58093 String Processing Algorithms

Lectures, Autumn 2014, 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
- Many algorithmic techniques are first developed for strings and later generalized for more complex types of data such as graphs.

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 problems:

- 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 and analysing algorithms rather than alphabets:

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

The assumption of a constant alphabet in the analysis of an algorithm often indicates one of two things:

- The effect of the alphabet on the complexity is complicated and the constant alphabet assumption is used to *simplify the analysis*.
- The time or space complexity of the algorithm is heavily (e.g., linearly) dependent on the alphabet size and the algorithm is effectively *unusable for large alphabets*.

An algorithm is called alphabet-independent if its complexity has no dependence on the alphabet size.

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:

$$S = S[1..n] = S[1]S[2]\dots S[n]$$
$$T = T[0..n) = T[0]T[1]\dots T[n-1]$$

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

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

- $\alpha, \beta \in \Sigma^*$
- $x = a_1a_2\dots a_k$ 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.

Some Interesting Strings

The Fibonacci strings are defined by the recurrence:

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

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

Example 0.3:

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

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.

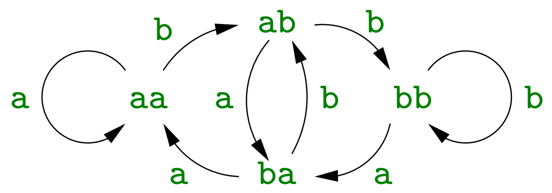
A De Bruijn sequence B_k of order k for an alphabet Σ of size σ is a cyclic string of length σ^k that contains every string of length k over the alphabet Σ as a factor exactly once. The cycle can be opened into a string of length $\sigma^k + k - 1$ with the same property.

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

$$\begin{aligned} B_2 &= \mathtt{aabb}(\mathtt{a}) \\ B_3 &= \mathtt{aaababbb}(\mathtt{aa}) \\ B_4 &= \mathtt{aaaabaabbababbbb}(\mathtt{aaa}) \end{aligned}$$

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 $\{\mathtt{a}, \mathtt{b}\}$ 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

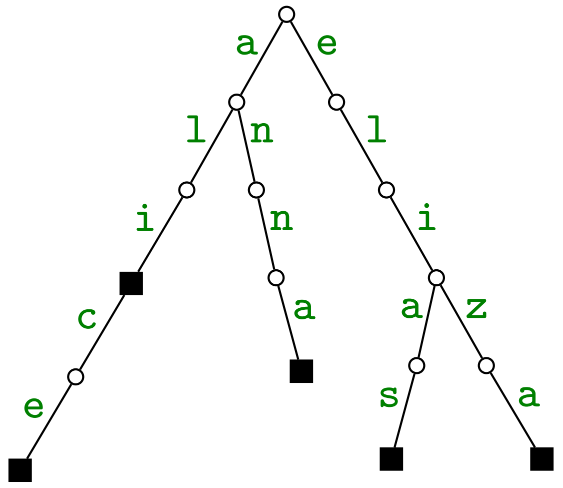
A simple but powerful data structure for a set of strings is the trie. It is a rooted tree with the following properties:

- 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*$(v, c) = \perp$ (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$(v, S[j]) \neq \perp$ do
(3)     $v \leftarrow$ child$(v, S[j])$; $j \leftarrow j + 1$
(4) while $j < m$ do
(5)     Create new node $u$ (initializes child$(u, c)$ to $\perp$ for all $c \in \Sigma$)
(6)     child$(v, S[j]) \leftarrow u$
(7)     $v \leftarrow u$; $j \leftarrow j + 1$
(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*$(v, c) \neq \perp$. 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:

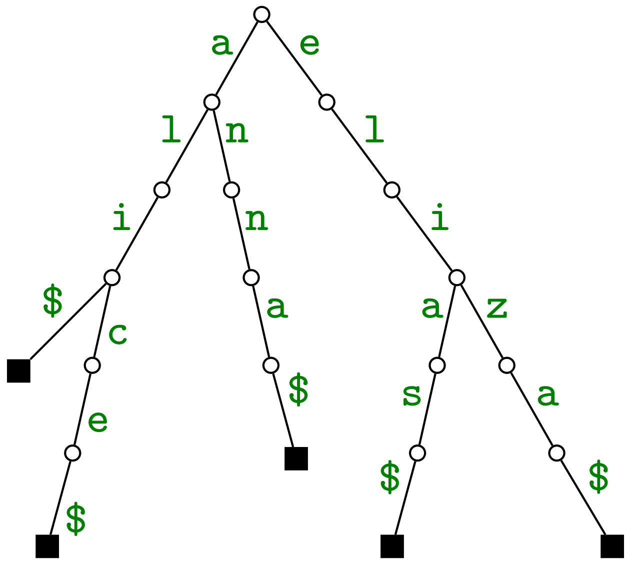
- Let $\texttt{\$} \not\in \Sigma$ be an extra symbol satisfying $\texttt{\$} < c$ for all $c \in \Sigma$.
- Append $\texttt{\$}$ to the end of every string in $\mathcal{R}$.

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 $\{\texttt{ali}, \texttt{alice}, \texttt{anna}, \texttt{elias}, \texttt{eliza}\}$ is not prefix free because `ali` is a prefix of `alice`, but $\{\texttt{ali\$}, \texttt{alice\$}, \texttt{anna\$}, \texttt{elias\$}, \texttt{eliza\$}\}$ 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

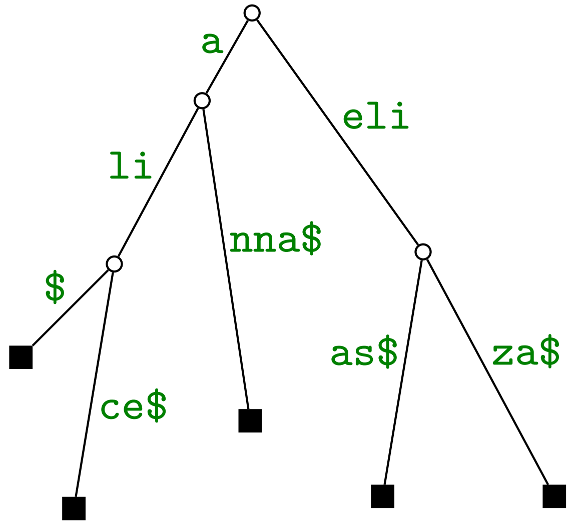
Tries suffer from a large number of nodes, close to $||\mathcal{R}||$ in the worst case.

- 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.

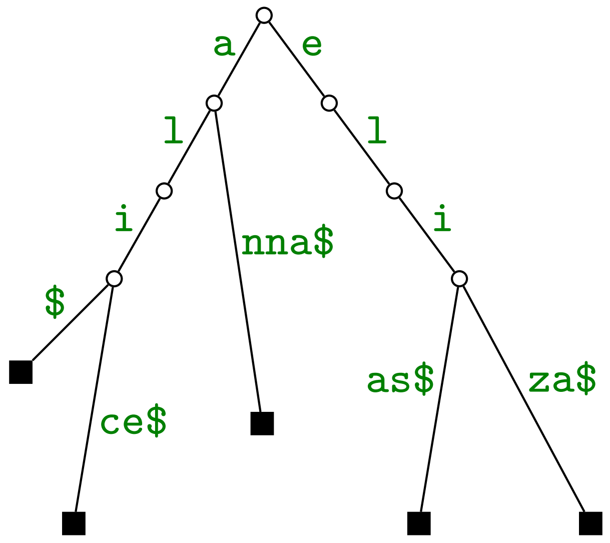
The time complexities are the same or better than for tries:

- 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

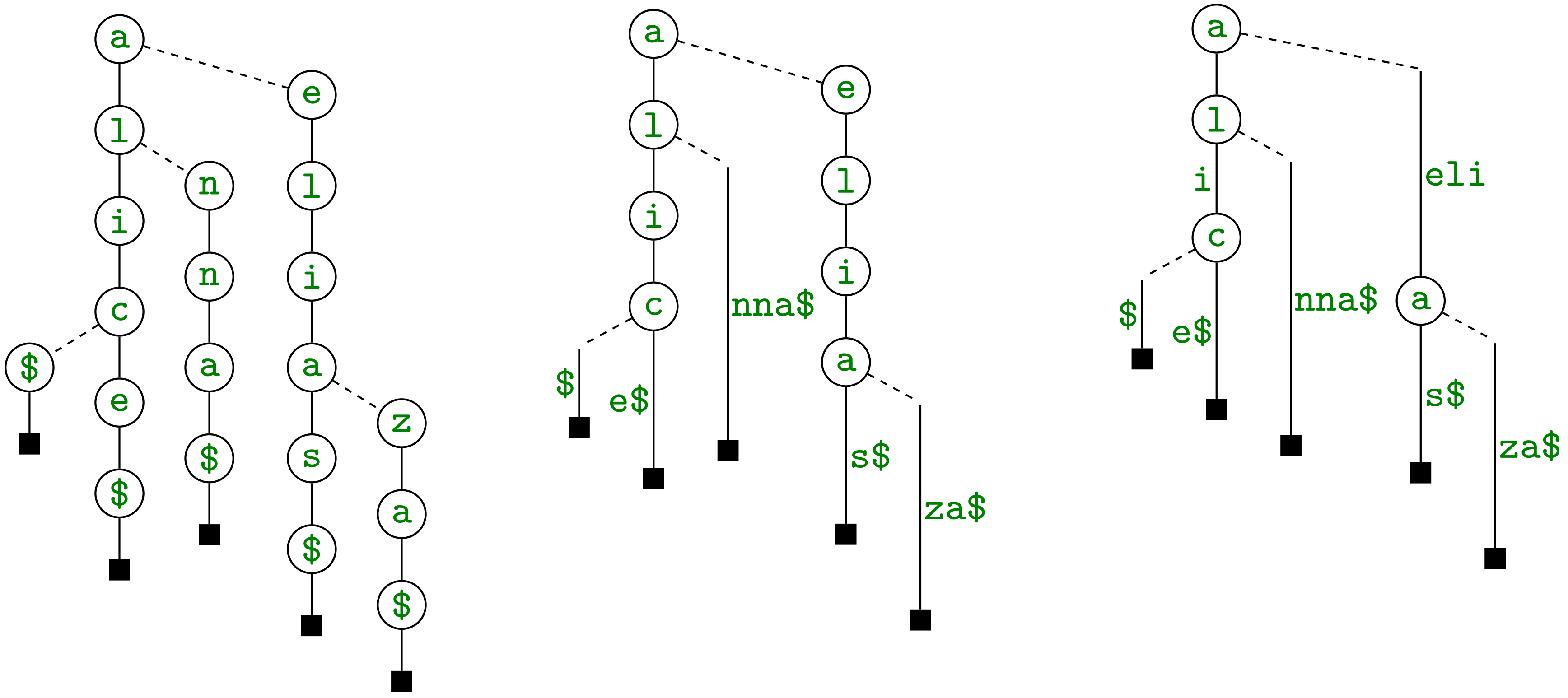
Tries can be implemented for ordered alphabets but a bit awkwardly using a comparison-based child function. Ternary trie is a simpler data structure based on symbol comparisons.

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.

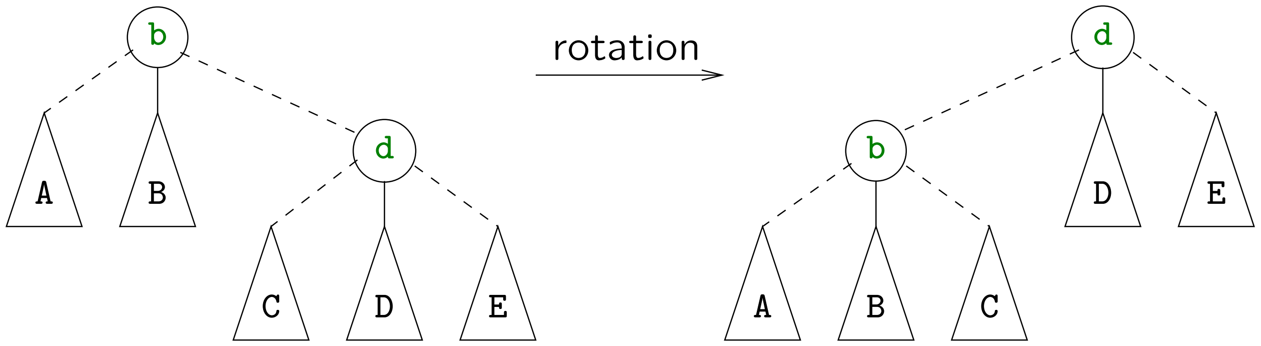
Example 1.8: Ternary tries for {ali\$, alice\$, anna\$, elias\$, eliza\$}.



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

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

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:

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

$\square$

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). $\square$

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. $\square$

- 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: 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	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 1.23: 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($\mathcal{R}$,$\ell$)
(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: $\mathcal{R} = \{\texttt{cat}, \texttt{him}, \texttt{ham}, \texttt{bat}\}$.

$$\begin{array}{l}\texttt{cat}\\\texttt{him}\\\texttt{ham}\\\texttt{bat}\end{array}\Longrightarrow\begin{array}{l}\texttt{hi}\\\texttt{ha}\\\texttt{ca}\\\texttt{ba}\end{array}\boxed{\begin{array}{c}\texttt{m}\\\texttt{m}\\\texttt{t}\\\texttt{t}\end{array}}\Longrightarrow\begin{array}{l}\texttt{h}\\\texttt{c}\\\texttt{b}\\\texttt{h}\end{array}\boxed{\begin{array}{c}\texttt{a}\\\texttt{a}\\\texttt{a}\\\texttt{i}\end{array}}\begin{array}{l}\texttt{m}\\\texttt{t}\\\texttt{t}\\\texttt{m}\end{array}\Longrightarrow\boxed{\begin{array}{c}\texttt{b}\\\texttt{c}\\\texttt{h}\\\texttt{h}\end{array}}\begin{array}{l}\texttt{at}\\\texttt{at}\\\texttt{am}\\\texttt{im}\end{array}$$

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.

al	p	habet		al	g	orithm
al	i	gnment		al	i	gnment
al	l	ocate		al	i	as
al	g	orithm	$\Longrightarrow$	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 1.29: 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 ℓ of their common prefix.
Output: $\mathcal{R}$ in ascending 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 1.30: 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 $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n \log \sigma)$.

$\square$

- 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.

Lcp-Comparisons

General (non-string) 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 processing the obtained information immediately after each symbol comparison.

An opposite approach is to replace a standard string comparison with an lcp-comparison, which is the operation LcpCompare(A, B, k):

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

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

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

Lemma 1.31: 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 $lcp(A,C) = \min\{lcp(A,B), lcp(B,C)\}$.

(c) If $lcp(A,B) \neq lcp(B,C)$, 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, $lcp(A,C) = \ell$.

(c) Now $lcp(A,B) < lcp(B,C)$. If $lcp(A,C) > \min\{lcp(A,B), lcp(B,C)\}$, then $lcp(A,B) < \min\{lcp(A,C), lcp(B,C)\}$, which violates (a).

$\square$

The above means that the three lcp values between three strings can never be three different values. At least two of them are the same and the third one is the same or bigger.

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

Lemma 1.32: Let A, B, B' and C be strings such that $A \leq B \leq C$ and $A \leq B' \leq C$.

(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 $B \geq B'$. Then by Lemma 1.31, $lcp(A, B) = \min\{lcp(A, B'), lcp(B', B)\} \leq lcp(A, B')$, which is a contradiction. $\square$

Intuitively, the above result makes sense if you think of $lcp(\cdot, \cdot)$ as a *measure of similarity* between two strings. The higher the lcp, the closer the two strings are lexicographically.

String Mergesort

String mergesort is a string sorting algorithm that uses lcp-comparisons. It has the same structure as the standard mergesort: sort the first half and the second half separately, and then merge the results.

Algorithm 1.33: StringMergesort($\mathcal{R}$)
Input: Set $\mathcal{R} = \{S_1, S_2, \dots, S_n\}$ of strings.
Output: $\mathcal{R}$ sorted and augmented with $LCP_{\mathcal{R}}$ values.

```
(1) if $|\mathcal{R}| = 1$ then return $((S_1, 0))$
(2) $m \leftarrow \lfloor n/2 \rfloor$
(3) $\mathcal{P} \leftarrow$ StringMergesort($\{S_1, S_2, \dots, S_m\}$)
(4) $\mathcal{Q} \leftarrow$ StringMergesort($\{S_{m+1}, S_{m+2}, \dots, S_n\}$)
(5) return StringMerge($\mathcal{P}, \mathcal{Q}$)
```

The output is of the form

$$((T_1, \ell_1), (T_2, \ell_2), \dots, (T_n, \ell_n))$$

where $\ell_i = lcp(T_i, T_{i-1})$ for $i > 1$ and $\ell_1 = 0$. In other words, $\ell_i = LCP_{\mathcal{R}}[i]$.

Thus 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 1.34: 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 1.35: 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 ε 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 1.32.
  - $lcp(S_i, T_j) = lcp(X, T_j)$ because, by Lemma 1.31, $lcp(X, T_j) = \min\{lcp(X, S_i), lcp(S_i, T_j)\}$.

  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 $lcp(S_i, T_j) \geq k_i$ 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. $\square$

Theorem 1.36: String mergesort sorts a set $\mathcal{R}$ of n strings in $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n \log n)$ 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 $t + 1$ symbol comparisons, one of the lcp values stored with the strings increases by t. Since the sum of the final lcp values is exactly $\Sigma LCP(\mathcal{R})$, the extra time spent in LcpCompare is bounded by $\mathcal{O}(\Sigma LCP(\mathcal{R}))$.

$\square$

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

String Binary Search

An ordered array is a simple static data structure supporting queries in $\mathcal{O}(\log n)$ time using binary search.

Algorithm 1.37: Binary search
Input: Ordered set $R = \{k_1, k_2, \ldots, k_n\}$, query value x.
Output: The number of elements in R that are smaller than x.

```
(1) $left \leftarrow 0$; $right \leftarrow n + 1$        // output value is in the range $[left..right)$
(2) while $right - left > 1$ do
(3)     $mid \leftarrow \lfloor (left + right)/2 \rfloor$
(4)     if $k_{mid} < x$ then $left \leftarrow mid$
(5)     else $right \leftarrow mid$
(6) return $left$
```

With strings as elements, however, the query time is

- $\mathcal{O}(m \log n)$ in the worst case for a query string of length m
- $\mathcal{O}(\log n \log_\sigma n)$ on average for a random set of strings.

We can use the lcp-comparison technique to improve binary search for strings. The following is a key result.

Lemma 1.38: Let A, B, B' and C be strings such that $A \leq B \leq C$ and $A \leq B' \leq C$. Then $lcp(B, B') \geq lcp(A, C)$.

Proof. Let $B_{min} = \min\{B, B'\}$ and $B_{max} = \max\{B, B'\}$. By Lemma 1.31,

$$\begin{aligned} lcp(A, C) &= \min(lcp(A, B_{max}), lcp(B_{max}, C)) \\ &\leq lcp(A, B_{max}) = \min(lcp(A, B_{min}), lcp(B_{min}, B_{max})) \\ &\leq lcp(B_{min}, B_{max}) = lcp(B, B') \end{aligned}$$

□

During the binary search of P in $\{S_1, S_2, \ldots, S_n\}$, the basic situation is the following:

- We want to compare P and S_{mid}.
- We have already compared P against S_{left} and S_{right}, and we know that $S_{left} \leq P, S_{mid} \leq S_{right}$.
- By using lcp-comparisons, we know $lcp(S_{left}, P)$ and $lcp(P, S_{right})$.

By Lemmas 1.31 and 1.38,

$$lcp(P, S_{mid}) \geq lcp(S_{left}, S_{right}) = \min\{lcp(S_{left}, P), lcp(P, S_{right})\}$$

Thus we can skip $\min\{lcp(S_{left}, P), lcp(P, S_{right})\}$ first characters when comparing P and S_{mid}.

Algorithm 1.39: String binary search (without precomputed lcps)
Input: Ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$, query string P.
Output: The number of strings in $\mathcal{R}$ that are smaller than P.

```
 (1) $left \leftarrow 0$; $right \leftarrow n+1$
 (2) $llcp \leftarrow 0$                // $llcp = lcp(S_{left}, P)$
 (3) $rlcp \leftarrow 0$                // $rlcp = lcp(P, S_{right})$
 (4) while $right - left > 1$ do
 (5)     $mid \leftarrow \lfloor (left + right)/2 \rfloor$
 (6)     $mlcp \leftarrow \min\{llcp, rlcp\}$
 (7)     $(x, mlcp) \leftarrow$ LcpCompare$(P, S_{mid}, mlcp)$
 (8)     if $x =$ "$<$" then $right \leftarrow mid$; $rlcp \leftarrow mclp$
 (9)     else $left \leftarrow mid$; $llcp \leftarrow mclp$
(10) return $left$
```

- The average case query time is now $\mathcal{O}(\log n)$.
- The worst case query time is still $\mathcal{O}(m \log n)$ (exercise).

We can further improve string binary search using precomputed information about the lcp's between the strings in $\mathcal{R}$.

Consider again the basic situation during string binary search:

- We want to compare P and S_{mid}.
- We have already compared P against S_{left} and S_{right}, and we know $lcp(S_{left}, P)$ and $lcp(P, S_{right})$.

The values $left$ and $right$ are fully determined by mid independently of P. That is, P only determines whether the search ends up at position mid at all, but if it does, $left$ and $right$ are always the same.

Thus, we can precompute and store the values

$$LLCP[mid] = lcp(S_{left}, S_{mid})$$
$$RLCP[mid] = lcp(S_{mid}, S_{right})$$

Now we know all lcp values between P, S_{left}, S_{mid}, S_{right} except $lcp(P, S_{mid})$. The following lemma shows how to utilize this.

Lemma 1.40: Let A, B, B' and C be strings such that $A \leq B \leq C$ and $A \leq B' \leq C$.

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

(b) If $lcp(A, B) < lcp(A, B')$, then $B > B'$ and $lcp(B, B') = lcp(A, B)$.

