Longest Common Prefixes

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 can define the lexicographical order using the concept of the *longest common prefix*.

Definition 1.4: 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 1.5: 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 $A[\ell] < B[\ell]$.

The concept of longest common prefixes can be generalized for sets:

Definition 1.6: For a string S and a string set $\mathcal{R}$, define

$$\begin{aligned} lcp(\mathcal{R}) &= \max\{\ell \mid A[0..\ell) = B[0..\ell) \text{ for all } A, B \in \mathcal{R}\} \\ lcp(S, \mathcal{R}) &= \max\{lcp(S, T) \mid T \in \mathcal{R}\} \\ \Sigma lcp(\mathcal{R}) &= \sum_{T \in \mathcal{R}} lcp(T, \mathcal{R} \setminus \{T\}) \end{aligned}$$

The concept of *distinguishing prefix* is closely related and often used in place of the longest common prefix for sets. The distinguishing prefix of a string is the shortest prefix that separates it from other strings in the set.

For a prefix free set $\mathcal{R}$ the sum of the lengths of the distinguishing prefixes is $\Sigma dp(\mathcal{R}) = \Sigma lcp(\mathcal{R}) + |\mathcal{R}|$. For a non-prefix free set, the distinguishing prefixes are not always really fully defined.

However, even more interesting is a third measure of longest common prefixes in a set defined next. It is slightly different from both $\Sigma lcp(\mathcal{R})$ and $\Sigma dp(\mathcal{R})$.

Definition 1.7: Let $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ be a set of strings and assume $S_1 < S_2 < \cdots < S_n$. Then the LCP array $LCP_{\mathcal{R}}[1..n]$ is defined by

$$LCP_{\mathcal{R}}[i] = lcp(S_i, \{S_1, \ldots, S_{i-1}\}) \ .$$

Furthermore, the LCP array sum is

$$\Sigma LCP(\mathcal{R}) = \sum_{i \in [1..n]} LCP_{\mathcal{R}}[i] \ .$$

Example 1.8: Let $\mathcal{R} = \{$`pot$`, `potato$`, `pottery$`, `tattoo$`, `tempo$`$\}$. Then $\Sigma lcp(\mathcal{R}) = 11$, $\Sigma dp(\mathcal{R}) = 16$, $\Sigma LCP(\mathcal{R}) = 7$ and the LCP array is:

$LCP_{\mathcal{R}}$	
0	`pot$`
3	`potato$`
3	`pottery$`
0	`tattoo$`
1	`tempo$`

Theorem 1.9: The number of nodes in $trie(\mathcal{R})$ is exactly $||\mathcal{R}|| - \Sigma LCP(\mathcal{R}) + 1$, where $||\mathcal{R}||$ is the total length of the strings in $\mathcal{R}$.

Proof. Consider the construction of $trie(\mathcal{R})$ by inserting the strings one by one in the lexicographical order using Algorithm 1.2. Initially, the trie has just one node, the root. When inserting a string S_i, the algorithm executes exactly $|S_i|$ rounds of the two while loops, because each round moves one step forward in S_i. The first loop follows existing edges as long as possible and thus the number of rounds is $LCP_{\mathcal{R}}[i] = lcp(S_i, \{S_1, \ldots, S_{i-1}\})$. This leaves $|S_i| - LCP_{\mathcal{R}}[i]$ rounds for the second loop, each of which adds one new node to the trie. Thus the total number of nodes in the trie at the end is:

$$1 + \sum_{i \in [1..n]} |S_i| - LCP_{\mathcal{R}}[i] = ||\mathcal{R}|| - \Sigma LCP(\mathcal{R}) + 1 \ .$$

□

The proof reveals a close connection between $LCP_{\mathcal{R}}$ and the structure of the trie. We will later see that $LCP_{\mathcal{R}}$ is useful as an actual data structure in its own right.

