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.

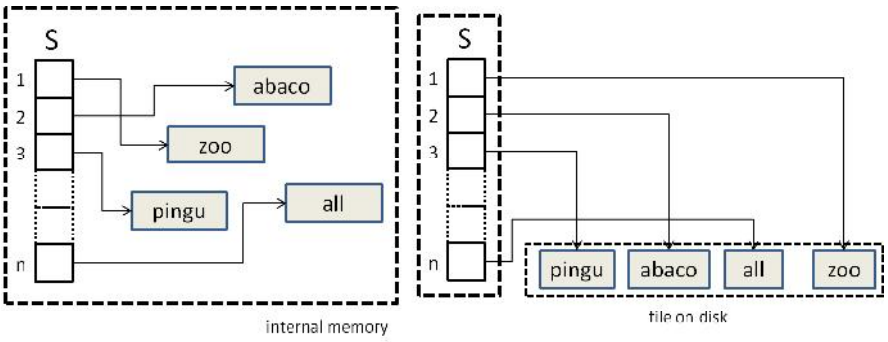

FIGURE 6.1: Two examples of string allocation, spread in the internal memory (left) and written contiguously in a file (right).

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.

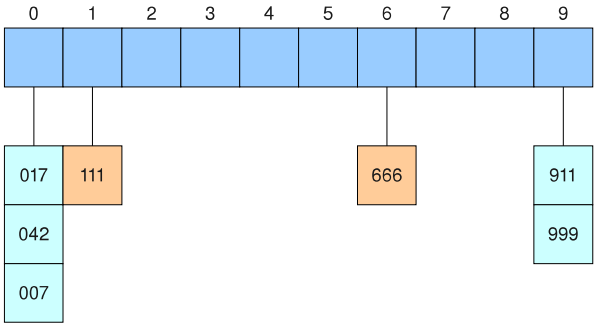

FIGURE 6.2: First distribution step in MSD-first RadixSort.

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

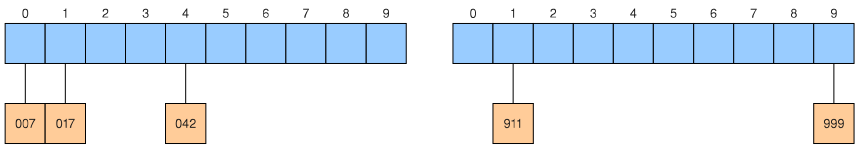

FIGURE 6.3: Recursive sort of bucket `0` (left) and bucket `9` (right) according to the second digit of their strings.

the name of *trie*, or *digital* search tree, and its main use is in string searching (as we will detail in the Chapter 7). An example of trie is given in Figure 6.4.

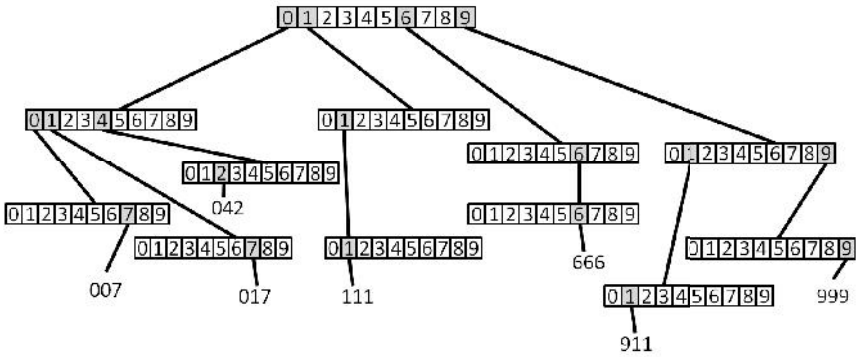

FIGURE 6.4: The trie-based view of MSD-first RadixSort for the strings of Figure 6.2

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[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 7.

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.

