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.

1

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, 7]. 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 1.1 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 1.1: 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 1.2.
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 [6].

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 1.2: 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 1.1 *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 1.2 *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*.

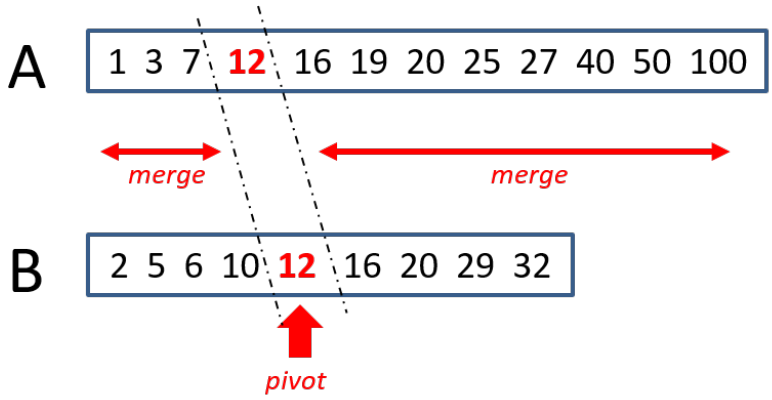

FIGURE 1.3: An example of the Intersection paradigm based on Mutual Partitioning: the pivot is 12, the median element of B.

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 1.1, and a running example is illustrated in Figure 1.3. 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 1.1 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 **??**, 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 1.3 *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 1.2 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 1.4 shows a running example, whereas Figure 1.2 shows the pseudo-code of the doubling search algorithm.

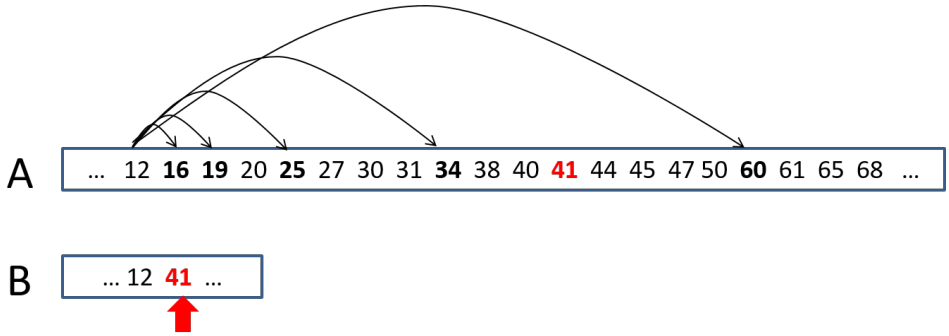

FIGURE 1.4: An example of the Doubling Search paradigm: the two sequences A and B are assumed to have been intersected up to the element 12. The next element in B, i.e. 41, is taken to be exponentially searched in the *suffix* of A following 12. This search checks A's elements at distances which are 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 1.2 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 1.4 *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 1.1 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 1.5 *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 and, surprisingly, also speed up performance. 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-compression 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. Moreover they can be easily decompressed if the algorithm proceeds by scanning rightward the sequence.

The Δ-compression scheme is clearly advantageous in space whenever the differences are much smaller than the universe size u from which the docIDs are taken. The distribution of docIDs 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 then apply the two-level approach above onto the permuted sequences.

More precisely, in the preprocessing phase, for each list A of length n we permute A according to the random permutation π and then assign its permuted elements to the buckets U_i according to their $\ell = \lceil \log_2 \frac{n}{L} \rceil$ most significant bits. Then 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.

In the query phase, the intersection of two sets A and B exploits the intuition that, since the permutation π is the same for all sets, if element $z \in A \cap B$ then $\pi(z)$ is the same for A and B. The algorithm proceeds as in the above two-level storage paradigm and thus solves the sorted set

