 1$, in a simple way. We decompose a suffix $T[i, n]$ as composed by its first character $T[i]$ and its remaining suffix $T[i + 1, n]$. Since $i \in P_1$, the next position $i + 1 \in P_{2,0}$, and thus the suffix $T[i + 1, n]$ occurs in $SA^{2,0}$. We can then encode the suffix $T[i, n]$ with a pair of integers $\langle T[i], \mathtt{pos}(i + 1)\rangle$, where $\mathtt{pos}(i+1)$ denotes the lexicographic rank in $SA^{2,0}$ of the suffix $T[i+1, n]$. If $i+1 = n+1$ then we set $\mathtt{pos}(n + 1) = 0$ given that the character \$ is assumed to be smaller than any other alphabet character.

Given this observation, two text suffixes starting at positions in P_1 can then be compared in constant time by comparing their corresponding pairs. Therefore SA^1 can be computed in $O(n)$ time by radix-sorting the $O(n)$ pairs encoding its suffixes.

In our example, this boils down to radix-sort the pairs:

Pairs/suffixes:	$\langle \mathtt{m}, 4\rangle$	$\langle \mathtt{s}, 3\rangle$	$\langle \mathtt{s}, 2\rangle$	$\langle \mathtt{p}, 1\rangle$	
	1	4	7	10	starting positions in P_1
Sorted pairs/suffixes:	$\langle \mathtt{m}, 4\rangle <$	$\langle \mathtt{p}, 1\rangle <$	$\langle \mathtt{s}, 2\rangle <$	$\langle \mathtt{s}, 3\rangle$	
	1	10	7	4	SA^1

Step 3. The final step merges the two sorted arrays SA^1 and $SA^{2,0}$ in linear $O(n)$ time by resorting an interesting observation which motivates the split $\frac{2}{3} : \frac{1}{3}$. Let us take two suffixes $T[i, n] \in SA^1$ and $T[j, n] \in SA^{2,0}$, which we wish to lexicographically compare for implementing the merge-step.

They belong to two different suffix arrays so we have no *lexicographic relation* known for them, and we cannot compare them character-by-character because this would incur in a very high cost. We deploy a decomposition idea similar to the one exploited in Step 2 above, which consists of looking at a suffix as composed by *one or two characters* plus the lexicographic rank of its remaining suffix. This decomposition becomes effective if the remaining suffixes of the compared ones lie in the same suffix array, so that their rank is enough to get their order in constant time. Elegantly enough this is possible with the split $\frac{2}{3} : \frac{1}{3}$, but it could not be possible with the split $\frac{1}{2} : \frac{1}{2}$. This observation is implemented as follows:

1. if $j \bmod 3 = 2$ then we compare $T[j,n] = T[j]T[j+1,n]$ against $T[i,n] = T[i]T[i+1,n]$. Both suffixes $T[j+1,n]$ and $T[i+1,n]$ occur in $SA^{2,0}$ (given that their starting positions are congruent 0 or 2 $\mathtt{mod}$ 3, respectively), so we can derive the above lexicographic comparison by comparing the pairs $\langle T[i], \mathtt{pos}(i+1)\rangle$ and $\langle T[j], \mathtt{pos}(j+1)\rangle$. This comparison takes $O(1)$ time, provided that the array $\mathtt{pos}$ is available.[2]
2. if $j \bmod 3 = 0$ then we compare $T[j,n] = T[j]T[j+1]T[j+2,n]$ against $T[i,n] = T[i]T[i+1]T[i+2,n]$. Both the suffixes $T[j+2,n]$ and $T[i+2,n]$ occur in $SA^{2,0}$ (given that their starting positions are congruent 0 or 2 $\mathtt{mod}$ 3, respectively), so we can derive the above lexicographic comparison by comparing the triples $\langle T[i], T[i+1], \mathtt{pos}(i+2)\rangle$ and $\langle T[j], T[j+1], \mathtt{pos}(j+2)\rangle$. This comparison takes $O(1)$ time, provided that the array $\mathtt{pos}$ is available.

In our running example we have that $T[8,11] < T[10,11]$, and in fact $\langle i, 5\rangle < \langle p, 1\rangle$. Also we have that $T[7,11] < T[6,11]$ and in fact $\langle s, i, 5\rangle < \langle s, s, 2\rangle$. In the following figure we depict all possible pairs of triples which may be involved in a comparison, where (★★) and (★★★) denote the pairs for rule 1 and 2 above, respectively. Conversely (★) denotes the starting position in T of the suffix. Notice that, since we do not know which suffix of $SA^{2,0}$ will be compared with a suffix of SA^1 during the merging process, for each of the latter suffixes we need to compute both representations (★★) and (★★★), hence as a pair and as a triplet.[3]

SA^1				$SA^{2,0}$								
1	10	7	4	12	11	8	5	2	9	6	3	(★)
$\langle \mathtt{m}, 4\rangle$	$\langle \mathtt{p}, 1\rangle$	$\langle \mathtt{s}, 2\rangle$	$\langle \mathtt{s}, 3\rangle$		$\langle \mathtt{i}, 0\rangle$	$\langle \mathtt{i}, 5\rangle$	$\langle \mathtt{i}, 6\rangle$	$\langle \mathtt{i}, 7\rangle$				(★★)
$\langle \mathtt{m}, \mathtt{i}, 7\rangle$	$\langle \mathtt{p}, \mathtt{i}, 0\rangle$	$\langle \mathtt{s}, \mathtt{i}, 5\rangle$	$\langle \mathtt{s}, \mathtt{i}, 6\rangle$	$\langle \$, \$, -1\rangle$					$\langle \mathtt{p}, \mathtt{p}, 1\rangle$	$\langle \mathtt{s}, \mathtt{s}, 2\rangle$	$\langle \mathtt{s}, \mathtt{s}, 3\rangle$	(★★★)

At the end of the merge step we obtain the final suffix array: $SA = (12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3)$.

From the discussion above it is clear that every step can be implemented via the *sorting* or the scanning of a set of n atomic items, which are possibly triplets of integers, taking each triplet $O(\log n)$ bits, so one memory word. Therefore the proposed method can be seen as a *algorithmic reduction* of the suffix-array construction problem to the classic problem of sorting n-items. This problem has been solved optimally in several models of computation, for the case of the two-level memory model see Chapter 5.

For what concerns the RAM model, the time complexity of the Skew algorithm can be modeled by the recurrence $T(n) = T(\frac{2n}{3}) + O(n)$, because Steps 2 and 3 cost $O(n)$ and the recursive call is executed over the string $T^{2,0}$ whose length is $(2/3)n$. This recurrence has solution $T(n) = O(n)$, which is clearly optimal. For what concerns the two-level memory model, the Skew algorithm can be implemented in $O(\frac{n}{B} \log_{M/B} \frac{n}{M})$ I/Os, that is the I/O-complexity of sorting n atomic items.

[2] Of course, the array $\mathtt{pos}$ can be derived from $SA^{2,0}$ in linear time, since it is its inverse.

[3] Recall that $\mathtt{pos}(n) = 0$, and for the sake of the lexicographic order, we can set $\mathtt{pos}(j) = -1$, for all $j > n$.

THEOREM 10.3 *The Skew algorithm builds the suffix array of a text string $T[1,n]$ in $O(Sort(n))$ I/Os and $O(n/B)$ disk pages. If the alphabet Σ has size polynomial in n, the CPU time is $O(n)$.*

The Scan-based Algorithm$^{\infty}$

Before the introduction of the Skew algorithm, the best known disk-based algorithm was the one proposed by Baeza-Yates, Gonnet and Sniders in 1992 [6]. It is also a divide&conquer algorithm whose divide step is strongly unbalanced, thus it executes a quadratic number of suffix comparisons which induce a *cubic* time complexity. Nevertheless the algorithm is fast in practice because it processes the data into passes thus deploying the high throughput of modern disks.

Let $\ell < 1$ be a positive constant, properly fixed to build the suffix array of a text piece of $m = \ell M$ characters in internal memory. Then assume that the text $T[1,n]$ is logically divided into pieces of m characters each, numbered rightward: namely $T = T_1 T_2 \cdots T_{n/m}$ where $T_h = T[hm+1,(h+1)m]$ for $h = 0, 1, \ldots$. The algorithm computes *incrementally* the suffix array of T in $\Theta(n/M)$ stages, rather than the logarithmic number of stages of the Skew algorithm. At the beginning of stage h, we assume to have on disk *the array SA^h that contains the sorted sequence of the first hm suffixes of T*. Initially $h = 0$ and thus SA^0 is the empty array. In the generic h-th stage, the algorithm loads the next text piece T^{h+1} in internal memory, builds SA' as the sorted sequence of suffixes starting in T^{h+1}, and then computes the new SA^{h+1} by merging the two sorted sequences SA^h and SA'.

There are two main issues when detailing this algorithmic idea in a running code: how to efficiently construct SA', since its suffixes start in T^{h+1} but may extend outside that block of characters up to the end of T; and how to efficiently merge the two sorted sequences SA^h and SA', since they involve suffixes whose length may be up to $\Theta(n)$ characters. For the first issue the algorithm does not implement any special trick, it just compares pairs of suffixes character-by-character in $O(n)$ time and $O(n/B)$ I/Os. This means that over the total execution of the $O(n/M)$ stages, the algorithm takes $O(\frac{n}{B}\frac{n}{m} m \log m) = O(\frac{n^2}{B}\log m)$ I/Os to construct SA'.

For the second issue, we note that the merge between SA' with SA^h is executed in a smart way by resorting the use of an auxiliary array $C[1, m+1]$ which counts in $C[j]$ the number of suffixes of SA^h that are lexicographically greater than the $SA'[j-1]$-th text suffix and smaller than the $SA'[j]$-th text suffix. Two special cases occur if $j = 1, m+1$: in the former case we assume that $SA'[0]$ is the empty suffix, in the latter case we assume that $SA'[m+1]$ is a special suffix larger than any string. Since SA^h is longer and longer, we process it streaming-like by devising a method that scans rightward the text T (from its beginning) and then searches each of its suffixes by binary-search in SA'. If the lexicographic position of the searched suffix is j, then the entry $C[j]$ is incremented. The binary search may involve a part of a suffix which lies outside the block T^{h+1} currently in internal memory, thus taking $O(n/B)$ I/Os per binary-search step. Over all the n/M stages, this binary search takes $O(\sum_{h=0}^{n/m-1} \frac{n}{B}(hm)\log m) = O(\frac{n^3}{MB}\log M)$ I/Os.

Array C is then exploited in the next substep to quickly merge the two arrays SA' (residing in internal memory) and SA^h (residing on disk): $C[j]$ indicates how many consecutive suffixes of SA^h lexicographically lie after $SA'[j-1]$ and before $SA'[j]$. Hence a disk scan of SA^h suffices to perform the merging process in $O(n/B)$ I/Os.

THEOREM 10.4 *The Scan-based algorithm builds the suffix array of a text string $T[1,n]$ in $O(\frac{n^3}{MB}\log M)$ I/Os and $O(n/B)$ disk pages.*

Since the worst-case number of total I/Os is cubic, a purely theoretical analysis would classify this algorithm as not interesting. However, in practical situations it is very reasonable to assume that each suffix comparison finds in internal memory all the characters used to compare the two involved suffixes. And indeed the practical behavior of this algorithm is better described by the formula $O(\frac{n^2}{MB})$ I/Os. Additionally, all I/Os in this analysis are sequential and the actual number of random seeks is only $O(n/M)$ (i.e., at most a constant number per stage). Consequently, the algorithm takes fully advantage of the large bandwidth of modern disks and of the high speed of current CPUs. As a final notice we remark that the suffix arrays SA^h and the text T are scanned sequentially, so some form of compression can be adopted to reduce the I/O-volume and thus further speed-up the underlying algorithm.

Before detailing a significant improvement to the previous approach, let us concentrate on the same running example used in the previous section to sketch the Skew algorithm.

$$\begin{array}{rcccccccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & {\scriptstyle 10} & {\scriptstyle 11} & {\scriptstyle 12} \\ T[1,12] = & \mathrm{m} & \mathrm{i} & \mathrm{s} & \mathrm{s} & \mathrm{i} & \mathrm{s} & \mathrm{s} & \mathrm{i} & \mathrm{p} & \mathrm{p} & \mathrm{i} & \$ \end{array}$$

Suppose that $m = 3$ and that, at the beginning of stage $h = 1$, the algorithm has already processed the text block $T^0 = T[1,3] =$ `mis` and thus stored on disk the array $SA^1 = (2,1,3)$ which corresponds to the lexicographic order of the text suffixes which start in that block: namely, `mississippi$`, `ississippi$` and `ssissippi$`. During the stage $h = 1$, the algorithm loads in internal memory the next block $T^1 = T[4,6] =$ `sis` and lexicographically sorts the text suffixes which start in positions $[4,6]$ and extend to the end of T, see figure 10.4.

Text suffixes	`sissippi$`	`issippi$`	`ssippi$`
		⇓	
		Lexicographic ordering	
		⇓	
Sorted suffixes	`issippi$`	`sissippi$`	`ssippi$`
SA′	5	4	6

FIGURE 10.4: Stage 1, step 1, of the Scan-based algorithm.

The figure shows that the comparison between the text suffixes: $T[4,12] =$ `sissippi$` and $T[6,12] =$ `ssippi$` involves characters that lie outside the text piece $T[4,6]$ loaded in internal memory, so that their comparison induces some I/Os.

The final step merges $SA^1 = (2,1,3)$ with $SA' = (5,4,6)$, in order to compute SA^2. This step uses the information of the counter array C. In this specific running example, see Figure 10.5, it is $C[1] = 2$ because two suffixes $T[1,12] =$ `mississippi$` and $T[2,12] =$ `ississippi$` are between the $SA'[0]$-th suffix `issippi$` and the $SA'[1]$-th suffix `sissippi$`.

Suffix Arrays	$\underbrace{\underline{SA' = [5,4,6] \quad SA^1 = [\mathbf{2},\mathbf{1},\mathbf{3}]}}$
Merge via C	$\Downarrow {\scriptstyle C=[0,2,0,1]}$
	$SA^2 = [5,\mathbf{2},\mathbf{1},4,6,\mathbf{3}]$

FIGURE 10.5: Stage 1, step 3, of the Scan-based algorithm

The second stage is summarized in Figure 10.6 where the text substring $T^2 = T[7,9] =$ `sip` is loaded in memory and the suffix array SA' for the suffixes starting at positions $[7,9]$ is built. Then, the suffix array SA' is merged with the suffix array SA^2 residing on disk and containing the suffixes which start in $T[1,6]$.

Stage 2:

(1) Load into internal memory $T^2 = T[7,9] =$ `sip`.

(2) Build SA' for the suffixes starting in $[7,9]$:

Text suffixes	`sippi$`	`ippi$`	`ppi$`
		⇓	
		Lexicographic ordering	
		⇓	
Sorted suffixes	`ippi$`	`ppi$`	`sippi$`
SA′	8	9	7

(3) Merge SA' with SA^2 exploiting C:

$$\begin{array}{r|c} \textbf{Suffix Arrays} & \underbrace{SA' = [8,9,7] \quad SA^2 = [\mathbf{5},\mathbf{2},\mathbf{1},\mathbf{4},\mathbf{6},\mathbf{3}]} \\ \textbf{Merge via } C & \Downarrow {\scriptstyle C=[0,3,0,3]} \\ & SA^3 = [8,\mathbf{5},\mathbf{2},\mathbf{1},9,7,\mathbf{4},\mathbf{6},\mathbf{3}] \end{array}$$

FIGURE 10.6: Stage 2 of the Scan-based algorithm.

The third and last stage is summarized in Figure 10.7 where the substring $T^3 = T[10,12] =$ `pi$` is loaded in memory and the suffix array SA' for the suffixes starting at positions $[10,12]$ is built. Then, the suffix array SA' is merged with the suffix array on disk SA^3 containing the suffixes which start in $T[1,9]$.

The performance of this algorithm can be improved via a simple observation [4]. Assume that, at the beginning of stage h, in addition to the SA^h we have on disk a bit array, called gt_h, such that $\mathsf{gt_h[i]} = 1$ if and only if the suffix $T[(hm+1)+i, n]$ is Greater Than the suffix $T[(hm+1), n]$. The computation of gt can occur efficiently, but this technicality is left to the original paper [4] and not detailed here.

During the h-th stage the algorithm loads into internal memory the substring $t[1,2m] = T^h T^{h+1}$ (so this is double in size with respect to the previous proposal) and the binary array $\mathsf{gt}_{h+1}[1, m-1]$ (so it refers to the second block of text loaded in internal memory). The key observation is that we can build SA' by deploying the two arrays above without performing any I/Os, other than the ones needed to load $t[1,2m]$ and $\mathsf{gt}_{h+1}[1, m-1]$. This seems surprising, but it descends from the fact that any two text suffixes starting at positions i and j within T^h, with $i < j$, can be compared lexicographically by looking first at their characters in the substring t, namely at the strings $t[i,m]$ and $t[j, j+m-i]$. These two strings have the same length and are completely in $t[1,2m]$, hence in internal memory. If these strings differ, their order is determined and we are done; otherwise, the order between these two suffixes is determined by the order of the remaining suffixes starting at the characters $t[m+1]$ and $t[j+m-i+1]$. This order is given by the bit stored in $\mathsf{gt}_{h+1}[j-i]$, also available in internal memory.

This argument shows that the two arrays t and gt_{h+1} contain all the information we need to build

Stage 3*:*

(1) Load into internal memory $T^3 = T[10, 12] = pi\$$.

(2) Build SA' for the suffixes starting in [10, 12]:

Text suffixes	`pi$ i$ $`
	⇓
	Lexicographic ordering
	⇓
Sorted suffixes	`$ i$ pi$`
SA′	12 11 10

(3) Merge SA' with SA^3 exploiting C:

Suffix Arrays	$\underbrace{SA' = [12, 11, 10] \quad SA^3 = [\mathbf{8}, \mathbf{5}, \mathbf{2}, \mathbf{1}, \mathbf{9}, \mathbf{7}, \mathbf{4}, \mathbf{6}, \mathbf{3}]}$
Merge via ***C***	$\Downarrow$ C=[0,0,4,5]
	$SA^4 = [12, 11, \mathbf{8}, \mathbf{5}, \mathbf{2}, \mathbf{1}, 10, \mathbf{9}, \mathbf{7}, \mathbf{4}, \mathbf{6}, \mathbf{3}]$

FIGURE 10.7: Stage 3 of the Scan-based algorithm.

SA^{h+1} working in internal memory, and thus without performing any I/Os.

THEOREM 10.5 *The new variant of the Scan-based algorithm builds the suffix array of a string $T[1, n]$ in $O(\frac{n^2}{MB})$ I/Os and $O(n/B)$ disk pages.*

As an example consider stage $h = 1$ and thus load in memory the text substring $t = T^h T^{h+1} = T[4, 9]$ = `sis sip` and the array $\mathsf{gt}_2 = (1, 0)$. Now consider the positions $i = 1$ and $j = 3$ in t, we can compare the text suffixes starting at these positions by first taking the substrings $t[1, 3] = T[4, 6]$ = `sis` with $t[3, 5] = T[6, 9]$ `ssi`. The strings are different so we obtain their order without accessing the disk. Now consider the positions $i = 3$ and $j = 4$ in t, they would not be taken into account by the algorithm since the block has size 3, but let us consider them for the sake of explanation. We can compare the text suffixes starting at these positions by first taking the substrings $t[3, 3]$ = `s` with $t[4, 4]$ = `s`. The strings are not different so we use $\mathsf{gt}_2[j - i] = \mathsf{gt}_2[1] = 1$, hence the remaining suffix $T[4, n]$ is lexicographically greater than $T[5, n]$ and this can be determined again without any I/Os.

10.3 The Suffix Tree

The *suffix tree* is a fundamental data structure used in many algorithms processing strings [5]. In its essence it is a compacted trie that stores all suffixes of an input string, each suffix is represented by a (unique) path from the root of the trie to one of its leaves. We already discussed compacted tries in the previous chapter, now we specialize the description in the context of suffix trees and point out some issues, and their efficient solutions, that arise when the dictionary of indexed strings is composed by suffixes of one single string.

Let us denote the suffix tree built over an input string $T[1, n]$ as ST_T (or just ST when the input is clear from the context) and assume, as done for suffix arrays, that the last character of T is the special symbol \$ which is smaller than any other alphabet character. The suffix tree has the following properties:

1. Each suffix of T is represented by a *unique* path descending from root of ST to one of its leaves. So there are n leaves, one per text suffix, and each leaf is labeled with the starting position in T of its corresponding suffix.
2. Each internal node of ST has at least two outgoing edges. So there are less than n internal nodes and less than $2n - 1$ edges. Every internal node u spells out a text substring, denoted by $s[u]$, which prefixes everyone of the suffixes descending from u in the suffix tree. Typically the value $|s[u]|$ is stored as satellite information of node u, and we use $occ[u]$ to indicate the number of leaves descending from u.
3. The edge labels are non empty substrings of T. The labels of the edges spurring from any internal node start with different characters, called *branching characters*. Edges are assumed to be ordered alphabetically according to their branching characters. So every node has at most σ outgoing edges.[4]

In Figure 10.8 we show the suffix tree built over our exemplar text $T[1, 12] =$ `mississippi$`. The presence of the special symbol $T[12] =$ \$ ensures that no suffix is a prefix of another suffix of T and thus every pair of suffixes differs in some character. So the paths from the root to the leaves of two different suffixes coincide up to their common longest prefix, which ends up in an internal node of ST.

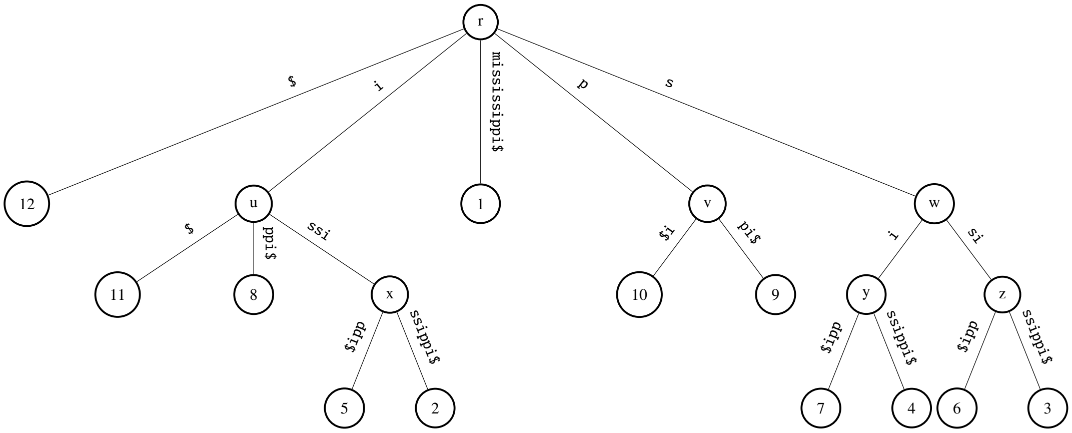


FIGURE 10.8: The suffix tree of the string `mississippi$`

It is evident that we cannot store explicitly the substrings labeling the edges because this would end up in a total space complexity of $\Theta(n^2)$. You can convince yourself by building the suffix tree for the string consisting of all distinct characters, and observe that the suffix tree consists of one root connected to n leaves with edges representing all suffixes. We can circumvent this space explosion by encoding the edge labels with pairs of integers which represent the starting position

[4] The special character \$ is included in the alphabet Σ.

of the labeling substring and its length. With reference to Figure 10.8 we have that the label of the edge leading to leaf 5, namely the substring $T[9, 12]$ = `ppi$`, can be encoded with the integer pair $\langle 9, 4\rangle$, where 9 is the offset in T and 4 is the length of the label. Other obvious encodings could be possible — say the pair $\langle 9, 12\rangle$ indicating the starting and ending position of the label—, but we will not detail them here. Anyway, whichever is the edge encoding adopted, it uses $O(1)$ space, and thus the storage of all edge labels takes $O(n)$ space, independently of the indexed string.

FACT 10.3 *The suffix tree of a string $T[1, n]$ consists of n leaves, at most $n - 1$ internal nodes and at most $2n - 2$ edges. Its space occupancy is $O(n)$, provided that a proper edge-label encoding is adopted.*

As a final notation, we call *locus* of a text substring t the node v whose spelled string is exactly t, hence $s[v] = t$. We call *extended locus* of t' the locus of its shortest extension that has defined locus in ST. In other words, the path spelling the string t' in ST ends within an edge label, say the label of the edge (u, v). This way $s[u]$ prefixes t' which in turn prefixes $s[v]$. Therefore v is the extended locus of t'. Of course if t' has a locus in ST then this coincides with its extended locus. As an example, the node z of the suffix tree in Figure 10.8 is the locus of the substring `ssi` and the extended locus of the substring `ss`.

There are few important properties that the suffix-tree data structure satisfies, they pervade most algorithms which hinge on this powerful data structure. We summarize few of them:

Property 10.6 *Let α be a substring of the text T, then there exists an internal node u such that $s[u] = \alpha$ (hence u is the locus of α) iff they do exist at least two occurrences of α in T followed by distinct characters.*

As an example, take node x in Figure 10.8, the substring $s[x]$ = `issi` occurs twice in T at positions 2 and 5, followed by characters `i` and `p`, respectively.

Property 10.7 *Let α be a substring of the text T that has extended locus in the suffix tree. Then every occurrence of α is followed by the same character in T.*

As an example, take the substring `iss` that has node x as extended locus in Figure 10.8. This substring occurs twice in T at positions 2 and 5, followed always by character `i`.

Property 10.8 *Every internal node u spells out a substring $s[u]$ of T which occurs at the positions $occ[u]$ and is* maximal*, in the sense that it cannot be extended by one character and yet occur at these positions.*

Now we introduce the notion of *lowest common ancestor* (shortly, `lca`) in trees, which is defined for every pair of leaves and denotes the deepest node being ancestor of both leaves in input. As an example in Figure 10.8, we have that u is the `lca` of leaf 8 and 2. Now we turn `lca` between leaves into `lcp` between their corresponding suffixes.

Property 10.9 *Given two suffixes $T[i, n]$ and $T[j, n]$, say ℓ is the length of the longest common prefix between them. This value can be identified by computing the lowest common ancestor $a(i, j)$ between the leaves in the suffix tree corresponding to those two suffixes. Therefore, we have $s[a(i, j)]$ = `lcp`$(T[i, n], T[j, n])$.*

As an example, take the suffixes $T[11, 12]$ = `i$` and $T[5, 12]$ = `issippi$`, their `lcp` is the single character `i` and the `lca` between their leaves is the node u, which indeed spells out the string $s[u]$ = `i`.

10.3.1 The substring-search problem

The search for a pattern $P[1, p]$ as a substring of the text $T[1, n]$, with the help of the suffix tree ST, consists of a tree traversal which starts from its root and proceeds downward as pattern characters are matched against characters labeling the tree edges (see Figure 10.9). Note that, since the first character of the edges outgoing from each traversed node is distinct, the matching of P can follow only one downward path. If the traversal determines a mismatch character, the pattern P does not occur in T; otherwise the pattern is fully matched, the extended locus of P is found, and all leaves of ST descending from this node identify all text suffixes which are prefixed by P. The text positions associated to these descending leaves are the positions of the *occ* occurrences of the pattern P in T. These positions can be retrieved in $O(occ)$ time by visiting the subtree that descends from the extended locus of P. In fact this subtree has size $O(occ)$ because its internal nodes have (at least) binary fan-out and consists of *occ* leaves.

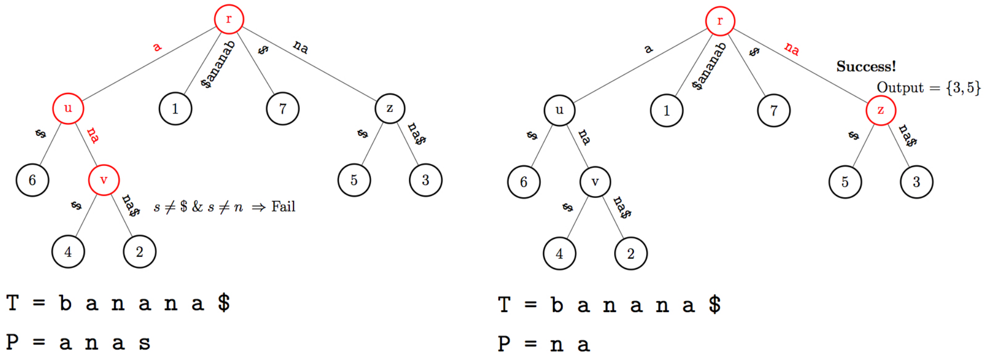


FIGURE 10.9: Two examples of substring searches over the suffix tree built for the text banana$. The search for the pattern P = anas fails, the other for the pattern P = na is successful.

In the running example of Figure 10.9, the pattern P = na occurs twice in T and in fact the traversal of ST fully matches P and stops at the node z, from which descend two leaves labeled 3 and 5. And indeed the pattern P occurs at positions 3 and 5 of T, since it prefixes the two suffixes $T[3, 12]$ and $T[5, 12]$. The cost of pattern searching is $O(pt_\sigma + occ)$ time in the worst case, where t_σ is the time to branch out of a node during the tree traversal. This cost depends on the alphabet size σ and the kind of data structure used to store the branching characters of the edges spurring from each node. We discussed this issue in the previous Chapter, when solving the prefix-search problem via compacted tries. There we observed that $t_\sigma = O(1)$ if we use a perfect-hash table indexed by the branching characters; it is $t_\sigma = O(\log \sigma)$ if we use a plain array and the branching is implemented by a binary search. In both cases the space occupancy is optimal, in that it is linear in the number of branching edges, and thus $O(n)$ overall.

FACT 10.4 *The occ occurrences of a pattern $P[1, p]$ in a text $T[1, n]$ can be found in $O(p + occ)$ time and $O(n)$ space by using a suffix tree built on the input text T, in which the branching characters at each node are indexed via a perfect hash table.*

10.3.2 Construction from Suffix Arrays and vice versa

It is not difficult to observe that the suffix array SA of the text T can be obtained from its suffix tree ST by performing an in-order visit: each time a leaf is encountered, the suffix-index stored in this leaf is written into the suffix array SA; each time an internal node u is encountered, its associated value is written into the array `lcp`.

FACT 10.5 *Given the suffix tree of a string* $T[1, n]$*, we can derive in* $O(n)$ *time and space the corresponding suffix array SA and the longest-common-prefix array* `lcp`*.*

Vice versa, we can derive the suffix tree ST from the two arrays SA and `lcp` in $O(n)$ time as follows. The algorithm constructs incrementally ST starting from a tree, say ST_1, that contains a root node denoting the empty string and one leaf labeled $SA[1]$, denoting the smallest suffix of T. At step $i > 1$, we have inductively constructed the partial suffix tree ST_{i-1} which contains all the $(i-1)$-smallest suffixes of T, hence the suffixes in $SA[1, i-1]$. During step i, the algorithm inserts in ST_{i-1} the i-th smallest suffix $SA[i]$. This requires the addition of one leaf labeled $SA[i]$ and, as we will prove next, at most one single internal node which becomes the father of the inserted leaf. After n steps, the final tree ST_n will be the suffix tree of the string $T[1, n]$.

The key issue here is to show how to insert the leaf $SA[i]$ into ST_{i-1} in constant amortized time. This will be enough to ensure a total time complexity of $O(n)$ for the overall construction process. The main difficulty consists in the detection of the node u father of the leaf $SA[i]$. This node u may already exist in ST_{i-1}, in this case $SA[i]$ is attached to u; otherwise, u must be created by splitting an edge of ST_{i-1}. Whether u exists or not is discovered by percolating ST_{i-1} upward (and not downward!), starting from the leaf $SA[i-1]$, which is the rightmost one in ST_{i-1} because of the lexicographic order, and stopping when a node x is reached such that $\texttt{lcp}[i] \leq |s[x]|$. Recall that $\texttt{lcp}[i]$ is the number of characters that the text suffix $suff_{SA[i-1]}$ shares with next suffix $suff_{SA[i]}$ in the lexicographic order. The leaves corresponding to these two suffixes are of course consecutive in the in-order visit of ST. At this point if $\texttt{lcp}[i] = |s[x]|$, the node x is the parent of the leaf labeled $SA[i]$, we connect them and the new ST_i is obtained. If instead $\texttt{lcp}[i] < |s[x]|$, the edge leading to x has to be split by inserting a node u that has two children: the left child is x and the right child is the leaf $SA[i]$ (because it is lexicographically larger than $SA[i-1]$). This node is associated with the value $\texttt{lcp}[i]$. The reader can run this algorithm over the string $T[1, 12] = $ `mississippi$` and convince herself that the final suffix tree ST_{12} is exactly the one showed in Figure 10.8.

The time complexity of the algorithm derives from an accounting argument which involves the edges traversed by the upward percolation of ST. Since the suffix $suff_{SA[i]}$ is lexicographically greater than the suffix $suff_{SA[i-1]}$, the leaf labeled $SA[i]$ lies to the right of the leaf $SA[i-1]$. So every time we traverse an edge, we either discard it from the next traversals and proceed upward, or we split it and a new leaf is inserted. In particular all edges from $SA[i-1]$ to x are never traversed again because they lie to the left of the newly inserted edge $(u, SA[i])$. The total number of these edges is bounded by the total number of edges in ST, which is $O(n)$ from Fact 10.3. The total number of edge-splits equals the number of inserted leaves, which is again $O(n)$.

FACT 10.6 *Given the suffix array and the longest-common-prefix array of a string* $T[1, n]$*, we can derive the corresponding suffix tree in* $O(n)$ *time and space.*

10.3.3 McCreight's algorithm$^{\infty}$

A naïve algorithm for constructing the suffix tree of an input string $T[1, n]$ could start with an empty trie and then iteratively insert text suffixes, one after the other. The algorithm maintains the property by which each intermediate trie is indeed a compacted trie of the suffixes inserted so far. In the worst case, the algorithm costs up to $O(n^2)$ time, take e.g. the highly repetitive string $T[1, n] = a^{n-1}\$$. The reason for this poor behavior is due to the *re-scanning* of parts of the text T that have been already examined during the insertion of previous suffixes. Interestingly enough do exist algorithms that construct the suffix tree directly, and thus without passing through the suffix- and lcp-arrays, and still take $O(n)$ time. Nowadays the space succinctness of suffix arrays and the existence of the Skew algorithm, drive the programmers to build suffix trees passing through suffix arrays (as explained in the previous section). However, if the *average lcp* among the text suffixes is small then the direct construction of the suffix tree may be advantageous both in internal memory and on disk. We refer the interested reader to [3] for a deeper analysis of these issues.

In what follows we present the classic McCreight's algorithm [11], introduced in 1976. It is based on a nice technique that adds some special pointers to the suffix tree that allow to avoid the *rescanning* mentioned before. These special pointers are called *suffix links* and are defined as follows. The suffix link $SL(z)$ connects the node z to the node z' such that $s[z] = as[z']$. So z' spells out a string that is obtained by dropping the first character from $s[z]$. The existence of z' in ST is not at all clear: Of course $s[z']$ is a substring of T, given that $s[z]$ is, and thus there exists a path in ST that ends up into the extended locus of $s[z']$; but nothing seems to ensure that $s[z']$ has indeed a locus in ST, and thus that z' exists. This property is derived by observing that the existence of z implies the existence of at least 2 suffixes, say $suff_i$ and $suff_j$ that have the node z as their lowest common ancestor in ST, and thus $s[z]$ is their longest common prefix (see Property 10.9). Looking at Figure 10.8, we can take for node z the suffixes $suff_3$ and $suff_6$ (which are actually children of z). Now take the two suffixes following those ones, namely $suff_{i+1}$ and $suff_{j+1}$ (i.e. $suff_4$ and $suff_7$ in the figure). They will share $s[z']$ as their longest common prefix, given that we dropped just their first character, and thus they will have z' as their lowest common ancestor. In Figure 10.8, $s[z] =$ `ssi`, $s[z'] =$ `si` and the node z' does exist and is indicated with y. In conclusion every node z has one suffix link correctly defined; more subtle is to observe that all suffix links form a tree rooted in the root of ST: just observe that $|s[z']| < |s[z]|$ so they cannot induce cycles and eventually end up in the root of the suffix tree (spelling out the empty string).

McCreight's algorithm works in n steps, it starts with the suffix tree ST_1 which consists of a root node, denoting the empty string, and one leaf labeled $suff_1 = T[1, n]$ (namely the entire text). In a generic step $i > 1$, the current suffix tree ST_{i-1} is the compacted trie built over all text suffixes $suff_j$ such that $j = 1, 2, \ldots, i-1$. Hence suffixes are inserted in ST from the longest to the shortest one, and at any step ST_{i-1} indexes the $(i-1)$ longest suffixes of T.

To ease the description of the algorithm we need to introduce the notation $head_i$ which denotes the longest prefix of suffix $suff_i$ which occurs in ST_{i-1}. Given that ST_{i-1} is a partial suffix tree, $head_i$ is the longest common prefix between $suff_i$ and any of its previous suffixes in T, namely $suff_j$ with $j = 1, 2, \ldots, i-1$. Given $head_i$ we denote by h_i the (extended) locus of that string in the current suffix tree: actually h_i is the extended locus in ST_{i-1} because $suff_i$ has not yet been inserted. After its insertion, we will have that $head_i = s[h_i]$ in ST_i, and indeed h_i is set as the parent of the leaf associated to the suffix $suff_i$. As an example, consider the suffix $suff_5 =$ `byabz$` in the partial suffix trees of Figure 10.10. We have that this suffix shares only the character `b` with the previous four suffixes of T, so $head_5 =$ `b` in ST_4, and $head_5$ has extended locus in ST_4, which is the leaf 2. But, after its insertion, we get the suffix tree ST_5 in which $h_5 = v$ in ST_5.

