58093 String Processing Algorithms

Lectures, Autumn 2011, period II

Juha Kärkkäinen

Contents

0. Introduction

Strings and sequences are one of the simplest, most natural and most used forms of storing information.

- natural language, biosequences, programming source code, XML, music, any data stored in a file

The area of algorithm research focusing on strings is sometimes known as *stringology*. Characteristic features include

- Huge data sets (document databases, biosequence databases, web crawls, etc.) require efficiency. Linear time and space complexity is the norm.
- Strings come with no explicit structure, but many algorithms discover implicit structures that they can utilize.

About this course

On this course we will cover a few cornerstone problems in stringology. We will describe several algorithms for the same problem:

- the best algorithms in theory and/or in practice
- algorithms using a variety of different techniques

The goal is to learn a toolbox of basic algorithms and techniques.

On the lectures, we will focus on the clean, basic problem. Exercises may include some variations and extensions. We will mostly ignore any application specific issues.

Strings

An *alphabet* is the set of *symbols* or *characters* that may occur in a string. We will usually denote an alphabet with the symbol Σ and its size with σ.

We consider three types of alphabets:

- *Ordered alphabet* $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$, where $c_1 < c_2 < \cdots < c_\sigma$.
- *Integer alphabet* $\Sigma = \{0, 1, 2, \ldots, \sigma - 1\}$.
- *Constant alphabet* An ordered alphabet for a (small) constant σ.

The alphabet types are really used for *classifying algorithms* rather than alphabets:

- Algorithms for ordered alphabet use only character comparisons.
- Algorithms for integer alphabet can use operations such as using a symbol as an address to a table.
- Algorithms for constant alphabet can perform almost any operation on characters and even sets of characters in constant time.

A string is a *sequence* of symbols. The set of all strings over an alphabet Σ is

$$\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$$

where

$$\begin{aligned} \Sigma^k &= \overbrace{\Sigma \times \Sigma \times \cdots \times \Sigma}^{k} \\ &= \{a_1 a_2 \ldots a_k \mid a_i \in \Sigma \text{ for } 1 \le i \le k\} \\ &= \{(a_1, a_2, \ldots, a_k) \mid a_i \in \Sigma \text{ for } 1 \le i \le k\} \end{aligned}$$

is the set of strings of length k. In particular, $\Sigma^0 = \{\varepsilon\}$, where ε is the *empty string*.

We will usually write a string using the notation $a_1 a_2 \ldots a_k$, but sometimes using $(a_1, a_2, \ldots, a_k)$ may avoid confusion.

There are many notations for strings.

When describing algorithms, we will typically use the array notation to emphasize that the string is stored in an array:

$$\begin{aligned} S &= S[1..n] = S[1]S[2] \ldots S[n] \\ T &= T[0..n) = T[0]T[1] \ldots T[n-1] \end{aligned}$$

Note the half-open range notation $[0..n)$ which is often convenient.

In abstract context, we often use other notations, for example:

- $\alpha, \beta \in \Sigma^*$
- $x = a_1 a_2 \ldots a_k$ where $a_i \in \Sigma$ for all i
- $w = uv$, $u, v \in \Sigma^*$ (w is the *concatenation* of u and v)

We will use $|w|$ to denote the *length* of a string w.

Individual characters or their positions usually do not matter. The significant entities are the substrings or factors.

Definition 0.1: Let $w = xyz$ for any $x, y, z \in \Sigma^*$. Then x is a *prefix*, y is a *factor* (*substring*), and z is a *suffix* of w.
If x is both a prefix and a suffix of w, then x is a *border* of w.

Example 0.2: Let $w = \texttt{bonobo}$. Then

- $\varepsilon, \texttt{b}, \texttt{bo}, \texttt{bon}, \texttt{bono}, \texttt{bonob}, \texttt{bonobo}$ are the prefixes of w
- $\varepsilon, \texttt{o}, \texttt{bo}, \texttt{obo}, \texttt{nobo}, \texttt{onobo}, \texttt{bonobo}$ are the suffixes of w
- $\varepsilon, \texttt{bo}, \texttt{bonobo}$ are the borders of w
- $\varepsilon, \texttt{b}, \texttt{o}, \texttt{n}, \texttt{bo}, \texttt{on}, \texttt{no}, \texttt{ob}, \texttt{bon}, \texttt{ono}, \texttt{nob}, \texttt{obo}, \texttt{bono}, \texttt{onob}, \texttt{nobo}, \texttt{bonob}, \texttt{onobo}, \texttt{bonobo}$ are the factors of w.

