5

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. This needs to implement the comparison function between pair of strings. The obvious way is to compare the two strings brute-forcedly from their beginning, character-by-character, find their mismatch character and then use it 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 $O(Ln \log n) = O(N \log n)$ 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 drawback 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 5.1 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 and apparently-unexpected 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.

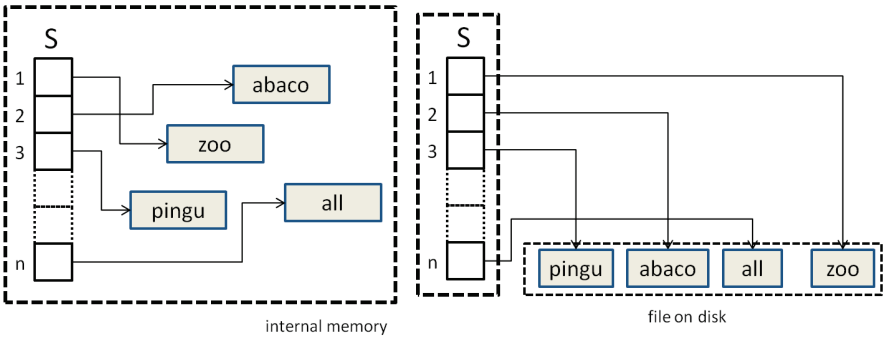

FIGURE 5.1: Two examples of string allocation, spread in the internal memory (left) and written contiguously in a file (right).

5.1 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 5.1, 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 sorting algorithm must compare the initial d_s characters of s, otherwise it would be unable to even only distinguish the string s from the others in S. So $\Omega(d)$ time 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. To be optimistic we can just restrict our attention to the sorting of the first characters of the n strings, which we may assume contain all alphabet characters σ. In the case that these characters can only be compared (and not indexed), then their sort takes $\Omega(n \log \sigma)$ time. This bound can be derived by a counting argument: the total number of distinct permutations that can be induced by n items, of which σ are distinct, is $\Theta(n^\sigma)$. Taking the logarithm, as done for the atomic sorting lower-bound, that term follows.

LEMMA 5.1 Any algorithm solving the string-sorting problem must execute $\Omega(d + n \log \sigma)$ 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 $\Omega(d + n \log n) = \Omega(N + n \log n) = \Omega(N)$ since $n \log n \leq n(\ell + \log n) = N$. But string sorters based on Mergesort or Quicksort (as the ones detailed

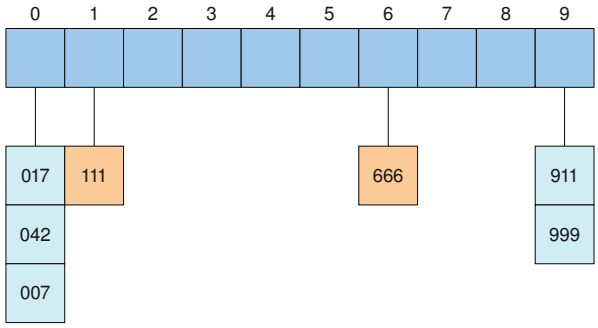

FIGURE 5.2: First distribution step in MSD-first RadixSort.

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 why this bound can be smaller than the input size N. This is due to the fact that nobody prevents to sort the strings without looking at their entire content. The introduction of the parameter d allowed us for a finer analysis.

5.2 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 $\{1, 2, \ldots, \sigma\}$ (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 if we wish to use them.

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.

5.2.1 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 using CountingSort. Figure 5.2 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 1 and 9 have to be sorted recursively according to the second digit of the strings contained into them. Figure 5.3 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, because of the usage of CountingSort, the proposed algorithm is not comparison-based.

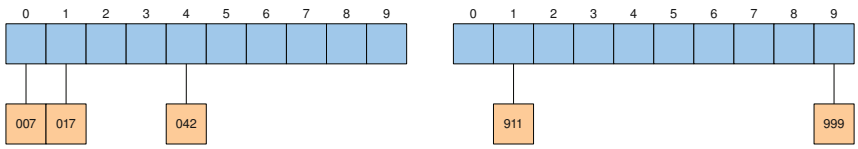

FIGURE 5.3: Recursive sort of bucket `0` (left) and bucket `9` (right) according to the second digit of their strings.