Now we are ready to describe the McCreight's algorithm in detail. To produce ST_i, we must locate in ST_{i-1} the (extended) locus h_i of $head_i$. If it is an extended locus, then the edge incident in this node is split by inserting an internal node, which corresponds to h_i, and spells out $head_i$, to which the leaf for $suff_i$ is attached. In the naïve algorithm, $head_i$ and h_i were found tracing a downward path in ST_{i-1} matching $suff_i$ character-by-character. However this induced a quadratic time complexity in the worst case. Instead McCreight's algorithm determines $head_i$ and h_i by using the information inductively available for string $head_{i-1}$, and its locus h_{i-1}, and the *suffix links* which

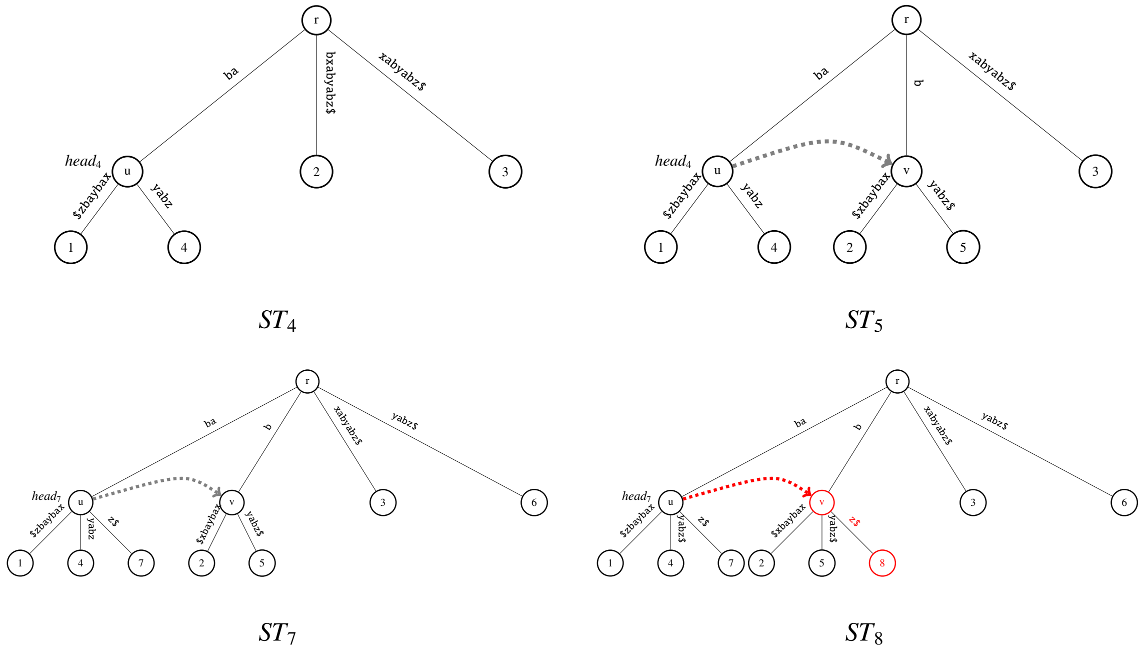


FIGURE 10.10: Several steps of the McCreight's algorithm for the string T =`abxabyabz$`.

are already available in ST_{i-1}.

FACT 10.7 *In ST_{i-1} the suffix link $SL(u)$ is defined for all nodes $u \neq h_{i-1}$. It may be the case that $SL(h_{i-1})$ is defined too, because that node was already present in ST_{i-1} before the insertion of $suff_{i-1}$.*

Proof Since $head_{i-1}$ prefixes $suff_{i-1}$, the second suffix of $head_{i-1}$ starts at position i and thus prefixes the suffix $suff_i$. We denote this second suffix with $head^-_{i-1}$. By definition $head_i$ is the longest prefix shared between $suff_i$ and anyone of the previous text suffixes, so that $|head_i| \geq |head_{i-1}| - 1$ and the string $head^-_{i-1}$ prefixes $head_i$. ∎

McCreight's algorithm starts with ST_1 that consists of two nodes: the root and the leaf for $suff_1$. At step 1 we have that $head_1$ is the empty string, h_1 is the root, and $SL(root)$ points to the root itself. At a generic step $i > 1$, we know $head_{i-1}$ and h_{i-1} (i.e. the parent of $suff_{i-1}$), and we wish to determine $head_i$ and h_i, in order to insert the leaf for $suff_i$ as a child of h_i. These data are found via the following three sub-steps:

1. if $SL(head_{i-1})$ is defined, we set $w = SL(head_{i-1})$ and we go to step 3;
2. Otherwise we need to perform a **rescanning** whose goal is to find/create the locus w of $head^-_{i-1}$ and consequently set the suffix link $SL(h_{i-1}) = w$. This is implemented by taking the parent f of $head_{i-1}$, jumping via its suffix link $f' = SL(f)$ (which is defined according to Fact 10.7), and then tracing a downward path from f' starting from the $(|s[f']| + 1)$-th character of $suff_i$. Since we know that $head^-_{i-1}$ occurs in T and it prefixes $suff_i$, this downward tracing to find w can be implemented by comparing only the branching characters of the traversed edges with $head^-_{i-1}$. If the landing node of this

traversal is the locus of $head^{-}_{i-1}$, then this landing node is the searched w; otherwise the landing node is the extended locus of $head^{-}_{i-1}$, so we split the last traversed edge and insert the node w such that $s[w] = head^{-}_{i-1}$. In all cases we set $SL(h_{i-1}) = w$;

3. Finally, we locate $head_i$ starting from w and **scanning** the rest of $suff_i$. If the locus of $head_i$ does exist, then we set it to h_i; otherwise the scanning of $head_i$ stopped within some edge, and so we split it by inserting h_i as the locus of $head_i$. We conclude the process by installing the leaf for $suff_i$ as a child of h_i.

Figure 10.10 shows an example of the advantage induced by suffix links. As step 8 we have the partial suffix tree ST_7, $head_7 = \texttt{ab}$, its locus $h_7 = u$, and we need to insert the suffix $suff_8 = \texttt{bz\$}$. Using McCreight's algorithm, we find that $SL(h_7)$ is defined and equal to v, so we reach that node following the suffix link (without rescanning $head^{-}_{i-1}$). Subsequently, we scan the rest of $suff_8$, namely `z$`, searching for the locus of $head_8$, but we find that actually $head_8 = head^{-}_{7}$, so $h_8 = v$ and we can attach there the leaf 8.

From the point of view of time complexity, we observe that the rescanning and the scanning steps perform two different types of traversals: the former traverses edges by comparing only the branching characters, since it is rescanning the string $head^{-}_{i-1}$ which is already known from the previous step $i - 1$; the latter traverses edges by comparing their labels in their entirety because it has to determine $head_i$. This last type of traversal always advances in T so the cost of the scanning phase is $O(n)$. The difficulty is to evaluate that the cost of rescanning is $O(n)$ too. The proof comes from an observation on the structure of suffix links and suffix trees: if $SL(u) = v$ then all ancestors of u point to a distinct ancestor of v. This comes from Fact 10.7 (all these suffix links do exist), and from the definition of suffix links (which ensures ancestorship). Hence the tree-depth of $v = SL(u)$, say $d[v]$, is larger than $d[u] - 1$ (where -1 is due to the dropping of the first character). Therefore, the execution of rescanning can decrease the current depth at most by 2 (i.e., one for reaching the father of h_{i-1}, one for crossing $SL(h_{i-1})$). Since the depth of ST is most n, and we loose at most two levels per SL-jump, then the number of edges traversed by rescanning is $O(n)$, and each edge traversal takes $O(1)$ time because only the branching character is matched.

The last issue to be considered regards the cost of branching out of a node during the re-scanning and the scanning steps. Previously we stated that this costs $O(1)$ by using perfect hash-tables built over the branching characters of each internal node of ST. In the context of suffix-tree construction the tree is dynamic and thus we should adopt dynamic perfect hash-tables, which is a pretty involved solution. A simpler approach consists of keeping the branching characters and their associated edges within a binary-search tree thus supporting the branching in $O(\log \sigma)$ time. Practically, programmers relax the requirement of worst-case complexity and use either hash tables with chaining, or dictionary data structures for integer values (such as the Van Emde-Boas tree, whose search complexity is $O(\log \log \sigma)$ time) because characters can be looked at as sequences of bits and hence integers.

THEOREM 10.10 *McCreight's algorithm builds the suffix tree of a string $T[1, n]$ in $O(n \log \sigma)$ time and $O(n)$ space.*

This algorithm is inefficient in an external-memory setting because it may elicit one I/O per each tree-edge traversal. Nevertheless, as we observed before, of the distribution of the lcps is skewed towards small values, then this construction might be I/O-efficient in that the top part of the suffix tree could be cached in the internal memory, and thus do not elicit any I/Os during the scanning and re-scanning steps. We refer the reader to [3] for details on this issue.

10.4 Some interesting problems

10.4.1 Approximate pattern matching

The problem of approximate pattern matching can be formulated as: *finding all substrings of a text* $T[1, n]$ *that match a pattern* $P[1, p]$ *with at most k errors*. In this section we restrict our discussion to the simplest type of errors, the ones called *mismatches* or *substitutions* (see Figure 10.11). This way the text substrings which "k-mismatch" the searched pattern P have length p and coincide with the pattern in all but at most k characters. The following figure provides an example by considering two DNA strings formed over the alphabet of four nucleotide bases $\{A, T, G, C\}$. The reason for this kind of strings is that Bio-informatics is the context which spurred interest around the approximate pattern-matching problem.

```
C  C  G  T  A  C  G  A  T  C  A  G  T  A
C  C  G  A  A  C  T
         ⇑        ⇑
```

FIGURE 10.11: An example of matching between T (top) and P (bottom) with $k = 2$ mismatches.

The naïve solution to this problem consists of trying to match P with every possible substring of T, having length p, counting the mismatches and returning the positions were their number is at most k. This would take $O(pn)$ time, independently of k. The inefficiency comes from the fact that each pattern-substring comparison starts from the beginning of P, thus taking $O(p)$ time. In what follows we describe a sophisticated solution which hinges on an elegant data structure that solves an apparently un-related problem formulated over an array of integers, and called *Range Minimum Query* (shortly, RMQ). This data structure is the backbone of many other algorithmic solutions in problems arising in Data Mining, Information Retrieval, and so on.

The following Algorithm 10.4 solves the k-mismatches problem in $O(nk)$ time by making the following basic observation. If P occurs in T with $j \leq k$ mismatches, then we can align the pattern P with a substring of T so that j or $j - 1$ substrings coincide and j characters mismatch. Actually equal substrings and mismatches interleave each other. As an example consider again Figure 10.11, the pattern occurs at position 1 in the text T with 2 mismatches, and in fact two substrings of P match their corresponding substrings of T. This means that if we could compare pattern and text substrings for equality in constant time, then we could execute the naïve-approach taking $O(nk)$ time, instead of $O(np)$ time. To be operational, this observation can be rephrased as follows: if $T[i, i + \ell] = P[j, j + \ell]$ is one of these matching substrings, then ℓ is the longest common prefix between the pattern and the text suffixes starting at the matching positions i and j. Algorithm 10.4 deploys this rephrasing to code a solution which takes $O(nk)$ time provided that `lcp`-computations take $O(1)$ time.

If we run the Algorithm 10.4 over the strings showed in Figure 10.11, we perform two `lcp`-computations and find that P occurs at text position 1 with 2-mismatches:

- `lcp`($T[1, 14], P[1, 7]$) = `lcp(CCGTACGATCAGTA, CCGTACG)` = `CCG`.
- `lcp`($T[5, 14], P[5, 7]$) = `lcp(ACGATCAGTA, ACG)` = `AC`.

How do we compute `lcp`($T[i + j - 1, n], P[j, p]$) in constant time? We know that suffix trees and suffix arrays have built-in some `lcp`-information, but we similarly recall that these data structures were built on one single string, namely the text T. Here we are talking of suffixes of P and T together. Nonetheless we can easily circumvent this difficulty by constructing the suffix array, or

Algorithm 10.4 Approximate-pattern matching based on LCP-computations

```
matches = {}
for $i = 1$ to $n$ do
    $m = 0, j = 1$;
    while $m \leq k$ and $j \leq p$ do
        $\ell = \mathtt{lcp}(T[i + j - 1, n], P[j, p]$;
        $j = j + \ell$;
        if $j \leq p$ then
            $m = m + 1$; $j = j + 1$;
        end if
    end while
    $j = 1$;
    if $m \leq k$ then
        $matches = matches \cup \{T[i, i + p - 1]\}$;
    end if
end for
return matches;
```

the suffix tree, over the string $X = T\#P$, where # is a new character not occurring elsewhere. This way each computation of the form $\mathtt{lcp}(T[i + j - 1, n], P[j, p])$ can now be turned into an lcp-computation between suffixes of X, precisely $\mathtt{lcp}(T[i + j - 1, n], P[n + 1 + j, n + 1 + p])$. We are therefore left with showing how these lcp-computations can be performed in constant time, whichever is the pair of compared suffixes. This is the topic of the next subsection.

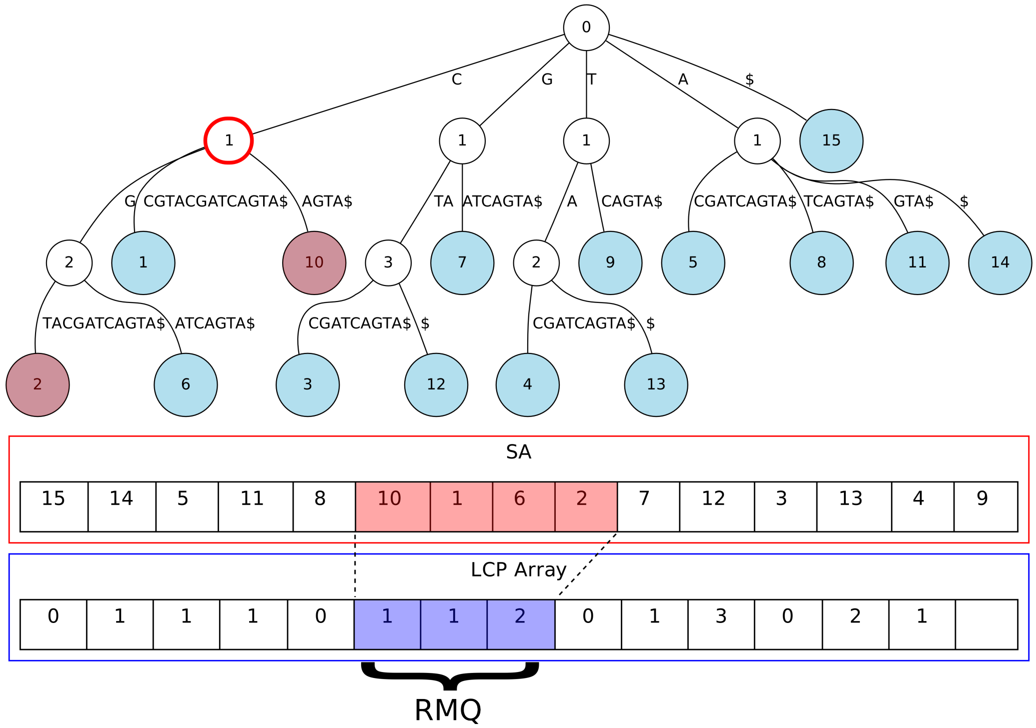
Lowest Common Ancestor, Range Minimum Query and Cartesian Tree

Let us start from an example, by considering the suffix tree ST_X and the suffix array SA_X built on the string $X = CCGTACGATCAGTA$. This string is not in the form $X = T\#P$ because we wish to stress the fact that the considerations and the algorithmic solutions proposed in this section apply to any string X, not necessarily the ones arising from the Approximate Pattern-Matching problem.

The key observation, whose correctness spurs immediately from Figure 10.12, is that there is a strong relation between the lcp-problem over X's suffixes and the computation of *lowest common ancestors* (lca) in the suffix tree ST_X. Consider the problem of finding the longest common prefix between suffixes $X[i, x]$ and $X[j, x]$) where $x = |X|$. It is not difficult to convince yourself that the node $u = \mathtt{lca}(X[i, x], X[j, x])$ in the suffix tree ST_X spells out their lcp, and thus the value $|s[u]|$ stored in node u is exactly the lcp-value we are searching for. Notice that this property holds independently of the lexicographic sortedness of the edge labels, and thus of the suffix tree leaves.

Equivalently, the same value can be derived by looking at the suffix array SA_X. In particular take the lexicographic positions i_p and j_p where those two suffixes occur in SA_X, say $SA_X[i_p] = i$ and $SA_X[j_p] = j$ (we are assuming for simplicity that $i_p < j_p$). It is not difficult to convince yourself that the *minimum value* in the sub-array $\mathtt{lcp}[i_p, j - 1]$[5] is exactly equal to $|s[u]|$ since the values contained in that sub-array are the values stored in the suffix-tree nodes of the subtree that descends from u. Actually the order of these values is the one given by the in-order visit of u's descendants. Anyway, this order is not important for our computation which actually takes the smallest value, because it is interested in the shallowest node (namely the root u) of that subtree.

[5] Recall that $\mathtt{lcp}[q]$ stores the length of the longest common prefix between suffix $SA[i]$ and its next suffix $SA[i + 1]$.



SA

15	14	5	11	8	10	1	6	2	7	12	3	13	4	9

LCP Array

0	1	1	1	0	1	1	2	0	1	3	0	2	1	

FIGURE 10.12: An example of suffix tree (unordered), suffix array, `lcp`-array for the string X = `CCGTACGATCAGTA`. The figure highlights that the computation of $\texttt{lcp}(X[2,16], X[10,16])$ boils down to finding the depth of the `lca`-node in ST_X between the leaf 2 and the leaf 10, as well as to solve a range minimum query on the sub-array $\texttt{lcp}[6,8]$ since $SA_X[6] = 10$ and $SA_X[9] = 2$.

Figure 10.12 provides a running example which clearly shows these two strong properties, which actually do not depend on the order of the children of suffix-tree nodes. As a result, we have two approaches to compute `lcp`in constant time, either through `lca`-computations over ST_X or through `RMQ`-computations over $\texttt{lcp}_X$. For the sake of presentation we introduce an elegant solution for the latter, which actually induces in turn an elegant solution for the former, given that their are strongly related.

In general terms the `RMQ` problem can be stated as follows:

The range-minimum-query problem. *Given an array $A[1,n]$ of elements drawn from an ordered universe, build a data structure $\texttt{RMQ}_A$ that is able to compute efficiently the position of a smallest element in $A[i,j]$, for any given queried range (i,j). We say ”a smallest” because the array may contain many minimum elements.*

We underline that this problem asks for the *position* of a minimum element in the queried sub-array, rather than its value. This is more general because the value of the minimum can be obviously retrieved from its position in A by accessing this array, which is available.

In this lecture we aim for constant-time queries [1]. The simplest solution achieves this goal via a table that stores the index of a minimum entry for each possible range (i,j). Such table requires $O(n^2)$ space and $O(n^2)$ time to be built. A better solution hinges on the following observation: any range (i,j) can be decomposed into two (possibly overlapping) ranges whose size is a power of two, namely $(i, i+2^L)$ and $(j-2^L, j)$ where $L = \lfloor \log(j-i+1) \rfloor$. This allows us to *sparsify* the previous quadratic-sized table by storing only ranges whose size is a power of two. This way, for each position i we store the answers to the queries $\mathtt{RMQ}_A(i, i + 2^L)$, thus occupying a total space of $O(n \log n)$ without impairing the time complexity of the query which is still constant and corresponds to return $\mathtt{RMQ}_A(i, j) = \mathtt{argmin}_{i,j}\{\mathtt{RMQ}_A(i, i + 2^L), \mathtt{RMQ}_A(j - 2^L, j)\}$.

In order to get the optimal $O(n)$ space occupancy, we need to dig into the structure of the RMQ-problem and make a twofold reduction which goes back-and-forth from RMQ-computations to lca-computations: namely, we reduce (1) the RMQ-computation over the lcp-array to an lca-computation over Cartesian Trees (that we define next); we then reduce (2) the lca-computation over Cartesian Trees to an RMQ-computation over a binary array. This last problem will then be solved in $O(n)$ space and constant query time. Clearly reduction (2) can be applied to any tree, and thus can be applied to Suffix Trees in order to solve lca-queries over them.

First reduction step: from RMQ to lca. We transform the $\mathtt{RMQ}_A$-problem "back" into an lca-problem over a special tree which is known as *Cartesian Tree* and is built over the entries of the array $A[1, n]$. The *Cartesian Tree* C_A is a binary tree of n nodes, each labeled with one of A's entries (i.e. value and position in A). The labeling is defined recursively as follows: the root of C_A is labeled by the minimum entry in $A[1, n]$, say this is $\langle A[m], m\rangle$. Then the left subtree of the root is recursively defined as the Cartesian Tree of the subarray $A[1, m - 1]$, and the right subtree is recursively defined as the Cartesian Tree of the subarray $A[m + 1, n]$. Tree C_A can be constructed in $O(n)$ time as follows: Suppose that we have already built the tree C_i for the array $A[1, i]$, then we insert the element $A[i + 1]$ in two steps (see Figure 10.13):

1. we climb the rightmost path of C_i and determine the first node u whose associated entry $A[j]$ is smaller than $A[i + 1]$;
2. we make the node corresponding to $A[i + 1]$ the right son of u, and turn the previous right subtree of u into the left subtree of $A[i + 1]$.

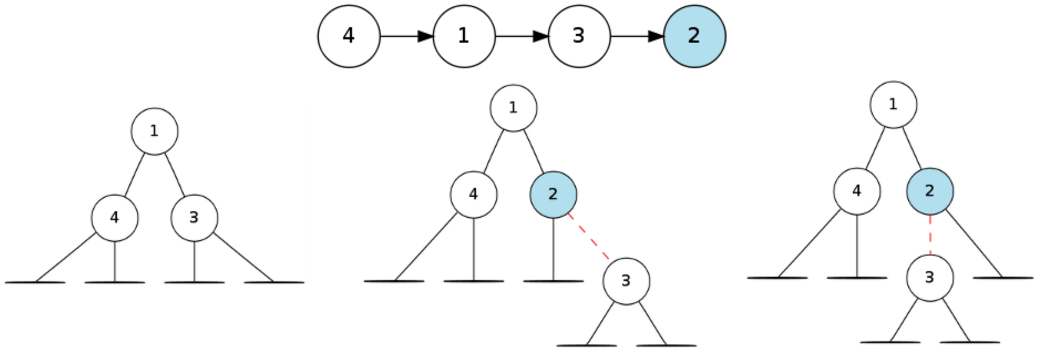
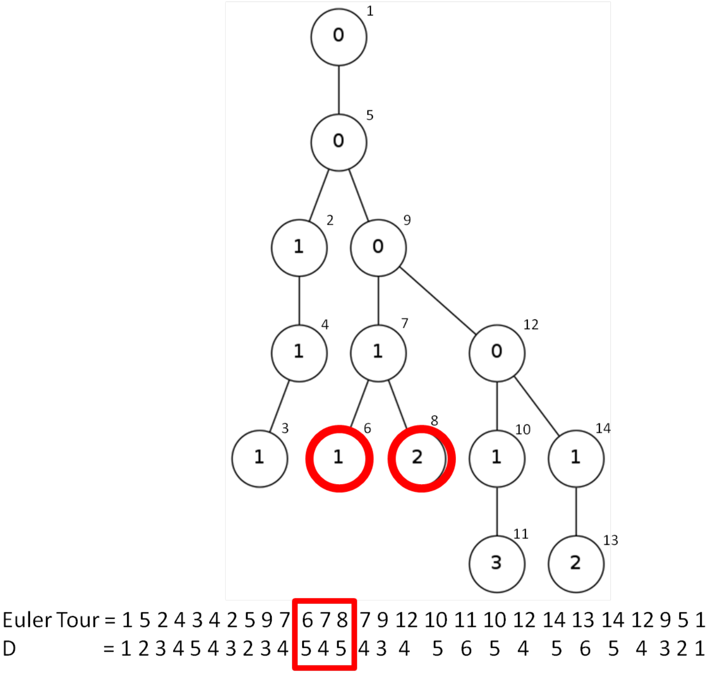


FIGURE 10.13: Example of Cartesian Tree built over the array $A[1, 3] = \{1, 4, 3\}$, and the insertion of the value 2. Observe that nodes of C_A store only A's values, their positions in A are dropped to ease the presentation.

The following Figure 10.14 shows the Cartesian tree built on the lcp-array depicted in Figure 10.12. Given the construction process, we can state that ranges in the lcp-array correspond to subtrees of the Cartesian tree. Therefore computing $\mathtt{RMQ}_A(i, j)$ boils down to compute an lca-query between the nodes of C_A associated to the entries i and j. Differently of what occurred for lca-queries on ST_X, where the arguments were leaves of that suffix tree, the queried nodes in the Cartesian Tree may be internal nodes, and actually it might occur that one node is ancestor of the other node. For example, executing $\mathtt{RMQ}_{\mathtt{lcp}}(6,8)$ equals to executing $\mathtt{lca}(6,8)$ over the Cartesian Tree $C_{\mathtt{lcp}}$ of Figure 10.14. The result of this query is the node $\langle \mathtt{lcp}[7], 7\rangle = \langle 1, 7\rangle$. Notice that we have another minimum value in $\mathtt{lcp}[6,8]$ at $\mathtt{lcp}[6] = 1$; the answer provided by the lca is one of the existing minima in the queried-range.



Euler Tour = 1 5 2 4 3 4 2 5 9 7 6 7 8 7 9 12 10 11 10 12 14 13 14 12 9 5 1
D = 1 2 3 4 5 4 3 2 3 4 5 4 5 4 3 4 5 6 5 4 5 6 5 4 3 2 1

FIGURE 10.14: Cartesian tree built on the lcp-array of Figure 10.12. On the bottom part are reported the Euler Tour of the Cartesian Tree and the array D of the depths of the nodes according to the Euler-Tour order.

Second reduction step: from lca to RMQ. We transform the lca-problem over the Cartesian Tree $C_{\mathtt{lcp}}$ "back" into an RMQ-problem over a special binary array $\Delta[1, 2e]$, where e is the number of edges in the Cartesian Tree (of course $e = O(n)$). It seems strange this "circular" sequence of reductions that now has turned us back into an RMQ-problem. But the current RMQ-problem, unlike the original one, is formulated on a binary array and thus admits an optimal solution in $O(n)$ space.

To build the binary array $\Delta[1, 2e]$ we need first to build the array $D[1, 2e]$ which is obtained as follows. Take the *Euler Tour* of Cartesian Tree C_A, visiting the tree in pre-order and writing down each node everytime the visit passes through it. A node can be visited multiple times, precisely it is visited/written as many times as its number of incident edges; except for the root which is written the number of incident edges plus 1.

Given the Euler Tour of the Cartesian Tree C_A, we build the array $D[1, 2e]$ which stores the depths of the visited nodes in the order stated by the Euler Tour (see Figure 10.14). Given D and the way the Euler Tour is built, we can conclude that query $\mathtt{lca}(i, j)$ in C_A boils down to compute the node of minimum depth in the sub-array $D[i', j']$ where i' (resp. j') is the position of the first (resp. last) occurrence of the node i (resp. j) in the Euler Tour. In fact, the range $D[i', j']$ corresponds to the part of the Euler Tour that starts at node i and ends at node j. The node of minimum depth encountered in this Euler sub-Tour is properly the $\mathtt{lca}(i, j)$.

So we reduced an `lca`-query over the Cartesian Tree into an `RMQ`-query over node depths. In our running example on Figure 10.14 this reduction transforms the query $\mathtt{lca}(6, 8)$ into a query $\mathtt{RMQ}_D(11, 13)$, which is highlighted by a red rectangle. Turning nodes into ranges can be done in constant time by simply storing two integers per node of the Cartesian Tree, denoting their first/last occurrence in the Euler Tour, thus taking $O(n)$ space.

We are again "back" to an `RMQ`-query over an integer array. But the current array D is special because its consecutive entries differ by 1 given that they refer to the depths of consecutive nodes in an Euler Tour. And in fact, two consecutive nodes in the Euler Tour are connected by an edge and thus one node is the parent of the other, and hence their depths differ by one unit. The net result of this property is that we can solve the `RMQ`-problem over $D[1, 2e]$ in $O(n)$ space and $O(1)$ time as follows. (Recall that $e = O(n)$.) Solution is based on two data structures which are detailed next.

First, we split the array D into $\frac{2e}{d}$ subarrays D_k of size $d = \frac{1}{2}\log e$ each. Next, we find the minimum element in each subarray D_k, and store its position at the entry $M[k]$ of a new array whose size is therefore $\frac{2e}{d}$. We finally build on the array M the sparse-table solution indicated above which takes superlinear space (in the size of M) and solves `RMQ`-queries in constant time. The key point here is that M's size is sublinear in e, and thus in n, so that the overall space taken by array M and its sparse-table is $O((\frac{e}{\log e}) * \log \frac{e}{\log e}) = O(e) = O(n)$.

The second data structure is built to efficiently answer `RMQ`-queries in which i and j are in the same block D_k. It is clear that we cannot tabulate all answers to all such possible pairs of indexes because this would end up in $\Theta(n^2)$ space occupancy. So the solution we describe here spurs from two simple, deep observations whose proof is immediate and left to the reader:

Binary entries: Every block D_k can be transformed into a pair that consists of its first element $D_k[1]$ and a binary array $\Delta_k[i] = D_k[i] - D_k[i-1]$ for $i = 2, \ldots, d$. Entries of Δ_k are either -1 or $+1$ because of the unit difference between adjacent entries of D.

Minimum location: The position of the minimum value in D_k depends only on the content of the binary sequence Δ_k and does not depend on the starting value $D_k[1]$.

Nicely, the possible configurations that every block D_k can assume are infinite, given that infinite is the number of ways we can instantiate the input array A on which we want to issue the `RMQ`-queries; but the possible configurations of the image Δ_k is finite and equal to 2^d. This suggests to apply the so called *Four Russians trick* to the binary arrays by tabulating all possible binary sequences Δ_k and, for each of them, storing the position of the minimum value. Since the blocks Δ_k have length $d = \frac{\log e}{2}$, the total number of possible binary sequences is $2^d = O(2^{\frac{\log e}{2}}) = O(\sqrt{e}) = O(\sqrt{n})$. Moreover, since both query-indexes i and j can take at most $d = \frac{\log e}{2}$ possible values, being internal in a block D_k, we can have at most $O(\log^2 e)$ queries of this third type. Consequently, we build a lookup table $T[i_o, j_o, \Delta_k]$ that is indexed by the possible query-offsets i_o and j_o within the block D_k and its binary configuration Δ_k. Table T stores at that entry the position of the minimum value in D_k. We also assume that, for each k, we have stored Δ_k so that the binary representation Δ_k of D_k can be retrieved in constant time. Each of these indexing parameters takes $O(\log e) = O(\log n)$ bits of space, hence one memory word, and thus can be managed in $O(1)$ time and space. In summary, the whole table T consists of $O(\sqrt{n}(\log n)^2) = o(n)$ entries. The time needed to build T is $O(n)$. The power of transforming D_k into Δ_k is evident now, every entry of $T[i_o, j_o, \Delta_k]$ is actually encoding the answer for an infinite number of blocks D_k, namely the ones that can be turned to the same binary configuration Δ_k.

At this point we are ready to design an algorithm that, using the three data structures illustrated above, answers a query $\mathtt{RMQ}_D(i, j)$ in constant time. If i, j are inside the same block D_k then the answer is retrieved in two steps: first we compute the offsets i_o and j_o with respect to the beginning of D_k and determine the binary configuration Δ_k from k; then we use this triple to access the proper entry of T. Otherwise the range (i, j) spans at least two blocks and can thus be decomposed in three parts: a suffix of some block $D_{i'}$, a consecutive sequence of blocks $D_{i'+1} \cdots D_{j'-1}$, and finally the prefix of block $D_{j'}$. The minimum for the suffix of $D_{i'}$ and the prefix of $D_{j'}$ can be retrieved from T, given that these ranges are inside two blocks. The minimum of the range spanned by $D_{i'+1} \cdots D_{j'-1}$ is stored in M. All this information can be accessed in constant time and the final minimum-position can be retrieved by comparing these three minimum values, in constant time too.

THEOREM 10.11 *Range-minimum queries over an array $A[1, n]$ of elements drawn from an ordered universe can be answered in constant time using a data structure that occupies $O(n)$ space.*

Given the stream of reductions we illustrated above, we can conclude that Theorem 10.11 applies also to computing `lca` in generic trees: it is enough to take the input tree in place of the Cartesian Tree.

THEOREM 10.12 *Lowest-common-ancestor queries over a generic tree of size n can be answered in constant time using a data structure that occupies $O(n)$ space.*

10.4.2 Text Compression

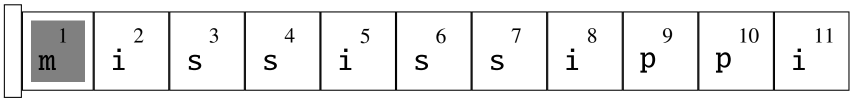
Data compression will be the topic of one of the following chapters; nonetheless in this section we address the problem of compressing a text via the simple algorithm which is at the core of the well known `gzip` compressor, named *LZ77* from the initials of its inventors (Abraham Lempel and Jacob Ziv [9]) and from the year of its publication (1977). We will show that there exists an optimal implementation of the LZ77-algorithm taking $O(n)$ time and using suffix trees.

Given a text string $T[1, n]$, the algorithm LZ77 produces a parsing of T into substrings that are defined as follows. Assume that it has already parsed the prefix $T[1, i - 1]$ (at the beginning this prefix is empty), then it decomposes the remaining text suffix $T[i, n]$ in three parts: the longest substring $T[i, i + \ell - 1]$ which starts at i and repeats before in the text T, the next character $T[i + \ell]$, and the remaining suffix $T[i + \ell + 1, n]$. The next substring to add to the parsing of T is $T[i, i + \ell]$, and thus corresponds to the shortest string that is *new* in $T[1, i - 1]$. Parsing then continues onto the remaining suffix $T[i + \ell + 1, n]$, if any.

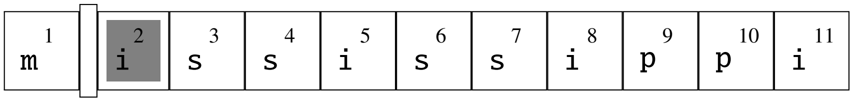
Compression is obtained by succinctly encoding the triple of integers $\langle d, \ell, T[i + \ell]\rangle$, where d is the distance (in characters) from i to the previous copy of $T[i, i + \ell - 1]$; ℓ is the length of the copied string; $T[i + \ell]$ is the appended character. By saying "previous copy" of $T[i, i + \ell - 1]$, we mean that its copy starts before position i but it might extend after this position, hence it could be $d < \ell$; furthermore, the previous copy can be any previous occurrence of $T[i, i + \ell - 1]$, although space-efficiency issues suggest us to take the closest copy (and thus the smallest d). Finally we observe that the reason for adding the character $T[i + \ell]$ to the emitted triple is that this character behaves like an *escape*-mechanism; in fact it is useful when no-copy is possible and thus $\ell = 0$ (this occurs when a new character is met in T).[6]

Before digging into an efficient implementation of the LZ77-algorithm let us consider our example string $T =$ `mississippi`. Its LZ77-parsing is computed as follows:

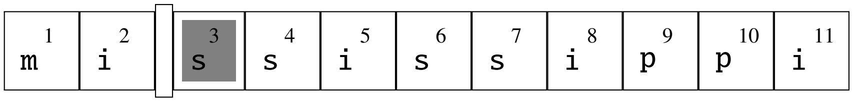
[6] We are not going to discuss the integer-encoding issue, since it will be the topic of a next chapter, we just mention here that efficiency is obtained in `gzip` by taking the rightmost copy and by encoding the values d and ℓ via a Huffman coder.



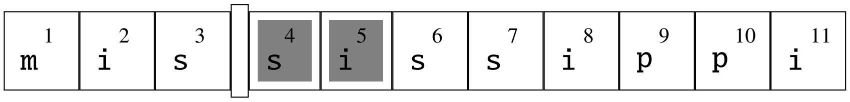
Output: $< 0, 0, m >$



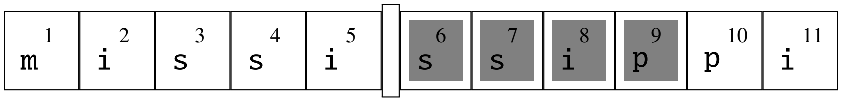
Output: $< 0, 0, i >$



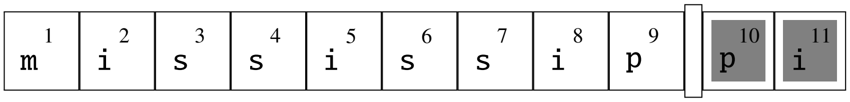
Output: $< 0, 0, s >$



Output: $< 1, 1, i >$



Output: $< 3, 3, p >$



Output: $< 1, 1, i >$

We can compute the LZ77-parsing in $O(n)$ time via an elegant algorithm that deploys the suffix tree ST. The difficulty is to find π_i, the longest substring that occurs at position i and repeats before in the text T. Say d is the distance of the previous occurrence. Given our notation above we have that $\ell = |\pi_i|$. Of course π_i is a prefix of $suff_i$ and a prefix of $suff_{i-d}$; actually, it is the longest common prefix of these two suffixes, and by maximality, there is no other previous suffix $suff_j$ (with $j < i$) that shares a longer prefix with $suff_i$. By properties of suffix trees, the lowest-common-ancestor of leaves i and j spells out π_i. However we cannot compute $\texttt{lca}(i, j)$ by issuing a query to the data structure of Theorem 10.12 because we do not know j, which is exactly the information we wish to compute. Similarly we cannot trace a downward path from the root of ST trying to match $suff_i$ because all suffixes of T are indexed in the suffix tree and thus we could detect a longer copy which follows position i, instead of preceding it.

To circumvent these problems we preprocess ST via a post-order visit that computes for every internal node u its minimum leaf $\min(u)$. Clearly $\min(u)$ is the leftmost position from which we can copy the substring $s[u]$. Given this information we can determine easily π_i, just trace a downward path from the root of ST scanning $suff_i$ and stopping as soon as the reached node v is such that $\min(v) = i$. At this point we take u as the parent of v and set $\pi_i = s[u]$, and $d = i - \min(u)$. Clearly, the chosen copy of π_i is the farthest one and not the closest one: this does not impact in the number of phrases in which T is parsed by LZ77, but possibly influences the magnitude of these distances and thus their succinct encoding. Devising an LZ77-parser that efficiently determines the closest copy of each π_i is non trivial and needs much more sophisticated data structures.

Take $T =$ `mississippi` as the string to be parsed (see above) and consider its suffix tree ST in Figure 10.8. Assume that the parsing is at the suffix $suff_2 =$ `ississippi`. Its tracing down ST stops immediately at the root of the suffix tree because the node to be visited next would be u, for which $\min(u) = 2$ which is not smaller than the current suffix position. Then consider the parsing at suffix $suff_6 =$ `ssippi`. We trace down ST and actually exhaust $suff_6$, so reaching the leaf 6, for which min is 3. So the selected node is its parent z, for which $s[z] =$ `ssi`. The emitted triple is correctly $< 3, 3, p >$.