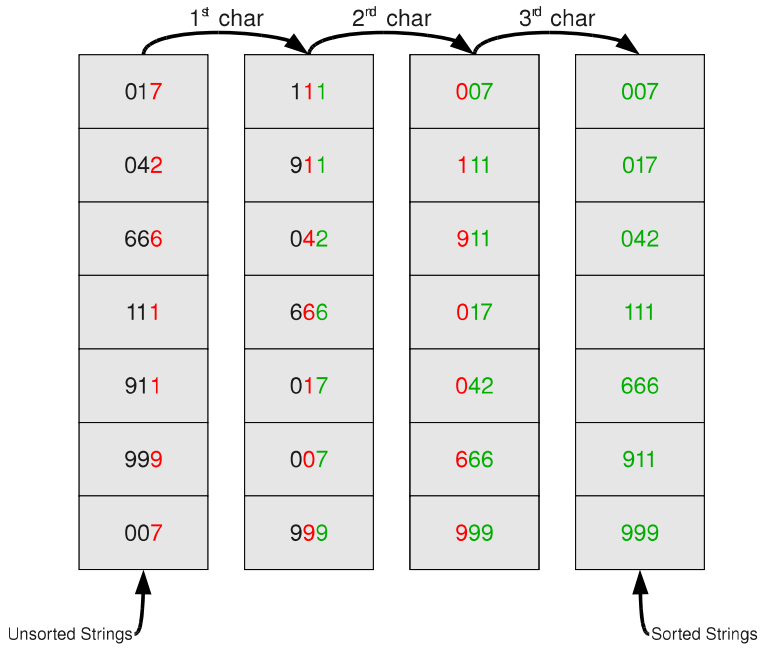

FIGURE 6.5: A running example for LSD-first RadixSort.

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| ≤ 1 then
2:      return R;
3:  else
4:      choose a pivot-string p ∈ R;
5:      R_< = {s ∈ R | s[i] < p[i]};
6:      R_= = {s ∈ R | s[i] = p[i]};
7:      R_> = {s ∈ R | 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

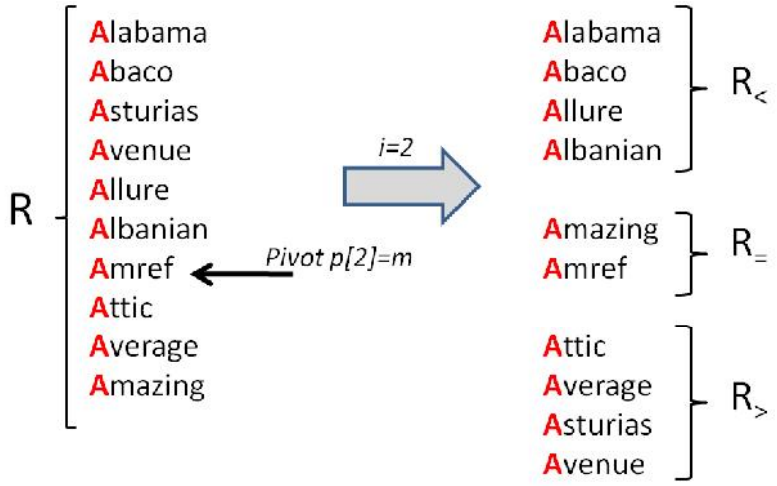

FIGURE 6.6: A running example for MULTIKEYQS(R, 2). In red we have the 1-long prefix shared by all strings in R.

$|R_< \cup R_>| \leq \alpha\, n$, 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

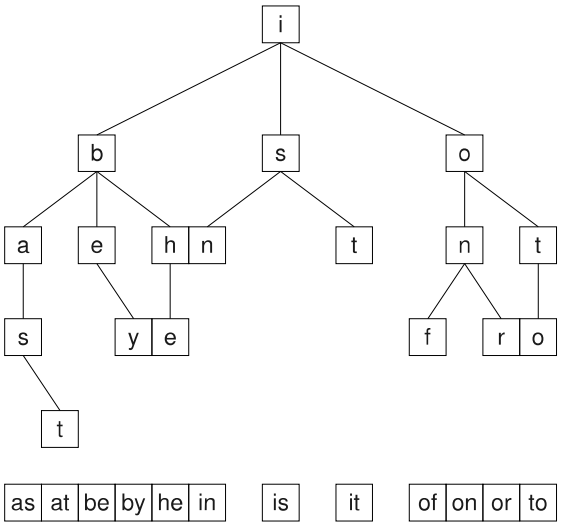

FIGURE 6.7: A ternary search tree for 12 two-letter strings. The low and high pointers are shown as solid lines, while the pointers to the equal-child are shown as dashed lines. The split character is indicated within the nodes.

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 $P[1] =$ `i`, the search proceeds down to the equal-child, increments x to 2, and thus reaches the node with split-character `s`. Here $P[2] =$ `r` $<$ `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 7) 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.