(c) If $lcp(B, C) > lcp(B', C)$, then $B > B'$ and $lcp(B, B') = lcp(B', C)$.

(d) If $lcp(B, C) < lcp(B', C)$, then $B < B'$ and $lcp(B, B') = lcp(B, C)$.

(e) If $lcp(A, B) = lcp(A, B')$ and $lcp(B, C) = lcp(B', C)$, then $lcp(B, B') \geq \max\{lcp(A, B), lcp(B, C)\}$.

Proof. Cases (a)–(d) are symmetrical, we show (a). $B < B'$ follows from Lemma 1.32. Then by Lemma 1.31, $lcp(A, B') = \min\{lcp(A, B), lcp(B, B')\}$. Since $lcp(A, B') < lcp(A, B)$, we must have $lcp(A, B') = lcp(B, B')$.

In case (e), we use Lemma 1.31:

$$lcp(B, B') \geq \min\{lcp(A, B), lcp(A, B')\} = lcp(A, B)$$
$$lcp(B, B') \geq \min\{lcp(B, C), lcp(B', C)\} = lcp(B, C)$$

Thus $lcp(B, B') \geq \max\{lcp(A, B), lcp(B, C)\}$. □

Algorithm 1.41: String binary search (with precomputed lcps)
Input: Ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$, arrays LLCP and RLCP, query string P.
Output: The number of strings in $\mathcal{R}$ that are smaller than P.

```
 (1)  $left \leftarrow 0$; $right \leftarrow n + 1$
 (2)  $llcp \leftarrow 0$; $rlcp \leftarrow 0$
 (3)  while $right - left > 1$ do
 (4)      $mid \leftarrow \lfloor (left + right)/2 \rfloor$
 (5)      if $LLCP[mid] > llcp$ then $left \leftarrow mid$
 (6)      else if $LLCP[mid] < llcp$ then $right \leftarrow mid$; $rlcp \leftarrow LLCP[mid]$
 (7)      else if $RLCP[mid] > rlcp$ then $right \leftarrow mid$
 (8)      else if $RLCP[mid] < rlcp$ then $left \leftarrow mid$; $llcp \leftarrow RLCP[mid]$
 (9)      else
(10)          $mlcp \leftarrow \max\{llcp, rlcp\}$
(11)          $(x, mlcp) \leftarrow \mathrm{LcpCompare}(P, S_{mid}, mlcp)$
(12)          if $x =$ "$<$" then $right \leftarrow mid$; $rlcp \leftarrow mclp$
(13)          else $left \leftarrow mid$; $llcp \leftarrow mclp$
(14)  return $left$
```

Theorem 1.42: An ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ can be preprocessed in $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n)$ time and $\mathcal{O}(n)$ space so that a binary search with a query string P can be executed in $\mathcal{O}(|P| + \log n)$ time.

Proof. The values $LLCP[mid]$ and $RLCP[mid]$ can be computed in $\mathcal{O}(lcp(S_{mid}, \mathcal{R} \setminus \{S_{mid}\}) + 1)$ time. Thus the arrays $LLCP$ and $RLCP$ can be computed in $\mathcal{O}(\Sigma lcp(\mathcal{R}) + n) = \mathcal{O}(\Sigma LCP(\mathcal{R}) + n)$ time and stored in $\mathcal{O}(n)$ space.

The main while loop in Algorithm 1.41 is executed $\mathcal{O}(\log n)$ times and everything except LcpCompare on line (11) needs constant time.

If a given LcpCompare call performs $t + 1$ symbol comparisons, $mclp$ increases by t on line (11). Then on lines (12)–(13), either $llcp$ or $rlcp$ increases by at least t, since $mlcp$ was $\max\{llcp, rlcp\}$ before LcpCompare. Since $llcp$ and $rlcp$ never decrease and never grow larger than $|P|$, the total number of extra symbol comparisons in LcpCompare during the binary search is $\mathcal{O}(|P|)$. □

Other comparison-based data structures such as binary search trees can be augmented with lcp information in the same way (study groups).

Hashing and Fingerprints

Hashing is a powerful technique for dealing with strings based on mapping each string to an integer using a hash function:

$$H : \Sigma^* \to [0..q) \subset \mathbb{N}$$

The most common use of hashing is with hash tables. Hash tables come in many flavors that can be used with strings as well as with any other type of object with an appropriate hash function. A drawback of using a hash table to store a set of strings is that they do not support lcp and prefix queries.

Hashing is also used in other situations, where one needs to check whether two strings S and T are the same or not:

- If $H(S) \neq H(T)$, then we must have $S \neq T$.

- If $H(S) = H(T)$, then $S = T$ and $S \neq T$ are both possible.
  If $S \neq T$, this is called a collision.

When used this way, the hash value is often called a fingerprint, and its range $[0..q)$ is typically large as it is not restricted by a hash table size.

Any good hash function must depend on all characters. Thus computing $H(S)$ needs $\Omega(|S|)$ time, which can defeat the advantages of hashing:

- A plain comparison of two strings is faster than computing the hashes.

- The main strength of hash tables is the support for constant time insertions and deletions, but inserting a string S into a hash table needs $\Omega(|S|)$ time when the hash computation time is included. Compare this to the $\mathcal{O}(|S|)$ time for a trie under a constant alphabet and the $\mathcal{O}(|S| + \log n)$ time for a ternary trie.

However, a hash table can still be competitive in practice. Furthermore, there are situations, where a full computation of the hash function can be avoided:

- A hash value can be computed once, stored, and used many times.

- Some hash functions can be computed more efficiently for a related set of strings. An example is the Karp–Rabin hash function.

Definition 1.43: The Karp–Rabin hash function for a string $S = s_0 s_1 \dots s_{m-1}$ over an integer alphabet is

$$H(S) = (s_0 r^{m-1} + s_1 r^{m-2} + \cdots + s_{m-2} r + s_{m-1}) \bmod q$$

for some fixed positive integers q and r.

Lemma 1.44: For any two strings A and B,

$$H(AB) = (H(A) \cdot r^{|B|} + H(B)) \bmod q$$
$$H(B) = (H(AB) - H(A) \cdot r^{|B|}) \bmod q$$

Proof. Without the modulo operation, the result would be obvious. The modulo does not interfere because of the rules of modular arithmetic:

$$(x + y) \bmod q = ((x \bmod q) + (y \bmod q)) \bmod q$$
$$(xy) \bmod q = ((x \bmod q)(y \bmod q)) \bmod q$$

$\square$

Thus we can quickly compute $H(AB)$ from $H(A)$ and $H(B)$, and $H(B)$ from $H(AB)$ and $H(A)$. We will see applications of this later.

If q and r are *coprime*, then r has a multiplicative inverse r^{-1} modulo q, and we can also compute $H(A) = ((H(AB) - H(B)) \cdot (r^{-1})^{|B|}) \bmod q$.

The parameters q and r have to be chosen with some care to ensure that collisions are rare for any reasonable set of strings.

- The original choice is $r = \sigma$ and q is a large prime.

- Another possibility is that q is a power of two and r is a small prime ($r = 37$ has been suggested). This is faster in practice, because the slow modulo operations can be replaced by bitwise shift operations. If $q = 2^w$, where w is the machine word size, the modulo operations can be omitted completely.

- If q and r were both powers of two, then only the last $\lceil (\log q)/\log r \rceil$ characters of the string would affect the hash value. More generally, q and r should be coprime, i.e, have no common divisors other than 1.

- The hash function can be randomized by choosing q or r randomly. For example, if q is a prime and r is chosen uniformly at random from $[0..q)$, the probability that two strings of length m collide is at most m/q.

- A random choice over a set of possibilities has the additional advantage that we can change the choice if the first choice leads to too many collisions.

Automata

Finite automata are a well known way of representing sets of strings. In this case, the set is often called a (regular) language.

A trie is a special type of an automaton.

- The root is the initial state, the leaves are accept states, ...
- Trie is generally not a *minimal* automaton.
- Trie techniques including path compaction can be applied to automata.

Automata are much more powerful than tries in representing languages:

- Infinite languages
- Nondeterministic automata
- Even an acyclic, deterministic automaton can represent a language of exponential size.

Automata support set inclusion testing but not other trie operations:

- No insertions and deletions
- No satellite data, i.e., data associated to each string

Sets of Strings: Summary

Efficient algorithms and data structures for sets of strings:

- Storing and searching: trie and ternary trie and their compact versions, string binary search, Karp–Rabin hashing.
- Sorting: string quicksort and mergesort, LSD and MSD radix sort.

Lower bounds:

- Many of the algorithms are optimal.
- General purpose algorithms are asymptotically slower.

The central role of longest common prefixes:

- LCP array $LCP_{\mathcal{R}}$ and its sum $\Sigma LCP(\mathcal{R})$.
- Lcp-comparison technique.

2. Exact String Matching

Let $T = T[0..n)$ be the text and $P = P[0..m)$ the pattern. We say that P occurs in T at position j if $T[j..j+m) = P$.

Example: $P =$ `aine` occurs at position 6 in $T =$ `karjalainen`.

In this part, we will describe algorithms that solve the following problem.

Problem 2.1: Given text $T[0..n)$ and pattern $P[0..m)$, report the first position in T where P occurs, or n if P does not occur in T.

The algorithms can be easily modified to solve the following problems too.

- Existence: Is P a factor of T?
- Counting: Count the number of occurrences of P in T.
- Listing: Report all occurrences of P in T.

The naive, brute force algorithm compares P against $T[0..m)$, then against $T[1..1+m)$, then against $T[2..2+m)$ etc. until an occurrence is found or the end of the text is reached. The text factor $T[j..j+m)$ that is currently being compared against the pattern is called the text window.

Algorithm 2.2: Brute force
Input: text $T = T[0 \ldots n)$, pattern $P = P[0 \ldots m)$
Output: position of the first occurrence of P in T

```
(1) $i \leftarrow 0; j \leftarrow 0$
(2) while $i < m$ and $j < n$ do
(3)     if $P[i] = T[j]$ then $i \leftarrow i + 1; j \leftarrow j + 1$
(4)     else $j \leftarrow j - i + 1; i \leftarrow 0$
(5) if $i = m$ then return $j - m$ else return $n$
```

The worst case time complexity is $\mathcal{O}(mn)$. This happens, for example, when $P = \mathtt{a}^{m-1}\mathtt{b} = \mathtt{aaa..ab}$ and $T = \mathtt{a}^n = \mathtt{aaaaaa..aa}$.

(Knuth–)Morris–Pratt

The Brute force algorithm forgets everything when it shifts the window.

The Morris–Pratt (MP) algorithm remembers matches. It never goes back to a text character that already matched.

The Knuth–Morris–Pratt (KMP) algorithm remembers mismatches too.

Example 2.3:

Brute force

```
ainaisesti-ainainen
ainai~~n~~en (6 comp.)
 ~~a~~inainen (1)
  ~~a~~inainen (1)
   ai~~n~~ainen (3)
    ~~a~~inainen (1)
     ~~a~~inainen (1)
```

Morris–Pratt

```
ainaisesti-ainainen
ainai~~n~~en (6)
   ai~~n~~ainen (1)
     ~~a~~inainen (1)
```

Knuth–Morris–Pratt

```
ainaisesti-ainainen
ainai~~n~~en (6)
     ~~a~~inainen (1)
```

MP and KMP algorithms never go backwards in the text. When they encounter a mismatch, they find another pattern position to compare against the same text position. If the mismatch occurs at pattern position i, then $fail[i]$ is the next pattern position to compare.

The only difference between MP and KMP is how they compute the failure function $fail$.

Algorithm 2.4: Knuth–Morris–Pratt / Morris–Pratt
Input: text $T = T[0 \ldots n)$, pattern $P = P[0 \ldots m)$
Output: position of the first occurrence of P in T

```
(1) compute $fail[0..m]$
(2) $i \leftarrow 0; j \leftarrow 0$
(3) while $i < m$ and $j < n$ do
(4)     if $i = -1$ or $P[i] = T[j]$ then $i \leftarrow i + 1; j \leftarrow j + 1$
(5)     else $i \leftarrow fail[i]$
(6) if $i = m$ then return $j - m$ else return $n$
```

- $fail[i] = -1$ means that there is no more pattern positions to compare against this text positions and we should move to the next text position.

- $fail[m]$ is never needed here, but if we wanted to find all occurrences, it would tell how to continue after a full match.

We will describe the MP failure function here. The KMP failure function is left for the exercises.

- When the algorithm finds a mismatch between $P[i]$ and $T[j]$, we know that $P[0..i) = T[j-i..j)$.
- Now we want to find a new $i' < i$ such that $P[0..i') = T[j-i'..j)$. Specifically, we want the largest such i'.
- This means that $P[0..i') = T[j-i'..j) = P[i-i'..i)$. In other words, $P[0..i')$ is the longest proper border of $P[0..i)$.

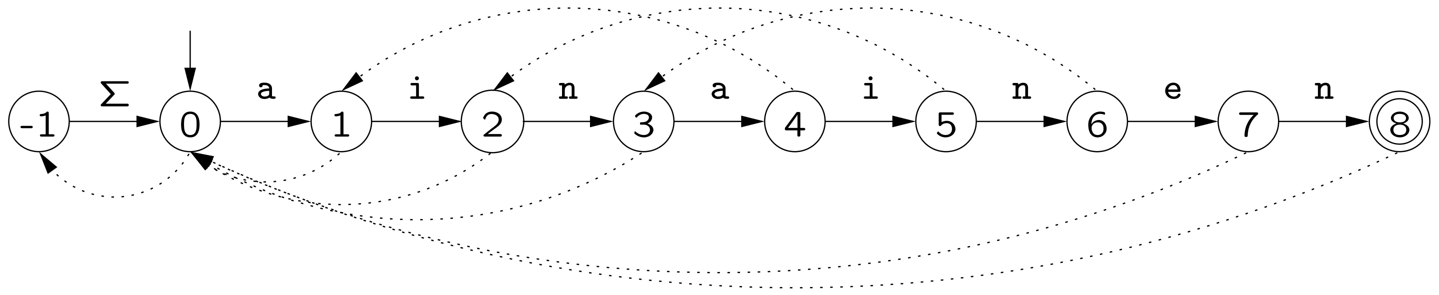
Example: `ai` is the longest proper border of `ainai`.

- Thus $fail[i]$ is the length of the longest proper border of $P[0..i)$.
- $P[0..0) = \varepsilon$ has no proper border. We set $fail[0] = -1$.

Example 2.5: Let $P =$ `ainainen`.

i	$P[0..i)$	border	$fail[i]$
0	ε	–	-1
1	`a`	ε	0
2	`ai`	ε	0
3	`ain`	ε	0
4	`aina`	`a`	1
5	`ainai`	`ai`	2
6	`ainain`	`ain`	3
7	`ainaine`	ε	0
8	`ainainen`	ε	0

The (K)MP algorithm operates like an automaton, since it never moves backwards in the text. Indeed, it can be described by an automaton that has a special failure transition, which is an ε-transition that can be taken only when there is no other transition to take.



An efficient algorithm for computing the failure function is very similar to the search algorithm itself!

- In the MP algorithm, when we find a match $P[i] = T[j]$, we know that $P[0..i] = T[j - i..j]$. More specifically, $P[0..i]$ is the longest prefix of P that matches a suffix of $T[0..j]$.
- Suppose $T = \#P[1..m)$, where $\#$ is a symbol that does not occur in P. Finding a match $P[i] = T[j]$, we know that $P[0..i]$ is the longest prefix of P that is a proper suffix of $P[0..j]$. Thus $fail[j + 1] = i + 1$.

Algorithm 2.6: Morris–Pratt failure function computation
Input: pattern $P = P[0 \dots m)$
Output: array $fail[0..m]$ for P

```
(1) $i \leftarrow -1; j \leftarrow 0$; $fail[j] \leftarrow i$
(2) while $j < m$ do
(3)       if $i = -1$ or $P[i] = P[j]$ then $i \leftarrow i + 1; j \leftarrow j + 1$; $fail[j] \leftarrow i$
(4)       else $i \leftarrow fail[i]$
(5) return $fail$
```

- When the algorithm reads $fail[i]$ on line 4, $fail[i]$ has already been computed.

Theorem 2.7: Algorithms MP and KMP preprocess a pattern in time $\mathcal{O}(m)$ and then search the text in time $\mathcal{O}(n)$ for ordered alphabet.

Proof. We show that the text search requires $\mathcal{O}(n)$ time. Exactly the same argument shows that pattern preprocessing needs $\mathcal{O}(m)$ time.

It is sufficient to count the number of comparisons that the algorithms make. After each comparison $P[i]$ vs. $T[j]$, one of the two conditional branches is executed:

then Here j is incremented. Since j never decreases, this branch can be taken at most $n + 1$ times.

else Here i decreases since $fail[i] < i$. Since i only increases in the then-branch, this branch cannot be taken more often than the then-branch.

□

Shift-And (Shift-Or)

When the MP algorithm is at position j in the text T, it computes the longest prefix of the pattern $P[0..m)$ that is a suffix of $T[0..j]$. The Shift-And algorithm computes all prefixes of P that are suffixes of $T[0..j]$.

- The information is stored in a bitvector D of length m, where $D.i = 1$ if $P[0..i] = T[j-i..j]$ and $D.i = 0$ otherwise. ($D.0$ is the least significant bit.)

- When $D.(m-1) = 1$, we have found an occurrence.

The bitvector D is updated at each text position j:

- There are precomputed bitvectors $B[c]$, for all $c \in \Sigma$, where $B[c].i = 1$ if $P[i] = c$ and $B[c].i = 0$ otherwise.

- D is updated in two steps:

  1. $D \leftarrow (D << 1) \mid 1$ (the bitwise shift and the bitwise or). Now D tells, which prefixes would match if $T[j]$ would match every character.

  2. $D \leftarrow D \,\&\, B[T[j]]$ (the bitwise and). Remove the prefixes where $T[j]$ does not match.

Let w be the wordsize of the computer, typically 64. Assume first that $m \leq w$. Then each bitvector can be stored in a single integer and the bitwise operations can be executed in constant time.

Algorithm 2.8: Shift-And
Input: text $T = T[0 \ldots n)$, pattern $P = P[0 \ldots m)$
Output: position of the first occurrence of P in T

```
Preprocess:
  (1) for $c \in \Sigma$ do $B[c] \leftarrow 0$
  (2) for $i \leftarrow 0$ to $m-1$ do $B[P[i]] \leftarrow B[P[i]] + 2^i$        // $B[P[i]].i \leftarrow 1$
Search:
  (3) $D \leftarrow 0$
  (4) for $j \leftarrow 0$ to $n-1$ do
  (5)       $D \leftarrow ((D << 1) \mid 1) \,\&\, B[T[j]]$
  (6)       if $D \,\&\, 2^{m-1} \neq 0$ then return $j - m + 1$        // $D.(m-1) = 1$
  (7) return $n$
```

Shift-Or is a minor optimization of Shift-And. It is the same algorithm except the roles of 0's and 1's in the bitvectors have been swapped. Then line 5 becomes $D \leftarrow (D << 1) \mid B[T[j]]$. Note that the "$\mid$ 1" was removed, because the shift already brings the correct bit to the least significant bit position.

Example 2.9: $P =$ `assi`, $T =$ `apassi`, bitvectors are columns.

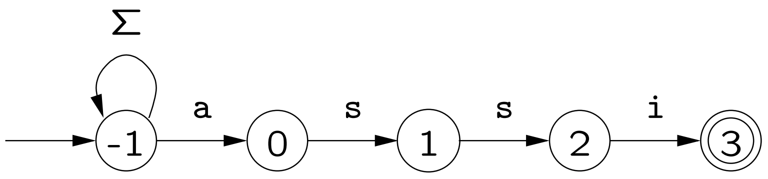
$B[c]$, $c \in \{$`a`,`i`,`p`,`s`$\}$

	a	i	p	s
a	1	0	0	0
s	0	0	0	1
s	0	0	0	1
i	0	1	0	0

D at each step

		a	p	a	s	s	i
a	0	1	0	1	0	0	0
s	0	0	0	0	1	0	0
s	0	0	0	0	0	1	0
i	0	0	0	0	0	0	1

The Shift-And algorithm can also be seen as a bitparallel simulation of the nondeterministic automaton that accepts a string ending with P.



After processing $T[j]$, $D.i = 1$ if and only if there is a path from the initial state (state -1) to state i with the string $T[0..j]$.

On an integer alphabet when $m \leq w$:

- Preprocessing time is $\mathcal{O}(\sigma + m)$.
- Search time is $\mathcal{O}(n)$.

If $m > w$, we can store the bitvectors in $\lceil m/w \rceil$ machine words and perform each bitvector operation in $\mathcal{O}(\lceil m/w \rceil)$ time.

- Preprocessing time is $\mathcal{O}(\sigma \lceil m/w \rceil + m)$.
- Search time is $\mathcal{O}(n \lceil m/w \rceil)$.

If no pattern prefix longer than w matches a current text suffix, then only the least significant machine word contains 1's. There is no need to update the other words; they will stay 0.

- Then the search time is $\mathcal{O}(n)$ on average.

Algorithms like Shift-And that take advantage of the implicit parallelism in bitvector operations are called bitparallel.

Karp–Rabin

The Karp–Rabin hash function (Definition 1.43) was originally developed for solving the exact string matching problem. The idea is to compute the hash values or fingerprints $H(P)$ and $H(T[j..j+m))$ for all $j \in [0..n-m]$.

- If $H(P) \neq H(T[j..j+m))$, then we must have $P \neq T[j..j+m)$.
- If $H(P) = H(T[j..j+m))$, the algorithm compares P and $T[j..j+m)$ in brute force manner. If $P \neq T[j..j+m)$, this is a false positive.

The text factor fingerprints are computed in a sliding window fashion. The fingerprint for $T[j+1..j+1+m) = \alpha T[j+m]$ is computed from the fingerprint for $T[j..j+m) = T[j]\alpha$ in constant time using Lemma 1.44:

$$\begin{aligned} H(T[j+1..j+1+m)) &= (H(T[j]\alpha) - H(T[j]) \cdot r^{m-1}) \cdot r + H(T[j+m])) \bmod q \\ &= (H(T[j..j+m)) - T[j] \cdot r^{m-1}) \cdot r + T[j+m]) \bmod q \ . \end{aligned}$$

A hash function that supports this kind of sliding window computation is known as a rolling hash function.