The LCP array $LCP_{\mathcal{R}}$ and its sum have other interesting properties:

- $\Sigma LCP(\mathcal{R}) \leq \Sigma lcp(\mathcal{R}) \leq 2 \cdot \Sigma LCP(\mathcal{R})$.
- For $i \in [2..n]$, $LCP_{\mathcal{R}}[i] = lcp(S_i, S_{i-1})$.
- Let $\pi : [1..n] \to [1..n]$ be an arbitrary permutation. Define

$$LCP_{\mathcal{R},\pi}[i] = lcp(S_{\pi(i)}, \{S_{\pi(1)}, \ldots, S_{\pi(i-1)}\})$$
$$\Sigma LCP_{\pi}(\mathcal{R}) = \sum_{i \in [1..n]} LCP_{\mathcal{R},\pi}[i] \ .$$

  In other words, $LCP_{\mathcal{R},\pi}$ and $\Sigma LCP_{\pi}(\mathcal{R})$ are the same as $LCP_{\mathcal{R}}$ and $\Sigma LCP(\mathcal{R})$ except the order of the strings is different. Then $\Sigma LCP_{\pi}(\mathcal{R}) = \Sigma LCP(\mathcal{R})$ and $LCP_{\mathcal{R},\pi}$ is a permutation of $LCP_{\mathcal{R}}$.

The proofs are left as exercises.

Compact Trie

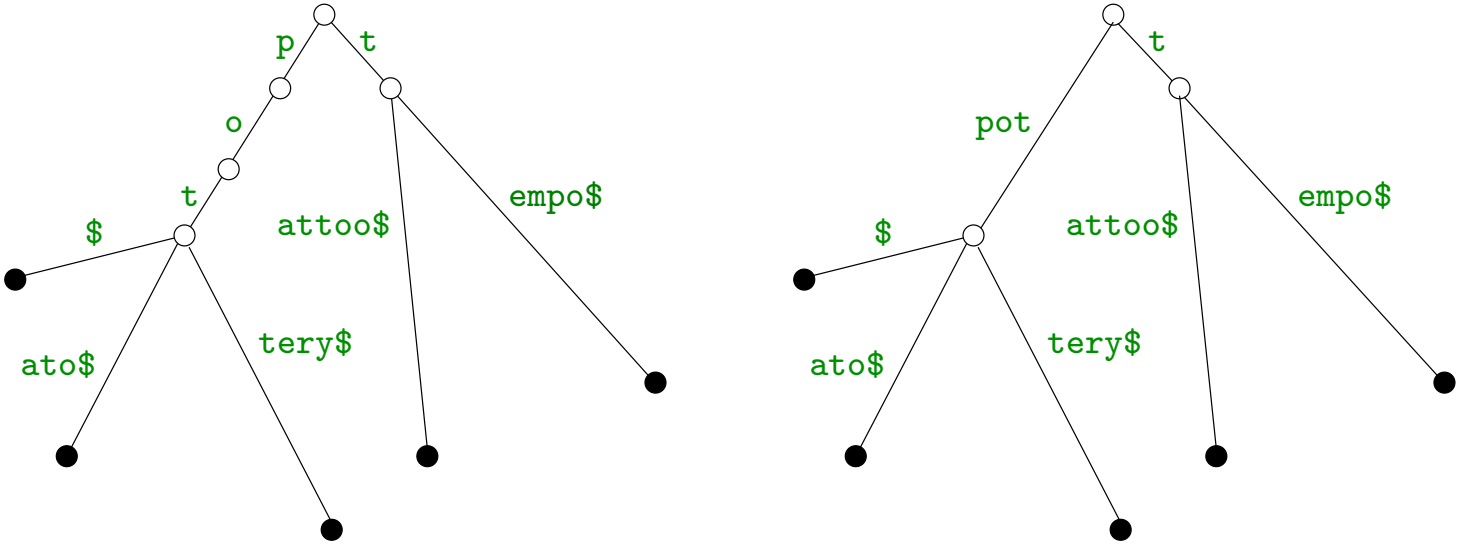
Tries suffer from a large number of nodes, close to $||\mathcal{R}||$ in the worst case. The space requirement can be problematic, since typically each node needs much more space than a single symbol.