The time complexity of this implementation of the LZ77-algorithm is $O(n)$ because the traversal of the suffix tree advances over the string T, and this may occur only n times. Branching out of suffix-tree nodes can be implemented in $O(1)$ time via perfect hash tables, as observed for the substring-search problem. The construction of the suffix tree costs $O(n)$ time, by using one of the algorithms we described in the previous sections. The computation of the values $\min(u)$, over all nodes u, takes $O(n)$ time via a post-order visit of ST.

THEOREM 10.13 *The LZ77-parsing of a string $T[1, n]$ can be computed in $O(n)$ time and space. Each substring of the parsing is copied from its farthest previous occurrence.*

10.4.3 Text Mining

In this section we briefly survey two examples of uses of suffix arrays and `lcp`-arrays in the solution of sophisticated text mining problems.

Let us consider the following question: *Check whether there exists a substring of $T[1, n]$ that repeats at least twice and has length L.* Solving this problem in a brute-force way would mean to take every text substring of length L, and count its number of occurrences in T. These substrings are $\Theta(n)$, searching each of them takes $O(nL)$ time, hence the overall time complexity of this trivial algorithm would be $O(n^2L)$. A smarter and faster, actually optimal, solution comes from the use either of the suffix tree or of the lcp-array `lcp`, built on the input text Ts.

The use of suffix tree is simple. Let us assume that such a string does exist, and it occurs at positions x and y of T. Now take the two leaves in the suffix tree which correspond to $suff_x$ and $suff_y$ and compute their lowest common ancestor, say $a(x, y)$. Since $T[x, x + L - 1] = T[y, y + L - 1]$, it is that $|s[a(x, y)]| \geq L$. We write ”greater or equal” because it could be the case that a longer substring is shared at positions x and y, in fact L is just fixed by the problem. The net result of this argument is that it does exist an *internal* node in the suffix tree whose label is greater or equal than L; a visit of the suffix tree is enough to search for any node such this one, thus taking $O(n)$ time.

The use of the suffix array is a little bit more involved, but follows a similar argument. Recall that suffixes in SA are lexicographically ordered, so the longest common prefix shared by suffix $SA[i]$ is with its adjacent suffixes, namely either with suffix $SA[i - 1]$ or with suffix $SA[i + 1]$. The length of these lcps is stored in the entries `lcp`$[i - 1, i]$. Now, if the repeated substring of length L does exist, and it occurs e.g. at text positions x and y, then we have `lcp`$(T[x, n], T[y, n]) \geq L$. These two suffixes not necessarily are contiguous in SA (this is the case when the substring occurs more than twice), nonetheless all suffixes occurring among them in SA will surely share a prefix of length L, because of their lexicographic order. Hence, if suffix $T[x, n]$ occurs at position q of the suffix array, i.e. $SA[q] = x$, then we have that either `lcp`$[q - 1] \geq L$ or `lcp`$[q] \geq L$, depending on the fact that $T[y, n] < T[x, n]$ or vice versa, respectively. Hence we can solve the question stated above by scanning `lcp` and searching for an entry $\geq L$. This takes $O(n)$ optimal time.

Let us now ask a more sophisticated question: *Check whether there exists a text substring that repeats at least C times and has length L.* This is the typical query in a text mining scenario, where we are interested not just in a repetitive event but in an event occurring with some *statistical evidence.* We can again solve this problem by trying all possible substrings and counting their occurrences. Again, a faster solution comes from the use either of the suffix tree or of the array `lcp`. Following the argument provided in the solution of the previous question we note that, if a substring of length L occurs (at least) C times, then it does exist (at least) C text suffixes that share (at least) L characters. So it does exist a node u in the suffix tree such that $|s[u]| \geq L$ and the number of descending leaves $occ[u] \geq C$. Equivalently, it does exist a sub-array in `lcp` of length $\geq C - 1$ that consists of entries $\geq L$. Both approaches provide an answer to the above question in $O(n)$ time.

Let us conclude this section by asking a query closer to a search-engine scenario: *Given two patterns P and Q, and a positive integer k, check whether there exists an occurrence of P whose distance from an occurrence of Q in an input text T is at most k.* This is also called *proximity search* over a text T which is given in advance to be preprocessed. The solution passes through the use of any search data structure, being it a suffix tree or a suffix array built over T, plus some sorting/merging steps. We search for P and Q in T and derive their occurrences, say O. Both suffix arrays and suffix trees return these occurrences unsorted. Therefore we sort them, in order to produce the ordered list of occurrences of P and Q. At this point it is easy to determine whether the question above admits a positive answer; if it does, then there do exist two *consecutive* occurrences of P and Q whose distance is at most k. To detect this situation it is enough to scan the sorted Sequence O and check, for every *consecutive* pair of positions which are occurrences of P and Q, whether the difference is at most k. This takes overall $O(|P| + |Q| + |O| \log |O|)$ time, which is clearly advantageous whenever the set O of candidate occurrences is small, and thus the queries P and Q are sufficiently selective.

References

[1] Michael A. Bender and Martin Farach-Colton. The LCA Problem Revisited. In *Procs of the Latin American Symposium on Theoretical Informatics (LATIN)*, 88-94, 2000.

[2] Martin Farach-Colton, Paolo Ferragina, S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6): 987-1011, 2000.

[3] Paolo Ferragina. String search in external memory: algorithms and data structures. *Handbook of Computational Molecular Biology*, edited by Srinivas Aluru. Chapman & Hall/CRC Computer and Information Science Series, chapter 35, Dicembre 2005.

[4] Paolo Ferragina and Travis Gagie and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica: Special issue on selected papers of LATIN 2010*, 63(3): 707-730, 2012.

[5] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* University Press, 1997.

[6] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: PAT trees and PAT arrays. In B. Frakes and R. A. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 5, pages 66–82, Prentice-Hall, 1992.

[7] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Procs of the International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science vol. 2791, Springer, 943–955, 2003.

[8] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Procs of the Symposium on Combinatorial Pattern Matching (CPM)*, Lecture Notes in Computer Science vol. 2089, Springer, 181–192, 2001.

[9] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3): 337-343, 1977.

[10] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[11] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2): 262-272, 1976.

11

Integer Coding

Everything should be made as simple as possible, but no simpler
Albert Einstein

In this chapter we will address a basic encoding problem which occurs in many contexts, and whose efficient dealing is frequently underestimated for the impact it may have on the total space occupancy and speed of the underlying application [2, 8].

Problem. *Let $S = s_1, \ldots, s_n$ be a sequence of positive integers s_i, possibly repeated. The goal is to represent the integers of S as binary sequences which are self-delimiting and use few bits.*

We note that the request about s_i of being *positive integers* can be relaxed by mapping a positive integer x to $2x$ and a negative integer x to $-2x+1$, thus turning again the set S to a set of just positive integers.

Let us comment two exemplar applications. Search engines store for each term t the list of documents (i.e. Web pages, blog posts, tweets, etc. etc.) where t occurs. Answering a user query, formulated as sequence of keywords $t_1 t_2 \ldots t_k$, then consists of finding the documents where all t_is occur. This is implemented by intersecting the document lists for these k terms. Documents are usually represented via integer IDs, which are assigned during the crawling of those documents from the Web. Storing these integers with a fixed-length binary encoding (i.e. 4 or 8 bytes) may require considerable space, and thus time for their retrieval, given that modern search engines index up to 20 billion documents. In order to reduce disk-space occupancy, as well as increase the amount of cached lists in internal memory, two kinds of compression tricks are adopted: the first one consists of sorting the document IDs in each list, and then encode each of them with the difference between it and its preceding ID in the list, the so called *d-gap*[1]; the second trick consists of encoding each d-gap with a variable-length sequence of bits which is short for small integers.

Another example of occurrence for the above problem relates to data compression. We have seen in Chapter 10 that the LZ77-compressor turns input files into sequence of triples in which the first two components are integers. Other known compressors (such as MTF, MPEG, RLE, BWT, etc.)

[1] Of course, the first document ID of a list is stored explicitly.

produce as intermediate output one or more sets of integers, with smaller values most probable and larger values increasingly less probable. The final coding stage of those compressors must therefore convert these integers into a bit stream, such that the total number of bits is minimized.

The main question we address in this chapter is how we design a variable-length binary representation for (unbounded) integers which takes as few bit as possible and is prefix-free, namely the encoding of s_is can be concatenated to produce an output bit stream, which preserves decodability, in the sense that each individual integer encoding can be identified and decoded.

The first and simplest idea to solve this problem is surely that one to take $m = \max_j s_j$ and then encode each integer $s_i \in S$ by using $1 + \lfloor \log_2 m \rfloor$ bits. This fixed-size encoding is efficient whenever the set S is not much spread and concentrated around the value zero. But this is a very unusual situation, in general, $m \gg s_i$ so that many bits are wasted in the output bit stream. So why not storing each s_i by using its binary encoding with $1 + \lfloor \log_2 s_i \rfloor$ bits. The subtle problem with this approach would be that this code is not self-delimiting, and in fact we cannot concatenate the binary encoding of all s_i and still be able to distinguish each codeword. As an example, take $S = \{1, 2, 3\}$ and the output bit sequence `11011` which would be produced by using their binary encoding. It is evident that we could derive many compatible sequence of integers from `11011`, such as S, but also $\{6, 1, 1\}$, as well as $\{1, 2, 1, 1\}$, and several others.

It is therefore clear that this simple encoding problem is challenging and deserves the attention that we dedicate in this chapter. We start by introducing one of the simplest integer codes known, the so called *unary code*. The unary code $U(x)$ for an integer $x \geq 1$ is given by a sequence of $x - 1$ bits set to 0, ended by a (delimiting) bit set to 1. The correctness of the condition that $x \neq 0$ is easily established. $U(x)$ requires x bits, which is *exponentially longer* than the length $\Theta(\log x)$ of its binary code, nonetheless this code is efficient for very small integers and soon becomes *space inefficient* as x increases.

This statement can be made more precise by recalling a basic fact coming from the Shannon's coding theory, which states that *the ideal code length $L(c)$ for a symbol c is equal to* $\log_2 \frac{1}{Pr[c]}$ *bits, where $P[c]$ is the probability of occurrence of symbol c.* This probability can be known in advance, if we have information about the source emitting c, or it can be estimated empirically by examining the occurrences of integers s_i in S. The reader should be careful in recalling that, in the scenario considered in this chapter, symbols are positive integers so the ideal code for the integer x consists of $\log_2 \frac{1}{Pr[x]}$ bits. So, by solving the equation $|U(x)| = \log_2 \frac{1}{Pr[x]}$ with respect to $P[x]$, we derive the distribution of the s_is for which the unary code is optimal. In this specific case it is $P[x] = 2^{-x}$. As far as efficiency is concerned, the unary code needs a lot of bit shifts which are slow to be implemented in modern PCs; again another reason to favor small integers.

FACT 11.1 *The unary code of a positive integer x takes x bits, and thus it is optimal for the distribution* $P[x] = 2^{-x}$.

Using this same argument we can also deduct that the fixed-length binary encoding, which uses $1 + \lfloor \log_2 m \rfloor$ bits, is optimal whenever integers in S are *distributed uniformly* within the range $\{1, 2, \ldots, m\}$.

FACT 11.2 *Given a set S of integers, of maximum value m, the fixed-length binary code represents each of them in* $1 + \lfloor \log_2 m \rfloor$ *bits, and thus it is optimal for the uniform distribution* $P[x] = 1/m$.

In general integers are not uniformly distributed, and in fact variable-length binary representations must be considered which eventually improve the simple unary code. There are many proposals in the literature, each offering a different *trade-off between space occupancy of the binary-code and* *time efficiency for its decoding.* The following subsections will detail the most useful and the most used among these codes, starting from the most simplest ones which use *fixed encoding models* for the integers (such as, e.g., γ and δ codes) and, then, moving to the more involved Huffman and Interpolative codes which use *dynamic* models that adapt themselves to the distribution of the integers in S. It is very well known that Huffman coding is *optimal*, but few times this optimality is dissected and made clear. In fact, this is crucial to explain some apparent contradictory statements about these more involved codes: such as the fact that in some cases Interpolative coding is better than Huffman coding. The reason is that Huffman coding is optimal among the family of *static* prefix-free codes, namely the ones that use a fixed model for encoding each single integer of S (specifically, the Huffman code of an integer x is defined according to $P[x]$). Vice versa, Interpolative coding uses a dynamic model that encodes x according to the distribution of other integers in S, thus possibly adopting different codes for the occurrences of x. Depending on the distribution of the integers in S, this adaptivity might be useful and thus originate a shorter output bit stream.

11.1 Elias codes: γ and δ

These are two very simple *universal* codes for integers which use a fixed model, they have been introduced in the '60s by Elias [3]. The adjective "universal" here relates to the property that the length of the code is $O(\log x)$ for any integer x. So it is just a *constant factor* more than the optimal binary code $B(x)$ having length $1 + \lfloor \log x \rfloor$, with the additional wishful property of being prefix-free.

γ-code represents the integer x as a binary sequence composed of two parts: a sequence of $|B(x)| - 1$ zero, followed by the binary representation $B(x)$. The initial sequence of zeros is delimited by the 1 which starts the binary representation $B(x)$. So $\gamma(x)$ can be decoded easily: count the consecutive number of zeros up to the first 1, say they are c; then, fetch the following $c + 1$ bits (included the 1), and interpret the sequence as the integer x.

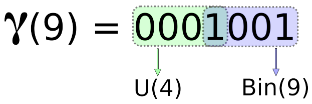


FIGURE 11.1: Representation for $\gamma(9)$.

The γ-code requires $2|B(x)| - 1$ bits, which is $2(1 + \lfloor \log_2 x \rfloor) - 1 = 2\lfloor \log_2 x \rfloor + 1$. In fact, the γ-code of the integer 9 needs $2\lfloor \log_2 9 \rfloor + 1 = 7$ bits. From Shannon's condition on ideal codes, we derive that the γ-code is optimal whenever the distribution of the values follows the formula $Pr[x] \approx \frac{1}{2x^2}$.

FACT 11.3 *The γ-code of a positive integer x takes $2\lfloor \log_2 x \rfloor + 1$ bits, and thus it is optimal for the distribution $P[x] \approx \frac{1}{2x^2}$, and it is a factor of 2 from the length of the optimal binary code.*

The inefficiency in the γ-code resides in the unary coding of the length $|B(x)|$ which is really costly as x becomes larger and larger. In order to mitigate this problem, Elias introduced the δ-code, which applies the γ-code in place of the unary code. So $\delta(x)$ consists of two parts: the first encodes $\gamma(|B(x)|)$, the second encodes $B(x)$. Notice that, since we are using the γ-code for $B(x)$'s length, the first and the second parts do not share any bits; moreover we observe that γ is applied to $|B(x)|$ which guarantees to be a number greater than zero. The decoding of $\delta(x)$ is easy, first we decode $\gamma(|B(x)|)$ and then fetch $B(x)$, so getting the value x in binary.

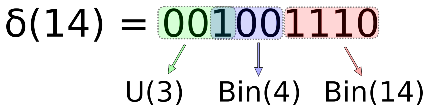


FIGURE 11.2: Representation for $\delta(14)$.

As far as the length in bits of $\delta(x)$ is concerned, we observe that it is $(1 + 2\lfloor\log_2 |B(x)|\rfloor) + |B(x)| \approx 1 + \log x + 2\log\log x$. This encoding is therefore a factor $1 + o(1)$ from the optimal binary code, hence it is universal.

FACT 11.4 *The δ-code of a positive integer x takes about $1 + \log_2 x + 2\log_2\log_2 x$ bits, and thus it is optimal for the distribution $P[x] \approx \frac{1}{2x(\log x)^2}$, and it is a factor of $1 + o(1)$ from the length of the optimal binary code.*

In conclusion, γ- and δ-codes are universal and pretty efficient whenever the set S is concentrated around zero; however, it must be noted that these two codes need a lot of *bit shifts* to be decoded and this may be slow if numbers are larger and thus encoded in many bits. The following codes trade space efficiency for decoding speed and, in fact, they are preferred in practical applications.

11.2 Rice code

There are situations in which integers are concentrated around some value, different from zero; here, Rice coding becomes advantageous both in compression ratio and decoding speed. Its special feature is to be a *parametric code*, namely one which depends from a positive integer k, which may be fixed according to the distribution of the integers in the set S. The Rice code $R_k(x)$ of an integer x, given the parameter k, consists of two parts: the quotient $q = \lfloor\frac{(x-1)}{2^k}\rfloor$ and the remainder $r = x - 2^k q - 1$. The quotient is stored in unary using $q + 1$ bits (the +1 is needed because q may be 0), the remainder r is in the range $[0, 2^k)$ and thus it is stored in binary using k bits. So the quotient is encoded in variable length, whereas the remainder is encoded in fixed length. The closer 2^k is to the value of x, the shorter is the representation of q, and thus the faster is its decoding. For this reason, k is chosen in such a way that 2^k is concentrated around the mean of S's elements.

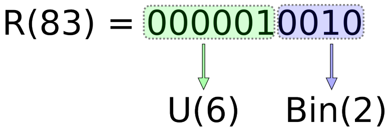


FIGURE 11.3: Representation for $R_4(83)$

The bit length of $R_k(x)$ is $q + k + 1$. This code is a particular case of the Golomb Code [8], it is optimal when the values to be encoded follow a geometric distribution with parameter p, namely $Pr[x] = (1 - p)^{x-1}p$. In this case, if $2^k \simeq \frac{ln(2)}{p} \simeq 0.69$ `mean`(S), the Rice and all Golomb codes generate an optimal prefix-code [8].

FACT 11.5 *The Rice code of a positive integer x takes $\lfloor\frac{(x-1)}{2^k}\rfloor + 1 + k$ bits, and it is optimal for the geometric distribution $Pr[x] = (1 - p)^{x-1}p$.*

11.3 PForDelta code

This method for compressing integers supports extremely fast decompression and achieves a small size in the compressed output whenever S's values follow a gaussian distribution. In detail, let us assume that most of S's values fall in an interval $[base, base + 2^b - 1]$, we translate the values in the new interval $[0, 2^b - 1]$ in order to encode them in b bits; the other values outside this range are called *exceptions* and they are represented in the compressed list with an *escape symbol* and also encoded explicitly in a separate list using a fixed-size representation of w bits (namely, a whole memory word). The good property of this code is that all values in S are encoded in fixed length, either b bits or $w + b$ bits, so that they can be decoded very fast and possibly in parallel by packing few of them in a memory word.

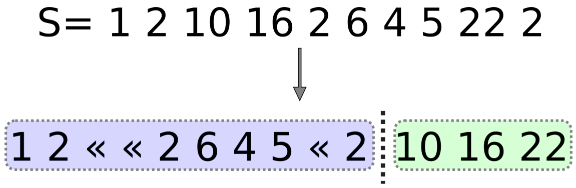


FIGURE 11.4: An example for PForDelta, with $b = 3$ and $base = 0$. The values in the range (blue box) are encoded using 3 bits, while the out-of-range values (green box) are encoded separately and an escape symbol $\ll$ is used as a place-holder.

FACT 11.6 *The PForDelta code of a positive integer x takes either b bits or $b+w$ bits, depending on the fact that $x \in [base, base + 2^b - 1]$ or not, respectively. This code is proper for a gaussian distribution of the integers to be encoded.*

The design of a PForDelta code needs to deal with two problems:

- *How to choose b:* in the original work, b was chosen such that about the 90% of the values in S are smaller than 2^b. An alternative solution is to trade between space wasting (choosing a greater b) or space saving (more exceptions, smaller b). In [7] it has been proposed a method based on dynamic programming, that computes the optimal b for a desired compression ratio. In particular, it returns the largest b that minimizes the number of exceptions and, thus, ensures a faster decompression.
- *How to encode the escape character:* a possible solution is to assign a special bit sequence for it, thus leaving $2^b - 1$ configurations for the values in the range.

In conclusion PForDelta encodes blocks of k consecutive integers so that they can be stored in a multi-word (i.e. multiple of 32 bits). Those integers that do not fit within b bits are treated as exceptions and stored in another array that is merged to the original sequence of codewords during the decoding phase (thus paying $w+b$ bits). PForDelta is surprisingly succinct in storing the integers which occur in search-engine indexes; but the actual positive feature which makes it very appealing for developers is that it is incredibly fast in decoding because of the word-alignment and the fact that there exist implementations which do not use `if`-statements, and thus avoid *branch* mispredictions.

11.4 Variable-byte code and (s, c)-dense codes

Another class of codes which trade speed by succinctness is the one of the so called (s, c)-*dense codes*. Their simplest instantiation, originally used in the Altavista search engine, is the *variable-byte code* which uses a sequence of bytes to represent an integer x. This byte-aligned coding is useful to achieve a significant decoding speed. It is constructed as follows: the binary representation $B(x)$ is partitioned into groups of 7-bits, possibly the first group is padded by appending 0s to its front; a flag-bit is appended to each group to indicate whether that group is the last one (bit set to 0) or not (bit set to 1) of the representation. The decoding is simple, we scan the byte sequence until we find a byte whose value is smaller than 128.

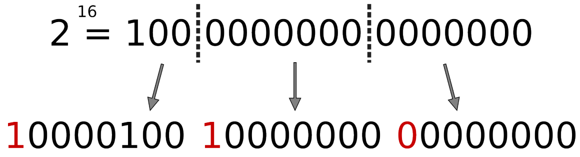


FIGURE 11.5: Variable-byte representation for the integer 2^{16}

The minimum amount of bits necessary to encode x is 8, and on average 4 bits are wasted because of the padding. Hence this method is proper for large values x.

FACT 11.7 *The Variable-byte code of a positive integer x takes* $\lceil \frac{|B(x)|}{7} \rceil$ *bytes, and thus* $8 * \lceil \frac{|B(x)|}{7} \rceil$ *bits. This code is optimal for the distribution* $P[x] \approx \sqrt[7]{1/x^8}$.

The use of the status bit induces a subtle issue, in that it partitions the configurations of each byte into two sets: the values smaller than 128 (status bit equal to 0, called *stoppers*) and the values larger or equal than 128 (status bit equal to 1, called *continuers*). For the sake of presentation we denote the cardinalities of the two sets by s and c, respectively. Of course, we have that $s + c = 256$ because they represent all possibly byte-configurations. During the decoding phase, whenever we encounter a continuer byte, we go on reading, otherwise we stop.

The drawback of this approach is that for any $x < 128$ we use always 1 byte. Therefore if the set S consists of very-small integers, we are wasting bits. Vice versa, if S consists of integers larger than 128, then it could be better to enlarge the set of stoppers. Indeed nobody prevents us to change the distribution of stoppers and continuers, provided that $s + c = 256$. Let us analyze how changes the number of integers which can be encoded with one of more bytes, depending on the choice of s and c:

- One byte can encode the first s integers;
- Two bytes can encode the subsequent sc integers.
- Three bytes can encode the subsequent sc^2 integers.
- k bytes can encode sc^{k-1} integers.

It is evident, at this point, that the choice of s and c depends on the distribution of the integers to be encoded. For example, assume that we want to encode the values $1, \dots, 15$ and they have decreasing frequency; moreover, assume that the word-length is 3 bits (instead of 8 bits), so that $s + c = 2^3 = 8$ (instead of 256).

Table 11.1 shows how the integers smaller than 15 are encoded by using two different choices for s and c: in the first case, the number of stoppers and continuers is 4; in the second case, the

Values	$s = c = 4$	$s = 6, c = 2$
0	000	000
1	001	001
2	010	010
3	011	011
4	100 000	100
5	100 001	101
6	100 010	110 000
7	100 011	110 001
8	101 000	110 010
9	101 001	110 011
10	101 010	110 100
11	101 011	110 101
12	110 000	111 000
13	110 001	111 001
14	110 010	111 010
15	110 011	111 011

TABLE 11.1 Example of (s, c)-encoding using two different values for s and c.

number of stoppers is 6 and the number of continuers is 2. Notice that in both cases we correctly have $s + c = 8$. We point out that in both cases, two words of 3 bits (*i.e.* 6 bits) are enough to encode all the 15 integers; but, while in the former case we can encode only the first four values with one word, in the latter the values encoded using one word are six. This can lead to a more compressed sequence according to the skewness of the distribution of $\{1, \ldots, 15\}$.

This shows, surprisingly, that can be advantageous to adapt the number of stoppers and continuers to the probability distribution of S's values. Figure 11.6 further details this observation, by showing the compression ratio as a function of s, for two different distributions `ZIFF` and `AP`, the former is the classic Zipfian distribution (i.e. $P[x] \approx 1/x$), the latter is the distribution derived from the words of the Associated-Press collection (i.e. $P[x]$ is the frequency of occurrence of the x-th most frequent word). When s is very small, the number of high frequency values encoded with one byte is also very small, but in this case c is large and therefore many words with low frequency will be encoded with few bytes. As s grows, we gain compression in more frequent values and loose compression in less frequent values. At some later point, the compression lost in the last values is larger than the compression gained in values at the beginning, and therefore the global compression ratio worsens. That point give us the optimal s value. In [1] it is shown that the minimum is unique and the authors propose an efficient algorithm to calculate that optimal s.

11.5 Interpolative code

This is an integer-encoding technique that is ideal whenever the sequence S shows *clustered* occurrences of integers, namely subsequences which are concentrated in small ranges. This is a typical situation which arises in the storage of posting lists of search engines [7]. Interpolative code is designed in a *recursive* way by assuming that the integer sequence to be compressed consists of *increasing values*: namely $S' = s'_1, \ldots, s'_n$ with $s'_i < s'_{i+1}$. We can turn the original problem to this one, by just setting $s'_i = \sum_{j=1}^{i} s_j$.

At each iteration we know, for the current subsequence $S'_{l,r}$ to be encoded, the following 5 parameters:

- the left index l and the right index r delimiting the subsequence $S'_{l,r} = \{s'_l, s'_{l+1}, \ldots, s'_r\}$;

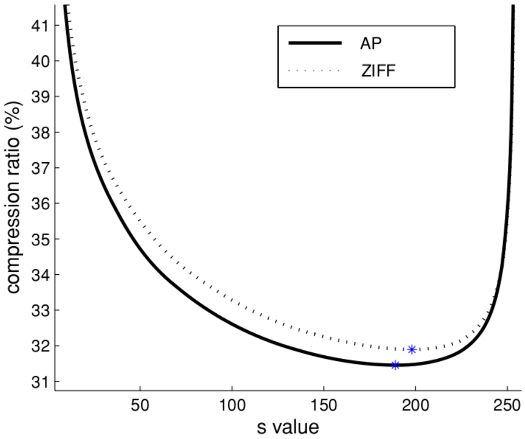


FIGURE 11.6: An example of how compression rate varies according to the choice of s, given that $c = 256 - s$.

- the number n of elements in subsequence $S'_{l,r}$;
- a lower-bound low to the lowest value in $S'_{l,r}$, and an upper-bound hi to the highest value in $S'_{l,r}$, hence $low \leq s'_l$ and $hi \geq s'_r$. The values low and hi do not necessarily coincide with s'_l and s'_r are lower and upper estimates of them during the execution of the recursive calls.

Initially we have $n = |S|$, $l = 1$, $r = n$, $low = s'_1$ and $hi = s'_n$; these five values are stored in the compressed file so that the decoder can read them.

At each recursive call we first encode the middle element s'_m, where $m = \lfloor \frac{l+r}{2} \rfloor$, given the information available for the quintuple $\langle n, l, r, low, hi \rangle$, and then recursively encode the two subsequences $s'_l, \ldots, s'_{m-1}$ and $s'_{m+1}, \ldots, s'_r$, by using a properly recomputed parameters $\langle n, l, r, low, hi \rangle$ for each of them.

In order to succinctly encode s'_m we deploy as much information as possible we can derive from $\langle n, l, r, low, hi \rangle$. Specifically, we observe that it is $s'_m \geq low + m - l$ (in the first half of $S'_{l,r}$ we have $m - l$ distinct values and the smallest one is larger than low) and $s'_m \leq hi - (r - m)$ (via a similar argument). Thus s'_m lies in the range $[low + m - l, hi - r + m]$ so we encode not just s'_m but the difference between this value and its known lower bound $(low + m - l)$ by using $\lceil \log_2 B \rceil$ bits, where $B = hi - low - r + l + 1$ is the size of that interval. In this way, interpolative coding can use very few bits per value whenever the sequence $S'_{l,r}$ is dense. The pseudocode is given in Figure 11.1, where procedure `BinaryCode`(x, a, b) emits the binary encoding of $(x - a)$ in $\lceil \log_2(b - a + 1) \rceil$ bits, by assuming that $x \in \{a, a + 1, \ldots, b - 1, b\}$.

With the exception of the values of the first iteration, which must be known to both the encoder and the decoder, all values for the subsequent iterations can be easily derived from the previous ones. In particular,

- for the subsequence $s'_l, \ldots, s'_{m-1}$, the parameter low is the same of the previous step, since s'_l has not changed; and we can set $hi = s'_m - 1$, since $s'_{m-1} < s'_m$ given that we assumed the integers to be distinct and increasing;
- for the subsequence $s'_{m+1}, \ldots, s'_r$, the parameter hi is the same as before, since s'_r has not changed; and we can set $low = s'_m + 1$, since $s'_{m+1} > s'_m$;
- the parameters l, r and n are recomputed accordingly.

Algorithm 11.1 Interpolative coding $\langle S'[l, r], low, hi\rangle$

```
if $r < l$ then
    return the empty string;
end if
if $l = r$ then
    return BinaryCode($S'[l], low, hi$);
end if
Compute $m = \lfloor \frac{l+r}{2} \rfloor$;
Compute the binary sequence $A_1$ = BinaryCode($S'[m], low + m - l, hi - r + m$);
Compute the binary sequence $A_2$ = Interpolative coding of $\langle S'[l, m-1], low, S'[m] - 1\rangle$;
Compute the binary sequence $A_3$ = Interpolative coding of $\langle S'[m+1, r], S'[m] + 1, hi\rangle$;
Return the concatenation of $A_1 \cdots A_2 \cdot A_3$;
```

Figure 11.7 shows a running example of the behavior of the algorithm. We conclude the description of Interpolative coding by noticing that the encoding of an integer s'_i is not fixed but depends on the distribution of the other integers in S'. This reflects onto the original sequence S in such a way that the same integer x may be encoded differently in its occurrences. This code is therefore *adaptive* and, additionally, it is *not* prefix-free; these two specialties may turn it better than Huffman code, which is optimal among the class of *static* prefix-free codes.

11.6 Elias-Fano code

Unlike Interpolative coding, the code described in this section does not depend on the distribution of the integers to be encoded and, very importantly, it can be *indexed* (by proper compressed data structures) in order to efficiently *access randomly* the encoded integers. This is a positive feature in some settings, and a negative features in other settings. It is positive in the context of storing inverted lists of search engines and adjacency lists of large graphs (as it occurs in Facebook's Unicorn system); it is negative whenever integers occur *clustered* and space is a main concern of the underlying applications. Some authors [6] have recently proposed a (sort of) dynamic-programming approach that turns Elias-Fano coding into a *distribution-sensitive code*, like Interpolative code, thus combining its efficiency in randomly access the encoded integers and its space succinctness which derives from the possible clustering of these integers. Experiments have shown that Interpolative coding is only $2\% - 8\%$ smaller than the optimized Elias-Fano code but up to 5.5 times slower; and Variable-byte code is $10\% - 40\%$ faster than the optimized Elias-Fano code but > 2.5 times larger in space. This means that the Elias-Fano code is a competitive choice whenever an integer sequence must be compressed and (randomly) accessed.

As for the Interpolative code, Elias-Fano code works on a *monotonically increasing* sequence $S' = s'_1, \ldots, s'_n$ with $s'_i < s'_{i+1}$ and $s'_n < u$. We assume that each integer s'_i is represented in binary over $b = \lceil \log_2 u \rceil$ bits. Let us consider a positive integer ℓ, we partition the binary representation of s'_i into two blocks: one is denoted by $H(s'_i)$ and consists of the $b - \ell$ most significant bits (the highest ones), whereas the other block is denoted by $L(s'_i)$ and consists of the ℓ less significant bits (the lowest ones). Clearly $|H(s'_i)| + |L(s'_i)| = b$ bits. The blocks $L(s'_i)$ are concatenated all together, in the order $i = 1, 2, \ldots, n$, thus forming the binary sequence L whose length is $n\ell$ bits. The blocks $H(s'_i)$ are encoded in an apparently strange way whose motivation will be clear when we will evaluate the overall space occupancy of the code. We start by observing that each block $H(s'_i)$ assumes values in $\{0, 1, 2, \ldots, \frac{u}{2^\ell} - 1\}$, each of these values is called *bucket*. So the Elias-Fano code iterates over the buckets $j = 0, 1, \ldots, \frac{u}{2^\ell} - 1$ and constructs the binary sequence H by writing, for each bucket j, the negative unary representation of the number x of elements s'_i for which $H(s'_i) = j$:

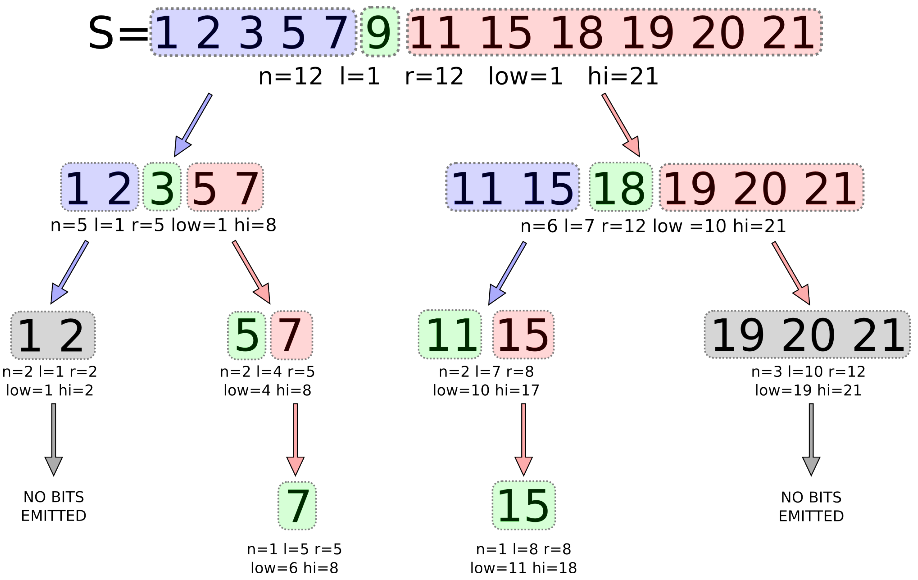


FIGURE 11.7: The blue and the red boxes are, respectively, the left and the right subsequence of each iteration. In the green boxes is indicated the integer s'_m to be encoded. The procedure performs, in practice, a preorder traversal of a balanced binary tree whose leaves are the integers in S. When it encounters a subsequence of the form $[low, low+1, \ldots, low+n-1]$, it doesn't emit anything. Thus, the items are encoded in the following order (in brackets the actual number encoded): 9 (3), 3 (0), 5 (1), 7 (1), 18 (6), 11 (1), 15 (4).

i.e., it writes $1^x\mathtt{0}$ (so that 1 is repeated x times, with $x = 0$ if no $H(s'_i) = j$). The written binary sequence is denoted by H, its length is $n + \frac{u}{2^\ell}$ bits because we write $\frac{u}{2^\ell}$ negative-unary sequences (i.e. one per bucket) consisting of $\frac{u}{2^\ell}$ bits set to $\mathtt{0}$ (i.e. one bit set to $\mathtt{0}$ per bucket) and n bits set to 1 (i.e. each s'_i generates one bit set to 1 into some bucket j). The described process may be easily inverted so that, from H and L, one can reconstruct S'. The total length of the Elias-Fano code is then $n\ell + n + \frac{u}{2^\ell}$ bits, which turns to be minimum for $\ell = \lceil \log \frac{u}{n} \rceil$.

THEOREM 11.1 *The Elias-Fano encoding of a monotonically increasing sequence of n integers in the range* $[0,u)$ *takes* $2n + n\log_2(u/n)$ *bits, regardless of their distribution. This is almost optimal (i.e.* $+2$ *bits per integer) if the integers are uniformly distributed in* $[0,u)$*.*

Figure 11.8 shows a running example of the coding process.

We can augment the Elias-Fano's coding of S' to support efficiently the following two operations:

- Access(i) which, given an index $i \leq n$, returns s'_i;
- NextGEQ(x) which, given an integer $x \leq u$, returns the smallest element s'_i which is greater than or equal to x.

We need to augment H with an auxiliary data structure that efficiently, in time and space, answers a well-known primitive: $\mathsf{Select}_1(p, H)$ which returns the position in H (counting from 1) of the p-th bit set to 1 (similarly it is defined $\mathsf{Select}_0(p, H)$). We do not want to enter here into the technicalities of the Select primitive, we content ourselves in pointing out that this query can be answered in constant time and $o(|H|) = o(n)$ bits of extra space (see [5] and refs therein). Given this

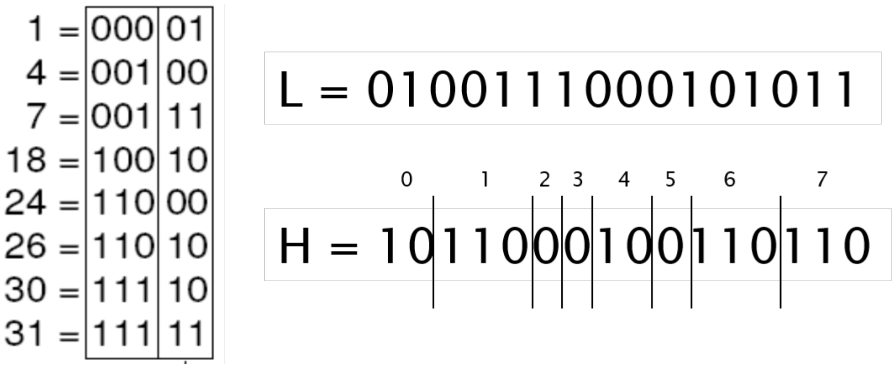


FIGURE 11.8: The Elias-Fano's code for the integer sequence $S' = 1, 4, 7, 18, 24, 26, 30, 31$. In this case $u = 32$ and $n = 8$, so that $\log_2 u = 5$ bits and $\ell = \lceil \log \frac{u}{n} \rceil = 2$. Notice that the value $j = 1$ occurs twice as $H(s'_i)$, namely for the integers 4 and 7, so the binary sequence H encodes $j = 1$ as 110, instead the value $j = 3$ does not occur as $H(s'_i)$ of any integer s'_i and so the binary sequence H encodes $j = 3$ as 0. The binary sequence H consists of $\frac{u}{2^\ell} = 8$ unary sequences (and thus 8 bits set to 0) and $n = 8$ bits set to 1.

data structure built upon the binary array H the two operations above can be implemented in $O(1)$ time as follows.