Algorithm 2.10: Karp-Rabin

Input: text $T = T[0 \ldots n)$, pattern $P = P[0 \ldots m)$
Output: position of the first occurrence of P in T

```
(1) Choose $q$ and $r$; $s \leftarrow r^{m-1} \bmod q$
(2) $hp \leftarrow 0; ht \leftarrow 0$
(3) for $i \leftarrow 0$ to $m-1$ do $hp \leftarrow (hp \cdot r + P[i]) \bmod q$     // $hp = H(P)$
(4) for $j \leftarrow 0$ to $m-1$ do $ht \leftarrow (ht \cdot r + T[j]) \bmod q$
(5) for $j \leftarrow 0$ to $n-m-1$ do
(6)     if $hp = ht$ then if $P = T[j \ldots j+m)$ then return $j$
(7)     $ht \leftarrow ((ht - T[j] \cdot s) \cdot r + T[j+m]) \bmod q$
(8) if $hp = ht$ then if $P = T[j \ldots j+m)$ then return $j$
(9) return $n$
```

On an integer alphabet:

- The worst case time complexity is $\mathcal{O}(mn)$.
- The average case time complexity is $\mathcal{O}(m+n)$.

Karp–Rabin is not competitive in practice for a single pattern, but can be for multiple patterns (exercise).

Horspool

The algorithms we have seen so far access every character of the text. If we start the comparison between the pattern and the current text position from the end, we can often skip some text characters completely.

There are many algorithms that start from the end. The simplest are the Horspool-type algorithms.

The Horspool algorithm checks first the last character of the text window, i.e., the character aligned with the last pattern character. If that doesn't match, it moves (shifts) the pattern forward until there is a match.

Example 2.11: Horspool

```
ainaisesti-ainainen
ainainen
        ainainen
           ainainen
```

More precisely, suppose we are currently comparing P against $T[j..j+m)$. Start by comparing $P[m-1]$ to $T[k]$, where $k = j+m-1$.

- If $P[m-1] \neq T[k]$, shift the pattern until the pattern character aligned with $T[k]$ matches, or until the full pattern is past $T[k]$.
- If $P[m-1] = T[k]$, compare the rest in a brute force manner. Then shift to the next position, where $T[k]$ matches.

The length of the shift is determined by the shift table that is *precomputed* for the pattern. $shift[c]$ is defined for all $c \in \Sigma$:

- If c does not occur in P, $shift[c] = m$.
- Otherwise, $shift[c] = m-1-i$, where $P[i] = c$ is the last occurrence of c in $P[0..m-2]$.

Example 2.12: $P =$ `ainainen`.

c	last occ.	$shift$
`a`	`ainainen`	4
`e`	`ainainen`	1
`i`	`ainainen`	3
`n`	`ainainen`	2
$\Sigma \setminus \{$`a,e,i,n`$\}$	—	8

Algorithm 2.13: Horspool
Input: text $T = T[0 \ldots n)$, pattern $P = P[0 \ldots m)$
Output: position of the first occurrence of P in T

```
Preprocess:
   (1)  for $c \in \Sigma$ do $shift[c] \leftarrow m$
   (2)  for $i \leftarrow 0$ to $m - 2$ do $shift[P[i]] \leftarrow m - 1 - i$
Search:
   (3)  $j \leftarrow 0$
   (4)  while $j + m \leq n$ do
   (5)      if $P[m-1] = T[j+m-1]$ then
   (6)          $i \leftarrow m - 2$
   (7)          while $i \geq 0$ and $P[i] = T[j+i]$ do $i \leftarrow i - 1$
   (8)          if $i = -1$ then return $j$
   (9)      $j \leftarrow j + shift[T[j+m-1]]$
  (10)  return $n$
```

On an integer alphabet:

- Preprocessing time is $\mathcal{O}(\sigma + m)$.
- In the worst case, the search time is $\mathcal{O}(mn)$.
  For example, $P = \mathtt{ba}^{m-1}$ and $T = \mathtt{a}^n$.
- In the best case, the search time is $\mathcal{O}(n/m)$.
  For example, $P = \mathtt{b}^m$ and $T = \mathtt{a}^n$.
- In the average case, the search time is $\mathcal{O}(n/\min(m, \sigma))$.
  This assumes that each pattern and text character is picked independently by uniform distribution.

In practice, a tuned implementation of Horspool is very fast when the alphabet is not too small.

BNDM

Starting the matching from the end enables long shifts.

- The Horspool algorithm bases the shift on a single character.
- The Boyer–Moore algorithm uses the matching suffix and the mismatching character.
- Factor based algorithms continue matching until no pattern factor matches. This may require more comparisons but it enables longer shifts.

Example 2.14: Horspool shift

```
varmasti-aikaisen-ainainen
ainaisen-ainainen
  ainaisen-ainainen
```

Boyer–Moore shift

```
varmasti-aikaisen-ainainen
ainaisen-ainainen
         ainaisen-ainainen
```

Factor shift

```
varmasti-aikaisen-ainainen
ainaisen-ainainen
                 ainaisen-ainainen
```

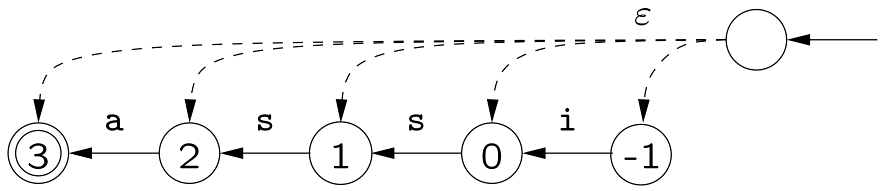
Factor based algorithms use an automaton that accepts suffixes of the reverse pattern P^R (or equivalently reverse prefixes of the pattern P).

- BDM (Backward DAWG Matching) uses a deterministic automaton that accepts exactly the suffixes of P^R.

  DAWG (Directed Acyclic Word Graph) is also known as suffix automaton.

- BNDM (Backward Nondeterministic DAWG Matching) simulates a nondeterministic automaton.

Example 2.15: $P =$ `assi`.



- BOM (Backward Oracle Matching) uses a much simpler deterministic automaton that accepts all suffixes of P^R but may also accept some other strings. This can cause shorter shifts but not incorrect behaviour.

Suppose we are currently comparing P against $T[j..j+m)$. We use the automaton to scan the text backwards from $T[j+m-1]$. When the automaton has scanned $T[j+i..j+m)$:

- If the automaton is in an accept state, then $T[j+i..j+m)$ is a prefix of P.

  ⇒ If $i = 0$, we found an occurrence.

  ⇒ Otherwise, mark the prefix match by setting $shift = i$. This is the length of the shift that would achieve a matching alignment.

- If the automaton can still reach an accept state, then $T[j+i..j+m)$ is a factor of P.

  ⇒ Continue scanning.

- When the automaton can no more reach an accept state:

  ⇒ Stop scanning and shift: $j \leftarrow j + shift$.

BNDM does a bitparallel simulation of the nondeterministic automaton, which is quite similar to Shift-And.

The state of the automaton is stored in a bitvector D. When the automaton has scanned $T[j+i..j+m)$:

- $D.k = 1$ if and only if there is a path from the initial state to state k with the string $(T[j+i..j+m))^R$, and thus $T[j+i..j+m) = P[m-k-1..2m-k-i-1)$.
- If $D.(m-1) = 1$, then $T[j+i..j+m)$ is a prefix of the pattern.
- If $D = 0$, then the automaton can no more reach an accept state.

Updating D uses precomputed bitvectors $B[c]$, for all $c \in \Sigma$:

- $B[c].i = 1$ if and only if $P[m-1-i] = P^R[i] = c$.

The update when reading $T[j+i]$ is familiar: $D = (D << 1)\ \&\ B[T[j+i]]$

- Note that there is no "$|\ 1$". This is because $D.(-1) = 0$ always after reading at least one character, so the shift brings the right bit to $D.0$.
- Before reading anything $D.(-1) = 1$. This exception is handled by starting the computation with the first shift already performed. Because of this, the shift is done at the end of the loop.

Algorithm 2.16: BNDM
Input: text $T = T[0 \ldots n)$, pattern $P = P[0 \ldots m)$
Output: position of the first occurrence of P in T

```
Preprocess:
  (1)  for $c \in \Sigma$ do $B[c] \leftarrow 0$
  (2)  for $i \leftarrow 0$ to $m-1$ do $B[P[m-1-i]] \leftarrow B[P[m-1-i]] + 2^i$
Search:
  (3)  $j \leftarrow 0$
  (4)  while $j + m \leq n$ do
  (5)      $i \leftarrow m$; $shift \leftarrow m$
  (6)      $D \leftarrow 2^m - 1$                    // $D \leftarrow 1^m$
  (7)      while $D \neq 0$ do
                // Now $T[j+i..j+m)$ is a pattern factor
  (8)          $i \leftarrow i - 1$
  (9)          $D \leftarrow D \mathbin{\&} B[T[j+i]]$
 (10)          if $D \mathbin{\&} 2^{m-1} \neq 0$ then
                    // Now $T[j+i..j+m)$ is a pattern prefix
 (11)              if $i = 0$ then return $j$
 (12)              else $shift \leftarrow i$
 (13)          $D \leftarrow D << 1$
 (14)      $j \leftarrow j + shift$
 (15)  return $n$
```

Example 2.17: $P =$ `assi`, $T =$ `apassi`.

$B[c]$, $c \in \{$`a`,`i`,`p`,`s`$\}$

$$\begin{array}{c|cccc} & \mathtt{a} & \mathtt{i} & \mathtt{p} & \mathtt{s} \\ \hline \mathtt{i} & 0 & 1 & 0 & 0 \\ \mathtt{s} & 0 & 0 & 0 & 1 \\ \mathtt{s} & 0 & 0 & 0 & 1 \\ \mathtt{a} & 1 & 0 & 0 & 0 \end{array}$$

D when scanning `apas` backwards

$$\begin{array}{c|cccc} & & \mathtt{s} & \mathtt{a} & \mathtt{p} & \mathtt{a} \\ \hline \mathtt{i} & 1 & 0 & 0 & 0 \\ \mathtt{s} & 1 & 1 & 0 & 0 \\ \mathtt{s} & 1 & 1 & 0 & 0 \\ \mathtt{a} & 1 & 0 & \underline{1} & 0 \end{array} \quad \Rightarrow \quad shift = 2$$

D when scanning `apassi` backwards

$$\begin{array}{c|ccccc} & & \mathtt{i} & \mathtt{s} & \mathtt{s} & \mathtt{a} & \mathtt{p} & \mathtt{a} \\ \hline \mathtt{i} & 1 & 1 & 0 & 0 & 0 \\ \mathtt{s} & 1 & 0 & 1 & 0 & 0 \\ \mathtt{s} & 1 & 0 & 0 & 1 & 0 \\ \mathtt{a} & 1 & 0 & 0 & 0 & \underline{1} \end{array} \quad \Rightarrow \quad \text{occurrence}$$

On an integer alphabet when $m \leq w$:

- Preprocessing time is $\mathcal{O}(\sigma + m)$.
- In the worst case, the search time is $\mathcal{O}(mn)$.
  For example, $P = \mathtt{a}^{m-1}\mathtt{b}$ and $T = \mathtt{a}^n$.
- In the best case, the search time is $\mathcal{O}(n/m)$.
  For example, $P = \mathtt{b}^m$ and $T = \mathtt{a}^n$.
- In the average case, the search time is $\mathcal{O}(n(\log_\sigma m)/m)$.
  This is optimal! It has been proven that any algorithm needs to inspect $\Omega(n(\log_\sigma m)/m)$ text characters on average.

When $m > w$, there are several options:

- Use multi-word bitvectors.
- Search for a pattern prefix of length w and check the rest when the prefix is found.
- Use BDM or BOM.

- The search time of BDM and BOM is $\mathcal{O}(n(\log_\sigma m)/m)$, which is optimal on average. (BNDM is optimal only when $m \leq w$.)

- MP and KMP are optimal in the worst case but not in the average case.

- There are also algorithms that are optimal in both cases. They are based on similar techniques, but we will not describe them here.

Crochemore

The Crochemore algorithm resembles the Morris–Pratt algorithm at a high level:

- When the pattern P is aligned against a text factor $T[j..j+m)$, they compute the longest common prefix $\ell = lcp(P, T[j..j+m))$ and report an occurrence if $\ell = m$. Otherwise, they shift the pattern forward.
- MP shifts the pattern forward by $\ell - \mathit{fail}[\ell]$ positions. In the next lcp computation, MP skips the first $\mathit{fail}[\ell]$ characters (cf. lcp-comparison).
- Crochemore either does the same shift and skip as MP, or a shorter shift than MP and starts the lcp comparison from scratch. Note that the latter case is inoptimal but always safe: no occurrence is missed.

Despite sometimes shorter shifts and less efficient lcp computation, Crochemore runs in linear time. More remarkably, it does so without any preprocessing and using only constant extra space in addition to P and T.

We will only outline the main ideas of the algorithm without detailed proofs. Even then we will need some concepts from combinatorics on words, a branch of mathematics that studies combinatorial properties of strings.

Definition 2.18: Let $S[0..m)$ be a string. An integer $p \in [1..m]$ is a period of S, if $S[i] = S[i+p]$ for all $i \in [0..m-p)$. The smallest period of S is denoted $per(S)$. S is k-periodic if $m \geq k \cdot per(S)$.

Example 2.19: The periods of $S_1 =$ `aabaaabaa` are 4,7,8 and 9. The periods of $S_2 =$ `abcabcabcabca` are 3, 6, 9, 12 and 13. S_2 is 3-periodic but S_1 is not.

There is a strong connection between periods and borders.

Lemma 2.20: p is a period of $S[0..m)$ if and only if S has a proper border of length $m - p$.

Proof. Both conditions hold if and only if $S[0..m-p) = S[p..m)$. □

Corollary 2.21: The length of the longest proper border of S is $m - per(S)$.

Recall that *fail*$[\ell]$ in MP is the length of the longest proper border of $P[0..\ell)$. Thus the pattern shift by MP is $\ell -$ *fail*$[\ell] = per(P[0..\ell))$ and the lcp skip is *fail*$[\ell] = \ell - per(P[0..\ell))$. Thus knowing $per(P[0..\ell))$ is sufficient to emulate MP shift and skip.

The Crochemore algorithm has two cases:

- If $P[0..\ell)$ is 3-periodic, then compute $per(P[0..\ell))$ and do the MP shift and skip.
- If $P[0..\ell)$ is not 3-periodic, then shift by $\lfloor \ell/3 \rfloor + 1 \leq per(P[0..\ell))$ and start the lcp comparison from scratch.

To find out if $P[0..\ell)$ is 3-periodic and to compute $per(P[0..\ell))$ if it is, Crochemore uses another combinatorial concept.

Definition 2.22: Let $MS(S)$ denote the lexicographically maximal suffix of a string S. If $S = MS(S)$, S is called self-maximal.

Period computation is easier for maximal suffixes and self-maximal strings than for arbitrary strings.

Lemma 2.23: Let $S[0..m)$ be a self-maximal string and let $p = per(S)$. For any $c \in \Sigma$,

$$\begin{array}{ll} MS(Sc) = Sc \text{ and } per(Sc) = p & \text{if } c = S[m-p] \\ MS(Sc) = Sc \text{ and } per(Sc) = m+1 & \text{if } c < S[m-p] \\ MS(Sc) \neq Sc & \text{if } c > S[m-p] \end{array}$$

Furthermore, let $r = m \bmod p$ and $R = S[m-r..m)$. Then R is self-maximal and

$$MS(Sc) = MS(Rc) \qquad \text{if } c > S[m-p]$$

Crochemore's algorithm computes the maximal suffix and its period for $P[0..\ell)$ incrementally using Lemma 2.23. The following algorithm updates the maximal suffix information when the match is extended by one character.

Algorithm 2.24: Update-MS(P, ℓ, s, p)
Input: a string P and integers ℓ, s, p such that
$MS(P[0..\ell)) = P[s..\ell)$ and $p = per(P[s..\ell))$.
Output: a triple $(\ell + 1, s', p')$ such that
$MS(P[0..\ell + 1)) = P[s'..\ell + 1)$ and $p' = per(P[s'..\ell + 1))$.

```
 (1)  if $\ell = 0$ then return $(1, 0, 1)$
 (2)  $i \leftarrow \ell$
 (3)  while $i < \ell + 1$ do
          // $P[s..i)$ is self-maximal and $p = per(P[s..i))$
 (4)      if $P[i] > P[i - p]$ then
 (5)          $i \leftarrow i - ((i - s) \bmod p)$
 (6)          $s \leftarrow i$
 (7)          $p \leftarrow 1$
 (8)      else if $P[i] < P[i - p]$ then
 (9)          $p \leftarrow i - s + 1$
(10)      $i \leftarrow i + 1$
(11)  return $(\ell + 1, s, p)$
```

As the final piece of the Crochemore algorithm, the following result shows how to use the maximal suffix information to obtain information about the periodicity of the full string.

Lemma 2.25: Let $S[0..m)$ be a string and let $S[s..m) = MS(S)$ and $p = per(MS(S))$.

- S is 3-periodic if and only if $p \leq m/3$ and $S[0..s) = S[p..p+s)$.
- If S is 3-periodic, then $per(S) = p$.

The algorithm is given on the next slide.

Algorithm 2.26: Crochemore

Input: strings $T[0..n)$ (text) and $P[0..m)$ (pattern).
Output: position of the first occurrence of P in T

```
 (1)  $j \leftarrow \ell \leftarrow p \leftarrow s \leftarrow 0$
 (2)  while $j + m \leq n$ do
 (3)      while $j + \ell < n$ and $\ell < m$ and $T[j + \ell] = P[\ell]$ do
 (4)          $(\ell, s, p) \leftarrow$ Update-MS$(P, \ell, s, p)$
          // $\ell = lcp(P, T[j..j + m))$
 (5)      if $\ell = m$ then return $j$
          // $MS(P[0..\ell)) = P[s..\ell)$ and $p = per(P[s..\ell))$
 (6)      if $p \leq \ell/3$ and $P[0..s) = P[p..p + s)$ then
              // $per(P[0..\ell)) = p$
 (7)          $j \leftarrow j + p$
 (8)          $\ell \leftarrow \ell - p$
 (9)      else    // $per(P[0..\ell)) > \ell/3$
(10)          $j \leftarrow j + \lfloor \ell/3 \rfloor + 1$
(11)          $(\ell, s, p) \leftarrow (0, 0, 0)$
(12)  return $n$
```

For ordered alphabet:

- The time complexity is $\mathcal{O}(n)$.
- The algorithm uses only a constant number of integer variables in addition to the strings P and T.

Crochemore is not competitive in practice. However, there are situations, where the pattern can be very long and the space complexity is more important than speed.

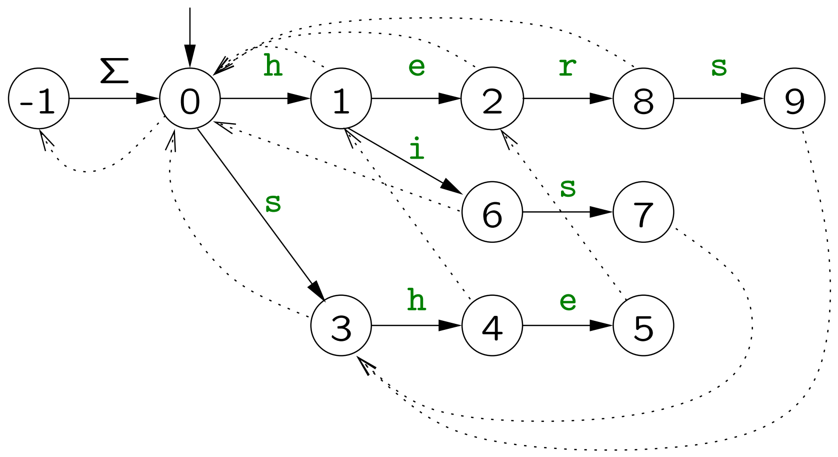
There are also other linear time, constant extra space algorithms. All of them are based on string periodicity in some way.

Aho–Corasick

Given a text T and a set $\mathcal{P} = \{P_1.P_2, \ldots, P_k\}$ of patterns, the multiple exact string matching problem asks for the occurrences of all the patterns in the text. The Aho–Corasick algorithm is an extension of the Morris–Pratt algorithm for multiple exact string matching.

Aho–Corasick uses the trie $trie(\mathcal{P})$ as an automaton and augments it with a failure function similar to the Morris-Pratt failure function.

Example 2.27: Aho–Corasick automaton for $\mathcal{P} = \{\texttt{he}, \texttt{she}, \texttt{his}, \texttt{hers}\}$.



Let S_v denote the string represented by a node v in the trie. The components of the AC automaton are:

- $root$ is the root and *child*() the child function of the trie.
- *fail*$(v) = u$ such that S_u is the longest proper suffix of S_v represented by any trie node u.
- *patterns*(v) is the set of pattern indices i such that P_i is a suffix of S_v.

Example 2.28: For the automaton in Example 2.27, *patterns*$(2) = \{1\}$ ($\{\texttt{he}\}$), *patterns*$(5) = \{1, 2\}$ ($\{\texttt{he}, \texttt{she}\}$), *patterns*$(7) = \{3\}$ ($\{\texttt{his}\}$), *patterns*$(9) = \{4\}$ ($\{\texttt{hers}\}$), and *patterns*$(v) = \emptyset$ for all other nodes v.

At each stage of the matching, the algorithm computes the node v such that S_v is the longest suffix of $T[0..j]$ represented by any node.

