Department of Mathematics and Computer Science January 24, 2017
University of Southern Denmark RF

Searching a Sorted Set of Strings

Assume we have a set of n strings in RAM, and know their sorted order in the form of an array S storing references to the strings in their lexicographically sorted order. That is, $i < j \Rightarrow S[i] \leq S[j]$, where by $S[i]$ we denote the string referenced by entry i of S (as opposed to the actual numerical value of the entry—i.e., we use a notation as in Java).

We will later discuss how to actually sort the strings. Here, we first focus on searching the sorted set of strings. The classical search problem on a set in the string setting looks like this, where x is a string:

> Search(x): return the (reference to the) string x in S, if it exists, or report that it does not exist.

Clearly, we can use binary search for this. Here is one formulation of it, which assumes that $S[0]$ references a sentinel string consisting of a single character `$` which is smaller than any other character and that $S[n+1]$ references a sentinel string consisting of a single character `&` which is larger than any other character.

```
l = 0
r = n+1
WHILE l+1 < r
  m = (l+r)/2  // integer division
  IF S[m] < x
    l = m
  ELSE
    r = m
IF S[r] == x
  return S[r]
ELSE
  report "Does not exist"
```

The algorithm maintains the invariant $\texttt{S}[\texttt{l}] < x \leq \texttt{S}[\texttt{r}]$, from which correctness follows, because $\texttt{l} + \texttt{1} = \texttt{r}$ at termination of the `WHILE`-loop and the array is sorted. More precisely, at termination, `S[r]` contains the smallest (wrt. the order) string larger than or equal to x. In case of multiple copies of x, it will be the left-most of these in the array.

Since comparing x to the string in $S[m]$ may take $|x|$ time, the same applies to each iteration, so the algorithm above takes $O(|x| \log n)$ time.

The main purpose of this set of notes is to describe an improved algorithm running in $O(|x| + \log n)$ time. The algorithm assumes access to $\text{lcp}(S[l], S[m])$ and $\text{lcp}(S[m], S[r])$, where $\text{lcp}(X, Y)$ for two strings X and Y denotes the length of their longest common prefix. We discuss how to get hold of this information later, and for now assume that it is available.

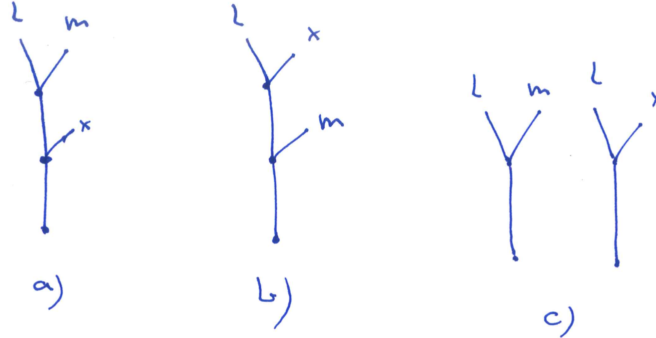
The algorithm is a variant of the binary search above. It maintains a value M defined by

$$M = \max\{\text{lcp}(S[l], x), \text{lcp}(S[r], x)\}$$

and a bit b saying whether l ($b = 0$) or r ($b = 1$) generates the maximum. As we will describe now, this extra information allows us to perform *all* the comparisons `IF S[m] < x` by a *single* scan of x (distributed over all iterations of the `WHILE`-loop).

Initially, we set $M = 0$ and $b = 0$ ($b = 1$ would also work). Consider an iteration in the binary search. Assume $b = 0$, i.e., M is generated by l (the other case is symmetric). The algorithm looks up the information $p = \text{lcp}(S[l], S[m])$ and takes actions as described below.

If $p > M$: This means $\text{lcp}(S[l], S[m]) > \text{lcp}(S[l], x)$, that is, $S[l]$ and $S[m]$ agree past the point where $S[l]$ and x diverge. This case is illustrated as a) in the figure below. Since $S[l] < x$, we can deduce $S[m] < x$ without comparing any characters of x. We can also deduce that $\text{lcp}(S[m], x) = \text{lcp}(S[l], x)$. Thus, the algorithm executes `l = m` and goes to the next iteration, keeping the values of M and b.



If $p < M$: This means $\text{lcp}(S[l], S[m]) < \text{lcp}(S[l], x)$, that is, $S[l]$ and x agree past the point where $S[l]$ and $S[m]$ diverge. This case is illustrated as b) in the figure above. Since $S[l] \leq S[m]$, we can deduce $x < S[m]$ without comparing any characters of x. We can also deduce that $\text{lcp}(S[m], x) < \text{lcp}(S[l], x)$. Thus, the algorithm executes `r = m` and goes to the next iteration, keeping the values of M and b.

