Chapter 9

String Sorting

This chapter is based on [34].

9.1 Introduction

Sort a set $R = \{s_1, s_2, \ldots, s_n\}$ of n (non-empty) strings
into the lexicographic order. N is the total length of strings, D the total length of *distinguishing prefixes*. The distinguishing prefix of a string s in R is the shortest prefix of s that is not a prefix of another string (or s if s is a prefix of another string). It is the shortest prefix of s that determines the rank of s in R. A sorting algorithm needs to access every character in the distinguishing prefixes, but no character outside the distinguishing prefixes.

alignment
all
allocate
alphabet
alternate
alternative

Figure 9.1: Example on distinguishing prefixes.

We can evaluate algorithms using different alphabet models: In an ordered alphabet, only comparisons of characters are allowed. In an ordered alphabet of constant size, a multiset of characters can be sorted in linear time. An integer alphabet is $\{1, \ldots, \sigma\}$ for integer $\sigma \geq 2$. Here, a multiset of k characters can be sorted in $\mathcal{O}(k + \sigma)$ time.

We have the following lower bounds for sorting using these models:

alphabet	lower bound
ordered	$\Omega(D + n \log n)$
constant	$\Omega(D)$
integer	$\Omega(D)$

Table 9.1: Simple lower bounds for string sorting using different alphabet models

If we use a standard sorting algorithm for strings, the worst case requires $\Theta(n \log n)$ string comparisons. Let $s_i = \alpha\beta_i$, where $|\alpha| = |\beta_i| = \log n$. This means $D = \Theta(n \log n)$. Our lower bound is $\Omega(D + n \log n) = \Omega(n \log n)$, but standard sorting has costs of $\Theta(n \log n) \cdot \Theta(\log n) = \Theta(n \log^2 n)$. In the next sections, we try to approach the lower bound for string sorting.

9.2 Multikey Quicksort

The Multikey Quicksort [35] performs in every recursion level a ternary partitioning of the data elements. In contrast to the standard algorithm, the pivot is not a whole key (which would be a complete word), but only the first character following the common prefix shared by all elements.

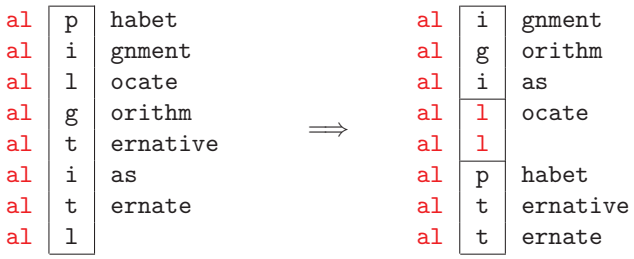

Figure 9.2: One partitioning step in multikey quicksort

Function *Multikey-quicksort*(R : *Sequence* **of** *String*, ℓ : *Integer*) : *Sequence* **of** *String*
  // ℓ is the length of the common prefix in R
  if $|R| \leq 1$ **then return** R
  choose pivot $p \in R$
  $R_< := \{s \in R \mid s[\ell+1] < p[\ell+1]\}$
  $R_= := \{s \in R \mid s[\ell+1] = p[\ell+1]\}$
  $R_> := \{s \in R \mid s[\ell+1] > p[\ell+1]\}$
  Multikey-quicksort($R_<$, ℓ)
  Multikey-quicksort($R_=$, ℓ+1)
  Multikey-quicksort($R_>$, ℓ)
  return *concatenation of* $R_<$, $R_=$, *and* $R_>$

