**Algorithm 1.16:** MSDRadixSort($\mathcal{R}, \ell$)
Input: (Multi)set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings over the alphabet $[0..\sigma)$ and the length $\ell$ of their common prefix.
Output: $\mathcal{R}$ in increasing lexicographical order.

*if* $|\mathcal{R}| < \sigma$ *then return* StringQuicksort($\mathcal{R}, \ell$)
$\mathcal{R}_\perp \leftarrow \{S \in \mathcal{R} \mid |S| = \ell\}$; $\mathcal{R} \leftarrow \mathcal{R} \setminus \mathcal{R}_\perp$
$(\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_{\sigma-1}) \leftarrow$ CountingSort($\mathcal{R}, \ell$)
*for* $i \leftarrow 0$ *to* $\sigma - 1$ *do* $\mathcal{R}_i \leftarrow$ MSDRadixSort($\mathcal{R}_i, \ell + 1$)
*return* $\mathcal{R}_\perp \cdot \mathcal{R}_0 \cdot \mathcal{R}_1 \cdots \mathcal{R}_{\sigma-1}$

- Here CountingSort($\mathcal{R}$,$\ell$) not only sorts but also returns the partitioning based on symbols at position $\ell$. 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.



## String Mergesort

Standard comparison based sorting algorithms are not optimal for sorting strings because of an *imbalance* between effort and result in a string comparison: it can take a lot of time but the result is only a bit or a trit of useful information.

String quicksort solves this problem by using symbol comparisons where the constant time is in balance with the information value of the result.

String mergesort takes the opposite approach. It replaces a standard string comparison with the operation LcpCompare($A, B, k$):

- The return value is the pair $(x, \ell)$, where $x \in \{<, =, >\}$ indicates the order, and $\ell$ is the length of the *longest common prefix (lcp)* of strings $A$ and $B$, denoted by $lcp(A, B)$.

- The input value $k$ is the length of a known common prefix, i.e., a lower bound on $lcp(A, B)$. The comparison can skip the first $k$ characters.

Any extra time spent in the comparison is balanced by the extra information obtained in the form of the lcp value.



The following result show how we can use the information from past comparisons to obtain a lower bound or even the exact value for an lcp.

**Lemma 1.18:** Let $A$, $B$ and $C$ be strings.

(a) $lcp(A, C) \geq \min\{lcp(A, B), lcp(B, C)\}$.

(b) If $A \leq B \leq C$, then $lcp(A, C) = \min\{lcp(A, B), lcp(B, C)\}$.

**Proof.** Assume $\ell = lcp(A, B) \leq lcp(B, C)$. The opposite case $lcp(A, B) \geq lcp(B, C)$ is symmetric.

(a) Now $A[0..\ell) = B[0..\ell) = C[0..\ell)$ and thus $lcp(A, C) \geq \ell$.

(b) Either $|A| = \ell$ or $A[\ell] < B[\ell] \leq C[\ell]$. In either case, $lcp(A, C) = \ell$.
$\square$

It can also be possible to determine the order of two strings without comparing them directly.

**Lemma 1.19:** Let $A \le B, B' \le C$ be strings.

(a) If $lcp(A,B) > lcp(A,B')$, then $B < B'$.

(b) If $lcp(B,C) > lcp(B',C)$, then $B > B'$.

**Proof.** We show (a); (b) is symmetric. Assume to the contrary that $B \ge B'$. Then by Lemma 1.18, $lcp(A,B) = \min\{lcp(A,B'), lcp(B',B)\} \le lcp(A,B')$, which is a contradiction. □



String mergesort has the same structure as the standard mergesort: sort the first half and the second half separately, and then merge the results.

**Algorithm 1.20:** StringMergesort($\mathcal{R}$)
Input: Set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings.
Output: $\mathcal{R}$ sorted and augmented with lcp information.
(1) *if* $|\mathcal{R}| = 1$ *then return* $\{(S_1, 0)\}$
(2) $m \leftarrow \lfloor n/2 \rfloor$
(3) $\mathcal{P} \leftarrow$ StringMergesort($\{S_1, S_2, \ldots, S_m\}$)
(4) $\mathcal{Q} \leftarrow$ StringMergesort($\{S_{m+1}, S_{m+2}, \ldots, S_n\}$)
(5) *return* StringMerge($\mathcal{P}, \mathcal{Q}$)

The output is of the form

$$\{(T_1, \ell_1), (T_2, \ell_2), \ldots, (T_n, \ell_n)\}$$

where $\ell_i = lcp(T_i, T_{i-1})$ for $i > 1$ and $\ell_1 = 0$.

In other words, we get not only the order of the strings but also a lot of information about their common prefixes. The procedure StringMerge uses this information effectively.