Note that ε and w are always suffixes, prefixes, and borders of w. A suffix/prefix/border of w is *proper* if it is not w, and *nontrivial* if it is not ε or w.

1. String Sorting

The standard ordering for strings is the *lexicographical order*. It is *induced* by an order over the alphabet. We will use the same symbols ($\leq$, $<$, $\geq$, $\not\leq$, etc.) for both the alphabet order and the induced lexicographical order. We define the lexicographical order using the closely related concept of the *longest common prefix*.

Definition 1.1: The length of the *longest common prefix* of two strings $A[0..m)$ and $B[0..n)$, denoted by $lcp(A, B)$, is the largest integer $\ell \leq \min\{m, n\}$ such that $A[0..\ell) = B[0..\ell)$.

Definition 1.2: Let A and B be two strings over an alphabet with a total order $\leq$, and let $\ell = lcp(A, B)$. Then A is *lexicographically* smaller than or equal to B, denoted by $A \leq B$, if and only if

1. either $|A| = \ell$
2. or $|A| > \ell$, $|B| > \ell$ and $A[\ell] < B[\ell]$.

$\Omega(n \log n)$ is a well known *lower bound* for the number of comparisons needed for sorting a set of n objects by any comparison based algorithm. This lower bound holds both in the worst case and in the average case.

There are many algorithms that match the lower bound, i.e., sort using $\mathcal{O}(n \log n)$ comparisons (worst or average case). Examples include quicksort, heapsort and mergesort.

If we use one of these algorithms for sorting a set of n strings, it is clear that the number of *symbol comparisons* can be more than $\mathcal{O}(n \log n)$ in the *worst case*. Determining the order of A and B needs at least $lcp(A, B)$ symbol comparisons and $lcp(A, B)$ can be arbitrarily large in general.

On the other hand, the *average* number of symbol comparisons for two random strings is $\mathcal{O}(1)$. Does this mean that we can sort a set of *random strings* in $\mathcal{O}(n \log n)$ time using a standard sorting algorithm?

The following theorem shows that we cannot achieve $\mathcal{O}(n \log n)$ symbol comparisons for *any* set of strings (when $\sigma = n^{o(1)}$).

Theorem 1.3: Let $\mathcal{A}$ be an algorithm that sorts a set of objects using only comparisons between the objects. Let $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ be a set of n strings over an ordered alphabet Σ of size σ. Sorting $\mathcal{R}$ using $\mathcal{A}$ requires $\Omega(n \log n \log_\sigma n)$ symbol comparisons on average, where the average is taken over the initial orders of $\mathcal{R}$.

- If σ is considered to be a constant, the lower bound is $\Omega(n(\log n)^2)$.
- An intuitive explanation for this result is that the comparisons made by a sorting algorithm are *not random*. In the later stages, the algorithm tends to compare strings that are close to each other in lexicographical order and thus are likely to have long common prefixes.

Proof of Theorem 1.3. Let $k = \lfloor (\log_\sigma n)/2 \rfloor$. For any string $\alpha \in \Sigma^k$, let $\mathcal{R}_\alpha$ be the set of strings in $\mathcal{R}$ having α as a prefix. Let $n_\alpha = |\mathcal{R}_\alpha|$.

Let us analyze the number of symbol comparisons when comparing strings in $\mathcal{R}_\alpha$ *against each other*.

- Each string comparison needs at least k symbol comparisons.
- No comparison between a string in $\mathcal{R}_\alpha$ and a string outside $\mathcal{R}_\alpha$ gives any information about the relative order of the strings in $\mathcal{R}_\alpha$.
- Thus $\mathcal{A}$ needs to do $\Omega(n_\alpha \log n_\alpha)$ string comparisons and $\Omega(kn_\alpha \log n_\alpha)$ symbol comparisons to determine the relative order of the strings in $\mathcal{R}_\alpha$.

Thus the total number of symbol comparisons is $\Omega\left(\sum_{\alpha \in \Sigma^k} kn_\alpha \log n_\alpha\right)$ and

$$\sum_{\alpha \in \Sigma^k} kn_\alpha \log n_\alpha \geq k(n - \sqrt{n}) \log \frac{n - \sqrt{n}}{\sigma^k} \geq k(n - \sqrt{n}) \log(\sqrt{n} - 1)$$
$$= \Omega(kn \log n) = \Omega(n \log n \log_\sigma n) \ .$$

Here we have used the facts that $\sigma^k \leq \sqrt{n}$, that $\sum_{\alpha \in \Sigma^k} n_\alpha > n - \sigma^k \geq n - \sqrt{n}$, and that $\sum_{\alpha \in \Sigma^k} n_\alpha \log n_\alpha > (n - \sqrt{n}) \log((n - \sqrt{n})/\sigma^k)$ (see exercises). $\square$

The preceding lower bound does not hold for algorithms *specialized for sorting strings*. To derive a lower bound for algorithms based on *symbol comparisons*, we need the following concept.