Algorithm 2.29: Aho–Corasick
Input: text T, pattern set $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$.
Output: all pairs (i, j) such that P_i occurs in T ending at j.

```
(1) $(root, \mathit{child}(), \mathit{fail}(), \mathit{patterns}()) \leftarrow$ Construct-AC-Automaton($\mathcal{P}$)
(2) $v \leftarrow root$
(3) for $j \leftarrow 0$ to $n-1$ do
(4)         while $\mathit{child}(v, T[j]) = \bot$ do $v \leftarrow \mathit{fail}(v)$
(5)         $v \leftarrow \mathit{child}(v, T[j])$
(6)         for $i \in \mathit{patterns}(v)$ do output $(i, j)$
```

The construction of the automaton is done in two phases: the trie construction and the failure links computation.

Algorithm 2.30: Construct-AC-Automaton
Input: pattern set $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$.
Output: AC automaton: $root$, *child*(), *fail*() and *patterns*().

```
(1) $(root, \mathit{child}(), \mathit{patterns}()) \leftarrow$ Construct-AC-Trie($\mathcal{P}$)
(2) $(\mathit{fail}(), \mathit{patterns}()) \leftarrow$ Compute-AC-Fail($root, \mathit{child}(), \mathit{patterns}()$)
(3) return $(root, \mathit{child}(), \mathit{fail}(), \mathit{patterns}())$
```

Algorithm 2.31: Construct-AC-Trie
Input: pattern set $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$.
Output: AC trie: $root$, *child*() and *patterns*().

```
 (1)  Create new node $root$
 (2)  for $i \leftarrow 1$ to $k$ do
 (3)      $v \leftarrow root$; $j \leftarrow 0$
 (4)      while child$(v, P_i[j]) \neq \perp$ do
 (5)          $v \leftarrow$ child$(v, P_i[j])$; $j \leftarrow j + 1$
 (6)      while $j < |P_i|$ do
 (7)          Create new node $u$
 (8)          child$(v, P_i[j]) \leftarrow u$
 (9)          $v \leftarrow u$; $j \leftarrow j + 1$
(10)      patterns$(v) \leftarrow \{i\}$
(11)  return $(root,$ child$(),$ patterns$())$
```

Lines (3)–(10) perform the standard trie insertion (Algorithm 1.2).

- Line (10) marks v as a representative of P_i.
- The creation of a new node v initializes *patterns*(v) to $\emptyset$ (in addition to initializing *child*(v, c) to $\perp$ for all $c \in \Sigma$).

Algorithm 2.32: Compute-AC-Fail
Input: AC trie: $root$, *child*() and *patterns*()
Output: AC failure function *fail*() and updated *patterns*()

```
(1)  Create new node $fallback$
(2)  for $c \in \Sigma$ do $child(fallback, c) \leftarrow root$
(3)  $fail(root) \leftarrow fallback$
(4)  $queue \leftarrow \{root\}$
(5)  while $queue \neq \emptyset$ do
(6)      $u \leftarrow$ popfront$(queue)$
(7)      for $c \in \Sigma$ such that $child(u, c) \neq \perp$ do
(8)          $v \leftarrow child(u, c)$
(9)          $w \leftarrow fail(u)$
(10)         while $child(w, c) = \perp$ do $w \leftarrow fail(w)$
(11)         $fail(v) \leftarrow child(w, c)$
(12)         $patterns(v) \leftarrow patterns(v) \cup patterns(fail(v))$
(13)         pushback$(queue, v)$
(14) return $(fail(), patterns())$
```

The algorithm does a breath first traversal of the trie. This ensures that correct values of *fail*() and *patterns*() are already computed when needed.

fail(v) is correctly computed on lines (8)–(11):

- Let *fail*$^*(v) = \{v, \textit{fail}(v), \textit{fail}(\textit{fail}(v)), \ldots, root\}$. These nodes are exactly the trie nodes that represent suffixes of S_v.
- Let $u =$ *parent*(v) and *child*$(u, c) = v$. Then $S_v = S_u c$ and a string S is a suffix of S_u iff Sc is suffix of S_v. Thus for any node w
  - If $w \in \textit{fail}^*(v) \setminus \{root\}$, then *parent*$(w) \in \textit{fail}^*(u)$.
  - If $w \in \textit{fail}^*(u)$ and *child*$(w, c) \neq \perp$, then *child*$(w, c) \in \textit{fail}^*(v)$.
- Therefore, *fail*$(v) =$ *child*(w, c), where w is the first node in *fail*$^*(u)$ other than u such that *child*$(w, c) \neq \perp$, or *fail*$(v) = root$ if no such node w exists.

patterns(v) is correctly computed on line (12):

$$\begin{aligned}\textit{patterns}(v) &= \{i \mid P_i \text{ is a suffix of } S_v\} \\ &= \{i \mid P_i = S_w \text{ and } w \in \textit{fail}^*(v)\} \\ &= \{i \mid P_i = S_v\} \cup \textit{patterns}(\textit{fail}(v))\end{aligned}$$

Assuming σ is constant:

- The search time is $\mathcal{O}(n)$.
- The space complexity is $\mathcal{O}(m)$, where $m = ||\mathcal{P}||$.
  - The implementation of *patterns*() requires care (exercise).
- The preprocessing time is $\mathcal{O}(m)$, where $m = ||\mathcal{P}||$.
  - The only non-trivial issue is the while-loop on line (10).
  - Let $root, v_1, v_2, \ldots, v_\ell$ be the nodes on the path from root to a node representing a pattern P_i. Let $w_j =$ *fail*(v_j) for all j. Let *depth*(v) be the depth of a node v (*depth*$(root) = 0$).
  - When processing v_j and computing $w_j =$ *fail*(v_j), we have *depth*$(w_j) =$ *depth*$(w_{j-1}) + 1$ before line (10) and *depth*$(w_j) \leq$ *depth*$(w_{j-1}) + 1 - t_j$ after line (10), where t_j is the number of rounds in the while-loop.
  - Thus, the total number of rounds in the while-loop when processing the nodes $v_1, v_2, \ldots, v_\ell$ is at most $\ell = |P_i|$, and thus over the whole algorithm at most $||\mathcal{P}||$.

The analysis when σ is not constant is left as an exercise.

Summary: Exact String Matching

Exact string matching is a fundamental problem in stringology. We have seen several different algorithms for solving the problem.

The properties of the algorithms vary with respect to worst case time complexity, average case time complexity, type of alphabet (ordered/integer) and even space complexity.

The algorithms use a wide range of completely different techniques:

- There exists numerous algorithms for exact string matching but most of them use variations or combinations of the techniques we have seen (study groups).
- Many of the techniques can be adapted to other problems. All of the techniques have some uses in practice.

3. Approximate String Matching

Often in applications we want to search a text for something that is similar to the pattern but not necessarily exactly the same.

To formalize this problem, we have to specify what does “similar” mean. This can be done by defining a similarity or a distance measure.

A natural and popular distance measure for strings is the edit distance, also known as the Levenshtein distance.

Edit distance

The edit distance $ed(A, B)$ of two strings A and B is the minimum number of edit operations needed to change A into B. The allowed edit operations are:

- S Substitution of a single character with another character.
- I Insertion of a single character.
- D Deletion of a single character.

Example 3.1: Let $A =$ `Lewensteinn` and $B =$ `Levenshtein`. Then $ed(A, B) = 3$.

The set of edit operations can be described

with an edit sequence:	`NNSNNNINNNND`
or with an alignment:	`Lewens-teinn`
	`Levenshtein-`

In the edit sequence, N means No edit.

There are many variations and extension of the edit distance, for example:

- Hamming distance allows only the subtitution operation.
- Damerau–Levenshtein distance adds an edit operation:

  T Transposition swaps two adjacent characters.
- With weighted edit distance, each operation has a cost or weight, which can be other than one.
- Allow insertions and deletions (indels) of factors at a cost that is lower than the sum of character indels.

We will focus on the basic Levenshtein distance.

Levenshtein distance has the following two useful properties, which are not shared by all variations (exercise):

- Levenshtein distance is a metric.
- If $ed(A, B) = k$, there exists an edit sequence and an alignment with k edit operations, but no edit sequence or alignment with less than k edit operations. An edit sequence and an alignment with $ed(A, B)$ edit operations is called optimal.

Computing Edit Distance

Given two strings $A[1..m]$ and $B[1..n]$, define the values d_{ij} with the recurrence:

$$
\begin{aligned}
d_{00} &= 0, \\
d_{i0} &= i,\ \ 1 \le i \le m, \\
d_{0j} &= j,\ \ 1 \le j \le n,\ \ \text{and} \\
d_{ij} &= \min \begin{cases} d_{i-1,j-1} + \delta(A[i], B[j]) \\ \quad d_{i-1,j} + 1 \\ \quad d_{i,j-1} + 1 \end{cases} \qquad 1 \le i \le m, 1 \le j \le n,
\end{aligned}
$$

where

$$
\delta(A[i], B[j]) = \begin{cases} 1 & \text{if } A[i] \ne B[j] \\ 0 & \text{if } A[i] = B[j] \end{cases}
$$

Theorem 3.2: $d_{ij} = ed(A[1..i], B[1..j])$ for all $0 \le i \le m$, $0 \le j \le n$. In particular, $d_{mn} = ed(A, B)$.

Example 3.3: $A = \texttt{ballad}, B = \texttt{handball}$

d		h	a	n	d	b	a	l	l
	0	1	2	3	4	5	6	7	8
b	1	1	2	3	4	4	5	6	7
a	2	2	1	2	3	4	4	5	6
l	3	3	2	2	3	4	5	4	5
l	4	4	3	3	3	4	5	5	4
a	5	5	4	4	4	4	4	5	5
d	6	6	5	5	4	5	5	5	6

$ed(A, B) = d_{mn} = d_{6,8} = 6$.

Proof of Theorem 3.2. We use induction with respect to $i + j$. For brevity, write $A_i = A[1..i]$ and $B_j = B[1..j]$.

Basis:

$$\begin{aligned} d_{00} &= 0 = ed(\epsilon, \epsilon) \\ d_{i0} &= i = ed(A_i, \epsilon) \quad (i \text{ deletions}) \\ d_{0j} &= j = ed(\epsilon, B_j) \quad (j \text{ insertions}) \end{aligned}$$

Induction step: We show that the claim holds for d_{ij}, $1 \le i \le m, 1 \le j \le n$. By induction assumption, $d_{pq} = ed(A_p, B_q)$ when $p + q < i + j$.

Let E_{ij} be an optimal edit sequence with the cost $ed(A_i, B_j)$. We have three cases depending on what the last operation symbol in E_{ij} is:

N or S: $E_{ij} = E_{i-1,j-1}$N or $E_{ij} = E_{i-1,j-1}$S and $ed(A_i, B_j) = ed(A_{i-1}, B_{j-1}) + \delta(A[i], B[j]) = d_{i-1,j-1} + \delta(A[i], B[j])$.

I: $E_{ij} = E_{i,j-1}$I and $ed(A_i, B_j) = ed(A_i, B_{j-1}) + 1 = d_{i,j-1} + 1$.

D: $E_{ij} = E_{i-1,j}$D and $ed(A_i, B_j) = ed(A_{i-1}, B_j) + 1 = d_{i-1,j} + 1$.

One of the cases above is always true, and since the edit sequence is optimal, it must be one with the minimum cost, which agrees with the definition of d_{ij}. $\square$

The recurrence gives directly a dynamic programming algorithm for computing the edit distance.

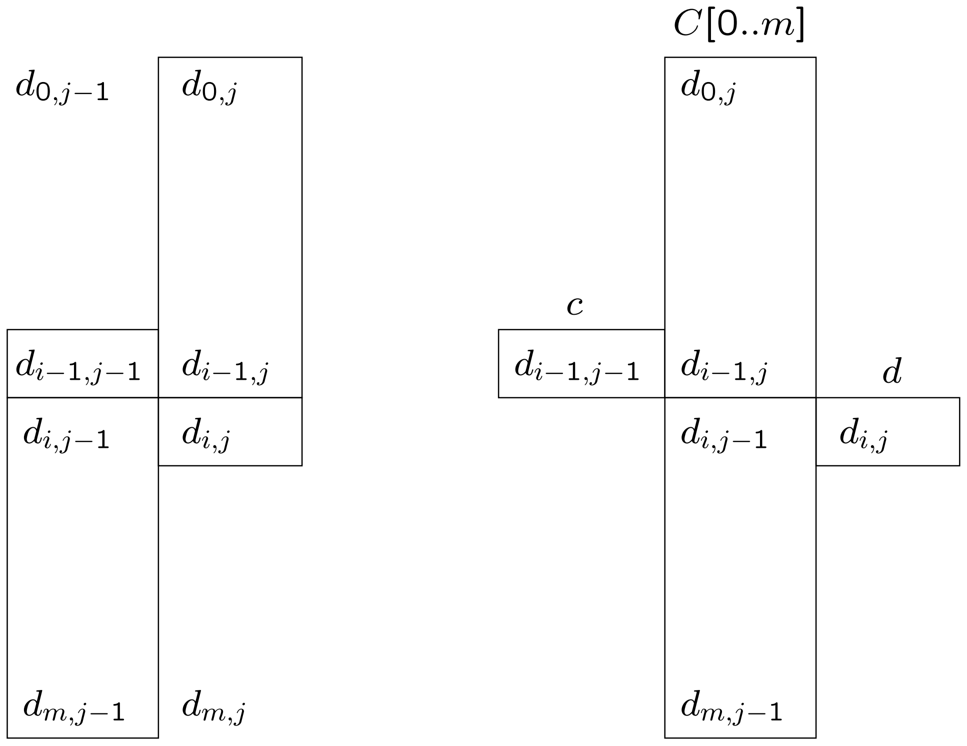
Algorithm 3.4: Edit distance
Input: strings $A[1..m]$ and $B[1..n]$
Output: $ed(A, B)$

```
(1) for $i \leftarrow 0$ to $m$ do $d_{i0} \leftarrow i$
(2) for $j \leftarrow 1$ to $n$ do $d_{0j} \leftarrow j$
(3) for $j \leftarrow 1$ to $n$ do
(4)     for $i \leftarrow 1$ to $m$ do
(5)         $d_{ij} \leftarrow \min\{d_{i-1,j-1} + \delta(A[i], B[j]), d_{i-1,j} + 1, d_{i,j-1} + 1\}$
(6) return $d_{mn}$
```

The time and space complexity is $\mathcal{O}(mn)$.

The space complexity can be reduced by noticing that each column of the matrix (d_{ij}) depends only on the previous column. We do not need to store older columns.

A more careful look reveals that, when computing d_{ij}, we only need to store the bottom part of column $j-1$ and the already computed top part of column j. We store these in an array $C[0..m]$ and variables c and d as shown below:



Algorithm 3.5: Edit distance in $\mathcal{O}(m)$ space
Input: strings $A[1..m]$ and $B[1..n]$
Output: $ed(A, B)$

```
(1) for $i \leftarrow 0$ to $m$ do $C[i] \leftarrow i$
(2) for $j \leftarrow 1$ to $n$ do
(3)         $c \leftarrow C[0]$; $C[0] \leftarrow j$
(4)         for $i \leftarrow 1$ to $m$ do
(5)                 $d \leftarrow \min\{c + \delta(A[i], B[j]), C[i-1] + 1, C[i] + 1\}$
(6)                 $c \leftarrow C[i]$
(7)                 $C[i] \leftarrow d$
(8) return $C[m]$
```

Note that because $ed(A, B) = ed(B, A)$ (exercise), we can always choose A to be the shorter string so that $m \leq n$.

It is also possible to find optimal edit sequences and alignments from the matrix d_{ij}.

An edit graph is a directed graph, where the nodes are the cells of the edit distance matrix, and the edges are as follows:

- If $A[i] = B[j]$ and $d_{ij} = d_{i-1,j-1}$, there is an edge $(i-1, j-1) \rightarrow (i, j)$ labelled with N.
- If $A[i] \neq B[j]$ and $d_{ij} = d_{i-1,j-1} + 1$, there is an edge $(i-1, j-1) \rightarrow (i, j)$ labelled with S.
- If $d_{ij} = d_{i,j-1} + 1$, there is an edge $(i, j-1) \rightarrow (i, j)$ labelled with I.
- If $d_{ij} = d_{i-1,j} + 1$, there is an edge $(i-1, j) \rightarrow (i, j)$ labelled with D.

Any path from $(0, 0)$ to (m, n) is labelled with an optimal edit sequence.

Example 3.6: $A =$ `ballad`, $B =$ `handball`

$$
\begin{array}{c|ccccccccccccccccc}
d & & \mathtt{h} & & \mathtt{a} & & \mathtt{n} & & \mathtt{d} & & \mathtt{b} & & \mathtt{a} & & \mathtt{l} & & \mathtt{l} & \\
\hline
 & 0 & \Rightarrow & 1 & \Rightarrow & 2 & \Rightarrow & 3 & \Rightarrow & 4 & \rightarrow & 5 & \rightarrow & 6 & \rightarrow & 7 & \rightarrow & 8 \\
\mathtt{b} & \downarrow & \Searrow & & \searrow & & \searrow & & \searrow & & \Searrow & & & & & & & \\
 & 1 & & 1 & \rightarrow & 2 & \rightarrow & 3 & \rightarrow & 4 & & 4 & \rightarrow & 5 & \rightarrow & 6 & \rightarrow & 7 \\
\mathtt{a} & \downarrow & \searrow & \downarrow & \Searrow & & & & & & & & \Searrow & & & & & \\
 & 2 & & 2 & & 1 & \Rightarrow & 2 & \rightarrow & 3 & \rightarrow & 4 & & 4 & \rightarrow & 5 & \rightarrow & 6 \\
\mathtt{l} & \downarrow & \searrow & \downarrow & & \downarrow & \Searrow & & \Searrow & & \searrow & & \searrow & \downarrow & \Searrow & & \searrow & \\
 & 3 & & 3 & & 2 & & 2 & \Rightarrow & 3 & \rightarrow & 4 & \rightarrow & 5 & & 4 & \rightarrow & 5 \\
\mathtt{l} & \downarrow & \searrow & \downarrow & & \downarrow & \searrow & \downarrow & \Searrow & & \Searrow & & \searrow & & \searrow & \downarrow & \Searrow & \\
 & 4 & & 4 & & 3 & & 3 & & 3 & \Rightarrow & 4 & \rightarrow & 5 & & 5 & & 4 \\
\mathtt{a} & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & \searrow & & \Searrow & & & & & \Downarrow \\
 & 5 & & 5 & & 4 & & 4 & & 4 & & 4 & & 4 & \Rightarrow & 5 & & 5 \\
\mathtt{d} & \downarrow & \searrow & \downarrow & & \downarrow & \searrow & \downarrow & \searrow & & \searrow & \downarrow & \searrow & \downarrow & \Searrow & & \Searrow & \Downarrow \\
 & 6 & & 6 & & 5 & & 5 & & 4 & \rightarrow & 5 & & 5 & & 5 & \Rightarrow & 6 \\
\hline
\end{array}
$$

There are 7 paths from $(0,0)$ to $(6,8)$ corresponding to 7 different optimal edit sequences and alignments, including the following three:

```
IIIINNNNDD    SNISSNIS    SNSSINSI
----ballad    ba-lla-d    ball-ad-
handball--    handball    handball
```

Approximate String Matching

Now we are ready to tackle the main problem of this part: approximate string matching.

Problem 3.7: Given a text $T[1..n]$, a pattern $P[1..m]$ and an integer $k \geq 0$, report all positions $j \in [1..m]$ such that $ed(P, T(j-\ell...j]) \leq k$ for some $\ell \geq 0$.

The factor $T(j-\ell...j]$ is called an approximate occurrence of P.

There can be multiple occurrences of different lengths ending at the same position j, but usually it is enough to report just the end positions.
We ask for the end position rather than the start position because that is more natural for the algorithms.

Define the values g_{ij} with the recurrence:

$$g_{0j} = 0,\ \ 0 \le j \le n,$$
$$g_{i0} = i,\ \ 1 \le i \le m,\ \text{and}$$
$$g_{ij} = \min \begin{cases} g_{i-1,j-1} + \delta(P[i], T[j]) \\ \quad g_{i-1,j} + 1 \\ \quad g_{i,j-1} + 1 \end{cases} \qquad 1 \le i \le m, 1 \le j \le n.$$

Theorem 3.8: For all $0 \le i \le m$, $0 \le j \le n$:

$$g_{ij} = \min\{ed(P[1..i], T(j-\ell...j]) \mid 0 \le \ell \le j\}\ .$$

In particular, j is an ending position of an approximate occurrence if and only if $g_{mj} \le k$.

Proof. We use induction with respect to $i + j$.

Basis:

$$\begin{aligned} g_{00} &= 0 = ed(\epsilon, \epsilon) \\ g_{0j} &= 0 = ed(\epsilon, \epsilon) = ed(\epsilon, T(j - 0..j]) \qquad \text{(min at } \ell = 0) \\ g_{i0} &= i = ed(P[1..i], \epsilon) = ed(P[1..i], T(0 - 0..0]) \quad (0 \le \ell \le j = 0) \end{aligned}$$

Induction step: Essentially the same as in the proof of Theorem 3.2.

Example 3.9: $P = \mathtt{match}$, $T = \mathtt{remachine}$, $k = 1$

$$
\begin{array}{c|ccccccccccccccccccc|}
g & & \mathtt{r} & & \mathtt{e} & & \mathtt{m} & & \mathtt{a} & & \mathtt{c} & & \mathtt{h} & & \mathtt{i} & & \mathtt{n} & & \mathtt{e} & \\
\hline
 & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 0 \\
\mathtt{m} & & & & & & \Searrow & & & & & & & & & & & & & \\
 & 1 & & 1 & & 1 & & 0 & & 1 & & 1 & & 1 & & 1 & & 1 & & 1 \\
\mathtt{a} & & & & & & & & \Searrow & & & & & & & & & & & \\
 & 2 & & 2 & & 2 & & 1 & & 0 & & 1 & & 2 & & 2 & & 2 & & 2 \\
\mathtt{t} & & & & & & & & & \Downarrow & & & & & & & & & & \\
 & 3 & & 3 & & 3 & & 2 & & 1 & & 1 & & 2 & & 3 & & 3 & & 3 \\
\mathtt{c} & & & & & & & & & & \Searrow & & & & & & & & & \\
 & 4 & & 4 & & 4 & & 3 & & 2 & & 1 & & 2 & & 3 & & 4 & & 4 \\
\mathtt{h} & & & & & & & & & & & & \Searrow & & & & & & & \\
 & 5 & & 5 & & 5 & & 4 & & 3 & & 2 & & 1 & & 2 & & 3 & & 4 \\
\hline
\end{array}
$$

One occurrence ending at position 6.

