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.

Some Interesting Strings

The Fibonacci strings are defined by the recurrence:

$$F_0 = \varepsilon$$
$$F_1 = \texttt{b}$$
$$F_2 = \texttt{a}$$
$$F_i = F_{i-1}F_{i-2} \text{ for } i > 2$$

The infinite Fibonacci string is the limit F_∞.
For all $i > 1$, F_i is a prefix of F_∞.

Example 0.3:

$F_3 = \texttt{ab}$
$F_4 = \texttt{aba}$
$F_5 = \texttt{abaab}$
$F_6 = \texttt{abaababa}$
$F_7 = \texttt{abaababaabaab}$
$F_8 = \texttt{abaababaabaababaababa}$

Fibonacci strings have many interesting properties:

- $|F_i| = f_i$, where f_i is the ith Fibonacci *number*.
- F_∞ has exactly $k+1$ distinct factors of length k.
- For all $i > 1$, we can obtain F_i from F_{i-1} by applying the substitutions $\texttt{a} \mapsto \texttt{ab}$ and $\texttt{b} \mapsto \texttt{a}$ to every character.

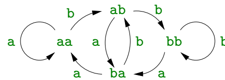
A De Bruijn sequence B_k of order k for an alphabet Σ of size σ is a cyclic string of length σ^k that contains every string of length k over the alphabet Σ as a factor exactly once. The cycle can be opened into a string of length $\sigma^k + k - 1$ with the same property.

Example 0.4: De Bruijn sequences for the alphabet $\{\texttt{a}, \texttt{b}\}$:

$$B_2 = \texttt{aabb}(\texttt{a})$$
$$B_3 = \texttt{aaababbb}(\texttt{aa})$$
$$B_4 = \texttt{aaaabaabbababbbb}(\texttt{aaa})$$

De Bruijn sequences are not unique. They can be constructed by finding Eulerian or Hamiltonian cycles in a De Bruijn graph.

Example 0.5: De Bruijn graph for the alphabet $\{\texttt{a}, \texttt{b}\}$ that can be used for constructing B_2 (Hamiltonian cycle) or B_3 (Eulerian cycle).



1. Sets of Strings

Basic operations on a set of objects include:

Insert: Add an object to the set

Delete: Remove an object from the set.

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

There can also be more complex queries:

Range query: Find all objects in a given range of values.

There are many other operations too but we will concentrate on these here.

An efficient execution of the operations requires that the set is stored as a suitable data structure.

- A (balanced) binary search tree supports the basic operations in $\mathcal{O}(\log n)$ time and range searching in $\mathcal{O}(\log n + r)$ time, where n is the size of the set and r is the size of the result.
- An ordered array supports lookup and range searching in the same time as binary search trees. It is simpler, faster and more space efficient in practice, but does not support insertions and deletions.
- A hash table supports the basic operations in constant time (usually randomized) but does not support range queries.

A data structure is called dynamic if it supports insertions and deletions (tree, hash table) and static if not (array). Static data structures are constructed once for the whole set of objects. In the case of an ordered array, this involves another important operation, sorting. Sorting can be done in $\mathcal{O}(n \log n)$ time using comparisons and even faster for integers.

The above time complexities assume that basic operations on the objects including comparisons can be performed in constant time. When the objects are strings, this is no more true:

- The worst case time for a string comparison is the length of the shorter string. Even the average case time for a random set of n strings is $\mathcal{O}(\log_\sigma n)$ in many cases, including for basic operations in a balanced binary search tree. We will show an even stronger result for sorting later. And sets of strings are rarely fully random.
- Computing a hash function is slower too. A good hash function depends on all characters and cannot be computed faster than the length of the string.

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

Prefix query: Find all strings in $\mathcal{R}$ that have the query string S as a prefix. This is a special type of range query.

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

Thus we need special set data structures and algorithms for strings.

Trie

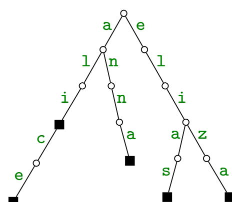
A simple but powerful data structure for a set of strings is the trie. It is a rooted tree with the following properties:

- Edges are labelled with symbols from an alphabet Σ.
- For every node v, the edges from v to its children have 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 string set $\mathcal{R}$, denoted by $trie(\mathcal{R})$, is the smallest trie that has nodes representing all the strings in $\mathcal{R}$. The nodes representing strings in $\mathcal{R}$ may be marked.