Access(i) needs to concatenate the higher and the lower bits of s'_i present in L and H, respectively. The block $L(s'_i)$ is easily retrieved by accessing the i-th block of ℓ bits in the binary sequence L. The retrieval of $H(s'_i)$ is a little bit more complicated and boils down to determine the rank of the negative unary sequence in H which contains the 1 corresponding to s'_i; $H(s'_i)$ is eventually obtained by encoding that rank in $\log_2(u/n)$ bits. As an example consider Figure 11.8 and assume to execute Access(5). The 1 corresponding to the integer s'_5 in H, occurs at position 11 and falls in the 6th bucket; and in fact, $H(s'_5) = 110$. More precisely the retrieval of $H(s'_i)$ first computes $\mathsf{Select}_1(i, H)$, which represents the position in H of the 1 denoting s'_i; then, we substract i (it is the i-th one!) to obtain the number of 0s occurring before that 1. By construction this value indicates the rank of the bucket which contains the 1 of s'_i. Referring again to the previous example, $L(s'_5) = 00$ because this is the fifth block of $\ell = 2$ bits in L; and the retrieval of $H(s'_5)$ is obtained by computing $\mathsf{Select}_1(5, H)$ $-5 = 11 - 5 = 6$, and by encoding 6 in $\log_2(u/n) = 3$ bits as $H(s'_5) = 110$. We have thus obtained $s'_5 = 110 \cdot 00 = 24$.

NextGEQ(x) is supported by observing that $p = \mathsf{Select}_0(H(x)) - H(x)$ is the number of integers in S' whose highest bits are smaller than $H(x)$. Thus, $p + 1$ is the starting position in S' of the elements whose highest bits are equal to $H(x)$ (if any) or larger. The integer answering NextGEQ(x) is then identified by scanning the elements at position $p + 1$ and beyond. If the element at position p has its highest bits larger than $H(x)$, then we are done: it is enough to execute Access(p+1). Otherwise, the element at position $p + 1$ has the same highest bits as x, then we have to check whether there exist an element in that bucket which is larger or equal to x. So we compare the corresponding lowest bits in L and scan them until an item $L(y) \geq L(x)$ is found in the bucket of x. But if not such element does exist, then we take the first item of the next bucket (which has larger highest bits than $H(x)$). As an example consider again Figure 11.8 and assume to execute NextGEQ(25). We compute $p = \mathsf{Select}_0(110) - 110 = \mathsf{Select}_0(6) - 6 = 10 - 6 = 4$, and then execute $\mathsf{Access}(\mathsf{p} + 1) = \mathsf{Access}(5) = 24$, since this value is smaller than the queried 25 we scan the (usually small) bucket of integers whose highest bits are $H(x) = 110$ until we meet the integer 26.

FACT 11.8 *It is possible to index the Elias-Fano encoding of a monotonically increasing sequence of n integers in the range $[0, u)$, taking $o(n)$ extra bits and support the random access to any of these integers (i.e.* Access(i) *operation) in constant time. Other operations (such as* NextGEQ(x)*) may be supported efficiently, too.*

A comment is in order at this point. Since Elias-Fano code represents a monotone sequence of integers regardless of its regularities, clustered sequences get significantly worse compression than what Interpolative code is able to achieve. Take, as an illustrative example, the sequence $S' = (1, 2, \ldots, n-1, u-1)$ of n integers. This sequence is highly compressible since the length of the first run and the value of $u-1$ can be encoded in $O(\log u)$ bits each. Conversely Elias-Fano code requires $2 + \log_2(u/n)$ bits per element. Some authors [6] have studied how to turn the Elias-Fano code into an *distribution-sensitive* code that takes advantage of the regularities present into the input sequence S'. They proposed two approaches. A simple one based on a two-level storage scheme: the sequence S' is partitioned into n/m chunks of m integers each, then the "first level" is created by using Elias-Fano to encode the last integer of each chunk (we expect that these m integers are well interspersed in $[0, u)$); then, the "second level" is created by using a specific Elias-Fano code on each chunk whose integers are delta-encoded with respect to the last integer of the previous chunk (available in the first level). This very simple scheme improves the space occupancy of the classic Elias-Fano code (which operates on the entire S') by up to 30% but it slows down the decompression time up to 10%; as far as Interpolative code is concerned, it worsen its space occupancy by up to 10% but it achieves three/four times faster decompression. A more sophisticated approach, based on a Shortest-Path interpretation of Elias-Fano's encoding of S' on a suitably constructed graph, comes even closer in space to Interpolative code and still achieves very fast decompression.

11.7 Concluding remarks

We wish to convince the reader about the generality of the Integer Compression problem, because more and more frequently other compression problems, such as the classic Text Compression, boil down to compressing sequences of integers. An example was given by the LZ77-compressor in Chapter 10. Another example can be obtained by looking at any text T as a sequence of tokens, being them words or single characters; each token can be represented with an integer (aka *token-ID*), so that the problem of compressing T can be solved by compressing the sequence of token-IDs. In order to better deploy one of the previous integer-encoding schemes, one can adopt an interesting strategy which consists of sorting the tokens by decreasing frequency of occurrence in T, and then assign as token-ID their *rank* in the ordered sequence. This way, the more frequent is the occurrence of the token in T, the smaller is the token-ID, and thus the shorter will be the codeword assigned to it by anyone of the previous integer-encoding schemes. Therefore this simple strategy implements the *golden rule* of data compression which consists of assigning short codewords to frequent tokens. If the distribution of the tokens follows one of the distributions indicated in the previous sections, those codewords have optimal length; otherwise, the codewords may be sub-optimal. In [4] it is shown that, if the i-th word follows a Zipfian distribution, such as $P[i] = c(1/i)^{\alpha}$ where c is a normalization constant and α is a parameter depending on the input text, then the previous algorithm using δ-coding achieves a performance close to the *entropy* of the input text.

References

[1] Nieves R. Brisaboa, Antonio Farina, Gonzalo Navarro, José R. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1-33, 2007.

[2] Alistair Moffat. Compressing Integer Sequences and Sets. In *Encyclopedia of Algorithms*. Springer, 2009.

[3] Peter Fenwick. Universal Codes. In *Lossless Data Compression Handbook*. Academic Press, 2003.

[4] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[5] Gonzalo Navarro and Eliana Providel. Fast, Small, Simple Rank/Select on Bitmaps. In *Procs of International Symposium on Experimental Algorithms (SEA)*, pp. 295-306, 2012.

[6] Giuseppe Ottaviano and Rossano Venturini. Partitioned Elias-Fano indexes. In *Procs of ACM SIGIR Conference*, pp. 273-282, 2014.

[7] Hao Yan, Shuai Ding, Torsten Suel. Inverted Index Compression and Query Processing with Optimized Document Ordering. In *Procs of WWW*, pp. 401-410, 2009.

[8] Ian H. Witten, Alistair Moffat, Timoty C. Bell. *Managing Gigabytes*. Morgan Kauffman, second edition, 1999.

12

Statistical Coding

The topic of this chapter is the *statistical coding* of sequences of symbols (aka *texts*) drawn from an alphabet Σ. Symbols may be characters, in this case the problem is named *text compression*, or they can be genomic-bases thus arising the Genomic-DB compression problem, or they can be bits and in this case we fall in the realm of classic data compression. If symbols are integers, then we have the Integer coding problem, addressed in the previous Chapter, which can be solved still with a statistical coder by just deriving statistical information on the integers occurring in the sequence S. In this latter case, the code we derive is an *optimal* prefix-free code for the integers of S, but its coding/decoding time is larger than the one incurred by the integer encoders of the previous Chapter, and indeed, this is the reason for their introduction.

Conceptually, statistical compression may be viewed as consisting of two phases: a *modeling* phase, followed by a *coding* phase. In the modeling phase the statistical properties of the input sequence are computed and a *model* is built. In the coding phase the model is used to compress the input sequence. In the first sections of this Chapter we will concentrate only on the second phase, whereas in the last section we will introduce a sophisticated modeling technique. We will survey the best known statistical compressors: Huffman coding, Arithmetic Coding, Range Coding, and finally Prediction by Partial Matching (PPM), thus providing a pretty complete picture of what can be done by statistical compressors. The net result will be to go from a compression performance that can be bounded in terms of 0-th order entropy, namely an entropy function depending on the probability of single symbols (which are therefore considered to occur i.i.d.), to the more precise k-th order entropy which depends on the probability of k-sized blocks of symbols and thus models the case e.g. of Markovian sources.

12.1 Huffman coding

First published in the early ’50s, Huffman coding was regarded as one of the best methods for data compression for several decades, until the Arithmetic coding made higher compression rates possible at the end of ’60s (see next chapter for a detailed discussion about this improved coder).

Huffman coding is based upon a *greedy algorithm* that constructs a binary tree whose leaves are the symbols in Σ, each provided with a probability $P[\sigma]$. At the beginning the tree consists only of its $|\Sigma|$ leaves, with probabilities set to the $P[\sigma]$s. These leaves constitute a so called *candidate set*, which will be kept updated during the construction of the Huffman tree. In a generic step, the Huffman algorithm selects the two nodes with the smallest probabilities from the candidate set, and creates their parent node whose probability is set equal to the sum of the probabilities of its two children. That parent node is inserted in the candidate set, while its two children are removed from it. Since each step adds one node and removes two nodes from the candidate set, the process stops after $|\Sigma| - 1$ steps, time in which the candidate set contains only the root of the tree. The Huffman tree has therefore size $t = |\Sigma| + (|\Sigma| - 1) = 2|\Sigma| - 1$.

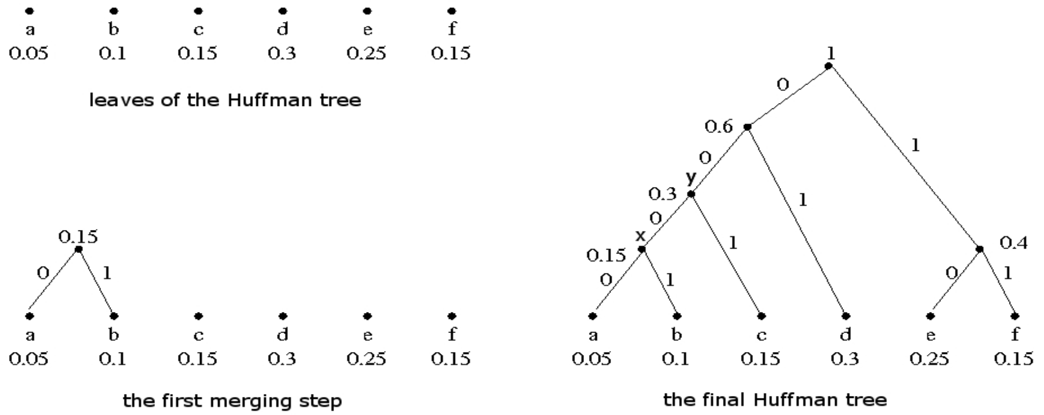


FIGURE 12.1: Constructing the Huffman tree for the alphabet $\Sigma = \{a, b, c, d, e, f\}$.

Figure 12.1 shows an example of Huffman tree for the alphabet $\Sigma = \{a, b, c, d, e, f\}$. The first merge (on the left) attaches the symbols a and b as children of the node x, whose probability is set to $0.05 + 0.1 = 0.15$. This node is added to the candidate set, whereas leaves a and b are removed from it. At the second step the two nodes with the smallest probabilities are the leaf c and the node x. Their merging updates the candidate set by deleting x and c, and by adding their parent node y whose probability is set to be $0.15 + 0.15 = 0.3$. The algorithm continues until there is left only one node (the root) with probability, of course, equal to 1.

In order to derive the Huffman code for the symbols in Σ, we assign binary labels to the tree edges. The typical labeling consists of assigning 0 to the left edge and 1 to the right edge spurring from each internal node. But this is one of the possible many choices. In fact a Huffman tree can originate $2^{|\Sigma|-1}$ labeled trees, because we have 2 labeling choices (i.e. 0-1 or 1-0) for the two edges spurring from each one of the $|\Sigma| - 1$ internal nodes. Given a labeled Huffman tree, the Huffman codeword for a symbol σ is derived by taking the binary labels encountered on the downward path that connects the root to the leaf associated to σ. This codeword has a length $L(\sigma)$ bits, which corresponds to the depth of the leaf σ in the Huffman tree. The Huffman code is *prefix-free* because every symbol is associated to a distinct leaf and thus no codeword is the prefix of another codeword.

We observe that the choice of the two nodes having minimum probability may be *not unique*, and the actual choices available may induce codes which are different in the structure but, nonetheless, they have all the same optimal average codeword length. In particular these codes may offer a *different maximum* codeword length. Minimizing this value is useful to reduce the size of the compression/decompression buffer, as well as the frequency of emitted symbols in the decoding process. Figure 12.2 provides an illustrative example of these multiple choices.

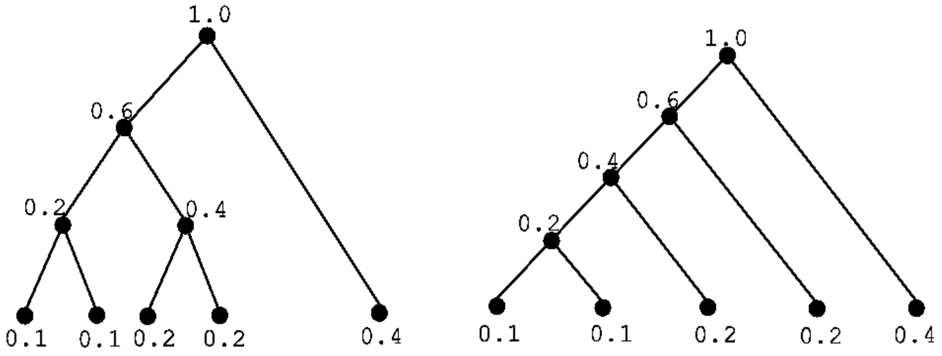


FIGURE 12.2: An example of two Huffman codes having the same average codeword length $\frac{22}{10}$, but different maximum codeword length.

A strategy to minimize the maximum codeword length is to choose the two *oldest nodes* among the ones having same probability and belonging to the current candidate set. Oldest nodes means that they are leaves or they are internal nodes that have been merged farther in the past than the other nodes in the candidate set. This strategy can be implemented by using two queues: the first one contains the symbols ordered by increasing probability, the second queue contains the internal nodes in the order they are created by the Huffman algorithm. It is not difficult to observe that the second queue is sorted by increasing probability too. In the presence of more than two minimum-probability nodes, the algorithm looks at the nodes in the first queue, after which it looks at the second queue. Figure 12.2 shows on the left the tree resulting by this algorithm and, on the right, the tree obtained by using an approach that makes an arbitrary choice.

The compressed file originated by Huffman algorithm consists of two parts: the *preamble* which contains an encoding of the Huffman tree, and thus has size $\Theta(|\Sigma|)$, and the *body* which contains the codewords of the symbols in the input sequence S. The size of the preamble is usually dropped from the evaluation of the length of the compressed file; even if this might be a significant size for large alphabets. So the alphabet size cannot be underestimated, and it must be carefully taken into account. In the rest of the section we will concentrate on the evaluation of the size in bits for the compressed body, and then turn to the efficient encoding of the Huffman tree by proposing the elegant *Canonical Huffman* version which offers space succinctness and very fast decoding speed.

Let $L_C = \sum_{\sigma\in\Sigma} L(\sigma)\, P[\sigma]$ be the average length of the codewords produced by a prefix-free code C, which encodes every symbol $\sigma \in \Sigma$ in $L(\sigma)$ bits. The following theorem states the *optimality* of Huffman coding:

THEOREM 12.1 *If C is an Huffman code, then L_C is the shortest possible average length among all prefix-free codes C', namely it is $L_C \leq L_{C'}$.*

To prove this result we first observe that a prefix-free code can be seen as a binary tree (more precisely, we should say binary trie), so the optimality of the Huffman code can be rephrased as the *minimality of the average depth* of the corresponding binary tree. This latter property can be proved by deploying the following key lemma, whose proof is left to the reader who should observe that, if the lemma does not hold, then a not minimum-probability leaf occurs at the deepest level of the binary tree; in which case it can be swapped with a minimum-probability leaf (therefore not occurring at the deepest level) and thus reduce the average depth of the resulting tree.

LEMMA 12.1 Let T be a binary tree whose average depth is minimum among the binary trees with $|\Sigma|$ leaves. Then the two leaves with minimum probabilities will be at the greatest depth of T, children of the same parent node.

Let us assume that the alphabet Σ consists of n symbols, and symbols x and y have the smallest probability. Let T_C be the binary tree generated by a code C applied onto this alphabet; and let us denote by R_C the *reduced* tree which is obtained by dropping the leaves for x and y. Thus the parent, say z, of leaves x and y is a leaf of R_C with probability $P[z] = P[x] + P[y]$. So the tree R_C is a tree with $n - 1$ leaves corresponding to the alphabet $\Sigma - \{x, y\} \cup \{z\}$ (see Figure 12.3).

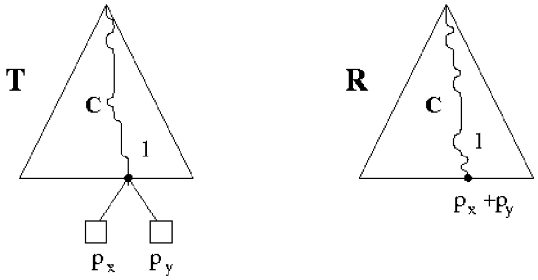


FIGURE 12.3: Relationship between a tree T and its corresponding reduced tree R.

LEMMA 12.2 The relation between the average depth of the tree T with the one of its reduced tree R is given by the formula $L_T = L_R + (P[x] + P[y])$, where x and y are the symbols having the smallest probability.

Proof It is enough to write down the equalities for L_T and L_R, by summing the length of all root-to-leaf paths multiplied by the probability of the landing leaf. So we have $L_T = \left(\sum_{\sigma \neq x,y} P[\sigma]\, L(\sigma)\right) + (P[x]+P[y])(L_T(z)+1)$, where z is the parent of x and y and thus $L_T(x) = L_T(y) = L_T(z)+1$. Similarly, we can write $L_R = \left(\sum_{\sigma \neq x,y} P[\sigma] L(\sigma)\right) + L(z)(P[x] + P[y])$. So the thesis follows. ■

The optimality of Huffman code (claimed in the previous Theorem 12.1) can now be proved by induction on the number n of symbols in Σ. The base $n = 2$ is obvious, because any prefix-free code must assign at least one bit to $|\Sigma|$'s symbols; therefore Huffman is optimal because it assigns the single bit 0 to one symbol and the single bit 1 to the other.

Let us now assume that $n > 2$ and, by induction, assume that Huffman code is optimal for an alphabet of $n - 1$ symbols. Take now $|\Sigma| = n$, and let C be an optimal code for Σ and its underlying distribution. Our goal will be to show that $L_C = L_H$, so that Huffman is optimal for n symbols too. Clearly $L_C \leq L_H$ because C was assumed to be an optimal code for Σ. Now we consider the two reduced trees, say R_C and R_H, which can be derived from T_C and T_H, respectively, by dropping the leaves x and y with the smallest probability and leaving their parent z. By Lemma 12.1 (for the optimal C) and the way Huffman works, this reduction is possible for both trees T_C and T_H. The two reduced trees define a prefix-code for an alphabet of $n-1$ symbols; so, given the inductive hypothesis, the code defined by R_H is optimal for the "reduced" alphabet $\Sigma \cup \{z\} - \{x, y\}$. Therefore $L_{R_H} \leq L_{R_C}$ over this "reduced" alphabet. By Lemma 12.2 we can write $L_H = L_{R_H} + P[x] + P[y]$ and, according to Lemma 12.1, we can write $L_C = L_{R_C} + P[x] + P[y]$. So it turns out that $L_H \leq L_C$ which, combined with the previous (opposite) inequality due to the optimality of C, gives $L_H = L_C$. This actually means that Huffman is an optimal code also for an alphabet of n symbols, and thus inductively proves that it is an optimal code for any alphabet size.

We remark that this statement does not mean that $C = H$, and indeed do exist optimal prefix-free codes which cannot be obtained via the Huffman algorithm (see Figure 12.4). Rather, the previous statement indicates that the average codeword length of C and H is equal. The next fundamental theorem provides a quantitative upper-bound to this average length.

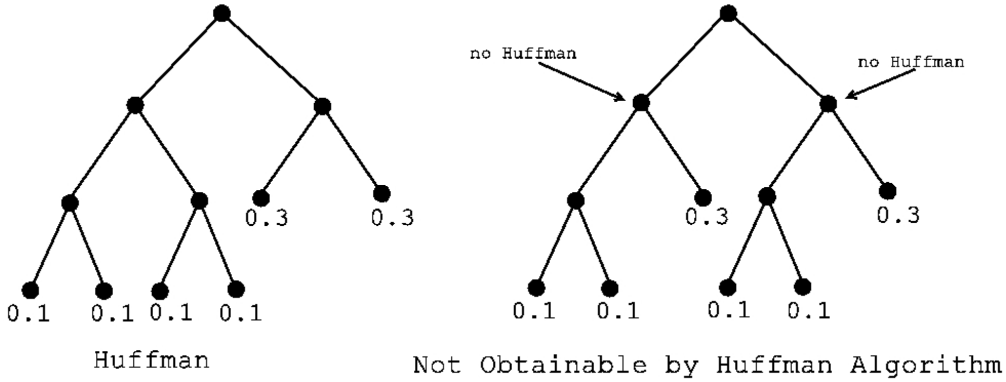


FIGURE 12.4: Example of an optimal code not obtainable by means of the Huffman algorithm.

THEOREM 12.2 *Let $\mathcal{H}$ be the entropy of the source emitting the symbols of an alphabet Σ, of size n, hence $\mathcal{H} = \sum_{i=1}^{n} P[\sigma_i] \log_2 \frac{1}{P[\sigma_i]}$. The average codeword length of the Huffman code satisfies the inequalities $\mathcal{H} \leq L_H < \mathcal{H} + 1$.*

This theorem states that the Huffman code can loose up to 1 bit per compressed symbol with respect to the entropy $\mathcal{H}$ of the underlying source. This extra-bit is a lot or a few depending on the value of $\mathcal{H}$. Clearly $\mathcal{H} \geq 0$, and it is equal to zero whenever the source emits just one symbol with probability 1 and all the other symbols with probability 0. Moreover it is also $\mathcal{H} \leq \log_2 |\Sigma|$, and it is equal to this upper bound for equiprobable symbols. As a result if $\mathcal{H} \gg 1$, the Huffman code is effective and the extra-bit is negligible; otherwise, the distribution is *skewed*, and the bit possibly lost by the Huffman code makes it inefficient. On the other hand Huffman, as any prefix-free code, cannot encode one symbol in less than 1 bit, so the best compression ratio that Huffman can obtain is $\geq \frac{1}{\log_2 |\Sigma|}$. If Σ is ASCII, hence $|\Sigma| = 256$, Huffman cannot achieve a compression ratio for any sequence S which is less than $1/8 = 12,57\%$.

In order to overcome this limitation, Shannon proposed in its famous article of 1948 a simple *blocking scheme* which considers an extended alphabet Σ^k whose symbols are substrings of k-symbols. This way, the new alphabet has size $|\Sigma|^k$ and thus, if we use Huffman on the symbol-blocks, the extra-bit lost is for a block of size k, rather than a single symbol. This actually means that we are loosing a fractional part of a bit per symbol, namely $1/k$, and this is indeed negligible for larger and larger values of k.

So why not taking longer and longer blocks as symbols of the new alphabet Σ^k? This would improve the coding of the input text, because of the blocking, but it would increase the encoding of the Huffman tree which constitutes the preamble of the compressed file: in fact, as k increases, the number of leaves/symbols also increases as $|\Sigma|^k$. The compressor should find the best trade-off between these two quantities, by possibly trying several values for k. This is clearly possible, but yet it is un-optimal; Section 12.2 will propose a provably optimal solution to this problem.

12.1.1 Canonical Huffman coding

Let us recall the two main limitations incurred by the Huffman code:

- It has to store the structure of the tree and this can be costly if the alphabet Σ is large, as it occurs when coding blocks of symbols, possibly words.
- Decoding is slow because it has to traverse the whole tree for each codeword, and every edge of the path (bit of the codeword) may elicit a cache miss. Thus the total number of cache misses could be equal to the total number of bits constituting the compressed file.

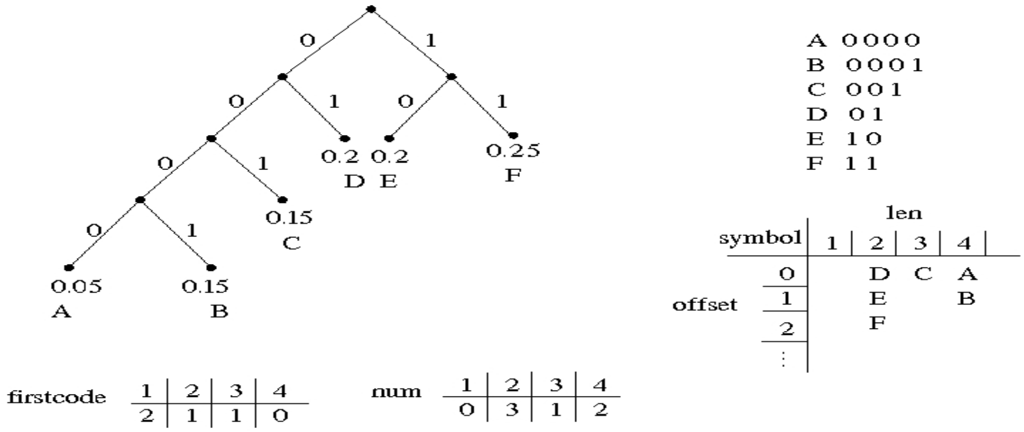
There is an elegant variant of the Huffman code, denoted as *Canonical Huffman*, that alleviates these problems by introducing a special restructuring of the Huffman tree that allows extremely fast decoding and a small memory footprint. This will be the topic of this subsection.

The Canonical Huffman code works as follows:

1. Compute the codeword length $L(\sigma)$ for each symbol $\sigma \in \Sigma$ according to the classical Huffman's algorithm.
2. Construct the array *num* which stores in the entry *num*[ℓ] the number of symbols having Huffman codeword of ℓ-bits.
3. Construct the array *symb* which stores in the entry *symb*[ℓ] the list of symbols having Huffman codeword of ℓ-bits.
4. Construct the array `fc`which stores in the entry `fc`[ℓ] the first codeword of all symbols encoded with ℓ bits;
5. Assign consecutive codewords to the symbols in *symb*[ℓ], starting from the codeword `fc`[ℓ].

Figure 12.5 provides an example of an Huffman tree which satisfies the Canonical property. The *num* array is actually useless, so that the Canonical Huffman needs only to store `fc` and `symb` arrays, which means at most $\mathtt{max}^2$ bits to store `fc` (i.e. `max` codewords of length at most `max` each), and at most $(|\Sigma| + \mathtt{max}) \log_2 (|\Sigma| + 1)$ bits to encode table `symb`. Consequently the key advantage of Canonical Huffman is that we do not need to store the tree-structure via pointers, with a saving of $\Theta(|\Sigma| \log_2 (|\Sigma| + 1))$ bits.

The other important advantage of Canonical Huffman resides in its decoding procedure which does not need to percolate the Huffman tree, but it only operates on the two available arrays, thus



A	0000
B	0001
C	001
D	01
E	10
F	11

	symbol	len 1	2	3	4
offset	0		D	C	A
	1		E		B
	2		F		
	⋮				

firstcode	1	2	3	4
	2	1	1	0

num	1	2	3	4
	0	3	1	2

FIGURE 12.5: Example of canonical Huffman coding.

inducing at most two cache-misses per decoded symbol.[1] The pseudo-code is summarized in the following 6 lines:

```
v = next_bit();
l = 1;
while( v < fc[l] )
    v = 2v + next_bit();
    l++;
return symb[ l, v-fc[l] ];
```

A running example of the decoding process is given un Figures 12.6–12.7. Let us assume that the compressed sequence is 01. The function `next_bit()` reads the incoming bit to be decoded, namely 0. At the first step (Figure 12.6), we have $\ell = 1$, $v = 0$ and $\mathtt{fc}[1] = 2$; so the *while* condition is satisfied (because $v = 0 < 2 = \mathtt{fc}[1]$) and therefore ℓ is incremented to 2 and v gets the next bit 1, thus assuming the value $v = 01 = 1$. At the second step (Figure 12.7), the *while* condition is no longer satisfied because $v = 1 < \mathtt{fc}[2]$ is false and the loop has to stop. The decoded codeword has length $\ell = 2$ and, since $v - \mathtt{fc}[2] = 0$, the algorithm returns the first symbol of *symb*[2] = *D*.

A subtle comment is in order at this point, the value $\mathtt{fc}[1] = 2$ seems impossible, because we cannot represent the value 2 with a codeword consisting of one single bit. This is a *special value* because this way $\mathtt{fc}[1]$ will be surely larger than any codeword of one bit, hence the Canonical Huffman algorithm will surely fetch another bit in the while-cycle.

The correctness of the decoding procedure can be inferred informally from Figure 12.8. The while-guard $v < \mathtt{fc}[\ell]$ actually checks whether the current codeword v is to the left of $\mathtt{fc}[\ell]$ and thus it is to the left of all symbols which are encoded with ℓ bits. In the figure this corresponds to the case $v = 0$ and $\ell = 4$, hence $v = \mathtt{0000}$ and $\mathtt{fc}[4] = \mathtt{0001}$. If this is the case, since the Canonical Huffman tree is skewed to the left, the codeword to be decoded has to be longer and thus a new bit is fetched by the while-body. In the figure this corresponds to fetch the bit 1, and thus set $v = 1$ and $\ell = 5$, so $v = \mathtt{00001}$. In the next step the while-guard is false, $v \geq \mathtt{fc}[\ell]$ (as indeed $\mathtt{fc}[5] = \mathtt{00000}$), and thus v lies to the right of $\mathtt{fc}[\ell]$ and can be decoded by looking at the symbols *symb*[5].

[1] It is reasonable to assume that the number of cache-misses is just 1 because the array `fc` is small and can be fit in cache.

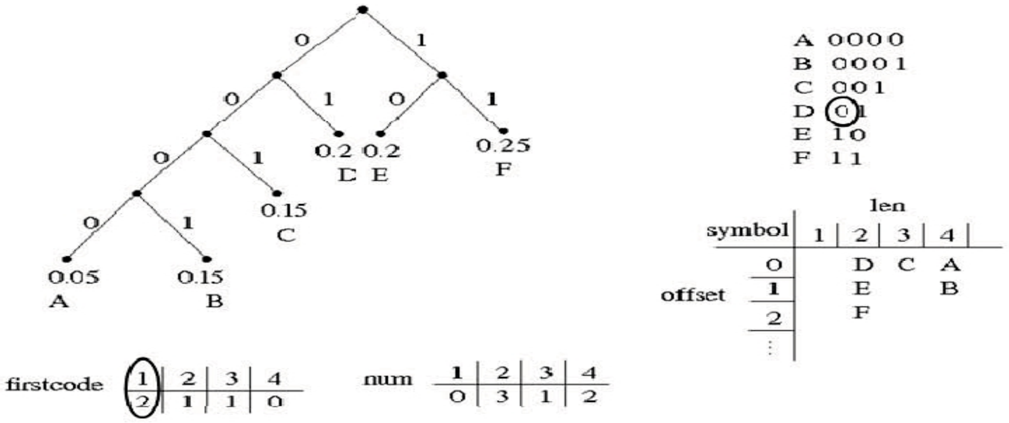


FIGURE 12.6: First Step of decoding 01 via the Canonical Huffman of Figure 12.5.

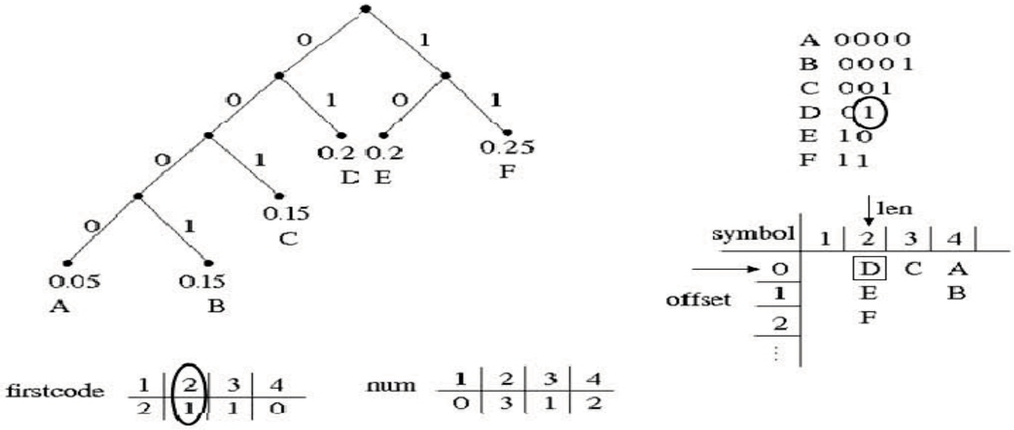


FIGURE 12.7: Second Step of decoding 01 via the Canonical Huffman of Figure 12.5.

The only issue it remains to detail is how to get a Canonical Huffman tree, whenever the underlying symbol distribution does not induce one with such a property. Figure 12.5 actually derived an Huffman tree which was canonical, but this is not necessarily the case. Take for example the distribution: $P[a] = P[b] = 0.05$, $P[c] = P[g] = 0.1$, $P[d] = P[f] = 0.2$, $P[e] = 0.3$, as shown in Figure 12.9. The Huffman algorithm on this tree generates a non Canonical tree, which can be turned into a Canonical one by means of the following few lines of pseudo-code, in which `max` indicates the longest codeword length assigned by the Huffman algorithm:

```
fc[max]=0;
for(l= max-1; l>=1; l--)
    fc[l]=(fc[l+1] + num[l+1])/2;
```

There are two key remarks to be done before digging into the proof of correctness of the algorithm. First, fc[ℓ] is the value of a codeword consisting of ℓ bits, so the reader should keep in mind that fc[5] = 4 means that the corresponding codeword is `00100`, which means that the binary

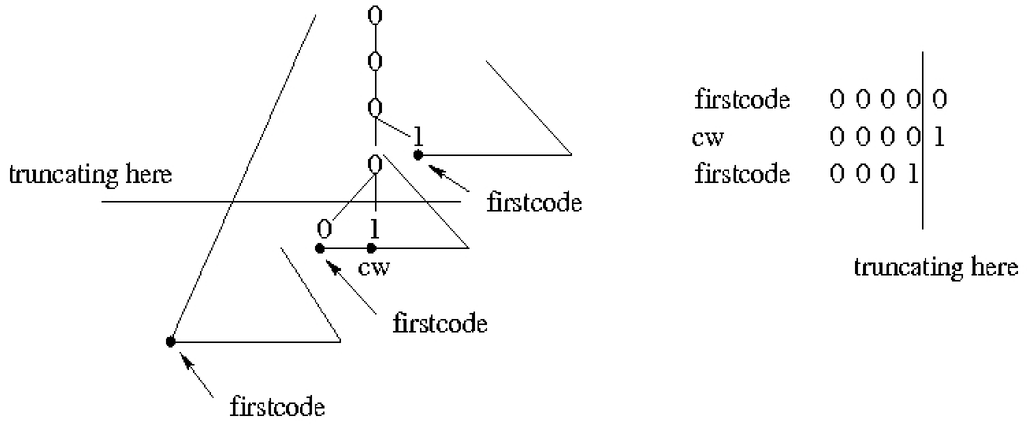


FIGURE 12.8: Tree of codewords.

representation of the value 4 is padded with zeros to have length 5. Second, since the algorithm sets $\texttt{fc}[max] = 0$, the longest codeword is a sequence of `max` zeros, and so the tree built by the Canonical Huffman is totally skewed to the left. If we analyze the formula that computes $\texttt{fc}[\ell]$ we can guess the reason of its correctness. The pseudo-code is reserving $num[\ell + 1]$ codewords of length $\ell + 1$ bits to the symbols in $symb[\ell + 1]$ starting from the value $\texttt{fc}[\ell + 1]$. The first *unused* codeword of $\ell + 1$ bits is therefore given by the value $\texttt{fc}[\ell + 1] + num[\ell + 1]$. So the formula then divides this value by 2, which corresponds to dropping the last $(\ell + 1)$-th bit from the binary encoding of that number. It can be proved that the resulting sequence of ℓ-bits can be taken as the first-codeword $\texttt{fc}[\ell]$ because it does not prefix any other codeword already assigned. The ”reason” can be derived graphically by looking at the binary tree which is being built by Canonical Huffman. In fact, the algorithm is taking the parent of the node at depth $\ell + 1$, whose binary-path represents the value $\texttt{fc}[\ell + 1] + num[\ell + 1]$. Since the tree is a fully binary and we are allocating leaves in the tree from left to right, this node is always a left child of its parent, so its parent is not an ancestor of any $(\ell + 1)$-bit codeword assigned before.

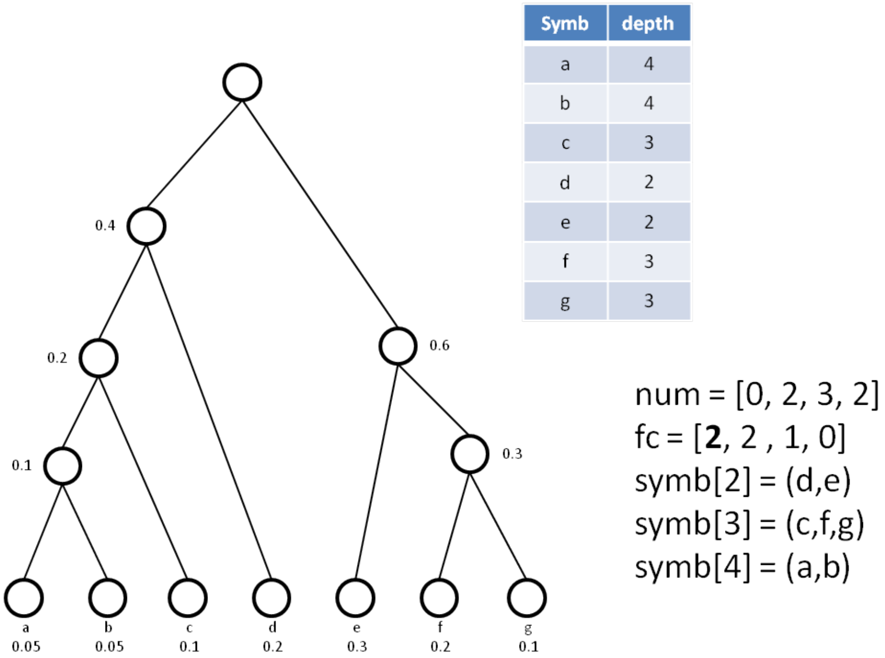
In Figure 12.9 we notice that $\texttt{fc}[1] = 2$ which is an impossible codeword because we cannot encode 2 in 1 bit; nevertheless this is the special case mentioned above that actually *encodes* the fact that no codeword of that length exists, and thus allows the decoder to find always $v < \texttt{fc}[1]$ after having read just one single bit, and thus execute `next_bit()` to fetch another bit from the input and thus consider a codeword of length 2.