Path compacted tries reduce the number of nodes by replacing branchless path segments with a single edge.

- Leaf path compaction applies this to path segments leading to a leaf. The number of nodes is now $|\mathcal{R}| + \Sigma lcp(\mathcal{R}) - \Sigma LCP(\mathcal{R}) + 1$ (exercise).
- Full path compaction applies this to all path segments. Then every internal node (except possibly the root) has at least two children. In such a tree, there is always at least as many leaves as internal nodes. Thus the number of nodes is at most $2|\mathcal{R}|$.

The full path compacted trie is called a compact trie.

Example 1.10: Path compacted tries for
$\mathcal{R} = \{\texttt{pot\$}, \texttt{potato\$}, \texttt{pottery\$}, \texttt{tattoo\$}, \texttt{tempo\$}\}$.



The egde labels are factors of the input strings. If the input strings are stored separately, the edge labels can be represented in constant space using pointers to the strings.

The time complexity of the basic operations on the compact trie is the same as for the trie (and depends on the implementation of the child operation in the same way), but prefix and range queries are faster on the compact trie (exercise).

Ternary Trie

Tries can be implemented for ordered alphabets but a bit awkwardly using a comparison-based child function. Ternary trie is a simpler data structure based on symbol comparisons.

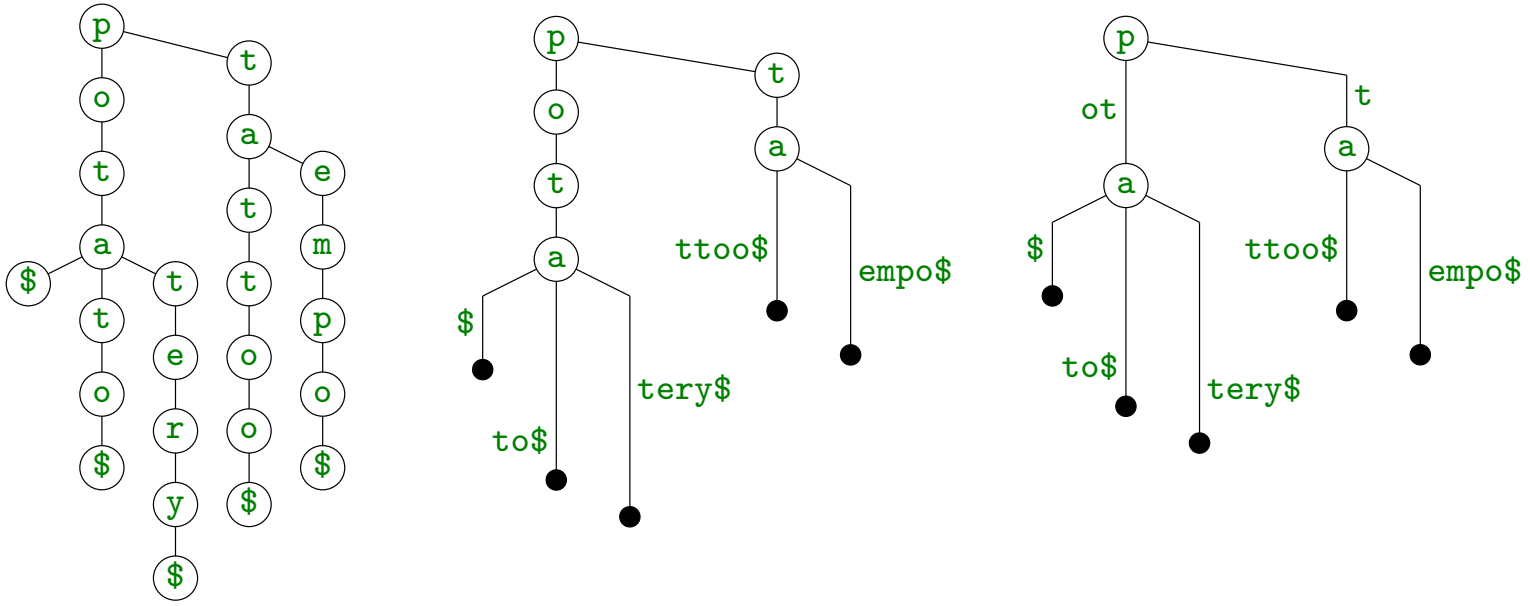
Ternary trie is like a binary search tree except:

- Each internal node has three children: smaller, equal and larger.
- The branching is based on a single symbol at a given position as in a trie. The position is zero (first symbol) at the root and increases along the middle branches but not along side branches.

Ternary trie has variants similar to the standard (σ-ary) trie:

- A basic ternary trie, which is a full representation of the strings.
- Compact ternary tries reduce space by compacting branchless path segments.

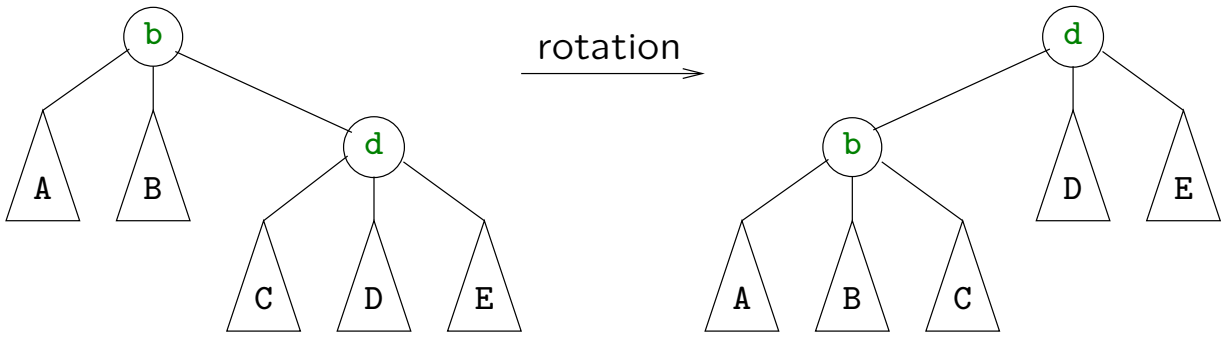
Example 1.11: Ternary tries for
$\mathcal{R} = \{\texttt{pot\$}, \texttt{potato\$}, \texttt{pottery\$}, \texttt{tattoo\$}, \texttt{tempo\$}\}$.



Ternary tries have the same asymptotic size as the corresponding tries.

A ternary trie is balanced if each left and right subtree contains at most half of the strings in its parent tree.

- The balance can be maintained by rotations similarly to binary search trees.



- We can also get reasonably close to a balance by inserting the strings in the tree in a random order.

In a balanced ternary trie each step down either

- moves the position forward (middle branch), or
- halves the number of strings remaining in the subtree (side branch).

Thus, in a balanced ternary trie storing n strings, any downward traversal following a string S passes at most $|S|$ middle edges and at most $\log n$ side edges.

Thus the time complexity of insertion, deletion, lookup and lcp query is $\mathcal{O}(|S| + \log n)$.

In comparison based tries, where the *child* function is implemented using binary search trees, the time complexities could be $\mathcal{O}(|S| \log \sigma)$, a multiplicative factor $\mathcal{O}(\log \sigma)$ instead of an additive factor $\mathcal{O}(\log n)$.

Prefix and range queries behave similarly (exercise).

String Sorting

$\Omega(n \log n)$ 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 1.12: 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 $\Omega(n(\log n)^2)$.
- Note that the theorem holds for *any* comparison based sorting algorithm $\mathcal{A}$ and *any* string set $\mathcal{R}$. In other words, we can choose $\mathcal{A}$ and $\mathcal{R}$ to minimize the number of comparisons and still not get below the bound.
- Only the initial order is random rather than "any". Otherwise, we could pick the correct order and use an algorithm that first checks if the order is correct, needing only $\mathcal{O}(n + \Sigma LCP(\mathcal{R}))$ symbol comparisons.