THEOREM 5.1 *MSD-first Radixsort solves the string-sorting problem over an integer alphabet of size σ in $O(d + n\sigma)$ time and space, plus the additional space cost induced by the recursive calls.*

Proof Let i be the position of the character to be compared. We distinguish two cases: (1) the strings share the same i-th character, and thus their distribution originates one single sub-bucket, (2) the strings present at least two distinct characters at position i, and thus their distribution originates at least two sub-buckets. The algorithm can be easily changed to check in $O(K)$ time which of the two cases occurs: just scan the i-th character of all strings and verify if they are equal. If case (1) occurs, then it is enough to increment i and recurse on the same strings, thus taking $O(K)$ time. If case (2) occurs, we distribute the strings among σ buckets in $O(K + \sigma)$ time and space, and then recurse on each bucket over the next character.

Let us discuss the two terms distinctly. Term $O(K)$ means actually $O(1)$ per string that participates to the partitioning step. Every partitioning advances the offset of the character to compare next. So a string s cannot participate to more than d_s distributions overall, which is the number of characters belonging to its distinguishing prefix. As a result, the term $O(K)$ which occurs in both cases induces a cost $O(d_s)$ per string, which is over all distributions $O(\sum_s d_s) = O(d)$.

Term $O(\sigma)$ must be accounted only for case (2), which we state occurs $O(n)$ times. This count can be obtained by observing that this type of partitioning is, in some sense, forming a tree of fan-out ≥ 2 and n leaves (total elements to be partitioned). We know that these types of trees have at most $O(n)$ internal nodes. Hence the overall cost of case (2) is $O(n\sigma)$. ∎

5.2.2 Trie-based sorting

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 the name of *trie*, or *digital* search tree, and its main use is in string searching (as we will detail in the Chapter **??**). An example of trie is given in Figure 5.2.

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

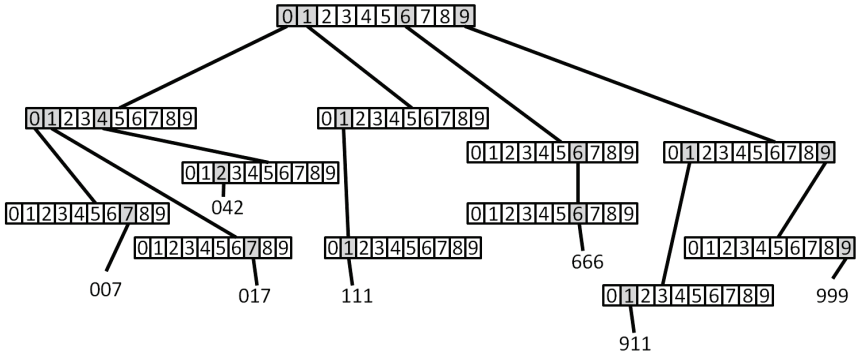

FIGURE 5.4: The trie-based view of MSD-first RadixSort for the strings of Figure 5.2

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 contains *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 $O(N)$ unary nodes, but no more than n branching nodes. The unary nodes correspond to the buckets formed by one single item in the distribution of MSD-first RadixSort (case 1 of the proof of Theorem 5.1).

If edge labels are alphabetically sorted, as in Figure 5.4, 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 for s cannot be matched with the edges outgoing from the reached node u, 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' (possibly no unary node is created). 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.

THEOREM 5.2 *Sorting strings drawn from an integer alphabet of size σ using the trie-based approach takes $O(d + n\sigma)$ time and space.*

Proof Each string insertion creates nodes only for the characters of the distinguishing prefix of s. Each node creation takes $O(\sigma)$ time because of the σ-sized array. So the insertion of all S's strings takes $O(d\sigma)$ time. However, by using the same trick as in the proof of Theorem 5.1 and create the array only for the branching characters, we have that the cost of array creation can be upper bounded by $O(n\sigma)$ time because it is done no more than $O(n)$ times.

Finally, we need to sort the e_u edges outgoing from each branching node u, thus taking $O(e_u \log e_u)$ time. This is $O(e_u \log \sigma)$ because $e_u \leq \sigma$. As in the proof of Theorem 5.1 we know that the edges outgoing from the branching nodes are no more than the number of leaves in a tree of fan-out ≥ 2. This number is n in our case, and thus $\sum e_u = O(n)$.