Definition 1.4: Let S be a string and $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ a set of strings. The *distinguishing prefix* of S in $\mathcal{R}$ is the shortest prefix of S that separates it from the (other) members of $\mathcal{R}$. Let $dp_{\mathcal{R}}(S)$ denote the length of the distinguishing prefix, and let $dp(\mathcal{R}) = \sum_{T \in \mathcal{R}} dp_{\mathcal{R}}(T)$ be the total length of distuinguishing prefixes in $\mathcal{R}$.

Example 1.5: Distuingishing prefixes:

```
a akkoselliseen
järjesty kseen
järjestä
m erkkijonot
n ämä
```

Here $dp(\mathcal{R}) = 1 + 8 + 8 + 1 + 1 = 19$.

Theorem 1.6: Let $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ be a set of n strings. Sorting $\mathcal{R}$ into the lexicographical order by any algorithm based on symbol comparisons requires $\Omega(dp(\mathcal{R}) + n \log n)$ symbol comparisons.

Proof. The algorithm must examine every symbol in the duistinguishing prefixes. This gives a lower bound $\Omega(dp(\mathcal{R}))$.

On the other hand, the general sorting lower bound $\Omega(n \log n)$ must hold here too.

The result follows from combining the two lower bounds. $\square$

- Note that the expected value of $dp(\mathcal{R})$ for a random set of n strings is $\mathcal{O}(n \log_\sigma n)$. The lower bound then becomes $\Omega(n \log n)$.

We will next see that there are algorithms that match this lower bound. Such algorithms can sort a random set of strings in $\mathcal{O}(n \log n)$ time.

String Quicksort (Multikey Quicksort)

Quicksort is one of the fastest general purpose sorting algorithms in practice.

Here is a variant of quicksort that partitions the input into three parts instead of the usual two parts.

Algorithm 1.7: TernaryQuicksort(R)

Input: (Multi)set R in arbitrary order.
Output: R in increasing order.
(1) *if* $|R| \le 1$ *then return* R
(2) select a pivot $x \in R$
(3) $R_< \leftarrow \{s \in R \mid s < x\}$
(4) $R_= \leftarrow \{s \in R \mid s = x\}$
(5) $R_> \leftarrow \{s \in R \mid s > x\}$
(6) $R_< \leftarrow$ TernaryQuicksort($R_<$)
(7) $R_> \leftarrow$ TernaryQuicksort($R_>$)
(8) *return* $R_< \cdot R_= \cdot R_>$

In the normal, *binary* quicksort, we would have two subsets $R_\le$ and $R_\ge$, both of which may contain elements that are equal to the pivot.

- Binary quicksort is slightly faster in practice for sorting sets.
- Ternary quicksort can be faster for sorting multisets with many duplicate keys (exercise).

The time complexity of both the binary and the ternary quicksort depends on the *selection of the pivot* (exercise).

In the following, we assume an optimal pivot selection.

String quicksort is similar to ternary quicksort, but it partitions using a single character position. String quicksort is also known as *multikey quicksort*.

Algorithm 1.8: StringQuicksort($\mathcal{R}, \ell$)

Input: Set $\mathcal{R}$ of strings and the length ℓ of their common prefix.
Output: R in increasing lexicographical order.

(1) *if* $|\mathcal{R}| \leq 1$ *then return* $\mathcal{R}$
(2) $\mathcal{R}_\perp \leftarrow \{S \in \mathcal{R} \mid |S| = \ell\}$; $\mathcal{R} \leftarrow \mathcal{R} \setminus \mathcal{R}_\perp$
(3) select pivot $X \in \mathcal{R}$
(4) $\mathcal{R}_< \leftarrow \{S \in \mathcal{R} \mid S[\ell] < X[\ell]\}$
(5) $\mathcal{R}_= \leftarrow \{S \in \mathcal{R} \mid S[\ell] = X[\ell]\}$
(6) $\mathcal{R}_> \leftarrow \{S \in \mathcal{R} \mid S[\ell] > X[\ell]\}$
(7) $\mathcal{R}_< \leftarrow$ StringQuicksort($\mathcal{R}_<, \ell$)
(8) $\mathcal{R}_= \leftarrow$ StringQuicksort($\mathcal{R}_=, \ell + 1$)
(9) $\mathcal{R}_> \leftarrow$ StringQuicksort($\mathcal{R}_>, \ell$)
(10) return $\mathcal{R}_\perp \cdot \mathcal{R}_< \cdot \mathcal{R}_= \cdot \mathcal{R}_>$

In the initial call, $\ell = 0$.