If $p = M$: This means $\text{lcp}(S[l], S[m]) = \text{lcp}(S[l], x) = M$. We can conclude that $S[m]$ and x agree for at least the first M characters. This case is illustrated as c) in the figure above. We find $\text{lcp}(x, S[m])$ by scanning x and $S[m]$ from position M until a position is found where their characters do not match (or the end of one string is met). This resolves the `IF S[m] < x` comparison, and the algorithm executes either `l = m` or `r = m`, as appropriate. If t matching positions were found during the scanning, it increases the value of M by t, making it equal to $\text{lcp}(x, S[m])$, which is the correct new value for M. Finally, b set to zero if `l = m` was executed, and is set to one if `r = m` was executed.

Correctness follows from correctness of the basic binary search above, since the decisions in all iterations are the same. For time complexity, note that each iteration now takes $O(1)$ time, except for the scanning in the last case. Let t_i be the enlargement of M the i'th time this case is met. Then the total time is $O(\sum_i t_i + \log n)$. Since $\sum_i t_i \leq |n|$ (because M never decreases and at most is $|n|$), the total time is $O(|n| + \log n)$.

It turns out that with the same tools can solve the following more general search problem in time $O(|x| + \log n + |R|)$:

PrefixSearch(x): return the (references to the) set R of strings in S which have x as a prefix (including x itself, if present).

To see this, recall that in the binary search (both versions above), at termination, `S[r]` contains the smallest (wrt. the order) string larger than or equal to x, and that in case of multiple copies of x, it will be the left-most of these in the array. Assume now that R is non-empty. We prove in Corollary 2 below that R forms a consecutive segment $S[i..j]$ of the array. If we in the binary search consider all y which have x as a prefix to be equal in order to x (this we can decide by what actions are taken when a scan of x and y reaches the end of x), `r` must have the value i at termination. Thus R is non-empty if and only if `S`[`r`] has x as a prefix (which can be checked in $O(|x|)$ time). If non-empty, we know i. The value j can be found by running a similar binary search algorithm, only this time maintaining the invariant `S`[`l`] $\leq x <$ `S`[`r`] (that is, equality goes to the `IF`-case), for which `l` must have the value j at termination, by a symmetric argument.[1]

Hence, we can return R in time $O(|x| + \log n + |R|)$

We now discuss how the information $p = \text{lcp}(S[l], S[m])$ (and in the symmetric case, $p = \text{lcp}(S[m], S[r])$) used in the search algorithm is found and stored. We first define $\text{lcp}(k) = \text{lcp}(S[k], S[k+1])$ for $0 < k < n$ (or for $0 \leq k \leq n$, assuming sentinel strings), for which we prove the following useful lemma. Corollary 2 was used above. Corollary 3 will be used finding said information.

Lemma 1 *For all $i < j$ we have*

$$\text{lcp}(S[i], S[j]) = \min_{i \leq k < j} \text{lcp}(k).$$

Proof: Let $l = \min_{i \leq k < j} \text{lcp}(k)$. Then clearly *all* the strings $S[k]$, $i \leq k \leq j$ agree on the first l characters. In particular, $\text{lcp}(S[i], S[j]) \geq l = \min_{i \leq k < j} \text{lcp}(k)$. Conversely, let $t = \text{lcp}(S[i], S[j])$. If there exists any $k \geq i$ for which $\text{lcp}(k) < t$, let k' be the smallest such k. Then the strings $S[i]$ and $S[k']$ agree on the first t characters, while $S[k']$ and $S[k'+1]$ differ within the first k characters, with $S[k'] < S[k'+1]$ by the ordering in $S[]$. Since $S[i]$ and $S[j]$ also agree on the first t characters, we must have $S[j] < S[k'+1]$,

[1] If the LCP(k) information mentioned below is available, we can also just scan from i first endpoint until the first time $\text{lcp}(k) < |x|$.

hence $j < k' + 1$, i.e., $j \leq k'$. So in $\min_{i \leq k < j} \text{lcp}(k)$, all values minimized over are at least $t = \text{lcp}(S[i], S[j])$. □

Corollary 2 *For any string x, the set R of strings in S which have x as a prefix forms a consecutive segment $S[i..j]$ of the array $S[]$.*

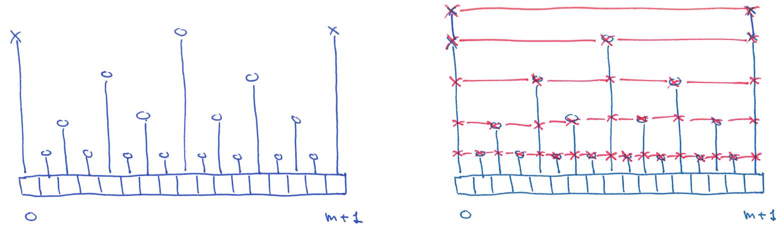
Proof: Let $S[i]$ be the left-most string having x as a prefix, and let $S[j]$ the the right-most. Clearly, $R \subseteq S[i..j]$. As $\text{lcp}(S[i], S[j]) = |x|$, Lemma 1 shows that all strings in $S[i..j]$ agree on the first $|x|$ characters, hence $R \supseteq S[i..j]$. □

Corollary 3 *For all $i < k < j$ we have*

$$\text{lcp}(S[i], S[j]) = \min\{\text{lcp}(S[i], S[k]), \text{lcp}(S[k], S[j])\}.$$