12.1.2 Bounding the length of codewords

If the codeword length exceeds 32 bits the operations can become costly because it is no longer possible to store codewords as a single machine word. It is therefore interesting to survey how likely codeword overflow might be in the Huffman algorithm.

Given that the optimal code assigns a codeword length $L(\sigma) \approx \log_2 1/P[\sigma]$ bits to symbol σ, one could conclude that $P[\sigma] \approx 2^{-33}$ in order to have $L(\sigma) > 32$, and hence conclude that this bad situation occurs only after about 2^{33} symbols have been processed. This first approximation is an excessive upper bound.

It is enough to consider a Huffman tree which has the structure of a binary tree, skewed to the left, whose leaf i has frequency $F(i)$ which is an increasing function, hence $F(i+1) < F(i+2)$. Moreover we assume that $\sum_{j=1}^{i} F(j) < F(i + 2)$ in order to induce the Huffman algorithm to join $F(i + 1)$ with



Symb	depth
a	4
b	4
c	3
d	2
e	2
f	3
g	3

num = [0, 2, 3, 2]
fc = [**2**, 2 , 1, 0]
symb[2] = (d,e)
symb[3] = (c,f,g)
symb[4] = (a,b)

FIGURE 12.9: From Huffman tree to a Canonical Huffman Tree.

the last created internal node rather than with leaf $i + 2$ (or all the other leaves $i + 3, i + 4, \ldots$). It is not difficult to observe that $F(i)$ may be taken to be the Fibonacci sequence, possibly with different initial conditions, such as $F(1) = F(2) = F(3) = 1$. The following two sequences show F's values and their cumulative sums for the modified Fibonacci sequence: $F = (1, 1, 1, 3, 4, 7, \ldots)$ and $\sum_{i=1}^{l+1} F(i) = (2, 3, 6, 10, 17, 28, \ldots)$. In particular it is $F(33) = 3.01 * 10^6$ and $\sum_{i=1}^{33} F(i) = 1.28 * 10^7$. The cumulative sum indicates how much text has to be read in order to force a codeword of length l. Thus, the pathological case can occur just after 10 Mb; considerably less than the preceding estimation!

They do exist methods to reduce the codeword lengths still guaranteeing a good compression performance. One approach consists of *scaling the frequency counts* until they form a good arrangement. An appropriate scaling rule is

$$\hat{c}_i = \left\lceil c_i \, \frac{\sum_{i=1}^{L+2} F'(i) - 1 - |\Sigma|}{(\sum_{i=1}^{|\Sigma|} c_i)/c_{min}} \right\rceil$$

where c_i is the actual frequency count of the $i\!-\!th$ symbol in the actual sequence, c_{min} is the minimum frequency count, $\hat{c}_i$ is the scaled approximate count for i-th symbol, L is the maximum bit length permitted in the final code and $\sum_{i=1}^{L+2} F(i)$ represents the length of the text which may induce a code of length $L + 1$.

Although simple to implement, this approach could fail in some situations. An example is when 32 symbols have to be coded in codewords with no more than $L = 5$ bits. Applying the scaling rule we obtain $\sum_{i=1}^{L+2} F(i) - 1 - |\Sigma| = 28 - 1 - 32 = -5$ and consequently negative frequency counts $\hat{c}_i$. It is nevertheless possible to build a code with 5 bits per symbol, just take the fixed-length one! Another solution, which is more time-consuming but not subject to the previous drawback, is the so called *iterative scaling* process. We construct a Huffman code and, if the longest codeword is larger than L bits, all the counts are reduced by some constant ratio (e.g. 2 or the golden ratio 1.618) and a new

Huffman code is constructed. This process is continued until a code of maximum codeword length L or less is generated. In the limit, all symbols will have their frequency equal to 1 thus leading to a fixed-length code.

12.2 Arithmetic Coding

The principal strength of this coding method, introduced by Elias in the '60s, is that it can code symbols arbitrarily close to the 0-th order entropy, thus resulting much better tha Huffman on skewed distributions. So in Shannon's sense it is optimal.

For the sake of clarity, let us consider the following example. Take an input alphabet $\Sigma = \{a, b\}$ with a skewed distribution: $P[a] = \frac{99}{100}$ and $P[b] = \frac{1}{100}$. According to Shannon, the *self information* of the symbols is respectively $i(a) = \log_2 \frac{1}{p_a} = \log_2 \frac{100}{99} \simeq 0,015$ bits and $i(b) = \log_2 \frac{1}{p_b} = \log_2 \frac{100}{99} \simeq 6,67$ bits. Hence the 0-th order entropy of this source is $\mathcal{H}_0 = P[a]\, i(a) + P[b]\, i(b) \simeq 0,08056$ bits. In contrast a Huffman coder, like any prefix-coders, applied to texts generated by this source must use at least one bit per symbol thus having average length $L_H = P[a]\, L(a) + P[b]\, L(b) = P[a] + P[b] = 1 \gg \mathcal{H}_0$. Consequently Huffman is far from the 0-th order entropy, and clearly, the more skewed is the symbol distribution the farthest is Huffman from optimality.

The problem is that Huffman replaces each input symbol with a codeword, formed by an integral number of bits, so the average length of a text T compressed by Huffman is $\Omega(|T|)$ bits. Therefore Huffman cannot achieve a compression ratio better than $\frac{1}{\log_2 |\Sigma|}$, the best case is when we substitute one symbol (encoded plainly with $\log_2 |\Sigma|$ bits) with just 1 bit. This is $1/8 = 12.5\%$ in the case that Σ are the characters of the ASCII code.

To overcome this problem, Arithmetic Coding relaxes the request to be a prefix-coder by adopting a different strategy:

- the compressed output is *not* a concatenation of codewords associated to the symbols of the alphabet.
- rather, a bit of the output can represent *more than one* input symbols.

This results in a better compression, at the cost of slowing down the algorithm and of loosing the capability to access/decode the compressed output from any position.

Another interesting feature of Arithmetic coding is that it works easily also in the case of a dynamic model, namely a model in which probabilities $P[\sigma]$ are updated as the input sequence S is processed. It is enough to set $P[\sigma] = (\ell_\sigma + 1)/(\ell + |\Sigma|)$ where ℓ is the length of the prefix of S processed so far, and ℓ_σ is the number of occurrences of symbol σ in that prefix. The reader can check that this is a sound probability distribution, initially set to the uniform one. Easily enough, these dynamic probabilities can be also kept updated by the decompression algorithm, so that both compressor and decompressor look at the same input distribution and thus decode the same symbols.

12.2.1 Bit streams and dyadic fractions

A bit stream $b_1 b_2 b_3 \ldots b_k$, possibly $k \to \infty$, can be interpreted as a real number in the range $[0, 1)$ by prepending "0." to it:

$$b_1 b_2 b_3 \ldots b_k \to 0.b_1 b_2 b_3 \ldots b_k = \sum_{i=1}^{k} b_i \cdot 2^{-i}$$

A real number x in the range $[0, 1)$ can be converted in a (possibly infinite) sequence of bits with the algorithm `Converter`, whose pseudocode is given below. This algorithm consists of a loop where the variable *output* is the output bitstream and where :: expresses concatenation among bits. The loop has to end when the condition *accuracy* is satisfied: we can decide a level of accuracy in the representation of x, we can stop when we emitted a certain number of bits in output or when we establish that the representation of x is periodic.

Algorithm 12.1 Converter (x)
Require: A real number $x \in [0, 1)$.
Ensure: The string of bits representing x.

```
1: repeat
2:     $x = 2 * x$
3:     if $x < 1$ then
4:         $output = output :: 0$
5:     else
6:         $output = output :: 1$
7:         $x = x - 1$
8:     end if
9: until accuracy
```

A key concept here is the one of *dyadic fraction*, namely a fraction of the form $\frac{v}{2^k}$ where v and k are positive integers. The real number associated to a finite bit stream $b_1 b_2 b_3 \ldots b_k$ is indeed the dyadic fraction $\frac{val(b_1 b_2 b_3 \ldots b_k)}{2^k}$, where $val(s)$ is the value of the binary string s. Vice versa a fraction $\frac{v}{2^k}$ can be written as $.bin_k(v)$, where $bin_k(v)$ is the binary representation of the integer v as a bit string of length k (eventually padded with zeroes).

In order to clarify how `Converter` works, we apply the pseudocode at the number $\frac{1}{3}$:

$$\frac{1}{3} \cdot 2 = \frac{2}{3} < 1 \rightarrow output = 0$$

$$\frac{2}{3} \cdot 2 = \frac{4}{3} \geq 1 \rightarrow output = 01$$

In this second iteration x is greater than 1, so we have concatenated the bit 1 to the output and, at this point, we need to update the value of x executing the line 7 in the pseudocode:

$$\frac{4}{3} - 1 = \frac{1}{3}$$

We have already encountered this value of x, so we can stop the loop and output the periodic representation $\overline{01}$ for $\frac{1}{3}$.
Let us consider another example; say $x = \frac{3}{32}$.

$$\frac{3}{32} \cdot 2 = \frac{6}{32} < 1 \rightarrow output = 0$$

$$\frac{6}{32} \cdot 2 = \frac{12}{32} < 1 \rightarrow output = 00$$

$$\frac{12}{32} \cdot 2 = \frac{24}{32} < 1 \rightarrow output = 000$$

$$\frac{24}{32} \cdot 2 = \frac{48}{32} \geq 1 \rightarrow output = 0001$$

$$\left(\frac{48}{32} - 1\right) \cdot 2 = 1 \geq 1 \rightarrow \mathit{output} = 00011$$

$$(1 - 1) \cdot 2 = 0 < 1 \rightarrow \mathit{output} = 000110$$

$$0 \cdot 2 = 0 < 1 \rightarrow \mathit{output} = 0001100$$

and so on. The binary representation for $\frac{3}{32}$ is $00011\bar{0}$.

12.2.2 Compression algorithm

Compression by Arithmetic coding is iterative: each step takes as input a subinterval of $[0, 1)$, representing the prefix of the input sequence compressed so far, and the *probabilities* and the *cumulative probabilities* of alphabet symbols,[2] and consumes the next input symbol. This subinterval is further subdivided into smaller subintervals, one for each symbol σ of Σ, whose lengths are proportional to their probabilities $P[\sigma]$. The step produces as output a new subinterval that is the one associated to the consumed input symbol, and is contained in the previous one. The number of steps is equal to the number of symbols to be encoded, and thus to the length of the input sequence.

More in detail, the algorithm starts considering the interval $[0, 1)$ and, consumed the entire input, produces the interval $[l, l + s)$ associated to the last symbol of the input sequence. The tricky issue here is that the output is not the pair $\langle l, s\rangle$ (hence two real numbers) but it is just one real $x \in [l, l+s)$, chosen to be a dyadic fraction, plus the length of the input sequence.

In the next section we will see how to choose this value in order to minimize the number of output bits, here we will concentrate on the overall compression stage whose pseudocode is indicated below: the variables l_i and s_i are, respectively, the starting point and the length of the interval encoding the i-th symbol of the input sequence.

Algorithm 12.2 AC-Coding (S)

Require: The input sequence S, of length n, the probabilities $P[\sigma]$ and the cumulative f_σ.
Ensure: A subinterval $[l, l + s)$ of $[0, 1)$.

```
$s_0 = 1$
$l_0 = 0$
$i = 1$
while $i \leq n$ do
    $s_i = s_{i-1} * P[S[i]]$
    $l_i = l_{i-1} + s_{i-1} * f_{S[i]}$
    $i = i + 1$
end while
$output = \langle x \in [l_n, l_n + s_n), n\rangle$
```

As an example, consider the input sequence $S = abac$ with probabilities $P[a] = \frac{1}{2}$, $P[b] = P[c] = \frac{1}{4}$ and cumulative probabilities $f_a = 0$, $f_b = P[a] = \frac{1}{2}$, and $f_c = P[a] + P[b] = \frac{3}{4}$. Following the pseudocode of `AC-Coding (S)` we have $n = 4$ and thus we repeat the internal loop four times.

[2] We recall that the cumulative probability of a symbol $\sigma \in \Sigma$ is computed as $\sum_{c<\sigma} P[c]$ and it is provided by the statistical model, constructed during the modeling phase of the compression process. In the case of a dynamic model, the probabilities and the cumulative probabilities change as the input sequence is scanned.

In the first iteration we consider the first symbol of the sequence, $S[1] = a$, and compute the new interval $[l_1, l_1 + s_1)$ given $P[a]$ and f_a from the static model:

$$s_1 = s_0\, P[S[1]] = 1 \times P[a] = \frac{1}{2}$$

$$l_1 = l_0 + s_0\, f_{S[1]} = 0 + 1 \times f_a = 0$$

In the second iteration we consider the second symbol, $S[2] = b$, the (cumulative) probabilities $P[b]$ and f_b, and determine the second interval $[l_2, l_2 + s_2)$:

$$s_2 = s_1\, P[S[2]] = \frac{1}{2} \times P[b] = \frac{1}{8}$$

$$l_2 = l_1 + s_1\, f_{S[2]} = 0 + \frac{1}{2} \times f_b = \frac{1}{4}$$

We continue this way for the third and the fourth symbols, namely $S[3] = a$ and $S[4] = c$, so at the end the final interval is:

$$[l_4, l_4 + s_4) = \left[\frac{19}{64}, \frac{19}{64} + \frac{1}{64}\right) = \left[\frac{19}{64}, \frac{20}{64}\right) = \left[\frac{19}{64}, \frac{5}{16}\right)$$

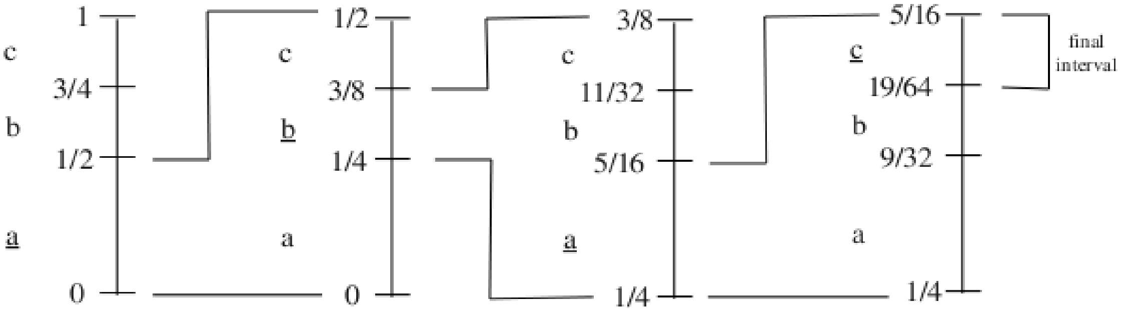


FIGURE 12.10: The algorithmic idea behind Arithmetic coding.

In Figure 12.10 we illustrate the execution of the algorithm in a graphical way. Each step zooms in the subinterval associated to the current symbol. The last step returns a real number inside the final subinterval, hence this number is *inside all* the previously generated intervals too. This number together with the value of n is sufficient to reconstruct the entire sequence of input symbols S, as we will show in the following subsection. In fact, all input sequences of a fixed length n will be associated to distinct sub-intervals which do not intersect and cover [0, 1); but sequences of different length might be nested.

12.2.3 Decompression algorithm

The input consists of the stream of bits resulting from the compression stage, the length of the input sequence n, the symbol probabilities $P[\sigma]$, and the output is the original sequence S given that Arithmetic coding is a *lossless compressor*. The decompressor works also in the case of a dynamic model:

Algorithm 12.3 AC-Decoding (b, n)

Require: The binary representation b of the compressed output, the length n of S, the probabilities $P[\sigma]$ and the cumulative f_σ.

Ensure: The original sequence S.

```
 1: $s_0 = 1$
 2: $l_0 = 0$
 3: $i = 1$
 4: while $i \leq n$ do
 5:     subdivide the interval $[l_{i-1}, l_{i-1} + s_{i-1})$ into subintervals of length proportional to the probabilities of the symbols in $\Sigma$ (in the predefined order)
 6:     take the symbol $\sigma$ corresponding to the subinterval in which $0.b$ lies
 7:     $S = S :: \sigma$
 8:     $s_i = s_{i-1} * P[\sigma]$
 9:     $l_i = l_{i-1} + s_{i-1} * f_\sigma$
10:     $i = i + 1$
11: end while
12: $output = S$
```

- at the first iteration, the statistical model is set to the uniform distribution over Σ;
- at every other iteration, a symbol is decoded using the current statistical model, which is then updated by increasing the frequency of that symbol.

Decoding is correct because encoder and decoder are synchronized, in fact they use the same statistical model to decompose the current interval, and both start from the interval $[0, 1)$. The difference is that the encoder uses symbols to choose the subintervals, whereas the decoder uses the number b to choose (the same) subintervals to zoom in.

As an example, take the result $\langle \frac{39}{128}, 4\rangle$ and assume that the input distribution is $P[a] = \frac{1}{2}$, $P[b] = P[c] = \frac{1}{4}$ (i.e. the one of the previous section). The decoder executes the decompression algorithm starting with the initial interval $[0, 1)$, we suggest the reader to parallel the decompression process with the compression one in Figure 12.10:

- in the first iteration the initial range will be subdivided in three subintervals one for each symbol of the alphabet. These intervals follow a predefined order; in particular, for this example, the first interval is associated to the symbol a, the second to b and the last one to c. The size of every subinterval is proportional to the probability of the respective symbol: $[0, \frac{1}{2})$, $[\frac{1}{2}, \frac{3}{4})$ and $[\frac{3}{4}, 1)$. At this point the algorithm will generate the symbol a because $\frac{39}{128} \in [0, \frac{1}{2})$. After that it will update the subinterval executing the steps 8 and 9, thus synchronizing itself with the encoder.
- in the second iteration the new interval $[0, \frac{1}{2})$ will be subdivided in the subintervals $[0, \frac{1}{4})$, $[\frac{1}{4}, \frac{3}{8})$, $[\frac{3}{8}, \frac{1}{2})$. The second symbol generated will be b because $\frac{39}{128} \in [\frac{1}{4}, \frac{3}{8})$.
- continuing in this way, the third and the fourth iterations will produce respectively the symbols a and c, and the initial sequence will be reconstructed correctly. Notice that we can stop after generating 4 symbols because that was the original sequence length, communicated to the decoder.

12.2.4 Efficiency

Intuitively this scheme performs well because we associate large subintervals to frequent symbols (given that the interval size s_i decreases as $P[S[i]] \leq 1$), and a large final interval requires fewer bits to specify a number inside it. From step 5 of the pseudocode of `AC-Coding (`S`)` is easy to determine the size s_n of the final interval associated to an input sequence S of length n:

$$s_n = s_{n-1} \times P[S[n]] = s_{n-2} \times P[S[n-1]] \times P[S[n-1]] = \ldots =$$

$$= s_0 \times P[S[1]] * \cdots * P[S[n]] = 1 \times \prod_{i=1}^{n} P[S[i]] \qquad (12.1)$$

The formula 12.1 is interesting because it says that s_n depends on the symbols forming S but not on their ordering within S. So the size of the interval returned by Arithmetic coding for S is the same whichever is that ordering. Now, since the size of the interval impacts onto the number of bits returned in the compressed output, we derive that the output size is *independent of* the permutation of S's characters. This does not contradicts with the previous statement, proved below, that Arithmetic coding achieves a performance close to the 0th order empirical entropy $\mathcal{H}_0 = \mathrm{Sum}_{i=1}^{|\Sigma|} P[\sigma_i] \log_2(1/P[\sigma_i])$ of the sequence S, given that entropy's formula is independent of S's symbol ordering too.

We are left with the problem of choosing a number inside the interval $[l_n, l_n + s_n)$ that has the form of a dyadic fraction $\frac{v}{2^k}$ and can be encoded with the fewest bits (i.e. smallest k). The following lemma is crucial to establish the performance and correctness of Arithmetic coding.

LEMMA 12.3 Take a real number $x = 0.b_1 b_2 \cdots$. If we truncate it to its first d bits, we obtain a real number $trunc_d(x) \in [x - 2^{-d}, x]$.

Proof The real number $x = 0.b_1 b_2 \cdots$ differs from its truncation, possibly, on the bits that follow the position d. Those bits have been reset to 0 in $trunc_d(x)$. Therefore we have:

$$x - \mathrm{trunc}_d(x) = \sum_{i=1}^{\infty} b_{d+i} 2^{-(d+i)} \leq \sum_{i=1}^{\infty} 1 \times 2^{-(d+i)} = 2^{-d} \sum_{i=1}^{\infty} \frac{1}{2^i} = 2^{-d}$$

So we have

$$x - \mathrm{trunc}_d(x) \leq 2^{-d} \Longleftrightarrow x - 2^{-d} \leq \mathrm{trunc}_d(x)$$

On the other hand, it is of course $\mathrm{trunc}_d(x) \leq x$ because we have reset possibly few bits to 0. ∎

COROLLARY 12.1 The truncation of $l + \frac{s}{2}$ to its first $\left\lceil \log_2 \frac{2}{s} \right\rceil$ bits falls in the interval $[l, l + s)$.

Proof It is enough to set $d = \left\lceil \log_2 \frac{2}{s} \right\rceil$ in Lemma 12.3, and observe that $2^{-d} \leq \frac{s}{2}$. ∎

At this point we can specialize `AC-Coding(`S`)` in order to return the first $\left\lceil \log_2 \frac{2}{s_n} \right\rceil$ bits of the binary representation of the value $l_n + \frac{s_n}{2}$. Nicely enough, algorithm `Converter` allows to incrementally generate these bits.

For the sake of clarity, let us resume the previous example taking the final interval $[l_4, l_4 + s_4) = [\frac{19}{64}, \frac{20}{64})$ found in the compression stage. We know that $l_4 = \frac{19}{64}$ and $s_4 = \frac{1}{64}$, hence the value to output is

$$l_4 + \frac{s_4}{2} = \frac{19}{64} + \frac{1}{64} \cdot \frac{1}{2} = \frac{39}{128}$$

truncated at the first $\left\lceil \log_2 \frac{2}{s_4} \right\rceil = \log_2 128 = 7$ bits. The resulting stream of bits associated to this value is obtained by executing the algorithm `Converter` for seven times, in this way:

$$\frac{39}{128} \cdot 2 = \frac{78}{128} < 1 \rightarrow \mathit{output} = 0$$

$$\frac{78}{128} \cdot 2 = \frac{156}{128} \geq 1 \rightarrow \mathit{output} = 01$$

$$\left(\frac{156}{128} - 1\right) \cdot 2 = \frac{56}{128} < 1 \rightarrow \mathit{output} = 010$$

$$\frac{56}{128} \cdot 2 = \frac{112}{128} < 1 \rightarrow \mathit{output} = 0100$$

$$\frac{112}{128} \cdot 2 = \frac{224}{128} \geq 1 \rightarrow \mathit{output} = 01001$$

$$\left(\frac{224}{128} - 1\right) \cdot 2 = \frac{192}{128} \geq 1 \rightarrow \mathit{output} = 010011$$

$$\left(\frac{192}{128} - 1\right) \cdot 2 = 1 \geq 1 \rightarrow \mathit{output} = 0100111$$

At the end the encoder sends the pair $\langle 0100111_2, 4 \rangle$ and the statistical model to the decoder given by $\Sigma = \{a, b, c\}$ and the symbol probabilities $P[a] = \frac{1}{2}$, $P[b] = \frac{1}{4}$, $P[c] = \frac{1}{4}$.

We are ready now to prove the main theorem of this section which relates the compression ratio achieved by Arithmetic coding with the 0-th order entropy of S.

THEOREM 12.3 *The number of bits emitted by Arithmetic Coding for a sequence S of length n is at most $2 + n\mathcal{H}_0$, where $\mathcal{H}_0$ is the 0-th order entropy of the input source.*

Proof By Corollary 12.1, we know that the number of output bits is:

$$\left\lceil \log_2 \frac{2}{s_n} \right\rceil \leq 2 - \log_2 s_n = 2 - \log_2 \left(\prod_{i=1}^{n} P[S[i]] \right) = 2 - \sum_{i=1}^{n} \log_2 P[S[i]]$$

If we let n_σ be the number of times a symbol σ occurs in S, and assume that n is sufficiently large, then we can estimate $P[\sigma] \simeq \frac{n_\sigma}{n}$. At this point we can rewrite the summation by iterating not over the positions i in S, but rather by grouping the same symbols and thus iterating over the symbols σ:

$$2 - \sum_{\sigma \in \Sigma} n_\sigma \log_2 P[\sigma] = 2 - n \left(\sum_{\sigma \in \Sigma} P[\sigma] \log_2 P[\sigma] \right) = 2 + n\mathcal{H}_0$$

■

We can draw some considerations from the result just proved:

- there is a waste of only two bits on an entire input sequence S, hence $\frac{2}{n}$ bits per symbol. This is a vanishing lost as the input sequence becomes longer and longer.
- the size of the output is a function of the set of symbols constituting S with their multiplicities, but not of their order.

In the previous section we have seen that Huffman coding requires $n + n\mathcal{H}_0$ bits for compressing a sequence of n symbols, so Arithmetic Coding is much better. Another advantage is that it calculates the representation on the fly thus it can easily accommodate the use of dynamic modeling. On the other hand, it must be said that (Canonical) Huffman is faster and can decompress any portion of the compressed file provided that we known its first codeword. This is however impossible for Arithmetic Coding which allows only the whole decompression of the compressed file. This property justifies the frequent use of Canonical Huffman coding in the context of compressing Web collections, where Σ consists of words/tokens. Such a variant is known as Huffword [7].

12.2.5 Arithmetic coding in practice

The implementation of Arithmetic coding presents two main problems that we comment below.

The number x produced by the coding phase is known only when the entire input is processed: this is a disadvantage in situations like digital communications, in which for the sake of speed, we desire to start encoding/decoding before the source/compressed string is completely scanned; some possible solutions are:

1. the text to be compressed is subdivided into blocks, which are compressed individually; this way, even the problem of specifying the length of the text is relieved: only the length of the last block must be sent to permit its decompression, or the original file can be padded to an integral number of blocks, if the real 'end of file' is not important.
2. the two extremes of the intervals produced at each compression step are compared and the binary prefix on which their binary representation coincides is emitted. This option does not solve the problem completely, in fact it can happen that they don't have any common prefix for a long time, nonetheless this is effective because it happens frequently in practice.

More significantly, the encoding and decoding algorithms presented above require *arithmetic with infinite precision* which is costly to be approximated. There are several proposals about using *finite precision arithmetic* (see e.g. [3, 8]), which nonetheless penalizes the compression ratio up to $n\mathcal{H}_0 + \frac{2}{100}n$. Even so Arithmetic coding is still better than Huffman: $\frac{2}{100}$ vs. 1 bit loss.

The next subsection describes a practical implementation for Arithmetic coding proposed by Witten, Neal and Clearly [8]; sometimes called *Range Coding* [5]. It is mathematically equivalent to Arithmetic Coding, which works with finite precision arithmetic so that subintervals have integral extremes.

12.2.6 Range Coding$^{\infty}$

The key idea is to make some approximations, in order to represent in finite precision real numbers:

- for every symbol σ in the sorted alphabet Σ the probability $P[\sigma]$ is approximated by an integer count $c[\sigma]$ of the number of occurrences of the symbol in the input sequence, and the cumulative probability f_σ with a cumulative count $C[\sigma]$ which sums the counts of all symbols preceding σ in Σ, hence $C[\sigma] = \sum_{\alpha<\sigma} c[\alpha]$. So we have

$$P[\sigma] = \frac{c[\sigma]}{C[|\Sigma| + 1]} \qquad f_\sigma = \frac{C[\sigma]}{C[|\Sigma| + 1]}$$

- the interval $[0, 1)$ is mapped into the integer interval $[0, M)$, where $M = 2^w$ depends on the length w in bits of the memory-word.

- during the i-th iteration of the compression or decompression stages the current subinterval (formerly $[l_i, l_i + s_i)$) will be chosen to have integer endpoints $[L_i, H_i)$ such that

$$L_i = L_{i-1} + \lfloor f_{S[i]} (H_{i-1} - L_{i-1}) \rfloor$$

$$H_i = L_i + \lfloor P[S[i]] (H_{i-1} - L_{i-1}) \rfloor$$

These approximations induce a compression loss empirically estimated (by the original authors) as 10^{-4} bits per input symbol. In order to clarify how it works, we will first explain the compression and decompression stages, and then we will illustrate an example.

Compression stage. In order to guarantee that every interval $[L_i, H_i)$ has non-empty subintervals, at each step we must have

$$H_i - L_i \geq \frac{M}{4} + 2 \tag{12.2}$$

In fact, if we compute the integer starting point of the subinterval L_{i+1} by

$$L_{i+1} = L_i + \lfloor f_{S[i+1]} \cdot (H_i - L_i) \rfloor = L_i + \left\lfloor \frac{C[S[i+1]]}{C[|\Sigma| + 1]} \cdot (H_i - L_i) \right\rfloor$$

since $C[i]$s are strictly increasing, in order to guarantee that we do not have empty subintervals, it is sufficient to have $\frac{H_i - L_i}{C[|\Sigma|+1]} \geq 1$ that can be obtained, from Equ. 12.2, by keeping

$$C[|\Sigma| + 1] \leq \frac{M}{4} + 2 \leq H_i - L_i \tag{12.3}$$

This means that an adaptive Arithmetic Coding, that computes the cumulative counts during compression, must reset these counts every $\frac{M}{4} + 2$ input symbols, or rescale these counts every e.g. $\frac{M}{8} + 1$ input symbols by dividing them by 2.

Rescaling. We proved that, if in each iteration the interval $[L_i, H_i)$ has size $\geq \frac{M}{4} + 2$, we can subdivide it in non-empty subintervals with integer endpoints and sizes proportional to the probabilities of the symbols. In order to guarantee this condition, one can adopt the following *expansion rules*, which are repeatedly checked before each step of the compression process:

1. $[L_i, H_i) \subseteq \left[0, \frac{M}{2}\right) \rightarrow$ output '0', and the new interval is:

$$[L_{i+1}, H_{i+1}) = [2 \cdot L_i, 2 \cdot (H_i - 1) + 2)$$

2. $[L_i, H_i) \subseteq \left[\frac{M}{2}, M\right) \rightarrow$ output '1', and the new interval is

$$[L_{i+1}, H_{i+1}) = \left[2 \cdot \left(L_i - \frac{M}{2}\right), 2 \cdot \left(H_i - 1 - \frac{M}{2}\right) + 2\right)$$

3. if $\frac{M}{4} \leq L_i < \frac{M}{2} < H_i \leq \frac{3M}{4}$ then we cannot output any bit, even if the size of the interval is less than $\frac{M}{4}$, so we have a so called *underflow condition* (that is managed as indicated below).
4. otherwise, it is $H_i - L_i \geq \frac{M}{4} + 2$ and we can continue the compression as is.

In the case of underflow, we cannot emit any bit until the interval falls in one of the two halves of $[0, M)$ (i.e. cases 1 or 2 above). If we suppose to continue and operate on the interval $[\frac{M}{4}, \frac{3M}{4})$ as we did with $[0, M)$, by properly rewriting conditions 1 and 2, the interval size can fall below $\frac{M}{8}$ and thus the same problem arises again. The solution is to use a parameter m that records the number of times that the underflow condition occurred, so that the current interval is within $[\frac{M}{2} - \frac{M}{2^{m+1}}, \frac{M}{2} + \frac{M}{2^{m+1}})$; and observe that, when eventually the interval will not include $\frac{M}{2}$, we will output 01^m if it is in the first half, or 10^m if it is in the second half. After that, we can expand the interval around its halfway point and count the number of expansions:

- mathematically, if $\frac{M}{4} \leq L_i < \frac{M}{2} < H_i \leq \frac{3M}{4}$ then we increment the number m of underflows and consider the new interval

$$[L_{i+1}, H_{i+1}) = \left[2 \cdot \left(L_i - \frac{M}{4}\right), 2 \cdot \left(H_i - 1 - \frac{M}{4}\right) + 2\right)$$

- when expansion 1 or 2 are operated, after the output of the bit, we output also m copies of the complement of that bit, and reset m to 0.

End of the input sequence. At the end of the input sequence, because of the expansions, the current interval satisfies at least one of the inequalities:

$$L_n < \frac{M}{4} < \frac{M}{2} < H_n \quad \text{or} \quad L_n < \frac{M}{2} < \frac{3M}{4} < H_n \tag{12.4}$$

It may be the case that $m > 0$ so that we have to complete the the output bit stream as follows:

- if the first inequality holds, we can emit $01^m1 = 01^{m+1}$ (if $m = 0$ this means to codify $\frac{M}{4}$)
- if the second inequality holds, we can emit $10^m0 = 10^{m+1}$ (if $m = 0$ this means to codify $\frac{3M}{4}$)

Decompression stage. The decoder must mimic the computations operated during the compression stage. It maintains a shift register v of $\lceil \log_2 M \rceil$ bits, which plays the role of x (in the classic Arithmetic coding) and thus it is used to find the next subinterval from the partition of the current interval. When the interval is expanded, v is modified accordingly, and a new bit from the compressed stream is loaded through the function `next_bit`:

1. $[L_i, H_i) \subseteq \left[0, \frac{M}{2}\right) \rightarrow$ consider the new interval

$$[L_{i+1}, H_{i+1}) = [2 \cdot L_i, 2 \cdot (H_i - 1) + 2), \quad v = 2v + \texttt{next\_bit}$$

2. $[L_i, H_i) \subseteq \left[\frac{M}{2}, M\right) \rightarrow$ consider the new interval

$$[L_{i+1}, H_{i+1}) = \left[2 \cdot \left(L_i - \frac{M}{2}\right), 2 \cdot \left(H_i - 1 - \frac{M}{2}\right) + 2\right),$$

$$v = 2 \cdot \left(v - \frac{M}{2}\right) + \texttt{next\_bit}$$

3. if $\frac{M}{4} \leq L_i < \frac{M}{2} < H_i \leq \frac{3M}{4}$ consider the new interval

$$[L_{i+1}, H_{i+1}) = \left[2 \cdot \left(L_i - \frac{M}{4}\right), 2 \cdot \left(H_i - 1 - \frac{M}{4}\right) + 2\right),$$

$$v = 2 \cdot \left(v - \frac{M}{4}\right) + \texttt{next\_bit}$$

4. otherwise it is $H_i - L_i \geq \frac{M}{4} + 2$ and thus we can continue the decompression as is.

In order to understand this decoding process, let us resume the example of the previous sections with the same input sequence $S = abac$ of length $n = 4$, ordered alphabet $\Sigma = \{a, b, c\}$, probabilities $P[a] = \frac{1}{2}$, $P[b] = P[c] = \frac{1}{4}$ and cumulative probabilities $f_a = 0$, $f_b = P[a] = \frac{1}{2}$, and $f_c = P[a] + P[b] = \frac{3}{4}$. We rewrite these probabilities by using the approximations that we have seen above, hence $C[|\Sigma| + 1] = 4$, and we set the initial interval as $[L_0, H_0) = [0, M)$, where M is chosen to satisfy the inequality 12.3:

$$C[|\Sigma| + 1] \leq \frac{M}{4} + 2 \Longleftrightarrow 4 \leq \frac{M}{4} + 2$$

so we can take $M = 16$ and have $\frac{M}{4} = 4$, $\frac{M}{2} = 8$ and $\frac{3M}{4} = 12$ (of course, this value of M is not based on the real machine word length but it is useful for our example). At this point, we have the initial interval

$$[L_0, H_0) = [0, 16)$$

and we are ready to compress the first symbol $S[1] = a$ using the expressions for the endpoints seen above:

$$L_1 = L_0 + \lfloor f_a \cdot (H_0 - L_0) \rfloor = 0 + \lfloor 0 \cdot 16 \rfloor = 0$$

$$H_1 = L_1 + \lfloor P[a] \cdot (H_0 - L_0) \rfloor = 0 + \left\lfloor \frac{2}{4} \cdot 16 \right\rfloor = 8$$

The new interval $[L_1, H_1) = [0, 8)$ satisfies the first expansion rule $[L_1, H_1) \subseteq \left[0, \frac{M}{2}\right)$, hence we output '0' and we consider the new interval (for the expansion rules we do not change index):

$$[L_1, H_1) = [2 \cdot L_1, 2 \cdot (H_1 - 1) + 2) = [0, 16)$$

In the second iteration we consider the second symbol $S[2] = b$, and the endpoints of the new interval are:

$$L_2 = L_1 + \lfloor f_b \cdot (H_1 - L_1) \rfloor = 8$$

$$H_2 = L_2 + \lfloor P[b] \cdot (H_1 - L_1) \rfloor = 12$$

This interval satisfies the second expansion rule $[L_2, H_2) \subseteq \left[\frac{M}{2}, M\right)$, hence we concatenate at the output the bit '1' (obtaining 01) and we consider the interval

$$[L_2, H_2) = \left[2 \cdot \left(L_2 - \frac{M}{2}\right), 2 \cdot \left(H_2 - 1 - \frac{M}{2}\right) + 2\right) = [0, 8)$$

that satisfies again one of the expansion rules, hence we apply the first rule and we obtain the result $output = 010$ and:

$$[L_2, H_2) = [2 \cdot L_1, 2 \cdot (H_2 - 1) + 2) = [0, 16)$$

For the third symbol $S[3] = a$, we obtain

$$L_3 = L_2 + \lfloor f_a \cdot (H_2 - L_2) \rfloor = 0$$

$$H_3 = L_3 + \lfloor P[a] \cdot (H_2 - L_2) \rfloor = 8$$

and this interval satisfies the first rule, hence $output = 0100$ and

$$[L_3, H_3) = [2 \cdot L_3, 2 \cdot (H_3 - 1) + 2) = [0, 16)$$

We continue this way for the last symbol $S[4] = c$:

$$[L_4, H_4) = [L_3 + \lfloor f_c \cdot (H_3 - L_3) \rfloor, L_4 + \lfloor P[c] \cdot (H_3 - L_3) \rfloor) = [12, 16)$$

so get the output sequence $output = 01001$ and

$$[L_4, H_4) = \left[2 \cdot \left(L_4 - \frac{M}{2}\right), 2 \cdot \left(H_4 - 1 - \frac{M}{2}\right) + 2\right) = [8, 16)$$

so we expand again and get the output sequence *output* = 010011 and

$$[L_4, H_4) = \left[2 \cdot \left(L_4 - \frac{M}{2}\right), 2 \cdot \left(H_4 - 1 - \frac{M}{2}\right) + 2\right) = [0, 16)$$

At the end of the input sequence the last interval should satisfy the Equ. 12.4. This is true for our interval, hence, as we know, at this point the encoder sends the pair $\langle 010011_2, 4\rangle$ and the statistical model $P[a] = \frac{1}{2}$, $P[b] = P[c] = \frac{1}{4}$ to the decoder (we are assuming a static model).