Example 1.9: A possible partitioning, when $\ell = 2$.

al	p	habet
al	i	gnment
al	l	ocate
al	g	orithm
al	t	ernative
al	i	as
al	t	ernate
al	l	

$\Longrightarrow$

al	i	gnment
al	g	orithm
al	i	as
al	l	ocate
al	l	
al	p	habet
al	t	ernative
al	t	ernate

Theorem 1.10: String quicksort sorts a set $\mathcal{R}$ of n strings in $\mathcal{O}(dp(\mathcal{R}) + n \log n)$ time.

- Thus string quicksort is an optimal symbol comparison based algorithm.
- String quicksort is also fast in practice.

Proof of Theorem 1.10. The time complexity is dominated by the symbol comparisons on lines (4)–(6). We charge the cost of each comparison either on a single symbol or on a string depending on the result of the comparison:

$S[\ell] = X[\ell]$**:** Charge the comparison on the symbol $S[\ell]$.

- Now the string S is placed in the set $\mathcal{R}_=$. The recursive call on $\mathcal{R}_=$ increases the common prefix length to $\ell + 1$. Thus $S[\ell]$ cannot be involved in any future comparison and the total charge on $S[\ell]$ is 1.
- The algorithm never accesses symbols outside the distinguishing prefixes. Thus the total number of symbol comparisons resulting equality is at most $dp(\mathcal{R})$.

$S[\ell] \neq X[\ell]$**:** Charge the comparison on the string S.

- Now the string S is placed in the set $\mathcal{R}_<$ or $\mathcal{R}_>$. Assuming an optimal choice of the pivot X, the size of either set is at most $|\mathcal{R}|/2$.
- Every comparison charged on S halves the size of the set containing S, and hence the total charge accumulated by S is at most $\log n$.
- Thus the total number of symbol comparisons resulting inequality is at most $\mathcal{O}(n \log n)$. $\square$

Radix Sort

The $\Omega(n \log n)$ sorting lower bound does not apply to algorithms that use stronger operations than comparisons. A basic example is *counting sort* for sorting integers.

Algorithm 1.11: CountingSort(R)

Input: (Multi)set $R = \{k_1, k_2, \ldots k_n\}$ of integers from the range $[0..\sigma)$.
Output: R in nondecreasing order in array $J[0..n)$.

(1) *for* $i \leftarrow 0$ *to* $\sigma - 1$ *do* $C[i] \leftarrow 0$
(2) *for* $i \leftarrow 1$ *to* n *do* $C[k_i] \leftarrow C[k_i] + 1$
(3) $sum \leftarrow 0$
(4) *for* $i \leftarrow 0$ *to* $\sigma - 1$ *do* // cumulative sums
(5) $\quad tmp \leftarrow C[i]$; $C[i] \leftarrow sum$; $sum \leftarrow sum + tmp$
(6) *for* $i \leftarrow 1$ *to* n *do* // distribute
(7) $\quad J[C[k_i]] \leftarrow k_i$; $C[k_i] \leftarrow C[k_i] + 1$
(8) *return* J

- The time complexity is $\mathcal{O}(n + \sigma)$.
- Counting sort is a *stable* sorting algorithm, i.e., the relative order of equal elements stays the same.

Similarly, the $\Omega(dp(\mathcal{R}) + n \log n)$ lower bound does not apply to string sorting algorithms that use stronger operations than symbol comparisons. *Radix sort* is such an algorithm for *integer alphabets*.

Radix sort was developed for sorting large integers, but it treats an integer as a *string of digits*, so it is really a string sorting algorithm (more on this in the exercises).

There are two types of radix sorting:

MSD radix sort starts sorting from the beginning of strings (most significant digit).

LSD radix sort starts sorting from the end of strings (least significant digit).

The LSD radix sort algorithm is very simple.

Algorithm 1.12: LSDRadixSort($\mathcal{R}$)
Input: Set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings of length m over the alphabet $[0..\sigma)$.
Output: $\mathcal{R}$ in increasing lexicographical order.
(1) *for* $\ell \leftarrow m-1$ *to* 0 *do* CountingSort($\mathcal{R}$,ℓ)
(2) *return* $\mathcal{R}$

- CountingSort($\mathcal{R}$,ℓ) sorts the strings in $\mathcal{R}$ by the symbols at position ℓ using counting sort (with k_i is replaced by $S_i[\ell]$). The time complexity is $\mathcal{O}(|\mathcal{R}| + \sigma)$.
- The *stability* of counting sort is essential.