Example 1.1: $trie(\mathcal{R})$ for $\mathcal{R} = \{\texttt{ali}, \texttt{alice}, \texttt{anna}, \texttt{elias}, \texttt{eliza}\}$.



The trie is conceptually simple but it is not simple to implement efficiently. 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*$(v, c) = \bot$ (null) if v has no such child.

As an example, here is the insertion algorithm:

Algorithm 1.2: Insertion into trie
Input: $trie(\mathcal{R})$ and a string $S[0..m) \notin \mathcal{R}$
Output: $trie(\mathcal{R} \cup \{S\})$
(1) $v \leftarrow root$; $j \leftarrow 0$
(2) **while** *child*$(v, S[j]) \neq \bot$ **do**
(3) $\quad v \leftarrow$ *child*$(v, S[j])$; $j \leftarrow j + 1$
(4) **while** $j < m$ **do**
(5) $\quad$ Create new node u (initializes *child*(u, c) to $\bot$ for all $c \in \Sigma$)
(6) $\quad$ *child*$(v, S[j]) \leftarrow u$
(7) $\quad v \leftarrow u$; $j \leftarrow j + 1$
(8) Mark v as representative of S

There are many implementation options for the child function including:

Array: Each node stores an array of size σ. The space complexity is $\mathcal{O}(\sigma N)$, where N is the number of nodes in $trie(\mathcal{R})$. The time complexity of the child operation is $\mathcal{O}(1)$. Requires an integer alphabet.

Binary tree: Replace the array with a binary tree. The space complexity is $\mathcal{O}(N)$ and the time complexity $\mathcal{O}(\log \sigma)$. Works for an ordered alphabet.

Hash table: One hash table for the whole trie, storing the values *child*$(v, c) \neq \bot$. Space complexity $\mathcal{O}(N)$, time complexity $\mathcal{O}(1)$. Requires an integer 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.

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

Prefix free string sets

Many data structures and algorithms for a string set $\mathcal{R}$ become simpler if $\mathcal{R}$ is prefix free.

Definition 1.3: A string set $\mathcal{R}$ is prefix free if no string in $\mathcal{R}$ is a prefix of another string in $\mathcal{R}$.

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

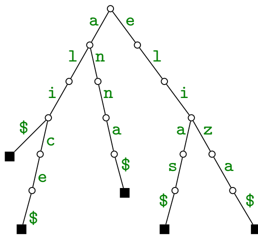
- Let $\$ \notin \Sigma$ be an extra symbol satisfying $\$ < c$ for all $c \in \Sigma$.
- Append $ to the end of every string in $\mathcal{R}$.

This has little or no effect on most operations on the set. The length of each string increases by one only, and the additional symbol could be there only virtually.

Example 1.4: The set {ali, alice, anna, elias, eliza} is not prefix free because ali is a prefix of alice, but {ali$, alice$, anna$, elias$, eliza$} is prefix free.

If $\mathcal{R}$ is prefix free, the leaves of $trie(\mathcal{R})$ represent exactly $\mathcal{R}$. This simplifies the implementation of the trie.

Example 1.5: The trie for {ali$, alice$, anna$, elias$, eliza$}.



Compact Trie

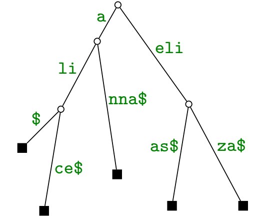
Tries suffer from a large number of nodes, close to $||\mathcal{R}||$ in the worst case.

- For a string set $\mathcal{R}$, we use $|\mathcal{R}|$ to denote the number of strings in $\mathcal{R}$ and $||\mathcal{R}||$ to denote the total length of the strings in $\mathcal{R}$.

The space requirement can be problematic, since typically each node needs much more space than a single symbol.

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

Example 1.6: Compact trie for {ali$, alice$, anna$, elias$, eliza$}.



The space complexity of a compact trie is $\mathcal{O}(|\mathcal{R}|)$ (in addition to the strings):

- In a compact trie, 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 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.