So the sorting takes $\sum e_u \log e_u \leq \sum e_u \log \sigma = O(n \log \sigma)$ time. Clearly, the final pre-order visit of the trie takes $O(d)$ time. ∎

An interesting change consists of replacing the σ-sized array into each node u with an hash table of size $O(e_u)$. This guarantees $O(1)$ average time for searching and inserting edges. The construction time becomes $O(d)$ on average, thus obtaining $O(d + n \log \sigma)$ average time for the total sorting process. At this point we could deploy the fact that we have an integer alphabet of size σ and thus use a faster integer sorter (such as the following LSD-first RadixSort). If $\sigma \leq n$ the sorting term gets reduced from $O(n \log \sigma)$ term to $O(n)$, without however impacting on the overall time complexity. For a general integer alphabet, we could use the best integer-sorter, which is known to take less than $O(n \log \log \sigma)$ time.

COROLLARY 5.1 Sorting strings drawn from an integer alphabet of size $\sigma \leq n$ using the trie-based approach with hashing takes $O(d + n \log \sigma)$ average time and $O(d)$ space.

We conclude this subsection by further noticing that, in the case of a general alphabet, we can avoid the naming of the alphabet characters with integers $\leq \sigma$ (as suggested at the beginning of this section) and drop the use of σ-size arrays in each node, but then we have to implement the branching out of a node in $O(\log \sigma)$ time using binary-search trees indexed by the branching characters. This would add a $O(d \log \sigma)$ term in the time complexity.

COROLLARY 5.2 The trie-based sorter applied on strings drawn from an arbitrary alphabet takes $O(d \log \sigma)$ character comparisons and $O(d)$ space. This is optimal whenever $\sigma = O(1)$ or $d = \Theta(n)$.

The space allocated for the trie can be reduced to $O(n)$ by using *compacted* tries, which compact the unary paths in single edges. The discussion of this data structure is deferred to Chapter **??**.

5.2.3 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 [6]. 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 sorter order the one they had in the input. This sorter is typically the CountingSort (see e.g. [4]), 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.

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 5.6: 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 5.6. 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

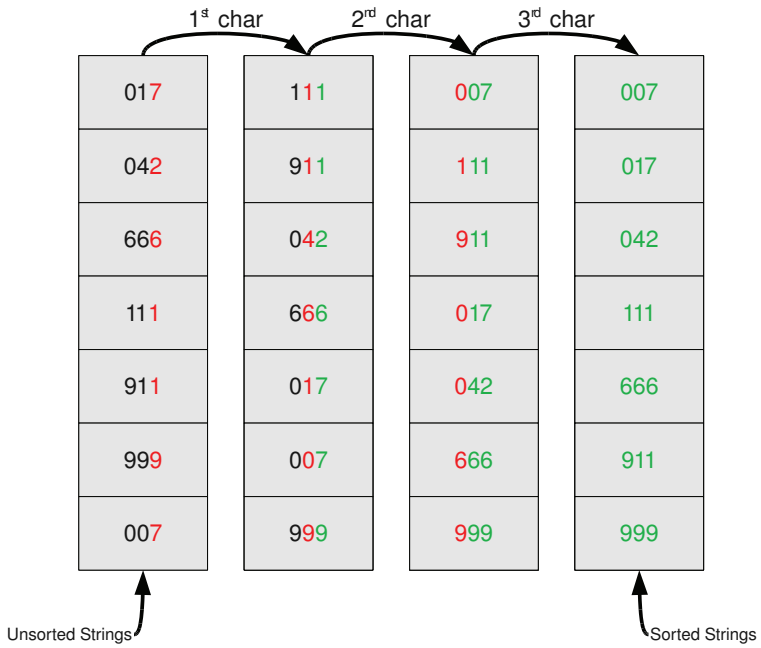

FIGURE 5.5: A running example for LSD-first RadixSort.

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 $O(L \times (n + \sigma))$. A nice property of this sorter is that it is in-place whenever the sorting black-box is in-place. This is not the case of CountingSort unless $\sigma = O(1)$.

LEMMA 5.2 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 have the same length (and 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 if L and σ are constants, the time complexity is $O(n)$ and this beats the comparison-based lower-bound (see Lemma 5.1). This is not surprising because the LSD-first RadixSort operates on an *integer* alphabet and uses CountingSort, so it is *not* a comparison-based string sorter.