**Algorithm 1.21:** StringMerge($\mathcal{P}$,$\mathcal{Q}$)
Input: Sequences $\mathcal{P} = \big((S_1, k_1), \ldots, (S_m, k_m)\big)$ and $\mathcal{Q} = \big((T_1, \ell_1), \ldots, (T_n, \ell_n)\big)$
Output: Merged sequence $\mathcal{R}$
(1) $\mathcal{R} \leftarrow \emptyset$; $i \leftarrow 1$; $j \leftarrow 1$
(2) *while* $i \le m$ *and* $j \le n$ *do*
(3) *if* $k_i > \ell_j$ *then* append $(S_i, k_i)$ to $\mathcal{R}$; $i \leftarrow i+1$
(4) *else if* $\ell_j > k_i$ *then* append $(T_j, \ell_j)$ to $\mathcal{R}$; $j \leftarrow j+1$
(5) *else* // $k_i = \ell_j$
(6) $(x, h) \leftarrow$ LcpCompare($S_i, T_j, k_i$)
(7) *if* $x = "<"$ *then*
(8) append $(S_i, k_i)$ to $\mathcal{R}$; $i \leftarrow i+1$
(9) $\ell_j \leftarrow h$
(10) *else*
(11) append $(T_j, \ell_j)$ to $\mathcal{R}$; $j \leftarrow j+1$
(12) $k_i \leftarrow h$
(13) *while* $i \le m$ *do* append $(S_i, k_i)$ to $\mathcal{R}$; $i \leftarrow i+1$
(14) *while* $j \le n$ *do* append $(T_j, \ell_j)$ to $\mathcal{R}$; $j \leftarrow j+1$
(15) *return* $\mathcal{R}$



**Lemma 1.22:** StringMerge performs the merging correctly.

**Proof.** We will show that the following *invariant* holds at the beginning of each round in the loop on lines (2)–(12):

> Let $X$ be the last string appended to $\mathcal{R}$ (or $\varepsilon$ if $\mathcal{R} = \emptyset$). Then $k_i = lcp(X, S_i)$ and $\ell_j = lcp(X, T_j)$.

The invariant is clearly true in the beginning. We will show that the invariant is maintained and the smaller string is chosen in each round of the loop.

- If $k_i > \ell_j$, then $lcp(X, S_i) > lcp(X, T_j)$ and thus
  - $S_i < T_j$ by Lemma 1.19.
  - $lcp(S_i, T_j) = lcp(X, T_j)$ because by Lemma 1.18 $lcp(X, T_j) = \min\{lcp(X, S_i), lcp(S_i, T_j)\}$.

  Hence, the algorithm chooses the smaller string and maintains the invariant. The case $\ell_j > k_i$ is symmetric.

- If $k_i = \ell_j$, then clearly $lcp(S_i, T_j) \ge k_i$ and the call to LcpCompare is safe, and the smaller string is chosen. The update $\ell_j \leftarrow h$ or $k_i \leftarrow h$ maintains the invariant. □

**Theorem 1.23:** String mergesort sorts a set $\mathcal{R}$ of $n$ strings in $\mathcal{O}(dp(\mathcal{R}) + n \log n)$ time.

**Proof.** If the calls to LcpCompare took constant time, the time complexity would be $\mathcal{O}(n \log n)$ by the same argument as with the standard mergesort.

Whenever LcpCompare makes more than one, say $1 + t$ symbol comparisons, one of the lcp values stored with the strings increases by $t$. The lcp value stored with a string $S$ cannot become larger than $dp_{\mathcal{R}}(S)$. Therefore, the extra time spent in LcpCompare is bounded by $\mathcal{O}(dp(\mathcal{R}))$. $\square$

- Other comparison based sorting algorithms, for example heapsort and insertion sort, can be adapted for strings using the lcp comparison technique.



# 2. Sets of Strings

A balanced binary search tree is a powerful data structure that stores a *set of objects* and supports many operations including:

**Insert and Delete.**

**Lookup:** Find if a given object is in the set, and if it is, possibly return some data associated with the object.

**Range query:** Find all objects in a given range.

The time complexity of the operations for a set of size $n$ is $\mathcal{O}(\log n)$ (plus the size of the result) assuming constant time comparisons.

There are also alternative data structures, particularly if we do not need to support all operations:

- A *hash table* supports operations in constant time but does not support range queries.
- An *ordered array* is simpler, faster and more space efficient in practice, but does not support insertions and deletions.

A data structure is called *dynamic* if it supports insertions and deletions and *static* if not.



When the objects are strings, operations slow down:

- Comparison are slower. For example, the average case time complexity is $\mathcal{O}(\log n \log_\sigma n)$ for operations in a binary search tree storing a random set of strings.
- Computing a hash function is slower too.