As far as the decoding stage is concerned, the first step initializes the shift register v (of length $\lceil \log_2 M \rceil = \lceil \log_2 16 \rceil = 4$) with the first $\lceil \log_2 16 \rceil = 4$ bits of the compressed sequence, hence $v = 0100_2 = 4_{10}$. At this point the initial interval $[L_0, H_0) = [0, 16)$ is subdivided in three different subintervals, one for every symbol in the alphabet, according to the probabilities: $[0, 8)$, $[8, 12)$ and $[12, 16)$. The symbol generated at the first iteration will be a because $v = 4 \in [0, 8)$. At this point we apply the first expansion rule of the decompression process because $[L_1, H_1) = [0, 8) \subseteq [0, \frac{M}{2})$, obtaining:

$$[L_1, H_1) = [2 \cdot L_1, 2 \cdot (H_1 - 1) + 2) = [0, 16)$$

$$v = 2 \cdot v + \texttt{next\_bit} = \texttt{shift}_{sx}(0100_2) + 1_2 = 1000_2 + 1_2 = 1001_2 = 9_{10}$$

In the second iteration the interval $[0, 16)$ is subdivided another time in the same ranges of integers and the generated symbol will be b because $v = 9 \in [8, 12)$. This last interval satisfies the second rule, hence:

$$[L_2, H_2) = \left[2 \cdot \left(L_2 - \frac{M}{2}\right), 2 \cdot \left(H_2 - 1 - \frac{M}{2}\right) + 2\right) = [0, 8)$$

$$v = 2 \cdot \left(v - \frac{M}{2}\right) + \texttt{next\_bit} = \texttt{shift}_{sx}(1001_2 - 1000_2) + 1_2 = 0010_2 + 1_2 = 0011_2 = 3_{10}$$

We apply now the first rule (the function `next_bit` returns '0' if there are not more bits in the compressed sequence):

$$[L_2, H_2) = [2 \cdot L_2, 2 \cdot (H_2 - 1) + 2) = [0, 16)$$

$$v = 2 \cdot v + \texttt{next\_bit} = \texttt{shift}_{sx}(0011_2) + 0_2 = 0110_2 = 6_{10}$$

This interval is subdivided again, in the third iteration, obtaining the subintervals $[0, 8)$, $[8, 12)$ and $[12, 16)$. Like in the previous steps, we find a as generated symbol because $v = 6 \in [0, 8)$, and we modify the interval $[L_3, H_3) = [0, 8)$ and the shift register v as the first rule specifies:

$$[L_3, H_3) = [2 \cdot L_3, 2 \cdot (H_3 - 1) + 2) = [0, 16)$$

$$v = 2 \cdot v + \texttt{next\_bit} = \texttt{shift}_{sx}(0110_2) + 0_2 = 1100_2 = 12_{10}$$

The last generated symbol will be c because $v = 12 \in [12, 16)$, and the entire input sequence is exactly reconstructed. The algorithm can stop because it has generated 4 symbols, which was provided as input to the decoder as length of S.

12.3 Prediction by Partial Matching$^\infty$

In order to improve compression we need better models for the symbol probabilities. A typical approach consists of estimating them by considering not just individual symbols, and thus assume that they occur independently of each other, but evaluating the *conditional probability* of their occurrence in S, given few previous symbols, the so called *context*. In this section we will look at a particular *adaptive* technique to build a context–model that can be combined very well with Arithmetic Coding because it generates skewed probabilities and thus high compression. This method is called *Prediction by Partial Matching* (shortly, PPM), it allows to move from 0-th order entropy coders to k-th order entropy coders. The implementation of PPM suffers two problems: (i) they need proper data structures to maintain updated in an efficient manner all conditional probabilities, as S is scanned; (ii) at the beginning the estimates of the context-based probabilities are poor thus producing an inefficient coding; so proper adjustments have to be imposed in order to quicker establish good statistics. In the rest of the section we will concentrate on the second issue, and refer the reader to the literature for the first one, namely [4, 7].

12.3.1 The algorithm

Let us dig into the algorithmic structure of PPM. It uses a suite of finite contexts in order to predict the next symbol. Frequency counts of these contexts are updated at each input symbol, namely PPM keeps counts for each symbol σ and for each context α of length ℓ, of how many times the string $\alpha\,\sigma$ occurs in the prefix of S processed so far. This way, at step i, PPM updates the counts for $\sigma = S[i]$ and its previous contexts $\alpha = S[i-\ell, i-1]$, for any $\ell = 0, 1, \ldots, K$ where K is the maximum context-model admitted.

In order to encode the symbol $S[i]$, PPM starts from the longest possible context of length K, namely $S[i-K, i-1]$, and then switches to shorter and shorter contexts until it finds the one, say $S[i-\ell, i-1]$, which is able to predict the current symbol. This means that PPM has a counting for $S[i]$ in $S[i-\ell, i-1]$ which is strictly larger than 0, and thus the estimated probability for this occurrence is not zero. Eventually it reaches the context of order -1 which corresponds to a model in which all symbols have the same probabilities. The key compression issue is how to encode the length ℓ of the model adopted to compress $S[i]$, so that also the decoder can use that context in the decompression phase. We could use an integer encoder, of course, but this would take an integral number of bits per symbol, thus vanishing all efforts of a good modeling; we need something smarter.

The algorithmic idea is to turn this problem into a symbol-encoding problem, by introducing an *escape* symbol (`esc`) which is emitted every time a context-switch has to be performed. So `esc` signals to the decoder to switch to the next shorter model. This escaping-process continues till a model where the symbol is not novel is reached. If the current symbol has never occurred before, $K+1$ escape symbols are transmitted to make the decoder to switch to the (-1)–order model that predicts all possible symbols according to the uniform distribution. Note that the 0–order context corresponds to a distribution of probabilities estimated by the frequency counts, just as in the classic Arithmetic coding. Using such strategy, the probability associated to a symbol is always the one that has been calculated in the longest context where the symbol has previously occurred; so it should be more precise than just the probabilities based on individual counts for the symbols in Σ.

We notice that at the beginning, some contexts may be missing; in particular, when $i \leq K$, the encoder and the decoder do not use the context with maximum order K, but the context of length $(i-1)$, hence the one read so far. When the second symbol is used, the 0–order context is used and, if the second symbol is novel, an `esc` is emitted. The longest context K is used when $i > K$: in this case K symbols have been read already, and the K–order context is available. To better understand how PPM works, we consider the following example.

Let the input sequence S be the string *abracadabra* and let $K = 2$ be the longest context used in the calculation of K–order models and its *conditional probabilities*. As previously said, the only model available when the algorithm starts is the (-1)–order model. So when the first symbol a is read, the (-1)–order assigns to it the uniform probability $\frac{1}{|\Sigma|} = \frac{1}{5}$. At the same time, PPM updates frequency counts in the 0–order model assigning a probability $P[a] = \frac{1}{2}$ and $P[\texttt{esc}] = \frac{1}{2}$. In this running example we assume that the escape symbol is given a count equal to the total number of different characters in the model. Other strategies to assign a probability to the escape symbol will be discussed in detail in section 12.3.3.

PPM then reads b and uses the 0–order model, which is currently the longest one, as explained above. An escape symbol is transmitted since b has never been read before. The (-1)–order model is so used to compress b and then both the 1–order and the 0–order models are updated. In the 0–order model we have $P[a] = \frac{1}{4}$, $P[b] = \frac{1}{4}$, $P[\texttt{esc}] = \frac{2}{4} = \frac{1}{2}$ (two distinct symbols have been read). In the 1–order model the probabilities are $P[b|a] = \frac{1}{2}$ and $P[\texttt{esc}|a] = \frac{1}{2}$ (only one distinct symbol is read). Now let us suppose that the current symbol is $S[6] = d$. Since it is the first occurrence of d in S, three escape symbols will be transmitted for switching from the 2–order to the (-1)–order model. Table 12.1 shows the predictions of the four models after that PPM, in its version C (see Sec. 12.3.3), has completed the processing of the input sequence.

We remark that PPM offers probability estimates which can then be used by an Arithmetic coder to compress a sequence of symbols. The idea is to use every model (either the one which offers a successful prediction or the one which leads to emit an `esc`) to partition the current interval of the Arithmetic coding stage and the current symbol to select the next sub-interval. This is the reason why PPM is better looked as a context-modeling technique rather than a compressor, the coding stage could be implemented via Arithmetic coding or any other statistical encoder.

Order $k = 2$			Order $k = 1$			Order $k = 0$			Order $k = -1$		
Predictions	c	p	Predictions	c	p	Predictions	c	p	Predictions	c	p
ab → r	2	$\frac{2}{3}$	a → b	2	$\frac{2}{7}$	→ a	5	$\frac{5}{16}$	→ $\forall\sigma[i]$	1	$\frac{1}{\lvert\Sigma\rvert} = \frac{1}{5}$
→ `esc`	1	$\frac{1}{3}$	→ c	1	$\frac{1}{7}$	→ b	2	$\frac{2}{16}$			
			→ d	1	$\frac{1}{7}$	→ c	1	$\frac{1}{16}$			
ac → a	1	$\frac{1}{2}$	→ `esc`	3	$\frac{2}{7}$	→ d	1	$\frac{1}{16}$			
→ `esc`	1	$\frac{1}{2}$				→ r	2	$\frac{2}{16}$			
			b → r	2	$\frac{2}{3}$	→ `esc`	5	$\frac{5}{16}$			
ad → a	1	$\frac{1}{2}$	→ `esc`	1	$\frac{1}{3}$						
→ `esc`	1	$\frac{1}{2}$									
			c → a	1	$\frac{1}{2}$						
br → a	2	$\frac{2}{3}$	→ `esc`	1	$\frac{1}{2}$						
→ `esc`	1	$\frac{1}{3}$									
			d → a	1	$\frac{1}{2}$						
ca → d	1	$\frac{1}{2}$	→ `esc`	1	$\frac{1}{2}$						
→ `esc`	1	$\frac{1}{2}$									
			r → a	2	$\frac{2}{3}$						
da → b	1	$\frac{1}{2}$	→ `esc`	1	$\frac{1}{3}$						
→ `esc`	1	$\frac{1}{2}$									
ra → c	1	$\frac{1}{2}$									
→ `esc`	1	$\frac{1}{2}$									

TABLE 12.1 Version C of PPM models after processing the whole $S = abracadabra$.

12.3.2 The principle of exclusion

It is possible to use the knowledge about symbol frequencies in k-order models to improve the compression rate when switching through escape symbols to low-order ones. Suppose that the whole input sequence S of the example above has been processed and that the following symbol to be encoded is *c*. The prediction *ra* → *c* is used (since *ra* is the current 2–order context, with $K = 2$) and thus *c* is encoded with a probability of $\frac{1}{2}$ using 1 bit. Suppose now that, instead of *c*, the character *d* follows *abracadabra*. In the context *ra* an escape symbol is transmitted (encoded with a $\frac{1}{2}$ probability) and PPM switches to the 1–order model. In this model, the prediction *a* → *d* can be used and *d* can be encoded with probability $\frac{1}{7}$. This prediction takes into account the fact that, earlier in the processing, the symbol *d* has followed *a*. However the fact that *ra* was seen before followed by *c* implies that the current symbol cannot be *c*. In fact, if it were followed by *c*, PPM would not have switched to the 1–order model and would have used the longest context *ra*. The decoder can so *exclude* the case of an *a* followed by a *c*. This reduces the frequency count of the group *a* → by one and the character *d* could thus be encoded with probability $\frac{1}{6}$.

Suppose now that after *abracadabra* the novel symbol *e* occurs. In this case a sequence of escape symbols is transmitted to make the decoder switch to the -1–order model. Without *exclusion* the novel symbol would be encoded with a probability of $\frac{1}{|\Sigma|}$. However, if PPM is now using the (-1)–order model, it means that none of the previously seen symbols is the current one. The symbols already read can thus be excluded in the computation of the prediction since it is impossible that they are occurring at this point of the input scan. The probability assigned to the novel symbol by the (-1)–order model using the *exclusion principle* is $\frac{1}{|\Sigma|-q}$ where q is the total number of distinct symbols read in S. Performing this technique takes a little extra time but gives a reasonable payback in terms of extra compression, because all nonexcluded symbols have their probability increased.

12.3.3 Zero Frequency Problem

It may seem that the performance of PPM should improve when the length of the maximum context is increased. Fig 12.11 shows an experimental evidence of this intuition.

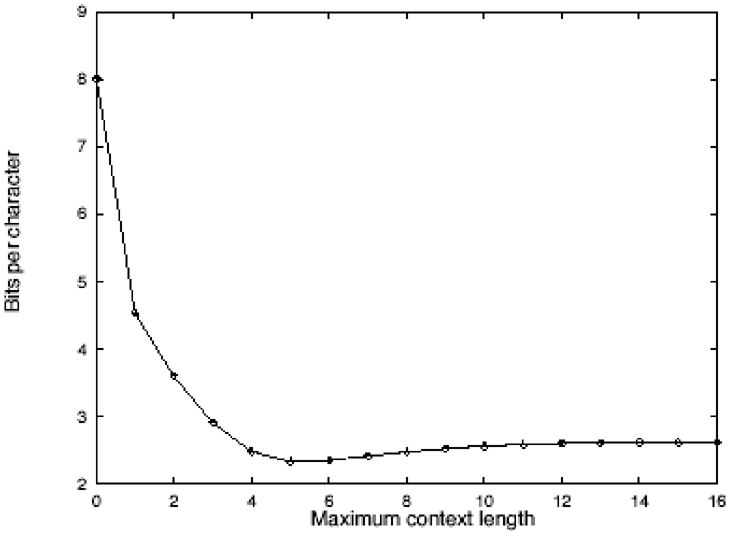


FIGURE 12.11: PPM compression ratio on increasing context length. Computed on Thomas Hardy's text *Far from the Madding Crowd*.

We notice that with contexts longer than 4–5 characters there is no improvement. The reason is that with longer contexts, there is a greater probability to use the escape mechanism, transmitting as many escape symbols as it is needed to reach the context length for which non-null predictions are available.

Anyway, whichever is the context length used, the choice of the encoding method for the escape symbol is very important. There is no sound theoretical basis for any particular choice to assign probabilities to the escape symbol if no a priori knowledge is available. A method is chosen according to the programmer experience only. The various implementations of PPM described below are identified by the escape method they use. For example, PPMA stands for PPM with escape method *A*. Likewise, PPMC or PPMD use escape methods *C* and *D*. The maximum order of the method can be included too. For example, PPMC5 stands for PPM with escape method *C* and maximum order-model $K = 5$. In the following, Cn will denote a context, σ a symbol in the input alphabet, $c(\sigma)$ the number of times that the symbol σ has occurred in context Cn, n the number of times the current context Cn has occurred, q the total number of distinct symbols read.

Method A [1, 6]. This technique allocates one count to the possibility that a symbol will occur in a context in which it has not been read before. The probability $P[\sigma]$ that σ will occur again in the same context is estimated by $P[\sigma] = \frac{c(\sigma)}{1+n}$. The probability that a novel symbol will occur in Cn, i.e. the escape probability $P[\mathtt{esc}]$, is then:

$$\begin{aligned} P[\mathtt{esc}] &= 1 - \sum_{\sigma\in\Sigma, c(\sigma)>0} P[\sigma] = 1 - \sum_{\sigma\in\Sigma, c(\sigma)>0} \frac{c(\sigma)}{1+n} \\ &= 1 - \frac{1}{1+n} \sum_{\sigma\in\Sigma, c(\sigma)>0} c(\sigma) = 1 - \frac{n}{1+n} = \frac{1}{1+n} \end{aligned}$$

`Method B [1, 6].` The second technique classifies a symbol as novel unless it has already occurred twice. The motivation is that a symbol that has occurred only once can be an anomaly. The probability of a symbol is thus estimated by $P[\sigma] = \frac{c(\sigma)-1}{n}$, and q is set as the number of distinct symbols seen so far in the input sequence. The escape probability now is:

$$\begin{aligned} P[\mathtt{esc}] &= 1 - \sum_{\sigma\in\Sigma, c(\sigma)>0} P[\sigma] \\ &= 1 - \sum_{\sigma\in\Sigma, c(\sigma)>0} \frac{c(\sigma)-1}{n} \\ &= 1 - \frac{1}{n} \sum_{\sigma\in\Sigma, c(\sigma)>0} (c(\sigma)-1) \\ &= 1 - \frac{1}{n} \left(\sum_{\sigma\in\Sigma, c(\sigma)>0} c(\sigma) - \sum_{\sigma\in\Sigma, c(\sigma)>0} 1 \right) \\ &= 1 - \frac{1}{n}(n-q) = \frac{q}{n} \end{aligned}$$

Method C [4]. It is an hybrid between the previous two methods. When a novel symbol occurs, a count of 1 is added both to the escape count and to the new symbol count. In this way the total count increases by 2. Thus it estimates the probability of a symbol σ as $P[\sigma] = \frac{c(\sigma)}{n+q}$ and the escape probability $P[\mathtt{esc}]$ as:

$$P[\mathtt{esc}] = 1 - \sum_{\sigma\in\Sigma, c(\sigma)>0} \frac{c(\sigma)}{n+q} = \frac{q}{n+q}$$

Method D. It is a minor modification of method C. It treats in a more uniform way the occurrence of a novel symbol: instead of adding 1, it adds $\frac{1}{2}$ both to the escape and to the new symbol. The probability of a symbol is then:

$$P[\sigma] = \frac{c(\sigma) - \frac{1}{2}}{n} = \left(\frac{2 \cdot c(\sigma) - 1}{2 \cdot n}\right)$$

The probability of the escape symbol is:

$$\begin{aligned} P[\mathtt{esc}] &= 1 - \sum_{\sigma\in\Sigma,c(\sigma)>0} P[\sigma] \\ &= 1 - \sum_{\sigma\in\Sigma,c(\sigma)>0} \left(\frac{c(\sigma) - \frac{1}{2}}{n}\right) \\ &= 1 - \frac{1}{n}\left(\sum_{\sigma\in\Sigma,c(\sigma)>0} c(\sigma) - \sum_{\sigma\in\Sigma,c(\sigma)>0} \frac{1}{2}\right) \\ &= 1 - \frac{n}{n} + \frac{1}{n}\overbrace{\sum_{\sigma\in\Sigma,c(\sigma)>0} \frac{1}{2}}^{\frac{q}{2}} \\ &= \frac{q}{2n} \end{aligned}$$

Method P [6]. The method is based on the assumption that symbols appear according to a Poisson process. Under such hypothesis, it is possible to extrapolate probabilities from an N symbol sample to a larger sample of size $N' = (1 + \theta)N$, where $\theta > 0$. Denoting with t_i the number of distinct symbols occurred exactly i times in the sample of size N, the number of novel symbols can be approximated by $t_1\theta - t_2\theta^2 + t_3\theta^3 - \cdots$. The probability that the next symbol will be novel equals the number of expected new symbols when $N' = N + 1$. In this case:

$$P[\mathtt{esc}] = t_1 \frac{1}{n} - t_2 \frac{1}{n^2} + t_3 \frac{1}{n^3} - \cdots$$

Method X [6]. This method approximates method P computing only the first term of the series since in most cases n is very large and t_i decreases rapidly as i increases: $P[\mathtt{esc}] = \frac{t_1}{n}$. The previous formula may also be interpreted as approximating the escape probability by counting the symbols that have occurred only once.

Method XC [6]. Both methods P and X break down when either $t_1 = 0$ or $t_1 = n$. In those cases the probability assigned to the novel symbol is, respectively, 0 or 1. To overcome this problem, method XC uses method C when method X breaks down:

$$P[\mathtt{esc}] = \begin{cases} \frac{t_1}{n} & 0 < t_1 < n \\ \frac{q}{n+q} & \text{otherwise} \end{cases}$$

Method X1. The last method described here is a simple variant of method X that avoids breaking down to a probability of 0 or 1. Instead of using two different methods, it adds 1 to the total count. The probability of a generic symbol σ is $P[\sigma] = \frac{c(\sigma)}{n+t_1+1}$, and the probability of the escape symbol is $P[\mathtt{esc}] = \frac{t_1+1}{n+t_1+1}$.

References

[1] John G. Clearly and Ian H. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32:396–402, 1984.

[2] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software Practice and Experience*, 327–336, 1994.

[3] Paul Howard and Jeffrey S. Vitter. Arithmetic Coding for Data Compression. *Proceedings of the IEEE*, 857–865, 1994.

[4] Alistair Moffat. Implementing the PPM Data Compression Scheme. *IEEE Transactions on Communications*, 38:1917–1921, 1990.

[5] G. Nigel and N. Martin. Range Encoding: an algorithm for removing redundancy from a digitised message. Presented in March 1979 to the *Video & Data Recording Conference*, 1979.

[6] Ian H. Witten and Timoty C. Bell. The Zero-Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression. *IEEE Transactions on Information Theory*, 37:1085–1094, 1991.

[7] Ian H. Witten, Alistair Moffat, Timoty C. Bell. *Managing Gigabytes*. Morgan Kauffman, second edition, 1999.

[8] Ian H. Witten, Radford M. Neal and John G. Clearly. Arithmetic Coding for data compression. *Communications of the ACM*, 520-540, 1987.

13

Dictionary-based compressors

The methods based on a *dictionary* take a totally different approach to compression than the statistical ones: here we are not interested in deriving the characteristics (i.e. probabilities) of the source which generated the input sequence S. Rather, we assume to have a *dictionary of strings*, and we look for those strings in S, replacing them with a *token* which identifies them in the dictionary. The choice of the dictionary is of course crucial in determining how well the file is compressed. An English dictionary will have a hard time to compress an Italian text, for instance; and it would be totally unappropriate to compress an executable file. Thus, while a *static* dictionary can be used to compress very well certain specific and known in advance kinds of files, it cannot be used for a good *general-purpose compressor*. Moreover, we don't want to transmit the full dictionary along with each compressed file – and it's often unreasonable to assume the receiver already has a copy of our dictionary.

So, starting from 1977, Ziv and Lempel introduced a family of compressors which addressed successfully these problems by designing two algorithms, named LZ77 and LZ78 from the initials of the inventors and the years of the proposal, which use the input sequence they are compressing as the dictionary, and substitute each occurrence of an already seen string with either the offset of its previous position or an ID assigned incrementally to new dictionary phrases. The dictionary is *dynamically built* in the sense that it starts empty and then it grows as the input sequence is processed; at the beginning low compression is achieved, but after some kbs good compression is obtained. For typical textual files, those methods achieve about 33% compression ratio. The Lempel-Ziv's compressors are very popular because of their `gzip` instantiation, and constitute the base of more sophisticated compressors in use today, such as `7zip` and `LZMA` and `LZ4`. In the following paragraphs, we will show them in detail, along with some interesting variants.

13.1 LZ77

Ziv and Lempel, in their seminal paper of 1977 [11], described their contribution as follows *"[...] universal coding scheme which can be applied to any discrete source and whose performance is comparable to certain optimal fixed code book scheme designed for completely specified sources [...]"*. They key expression is "comparable to [...] fixed code book scheme designed for completely specified sources", because the authors compare to previously designed statistical compressors, such as Huffman and Arithmetic, for which a statistical characterization of the source was necessary.

Conversely, dictionary-based compressors waive this characterization which is derived *implicitly* by observing *substring repetitiveness* via a fully syntactic approach.

We will not dig into the observations which provide a mathematical ground to these comments [11, 3], rather we will concentrate only on the algorithmic issues. LZ77's compressor is based on a *sliding window* $W[1, w]$ which contains a portion of the input sequence that has been processed so far, typically consisting of the last w characters, and a *look-ahead buffer* B which contains the suffix of the text still to be processed. In the following picture the window $W = aabbababb$ is of size 9, and the rest of the input sequence is $B = baababaabbaa\$$.

$$\longleftarrow \cdots \boxed{\text{aabbababb}}\; baababaabbaa\$ \longrightarrow$$

The algorithm works inductively by assuming that everything occurring before B has been processed and compressed by LZ77; where W is initially set to the empty string. The compressor operates in two main stages: *parsing* and *encoding*. Parsing consists of transforming the input sequence S into a sequence of triples of integers (called *phrases*). Encoding turns these triples into a (compressed) bit stream by applying either a statistical compressor (i.e. Huffman or Arithmetic) to each triplet-component individually, or an integer encoding scheme.

So the interesting algorithmic stage is the parsing stage, which works as follows. LZ77 searches for the longest prefix α of B which occurs as a substring of WB. We write the concatenation WB rather than the single string B because the previous occurrence we are searching for may start in W and extend up to within B. Say α occurs at distance d from the current position (namely the beginning of B), and it is followed by character c in B, then the triple generated by LZ77 is $\langle d, |\alpha|, c\rangle$ where $|\alpha|$ is the length of the *copied* string. If a match is not found the output triple becomes $\langle 0, 0, B[1]\rangle$. We notice that any occurrence of α in W must be followed by a character different of c, otherwise α would be *not* the longest prefix of B which repeats in W.

After that this triple is emitted, LZ77 advances in B by $|\alpha| + 1$ positions, and slides W correspondingly. We talk about LZ77 as a dictionary-based compressor because "the dictionary" is not explicitly stored, rather it is implicitly formed by all substrings of S which start in W and extend rightward, possibly ending in B. Each of those substrings is represented by the triple indicated above. The dictionary is *dynamic* because at every shift it has to be updated by removing the substrings starting in $W[1, |\alpha| + 1]$, and adding the substrings starting in $B[1, |\alpha| + 1]$.

The role of the sliding window is easy to explain, it delimits the size of the dictionary which is quadratic in W's length, so it impacts significantly onto the time cost for the search of α. As a running example, let us consider the following sequence of LZ77-parsing steps:

$$\begin{array}{rcl}
\boxed{\,\boxed{\text{aabbabab}}} & \Longrightarrow & \langle 0, 0, a\rangle \\
\boxed{\text{a}\boxed{\text{abbabab}}} & \Longrightarrow & \langle 1, 1, b\rangle \\
\boxed{\text{aab}\boxed{\text{babab}}} & \Longrightarrow & \langle 1, 1, a\rangle \\
\boxed{\text{aabba}\boxed{\text{bab}}} & \Longrightarrow & \langle 2, 3, EOF\rangle \\
 & \cdots\cdots \quad \cdots\cdots &
\end{array}$$

It is interesting to note that the last phrase $\langle 2, 3, EOF\rangle$ presents a copy-length which is larger than the copy-distance; this actually indicates the special situation mentioned above in which α starts in W and ends in B. Even if this *overlapping* occurs, the copy-step that must be executed by LZ77 in decompression is not affected, provided that it is executed sequentially according to the following piece of code:

```
for i = 0 to L-1 do { S[s+i] = S[s-d+i]; }
s = s+L;
```

where the triple to be decoded in $\langle d, L, c\rangle$ and $S[1, s-1]$ is the prefix of the input sequence which has been already decompressed. Since $d \leq |W|$ and the window size is up to few Megabytes, the copy operation does not elicit any cache miss, thus making the decompression process very fast indeed. The longer is the window W, the longer are possibly the phrases, the fewer is their number and thus possibly the shorter is the compressed output; but of course, in terms of compression time, the longer is the time to search for the longest copied α. Vice versa, the shorter is W, the worse is the compression ratio but the faster is the compression time. This trade-off is evident and its magnitude depends on the input sequence.

To slightly improve compression we make the following observation which is due to Storer and Szymanski [8] and dates back to 1982. In the parsing process two situations may occur: a longest match has been found, or it has not. In the former case it is not reasonable to add the character following α (third component in the triple), given that we anyway advance in the input sequence. In the latter case it is not reasonable to emit two 0s (first two components in the triple) and thus waste one integer encoding. The simplest solution to these two inefficiencies is to always output a pair, rather than a triple, with the form: $\langle d, |\alpha|\rangle$ or $\langle 0, B[1]\rangle$. This variant of LZ77 is named LZss, and it is often confused with LZ77, so we will use it from this point on.

By referring to the previous running example, LZss would obtain the parsing:

$$\begin{array}{rcl}
|\text{aabbabab} & \Longrightarrow & \langle 0, a\rangle \\
\text{a}|\text{abbabab} & \Longrightarrow & \langle 1, 1\rangle \\
\text{aa}|\text{bbabab} & \Longrightarrow & \langle 0, b\rangle \\
\text{aab}|\text{babab} & \Longrightarrow & \langle 1, 1\rangle \\
\text{aabb}|\text{abab} & \Longrightarrow & \langle 3, 2\rangle \\
\text{aabbab}|\text{ab} & \Longrightarrow & \langle 2, 2\rangle
\end{array}$$

Gzip: a smart and fast implementation of LZ77. The key programming problem when implementing LZ77 is the *fast search* for the longest prefix α of B which repeats in W. A brute-force algorithm that checks the occurrence of every prefix of B in W, via a linear backward scan, would be very time-consuming and thus unacceptable for compressing Gbs files.

Fortunately, this process can be accelerated by using a suitable data structure. `Gzip`, the most popular implementation of LZ77, uses a hash table to determine α and find its previous occurrence in W. The idea is to store in the hash table all 3-grams occurring in W, namely all triplets of contiguous characters, by using as `key` the 3-gram and as its `satellite` data the position in B where that 3-gram occurs. Since a 3-gram may repeat multiple times in W, the hash table saves for a given 3-gram all of its multiple occurrences, sorted by increasing position in S. This way, when W shifts to the right because of the emission of the pair $\langle d, \ell\rangle$, the hash table can be updated by deleting the ℓ 3-grams starting at $W[1, \ell]$, and inserting the ℓ 3-grams starting at $B[1, \ell]$.

The search for α is implemented as follows:

- first, the 3-gram $B[1, 3]$ is searched in the hash table. If it does not occur, then `Gzip` emits the phrase $\langle 0, B[1]\rangle$, and the parsing advances of one single character. Otherwise, it determines the list $\mathcal{L}$ of occurrences of $B[1, 3]$ in W.
- second, for each position i in $\mathcal{L}$ (which is expressed as absolute position in S), the algorithm compares character-by-character $S[i, n]$ against B in order to compute their longest common prefix. At the end, the position $i^* \in \mathcal{L}$ sharing this longest common prefix is determined, as well as it is found α.

- finally, let p be the current position of B in S, the algorithm emits the pair $\langle p - i^*, |\alpha|\rangle$, and advances the parsing of $|\alpha|$ positions.

`Gzip` implements the encoding of the phrases by using Huffman over two alphabets: the one formed by the lengths of the copies plus the literals, and the one formed by the distances of the copies. This trick is sufficiently smart to save one extra bit to distinguish between the two types of pairs. In fact, $\langle 0, c\rangle$ is represented as the Huffman encoding of c, and $\langle d, \ell\rangle$ is represented reversed by anticipating the Huffman encoding of ℓ. Given that literals and copy-lengths are encoded within the same alphabet, the decoder fetches the next codeword and decompresses it, so being able to distinguishing whether the next item is a character c or a length ℓ. According to the result, it can either restart the decoding of the next pair (c has been decoded), or it can decode d (ℓ has been decoded) by using the other Huffman code.

`Gzip` deploys an additional programming trick that further speed ups the compression process. It consists of sorting the list of occurrences of the 3-grams from recent to oldest matches, and possibly stop the search for α when a sufficient number of candidates has been checked. This trades the length of the longest match against the speed of the search. As far as the size of the window W is concerned, `Gzip` allows to specify `-1`, . . . , `-9` which actually means that the size may vary from 100Kb to 900Kb, with a consequent improvement of the compression ratio, at the cost of slowing down the compression speed. Not surprisingly, the longer is W, the faster is the decompression because the smaller is the number of encoded phrases, and thus the smaller is the number of cache misses induced by the Huffman decoding process.

For other implementations of LZ77, the reader can look at Chapter 10 where we discussed the use of the Suffix Tree and unbounded window; as well as we refer to [4] for details about implementations which take into account the size of the compressed output (in bits) which clearly depends on the number of phrases but also from the values of their integer components, in a way that cannot be underestimated. Briefly, it is not necessarily the case that a longer α induces a shorter compressed output, because it might need to copy α from a far distance d, thus taking many bits for its encoding; rather, it might be better to divide α into two substrings which can be copied closer enough that the total number of bits required for their encoding is less than the ones needed for d.

13.2 LZ78

The sliding window used by LZ77 from the one hand speeds up the search for the longest phrase to encode, but from the other hand limits the search space, and thus the ultimate compression ratio. In order to avoid this problem and still keep a fast compression stage, Ziv and Lempel devised in 1978 another algorithm, which has been consequently called LZ78 [12]. The key idea is to build *incrementally* an *explicit dictionary* that contains only a subset of the substrings of the input sequence S, selected according to a simple rule that is detailed below. Concurrently, S is decomposed into phrases which are taken from the current dictionary.

Phrase detection and dictionary update are deeply intermingled. Let S' be the sequence to be parsed yet, and let $\mathcal{D}$ be the current dictionary in which every phrase f is identified via the integer $\mathtt{id}(f)$. The parsing of S' consists of determining its longest prefix f' which is also a phrase of $\mathcal{D}$, and substituting it with the pair $\langle \mathtt{id}(f'), c\rangle$ where c is the character following f' in S'. Next, $\mathcal{D}$ is updated by adding the new phrase $f'c$, which is just one character longer than the phrase $f' \in \mathcal{D}$. Therefore the dictionary is *prefix-complete* because it will contain all the prefixes of every phrase in $\mathcal{D}$, moreover its size grows with the length of the input sequence.

It goes without saying that, as it occurred for LZ77, the stream of pairs generated by the LZ78-parsing will be encoded via a statistical compressor (such as Huffman or Arithmetic) or via any variable-length integer encoder. This will produce the compressed bit stream, eventual output of LZ78.

Input	Output	Dictionary
-	-	0: empty
a	<0, a>	1: a
ab	<1, b>	2: ab
b	<0, b>	3: b
aba	<2, a>	4: aba
bb	<3, b>	5: bb
ba	<3, a>	6: ba
abab	<4, b>	7: abab
aa	<1, a>	8: aa

TABLE 13.1 LZ78-parsing of the string $S = aabbababbbaababaa$.

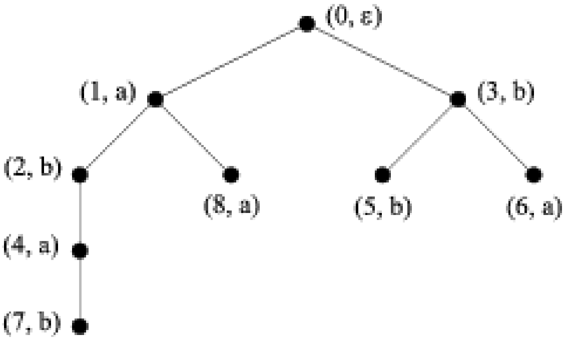


FIGURE 13.1: The *trie* for the dictionary in table 13.1

As an illustrative example, let us consider the sequence $S = abcdefg\ldots$ and the dictionary $\mathcal{D}$ containing the phrase *abcd* with id = 43, but not containing the phrase *abcde*. Given this scenario, LZ78 outputs the pair $\langle 43, e\rangle$ and adds the new phrase *abcde* to the current dictionary. The parsing will then continue over the suffix $S = fg\ldots$. Table 13.1 reports a full running example.

The decompressor works in a very similar way: it reads a pair $\langle \mathtt{id}, x\rangle$, it determines the phrase f corresponding to the integer id in the current dictionary $\mathcal{D}$, emits the substring fc and updates the current dictionary by adding that substring as a new phrase.

The LZ78 algorithm needs an efficient data structure to manage the dictionary, which can be easily found in the *trie* (see Chapter 9), given that it supports fast insertion and prefix-search of strings. The prefix-complete property satisfied by $\mathcal{D}$ ensures that the trie is *uncompacted*, namely every edge is labeled with a single character. The encoding algorithm fits nicely on this structure (see Figure 13.1). Searching for the longest prefix of S' which is a dictionary phrase, can be implemented by traversing the trie according to S''s characters until a trie leaf is reached. That is the detected phrase f'. In addition, the new phrase $f'c$ is inserted in the trie by just appending a new node to the leaf for f' and labeling it with the single character c. That new node will be actually a leaf of the trie.

The final question is how do we manage large files and, thus, large dictionaries which host longer and longer phrases. There are a few possibilities to cope with this problem:

1. Freeze the dictionary, disallowing the entry of new strings. This is the simplest option.
2. Discard the dictionary, starting with a new empty one. This can also be an advantage if the file can be seen as structured in blocks, each one with a different set of recurring strings.

Input	Output	Dictionary
a	-	0-255:'\0'-'\255'
a	97 (a)	256: aa
b	97 (a)	257: ab
b	98 (b)	258: bb
ab	98 (b)	259: ba
aba	257 (ab)	260: aba
eof	97 (aba)	261: aba EOF

TABLE 13.2 LZW-encoding of $S = aabbababa$; 97 and 98 are the ASCII codes for a and b.

3. Before inserting a new string, delete the least recently used one. This solution recalls a sort of LRU model in the dictionary access and management.

13.3 LZW

LZW is a very popular variant of LZ78, developed by Welch in 1984 [10]. Its main objective is to avoid the need for the second component of the pair $\langle \texttt{id}(f'), c\rangle$, and thus for the byte representing a character. To accomplish this goal, before the start of the algorithm, all possible one-character strings are written into the dictionary. This means that the phrase-ids from 0 to 255 have been allocated to these characters. Next, the parsing of S starts searching for the longest prefix f' which matches a phrase in $\mathcal{D}$. Since the next prefix $f'c$ of S' does not occur in $\mathcal{D}$, then $f'c$ is added to the dictionary, taking the next available id, and the next phrase to detect starts from *c rather than* from the following character, as instead done in LZ78 and LZ77. So parsing and dictionary updating are misaligned.