Proof: This is clear from Lemma 1 by the way minimization works. □

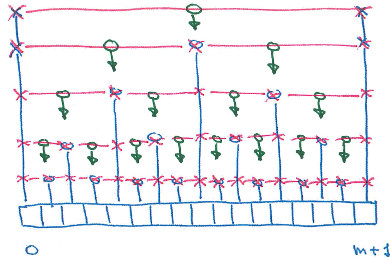
To find and store the information $\text{lcp}(S[l], S[m])$ and $\text{lcp}(S[m], S[r]))$ used in the search algorithm, we note that possible query pairs (l and m, m and r) used in *all* possible binary searches can be represented by a binary tree over the array $S[]$. In the figure on the left below, the initial values of `l` and `r` are shown by x's, and all possible later values of `m` (and hence later values of `l` and `r`) are shown by o's. The corresponding query pairs are shown in the figure on the right as red, horizontal lines.



This means that there are only $O(n)$ such pairs used (not $\Theta(n^2)$). It also means that assuming that the $\text{lcp}(k) = \text{lcp}(S[k], S[k+1])$ values are known, the answers to the used queries can be precomputed from them in $O(n)$ time by Corollary 3: just merge the answers for neighboring segments in

a bottom-up fashion (using Corollary 3), with the lcp(k) values being the answers to the size two segments at the bottom.

For storing the values, the array of the lcp(k) values already stores the answers for the cases $\mathtt{l} + 1 = \mathtt{r}$. When $\mathtt{l} + 1 < \mathtt{r}$, we can uniquely store the answer in another array, at position $\lfloor (\mathtt{l} + \mathtt{r})/2 \rfloor$, as can be seen in the following picture (green arrows indicate the array entry where the answer to the query represented by the corresponding red line is stored):



For sorting the strings to begin with, standard sorting algorithms like Quicksort or Mergesort are not well suited, as comparisons between two strings may take time proportional to their length. One algorithm designed for strings is Radixsort. The Least Significant Digit (LSD) version of it known from the course Algoritms and Data Structures is designed for strings of equal length. More general is the Most Significant Digit (MSD), a top-down distribution sort deriving very naturally from the definition of lexicographical order. It can be described as follows, where S is the set of strings (given as an array of references to the strings).

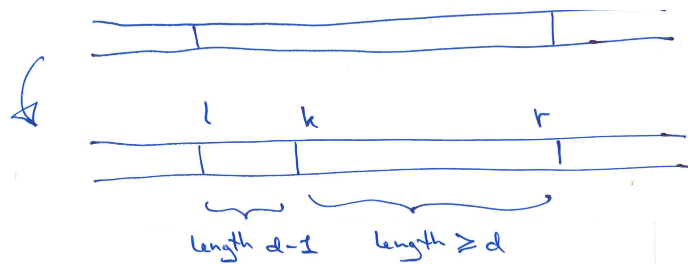
```
Radixsort(S):
  n = |S|
  RETURN Rsort(0,n-1,1,S)

Rsort(l,r,d,S)
  if l < r
    k = ShiftShortToFront(l,r,d-1,A)
    sort the strings in S[k,r] in their d'th character
    FOR all d-level bucket segments S[i..j] in S[k..r]:
      RSort(i,j,d+1,S)
```

The core idea of the algorithm is captured by the following invariant:

> At a call `Rsort(l,r,d,S)`, the strings with representatives in $S[l..r]$ all have length at least $d-1$, and their first $d-1$ characters are the same.

The call `k = ShiftShortToFront(l,r,d-1,S)` is a linear time scan of $S[l..r]$ which partitions it such that all strings of length $d-1$ appears first (leftmost) and returns the left border k of the segment of S with the remaining longer strings:



A *d-level bucket segment* $S[i..j]$ is a maximal contiguous segment of S where the strings agree on their d'th character (and this character is present). In the `FOR`-loop above, these buckets segments are found by a scan of $S[k..r]$.

Correctness follows from the definition of lexicographical order. The time depends on the sorting algorithm used in the third last line. If the alphabet is considered of size $O(1)$ and we use Countingsort, the sorting of the $r-k+1$ characters in the third last line takes $O(r-k)$ time, hence each character of each string of incurs $O(1)$ time. Thus, the total time is $O(L)$, where L is the sum of the lengths of the strings.

If the alphabet is considered infinite (is comparison based) and we use an optimal comparison based sorting algorithm (such as Mergesort), the sorting of the $r-k+1$ characters in the third last line takes $O((r-k)\log(r-k))$ time, hence each character of each string of incurs $O(\log n)$ time. Thus, the total time is $O(L \log n)$, where L is the sum of the lengths of the strings.

The recursion tree of MSD Radixsort is actually the trie of the strings (see the later handout from the Goodrich and Tamassia book for tries). This trie can of course be generated during the algorithm. By a DFS-traversal of this trie, we can then easily generate the lcp(k) values (see details in note on suffix trees and suffix arrays).