The efficiency of LSD-first RadixSort goes beyond this limited case, and hinges onto the observation that nobody prevents a phase to sort groups of digits rather than a single one. Clearly the longer is this group, the larger is the time complexity of a phase, but the smaller is the number of phases. It is evident that 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 5.3 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 parameter 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 5.3 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 5.3 *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.*

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

COROLLARY 5.3 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.

Comparing the trie-based construction and the LSD-first algorithm we conclude that the former is always better than the latter, asymptotically. 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.

5.3 Multi-key Quicksort

This is a string-based variant of the well-known Quicksort algorithm extended to manage items of variable length. It is therefore a comparison-based string sorter which comes very close to the lower bound of $\Omega(d + n \log \sigma)$ stated in Lemma 5.1, and actually matches it whenever $\sigma = \Omega(n)$. 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 4 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 4 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 **??** 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 5.1 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 within R. The real implementation of this three-way partitioning can follow the Partition procedure of Chapter 4. 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 $S[1, n]$ is obtained by invoking MultikeyQS$(S, 1)$, which ensures that the invariant trivially holds for the initial sequence S. 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.

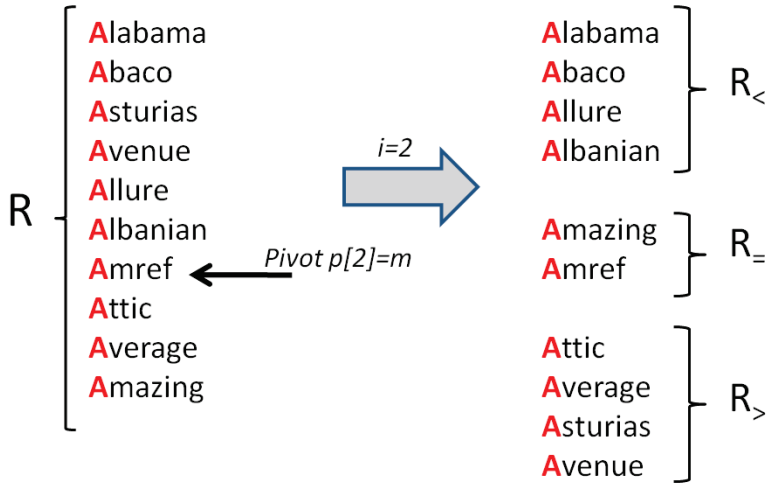

FIGURE 5.6: A running example for MultikeyQS$(R, 3)$. In red we have the i-long prefix shared by all strings in R, hence $i = 1$.

These observations make correctness immediate. We are therefore left with the problem of computing the average time complexity of MultikeyQS$(S, 1)$. Let us concentrate on a single string, say $s \in S$, 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 4), the three-way partitions are balanced and thus $|R_< \cup R_>| \leq \alpha\, n$, 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 S 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 S 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 5.4 *Multi-key Quicksort solves the string-sorting problem on a general alphabet by performing $O(d+n\log n)$ character comparisons on average. This is optimal whenever $\sigma = \Theta(n)$. The bound can be turned into a worst-case bound by adopting a worst-case linear-time median selection.*

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 (Corollary 5.2), but the algorithm is much simpler, and indeed it is the typical choice in practice.

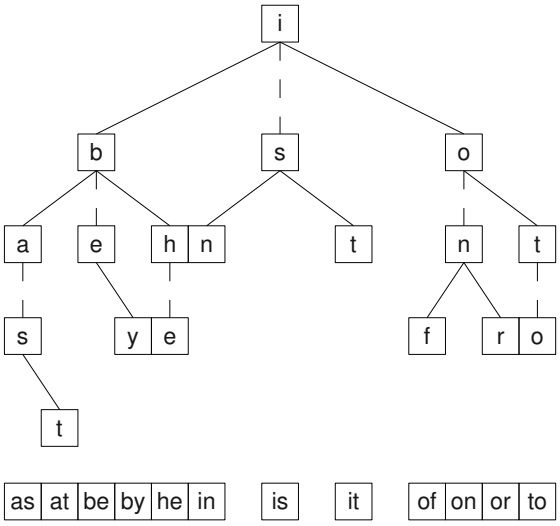

FIGURE 5.7: A ternary search tree for 12 two-letter strings. The low and high pointers are shown as solid lines, while the pointers to the equal-child are shown as dashed lines. The split character is indicated within the nodes.