Algorithm 3.10: Approximate string matching
Input: text $T[1..n]$, pattern $P[1..m]$, and integer k
Output: end positions of all approximate occurrences of P

```
(1) for $i \leftarrow 0$ to $m$ do $g_{i0} \leftarrow i$
(2) for $j \leftarrow 1$ to $n$ do $g_{0j} \leftarrow 0$
(3) for $j \leftarrow 1$ to $n$ do
(4)     for $i \leftarrow 1$ to $m$ do
(5)         $g_{ij} \leftarrow \min\{g_{i-1,j-1} + \delta(A[i], B[j]), g_{i-1,j} + 1, g_{i,j-1} + 1\}$
(6)     if $q_{mj} \leq k$ then output $j$
```

- Time and space complexity is $\mathcal{O}(mn)$ on ordered alphabet.
- The space complexity can be reduced to $\mathcal{O}(m)$ by storing only one column as in Algorithm 3.5.

Ukkonen's Cut-off Heuristic

We can speed up the algorithm using the diagonal monotonicity of the matrix (g_{ij}):

> A diagonal d, $-m \le d \le n$, consists of the cells g_{ij} with $j - i = d$. Every diagonal in (g_{ij}) is monotonically non-decreasing.

Example 3.11: Diagonals -3 and 2.

$$
\begin{array}{c|ccccccccccccccccccc|}
g & & \mathtt{r} & & \mathtt{e} & & \mathtt{m} & & \mathtt{a} & & \mathtt{c} & & \mathtt{h} & & \mathtt{i} & & \mathtt{n} & & \mathtt{e} & \\
\hline
 & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 0 & & 0 \\
\mathtt{m} & & & & & & \diagdown \\
 & 1 & & 1 & & 1 & & 0 & & 1 & & 1 & & 1 & & 1 & & 1 & & 1 \\
\mathtt{a} & & & & & & & & \diagdown \\
 & 2 & & 2 & & 2 & & 1 & & 0 & & 1 & & 2 & & 2 & & 2 & & 2 \\
\mathtt{t} & & & & & & & & & & \diagdown \\
 & 3 & & 3 & & 3 & & 2 & & 1 & & 1 & & 2 & & 3 & & 3 & & 3 \\
\mathtt{c} & & \diagdown & & & & & & & & & & \diagdown \\
 & 4 & & 4 & & 4 & & 3 & & 2 & & 1 & & 2 & & 3 & & 4 & & 4 \\
\mathtt{h} & & & & \diagdown & & & & & & & & & & \diagdown \\
 & 5 & & 5 & & 5 & & 4 & & 3 & & 2 & & 1 & & 2 & & 3 & & 4 \\
\hline
\end{array}
$$

Lemma 3.12: For every $i \in [1..m]$ and every $j \in [1..n]$, $g_{ij} = g_{i-1,j-1}$ or $g_{ij} = g_{i-1,j-1} + 1$.

Proof. By definition, $g_{ij} \leq g_{i-1,j-1} + \delta(P[i], T[j]) \leq g_{i-1,j-1} + 1$. We show that $g_{ij} \geq g_{i-1,j-1}$ by induction on $i + j$.

The induction assumption is that $g_{pq} \geq g_{p-1,q-1}$ when $p \in [1..m]$, $q \in [1..n]$ and $p + q < i + j$. At least one of the following holds:

1. $g_{ij} = g_{i-1,j-1} + \delta(P[i], T[j])$. Then $g_{ij} \geq g_{i-1,j-1}$.
2. $g_{ij} = g_{i-1,j} + 1$ and $i > 1$. Then
$$g_{ij} = g_{i-1,j} + 1 \overset{\text{ind. assump.}}{\geq} g_{i-2,j-1} + 1 \overset{\text{definition}}{\geq} g_{i-1,j-1}$$
3. $g_{ij} = g_{i,j-1} + 1$ and $j > 1$. Then
$$g_{ij} = g_{i,j-1} + 1 \overset{\text{ind. assump.}}{\geq} g_{i-1,j-2} + 1 \overset{\text{definition}}{\geq} g_{i-1,j-1}$$
4. $g_{ij} = g_{i-1,j} + 1$ and $i = 1$. Then $g_{ij} = 0 + 1 > 0 = g_{i-1,j-1}$.
5. $g_{ij} = g_{i,j-1} + 1$ and $j = 1$. Then $g_{ij} = i + 1 = (i-1) + 2 = g_{i-1,j-1} + 2$, which cannot be true. Thus this case can never happen. □

We can reduce computation using diagonal monotonicity:

- Whenever the value on a diagonal d grows larger than k, we can discard d from consideration, because we are only interested in values at most k on the row m.
- We keep track of the smallest undiscarded diagonal d. Each column is computed only up to diagonal $d+1$.

Example 3.13: $P =$ strict, $T =$ datastructure, $k = 1$

g		d	a	t	a	s	t	r	u	c	t	u	r	e
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s	1	1	1	1	1	0	1	1	1	1	1	1	1	1
t	2	2	2	1	2	1	0	1	2	2	1	2	2	2
r				2	2	2	1	0	1	2	2	2		
i							2	1	1	2	3	3		
c								2	2	1	2	3		
t										2	1	2		

The position of the smallest undiscarded diagonal on the current column is kept in a variable top.

Algorithm 3.14: Ukkonen's cut-off algorithm
Input: text $T[1..n]$, pattern $P[1..m]$, and integer k
Output: end positions of all approximate occurrences of P

```
(1) $top \leftarrow \min(k+1, m)$
(2) for $i \leftarrow 0$ to $top$ do $g_{i0} \leftarrow i$
(3) for $j \leftarrow 1$ to $n$ do $g_{0j} \leftarrow 0$
(4) for $j \leftarrow 1$ to $n$ do
(5)         for $i \leftarrow 1$ to $top$ do
(6)                 $g_{ij} \leftarrow \min\{g_{i-1,j-1} + \delta(A[i], B[j]), g_{i-1,j} + 1, g_{i,j-1} + 1\}$
(7)         while $g_{top,j} > k$ do $top \leftarrow top - 1$
(8)         if $top = m$ then output $j$
(9)         else $top \leftarrow top + 1$; $g_{top,j} \leftarrow k + 1$
```

The time complexity is proportional to the computed area in the matrix (g_{ij}).

- The worst case time complexity is still $\mathcal{O}(mn)$ on ordered alphabet.
- The average case time complexity is $\mathcal{O}(kn)$. The proof is not trivial.

There are many other algorithms based on diagonal monotonicity. Some of them achieve $\mathcal{O}(kn)$ worst case time complexity.

Myers' Bitparallel Algorithm

Another way to speed up the computation is bitparallelism.

Instead of the matrix (g_{ij}), we store differences between adjacent cells:

> Vertical delta: $\Delta v_{ij} = g_{ij} - g_{i-1,j}$
>
> Horizontal delta: $\Delta h_{ij} = g_{ij} - g_{i,j-1}$
>
> Diagonal delta: $\Delta d_{ij} = g_{ij} - g_{i-1,j-1}$

Because $g_{i0} = i$ ja $g_{0j} = 0$,

$$\begin{aligned} g_{ij} &= \Delta v_{1j} + \Delta v_{2j} + \cdots + \Delta v_{ij} \\ &= i + \Delta h_{i1} + \Delta h_{i2} + \cdots + \Delta h_{ij} \end{aligned}$$

Because of diagonal monotonicity, $\Delta d_{ij} \in \{0, 1\}$ and it can be stored in one bit. By the following result, Δh_{ij} and Δv_{ij} can be stored in two bits.

Lemma 3.15: $\Delta h_{ij}, \Delta v_{ij} \in \{-1, 0, 1\}$ for every i, j that they are defined for.

The proof is left as an exercise.

Example 3.16: '−' means −1, '=' means 0 and '+' means +1

		r		e		m		a		c		h		i		n		e	
	0	=	0	=	0	=	0	=	0	=	0	=	0	=	0	=	0	=	0
m	+	+	+	+	+	=	=	+	+	+	+	+	+	+	+	+	+	+	+
	1	=	1	=	1	−	0	+	1	=	1	=	1	=	1	=	1	=	1
a	+	+	+	+	+	=	+	=	−	=	=	+	+	+	+	+	+	+	+
	2	=	2	=	2	−	1	−	0	+	1	+	2	=	2	=	2	=	2
t	+	+	+	+	+	=	+	=	+	+	=	+	=	+	+	+	+	+	+
	3	=	3	=	3	−	2	−	1	=	1	+	2	+	3	=	3	=	3
c	+	+	+	+	+	=	+	=	+	=	=	+	=	+	=	+	+	+	+
	4	=	4	=	4	−	3	−	2	−	1	+	2	+	3	+	4	=	4
h	+	+	+	+	+	=	+	=	+	=	+	=	−	=	−	=	−	=	=
	5	=	5	=	5	−	4	−	3	−	2	−	1	+	2	+	3	+	4

In the standard computation of a cell:

- Input is $g_{i-1,j}$, $g_{i-1,j-1}$, $g_{i,j-1}$ and $\delta(P[i],T[j])$.
- Output is g_{ij}.

In the corresponding bitparallel computation:

- Input is $\Delta v^{\text{in}} = \Delta v_{i,j-1}$, $\Delta h^{\text{in}} = \Delta h_{i-1,j}$ and $Eq_{ij} = 1 - \delta(P[i],T[j])$.
- Output is $\Delta v^{\text{out}} = \Delta v_{i,j}$ and $\Delta h^{\text{out}} = \Delta h_{i,j}$.

$$\begin{tikzcd}
g_{i-1,j-1} \arrow[r, "\Delta h^{\text{in}}"] \arrow[d, "\Delta v^{\text{in}}"'] & g_{i-1,j} \arrow[d, "\Delta v^{\text{out}}"] \\
g_{i,j-1} \arrow[r, "\Delta h^{\text{out}}"'] & g_{ij}
\end{tikzcd}$$

The algorithm does not compute the Δd values but they are useful in the proofs.

The computation rule is defined by the following result.

Lemma 3.17: If $Eq = 1$ or $\Delta v^{\mathsf{in}} = -1$ or $\Delta h^{\mathsf{in}} = -1$,
then $\Delta d = 0$, $\Delta v^{\mathsf{out}} = -\Delta h^{\mathsf{in}}$ and $\Delta h^{\mathsf{out}} = -\Delta v^{\mathsf{in}}$.
Otherwise $\Delta d = 1$, $\Delta v^{\mathsf{out}} = 1 - \Delta h^{\mathsf{in}}$ and $\Delta h^{\mathsf{out}} = 1 - \Delta v^{\mathsf{in}}$.

Proof. We can write the recurrence for g_{ij} as

$$\begin{aligned} g_{ij} &= \min\{g_{i-1,j-1} + \delta(P[i], T[j]), g_{i,j-1} + 1, g_{i-1,j} + 1\} \\ &= g_{i-1,j-1} + \min\{1 - Eq, \Delta v^{\mathsf{in}} + 1, \Delta h^{\mathsf{in}} + 1\}. \end{aligned}$$

Then $\Delta d = g_{ij} - g_{i-1,j-1} = \min\{1 - Eq, \Delta v^{\mathsf{in}} + 1, \Delta h^{\mathsf{in}} + 1\}$
which is 0 if $Eq = 1$ or $\Delta v^{\mathsf{in}} = -1$ or $\Delta h^{\mathsf{in}} = -1$ and 1 otherwise.

Clearly $\Delta d = \Delta v^{\mathsf{in}} + \Delta h^{\mathsf{out}} = \Delta h^{\mathsf{in}} + \Delta v^{\mathsf{out}}$.
Thus $\Delta v^{\mathsf{out}} = \Delta d - \Delta h^{\mathsf{in}}$ and $\Delta h^{\mathsf{out}} = \Delta d - \Delta v^{\mathsf{in}}$. $\square$

To enable bitparallel operation, we need two changes:

- The Δv and Δh values are "trits" not bits. We encode each of them with two bits as follows:

$$Pv = \begin{cases} 1 & \text{if } \Delta v = +1 \\ 0 & \text{otherwise} \end{cases} \qquad Mv = \begin{cases} 1 & \text{if } \Delta v = -1 \\ 0 & \text{otherwise} \end{cases}$$

$$Ph = \begin{cases} 1 & \text{if } \Delta h = +1 \\ 0 & \text{otherwise} \end{cases} \qquad Mh = \begin{cases} 1 & \text{if } \Delta h = -1 \\ 0 & \text{otherwise} \end{cases}$$

  Then

$$\Delta v = Pv - Mv$$
$$\Delta h = Ph - Mh$$

- We replace arithmetic operations ($+$, $-$, min) with Boolean (logical) operations ($\wedge$, $\vee$, $\neg$).

Now the computation rules can be expressed as follows.

Lemma 3.18:

$$\begin{aligned} Pv^{\text{out}} &= Mh^{\text{in}} \vee \neg(Xv \vee Ph^{\text{in}}) & \qquad Mv^{\text{out}} &= Ph^{\text{in}} \wedge Xv \\ Ph^{\text{out}} &= Mv^{\text{in}} \vee \neg(Xh \vee Pv^{\text{in}}) & \qquad Mh^{\text{out}} &= Pv^{\text{in}} \wedge Xh \end{aligned}$$

where $Xv = Eq \vee Mv^{\text{in}}$ and $Xh = Eq \vee Mh^{\text{in}}$.

Proof. We show the claim for Pv and Mv only. Ph and Mh are symmetrical.

By Lemma 3.17,

$$\begin{aligned} Pv^{\text{out}} &= (\neg\Delta d \wedge Mh^{\text{in}}) \vee (\Delta d \wedge \neg Ph^{\text{in}}) \\ Mv^{\text{out}} &= (\neg\Delta d \wedge Ph^{\text{in}}) \vee (\Delta d \wedge 0) = \neg\Delta d \wedge Ph^{\text{in}} \end{aligned}$$

Because $\Delta d = \neg(Eq \vee Mv^{\text{in}} \vee Mh^{\text{in}}) = \neg(Xv \vee Mh^{\text{in}}) = \neg Xv \wedge \neg Mh^{\text{in}}$,

$$\begin{aligned} Pv^{\text{out}} &= ((Xv \vee Mh^{\text{in}}) \wedge Mh^{\text{in}}) \vee (\neg Xv \wedge \neg Mh^{\text{in}} \wedge \neg Ph^{\text{in}}) \\ &= Mh^{\text{in}} \vee \neg(Xv \vee Mh^{\text{in}} \vee Ph^{\text{in}}) = Mh^{\text{in}} \vee \neg(Xv \vee Ph^{\text{in}}) \\ Mv^{\text{out}} &= (Xv \vee Mh^{\text{in}}) \wedge Ph^{\text{in}} = (Xv \wedge Ph^{\text{in}}) \vee (Mh^{\text{in}} \wedge Ph^{\text{in}}) = Xv \wedge Ph^{\text{in}} \end{aligned}$$

All the steps above use just basic laws of Boolean algebra except the last step, where we use the fact that Mh^{in} and Ph^{in} cannot be 1 simultaneously.
$\square$

According to Lemma 3.18, the bit representation of the matrix can be computed as follows.

```
for $i \leftarrow 1$ to $m$ do
    $Pv_{i0} \leftarrow 1$; $Mv_{i0} \leftarrow 0$
for $j \leftarrow 1$ to $n$ do
    $Ph_{0j} \leftarrow 0$; $Mh_{0j} \leftarrow 0$
    for $i \leftarrow 1$ to $m$ do
        $Xh_{ij} \leftarrow Eq_{ij} \vee Mh_{i-1,j}$
        $Ph_{ij} \leftarrow Mv_{i,j-1} \vee \neg(Xh_{ij} \vee \ Pv_{i,j-1})$
        $Mh_{ij} \leftarrow Pv_{i,j-1} \wedge Xh_{ij}$
    for $i \leftarrow 1$ to $m$ do
        $Xv_{ij} \leftarrow Eq_{ij} \vee Mv_{i,j-1}$
        $Pv_{ij} \leftarrow Mh_{i-1,j} \vee \neg(Xv_{ij} \vee \ Ph_{i-1,j})$
        $Mv_{ij} \leftarrow Ph_{i-1,j} \wedge Xv_{ij}$
```

This is not yet bitparallel though.

To obtain a bitparallel algorithm, the columns Pv_{*j}, Mv_{*j}, Xv_{*j}, Ph_{*j}, Mh_{*j}, Xh_{*j} and Eq_{*j} are stored in bitvectors.

Now the second inner loop can be replaced with the code

$$\begin{aligned}&Xv_{*j} \leftarrow Eq_{*j} \vee Mv_{*,j-1}\\&Pv_{*j} \leftarrow (Mh_{*,j} << 1) \vee \neg(Xv_{*j} \vee \ (Ph_{*j} << 1))\\&Mv_{*j} \leftarrow (Ph_{*j} << 1) \wedge Xv_{*j}\end{aligned}$$

A similar attempt with the for first inner loop leads to a problem:

$$\begin{aligned}&Xh_{*j} \leftarrow Eq_{*j} \vee (Mh_{*j} << 1)\\&Ph_{*j} \leftarrow Mv_{*,j-1} \vee \neg(Xh_{*j} \vee \ Pv_{*,j-1})\\&Mh_{*j} \leftarrow Pv_{*,j-1} \wedge Xh_{*j}\end{aligned}$$

Now the vector Mh_{*j} is used in computing Xh_{*j} before Mh_{*j} itself is computed! Changing the order does not help, because Xh_{*j} is needed to compute Mh_{*j}.

To get out of this dependency loop, we compute Xh_{*j} without Mh_{*j} using only Eq_{*j} and $Pv_{*,j-1}$ which are already available when we compute Xh_{*j}.

Lemma 3.19: $Xh_{ij} = \exists \ell \in [1,i] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-1] : Pv_{x,j-1})$.

Proof. We use induction on i.

Basis $i = 1$: The right-hand side reduces to Eq_{1j}, because $\ell = 1$. By Lemma 3.18, $Xh_{1j} = Eq_{1j} \vee Mh_{0j}$, which is Eq_{1j} because $Mh_{0j} = 0$ for all j.

Induction step: The induction assumption is that $Xh_{i-1,j}$ is as claimed. Now we have

$$\begin{aligned}
&\exists \ell \in [1,i] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-1] : Pv_{x,j-1}) \\
&= Eq_{ij} \vee \exists \ell \in [1,i-1] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-1] : Pv_{x,j-1}) \\
&= Eq_{ij} \vee (Pv_{i-1,j-1} \wedge \exists \ell \in [1,i-1] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-2] : Pv_{x,j-1})) \\
&= Eq_{ij} \vee (Pv_{i-1,j-1} \wedge Xh_{i-1,j}) \qquad \text{(ind. assump.)} \\
&= Eq_{ij} \vee Mh_{i-1,j} \qquad \text{(Lemma 3.18)} \\
&= Xh_{ij} \qquad \text{(Lemma 3.18)}
\end{aligned}$$

□

At first sight, we cannot use Lemma 3.19 to compute even a single bit in constant time, let alone a whole vector Xh_{*j}. However, it can be done, but we need more bit operations:

- Let $\veebar$ denote the xor-operation: $0 \veebar 1 = 1 \veebar 0 = 1$ and $0 \veebar 0 = 1 \veebar 1 = 0$.
- A bitvector is interpreted as an integer and we use addition as a bit operation. The carry mechanism in addition plays a key role. For example $0001 + 0111 = 1000$.

In the following, for a bitvector B, we will write

$$B = B[1..m] = B[m]B[m-1]\ldots B[1]$$

The reverse order of the bits reflects the interpretation as an integer.

Lemma 3.20: Denote $X = Xh_{*j}$, $E = Eq_{*j}$, $P = Pv_{*,j-1}$ and let $Y = (((E \wedge P) + P) \veebar P) \vee E$. Then $X = Y$.

Proof. By Lemma 3.19, $X[i] = 1$ iff and only if

a) $E[i] = 1$ or

b) $\exists \ell \in [1, i] : E[\ell \dots i] = 00 \cdots 01 \wedge P[\ell \dots i-1] = 11 \cdots 1$.

and $X[i] = 0$ iff and only if

c) $E_{1\dots i} = 00 \cdots 0$ or

d) $\exists \ell \in [1, i] : E[\ell \dots i] = 00 \cdots 01 \wedge P[\ell \dots i-1] \neq 11 \cdots 1$.

We prove that $Y[i] = X[i]$ in all of these cases:

a) The definition of Y ends with "$\vee E$" which ensures that $Y[i] = 1$ in this case.

b) The following calculation shows that $Y[i] = 1$ in this case:

$$\begin{aligned}
& \;\; i \qquad\quad \ell \\
E[\ell \dots i] &= \texttt{00...01} \\
P[\ell \dots i] &= \texttt{b1...11} \\
(E \wedge P)[\ell \dots i] &= \texttt{00...01} \\
((E \wedge P) + P)[\ell \dots i] &= \bar{\texttt{b}}\texttt{0...0c} \\
(((E \wedge P) + P) \veebar P)[\ell \dots i] &= \texttt{11...1}\bar{\texttt{c}} \\
Y = ((((E \wedge P) + P) \veebar P) \vee E)[\ell \dots i] &= \texttt{11...11}
\end{aligned}$$

where $\texttt{b}$ is the unknown bit $P[i]$, $\texttt{c}$ is the possible carry bit coming from the summation of bits $1 \dots, \ell - 1$, and $\bar{\texttt{b}}$ and $\bar{\texttt{c}}$ are their negations.

c) Because for all bitvectors B, $0 \wedge B = 0$ ja $0 + B = B$, we get $Y = (((0 \wedge P) + P) \veebar P) \vee 0 = (P \veebar P) \vee 0 = 0$.

d) Consider the calculation in case b). A key point there is that the carry bit in the summation travels from position ℓ to i and produces $\bar{\texttt{b}}$ to position i. The difference in this case is that at least one bit $P[k]$, $\ell \leq k < i$, is zero, which stops the carry at position k. Thus $((E \wedge P) + P)[i] = \texttt{b}$ and $Y[i] = (b \veebar b) \vee 0 = 0$.