Decoding is a bit tricky because of this misalignment. Assume that decoding has to manage two ids i' and i'', and call their corresponding dictionary phrases f' and f''. The decoder, in order to re-align the dictionary, has to create the phrase n' from the reading of i' and i'', setting $n' = f'f''[1]$, where $f''[1]$ is the first character of the phrase f''. This seems easy but, indeed, it is not!

If f' and f'' are already available in $\mathcal{D}$, the construction of n' is a trivial task, just set $n' = f'f''[1]$. But if f'' is not available the situation gets complicate! Let us look how this can happen. Take the compression stage when the compressor emitted i' for f' and inserted $n' = f'f''[1]$ in $\mathcal{D}$. Clearly the compressor knows f'' and so can do this insertion.

But if the next phrase f'' starts with n', then the decompressor is in trouble. In fact the decompressor sees i', decodes it and obtains f', but then it needs to construct n' and insert it in $\mathcal{D}$. This seems not possible because n' needs $f''[1]$ and we have that f'' is prefixed by n' which is exactly the string we are willing to reconstruct. Thus we falled in a sort of circular definition.

To circumvent this circularity it is enough to observe that f'' consists of at least one character (recall that we initialized $\mathcal{D}$ with all single characters) and it is $f''[1] = n'[1] = (f'f''[1])[1] = f'[1]$ which is indeed available! So the reconstruction of n' is possible even in this special case, by setting $n' = f'f'[1]$. Hence the decompressor can correctly construct n', insert it in $\mathcal{D}$, and thus re-align the dictionaries available at LZW's compressor and decompressor after the reading of i'.

Table 13.3 shows the actions taken by the LZW-decoder over the stream of ids obtained from Table 13.2. The dictionary starts with all the possible characters in the ASCII code, each one with its value, so the new phrases take ids from 256. Notice that the special case occurs when 260 is read from the compressed sequence but the phrase 260 is not in the dictionary because it has yet to be constructed. Nevertheless, by using the observations above, we can conclude that $n' = f'f'[1] = aba$.

Input	Dictionary	Output
-	0-255:'\0'-'\255'	
97	256: a?	a
97	256: aa	a
	257: a?	
98	257: ab	b
	258: b?	
98	258: bb	b
	259: b?	
257	259: ba	ab
	260: ab?	
260	261: aba	aba

TABLE 13.3 LZW-decoding of the id-stream: 97,97,98,98,257,257,258,256,259,259,97.

As the LZ77 algorithm, LZW has many common-use implementations among which we point out the popular GIF image format [9]. It assumes that the original (uncompressed) image is rectangular and uses 8 bits per pixel, so the alphabet has size 256 and the input sequence S comes as a normal stream of bytes, obtained by reading line-by-line the pixels of the image[1]. Since 8 bits are very few to represent all possible colors of an image, each value actually is an index in a *palette*, whose entries are 24-bits descriptions of the actual color (the typical RGB format). This restricts the maximum number of different colors present in an image to 256.

Some researchers [5] explored the possibility to introduce a lossy variant of GIF compression without changing the way the output is represented: this would give the possibility to have a shorter format, using however standard decoders. The basic idea is in fact quite simple: instead of looking for the longest *exact* match in the dictionary while parsing, we perform some kind of *approximate* matching. In this way we can find longer matches, thus reducing the output size, but at the cost of representing a slightly different image. Approximate matching of two strings of colors is done with a measure of difference based on their actual RGB values, which must be guaranteed to not exceed a threshold value.

13.4 On the optimality of compressors$^{\infty}$

The literature shows many results regarding the optimality of LZ-inspired algorithms. Ziv and Lempel themselves demonstrated that LZ77 is optimal for a certain family of sources (see [11]), and LZ78 asymptotically reaches the best compression ratio among finite-state compressors (see [12]). Optimality here means that, assuming the string to compress is infinite and is produced by a *stationary ergodic source with a finite alphabet*, then the compression ratio asymptotically tends to the entropy of the underlying source. More recent results made it possible to have a quantitative estimate of algorithms' *redundancy*, which is a measure of the distance between the source's entropy and the compression ratio, and can thereby be seen as a measure of "how fast" the algorithm reaches the source's entropy.

All these measures are very interesting but unrealistic because it is actually quite unusual, if not impossible, to know the entropy of the source which generated the string we're going to compress. In order to circumvent this problem a different empirical approach has been taken by introducing

[1] Actually the GIF format can also present the lines in an *interleaved* format, the details of which are out of the scope of this brief discussion; the compression algorithm is however the same.

the notion of *k-th order empirical entropy* of a string S, denoted by $\mathcal{H}_k(S)$. In Chapter 12 we talked about the case $k = 0$, which depends on the frequencies of the individual symbols occurring in S. With $\mathcal{H}_k(S)$ we wish to empower the entropy definition by considering the frequencies of k-grams in S, thus taking into account sequences of symbols, hence the *compositional structure* of S.

More precisely, let S be a string over an alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_h\}$, and let us denote by n_ω the number of occurrences of the substring ω in S. We use the notation $\omega \in \Sigma^k$ to specify that the length of ω is k. Given this notation, we can define

$$\mathcal{H}_k(S) = \frac{1}{|S|} \sum_{\omega \in \Sigma^k} \left(\sum_{i=1}^{h} n_{\omega\sigma_i} \log\left(\frac{n_\omega}{n_{\omega\sigma_i}}\right)\right) \tag{13.1}$$

A compression algorithm is then defined to be *coarsely optimal* iff, for all k there exists a function $f_k(n)$ tending to 0 as $n \to \infty$ and such that, for all sequences S of increasing length, it holds that the compression ratio of the evaluated algorithm is at most $\mathcal{H}_k(S) + f_k(|S|)$.

Plotnik *et al.* [6] proved the coarse optimality of LZ78; Kosaraju and Manzini [3] noticed that the notion of coarse optimality does not necessarily imply a good algorithm because, if the entropy of the string S approaches zero, the algorithm can compress badly. This observation makes the par with the one we made for Huffman, related to the extra-bit needed for each encoded symbol. That extra-bit was ok for large entropies, but it was considered bad for entropies approaching 0.

LEMMA 13.1 There exist strings for which the compression ratio achieved by LZ78 is at least $g(|S|)\mathcal{H}_0(S)$, with $g(n)$ such that $\lim_{n\to\infty} g(n) = \infty$.

Proof Consider the string $S = 01^{n-1}$, which has entropy $\mathcal{H}_0(S) \in \Theta(\frac{\log n}{n})$. It is easy to see that LZ78 parses S with $\Theta(\sqrt{n})$ phrases. Thus we get $g(n) = \frac{\sqrt{n}}{\log n}$.

To circumvent these inefficiencies, Kosaraju and Manzini introduced a stricter version of optimality, called λ-optimality: it applies to an algorithm whose compression ratio can be bounded by $\lambda\mathcal{H}_k(S) + o(\mathcal{H}_k(S))$. As the previous lemma clearly demonstrates, LZ78 is not λ-optimal, however there exists a modified version of LZ78 combined with run-length compression (RLE) that is 3-optimal with respect to $\mathcal{H}_0$, but cannot be λ-optimal for any $k \geq 1$.

Let us now turn our attention to LZ77, which seems more powerful than LZ78, given that its dictionary is larger. The practical variant of LZ77 that uses a fixed-size compression window is not much good, and actually worse than LZ78:

LEMMA 13.2 The LZ77 algorithm, with a bounded sliding window, is not coarsely optimal.

Proof We will show that, for each size L of the sliding window, we can find a string S for which the compression ratio exceeds the k-th order entropy. Consider in fact the string $(0^k1^k)^n1$ of length $2kn + 1$, where we choose $k = L - 1$. Due to the sliding window, LZ77 parses S in the following way:

$$\underline{0}\ \underline{0^{k-1}1}\ \underline{1^{k-1}0}\ \underline{0^{k-1}1} \ldots \underline{1^{k-1}0}\ \underline{0^{k-1}1}\ \underline{1^k}$$

Every phrase has then length up to k, splitting the input in $\Theta(n)$ phrases. In order to compute $\mathcal{H}_k(S)$ we need to work on all different k-length substrings of S, which are $2k$: $\{0^i1^{k-i}\}_{i=1\ldots k} \cup \{1^i0^{k-i}\}_{i=1\ldots k}$. Now, all strings in the form 0^i1^{k-i} are always followed by a 1. Similarly, for $i < k$ all strings 1^i0^{k-i}

are always followed by a 0. Only the string 1^k is followed $n-1$ times by a 0, and once by a 1. So we can split the sum over the k-grams ω within the definition of $\mathcal{H}_k(S)$ into 4 parts:

$$\begin{array}{lll} \omega \in \{0^i 1^{k-i}\}_{i=1\ldots k} & \rightarrow n_{\omega 0} = 0 & n_{\omega 1} = n \\ \omega \in \{1^i 0^{k-i}\}_{i=1\ldots k-1} & \rightarrow n_{\omega 0} = n & n_{\omega 1} = 0 \\ \omega = 1^k & \rightarrow n_{\omega 0} = n-1 & n_{\omega 1} = 1 \\ \text{else} & \rightarrow n_{\omega 0} = 0 & n_{\omega 1} = 0 \end{array}$$

It is now easy to calculate that

$$|S|\,\mathcal{H}_k(S) = \log n + (n-1)\log\frac{n}{n-1} \in \Theta(\log n)$$

and the lemma follows.

Nevertheless it does exist a modified LZ77, with no sliding window, which is coarsely optimal and also 8-optimal with respect to $\mathcal{H}_0$. However it is not λ-optimal for any $k \geq 1$:

LEMMA 13.3 There exist strings for which the compression ratio of LZ77, with no sliding window, which is at least $g(|S|)\,\mathcal{H}_1(S)$, with $g(n)$ such that $\lim_{n\rightarrow\infty} g(n) = \infty$.

Proof Consider the string $10^k\,2^{2^k}\,1\;101\;10^2 1\;10^3 1\ldots 10^k 1$ of length $2^k + O(k^2)$, and compression lower bound $|S|\,\mathcal{H}_k(S) = k\log k + O(k)$. The string is parsed with $k+4$ words:

$$\underline{1}\;\underline{0}\;\underline{0^{k-1}2}\;\underline{2^{2^k-1}1}\;\underline{101}\;\underline{10^2 1}\ldots\underline{10^k 1}$$

The problem is that the last k phrases refer back to the beginning of S, which is 2^k characters away. This generates $\Omega(k)$ long phrases, thus an overall output size of $\Omega(k^2)$.

So this variant of LZ77 is better than LZ78, as expected, but not yet good as we would like to for $k \geq 1$. The next chapter will introduce the Burrows-Wheeler Transform, proposed in 1994, which allows to surpass the inefficiencies of LZ-based methods by devising a novel approach to data compression which achieves λ-optimality, for very small λ and simultaneously for all $k \geq 0$. It is therefore not surprising that the BWT-based compressor `bzip2`, available in most Linux distributions, produces a more succinct output than `gzip`.

References

[1] Jon Bentley and Doug Mc Ilroy Data compression using long commons strings. *Proc. IEEE Data Compression Conference (DCC)*, 287-295, 1999.

[2] Richard Karp and Michael Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):319-327, 1987.

[3] S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *Siam Journal on Computing*, 29(3):893-911, 1999.

[4] Paolo Ferragina, Igor Nitto, and Rossano Venturini. On the bit-complexity of lempel-ziv compression. In *Procs of the ACM-SIAM Symposium on Algorithms (SODA)*, pages 768–777, 2009.

[5] Steven Pigeon. An optimizing lossy generalization of LZW. *Procs of IEEE Data Compression Conference*, 509, 2001.

[6] Eli Plotnik, Marcelo Weinberger and Jacob Ziv. Upper bounds on the probability of sequences emitted by finite-state sources and on the redundancy of the Lempel-Ziv algorithm. *IEEE Transactions on Information Theory*, 38:16-24, 1992.

[7] Kunihio Sadakane and Hiroshi Imai. Improving the speed of LZ77 compression by hashing and suffix sorting. *IEICE Trans. Fundamentals Vol. E83-A*, 2000.

[8] James A. Storer and Thomas G.Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928-951, 1982.

[9] The Graphics Interchange Format, v89a. *Compuserve Incorporated, Columbus, Ohio*, 1990.

[10] Terry A. Welch. A technique for high-performance data compression. *Computer*, 8-19, 1984.

[11] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337-343, 1977.

[12] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530-536, 1978.

14

The Burrows-Wheeler Transform

This chapter describes a lossless data compression technique devised by Michael Burrows and David Wheeler in 1994 at the DEC Systems Research Center. This technique was published in a Technical Report of the company [4, 8],[1] and since it was rejected from the Data Compression Conference (as Mike Burrows stated in its foreword to [9][2]), the two authors decided of not publishing their paper anywhere. Fortunately, Mark Nelson drew attention to it in a Dr. Dobbs article, and that was enough to ensure its survival.

A wonderful thing about publishing an idea is that a greater number of minds can be brought to bear on the surrounding problems. This is what happened around the Burrows-Wheeler Transform, whose studies exploded around the year 2000, leading me, Giovanni Manzini and S. Muthukrishnan to celebrate a ten-years-later resume in a special issue of *Theoretical Computer Science* [9]. In that volume, Mike Burrows again declined to publish the original TR but wrote a wonderful Foreword dedicated to the memory of David Wheeler, who passed away in 2004, and finally stated: *"This issue of Theoretical Computer Science is an example of how an idea can be improved and generalized when more people are involved. I feel sure that David Wheeler would be pleased to see that his technique has inspired so much interesting work."*

The so called *Burrows-Wheeler Transform* (or BWT) offered a revolutionary alternative to dictionary-based compressors and actually initiated a new family of compressors (such as bzip2 [18] or the booster [6]) as well as a new powerful family of compressed indexes (such as FM-index [7], and many variations [15]). In the following we will detail the BWT and the other two simple compressors, i.e. Move-To-Front and Run-Length Encoding, whose combination constitutes the bzip-based compressors. We will also briefly mention few theoretical issues about the BWT performance expressed in terms of the k-th order empirical entropy of the data to be compressed.

[1] M. Burrows: "In the technical report that described the BWT, I gave the year as 1981, but later, with access to the memory of his wife Joyce, we deduced that it must have been 1978."

[2] Years passed, and it became clear that David had no thought of publishing the algorithm—he was too busy thinking of new things. Eventually, I decided to force his hand: I could not make him write a paper, but I could write a paper with him, given the right excuse.

14.1 The Burrows-Wheeler Transform

The *Burrows-Wheeler Transform* (`BWT`) is not a compression algorithm *per se*, as it does not squeeze the input size, it is a *permutation* (and thus, a lossless transformation) of the input symbols which are laid down in a way that the resulting string is most suitable to be compressed via simple algorithms, such as *Move-To-Front coding* (shortly MTF) and *Run Length Encoding* (shortly RLE), both to be described in Section 14.2. This permutation forces some "locally homogeneous" properties in the ordering of the symbols that can be fully deployed, efficiently and efficaciously, by the combination `MTF + RLE`. A last statistical encoding step (e.g. Huffman or Arithmetic) is finally executed in order to eventually squeeze the output bit stream. All these steps constitute the backbone of any `bzip`-like compressor which will be discussed in Section 14.3.

The `BWT` consists of a pair of inverse transformations: a *forward transform*, which rearranges the symbols in the input string; and a *backward transform*, which somewhat magically reconstructs the original string from its `BWT`. It goes without saying that the invertibility of `BWT` is necessary to guarantee the decompression of the input file!

14.1.1 The forward transform

Let $s = s_1 s_2 ... s_n$ be an input string on n symbols drawn from an *ordered* alphabet Σ. We append to s a special symbol `$` which does not occur in Σ and it is assumed to be smaller than any other symbol in the alphabet, according to its total ordering.[3] The forward transform proceeds as follows:

1. Build the string $s\$$.
2. Consider the *conceptual* matrix $\mathcal{M}$ of size $(n+1) \times (n+1)$, whose rows contain all the cyclic left-shifts of string $s\$$. M is called the *rotation matrix* of s.[4]
3. Sort the rows of $\mathcal{M}$ reading them *left-to-right* and according to the ordering defined on alphabet $\Sigma \cup \{\texttt{\$}\}$. The final matrix is called $\mathcal{M}'$. Since `$` is smaller than any other symbol in Σ and, by construction, appears only once, the first row of $\mathcal{M}'$ is $\texttt{\$}s$.
4. Set $bw(s) = (\widehat{L}, r)$ as the output of the algorithm, where $\widehat{L}$ is the string obtained by reading the last column of M', sans symbol `$`, and r is the position of `$` there.

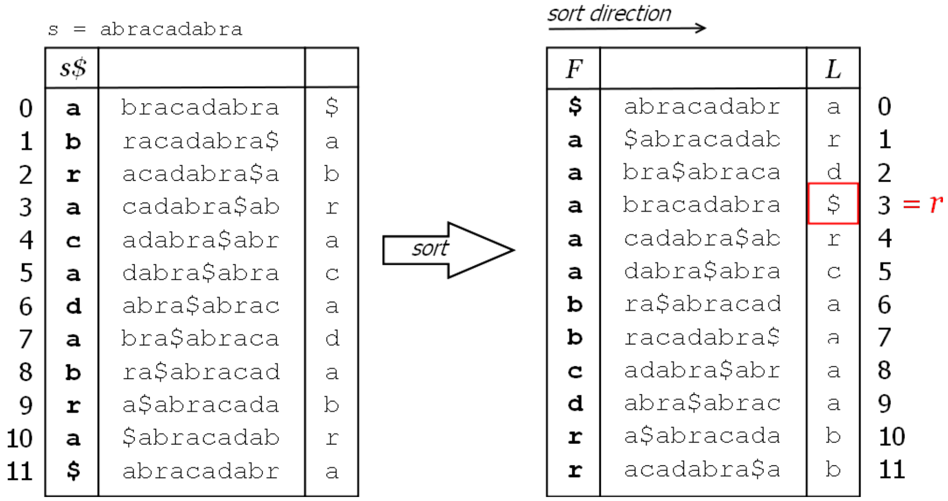
We said above that $\mathcal{M}$ is a conceptual matrix because we have to avoid its explicit construction, which otherwise would make the `BWT` an elegant mathematical object: the size of $\mathcal{M}$ is quadratic in $bw(s)$'s length, so the conceptual matrix has size $2^{48} \approx 1000$Tb just for transforming a string of 16Mb. In Section 14.3 we will actually show that M' can be built in time and space linear in the length of the input string s, by resorting Suffix Arrays.

An alternate enunciation of the algorithm, less frequent yet still present in the literature [19], constructs matrix $\mathcal{M}'$ by sorting the rows of $\mathcal{M}$ reading them *right-to-left* (i.e. starting from the last symbol of every row). Then, it takes the string $\widehat{F}$ formed by scanning the first column of $\mathcal{M}'$ top-to-bottom and, again, skipping symbol `$` and storing its position in r'. The output is then $bw(s) = (\widehat{F}, r')$. This enunciation is the dual of the one given above because it is possible to formally prove that both strings $\widehat{F}$ and $\widehat{L}$ exhibit the same *local-homogeneity* properties and thus compression, to be illustrated below. In the rest of the chapter we will refer to the left-to-right sorting of M's rows and to $(\widehat{L}, r)$ as the `BWT` of the string s, somehow forgetting the integer r.

[3] The step that concatenates the special symbol `$` to the initial string was not part of the original version of the algorithm as described by Burrows and Wheeler. It is here introduced with the intent to simplify the description.

[4] The left shift of a string $a\alpha$ is the string αa, namely the first symbol is moved to the end of the original string.

In order to better understand the power of the Burrows-Wheeler Transform, let us consider the following running example formulated over the string s = `abracadabra`. The left side of Figure 14.1 shows the rotated matrix $\mathcal{M}$ built over s; whereas the right side of Figure 14.1 shows sorted matrix M'. Because the first row of $\mathcal{M}$ is the only one to end with $, which is the lowest-ordered symbol in the alphabet, row `$abracadabra` is the first row of $\mathcal{M}'$. The other three rows of $\mathcal{M}'$ are the ones beginning with `a`, and then follow the rows starting with `b`, `c`, `d` and finally `r`, respectively.



	$s\$$		
0	a	bracadabra	$
1	b	racadabra$	a
2	r	acadabra$a	b
3	a	cadabra$ab	r
4	c	adabra$abr	a
5	a	dabra$abra	c
6	d	abra$abrac	a
7	a	bra$abraca	d
8	b	ra$abracad	a
9	r	a$abracada	b
10	a	$abracadab	r
11	$	abracadabr	a

F		L	
$	abracadabr	a	0
a	$abracadab	r	1
a	bra$abraca	d	2
a	bracadabra	$	3 $= r$
a	cadabra$ab	r	4
a	dabra$abra	c	5
b	ra$abracad	a	6
b	racadabra$	a	7
c	adabra$abr	a	8
d	abra$abrac	a	9
r	a$abracada	b	10
r	acadabra$a	b	11

FIGURE 14.1: Forward Burrows-Wheeler Transform of the string s = `abracadabra`.

If we read the first column of $\mathcal{M}'$, denoted by F, we obtain the string `aaaaabbcdr` which is the sorted sequence of all symbols in s. We finally obtain $\widehat{L}$ by excluding the single occurrence of $ from the last column L, so $\widehat{L}$ = `ardrcaaaabb`, and set $r = 3$.

The example is illustrative of the locally-homogeneous property we were mentioning before: the last 6 symbols of the last column of M form a highly repetitive string `aaaabb` which can be easily and highly compressed via the two simple compressors `MTF + RLE` (described below). The soundness of this statement will be mathematically sustained in the following pages, here we content ourselves by observing that this repetitiveness occurs not by chance but it is induced by the way M's rows are sorted (left-to-right) and texts are written down by humans (left-to-right). The nice issue here is that many real sources (they are called Markovian) do exist that generate data sequences, other than texts, that can be turned to be locally homogeneous via the Burrows-Wheeler Transformation, and thus can be highly compressed by `bzip`-like compressors.

14.1.2 The backward transform

We observe, both by construction and from the example provided above, that each column of the sorted cyclic-shift matrix $\mathcal{M}'$ (and also M) contains a permutation of s`$`. In particular, its first column F = `aaaaabbcdr` is alphabetically sorted and thus it represents the best-compressible transformation of the original input block. But unfortunately F cannot be used as `BWT` because it is not invertible: every text of length 11 and consisting of 5 occurrences of symbol `a`, 2 occurrences of `b`, 1 occurrence `c`, `d`, `r` respectively, originates a `BWT` whose F is the same as the one above.

The Burrows-Wheeler transform represents, in some sense, the best column of M' to be chosen as transformed s in terms of reversibility and compressibility of s.

In order to prove these properties more formally, let us define a useful function that tells us how to locate in $\mathcal{M}'$ the predecessor of a symbol at a given index in s.

FACT 14.1 *For $1 \leq i \leq n$, let $s[k_i, n-1]$ denote the suffix of s prefixing row i of $\mathcal{M}'$. Clearly, row i is then followed by symbol \$, and then by the prefix $s[1, k_i - 1]$ because of the leftward cyclic shift.*

For example in Figure 14.1, row 2 of $\mathcal{M}'$ is prefixed by `abra`, followed by `$abracad`.

Property 14.1 *The symbol $L[i]$ precedes the symbol $F[i]$ in the string s, except for the row i such that $L[i] =$ \$, in which case $F[i] = s[1]$.*

Proof Because of Fact 14.1 the last symbol of the row i is $L[i] = s[k_i - 1]$ and its first symbol is $F[i] = s[k_i]$. So the statement follows. ■

Intuitively, this property descends from the very nature of every row in $\mathcal{M}$ and $\mathcal{M}'$ that is a *left* cyclic-shift of $s\$$, so if we take two extremes of each row, the symbol on the right extreme (i.e. on L) is immediately followed by the one on the left extreme (i.e. on F) over the string s.

Property 14.2 *All the occurrences of a same symbol c in L maintain the same relative order as in F. This means that the kth occurrence in L of symbol c corresponds to the kth occurrence of the symbol c in F.*

Proof Given two strings t and t', we shall use the notation $t \prec t'$ to indicate that string t lexicographically precedes string t'.

Fix now the symbol c. If c occurs once in s then the proof derives immediately because the single occurrence of c in F obviously maps to the single occurrence of c in L. (Both columns are permutations of s.) To prove the more complicate situation that c occurs at least twice in s, let us fix two of these occurrences and pick their rows of the sorted matrix $\mathcal{M}'$, say $r(i)$ and $r(j)$ with $i < j$. We can observe few interesting things:

- row $r(i)$ precedes lexicographically row $r(j)$, given the ordering of $\mathcal{M}'$'s rows and the fact that $i < j$, by assumption;
- both rows $r(i)$ and $r(j)$ start with symbol c, by assumption;
- given that $r(i) = c\,\alpha$ and $r(j) = c\beta$, it is $\alpha \prec \beta$.

Since we are interested in the respective positions of those two occurrences of c when they are mapped to L, we consider the two rows $r(i')$ and $r(j')$ which are obtained by rotating those two rows leftward by one single symbol: $r(i') = \alpha\, c$ and $r(j') = \beta\, c$. This way, this rotation brings the first symbol $F[i]$ (resp. $F[j]$) into the last symbol $L[i']$ (resp. $L[j')$ of the rotated rows. Since $\alpha \prec \beta$, it is $r(i') \prec r(j')$ and so the preservation of the ordering in L holds true for that pair of occurrences of c. Given that this order-preserving property holds for every pair of occurrences of c in F/L, it holds true for all of them. ■

We have now all mathematical tools to design an algorithm which reconstructs s from its $bw(s) = (\widehat{L}, r)$ by exploiting the so called LF-mapping, an array of n integers in the range $[0, n-1]$.

Algorithm 14.1 Constructing the LF-mapping from column L

```
for $i = 0, 1, \ldots, n-1$ do
    $C[L[i]]$++;
end for
$temp = 0,\ sum = 0$;
for $i = 0, 1, \ldots, |\Sigma|$ do
    $temp = C[i]$;
    $C[i] = sum$;
    $sum$+= $temp$;
end for
for $i = 0, 1, \ldots, n-1$ do
    $LF[i] = C[L[i]]$;
    $C[L[i]]$++;
end for
```

DEFINITION 14.1 It is $LF[i] = j$ iff the symbol $L[i]$ maps to symbol $F[j]$. This way, if $L[i]$ is the kth occurrence in L of symbol c, then $F[LF[i]]$ is the kth occurrence of c in F.

Building LF is pretty straightforward for symbols that occur only once, as it is the case of `$`, `c` and `d` in s = `abracadabra$`, see Figure 14.1. But when it comes to symbols `a`, `b` and `r`, which occur several times in the string s, computing LF efficiently is no longer trivial. Nonetheless it can be solved in optimal $O(n)$ time thanks to Property 14.2, as Algorithm 14.1 details. This algorithm uses an auxiliary vector C, of size $|\Sigma| + 1$. For the sake of description, we assume that array C is indexed by a *symbol* rather than by a integer.[5]

The first `for`-cycle computes, for each symbol c, the number n_c of its occurrences in L, and stores $C[c] = n_c$. Then, the second `for`-cycle, turns these symbol-wise occurrences into a cumulative sum, so that the new $C[c]$ stores the total number of occurrences in L of symbols *smaller than* c, namely $C[c] = \sum_{x<c} n_x$. This is done by adopting two auxiliary variables, so that the overall working space is still $O(n)$. We notice that $C[c]$ gives the first position in F where symbol c occurs. Therefore, before the last `for`-cycle starts, $C[c]$ is the landing position in F of the first c in L (we thus know the LF-mapping for the first occurrence of every alphabet symbol). Finally, the last `for`-cycle scans the column L and, whenever it encounters symbol $L[i] = c$, then it sets $LF[i] = C[c]$. This is correct when c is met for the first time; then $C[c]$ is incremented so that the next occurrence of c in L will map to the next position in F (given the contiguities in F of all rows starting with that symbol). So the algorithm keeps the invariant that $LF[i] = \sum_{x<c} n_x + k$, after that k occurrences of c in L have been processed. It is easy to derive the time complexity of such computation which is $O(n)$.

Given the LF-mapping and the fundamental properties shown above, we are able to reconstruct s backwards starting from the transformed output $bw(s) = (\widehat{L}, r)$ in $O(n)$ time and space. Clearly it is easy from $bw(s)$ to construct L, just insert \$ at position r of $\widehat{L}$. The algorithm then picks the last symbol of s, namely $s[n-1]$, which can be easily identified at $L[0]$, given that the first row of M' is \$$s$. Then it proceeds by moving one symbol at a time to the left in s, deploying the two Properties above: Property 14.2 allows to map the current symbol occurring in L (initially $L[0]$) to its corresponding copy in F; then Property 14.1 allows to find the symbol which precedes that copy in F by taking the symbol at the end of the same row (i.e. the one in L). This double step, which

[5] Just implement C as a hash table, or observe that in practice any symbol is encoded via an integer (ASCII code maps to the range $0, \ldots, 255$) which can be used as its index in C.

Algorithm 14.2 Reconstructing s from $bw(s)$

```
Derive column $L$ from $bw(s)$;
Compute $LF[0, n-1]$ from $L$;
$k = 0$;  $i = n - 1$;
while $i \geq 0$ do
    $s[i] = L[k]$;
    $k = LF[k]$;
    $i--$;
end while
```

returns on L, allows to move one symbol leftward in s. Repeating this up to the beginning of s we are able to reconstruct this string. The pseudo-code is reported in Algorithm 14.2.

As an example, refer to Figure 14.1 where we have that $L[0] = s[n-1] = \mathtt{a}$, and execute the while-cycle of Algorithm 14.2. Definition 14.1 guarantees that $LF[0]$ points to the first row starting with a, this is the row 1. So that copy of a is LF-mapped to $F[1]$ (and in fact $F[1] = \mathtt{a}$), and the preceding symbol in s is thus $L[1] = \mathtt{r}$. These two basic steps are repeated until the whole string s is reconstructed. Just continuing the previous running example, we have that $L[1] = \mathtt{r}$ is LF-mapped to the symbol in F at position $LF[1] = 10$ (and indeed $F[10] = \mathtt{r}$). In fact, $L[1]$ and $F[10]$ is the first occurrence of symbol r in both columns L and F, respectively. The algorithm then takes as preceding symbol of r in s the symbol $L[10] = \mathtt{b}$. And so on...

THEOREM 14.3 *The original string s can be reconstructed from its* BWT *in $O(n)$ time and space. Algorithm 14.2 elicits possibly one cache-miss per symbol.*

Several recent results addressed the problem of reducing the number of cache misses as well as the working space of algorithms inverting BWT. Some progress has been made in the literature (see e.g. [17, 12, 13, 11]), but yet reductions are limited, e.g. small constants for the cache-misses, say $2 \div 4$, which get larger if the data is highly repetitive. Much has still to be discovered here!

14.2 Two other simple transforms

Let us now focus on two simple algorithms that come in very useful to design the compressor bzip2. These algorithms are called *Move-To-Front* (MTF) and *Run-Length Encoding* (RLE). The former maps symbols into integers, the latter maps runs of equal symbols into pairs. For the sake of completeness we observe that RLE is a compressor indeed, because the output sequence may be reduced in length in the presence of long runs of equal symbols; while MTF can be turned into a compressor by encoding the run-lengths via proper integer encoders [3]. In general the compression performance of those algorithms is very poor: BWT is magically their killer application!

14.2.1 The Move-To-Front transform

The MTF-transformation [3] implements the idea that every symbol of a string s can be replaced with its index in a proper *dynamic* list $\mathcal{L}^{MTF}$ containing all alphabet symbols. The string produced in output, denoted hereafter as s^{MTF}, is initialized to the empty string and contains as symbols integers in the range $[0, |\Sigma| - 1]$. At each step i, we process the symbol $s[i]$ and find its position p in $\mathcal{L}^{MTF}$. Then p is added to the string s^{MTF}, and $\mathcal{L}^{MTF}$ is modified by *moving* the symbol $s[i]$ to the *front* of the list.

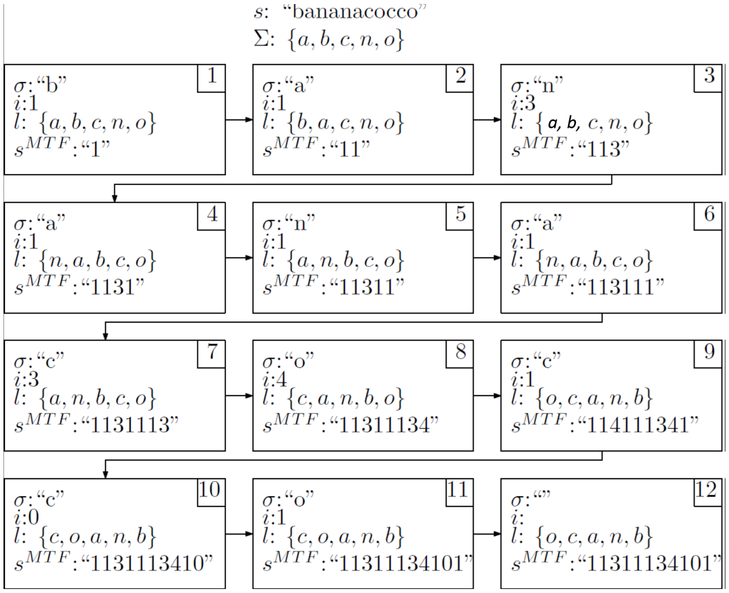


FIGURE 14.2: An example of MTF-transform over the string t = bananacocco, alphabet $\Sigma = \{a, b, c, n, o\}$ and thus index set $\{0, 1, 2, 3, 4\}$.

It is greatly advantageous to apply this processing over the column L of $bw(s)$ because, as it will be clear next, it transforms *locally homogeneous substrings* of L into a *globally homogeneous string* L^{MTF} in which abound small integers. At this point we could apply any integer compressor, described in Chapter 11, instead the bzip deploys the structural properties of L^{MTF} to apply, in cascade, RLE and finally a Statistical encoder (such as Huffman, Arithmetic, or some of their variations, see Chapter 12).

Figure 14.2 shows a running example for MTF over the string t = bananacocco which consists of 5 distinct symbols. It is evident that more frequent symbols are to the front of the list $\mathcal{L}^{MTF}$ and thus get smaller indices in s^{MTF}; this is the principle exploited in [3] to prove some compressibility bounds for the compressor that applies δ-coding over the integers in s^{MTF} (see Theorem 14.5 below).

We notice two local homogeneous substrings in s— "anana" and "cocco"— which show individually some redundancy in a few symbols. This is turned by MTF into two substrings of s^{MTF} consisting of small integers. The nice thing of the MTF-mapping is that homogeneous substrings which possibly involve different symbols (such as $\{a, n\}$ and $\{c, o\}$ in our running example), are changed into the homogeneous string s^{MTF} = 11311134101 which involves small numbers (mostly 0s and 1s) and is thus defined over a unique (integer) alphabet. The strong local-homogeneity properties of the column L in $bw(s)$ will make L^{MTF} full of 0s, so that the use the *single and simple compressor* RLE is worth and effective.

Inverting s^{MTF} is easy provided that we start with the same initial list $\mathcal{L}^{MTF}$ used for the MTF-transformation of s. A running example is provided in Figure 14.3. The algorithm maps an integer i in s^{MTF} onto the symbol which occurs at position i in $\mathcal{L}^{MTF}$, and then *moves* that symbol *to the front* of the list. This way the inversion algorithm mimics the transformation algorithm, by keeping both MTF-lists synchronized.

THEOREM 14.4 *Transforming a string s via* MTF *takes $O(|s|)$ time and $O(|\Sigma|)$ working space.*

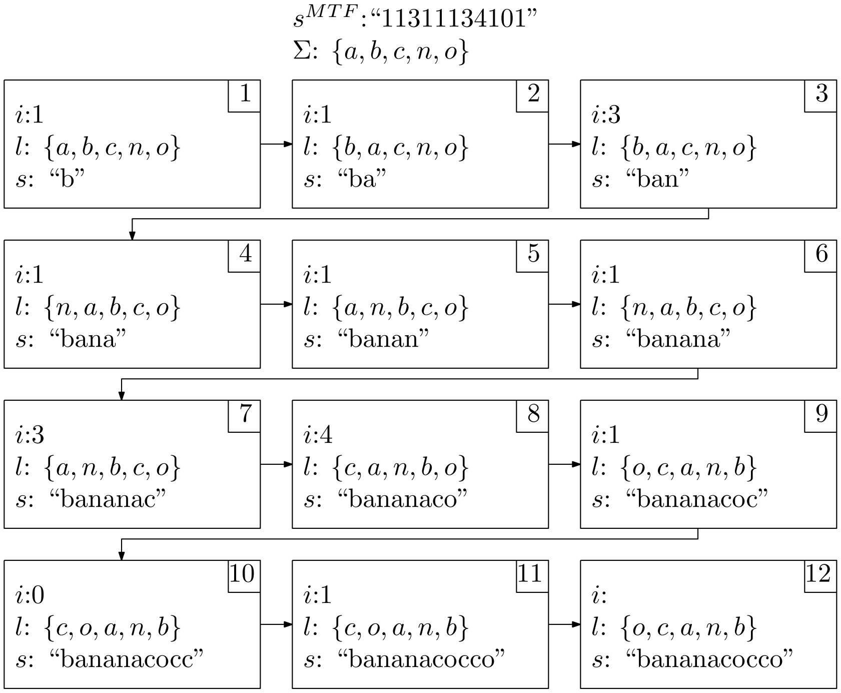


FIGURE 14.3: An example of MTF-inversion over the string s^{MTF} = `11311134101`, starting with the list $\mathcal{L}^{MTF} = \{a, b, c, n, o\}$.

A key concept for evaluating the compression performance of `MTF` is the one named *locality of reference*, which we have previously called *locally homogeneous substrings*. Locality of references in s means that the distance between consecutive occurrences of the same symbol are small. For example the string `bananacocco` shows this feature in the substrings `anana` and `cocco`. We are perfectly aware that this concept is roughly specified but, for now, let us stick onto this abstract formulation which we will make mathematically precise next.