We conclude this section by noticing an interesting parallel between Multikey Quicksort and *ternary* search trees, as discussed in [2]. 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 d, its low and high children also split on d-th character. Instead, the equal-child splits on the next $(d+1)$-th character. Ternary search trees may be balanced by either inserting elements in random order or applying a variety of known schemes. Searching proceeds by following edges according to the split-characters of the encountered nodes. Figure 5.7 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 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 5.5 *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.*

5.4 A simpler and fast hybrid algorithm

We already observed that trie-based string sorter is very fast when σ is small, but incurs in some memory-limitations which may be significant on large string sets. On the other hand, Multikey Quicksort is memory-conscious and has a time complexity which is alphabet-independent, but its time cost is sub-optimal for small-sized alphabets. Now we present a hybrid solution that combines MSD-first Radix Sort and Multikey Quicksort thus resulting more time efficient and offering a better memory utilization.

These two algorithms implement a divide&conquer scheme but in a complemental way. In fact MSD-first RadixSort takes $O(K + \sigma)$ time per distribution phase over K strings, by using a σ-way partitioning of the input sequence; vice versa, Multikey Quicksort takes $O(n)$ time, by using a three-way partitioning of the input sequence. So the first one is advantageous whenever the alphabet size σ is small with respect to K; while the latter is advantageous when the alphabet size is large with respect to K. Consequently, our new hybrid algorithm follows that divide&conquer approach but chooses to apply one or the other partitioning-scheme over the current string set R depending on the relation between σ and $|R|$. The net result is stated in the following theorem:

THEOREM 5.6 *The combination of MSD-first RadixSort and Multikey Quicksort solves the string-sorting problem on an integer alphabet of size $\sigma \leq n$ in $O(d + n \log \sigma)$ time.*

Proof At the beginning we can expect that $|R| > \sigma$ so it is convenient to use MSD-first RadixSort and thus partition R in σ-buckets taking $O(|R| + \sigma)$ time. Each partition advances the offset of the character to be compared at the next recursive call (see Section 5.2.1). We proceed this way until the subsequence R to sort has size smaller than σ. At this point we apply Multikey Quicksort taking $O(d_R + |R| \log |R|)$ time, where d_R is the total length of the distinguishing prefixes of the strings in R. This bound is $O(d_R + |R| \log \sigma)$ because of the condition upon $|R|$ when Multikey Quicksort is applied. Since MSD-first RadixSort induces a partition of S, the total time to sort all small subsets R is $O(d + n \log \sigma)$.

It remains to show that the cost incurred by MSD-first RadixSort to produce these small subsets is also within that time bound. This follows from the observation that at each recursive call, MSD-first RadixSort takes $O(|R| + \sigma) = O(|R|)$ time, since $|R| > \sigma$ in this case. This is linear in the number of strings to be partitioned and thus costs $O(1)$ per string. Each string participates in no more than $O(d_s)$ partitions, because at each of them we advance the compared character. So MSD-first RadixSort costs $O(\sum_s d_s) = O(d)$ time. ∎

5.5 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 short strings, whose total length is N_s, n_l is the number of long strings, 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 5.7 *In Model A, string sorting takes* $\Theta(\frac{N_1}{B}\log_{M/B}\frac{N_1}{B} + n_2\log_{M/B}n_2 + \frac{N_1+N_2}{B})$ *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_1, while sorting long strings is as difficult as sorting their first B characters. The lower bound for small strings in Theorem 5.7 is proved by extending the technique used in Chapter **??** and considering the special case where all n_1 small strings have the same length N_1/n_1. The lower bound for the long strings is proved by considering the n_2 small strings obtained by looking at their first B characters. The upper bounds in Theorem 5.7 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 5.8 *In Model B, sorting long strings takes* $\Theta(n_2\log_M n_2 + \frac{N_2}{B})$ *I/Os, whereas sorting short strings takes* $O(\min\{n_1\log_M n_1, \frac{N_1}{B}\log_{M/B}\frac{N_1}{B}\})$ *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 5.7, 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 **??**) 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_1/n_1 is $O(\frac{B}{\log_{M/B}M})$. For the (in practice) narrow range $\frac{B}{\log_{M/B}M} < \frac{N_1}{n_1} < B$, the cost of sorting short strings becomes $O(n_1\log_M n_1)$. 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 via an hashing of 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 5.9 *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] J.L. Bentley and M.D. 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] Cormen T.H. Leiserson C.E. Rivest R.L. Stein C. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

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

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