$\square$

As a final detail, we compute the bottom row values g_{mj} using the equalities $g_{m0} = m$ ja $g_{mj} = g_{m,j-1} + \Delta h_{mj}$.

Algorithm 3.21: Myers' bitparallel algorithm
Input: text $T[1..n]$, pattern $P[1..m]$, and integer k
Output: end positions of all approximate occurrences of P

```
 (1)  for $c \in \Sigma$ do $B[c] \leftarrow 0^m$
 (2)  for $i \leftarrow 1$ to $m$ do $B[P[i]][i] = 1$
 (3)  $Pv \leftarrow 1^m$; $Mv \leftarrow 0$; $g \leftarrow m$
 (4)  for $j \leftarrow 1$ to $n$ do
 (5)      $Eq \leftarrow B[T[j]]$
 (6)      $Xh \leftarrow (((Eq \wedge Pv) + Pv) \veebar Pv) \vee Eq$
 (7)      $Ph \leftarrow Mv \vee \neg(Xh \vee Pv)$
 (8)      $Mh \leftarrow Pv \wedge Xh$
 (9)      $Xv \leftarrow Eq \vee Mv$
(10)      $Pv \leftarrow (Mh << 1) \vee \neg(Xv \vee (Ph << 1))$
(11)      $Mv \leftarrow (Ph << 1) \wedge Xv$
(12)      $g \leftarrow g + Ph[m] - Mh[m]$
(13)      if $g \leq k$ then output $j$
```

On an integer alphabet, when $m \le w$:

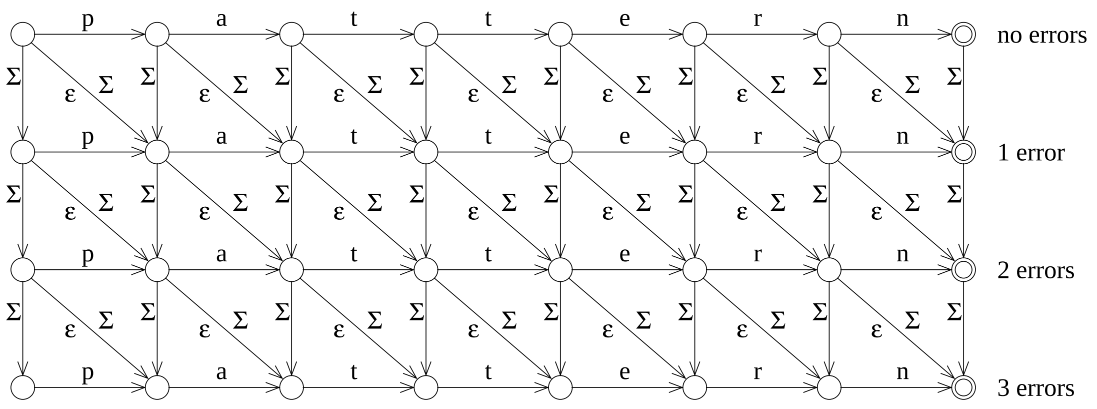
- Pattern preprocessing time is $\mathcal{O}(m + \sigma)$.
- Search time is $\mathcal{O}(n)$.

When $m > w$, we can store each bit vector in $\lceil m/w \rceil$ machine words:

- The worst case search time is $\mathcal{O}(n\lceil m/w \rceil)$.
- Using Ukkonen's cut-off heuristic, it is possible reduce the average case search time to $\mathcal{O}(n\lceil k/w \rceil)$.

There are also algorithms for bitparallel simulation of a nondeterministic automaton that recognizes the aprroximate occurrences of the pattern.

Example 3.22:
$P =$ `pattern`, $k = 3$



Another way to utilize Lemma 3.15 ($\Delta h_{ij}, \Delta v_{ij} \in \{-1, 0, 1\}$) is to use precomputed tables to process multiple matrix cells at a time.

- There are at most 3^m different columns. Thus there exists a deterministic automaton with 3^m states and $\sigma 3^m$ transitions that can find all approximate occurrences in $\mathcal{O}(n)$ time. However, the space and constructions time of the automaton can be too big to be practical.

- There is a super-alphabet algorithm that processes $\mathcal{O}(\log_\sigma n)$ characters at a time and $\mathcal{O}(\log_\sigma^2 n)$ matrix cells at a time using lookup tables of size $\mathcal{O}(n)$. This gives time complexity $\mathcal{O}(mn/\log_\sigma^2 n)$.

- A practical variant uses smaller lookup tables to compute multiple entries of a column at a time.

Baeza-Yates–Perleberg Filtering Algorithm

A filtering algorithm for approximate string matching searches the text for factors having some property that satisfies the following conditions:

1. Every approximate occurrence of the pattern has this property.
2. Strings having this property are reasonably rare.
3. Text factors having this property can be found quickly.

Each text factor with the property is a potential occurrence, which is then verified for whether it is an actual approximate occurrence.

Filtering algorithms can achieve linear or even sublinear average case time complexity.

The following lemma shows the property used by the Baeza-Yates–Perleberg algorithm and proves that it satisfies the first condition.

Lemma 3.23: Let $P_1P_2 \ldots P_{k+1} = P$ be a partitioning of the pattern P into $k+1$ nonempty factors. Any string S with $ed(P,S) \leq k$ contains P_i as a factor for some $i \in [1..k+1]$.

Proof. Each single symbol edit operation can change at most one of the pattern factors P_i. Thus any set of at most k edit operations leaves at least one of the factors untouched. □

The algorithm has two phases:

Filtration: Search the text T for exact occurrences of the pattern factors P_i. Using the Aho–Corasick algorithm this takes $\mathcal{O}(n)$ time for a constant alphabet.

Verification: An area of length $\mathcal{O}(m)$ surrounding each potential occurrence found in the filtration phase is searched using the standard dynamic programming algorithm in $\mathcal{O}(m^2)$ time.

The worst case time complexity is $\mathcal{O}(m^2 n)$, which can be reduced to $\mathcal{O}(mn)$ by combining any overlapping areas to be searched.

Let us analyze the average case time complexity of the verification phase.

- The best pattern partitioning is as even as possible. Then each pattern factor has length at least $r = \lfloor m/(k+1) \rfloor$.
- The expected number of exact occurrences of a random string of length r in a random text of length n is at most n/σ^r.
- The expected total verification time is at most

$$\mathcal{O}\left(\frac{m^2(k+1)n}{\sigma^r}\right) \le \mathcal{O}\left(\frac{m^3 n}{\sigma^r}\right) \quad .$$

  This is $\mathcal{O}(n)$ if $r \ge 3\log_\sigma m$.
- The condition $r \ge 3\log_\sigma m$ is satisfied when $(k+1) \le m/(3\log_\sigma m + 1)$.

Theorem 3.24: The average case time complexity of the Baeza-Yates–Perleberg algorithm is $\mathcal{O}(n)$ when $k \le m/(3\log_\sigma m + 1) - 1$.

Many variations of the algorithm have been suggested:

- The filtration can be done with a different multiple exact string matching algorithm.
- The verification time can be reduced using a technique called hierarchical verification.
- The pattern can be partitioned into fewer than $k + 1$ pieces, which are searched allowing a small number of errors.

A lower bound on the average case time complexity is $\Omega(n(k + \log_\sigma m)/m)$, and there exists a filtering algorithm matching this bound.

Summary: Approximate String Matching

We have seen two main types of algorithms for approximate string matching:

- Basic dynamic programming time complexity is $\mathcal{O}(mn)$. The time complexity can be improved to $\mathcal{O}(kn)$ using diagonal monotonicity, and to $\mathcal{O}(n\lceil m/w \rceil)$ using bitparallelism.
- Filtering algorithms can improve average case time complexity and are the fastest in practice when k is not too large.

Similar techniques can be useful for other variants of edit distance but not always straightforwardly.

4. Suffix Trees and Arrays

Let $T = T[0..n)$ be the text. For $i \in [0..n]$, let T_i denote the suffix $T[i..n)$. Furthermore, for any subset $C \in [0..n]$, we write $T_C = \{T_i \mid i \in C\}$. In particular, $T_{[0..n]}$ is the set of all suffixes of T.

Suffix tree and suffix array are search data structures for the set $T_{[0..n]}$.

- Suffix tree is a compact trie for $T_{[0..n]}$.
- Suffix array is an ordered array for $T_{[0..n]}$.

They support fast exact string matching on T:

- A pattern P has an occurrence starting at position i if and only if P is a prefix of T_i.
- Thus we can find all occurrences of P by a prefix search in $T_{[0..n]}$.

A data structure supporting fast string matching is called a text index.

There are numerous other applications too, as we will see later.

The set $T_{[0..n]}$ contains $|T_{[0..n]}| = n + 1$ strings of total length $||T_{[0..n]}|| = \Theta(n^2)$. It is also possible that $\Sigma LCP(T_{[0..n]}) = \Theta(n^2)$, for example, when $T = \mathtt{a}^n$ or $T = XX$ for any string X.

- A basic trie has $\Theta(n^2)$ nodes for most texts, which is too much.
- A compact trie with $\mathcal{O}(n)$ nodes and an ordered array with $n + 1$ entries have linear size.
- A compact ternary trie has $\mathcal{O}(n)$ nodes too. However, the construction algorithms and some other algorithms we will see are not straightforward to adapt for it.

Even for a compact trie or an ordered array, we need a specialized construction algorithm, because any general construction algorithm would need $\Omega(\Sigma LCP(T_{[0..n]}))$ time.

Suffix Tree

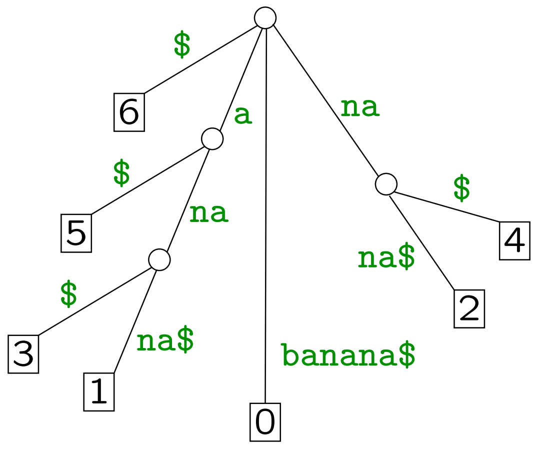
The suffix tree of a text T is the compact trie of the set $T_{[0..n]}$ of all suffixes of T.

We assume that there is an extra character $\$ \notin \Sigma$ at the end of the text. That is, $T[n] = \$$ and $T_i = T[i..n]$ for all $i \in [0..n]$. Then:

- No suffix is a prefix of another suffix, i.e., the set $T_{[0..n]}$ is prefix free.
- All nodes in the suffix tree representing a suffix are leaves.

This simplifies algorithms.

Example 4.1: $T =$ banana$.



As with tries, there are many possibilities for implementing the child operation. We again avoid this complication by assuming that σ is constant. Then the size of the suffix tree is $\mathcal{O}(n)$:

- There are exactly $n + 1$ leaves and at most n internal nodes.
- There are at most $2n$ edges. The edge labels are factors of the text and can be represented by pointers to the text.

Given the suffix tree of T, all occurrences of P in T can be found in time $\mathcal{O}(|P| + occ)$, where occ is the number of occurrences.

Brute Force Construction

Let us now look at algorithms for constructing the suffix tree. We start with a brute force algorithm with time complexity $\Theta(\Sigma LCP(T_{[0..n]}))$. Later we will modify this algorithm to obtain a linear time complexity.

The idea is to add suffixes to the trie one at a time starting from the longest suffix. The insertion procedure is essentially the same as we saw in Algorithm 1.2 (insertion into trie) except it has been modified to work on a compact trie instead of a trie.

Let S_u denote the string represented by a node u. The suffix tree representation uses four functions:

child(u, c) is the child v of node u such that the label of the edge (u, v) starts with the symbol c, and $\perp$ if u has no such child.

parent(u) is the parent of u.

depth(u) is the length of S_u.

start(u) is the starting position of some occurrence of S_u in T.

Then

- $S_u = T[\mathit{start}(u) \ldots \mathit{start}(u) + \mathit{depth}(u))$.
- $T[\mathit{start}(u) + \mathit{depth}(\mathit{parent}(u)) \ldots \mathit{start}(u) + \mathit{depth}(u))$ is the label of the edge $(\mathit{parent}(u), u)$.

A locus in the suffix tree is a pair (u, d) where $\mathit{depth}(\mathit{parent}(u)) < d \leq \mathit{depth}(u)$. It represents

- the uncompact trie node that would be at depth d along the edge $(\mathit{parent}(u), u)$, and
- the corresponding string $S_{(u,d)} = T[\mathit{start}(u) \ldots \mathit{start}(u) + d)$.

Every factor of T is a prefix of a suffix and thus has a locus along the path from the root to the leaf representing that suffix.

During the construction, we need to create nodes at an existing locus in the middle of an edge, splitting the edge into two edges:

```
CreateNode($u, d$)        // $d < \mathit{depth}(u)$
  (1) $i \leftarrow \mathit{start}(u)$; $p \leftarrow \mathit{parent}(u)$
  (2) create new node $v$
  (3) $\mathit{start}(v) \leftarrow i$; $\mathit{depth}(v) \leftarrow d$
  (4) $\mathit{child}(v, T[i + d]) \leftarrow u$; $\mathit{parent}(u) \leftarrow v$
  (5) $\mathit{child}(p, T[i + \mathit{depth}(p)]) \leftarrow v$; $\mathit{parent}(v) \leftarrow p$
  (6) return $v$
```

Now we are ready to describe the construction algorithm.

Algorithm 4.2: Brute force suffix tree construction
Input: text $T[0..n]$ ($T[n] = \$$)
Output: suffix tree of T: $root$, *child*, *parent*, *depth*, *start*

```
 (1) create new node $root$; $depth(root) \leftarrow 0$
 (2) $u \leftarrow root$; $d \leftarrow 0$        // $(u, d)$ is the active locus
 (3) for $i \leftarrow 0$ to $n$ do         // insert suffix $T_i$
 (4)       while $d = depth(u)$ and $child(u, T[i+d]) \neq \perp$ do
 (5)             $u \leftarrow child(u, T[i+d])$; $d \leftarrow d + 1$
 (6)             while $d < depth(u)$ and $T[start(u) + d] = T[i+d]$ do $d \leftarrow d + 1$
 (7)       if  $d < depth(u)$ then        // $(u, d)$ is in the middle of an edge
 (8)             $u \leftarrow$ CreateNode$(u, d)$
 (9)       CreateLeaf$(i, u)$
(10)       $u \leftarrow root$; $d \leftarrow 0$
```

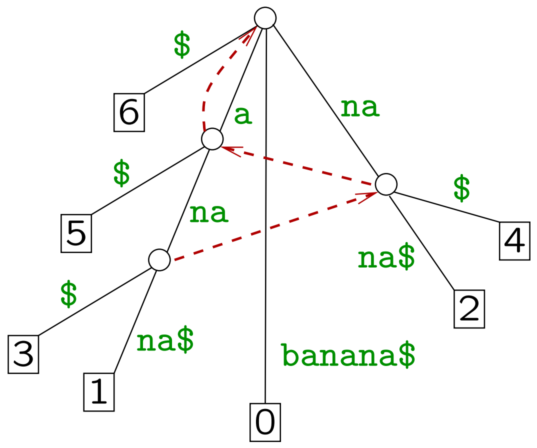
```
CreateLeaf$(i, u)$        // Create leaf representing suffix $T_i$
 (1) create new leaf $w$
 (2) $start(w) \leftarrow i$; $depth(w) \leftarrow n - i + 1$
 (3) $child(u, T[i+d]) \leftarrow w$; $parent(w) \leftarrow u$        // Set $u$ as parent
 (4) return $w$
```

Suffix Links

The key to efficient suffix tree construction are suffix links:

> *slink*(u) is the node v such that S_v is the longest proper suffix of S_u, i.e., if $S_u = T[i..j)$ then $S_v = T[i+1..j)$.

Example 4.3: The suffix tree of $T =$ `banana$` with internal node suffix links.



Suffix links are well defined for all nodes except the root.

Lemma 4.4: If the suffix tree of T has a node u representing $T[i..j)$ for any $0 \leq i < j \leq n$, then it has a node v representing $T[i+1..j)$.

Proof. If u is the leaf representing the suffix T_i, then v is the leaf representing the suffix T_{i+1}.

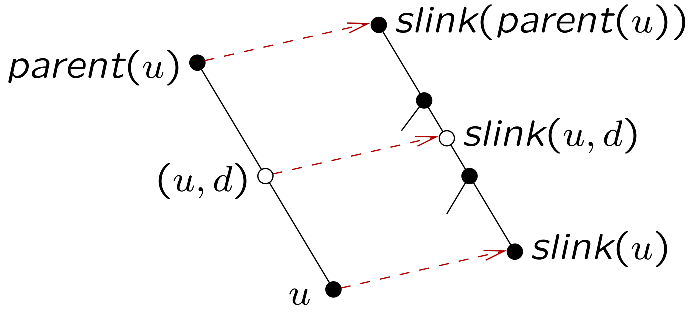
If u is an internal node, then it has two child edges with labels starting with different symbols, say a and b, which means that $T[i..j)a$ and $T[i..j)b$ are both factors of T. Then, $T[i+1..j)a$ and $T[i+1..j)b$ are factors of T too, and thus there must be a branching node v representing $T[i+1..j)$. $\square$

Usually, suffix links are needed only for internal nodes. For root, we define *slink*$(root) = root$.

Suffix links are the same as Aho–Corasick failure links but Lemma 4.4 ensures that *depth*(*slink*(u)) = *depth*(u) – 1. This is not the case for an arbitrary trie or a compact trie.

Suffix links are stored for compact trie nodes only, but we can define and compute them for any locus (u, d):

```
slink(u, d)
(1) $v \leftarrow$ slink(parent($u$))
(2) while depth($v$) $< d - 1$ do
(3)         $v \leftarrow$ child($v, T$[start($u$) + depth($v$) + 1])
(4) return $(v, d - 1)$
```



The same idea can be used for computing the suffix links during or after the brute force construction.

```
ComputeSlink($u$)
  (1) $d \leftarrow \textit{depth}(u)$
  (2) $v \leftarrow \textit{slink}(\textit{parent}(u))$
  (3) while $\textit{depth}(v) < d - 1$ do
  (4)       $v \leftarrow \textit{child}(v, T[\textit{start}(u) + \textit{depth}(v) + 1])$
  (5) if $\textit{depth}(v) > d - 1$ then          // no node at $(v, d-1)$
  (6)       $v \leftarrow$ CreateNode$(v, d-1)$
  (7) $\textit{slink}(u) \leftarrow v$
```

The procedure CreateNode$(v, d-1)$ sets $\textit{slink}(v) = \perp$.

The algorithm uses the suffix link of the parent, which must have been computed before. Otherwise the order of computation does not matter.

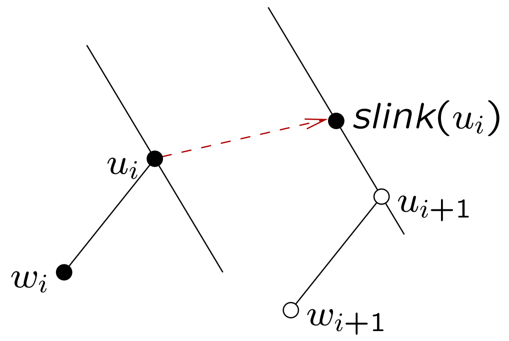
The creation of a new node on line (6) is never needed in a fully constructed suffix tree, but during the brute force algorithm the necessary node may not exist yet:

- If a new internal node u_i was created during the insertion of the suffix T_i, there exists an earlier suffix T_j, $j < i$ that branches at u_i into a different direction than T_i.
- Then *slink*(u_i) represents a prefix of T_{j+1} and thus exists at least as a locus on the path labelled T_{j+1}. However, it might not become a branching node until the insertion of T_{i+1}.
- In such a case, ComputeSlink(u_i) creates *slink*(u_i) a moment before it would otherwise be created by the brute force construction.

McCreight's Algorithm

McCreight's suffix tree construction is a simple modification of the brute force algorithm that computes the suffix links during the construction and uses them as short cuts:

- Consider the situation, where we have just added a leaf w_i representing the suffix T_i as a child to a node u_i. The next step is to add w_{i+1} as a child to a node u_{i+1}.
- The brute force algorithm finds u_{i+1} by traversing from the root. McCreight's algorithm takes a short cut to *slink*(u_i).



- This is safe because *slink*(u_i) represents a prefix of T_{i+1}.

Algorithm 4.5: McCreight
Input: text $T[0..n]$ $(T[n] = \$)$
Output: suffix tree of T: $root$, *child*, *parent*, *depth*, *start*, *slink*

```
 (1)  create new node $root$; $\mathit{depth}(root) \leftarrow 0$; $\mathit{slink}(root) \leftarrow root$
 (2)  $u \leftarrow root$; $d \leftarrow 0$        // $(u, d)$ is the active locus
 (3)  for $i \leftarrow 0$ to $n$ do        // insert suffix $T_i$
 (4)      while $d = \mathit{depth}(u)$ and $\mathit{child}(u, T[i+d]) \neq \bot$ do
 (5)          $u \leftarrow \mathit{child}(u, T[i+d])$; $d \leftarrow d + 1$
 (6)          while $d < \mathit{depth}(u)$ and $T[\mathit{start}(u) + d] = T[i+d]$ do $d \leftarrow d + 1$
 (7)      if $d < \mathit{depth}(u)$ then        // $(u, d)$ is in the middle of an edge
 (8)          $u \leftarrow$ CreateNode$(u, d)$
 (9)      CreateLeaf$(i, u)$
(10)      if $\mathit{slink}(u) = \bot$ then ComputeSlink$(u)$
(11)      $u \leftarrow \mathit{slink}(u)$; $d \leftarrow d - 1$
```

Theorem 4.6: Let T be a string of length n over an alphabet of constant size. McCreight's algorithm computes the suffix tree of T in $\mathcal{O}(n)$ time.