For a string set $\mathcal{R}$, there are also new types of queries:

**Lcp query:** What is the length of the longest prefix of the query string $S$ that is also a prefix of some string in $\mathcal{R}$.

**Prefix query:** Find all strings in $\mathcal{R}$ that have $S$ as a prefix.

The prefix query is a special type of range query.



## Trie

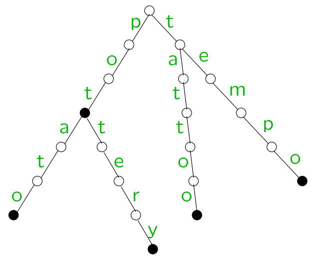
A trie is a *rooted tree* with the following properties:

- Edges are labelled with symbols from an alphabet $\Sigma$.
- The edges from any node $v$ to its children have all different labels.

Each node represents the string obtained by concatenating the symbols on the path from the root to that node.

The trie for a strings set $\mathcal{R}$, denoted by $trie(\mathcal{R})$, is the smallest trie that has nodes representing all the strings in $\mathcal{R}$.

**Example 2.1:** $trie(\mathcal{R})$ for $\mathcal{R} = \{\texttt{pot}, \texttt{potato}, \texttt{pottery}, \texttt{tattoo}, \texttt{tempo}\}$.



The time and space complexity of a trie depends on the implementation of the *child function*:

> For a node $v$ and a symbol $c \in \Sigma$, *child*$(v, c)$ is $u$ if $u$ is a child of $v$ and the edge $(v, u)$ is labelled with $c$, and *child*$(v, c) = \perp$ if $v$ has no such child.

There are many implementation options including:

**Array:** Each node stores an array of size $\sigma$. The space complexity is $\mathcal{O}(\sigma||\mathcal{R}||)$, where $||\mathcal{R}||$ is the total length of the strings in $\mathcal{R}$. The time complexity of the child operation is $\mathcal{O}(1)$.

**Binary tree:** Replace the array with a binary tree. The space complexity is $\mathcal{O}(||\mathcal{R}||)$ and the time complexity $\mathcal{O}(\log \sigma)$.

**Hash table:** One hash table for the whole trie, storing the values *child*$(v, c) \neq \perp$. Space complexity $\mathcal{O}(||\mathcal{R}||)$, time complexity $\mathcal{O}(1)$.

Array and hash table implementations require an integer alphabet; the binary tree implementation works for an ordered alphabet.



A common simplification in the analysis of tries is to assume a *constant alphabet*. Then the implementation does not matter:

- Insertion, deletion, lookup and lcp query for a string $S$ take $\mathcal{O}(|S|)$ time.
- Prefix query takes $\mathcal{O}(|S| + \ell)$ time, $\ell$ is the total length of the strings in the answer.

The potential slowness of prefix (and range) queries is one of the main drawbacks of tries.

Note that a trie is a complete representation of the strings. There is no need to store the strings elsewhere.



We can construct the trie using the following algorithm, which inserts the strings one at a time.

**Algorithm 2.2:** Trie construction
Input: Set of strings $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$
Output: $trie(\mathcal{R})$: $root$, *child*() and *rep*()

```
Create new node root
for i ← 1 to n do
    v ← root; j ← 0
    while child(v, S_i[j]) ≠ ⊥ do
        v ← child(v, S_i[j]); j ← j + 1
    while j < |S_i| do
        Create new node u
        child(v, S_i[j]) ← u
        v ← u; j ← j + 1
    rep(v) ← i
```

- The creation of a new node $v$ initializes *rep*$(v)$ and *child*$(v, c)$ to $\perp$ for all $c \in \Sigma$.
- After the execution, *rep*$(v) = i$ if $v$ represents $S_i$ and *rep*$(v) = \perp$ if $v$ does not represent any string in $\mathcal{R}$.



The data structures for a string set $\mathcal{R}$ become simpler, if $\mathcal{R}$ is *prefix free*.

**Definition 2.3:** A string set $\mathcal{R}$ is *prefix free* if no string in $\mathcal{R}$ is a prefix of another string in $\mathcal{R}$.

- For example, in $trie(\mathcal{R})$ for a prefix free $\mathcal{R}$, *rep*$(v) \neq \perp$ if and only if $v$ is a leaf. Then we need to store *rep*$(v)$ only for leaves. Without prefix freeness, some internal nodes can have *rep*$(v) \neq \perp$ too.

There is a simple way to make any string set prefix free:

- Let $\$ \notin \Sigma$ be an extra symbol satisfying $\$ < c$ for all $c \in \Sigma$.
- Append $\$$ to the end of every string in $\mathcal{R}$.