Example 1.13: $\mathcal{R} = \{\texttt{cat}, \texttt{him}, \texttt{ham}, \texttt{bat}\}$.

```
cat          hi|m|        h|a|m        |b|at
him          ha|m|        c|a|t        |c|at
ham   ==>    ca|t|   ==>  b|a|t  ==>   |h|am
bat          ba|t|        h|i|m        |h|im
```


The algorithm assumes that all strings have the same length m, but it can be modified to handle strings of different lengths (exercise).

Theorem 1.14: LSD radix sort sorts a set $\mathcal{R}$ of strings over the alphabet $[0..\sigma)$ in $\mathcal{O}(||\mathcal{R}|| + m\sigma)$ time, where $||\mathcal{R}||$ is the total length of the strings in $\mathcal{R}$ and m is the length of the longest string in $\mathcal{R}$.

- The weakness of LSD radix sort is that it uses $\Omega(||\mathcal{R}||)$ time even when $dp(\mathcal{R})$ is much smaller than $||\mathcal{R}||$.
- It is best suited for sorting short strings and integers.

MSD radix sort resembles string quicksort but partitions the strings into σ parts instead of three parts.

Example 1.15: MSD radix sort partitioning.

```
al|p|habet              al|g|orithm
al|i|gnment             al|i|gnment
al|l|ocate              al|i|as
al|g|orithm     ==>     al|l|ocate
al|t|ernative           al|l|
al|i|as                 al|p|habet
al|t|ernate             al|t|ernative
al|l|                   al|t|ernate
```

Algorithm 1.16: MSDRadixSort($\mathcal{R}, \ell$)
Input: Set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings over the alphabet $[0..\sigma)$
and the length ℓ of their common prefix.
Output: $\mathcal{R}$ in increasing lexicographical order.
(1) *if* $|\mathcal{R}| < \sigma$ *then return* StringQuicksort($\mathcal{R}, \ell$)
(2) $\mathcal{R}_\perp \leftarrow \{S \in \mathcal{R} \mid |S| = \ell\}$; $\mathcal{R} \leftarrow \mathcal{R} \setminus \mathcal{R}_\perp$
(3) $(\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_{\sigma-1}) \leftarrow$ CountingSort($\mathcal{R}, \ell$)
(4) *for* $i \leftarrow 0$ *to* $\sigma - 1$ *do* $\mathcal{R}_i \leftarrow$ MSDRadixSort($\mathcal{R}_i, \ell + 1$)
(5) *return* $\mathcal{R}_\perp \cdot \mathcal{R}_0 \cdot \mathcal{R}_1 \cdots \mathcal{R}_{\sigma-1}$

- Here CountingSort($\mathcal{R}$,ℓ) not only sorts but also returns the partitioning based on symbols at position ℓ. The time complexity is still $\mathcal{O}(|\mathcal{R}| + \sigma)$.

- The recursive calls eventually lead to a large number of very small sets, but counting sort needs $\Omega(\sigma)$ time no matter how small the set is. To avoid the potentially high cost, the algorithm switches to string quicksort for small sets.

Theorem 1.17: MSD radix sort sorts a set $\mathcal{R}$ of n strings over the alphabet $[0..\sigma)$ in $\mathcal{O}(dp(\mathcal{R}) + n \log \sigma)$ time.

Proof. Consider a call processing a subset of size $k \geq \sigma$:

- The time excluding the recursive call but including the call to counting sort is $\mathcal{O}(k + \sigma) = \mathcal{O}(k)$. The k symbols accessed here will not be accessed again.
- The algorithm does not access any symbols outside the distinguishing prefixes. Thus the total time spent in this kind of calls is $\mathcal{O}(dp(\mathcal{R}))$.

This still leaves the time spent in the calls for a subsets of size $k < \sigma$, which are handled by string quicksort. No string is included in two such calls. Therefore, the total time over all calls is $\mathcal{O}(dp(\mathcal{R}) + n \log \sigma)$.
$\square$

- There exists a more complicated variant of MSD radix sort with time complexity $\mathcal{O}(dp(\mathcal{R}) + \sigma)$.
- $\Omega(dp(\mathcal{R}))$ is a lower bound for any algorithm that must access symbols one at a time.
- In practice, MSD radix sort is very fast, but it is sensitive to implementation details.