If the input string s exhibits locality of references, then the `MTF`-compressor (namely one that `MTF`-transforms s and then compresses someway the integers in s^{MTF}) performs better than the Huffman's compressor. This might appear surprising because Huffman's compressor is an *optimal prefix-code*; but, actually this is not surprising, because the `MTF`-compressor is not a prefix-code given that a symbol may be *dynamically* associated to different codewords. As an example look at Figure 14.2 and notice that symbol `c` gets three different numbers in s^{MTF}— i.e. $3, 1, 0$— and thus three different codewords.

Conversely, if the input string s does not exhibits any kind of locality of reference (e.g. it is a (quasi-)random string over the alphabet Σ), then the `MTF`-compressor performs much worse than Huffman's compressor. The following theorem (proved in [3]) makes this rough analysis precise by combining the `MTF`-transform with the γ-code. It goes without saying that the upper bound stated below could be made closer to the entropy H by substituting the γ-code with the δ-code or any other better universal compressor for integers (see Chapter 11).

THEOREM 14.5 *Let n_c be the number of occurrences of a symbol c in the input string s, whose total length is $n = |s|$. We denote by $\rho_{MTF}(s)$ the average number of bits per symbol used by the* *compressor that squeezes the string s^{MTF} using the γ-code over its integers. It is $\rho_{MTF}(s) \leq 2H + 1$, namely that compressor can be no more than twice worse than the entropy of the source, and thus it cannot be more than twice worse than the Huffman compressor.*

Proof Let $p_1, \ldots, p_{n_c}$ be the positions in s where symbol c occurs. Clearly, between any two consecutive occurrences of c in s, say p_i and p_{i-1}, there may exist no more than $p_i - p_{i-1}$ distinct symbols (including c itself). So the index encoded by the MTF-compressor for the occurrence of c at position p_i is at most $p_i - p_{i-1}$. In fact, when processing position p_{i-1} the symbol c is moved to the front of the list, then it can move (at most) one position back per symbol processed subsequently, until we reach the occurrence of c at position p_i. This means that the integer emitted for the occurrence of c at position p_i is $\leq p_i - p_{i-1}$ (number of symbols processed). This integer is then encoded via γ-code, thus using no more than $\gamma(p_i - p_{i-1}) \leq 2(\log_2(p_i - p_{i-1})) + 1$ bits. As far as the first occurrence of c is concerned, we can assume that $p_0 = 0$, and thus encode it with at most $\gamma(p_1) \leq 2(\log_2 p_1) + 1$ bits. Overall the cost in bits for storing the occurrences of c in string s is

$$\begin{aligned} &\leq \gamma(p_1) + \sum_{i=2}^{n_c} \gamma(p_i - p_{i-1}) \\ &\leq 2\log_2(p_1) + 1 + \sum_{i=2}^{n_c} (2\log_2(p_i - p_{i-1}) + 1) \\ &\leq \sum_{i=1}^{n_c} (2\log_2(p_i - p_{i-1}) + 1). \end{aligned} \tag{14.1}$$

By applying Jensen's inequality we can move the logarithm function outside the summation, so that a telescopic sum comes out:

$$\begin{aligned} &\leq n_c \left(2\ \log_2\left(\frac{1}{n_c}\left(\sum_{i=1}^{n_c} (p_i - p_{i-1})\right)\right) + 1\right) \\ &= n_c \left(2\ \log_2\left(\frac{p_{n_c}}{n_c}\right) + 1\right) \\ &\leq n_c \left(2\ \log_2\left(\frac{n}{n_c}\right) + 1\right) \end{aligned} \tag{14.2}$$

where the last inequality comes from the simple observation that $p_{n_c} \leq n$. If now we sum for every symbol $c \in \Sigma$ and divide for the string length n, because the Theorem is stated as number of bits per symbol in s, we get:

$$\rho_{MTF}(s) \leq 2\left(\sum_{c\in\Sigma} \frac{n_c}{n} \log_2\left(\frac{n}{n_c}\right)\right) + 1 \ \leq \ 2H + 1 \tag{14.3}$$

The thesis follows because H lower bounds the average codeword length of Huffman's code. ■

There do exist cases for which the MTF-based compressor performs much better than Huffman's compressor.

LEMMA 14.1 The compressor based on the combination of MTF-transform and γ-code can be better than Huffman compressor by the unbounded factor $\Omega(\log n)$, where n is the length of the string to be compressed.

Proof Take the string $s = 1^n 2^n \cdots n^n$ defined over an alphabet of size n and having length $|s| = n^2$. Since every symbol occurs n times, the distribution is uniform and thus Huffman uses for each symbol $\log_2 n$ bits. The overall compression of s by Huffman takes $\Theta(|s| \log n) = \Theta(n^2 \log n)$ bits. We used the asymptotic notation because constants here do not matter.

If we adopt the MTF-transform we get the string $s^{MTF} = 0^n\, 10^{n-1}\, 20^{n-1}\, 30^{n-1} \cdots$. Applying the γ-code, with the warning that integer i is encoded as $\gamma(i+1)$ since i may be null, we get an output bit sequence of length $O(n^2 + n \log n)$. This is due to the fact that the $\Theta(n^2)$ integers equal to 0 are encoded as $\gamma(1) = 1$, thus taking 1 bit, whereas all other integers (they are $n-1$ and smaller than n) are encoded with $O(\log n)$ bits each. ∎

14.2.2 The RLE transform

This is a very simple transform which maps every maximal contiguous substring of ℓ occurrences of symbol c into a pair $\langle \ell, c \rangle$. As an example, suppose we have to compress the following string which represents a line of pixels of a monochromatic bitmap (where W stands for "white" and B for "Black").

$$WWWWWWWWWWWBWWWWWWWWWWWWBBBBBWWWWWW$$

We can take the first block of W and compress in the following way:

$$\underbrace{WWWWWWWWWWW}_{\langle 11, W \rangle}BWWWWWWWWWWWWBBBBBWWWWWW$$

We can proceed in the same way until the end of the line is encountered, thus obtaining the sequence of pairs $\langle 11, W \rangle, \langle 1, B \rangle, \langle 12, W \rangle, \langle 5, B \rangle, \langle 6, W \rangle$. It is easy to see that the encoding is lossless and simple to reverse. A remarkable observation is that if $|\Sigma| = 2$, as in the previous example, we can simply emit individual numbers (which indicate the run length) rather than pairs, plus the first symbol of the string to compress (W in the example), and still be able to decode back to the original string. In the previous example we could emit: $W, 11, 1, 12, 5, 6$.

RLE is actually more than a transform because it can be turned into a simple compressor by combining it with an integer encoder (as we did for MTF). Its best known context of application is fax transmission [19]: a sheet of paper is viewed as a binary (i.e. monochromatic) bitmap, this bitmap is first transformed by XORing two consecutive lines of pixels, then every output line is RLE-transformed and, finally, the integers are compressed via Huffman or Arithmetic (recall that in binary images, the alphabet has size two). Provided that the paper to be faxed is pretty regular, the XORed lines will be full of 0s, and thus their RLE-transformation will originate few runs, whose compression will be significant. Nothing prevents to apply this argument to colored images, but the XORing of contiguous lines will get less 0s. More sophisticated methods are needed in this setting!

RLE can perform better or worse than the Huffman scheme: this depends on the message we want to encode. The following lemma shows that RLE can be much better than Huffman, by adopting the same string we used to prove Lemma 14.1.

LEMMA 14.2 The compressor based on the combination of the RLE-transform and the γ-code can be better than Huffman's compressor by the unbounded factor $\Omega(n)$, where n is the length of the string to be compressed.

Proof Take the string $s = 1^n 2^n \cdots n^n$, and recall from the proof of Lemma 14.1 that Huffman's code takes $\Theta(n^2 \log n)$ bits to compress it. If we apply the `RLE`-transform on the string s we get the string $s^{RLE} = \langle 1, n\rangle \langle 2, n\rangle \langle 3, n\rangle \cdots \langle n, n\rangle$. The γ-code over the integers of s^{RLE} will use $O(\log n)$ bits per pair and thus $O(n \log n)$ bits overall. ■

But there are cases, of course, in which `RLE`-compressor can perform much worse than Huffman's. Just consider a string s in which runs are short, namely any English text!

14.3 The `bzip` compressor

As we anticipated in the previous sections, the compressor `bzip` hinges on the sequential combination of three transforms— i.e. `BWT`, `MTF` and `RLE`— which produce an output that is suitable to be highly squeezed by a classical statistical compressor— such as Huffman, Arithmetic, or some of their variations. The most time consuming step in this sequence is the computation/inversion of the `BWT`, both at compression/decompression time respectively. This is not just in terms of number of operations, which are $O(n)$ for all transforms and the statistical compressor, but because of the pattern of memory accesses that is very scattered thus inducing a lot of cache misses. This is an issue that we will comment more deeply next.

The key property that makes `bzip` work well is the local homogeneity of the string produced by the Burrows-Wheeler transform. To convince yourself of this property let us consider the input string s and one of its substrings w, which is assumed to occur n_w times in s. Say $c_1, \ldots, c_{n_w}$ are the symbols preceding the occurrences of w in s. Now given the way $bw(s)$ is computed, we can conclude that all rows prefixed by the substring w in M' (they are of course n_w) are contiguous, but possibly shuffled depending on the symbols which follow w in each of those rows. In any case, the symbols c_i which precede w are contiguous in L (shuffled, accordingly), and thus constitute a substring of L. If the string s is Markovian, in the sense that symbols are emitted based on their previous ones (like linguistic texts), then the symbols c_i are expected to be a few distinct ones, and this property holds the more the longer is w. Given that w can be of any length, we say that L is locally homogeneous because, as we observed, picking any of its substrings it will possibly show few distinct symbols. This homogeneity is the core property that makes the subsequent steps in `bzip` very effective in compressing L.

For the sake of clearness, let us consider the following example which runs `bzip` over the string s defined as the concatenation for three times of the string `mississippi`. This way a high repetitiveness is induced over s. The first step consists of computing $bw(s)$, for space reasons we do not detail this computation but just show the result that can be checked by hands: $L =$ `ippp ssss ssmm miip ppii isss sssi iiii i`, where groups of 4 symbols simplify the reading, and $r = 15$ (counting from 0). The next step is to apply the `MTF`-transform to L starting with a list $\mathcal{L}^{MTF} = \{i, m, p, s\}$ which consists of the distinct symbols appearing in s. The storage of r (using 4-8 bytes) and of $\mathcal{L}^{MTF}$ (plainly) occurs in the preamble of the compressed file. The result of `MTF` is the string $L^{MTF} =$ `0200 3000 0030 0303 0010 0300 0001 0000 0`. Notice that runs of equal symbols generate runs of 0, except for the first symbol of the run which is mapped to an integer which represents its position in $\mathcal{L}^{MTF}$ at the time of its processing.

The first specialty introduced by `bzip` is that `RLE` in not applied onto L^{MTF} but on a slightly different string in which all numbers, except 0, are increased by one: $L^{MTF+} =$ `0300 4000 0040 0404 0040 0400 0002 0000 0`. The ratio behind this change relies on the way runs of 0 are encoded. In fact `bzip` does not apply `RLE` to runs of all possible symbols, rather it applies a restricted variant, called `RLE0`, which squeezes only the runs consisting of 0s. So the construction of L^{MTF+}, instead of L^{MTF}, can be looked as a smart way to reserve the integers 0 and 1 for the binary encoding of the 0-runs. More precisely, the run `00000` consisting of 5 occurrences of 0s is encoded according to the following scheme, known as Wheeler's code: the length is increased by 1, hence $5 + 1 = 6$, then the binary encoding of 6 is returned, hence `110`, and finally the first bit (surely 1) is removed thus outputting the binary sequence `10`. The first increment guarantees that the (increased) run-length is at least 2, and thus it is represented in at least 2 binary digits in which the first one is surely a 1. So the 1-bit removal leaves at least one bit to be output. Decoding Wheeler's code is easy, just repeat the above steps in reverse order.

The key property of Wheeler's code is that the output bit sequence consists of no more digits than numbers in L^{MTF+}, so this step can be considered as a preliminary compression, which is more and more effective as longer and longer are the 0-runs in L^{MTF+}. The binary output for the sequence of our running example above is: `RLE0 = 0314 1041 4031 4141 0210`. It is evident that the decompressor can easily identify the run's encodings because they consist of maximal sequences of 0s and 1s; recall that these numbers have been reserved explicitly for this purpose.

Finally `RLE0` is compressed by using a Statistical compressor that operates on an alphabet which consists of integers in the range $[0, |\Sigma|]$. We observe that the alphabet size is $|\Sigma| + 1$, rather than $|\Sigma|$, because of the increment we did onto the non-null numbers in L^{MTF} to derive L^{MTF+}. The reader can look at the home page of `bzip2` [18] for further details, especially regarding the statistical-encoding step.

Just to have an idea of the power of the BW-Transform, we report here few experiments that compare a BWT-based compressor[6] against a few other well-known compression algorithms such as LZMA (Lempel-Ziv-Markov chain algorithm)[7], LZO1A (LZ-Oberhumer zip)[8], and the classic ZIP[9]. Tests were run in a commodity PC with 2GB RAM (using ramfs), AMD Athlon(tm)X2 Dual-Core QL-64, running Linux. We used three datasets of different type and size: La Divina Commedia, a raw monochromatic non-compressed image, and the package `gcc-4.4.3`. These experiments are not intended to provide an official comparison among these compressors, rather to give the reader a flavor of the differences in performance among them.

From Figures 14.4–14.6 we can easily draw some conclusions. First of all, `LZMA` is bad on these datasets; it has ever the worst compression time and it does not reach the best compression rate. `bzip2` seems to be quite in the middle: sometimes it takes a lot of time to compress, but it reaches the best compression rate. By considering the decompression time, `bzip2` slows down too much as the size of the file grows, and this is not a surprise because of its algorithmic structure. Perhaps the best solution seems to be `zip`: it takes short time to compress/decompress and reaches a very good compression rate. `LZO` is the fastest algorithm we tested, but unfortunately its compression ratio seems to be not appealing, and this is due to the fact that it was engineered for speed rather than for space savings. We restate here that these considerations are not definitive for those compressors, they are just suitable for giving a glimpse on them about these three datasets. For more official and robust comparisons we refer the reader to the page of Matt Mahoney.[10]

We are left with the problem of constructing the Burrows-Wheeler forward transform given that, as we observed above, we cannot construct explicitly the rotation matrix $\mathcal{M}$, and a fortiori its sorted version $\mathcal{M}'$, because this would take $\Theta(n^2)$ working space for a text s of length n. That is the why most `BWT`-based compressors exploit some "tricks" in order to avoid the construction of these matrices. One such "trick" involves the usage of Suffix Arrays, which were described in Chapter 10,

[6] We used `http://www.bzip.org, version 1.0.5-r1`

[7] We used the "lzip" package from `http://www.nongnu.org, version 1.10`

[8] We used the version `1.02_rc1-r1`. LZO1A takes care about long matches and long literal runs so that it produces good results on high redundant data and deals acceptably with non-compressible data.

[9] We used the package from `http://www.info-zip.org/`, version 3.0.

[10] `http://mattmahoney.net/dc/dce.html`

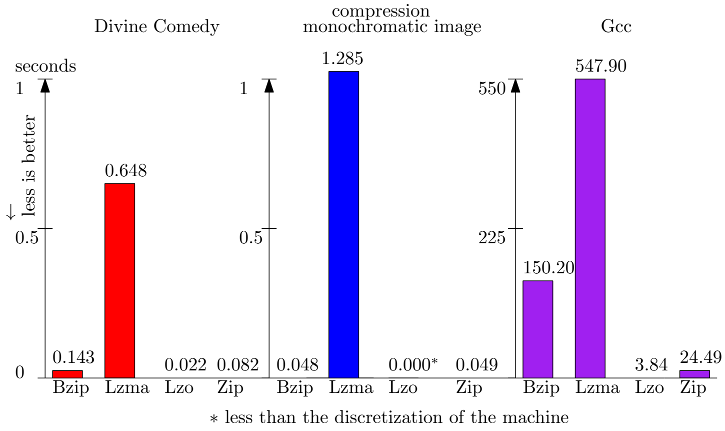


FIGURE 14.4: Compression speed (lower is better)

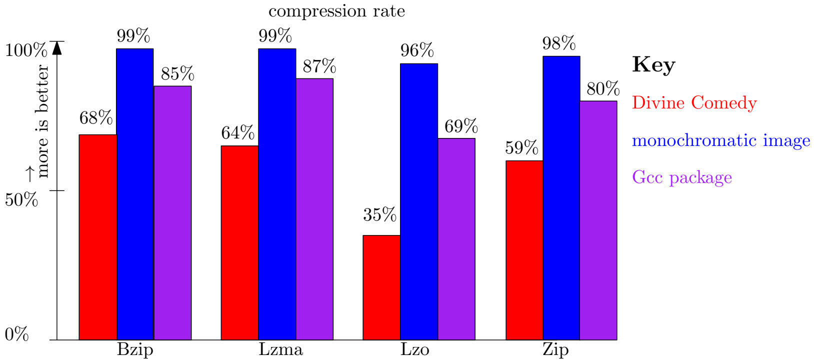


FIGURE 14.5: Compression savings (higher is better)

where we also detailed several algorithms to build them efficiently. The construction of BWT deploys one of them[11] and this use motivates the increased interest in the literature about the Suffix-Array construction problem (see e.g. [14, 16, 1]).

To see why Suffix Arrays and BWT are connected, let us consider the following example. Take the string abracadabra$ and compute its Suffix Array $[11, 10, 7, 0, 3, 5, 1, 4, 6, 9, 2]$. Figure 14.7 summarizes these data structures for the running example at hand. The first four columns show the suffixes of the string s and its suffix array $\mathcal{SA}$. The fifth column shows the corresponding sorted-rotated matrix $\mathcal{M}'$ with its last column L. It is easy to notice that sorting suffixes is equivalent to sorting rows of $\mathcal{M}$, given the presence of the sentinel symbol $. The reader can check that the formula below ties $\mathcal{SA}$ with L:

[11] M. Burrows: “So I enlisted his help in finding ways to execute the algorithm’s sorting step efficiently, which involved considering constant factors as much as asymptotic behavior. We tried many things, only some of which made it into the paper, but we met my goals: we showed that the algorithm could be made fast enough to see practical use on modern machines...”.

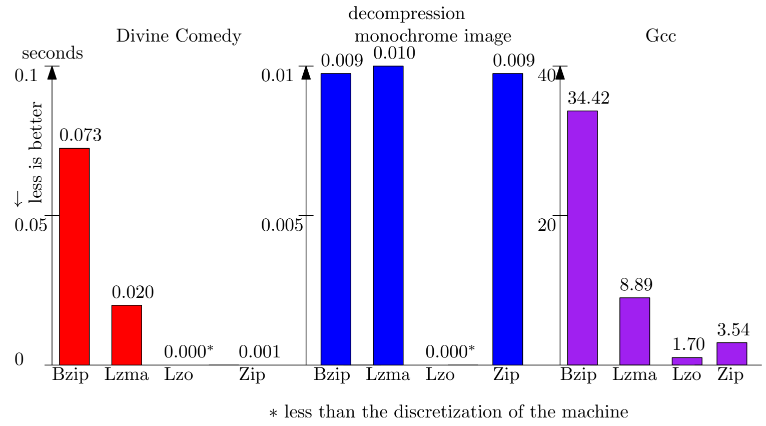


FIGURE 14.6: Decompression speed (lower is better)

suffix	index	sorted suffix	value	$\mathcal{M}'$	L
abracadabra$	0	$	11	$abracadabra	a
bracadabra$	1	a$	10	a$abracadabr	r
racadabra$	2	abra$	7	abra$abracad	d
acadabra$	3	abracadabra$	0	abracadabra$	$
cadabra$	4	acadabra$	3	acadabra$abr	r
adabra$	5	adabra$	5	adabra$abrac	c
dabra$	6	bra$	8	bra$abracada	a
abra$	7	bracadabra$	1	bracadabra$a	a
bra$	8	cadabra$	4	cadabra$abra	a
ra$	9	dabra$	6	dabra$abraca	a
a$	10	ra$	9	ra$abracadab	b
$	11	racadabra$	2	racadabra$ab	b

FIGURE 14.7: Suffix Array *versus* sorted rotated matrix M' over the string $s = \texttt{abracadabra\$}$.

$$L[i] = \begin{cases} s[\mathcal{SA}[i]-1] & \text{if } \mathcal{SA}[i] \neq 0 \\ \$ & \text{otherwise} \end{cases}$$

This means that every symbol $L[i]$ equals to the symbol of s that precedes the suffix $\mathcal{SA}[i]$ which prefixes the ith row of M'. If, however, that suffix is the whole string s (thus $\mathcal{SA}[i] = 0$), then $ will be used as preceding symbol.

So, given the suffix array of string s, it takes only linear time to derive the string L. We have therefore proved the following:

THEOREM 14.6 *Let us given an input string s, constructing bw(s) takes a time/IO complexity which is the one of Suffix Array construction. By using the Skew Algorithm, the overall cost of building bw(s) is optimal in several model of computations. In particular, this is O(n) for the RAM model and O(*`Sort`*(n)) for the external-memory model, where* `Sort` *is the I/O-cost of sorting n atomic items in a model in which M is the size of the internal memory and B is the disk-page size.*

We conclude this section by observing that, in practice, $bw(s)$ is costly to be computed so that its implementations divide the input text into blocks and then apply the transform *block-wise*. This is the reason why these compressors are called *block-wise compressors*. Likewise dictionary-based compressors, the size of the block impacts onto the trade-off compression ratio *versus* compression speed; but, unlike dictionary-based compressors, this impacts unfavorably also onto the decompression speed which is slowed down when working on larger and larger blocks. Anyway, the current implementation of `bzip2` allows to specify the size of the block at compression time with the parameter `-1`, . . ., `-9`, that actually indicate a block of size 100Kb, . . ., 900Kb.

14.4 On compression boosting$^\infty$

Let us first recall the notion of entropy as a measure of uncertainty (or information) associated with a random source S emitting n symbols $\{x_1, \ldots, x_n\}$ with probabilities $p(x_i)$:

$$H(S) = \sum_{i=1}^{n} p(x_i) \times \frac{1}{\log p(x_i)}$$

The previous formula is often called 0th order entropy, and it is indicated with H_0, because it is computed with respect to the probabilities of the single symbols emitted by the source S, without exploiting any context (or equivalently, exploiting an empty context, hence of length 0). Given that we are dealing with compressors and real strings, most evaluations of their performance drop probabilities in favor of frequencies: hence $p(x_i)$ is the ratio between the number of occurrences of x_i in the input string s and the total length of s, say $|s|$. Clearly, in this setting any string containing $n/2$ symbols `a` and $n/2$ symbols `b` has entropy $H_0 = 1$ independently of the fact that it is either a random string or the regular string $\mathtt{a}^{n/2}\mathtt{b}^{n/2}$.

A more precise modeling of the information content of a string s (of of its uncertainty) can be obtained by measuring the entropy over blocks of k-symbols. This is called kth order (empirical) entropy of the string s, and can be computed as follows:

$$H_k(s) = \frac{1}{|s|} \sum_{w \in A^k} |w_s| \, H_0(w_s)$$

where w_s represents the set of all symbols that follow w in s. Clearly $H_k(s) \leq H_0(s)$, but it can be much smaller, and for $|s|$ and k that go to ∞ this value converges to the entropy of the source that emitted s.

We are interested in this formula because it suggests a way to design a compressor that achieves $H_k(s)$ starting from a compressor that achieves H_0 of its input strings. This kind of algorithm is called a *Compression Booster* because it is able to boost a compression performance up to H_0 into a compression performance up to H_k. The algorithmic tool to achieve this is, surprisingly, the Burrows-Wheeler Transform [6]. In order to illustrate this innovative and powerful idea, let us consider a generic 0-order statistical compressor C_0 whose performance, in bits per symbol, over a string t is bounded by $H_0(t) + f(|t|)$. We notice that function $f(|t|) = 2/(|t|)$ for the Arithmetic coding and it is $f(|t|) = 1$ for Huffman coding (see Chapter 12).

In order to turn C_0 into an effective kth order compressor C_k, we proceed as follows.

- Compute the Burrows-Wheeler Transform $bw(s)$ of the input string s.
- Take all possible substrings w of the string t, and partition the column L in such a way that L_w is formed by the last symbols of rows prefixed by w.
- Compress each L_w with C_0, and concatenated the output bit sequences by alphabetically increasing w (or, equivalently, by occurrence of L_w in L).

It is immediate to notice that L_w is a substring of L, and not a subsequence, because rows prefixed by w in M' are contiguous. Given the LCP-array of string s the partitioning of L takes linear time (see Chapter 10) and thus it does not impact onto the efficiency of the final compressor C_k. As far as the compression performance per symbol is concerned, we easily derive that it can be bounded as:

$$\frac{1}{|s|} \sum_{w_s \in \Sigma^k} |w_s| \, (H_0(w_s) + f(|w_s|) \; = \; H_k(s) + O(|\Sigma|^k)$$

where we have applied the definition of $H_k(s)$ onto the summation of the $H_0(w_s)$, and the fact that $f(|w_s|) \leq 1$. It is clear that the more effective is the 0-th order compressor, the more it is closer to H_0, the more evanishing is the term $f(|w_s|)$ and thus negligible is the additive term $O(|\Sigma|^k)$. In [6] the authors showed that one actually does not need to fix k, since it does exist a Compression Booster which identifies in optimal $O(|s|)$ time a partition of L which achieves a compression ratio which is better than the one obtained by C_k, for any possible $k \geq 0$. The algorithm is elegant and not much involved, but it would require some space to be described in sufficient details, so that we refer the interested reader to that paper.

14.5 On compressed indexing$^{\infty}$

We have already highlighted the bijective correspondence between the rows of the rotated matrix M and the suffixes of the string s, as well as the strong relationship between the string L and the suffix array built on s (see Figure 14.7). This is relationship is at the core of FM-index's design, which has been the first compressed full-text index to achieve efficient substring search and space occupancy up to the k-th order empirical entropy of the indexed string. Given these features we can look at the FM-index as the *compressed version* of a suffix array, or as the *searchable version* of `bzip`-compressed format. The nature of these notes does not allow to dig into the technical details of the FM-index, so in the rest of this chapter we will just fly over its technicalities and concentrate on the main algorithmic ideas; the interested reader may look at the seminal paper [7] and the survey [15].

In order to simplify the presentation we distinguish between three basic operations, which underlie the design of many search toolbox:

- `Count`(P) returns the *range of rows* $[first, last]$ in M (and thus suffixes in the suffix array) which are prefixed by the string P. The value $(last - first + 1)$ accounts for the number of these pattern occurrences.
- `Locate`(P) returns the *list* of all positions in s where P occurs (possibly unsorted).
- `Extract`(i, j) returns the substring $s[i, j]$ by accessing its compressed representation in FM-index.

For example, in Figure 14.7 for the pattern $P =$ `ab` we have $first = 2$ and $last = 3$ for a total of two occurrences. These two rows correspond, as the picture clearly illustrates, to the two suffixes $s[0,]$ and $s[7,]$ which are prefixed by P.

Let us start from the description of `Count`(P). The retrieval of the rows $first$ and $last$ is not implemented via a binary search, as it occurred in Suffix Arrays (see Chapter 10), but it uses a peculiar search method which deploys the column L, the array C (which counts in $C[c]$ the number of occurrences in s of all symbols smaller than c) and an additional data structure which supports efficiently the very basic counting `Rank`(c, k) which reports the number of occurrences of the symbol c in the string prefix $L[0, k - 1]$. All data structures L, C and `Rank` can be stored compressed and still retrieve efficiently their entries: namely access $L[i]$ or $C[c]$, or answer `Rank`(c, k). The literature offers many solutions for this problem (see e.g. some classic results [7, 10, 2, 15]), here we report some of them (possibly not the best ones at the time we write these notes):

Algorithm 14.3 Counting the occurrences of pattern $P[0, p-1]$ in s

```
$i = p - 1, c = P[p - 1]$;
$first = C[c], last = C[c + 1] - 1$;
while ($first \leq last$) and $i \geq 1$ do
    $c = P[i - 1]$;
    $first = C[c] + \mathtt{Rank}(c, first - 1)$;
    $last = C[c] + \mathtt{Rank}(c, last) - 1$;
    $i = i - 1$;
end while
return ($first, last$).
```

LEMMA 14.3 Let $s[0, n-1]$ be a string over alphabet Σ and let L be its BW-Transform.

- For $|\Sigma| = O(\mathtt{polylog}(n))$, there exists a data structure which supports `Rank` queries on L in $O(1)$ time using $nH_k(s) + o(n)$ bits of space, for any $k = o(log_{|\Sigma|}n)$, and retrieves any symbol L in the same time bound.
- For general Σ, there exists a data structure which supports `Rank` queries on L in $O(\log\log|\Sigma|)$ time, using $nH_k(s) + o(n\log|\Sigma|)$ bits of space, for any $k = o(log_{|\Sigma|}n)$, and retrieves any symbol of L in the same time bound.

This means that `Rank` can be implemented in constant, or almost constant time and in space which is very much close to the k-th order entropy of the string s we wish to index. The array C takes only $O(|\Sigma|)$ space, which is negligible for real alphabets. This means that this ensemble of data structures is very compact indeed.

We are left to show how this ensemble allows us to implement `Count`(P). Algorithm 14.3, usually called *backward search*, reports the pseudo-code of such implementation which takes $O(p)$ optimal time, working in p constant-time phases numbered from $p-1$ to 0. Each phase preserves the following invariant: *At the i-th phase, the parameter "first" points to the first row of the sorted rotated matrix M' prefixed by* $P[i, p-1]$ *and the parameter "last" points to the last row of M' prefixed by* $P[i, p-1]$. Initially the invariant is true by construction: $F[C[c]]$ is the first row in M' starting with c, and $F[C[c+1]-1]$ is the last row in M' starting with c (recall that rows are numbered from 0).[12] As running example take $P = \mathtt{ab}$, so at the beginning we have: $C[b] = 6$ and $C[b+1] = C[c] = 8$ in Figure 14.7 and thus $[6, 7]$ is the range of rows prefixes by b before that the backward-search starts.

At each subsequent phase, Algorithm 14.3 has found the range of rows $[first, last]$ prefixed by $P[i, p-1]$. Then it determines the new range of rows $[first, last]$ prefixed by $P[i-1, p-1] = P[i-1]\,P[i, p-1]$ by proceeding as follows. First it determines the first and last occurrence of the symbol $c = P[i-1]$ in the substring $L[first, last]$ by deploying the function `Rank` properly queried. Specifically `Rank`$(c, first - 1)$ counts how many occurrences of c precede position $first$ in L, and `Rank`$(c, last)$ counts how many occurrences of c precede position $last$ in L. These values are then used to compute the LF-mapping of those first/last occurrences of c. In fact Property 14.2 and Definition 14.1 imply the equality $LF[i] = C[L[i]] + \mathtt{Rank}(L[i], i)$. This means that the computation of the LF-mapping can occur efficiently and succinctly provided that we store compactly the data structure that implements `Rank`(c, k). For a formal proof that this mapping actually retrieves the new

[12] We adopt the shorthand notation that $C[c+1]$ is the entry storing the counting for the symbol following c in the alphabet.

range of rows $[first, last]$ prefixed by $P[i-1, p-1]$ we refer the reader to the seminal publication [7]. Here we make an example to convince experimentally the reader that everything works fine. Refer again to Figure 14.7 and consider, as before, the pattern $P = \mathtt{ab}$ and the range $[6, 7]$ of rows in M' prefixes by $P[2] = \mathtt{b}$. Now pick the previous pattern symbol $P[1] = \mathtt{a}$, Algorithm 14.3 computes $\mathtt{Rank}(a, 5) = 1$ and $\mathtt{Rank}(a, 7) = 3$ because $L[0, first-1]$ contains 1 occurrences of a and $L[0, last]$ contains 3 occurrences of a. So the algorithm computes the new range as: $first = C[a] + \mathtt{Rank}(a, 5) = 1 + 1 = 2$, $last = C[a] + \mathtt{Rank}(a, 7) - 1 = 1 + 3 - 1 = 3$, which is indeed the contiguous range of rows prefixed by the pattern $P = \mathtt{ab}$.

After the final phase (i.e. $i = 0$), $first$ and $last$ will delimit the rows of M' containing all the suffixes prefixed by P. Clearly if $last < first$ the pattern P does not occur in s. The following theorem summarizes what we have sketched.

THEOREM 14.7 *Given a string $s[0, n-1]$ drawn from an alphabet Σ, there exists a compressed index that takes $O(p \times t_{\mathtt{rank}})$ time to support $\mathtt{Count}(P[0, p-1])$, where $t_{\mathtt{rank}}$ is the time cost of a single* Rank *operation over the BW-transform L of string s. The space usage is bounded by $nH_k(s) + o(n \log |\Sigma|)$ bits, for any $k = o(\log_{|\Sigma|} n)$.*

The interesting corollary of the Theorem above is that, by plugging Lemma 14.3, we get an implementation of $\mathtt{Count}(P)$ which takes optimal $O(p)$ time and compressed space. However this solution suffers of I/O-inefficiency because every phase elicits some cache/IO misses due to the jumping around L and Rank. Several efforts have been dedicated in the literature to make FM-index cache-oblivious or cache-aware but yet an equally elegant solution for those issues is still missing.

Let us now describe the implementation of the location of pattern occurrences via procedure $\mathtt{Locate}(P)$. For a fixed parameter μ, we sample the rows i of M' which correspond to suffixes that start at positions of the form $pos(i) = j\mu$, for $j = 0, 1, 2, \ldots$. Each such pair $\langle i, pos(i)\rangle$ is stored explicitly in a data structure $\mathcal{P}$ that supports membership queries in constant time (on the *row*-component). Now, given a row index r, the value $pos(r)$ can be derived immediately if $r \in \mathcal{P}$ is a sampled row; otherwise, the algorithm computes $j = LF^t(r)$, for $t = 1, 2, \ldots$, until j is a sampled row and thus is found in $\mathcal{P}$. In this case, $pos(r) = pos(j) + t$. The sampling strategy ensures that a row in $\mathcal{P}$ is found in at most μ iterations, and thus the occ occurrences of the pattern P can be located via $O(\mu \times occ)$ queries to the Rank-data structure.

THEOREM 14.8 *Given a string $s[0, n-1]$ drawn from an alphabet Σ, there exists a compressed index that takes $O(\mu\ occ)$ time and $O(\frac{n}{\mu} \log n)$ bits of space to support $\mathtt{Locate}(P)$, provided that the range $[first, last]$ of rows prefixed by P is available.*

By fixing $\mu = \log^{1+\epsilon} n$, the solution above takes poly-logarithmic time per occurrence and sub-linear space (in bits). Trade-offs are possible and they were investigated [5].

Not much surprising is that $\mathtt{Count}(P)$ can be adapted to implement the last basic operation supported by FM-index: $\mathtt{Extract}(i, j)$. Let r be the row of M' prefixed by the suffix $s[j, n-1]$, and assume that the value of r is known. The algorithm sets $s[j] = F[r]$ and then starts a cycle which sets $s[j-1-t] = L[LF^t[r]]$, for $t = 0, 1, \ldots, j-i-1$. The main idea underlying this cycle is that we repeatedly compute the LF-mapping (implemented via the Rank-data structure) of the current symbol, so jumping backward in s starting from $s[j-1]$ (in fact $s[j]$ is found via F-array). We stop after $j-i-1$ steps, when we have reached $s[i]$. This approach reminds the one we have taken in BWT-inversion, the difference relies in the fact that the array LF is not explicitly available, but its entries are generated on-the-fly via Rank-computations. This guarantees still constant-time access to LF-array, but succinct space storage (thanks to Lemma 14.3).

Given the appealing asymptotical performance and structural properties of the FM-index, several authors have investigated its practical behavior by performing an extensive set of experiments. We invite the reader to check paper [5] and to look at the `Pizza&Chili`'s site[13] which offers many implementations of compressed indexes, not just FM-index. Experiments have shown that the FM-index is compact (its space occupancy is close to the one achieved by `bzip`), it is fast in counting the number of pattern occurrences (few micro-secs per pattern's symbol), and the cost of their retrieval is reasonable when they are few (about 100k occurrences/sec). In addition the FM-index allows to trade space occupancy for search time by choosing the amount of auxiliary information stored into it (i.e. parameter μ and few other parameters arising in the implementation of `Rank`). As a result the FM-index combines compression and full-text indexing: like `bzip` it encapsulates a compressed version of the original file (accessible via `Extract`); like suffix trees and suffix arrays it allows to search for arbitrary patterns (via `Count` and `Locate`). Everything works by looking only at a small portion of the compressed file, thus avoiding its full decompression.

References

[1] Donald Adjeroh, Tim Bell, and Amar Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer, 2008.

[2] Jérémy Barbay, Meng He, J. Ian Munro and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multi-labeled trees. *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 680-689, 2007.

[3] Jon L. Bentley, David D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communication of the ACM*, 29(4):320–330, 1986.

[4] Mike Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. *Technical Report 124*, Digital Systems Research Center (SRC), 1994.

[5] P. Ferragina, R. Gonzalez, G. Navarro, R. Venturini. Compressed Text Indexes: From Theory to Practice. *ACM journal on Experimental Algorithmics*, vol. 13, art. 12, Febbraio 2009.

[6] Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, Marinella Sciortino. Compression boosting in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005.

[7] Paolo Ferragina, Giovanni Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[8] Paolo Ferragina and Giovanni Manzini. Burrows-wheeler transform. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer US, 2008.

[9] Paolo Ferragina, Giovanni Manzini, S. Muthukrishnan, co-editors. *Special Issue on the Burrows-Wheeler Transform*. Theoretical Computer Science, 387(3), 2007.

[10] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(20), 2007.

[11] Juha Karkkainen, Dominik Kempa, and Simon J. Puglisi. Slashing the time for BWT inversion. *Proc. IEEE Data Compression Conference*, 99–108, 2012.

[12] Juha Karkkainen and Simon J. Puglisi. Medium-space algorithms for inverse BWT. *Proc. European Symposium on Algorithms*, LNCS 6346, 451–462, 2010.

[13] Juha Karkkainen and Simon J. Puglisi. Cache-friendly Burrows-Wheeler inversion.

[13] `http://pizzachili.dcc.uchile.cl/`

Proc. International Conference on Data Compression, Communication and Processing, 38–42, 2011.

[14] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[15] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[16] Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *Procs of the Prague Stringology Conference*, 1–30, 2005.

[17] Julian Seward. Space-time tradeoffs in the inverse B-W transform. In *Proc. IEEE Data Compression Conference*, 439–448, 2001.

[18] Julian Seward. The `bzip2` home page. `http://sources.redhat.com/bzip2`.

[19] Ian H. Witten, Alistair Moffat and Timothy C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition. Morgan Kaufmann, 1999.