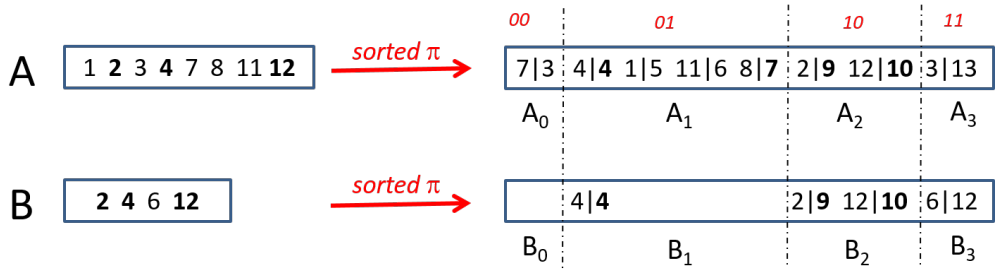

FIGURE 1.5: An example of the Random Permuting and Splitting paradigm. We assume universe $U = \{1, \ldots, 13\}$, set $L = 2$ and $n = 8$, and consider the permutation $\pi(x) = 1 + (4x \mod 13)$. So U is partitioned in $n/L = 4$ buckets identified by the most significant bits $\ell = \lceil \log_2 n/L \rceil = \lceil \log_2 4 \rceil = 2$ 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 most significant bits 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 most significant bits. 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, $\{2, 4, 12\}$.

intersection problem in $O(\frac{n}{L} + \min\{n, mL\})$ time *on average* (because of the random permuting) and $O(\frac{n}{LB} + \frac{mL}{B} + m)$ I/Os (see Fact 1.5). Note that we have available π^{-1}, so we can recover the original shared item z after having matched $\pi(z)$. A running example is shown in Figure 1.6.

As far as the space occupancy is concerned we notice that there are $\Theta(n/L)$ buckets for A, and the largest difference between two bucket entries can be bounded in two ways: the bucket width $O(uL/n)$ 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 $\log_2 \min\{\frac{uL}{n}, \frac{u}{n} \log n\} = \log_2 \frac{u}{n} + \log_2 \min\{L, \log n\} + O(1)$ bits per list element (after π-mapping). The first term is unavoidable since it already shows up in the information theoretic lower bound, the other is expected to be very small. In addition to this we should consider $O(\frac{n}{L} \log n)$ bits for each list in order to account for the cost of the $O(\log n)$-bits pointer to (the beginning of) each sublist of A, which are at most equal to the number of buckets. This term is $O(\frac{\log n}{L})$ per element of A and thus it is negligible indeed in practice.

FACT 1.6 *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. The space cost for storing a list of length n is $n(\log_2 \frac{u}{n} + \log_2 \min\{L, \log n\} + \frac{\log n}{L})$ 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 (by exploting the π-mapping like an hash-based merge), without looking at them; this allows to drop the term $\frac{n}{L}$

occurring in Fact 1.5. 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 1.1), *skipper* is the two-level algorithm (Fact 1.5, with $L = 32$), *Baeza-Yates* is the mutual-intersection algorithm (Fact 1.3, 32 denotes the bucket size for which recursion is stopped), *lookup* is our last proposal (Fact 1.6, $L = 8$).

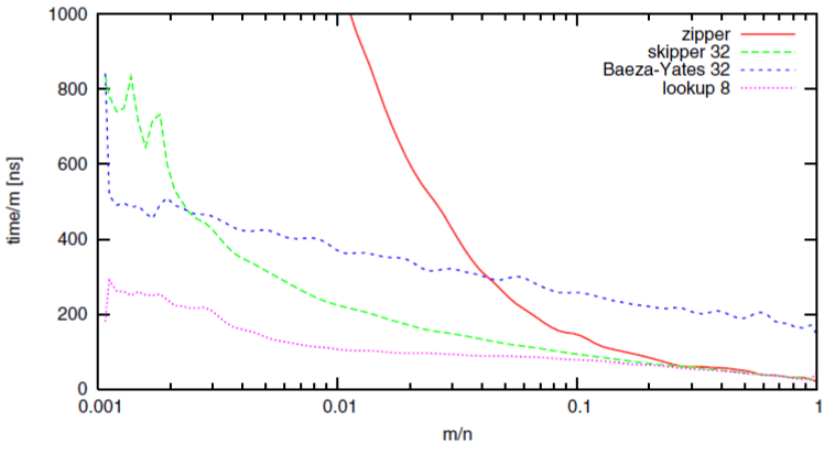

FIGURE 1.6: An experimental comparison among four sorted-set intersection algorithms.

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.