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 1.12. 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

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

$$= \Omega\left(kn \log n\right) = \Omega\left(n \log n \log_\sigma n\right) \ .$$

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). □

The preceding lower bound does not hold for algorithms specialized for sorting strings.

Theorem 1.13: 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 $\Omega(\Sigma LCP(\mathcal{R}) + n \log n)$ symbol comparisons.

Proof. If we are given the strings in the correct order and the job is to verify that this is indeed so, we need at least $\Sigma LCP(\mathcal{R})$ symbol comparisons. No sorting algorithm could possibly do its job with less symbol comparisons. This gives a lower bound $\Omega(\Sigma LCP(\mathcal{R}))$.

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. □

- Note that the expected value of $\Sigma LCP(\mathcal{R})$ 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 1.14: TernaryQuicksort(R)

Input: (Multi)set R in arbitrary order.
Output: R in ascending order.

(1) if $|R| \leq 1$ 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 $R_{\leq}$ and $R_{\geq}$, 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. Sorting a multiset of size n with σ distinct elements takes $\mathcal{O}(n \log \sigma)$ comparisons (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 giving $\mathcal{O}(n \log n)$ worst case time complexity.

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

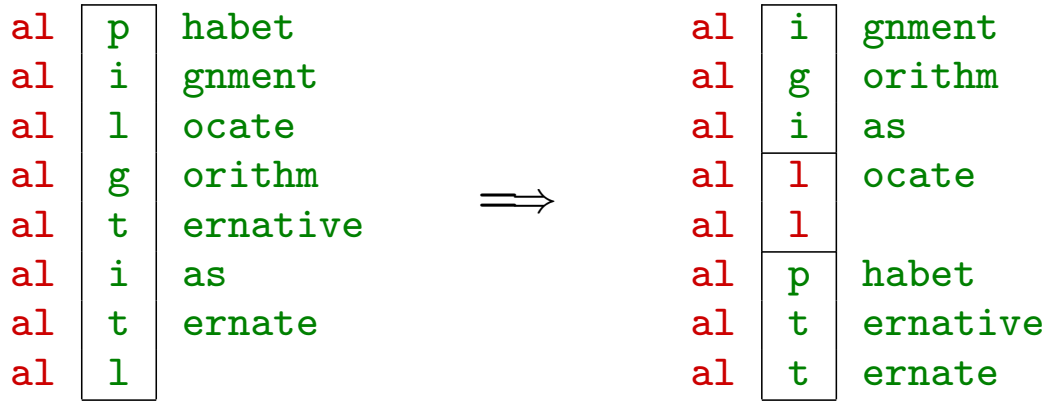
Algorithm 1.15: StringQuicksort($\mathcal{R}, \ell$)

Input: (Multi)set $\mathcal{R}$ of strings and the length ℓ of their common prefix.
Output: R in ascending 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 1.16: A possible partitioning, when $\ell = 2$.



Theorem 1.17: String quicksort sorts a set $\mathcal{R}$ of n strings in $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n \log n)$ time.

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

Proof of Theorem 1.17. 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.
- Only $lcp(S, \mathcal{R} \setminus \{S\})$ symbols in S can be involved in these comparisons. Thus the total number of symbol comparisons resulting equality is at most $\Sigma lcp(\mathcal{R}) = \Theta(\Sigma LCP(\mathcal{R}))$. (Exercise: Show that the number is exactly $\Sigma LCP(\mathcal{R})$.)

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

- Now the string S is placed in the set $\mathcal{R}_<$ or $\mathcal{R}_>$. The size of either set is at most $|\mathcal{R}|/2$ assuming an optimal choice of the pivot X.
- 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 $\mathcal{O}(n \log n)$. □