Proof. Insertion of a suffix T_i takes constant time except in two points:

- The while loops on lines (4)–(6) traverse from the node *slink*(u_i) to u_{i+1}. Every round in these loops increments d. The only place where d decreases is on line (11) and even then by one. Since d can never exceed n, the total time on lines (4)–(6) is $\mathcal{O}(n)$.

- The while loop on lines (3)–(4) during a call to ComputeSlink(u_i) traverses from the node *slink*(*parent*(u_i)) to *slink*(u_i). Let d'_i be the depth of *parent*(u_i). Clearly, $d'_{i+1} \geq d'_i - 1$, and every round in the while loop during ComputeSlink(u_i) increases d'_{i+1}. Since d'_i can never be larger than n, the total time in the loop on lines (3)–(4) in ComputeSlink is $\mathcal{O}(n)$.

$\square$

There are other linear time algorithms for suffix tree construction:

- Weiner's algorithm was the first. It inserts the suffixes into the tree in the opposite order: $T_n, T_{n-1}, \ldots, T_0$.

- Ukkonen's algorithm constructs suffix tree first for $T[0..1)$ then for $T[0..2)$, etc.. The algorithm is structured differently, but performs essentially the same tree traversal as McCreight's algorithm.

- All of the above are linear time only for constant alphabet size. Farach's algorithm achieves linear time for an integer alphabet of polynomial size. The algorithm is complicated and unpractical.

- Practical linear time construction for an integer alphabet is possible via suffix array.

Applications of Suffix Tree

Let us have a glimpse of the numerous applications of suffix trees.

Exact String Matching

As already mentioned earlier, given the suffix tree of the text, all occ occurrences of a pattern P can be found in time $\mathcal{O}(|P| + occ)$.

Even if we take into account the time for constructing the suffix tree, this is asymptotically as fast as Knuth–Morris–Pratt for a single pattern and Aho–Corasick for multiple patterns.

However, the primary use of suffix trees is in indexed string matching, where we can afford to spend a lot of time in preprocessing the text, but must then answer queries very quickly.

Approximate String Matching

Several approximate string matching algorithms achieving $\mathcal{O}(kn)$ worst case time complexity are based on suffix trees.

Filtering algorithms that reduce approximate string matching to exact string matching such as partitioning the pattern into $k+1$ factors, can use suffix trees in the filtering phase.

Another approach is to generate all strings in the k-neighborhood of the pattern, i.e., all strings within edit distance k from the pattern and search for them in the suffix tree.

The best practical algorithms for indexed approximate string matching are hybrids of the last two approaches. For example, partition the pattern into $\ell \leq k+1$ factors and find approximate occurrences of the factors with edit distance $\lfloor k/\ell \rfloor$ using the neighborhood method in the filtering phase.

Text Statistics

Suffix tree is useful for computing all kinds of statistics on the text. For example:

- Every locus in the suffix tree represents a factor of the text and, vice versa, every factor is represented by some locus. Thus the number of distinct factors in the text is exactly the number of distinct locuses, which can be computed by a traversal of the suffix tree in $\mathcal{O}(n)$ time even though the resulting value is typically $\Theta(n^2)$.

- The longest repeating factor of the text is the longest string that occurs at least twice in the text. It is represented by the deepest internal node in the suffix tree.

Generalized Suffix Tree

A generalized suffix tree of two strings S and T is the suffix tree of the string $S£T\$$, where $£$ and \$ are symbols that do not occur elsewhere in S and T.

Each leaf is marked as an S-leaf or a T-leaf according to the starting position of the suffix it represents. Using a depth first traversal, we determine for each internal node if its subtree contains only S-leafs, only T-leafs, or both. The deepest node that contains both represents the longest common factor of S and T. It can be computed in linear time.

The generalized suffix tree can also be defined for more than two strings.

AC Automaton for the Set of Suffixes

As already mentioned, a suffix tree with suffix links is essentially an Aho–Corasick automaton for the set of all suffixes.

- We saw that it is possible to follow suffix link / failure transition from any locus, not just from suffix tree nodes.
- Following such an implicit suffix link may take more than a constant time, but the total time during the scanning of a string with the automaton is linear in the length of the string. This can be shown with a similar argument as in the construction algorithm.

Thus suffix tree is asymptotically as fast to operate as the AC automaton, but needs much less space.

Matching Statistics

The matching statistics of a string $S[0..n)$ with respect to a string T is an array $MS[0..n)$, where $MS[i]$ is a pair (ℓ_i, p_i) such that

1. $S[i..i + \ell_i)$ is the longest prefix of S_i that is a factor of T, and
2. $T[p_i..p_i + \ell_i) = S[i..i + \ell_i)$.

Matching statistics can be computed by using the suffix tree of T as an AC-automaton and scanning S with it.

- If before reading $S[i]$ we are at the locus (v, d) in the automaton, then $S[i - d..i) = T[j..j + d)$, where $j = \mathit{start}(v)$. If reading $S[i]$ causes a failure transition, then $MS[i - d] = (d, j)$.
- Following the failure transition decrements d and thus increments $i - d$ by one. Following a normal transition/edge, increments both i and d by one, and thus $i - d$ stays the same. Thus all entries are computed.

From the matching statistics, we can easily compute the longest common factor of S and T. Because we need the suffix tree only for T, this saves space compared to a generalized suffix tree.

Matching statistics are also used in some approximate string matching algorithms.

LCA Preprocessing

The lowest common ancestor (LCA) of two nodes u and v is the deepest node that is an ancestor of both u and v. Any tree can be preprocessed in linear time so that the LCA of any two nodes can be computed in constant time. The details are omitted here.

- Let w_i and w_j be the leaves of the suffix tree of T that represent the suffixes T_i and T_j. The lowest common ancestor of w_i and w_j represents the longest common prefix of T_i and T_j. Thus

  $$lcp(T_i, T_j) = depth(LCA(w_i, w_j)),$$

  which can be computed in constant time using the suffix tree with LCA preprocessing.

- The longest common prefix of two suffixes S_i and T_j from two different strings S and T is called the longest common extension. Using the generalized suffix tree with LCA preprocessing, the longest common extension for any pair of suffixes can be computed in constant time.

Some $\mathcal{O}(kn)$ worst case time approximate string matching algorithms use longest common extension data structures.

Longest Palindrome

A palindrome is a string that is its own reverse. For example, `saippuakauppias` is a palindrome.

We can use the LCA preprocessed generalized suffix tree of a string T and its reverse T^R to find the longest palindrome in T in linear time.

- Let k_i be the length of the longest common extension of T_{i+1} and T^R_{n-i}, which can be computed in constant time. Then $T[i - k_i..i + k_i]$ is the longest odd length palindrome with the middle at i.
- We can find the longest odd length palindrome by computing k_i for all $i \in [0..n)$ in $\mathcal{O}(n)$ time.
- The longest even length palindrome can be found similarly in $\mathcal{O}(n)$ time. The longest palindrome overall is the longer of the two.

Suffix Array

The suffix array of a text T is a lexicographically ordered array of the set $T_{[0..n]}$ of all suffixes of T. More precisely, the suffix array is an array $SA[0..n]$ of integers containing a permutation of the set $[0..n]$ such that $T_{SA[0]} < T_{SA[1]} < \cdots < T_{SA[n]}$.

A related array is the inverse suffix array SA^{-1} which is the inverse permutation, i.e., $SA^{-1}[SA[i]] = i$ for all $i \in [0..n]$. The value $SA^{-1}[j]$ is the lexicographical rank of the suffix T_j

As with suffix trees, it is common to add the end symbol $T[n] = \$$. It has no effect on the suffix array assuming $ is smaller than any other symbol.

Example 4.7: The suffix array and the inverse suffix array of the text $T =$ `banana$`.

i	$SA[i]$	$T_{SA[i]}$
0	6	`$`
1	5	`a$`
2	3	`ana$`
3	1	`anana$`
4	0	`banana$`
5	4	`na$`
6	2	`nana$`

j	$SA^{-1}[j]$	
0	4	`banana$`
1	3	`anana$`
2	6	`nana$`
3	2	`ana$`
4	5	`na$`
5	1	`a$`
6	0	`$`

Suffix array is much simpler data structure than suffix tree. In particular, the type and the size of the alphabet are usually not a concern.

- The size on the suffix array is $\mathcal{O}(n)$ on any alphabet.
- We will later see that the suffix array can be constructed in the same asymptotic time it takes to sort the characters of the text.

Suffix array construction algorithms are quite fast in practice too. Probably the fastest way to construct a suffix tree is to construct a suffix array first and then use it to construct the suffix tree. (We will see how in a moment.)

Suffix arrays are rarely used alone but are augmented with other arrays and data structures depending on the application. We will see some of them in the next slides.

Exact String Matching

As with suffix trees, exact string matching in T can be performed by a prefix search on the suffix array. The answer can be conveniently given as a contiguous interval $SA[b..e)$ that contains the suffixes with the given prefix. The interval can be found using string binary search.

- If we have the additional arrays $LLCP$ and $RLCP$, the result interval can be computed in $\mathcal{O}(|P| + \log n)$ time.
- Without the additional arrays, we have the same time complexity on average but the worst case time complexity is $\mathcal{O}(|P| \log n)$.
- We can then count the number of occurrences in $\mathcal{O}(1)$ time, list all occ occurrences in $\mathcal{O}(occ)$ time, or list a sample of k occurrences in $\mathcal{O}(k)$ time.

An alternative algorithm for computing the interval $SA[b..e)$ is called backward search. It is commonly used with compressed representations of suffix arrays and will be covered in the course Data Compression Techniques.

LCP Array

Efficient string binary search uses the arrays $LLCP$ and $RLCP$. However, for many applications, the suffix array is augmented with the lcp array of Definition 1.11 (Lecture 2). For all $i \in [1..n]$, we store

$$LCP[i] = lcp(T_{SA[i]}, T_{SA[i-1]})$$

Example 4.8: The LCP array for $T =$ `banana$`.

i	$SA[i]$	$LCP[i]$	$T_{SA[i]}$
0	6		`$`
1	5	0	`a$`
2	3	1	`ana$`
3	1	3	`anana$`
4	0	0	`banana$`
5	4	0	`na$`
6	2	2	`nana$`

Using the solution of Exercise 2.4 (construction of compact trie from sorted array and LCP array), the suffix tree can be constructed from the suffix and LCP arrays in linear time.

However, many suffix tree applications can be solved using the suffix and LCP arrays directly. For example:

- The longest repeating factor is marked by the maximum value in the LCP array.
- The number of distinct factors can be compute by the formula

$$\frac{n(n+1)}{2} + 1 - \sum_{i=1}^{n} LCP[i]$$

  since it equals the number of nodes in the uncompact suffix trie, for which we can use Theorem 1.17.
- Matching statistics of S with respect to T can be computed in linear time using the generalized suffix array of S and T (i.e., the suffix array of $S£T\$$) and its LCP array (exercise).

LCP Array Construction

The LCP array is easy to compute in linear time using the suffix array SA and its inverse SA^{-1}. The idea is to compute the lcp values by comparing the suffixes, but skip a prefix based on a known lower bound for the lcp value obtained using the following result.

Lemma 4.9: For any $i \in [0..n)$, $LCP[SA^{-1}[i]] \geq LCP[SA^{-1}[i-1]] - 1$

Proof. For each $j \in [0..n)$, let $\Phi(j) = SA[SA^{-1}[j] - 1]$. Then $T_{\Phi(j)}$ is the immediate lexicographical predecessor of T_j and $LCP[SA^{-1}[j]] = lcp(T_j, T_{\Phi(j)})$.

- Let $\ell = LCP[SA^{-1}[i-1]]$ and $\ell' = LCP[SA^{-1}[i]]$. We want to show that $\ell' \geq \ell - 1$. If $\ell = 0$, the claim is trivially true.
- If $\ell > 0$, then for some symbol c, $T_{i-1} = cT_i$ and $T_{\Phi(i-1)} = cT_{\Phi(i-1)+1}$. Thus $T_{\Phi(i-1)+1} < T_i$ and $lcp(T_i, T_{\phi(i-1)+1}) = lcp(T_{i-1}, T_{\Phi(i-1)}) - 1 = \ell - 1$.
- If $\Phi(i) = \Phi(i-1) + 1$, then $\ell' = lcp(T_i, T_{\Phi(i)}) = lcp(T_i, T_{\Phi(i-1)+1}) = \ell - 1$.
- If $\Phi(i) \neq \Phi(i-1) + 1$, then $T_{\Phi(i-1)+1} < T_{\Phi(i)} < T_i$ and $\ell' = lcp(T_i, T_{\Phi(i)}) \geq lcp(T_i, T_{\Phi(i-1)+1}) = \ell - 1$.

$\square$

The algorithm computes the lcp values in the order that makes it easy to use the above lower bound.

Algorithm 4.10: LCP array construction
Input: text $T[0..n]$, suffix array $SA[0..n]$, inverse suffix array $SA^{-1}[0..n]$
Output: LCP array $LCP[1..n]$

```
(1) $\ell \leftarrow 0$
(2) for $i \leftarrow 0$ to $n-1$ do
(3)       $k \leftarrow SA^{-1}[i]$
(4)       $j \leftarrow SA[k-1]$        // $j = \Phi(i)$
(5)       while $T[i+\ell] = T[j+\ell]$ do $\ell \leftarrow \ell + 1$
(6)       $LCP[k] \leftarrow \ell$
(7)       if $\ell > 0$ then $\ell \leftarrow \ell - 1$
(8) return $LCP$
```

The time complexity is $\mathcal{O}(n)$:

- Everything except the while loop on line (5) takes clearly linear time.
- Each round in the loop increments ℓ. Since ℓ is decremented at most n times on line (7) and cannot grow larger than n, the loop is executed $\mathcal{O}(n)$ times in total.

RMQ Preprocessing

The range minimum query (RMQ) asks for the smallest value in a given range in an array. Any array can be preprocessed in linear time so that RMQ for any range can be answered in constant time.

In the LCP array, RMQ can be used for computing the lcp of any two suffixes.

Lemma 4.11: The length of the longest common prefix of two suffixes $T_i < T_j$ is $lcp(T_i, T_j) = \min\{LCP[k] \mid k \in [SA^{-1}[i] + 1..SA^{-1}[j]]\}$.

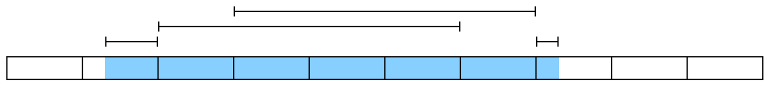
The lemma can be seen as a generalization of Lemma 1.31 (Lecture 3) and holds for any sorted array of strings. The proof is left as an exercise.

- The RMQ preprocessing of the LCP array supports the same kind of applications as the LCA preprocessing of the suffix tree, but RMQ preprocessing is simpler than LCA preprocessing.

- The RMQ preprocessed LCP array can also replace the LLCP and RLCP arrays in binary searching.

We will next describe the RMQ data structure for an arbitrary array $L[1..n]$ of integers.

- We precompute and store the minimum values for the following collection of ranges:
  - Divide $L[1..n]$ into blocks of size $\log n$.
  - For all $0 \le \ell \le \log(n/\log n))$, include all ranges that consist of 2^ℓ blocks. There are $\mathcal{O}(\log n \cdot \frac{n}{\log n}) = \mathcal{O}(n)$ such ranges.
  - Include all prefixes and suffixes of blocks. There are a total of $\mathcal{O}(n)$ of them.
- Now any range $L[i..j]$ that overlaps or touches a block boundary can be exactly covered by at most four ranges in the collection.



The minimum value in $L[i..j]$ is the minimum of the minimums of the covering ranges and can be computed in constant time.

Ranges $L[i..j]$ that are completely inside one block are handled differently.

- Let $NSV(i) = \min\{k > i \mid L[k] < L[i]\}$ (NSV=Next Smaller Value). Then the position of the minimum value in the range $L[i..j]$ is the last position in the sequence $i, NSV(i), NSV(NSV(i)), \ldots$ that is in the range. We call these the NSV positions for i.

- For each i, store the NSV positions for i up to the end of the block containing i as a bit vector $B(i)$. Each bit corresponds to a position within the block and is one if it is an NSV position. The size of $B(i)$ is $\log n$ bits and we can assume that it fits in a single machine word. Thus we need $\mathcal{O}(n)$ words to store $B(i)$ for all i.

- The position of the minimum in $L[i..j]$ is found as follows:
  - Turn all bits in $B(i)$ after position j into zeros. This can be done in constant time using bitwise shift -operations.
  - The right-most 1-bit indicates the position of the minimum. It can be found in constant time using a lookup table of size $\mathcal{O}(n)$.

All the data structures can be constructed in $\mathcal{O}(n)$ time (exercise).

Enhanced Suffix Array

The enhanced suffix array adds two more arrays to the suffix and LCP arrays to make the data structure fully equivalent to suffix tree.

- The idea is to represent a suffix tree node v representing a factor S_v by the suffix array interval of the suffixes that begin with S_v. That interval contains exactly the suffixes that are in the subtree rooted at v.
- The additional arrays support navigation in the suffix tree using this representation: one array along the regular edges, the other along suffix links.

With all the additional arrays the suffix array is not very space efficient data structure any more. Nowadays suffix arrays and trees are often replaced with compressed text indexes that provide the same functionality in much smaller space. These will be covered in the course Data Compression Techniques.

Burrows–Wheeler Transform

The Burrows–Wheeler transform (BWT) is an important technique for text compression, text indexing, and their combination compressed text indexing.

Let $T[0..n]$ be the text with $T[n] = \texttt{\$}$. For any $i \in [0..n]$, $T[i..n]T[0..i)$ is a rotation of T. Let $\mathcal{M}$ be the matrix, where the rows are all the rotations of T in lexicographical order. All columns of $\mathcal{M}$ are permutations of T. In particular:

- The first column F contains the text characters in order.
- The last column L is the BWT of T.

Example 4.12: The BWT of $T = \texttt{banana\$}$ is $L = \texttt{annb\$aa}$.

$$\begin{array}{c|c|ccccc|c|}
\multicolumn{1}{c}{} & \multicolumn{1}{c}{F} & & & & & \multicolumn{1}{c}{} & \multicolumn{1}{c}{L} \\
\cline{2-2}\cline{8-8}
 & \texttt{\$} & \texttt{b} & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{n} & \texttt{a} \\
 & \texttt{a} & \texttt{\$} & \texttt{b} & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{n} \\
 & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{\$} & \texttt{b} & \texttt{a} & \texttt{n} \\
 & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{\$} & \texttt{b} \\
 & \texttt{b} & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{\$} \\
 & \texttt{n} & \texttt{a} & \texttt{\$} & \texttt{b} & \texttt{a} & \texttt{n} & \texttt{a} \\
 & \texttt{n} & \texttt{a} & \texttt{n} & \texttt{a} & \texttt{\$} & \texttt{b} & \texttt{a} \\
\cline{2-2}\cline{8-8}
\end{array}$$

Here are some of the key properties of the BWT.

- The BWT is easy to compute using the suffix array:

$$L[i] = \begin{cases} \$ & \text{if } SA[i] = 0 \\ T[SA[i]-1] & \text{otherwise} \end{cases}$$

- The BWT is invertible, i.e., T can be reconstructed from the BWT L alone. The inverse BWT can be computed in the same time it takes to sort the characters.

- The BWT L is typically easier to compress than the text T. Many text compression algorithms are based on compressing the BWT.

- The BWT supports backward searching, a different technique for indexed exact string matching. This is used in many compressed text indexes.

BWT will be covered in more detail in the course Data Compression Techniques.

Suffix Array Construction

Suffix array construction means simply sorting the set of all suffixes.

- Using standard sorting or string sorting the time complexity is $\Omega(\Sigma LCP(T_{[0..n]}))$.
- Another possibility is to first construct the suffix tree and then traverse it from left to right to collect the suffixes in lexicographical order. The time complexity is $\mathcal{O}(n)$ on a constant alphabet.

Specialized suffix array construction algorithms are a better option, though.

Prefix Doubling

Our first specialized suffix array construction algorithm is a conceptually simple algorithm achieving $\mathcal{O}(n \log n)$ time.

Let T_i^ℓ denote the text factor $T[i..\min\{i+\ell, n+1\})$ and call it an ℓ-factor. In other words:

- T_i^ℓ is the factor starting at i and of length ℓ except when the factor is cut short by the end of the text.
- T_i^ℓ is the prefix of the suffix T_i of length ℓ, or T_i when $|T_i| < \ell$.

The idea is to sort the sets $T_{[0..n]}^\ell$ for ever increasing values of ℓ.

- First sort $T_{[0..n]}^1$, which is equivalent to sorting individual characters. This can be done in $\mathcal{O}(n \log n)$ time.
- Then, for $\ell = 1, 2, 4, 8, \ldots$, use the sorted set $T_{[0..n]}^\ell$ to sort the set $T_{[0..n]}^{2\ell}$ in $\mathcal{O}(n)$ time.
- After $\mathcal{O}(\log n)$ rounds, $\ell > n$ and $T_{[0..n]}^\ell = T_{[0..n]}$, so we have sorted the set of all suffixes.

We still need to specify, how to use the order for the set $T^{\ell}_{[0..n]}$ to sort the set $T^{2\ell}_{[0..n]}$. The key idea is assigning order preserving names (lexicographical names) for the factors in $T^{\ell}_{[0..n]}$. For $i \in [0..n]$, let N^{ℓ}_i be an integer in the range $[0..n]$ such that, for all $i, j \in [0..n]$:

$$N^{\ell}_i \leq N^{\ell}_j \text{ if and only if } T^{\ell}_i \leq T^{\ell}_j \ .$$

Then, for $\ell > n$, $N^{\ell}_i = SA^{-1}[i]$.