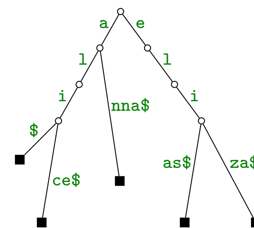
The time complexities are the same or better than for tries:

- An insertion adds and a deletion removes at most two nodes.
- Lookups may execute fewer calls to the child operation, though the worst case complexity is the same.
- Prefix and range queries are faster even on a constant alphabet (exercise).

There is also an intermediate form of trie called leaf-path-compacted trie, where branchless path segments are compacted only when they end in a leaf.

- Typically (though not in the worst case) this achieves most of the advantages of a compact trie.
- For trie algorithms, this means stopping the normal search, when only one string is remaining in the subtree.

Example 1.7: Leaf-path-compacted trie for {ali$, alice$, anna$, elias$, eliza$}.



Ternary Trie

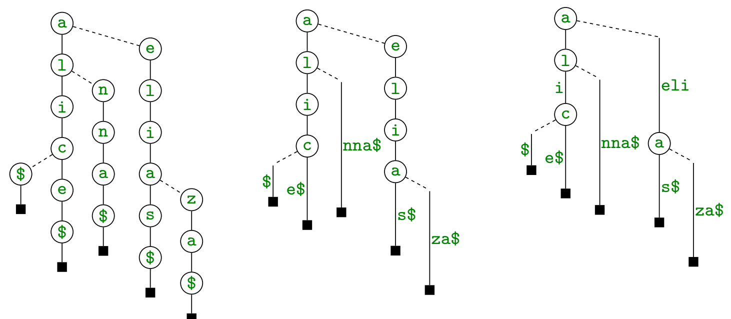
Tries can be implemented for ordered alphabets but a bit awkwardly using a comparison-based child function. Ternary trie is a simpler data structure based on symbol comparisons.

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.

There are also compact ternary tries and leaf-path-compated ternary tries based on compacting branchless path segments.

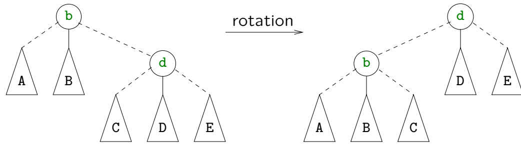
Example 1.8: Ternary tries for {ali$, alice$, anna$, elias$, eliza$}.



Ternary tries have the same asymptotic size as the corresponding (σ-ary) 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.

Note that there is no restriction on the size of the middle subtree.

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).

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 1.9: 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 1.10: 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 $A[\ell] < B[\ell]$.

An important concept for sets of strings is the LCP (longest common prefix) array and its sum.

Definition 1.11: 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 so that $LCP_{\mathcal{R}}[1] = 0$ and for $i \in [2..n]$

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

Furthermore, the LCP array sum is

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

Example 1.12: For $\mathcal{R} = \{$ali\$, alice\$, anna\$, elias\$, eliza\$$\}$, $\Sigma LCP(\mathcal{R}) = 7$ and the LCP array is:

$LCP_{\mathcal{R}}$	
0	ali\$
3	alice\$
1	anna\$
0	elias\$
3	eliza\$

A variant of the LCP array sum is sometimes useful:

Definition 1.13: For a string S and a string set $\mathcal{R}$, define

$$lcp(S, \mathcal{R}) = \max\{lcp(S, T) \mid T \in \mathcal{R}\}$$
$$\Sigma lcp(\mathcal{R}) = \sum_{S \in \mathcal{R}} lcp(S, \mathcal{R} \setminus \{S\})$$

The relationship of the two measures is shown by the following two results:

Lemma 1.14: For $i \in [2..n]$, $LCP_{\mathcal{R}}[i] = lcp(S_i, \{S_1, \ldots, S_{i-1}\})$.

Lemma 1.15: $\Sigma LCP(\mathcal{R}) \leq \Sigma lcp(\mathcal{R}) \leq 2 \cdot \Sigma LCP(\mathcal{R})$.

The proofs are left as an exercise.

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. It is easy to see that $dp(S, \mathcal{R} \setminus S) = lcp(S, \mathcal{R} \setminus S) + 1$ (at least for a prefix free $\mathcal{R}$).

Example 1.16: For $\mathcal{R} = \{$ali\$, alice\$, anna\$, elias\$, eliza\$$\}$, $\Sigma lcp(\mathcal{R}) = 13$ and $\Sigma dp(\mathcal{R}) = 18$.