Figure 9.3: Pseudocode for Multikey Quicksort

We will now analyse the algorithm given in pseudo code in figure 9.3. The running time is dominated by the comparisons done in the partitioning step. We will use amortized analysis to count these comparisons. If $s[\ell+1] \neq p[\ell+1]$, we charge the comparison on s. Assuming a perfect choice for the pivot element, we see that the total charge on s for this type of comparison is $\leq \log n$, as the parition containing s is at least halved.
If we have $s[\ell+1] = p[\ell+1]$, we charge the comparison on $s[\ell+1]$. After that, $s[\ell+1]$ becomes part of the common prefix in its partition and will never again be chosen as pivot character. Therefore, the charge on $s[\ell+1]$ is ≤ 1 and the total charge on all characters is $\leq D$. Combining this with the above result, we get a total runtime of $\mathcal{O}(D + n \log n)$. The only flaw in the above analysis is the assumption of a perfect pivot. Like in the analysis of standard quicksort, we can show that the expected number of $\neq$ comparisons is $2n \ln n$ when using a random pivot character.

9.3 Radix Sort

Another classic string sorting algorithm is radix sort. There exist two main variants: LSD-first radix sort starts from the end of the strings (Least Significant Digit first) and moves backward. In every phase, it partitions all strings according to the character at the current position (one group for every possible character). When this is done, the strings are recollected, starting with the group corresponding to the "smallest" character. For correct sorting, this has to be done in a stable way within a group. The LSD variant accesses all characters (as we have to reach the first character of each word for correct sorting), which implies costs of $\Omega(N)$ time. This is poor when $D \ll N$.
MSD-first radix sort on the other hand starts from the beginning of the strings (Most Significant Digit first). It distributes the strings (using counting sort) to groups according to the character at the current position and sorts these groups recursively (increasing the position of the relevant character by 1). Then, all groups are concatenated, in the order of the corresponding characters[1]. This variant accesses only distinguishing prefixes.

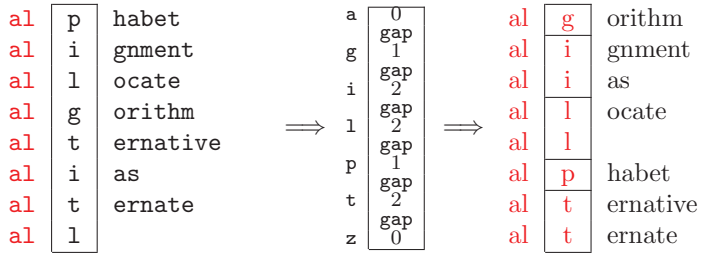

Figure 9.4: Example of one partitioning phase in MSD-first radix sort using counting sort for allocation

What is the running time of MSD-first radix sort? Partitioning a group of k strings in σ buckets takes $\mathcal{O}(k + \sigma)$ time. As the total size of the partitioned groups is D, we have $\mathcal{O}(D)$ total time on constant alphabets.

[1]When implementing this algorithm, many ideas used for Super Scalar Sample Sort (e.g. the two-pass-approach to determine the optimal bucket size) will also help for MSD-first radix sort. In fact, MSD-first radix sort inspired the development of SSSS

The total number of times any string is assigned to a group is D (the total size of all groups created while sorting). For every non-trivial partitioning step (where not all characters are equal), additional costs of $\mathcal{O}(\sigma)$ for creating groups occur. Obviously, the number of non-trivial partitionings is $\leq n$. We therefore have costs of $\mathcal{O}(D + n\sigma)$, which becomes $\mathcal{O}(D)$ for constant alphabets. When dealing with integer alphabets, another improvement helps lowering the running time: When $k < \sigma$, where k is the number of strings to be partitioned in a certain step, switch to multikey quicksort. This results in a running time of $\mathcal{O}(D + n \log \sigma)$.

Table 9.5 gives an overview over the results of this chapter. Some gaps could be closed, others require more elaborated techniques beyond this text.

alphabet	lower bound	upper bound	algorithm
ordered	$\Omega(D + n \log n)$	$\mathcal{O}(() D + n \log n)$	multikey quicksort
constant	$\Omega(D)$	$\mathcal{O}(() D)$	radix sort
integer	$\Omega(D)$	$\mathcal{O}(() D + n \log \sigma)$	radix sort + multikey quicksort

Figure 9.5: Overview on upper and lower bounds using different alphabet models