For smaller values of ℓ, there can be many ways of satisfying the conditions and any one of them will do. A simple choice is

$$N^{\ell}_i = |\{j \in [0, n] \mid T^{\ell}_j < T^{\ell}_i\}| \ .$$

Example 4.13: Prefix doubling for $T =$ `banana$`.

N^1		N^2		N^4		$N^8 = SA^{-1}$	
4	`b`	4	`ba`	4	`bana`	4	`banana$`
1	`a`	2	`an`	3	`anan`	3	`anana$`
5	`n`	5	`na`	6	`nana`	6	`nana$`
1	`a`	2	`an`	2	`ana$`	2	`ana$`
5	`n`	5	`na`	5	`na$`	5	`na$`
1	`a`	1	`a$`	1	`a$`	1	`a$`
0	`$`	0	`$`	0	`$`	0	`$`

Now, given N^ℓ, for the purpose of sorting, we can use

- N_i^ℓ to represent T_i^ℓ
- the pair $(N_i^\ell, N_{i+\ell}^\ell)$ to represent $T_i^{2\ell} = T_i^\ell T_{i+\ell}^\ell$.

Thus we can sort $T_{[0..n]}^{2\ell}$ by sorting pairs of integers, which can be done in $\mathcal{O}(n)$ time using LSD radix sort.

Theorem 4.14: The suffix array of a string $T[0..n]$ can be constructed in $\mathcal{O}(n \log n)$ time using prefix doubling.

- The technique of assigning order preserving names to factors whose lengths are powers of two is called the Karp–Miller–Rosenberg naming technique. It was developed for other purposes in the early seventies when suffix arrays did not exist yet.
- The best practical variant is the Larsson–Sadakane algorithm, which uses ternary quicksort instead of LSD radix sort for sorting the pairs, but still achieves $\mathcal{O}(n \log n)$ total time.

Let us return to the first phase of the prefix doubling algorithm: assigning names N_i^1 to individual characters. This is done by sorting the characters, which is easily within the time bound $\mathcal{O}(n \log n)$, but sometimes we can do it faster:

- On an ordered alphabet, we can use ternary quicksort for time complexity $\mathcal{O}(n \log \sigma_T)$ where σ_T is the number of distinct symbols in T.
- On an integer alphabet of size n^c for any constant c, we can use LSD radix sort with radix n for time complexity $\mathcal{O}(n)$.

After this, we can replace each character $T[i]$ with N_i^1 to obtain a new string T':

- The characters of T' are integers in the range $[0..n]$.
- The character $T'[n] = 0$ is the unique, smallest symbol, i.e., $.
- The suffix arrays of T and T' are exactly the same.

Thus we can construct the suffix array using T' as the text instead of T.

As we will see next, the suffix array of T' can be constructed in linear time. Then sorting the characters of T to obtain T' is the asymptotically most expensive operation in the suffix array construction of T for any alphabet.

Recursive Suffix Array Construction

Let us now describe linear time algorithms for suffix array construction. We assume that the alphabet of the text $T[0..n)$ is $[1..n]$ and that $T[n] = 0$ (=$ in the examples).

The outline of the algorithms is:

0. Choose a subset $C \subset [0..n]$.

1. Sort the set T_C. This is done as follows:

  (a) Construct a reduced string R of length $|C|$, whose characters are order preserving names of text factors starting at the positions in C.

  (b) Construct the suffix array of R recursively.

2. Sort the set $T_{[0..n]}$ using the order of T_C.

Assume that

- $|C| \leq \alpha n$ for a constant $\alpha < 1$, and
- excluding the recursive call, all steps in the algorithm take linear time.

Then the total time complexity can be expressed as the recurrence $t(n) = \mathcal{O}(n) + t(\alpha n)$, whose solution is $t(n) = \mathcal{O}(n)$.

To make the scheme work, the set C must satisfy two nontrivial conditions:

1. There exists an appropriate reduced string R.
2. Given sorted T_C the suffix array of T is easy to construct.

Finding sets C that satisfy both conditions is difficult, but there are two different methods leading to two different algorithms:

- DC3 uses difference cover sampling
- SAIS uses induced sorting

Difference Cover Sampling

A difference cover D_q modulo q is a subset of $[0..q)$ such that all values in $[0..q)$ can be expressed as a difference of two elements in D_q modulo q. In other words:

$$[0..q) = \{i - j \bmod q \mid i, j \in D_q\} \ .$$

Example 4.15: $D_7 = \{1, 2, 4\}$

$$\begin{array}{lll}
1 - 1 = 0 & 1 - 4 = -3 \equiv 4 & (\mathrm{mod}\ q) \\
2 - 1 = 1 & 2 - 4 = -2 \equiv 5 & (\mathrm{mod}\ q) \\
4 - 2 = 2 & 1 - 2 = -1 \equiv 6 & (\mathrm{mod}\ q) \\
4 - 1 = 3 & &
\end{array}$$

In general, we want the smallest possible difference cover for a given q.

- For any q, there exist a difference cover D_q of size $\mathcal{O}(\sqrt{q})$.
- The DC3 algorithm uses the simplest non-trivial difference cover $D_3 = \{1, 2\}$.

A **difference cover sample** is a set T_C of suffixes, where

$$C = \{i \in [0..n] \mid (i \bmod q) \in D_q\} .$$

Example 4.16: If $T = \texttt{banana\$}$ and $D_3 = \{1, 2\}$,
then $C = \{1, 2, 4, 5\}$ and $T_C = \{\texttt{anana\$}, \texttt{nana\$}, \texttt{na\$}, \texttt{a\$}\}$.

Once we have sorted the difference cover sample T_C, we can compare any two suffixes in $\mathcal{O}(q)$ time. To compare suffixes T_i and T_j:

- If $i \in C$ and $j \in C$, then we already know their order from T_C.
- Otherwise, find ℓ such that $i + \ell \in C$ and $j + \ell \in C$. There always exists such $\ell \in [0..q)$. Then compare:

$$T_i = T[i..i + \ell)T_{i+\ell}$$
$$T_j = T[j..j + \ell)T_{j+\ell}$$

  That is, compare first $T[i..i + \ell)$ to $T[j..j + \ell)$, and if they are the same, then $T_{i+\ell}$ to $T_{j+\ell}$ using the sorted T_C.

Example 4.17: $D_3 = \{1, 2\}$ and $C = \{1, 2, 4, 5, \ldots\}$

$T_0 = T[0]T_1$	$T_0 = T[0]T[1]T_2$	$T_0 = T[0]T_1$
$T_1 = T[1]T_2$	$T_2 = T[2]T[3]T_4$	$T_3 = T[3]T_4$

Algorithm 4.18: DC3

Step 0: Choose C.

- Use difference cover $D_3 = \{1,2\}$.
- For $k \in \{0,1,2\}$, define $C_k = \{i \in [0..n] \mid i \bmod 3 = k\}$.
- Let $C = C_1 \cup C_2$ and $\bar{C} = C_0$.

Example 4.19:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	y	a	b	b	a	d	a	b	b	a	d	o	$

$\bar{C} = C_0 = \{0,3,6,9,12\}$, $C_1 = \{1,4,7,10\}$, $C_2 = \{2,5,8,11\}$ and
$C = \{1,2,4,5,7,8,10,11\}$.

Step 1: Sort T_C.

- For $k \in \{1,2\}$, Construct the strings $R_k = (T^3_k, T^3_{k+3}, T^3_{k+6}, \ldots, T^3_{\max C_k})$ whose characters are 3-factors of the text, and let $R = R_1 R_2$.
- Replace each factor T^3_i in R with an order preserving name $N^3_i \in [1..|R|]$. The names can be computed by sorting the factors with LSD radix sort in $\mathcal{O}(n)$ time. Let R' be the result appended with 0.
- Construct the inverse suffix array $SA^{-1}_{R'}$ of R'. This is done recursively using DC3 unless all symbols in R' are unique, in which case $SA^{-1}_{R'} = R'$.
- From $SA^{-1}_{R'}$, we get order preserving names for suffixes in T_C. For $i \in C$, let $N_i = SA^{-1}_{R'}[j]$, where j is the position of T^3_i in R. For $i \in \bar{C}$, let $N_i = \perp$. Also let $N_{n+1} = N_{n+2} = 0$.

Example 4.20:

R	abb	ada	bba	do\$	bba	dab	bad	o\$	
R'	1	2	4	7	4	6	3	8	0
$SA^{-1}_{R'}$	1	2	5	7	4	6	3	8	0

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$T[i]$	y	a	b	b	a	d	a	b	b	a	d	o	\$		
N_i	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$	0	0

Step 2(a): Sort $T_{\bar{C}}$.

- For each $i \in \bar{C}$, we represent T_i with the pair $(T[i], N_{i+1})$. Then
$$T_i \le T_j \iff (T[i], N_{i+1}) \le (T[j], N_{j+1}) \ .$$
Note that $N_{i+1} \neq \perp$ for all $i \in \bar{C}$.

- The pairs $(T[i], N_{i+1})$ are sorted by LSD radix sort in $\mathcal{O}(n)$ time.

Example 4.21:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	y	a	b	b	a	d	a	b	b	a	d	o	$
N_i	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$

$T_{12} < T_6 < T_9 < T_3 < T_0$ because $(\$, 0) < (a, 5) < (a, 7) < (b, 2) < (y, 1)$.

Step 2(b): Merge T_C and $T_{\bar{C}}$.

- Use comparison based merging algorithm needing $\mathcal{O}(n)$ comparisons.
- To compare $T_i \in T_C$ and $T_j \in T_{\bar{C}}$, we have two cases:

$$\begin{aligned} i \in C_1 &: T_i \le T_j \iff (T[i], N_{i+1}) \le (T[j], N_{j+1}) \\ i \in C_2 &: T_i \le T_j \iff (T[i], T[i+1], N_{i+2}) \le (T[j], T[j+1], N_{j+2}) \end{aligned}$$

Note that none of the N-values is $\perp$.

Example 4.22:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$T[i]$	y	a	b	b	a	d	a	b	b	a	d	o	$
N_i	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$

$T_1 < T_6$ because $(\mathtt{a}, 4) < (\mathtt{a}, 5)$.
$T_3 < T_8$ because $(\mathtt{b}, \mathtt{a}, 6) < (\mathtt{b}, \mathtt{a}, 7)$.

Theorem 4.23: Algorithm DC3 constructs the suffix array of a string $T[0..n)$ in $\mathcal{O}(n)$ time plus the time needed to sort the characters of T.

There are many variants:

- DC3 is an optimal algorithm under several parallel and external memory computation models, too. There exists both parallel and external memory implementations of DC3.

- Using a larger value of q, we obtain more space efficient algorithms. For example, using $q = \log n$, the time complexity is $\mathcal{O}(n \log n)$ and the space needed in addition to the text and the suffix array is $\mathcal{O}(n/\sqrt{\log n})$.

Induced Sorting

Define three type of suffixes $-$, $+$ and $*$ as follows:

$$C^- = \{i \in [0..n) \mid T_i > T_{i+1}\}$$
$$C^+ = \{i \in [0..n) \mid T_i < T_{i+1}\}$$
$$C^* = \{i \in C^+ \mid i-1 \in C^-\}$$

Example 4.24:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$T[i]$	m	m	i	s	s	i	s	s	i	i	p	p	i	i	\$
type of T_i	$-$	$-$	$*$	$-$	$-$	$*$	$-$	$-$	$*$	$+$	$-$	$-$	$-$	$-$	

For every $a \in \Sigma$ and $x \in \{-,+.*\}$ define

$$C_a = \{i \in [0..n] \mid T[i] = a\}$$
$$C_a^x = C_a \cap C^x$$

Then

$$C_a^- = \{i \in C_a \mid T_i < a^\infty\}$$
$$C_a^+ = \{i \in C_a \mid T_i > a^\infty\}$$

and thus, if $i \in C_a^-$ and $j \in C_a^+$, then $T_i < T_j$. Hence the suffix array is $nC_1C_2 \ldots C_{\sigma-1} = nC_1^-C_1^+C_2^-C_2^+ \ldots C_{\sigma-1}^-C_{\sigma-1}^+$.

The basic idea of induced sorting is to use information about the order of T_i to **induce** the order of the suffix $T_{i-1} = T[i-1]T_i$. The main steps are:

1. Sort the sets C^*_a, $a \in [1..\sigma)$.
2. Use C^*_a, $a \in [1..\sigma)$, to induce the order of the sets C^-_a, $a \in [1..\sigma)$.
3. Use C^-_a, $a \in [1..\sigma)$, to induce the order of the sets C^+_a, $a \in [1..\sigma)$.

The suffixes involved in the induction steps can be indentified using the following rules (proof is left as an exercise).

Lemma 4.25: For all $a \in [1..\sigma)$

(a) $i - 1 \in C^-_a$ iff $i > 0$ and $T[i-1] = a$ and one of the following holds

1. $i = n$
2. $i \in C^*$
3. $i \in C^-$ and $T[i-1] \geq T[i]$.

(b) $i - 1 \in C^+_a$ iff $i > 0$ and $T[i-1] = a$ and one of the following holds

1. $i \in C^-$ and $T[i-1] < T[i]$
2. $i \in C^+$ and $T[i-1] \leq T[i]$.

To induce C^- suffixes:

1. Set C_a^- empty for all $a \in [1..\sigma)$.
2. For all suffixes T_i such that $i-1 \in C^-$ **in lexicographical order**, append $i-1$ into $C^-_{T[i-1]}$.

By Lemma 4.25(a), Step 2 can be done by checking the relevant conditions for all $i \in nC_1^- C_1^* C_2^- C_2^* \ldots$.

Algorithm 4.26: InduceMinusSuffixes
Input: Lexicographically sorted lists C_a^*, $a \in \Sigma$
Output: Lexicographically sorted lists C_a^-, $a \in \Sigma$

```
(1) for $a \in \Sigma$ do $C_a^- \leftarrow \emptyset$
(2) pushback($n-1, C^-_{T[n-1]}$)
(3) for $a \leftarrow 1$ to $\sigma - 1$ do
(4)     for $i \in C_a^-$ do    // include elements added during the loop
(5)         if $i > 0$ and $T[i-1] \geq a$ then pushback($i-1, C^-_{T[i-1]}$)
(6)     for $i \in C_a^*$ do pushback($i-1, C^-_{T[i-1]}$)
```

Note that since $T_{i-1} > T_i$ by definition of C^-, we always have i inserted before $i-1$.

Inducing $+$-type suffixes goes similarly but in reverse order so that again i is always inserted before $i-1$:

1. Set C_a^+ empty for all $a \in [1..\sigma)$.
2. For all suffixes T_i such that $i-1 \in C^+$ in **descending** lexicographical order, append $i-1$ into $C^+_{T[i-1]}$.

Algorithm 4.27: InducePlusSuffixes
Input: Lexicographically sorted lists C_a^-, $a \in \Sigma$
Output: Lexicographically sorted lists C_a^+, $a \in \Sigma$

```
(1)  for $a \in \Sigma$ do $C_a^+ \leftarrow \emptyset$
(2)  for $a \leftarrow \sigma - 1$ downto 1 do
(3)      for $i \in C_a^+$ in reverse order do // include elements added during loop
(4)          if $i > 0$ and $T[i-1] \leq a$ then pushfront($i-1, C^+_{T[i-1]}$)
(5)      for $i \in C_a^-$ in reverse order do
(6)          if $i > 0$ and $T[i-1] < a$ then pushfront($i-1, C^+_{T[i-1]}$)
```

We still need to explain how to sort the $*$-type suffixes. Define

$$\begin{aligned} F[i] &= \min\{k \in [i+1..n] \mid k \in C^* \text{ or } k = n\} \\ S_i &= T[i..F[i]] \\ S'_i &= S_i\sigma \end{aligned}$$

where σ is a special symbol larger than any other symbol.

Lemma 4.28: For any $i, j \in [0..n)$, $T_i < T_j$ iff $S'_i < S'_j$ or $S'_i = S'_j$ and $T_{F[i]} < T_{F[j]}$.

Proof. The claim is trivially true except in the case that S_j is a proper prefix of S_i (or vice versa). In that case, $S_i > S_j$ but $S'_i < S'_j$ and thus $T_i < T_j$ by the claim. We will show that this is correct.

Let $\ell = F[j]$ and $k = i + \ell - j$. Then

- $\ell \in C^*$ and thus $\ell - 1 \in C^-$. By Lemma 4.25(b), $T[\ell - 1] > T[\ell]$.
- $T[k-1..k] = T[\ell-1..\ell]$ and thus $T[k-1] > T[k]$. If we had $k \in C^+$, we would have $k \in C^*$. Since this is not the case, we must have $k \in C^-$.
- Let $a = T[\ell]$. Since $\ell \in C^+_a$ and $k \in C^-_a$, we must have $T_k < a^\infty < T_\ell$.
- Since $T[i..k) = T[j..\ell)$ and $T_k < T_\ell$, we have $T_i < T_j$.

$\square$

Algorithm 4.29: SAIS

Step 0: Choose C.

- Compute the types of suffixes. This can be done in $\mathcal{O}(n)$ time based on Lemma 4.25.
- Set $C = \cup_{a\in[1..\sigma)} C_a^* \cup \{n\}$. Note that $|C| \leq n/2$, since for all $i \in C$, $i - 1 \in C^- \subseteq \bar{C}$.

Example 4.30:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$T[i]$	m	m	i	s	s	i	s	s	i	i	p	p	i	i	$
type of T_i	$-$	$-$	$*$	$-$	$-$	$*$	$-$	$-$	$*$	$+$	$-$	$-$	$-$	$-$	

$C_{\mathtt{i}}^* = \{2, 5, 8\}$, $C_{\mathtt{m}}^* = C_{\mathtt{p}}^* = C_{\mathtt{s}}^* = \emptyset$, $C = \{2, 5, 8, 14\}$.

Step 1: Sort T_C.

- Sort the strings S'_i, $i \in C^*$. Since the total length of the strings S'_i is $\mathcal{O}(n)$, the sorting can be done in $\mathcal{O}(n)$ time using LSD radix sort.
- Assign order preserving names $N_i \in [1..|C|-1]$ to the string S'_i so that $N_i \leq N_j$ iff $S'_i \leq S'_j$.
- Construct the sequence $R = N_{i_1} N_{i_2} \ldots N_k 0$, where $i_1 < i_3 < \cdots < i_k$ are the *-type positions.
- Construct the suffix array SA_R of R. This is done recursively unless all symbols in R are unique, in which case a simple counting sort is sufficient.
- The order of the suffixes of R corresponds to the order of $*$-type suffixes of T. Thus we can construct the lexicographically ordered lists C^*_a, $a \in [1..\sigma)$.

Example 4.31:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$T[i]$	m	m	i	s	s	i	s	s	i	i	p	p	i	i	$
N_i			2			2			1						0

$R = [\texttt{issi}\sigma][\texttt{issi}\sigma][\texttt{iippii\$}\sigma]\$ = 2210$, $SA_R = (3,2,1,0)$, $C^*_{\texttt{i}} = (8,5,2)$

Step 2: Sort $T_{[0..n]}$.

- Run InduceMinusSuffixes to construct the sorted lists C_a^-, $a \in [1..\sigma)$.
- Run InducePlusSuffixes to construct the sorted lists C_a^+, $a \in [1..\sigma)$.
- The suffix array is $SA = nC_1^- C_1^+ C_2^- C_2^+ \ldots C_{\sigma-1}^- C_{\sigma-1}^+$.

Example 4.32:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$T[i]$	m	m	i	s	s	i	s	s	i	i	p	p	i	i	$
type of T_i	$-$	$-$	$*$	$-$	$-$	$*$	$-$	$-$	$*$	$+$	$-$	$-$	$-$	$-$	

$n = 14 \quad \Rightarrow \quad C_{\mathtt{i}}^- = (13, 12)$

$C_{\mathtt{i}}^- C_{\mathtt{i}}^* = (13, 12, 8, 5, 2) \quad \Rightarrow \quad C_{\mathtt{m}}^- = (1, 0),\ C_{\mathtt{p}}^- = (11, 10),\ C_{\mathtt{s}}^- = (7, 4, 6, 3)$

$\Rightarrow \quad C_{\mathtt{i}}^+ = (8, 9, 5, 2)$

$\Rightarrow \quad SA = C_{\$} C_{\mathtt{i}}^- C_{\mathtt{i}}^+ C_{\mathtt{m}}^- C_{\mathtt{p}}^- C_{\mathtt{s}}^- = (14, 13, 12, 8, 9, 5, 2, 1, 0, 11, 10, 7, 4, 6, 3)$

Theorem 4.33: Algorithm SAIS constructs the suffix array of a string $T[0..n)$ in $\mathcal{O}(n)$ time plus the time needed to sort the characters of T.

- In Step 1, to sort the strings S'_i, $i \in C^*$, SAIS does not actually use LSD radix sort but the following procedure:

  1. Construct the sets C^*_a, $a \in [1..\sigma)$ **in arbitrary order**.

  2. Run InduceMinusSuffixes to construct the lists C^-_a, $a \in [1..\sigma)$.

  3. Run InducePlusSuffixes to construct the lists C^-_a, $a \in [1..\sigma)$.

  4. Remove non-$*$-type positions from $C^+_1 C^+_2 \ldots C^+_{\sigma-1}$.

  With this change, most of the work is done in the induction procedures. This is very fast in practice, because all the lists C^x_a are accessed **sequentially** during the procedures.

- The currently fastest suffix sorting implementation in practice is probably divsufsort by Yuta Mori. It sorts the $*$-type suffixes non-recursively in $\mathcal{O}(n \log n)$ time and then continues as SAIS.

Summary: Suffix Trees and Arrays

The most important data structures for string processing:

- Designed for indexed exact string matching.
- Used in efficient solutions to a huge variety of different problems.

Construction algorithms are among the most important algorithms for string processing:

- Linear time for constant and integer alphabet.

Often augmented with additional data structures:

- suffix links, LCA preprocessing
- LCP array, RMQ preprocessing, BWT, ...