Theorem 1.17: 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.

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 1.18: 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 1.18. 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). □

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

Theorem 1.19: 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 1.20: 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 1.21: StringQuicksort($\mathcal{R}, \ell$)
Input: (Multi)set $\mathcal{R}$ of strings and the length ℓ of their common prefix.
Output: $\mathcal{R}$ 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 1.22: A possible partitioning, when $\ell = 2$.

```
al p habet                al i gnment
al i gnment               al g orithm
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
```

Theorem 1.23: 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 1.23. 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 $\mathcal{O}(n \log n)$. □

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 1.24: 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) $tmp \leftarrow C[i]$; $C[i] \leftarrow sum$; $sum \leftarrow sum + tmp$
(6) for $i \leftarrow 1$ to n do // distribute
(7) $J[C[k_i]] \leftarrow k_i$; $C[k_i] \leftarrow C[k_i] + 1$
(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 $\Omega(\Sigma LCP(\mathcal{R}) + n \log n)$ 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.

There are two types of radix sorting:

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

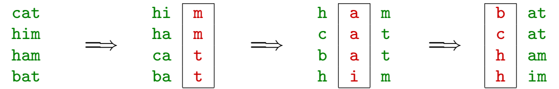
LSD radix sort starts sorting from the end of strings (least significant digit).

The LSD radix sort algorithm is very simple.

Algorithm 1.25: LSDRadixSort($\mathcal{R}$)
Input: (Multi)set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings of length m over alphabet $[0..\sigma)$.
Output: $\mathcal{R}$ in ascending 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 replaced by $S_i[\ell]$). The time complexity is $\mathcal{O}(|\mathcal{R}| + \sigma)$.
- The stability of counting sort is essential.

Example 1.26: $\mathcal{R} = \{$cat, him, ham, bat$\}$.



It is easy to show that after i rounds, the strings are sorted by suffix of length i. Thus, they are fully sorted at the end.

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

Theorem 1.27: 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 $\mathcal{R}$.

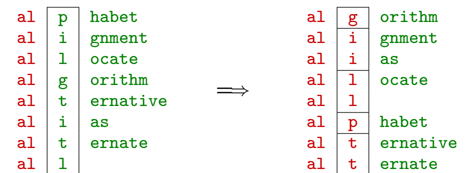
Proof. Assume all strings have length m. The LSD radix sort performs m rounds with each round taking $\mathcal{O}(n + \sigma)$ time. The total time is $\mathcal{O}(mn + m\sigma) = \mathcal{O}(||\mathcal{R}|| + m\sigma)$.

The case of variable lengths is left as an exercise. □

- The weakness of LSD radix sort is that it uses $\Omega(||\mathcal{R}||)$ time even when $\Sigma LCP(\mathcal{R})$ 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 1.28: MSD radix sort partitioning.



Algorithm 1.29: MSDRadixSort($\mathcal{R}$, ℓ)
Input: (Multi)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 ascending lexicographical order.
(1) if $|\mathcal{R}| < \sigma$ then return StringQuicksort($\mathcal{R}$, ℓ)
(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($\mathcal{R}$, ℓ)
(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 1.30: MSD radix sort sorts a set $\mathcal{R}$ of n strings over the alphabet $[0..\sigma)$ in $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n \log \sigma)$ time.

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

- The time excluding the recursive calls 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.
- At most $dp(S, \mathcal{R} \setminus \{S\}) \leq lcp(S, \mathcal{R} \setminus \{S\}) + 1$ symbols in S will be accessed by the algorithm. Thus the total time spent in this kind of calls is $\mathcal{O}(\Sigma dp(\mathcal{R})) = \mathcal{O}(\Sigma lcp(\mathcal{R}) + n) = \mathcal{O}(\Sigma LCP(\mathcal{R}) + n)$.

The calls for a subsets of size $k < \sigma$ are handled by string quicksort. Each string is involved in at most one such call. Therefore, the total time over all calls to string quicksort is $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n \log \sigma)$. □

- There exists a more complicated variant of MSD radix sort with time complexity $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n + \sigma)$.
- $\Omega(\Sigma LCP(\mathcal{R}) + n)$ 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.