58093 String Processing Algorithms

Lectures, Fall 2010, period II

Juha Kärkkäinen

Who is this course for?

- Master's level course in Computer Science, 4 cr
- Subprogram of Algorithms and Machine Learning
  - Together with Project in String Processing Algorithms (in the next teaching period) one of the three special course combinations, one of which must be included in the Master's degree.
- Suitable addition to Master's degree program for Bioinformatics, particularly for those interested in biological sequence analysis
- Prerequisites: basic knowledge on algorithms, data structures, finite automata, algorithm analysis.
  - Recommended prerequisite courses: Data Structures,Models of Computation, Designs and Analysis of Algorithms

Course components

Lectures: Tue and Thu 12–14 in B222

- Lecture notes will be available on the course home page. Attending lectures is not obligatory but can be useful. There will be additional examples and explanations.
- The lectures do not follow any book, but the books mentioned on the home page can provide useful additional material.

Exercises: Thu 14-16 in B119 and 16-18 in BK107 (starting 11.11.)

- Exercise problems will be available on the course home page a week before the exercise session. Model solutions will be available on the home page after the exercise session.
- You should solve the problems at home and be prepared to present your solutions at the exercise session. You do not need to solve all problems but additional points are awarded according to how many problems you have solved (see later).

Course components (continued)

Exam: Thu 16.12. at 16–19 (check time and place later)

- The exam consists of problems similar to those in the exercises. No notes or other material is allowed in the exam.
- If you fail, there is a renewal exam (exercise points count) and separate exams (no exercise points).

In the next period, there is a follow up course Project in String Processing Algorithms (2 cr).

- Consist of implementing some algorithms on this course, experimenting with them, and presenting the results.

Grading

Exercises 1–10 points

- Scoring is based on the number of exercises solved.
- About 20% is *required* and gives 1 point.
- About 80% gives the maximum of 10 points.

Exam 25–50 points

- 25 points is *required*

Total 30–60 points

- 30 points is required to pass and gives the lowest grade 1.
- 50 points gives the highest grade 5.

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

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 problem:

- 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 two types of alphabets:

- *Ordered alphabet* $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$, where $c_1 < c_2 < \cdots < c_\sigma$.
- *Integer alphabet* $\Sigma = \{0, 1, 2, \ldots, \sigma - 1\}$.

The alphabet types are really used for *classifying algorithms* rather than alphabets:

- Algorithms for ordered alphabets use only character comparisons.
- Algorithms for integer alphabets can use operations such as using a symbol as an address to a table.

Algorithms for integer alphabets are more restricted in their applicability.

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:

$$\begin{aligned}S &= S[1..n] = S[1]S[2]\ldots S[n] \\ T &= T[0..n) = T[0]T[1]\ldots T[n-1]\end{aligned}$$

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

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

- $\alpha, \beta \in \Sigma^*$
- $x = a_1 a_2 \ldots a_k$ where $a_i \in \Sigma$ for all i
- $w = uv$, $u, v \in \Sigma^*$ (w is the *concatenation* of u and v)

Individual characters or their positions usually do not matter. The significant entities are the substrings or factor.

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 $w = \mathtt{bonobo}$. Then

- $\varepsilon, \mathtt{b}, \mathtt{bo}, \mathtt{bon}, \mathtt{bono}, \mathtt{bonob}, \mathtt{bonobo}$ are the prefixes of w
- $\varepsilon, \mathtt{o}, \mathtt{bo}, \mathtt{obo}, \mathtt{nobo}, \mathtt{onobo}, \mathtt{bonobo}$ are the suffixes of w
- $\varepsilon, \mathtt{bo}, \mathtt{bonobo}$ are the borders of w.
- $\varepsilon, \mathtt{b}, \mathtt{o}, \mathtt{n}, \mathtt{bo}, \mathtt{on}, \mathtt{no}, \mathtt{ob}, \mathtt{bon}, \mathtt{ono}, \mathtt{nob}, \mathtt{obo}, \mathtt{bono}, \mathtt{onob}, \mathtt{nobo}, \mathtt{bonob}, \mathtt{onobo}, \mathtt{bonobo}$ 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.

1. 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 1.1: 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.

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

The naive, brute force algorithm compares P against $T[0..m)$, then against $T[1..m+1)$, then against $T[2..m+2)$ etc. until an occurrence is found or the end of the text is reached.

Algorithm 1.2: Brute force
Input: text $T = T[0 \dots n)$, pattern $P = P[0 \dots m)$
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* output $j-m$ *else* output 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 moves to the next text position.

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 1.3:

Brute force	Morris–Pratt	Knuth–Morris–Pratt
ainaisesti-ainainen	ainaisesti-ainainen	ainaisesti-ainainen
ainai~~n~~en (6 comp.)	ainai~~n~~en (6)	ainai~~n~~en (6)
~~a~~inainen (1)	ai~~n~~ainen (1)	~~a~~inainen (1)
~~a~~inainen (1)	~~a~~inainen (1)	
ai~~n~~ainen (3)		
~~a~~inainen (1)		
~~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 1.4: Knuth–Morris–Pratt / Morris–Pratt
Input: text $T = T[0 \dots n)$, pattern $P = P[0 \dots m)$
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* output $j-m$ *else* output 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 $P[0..i)$.

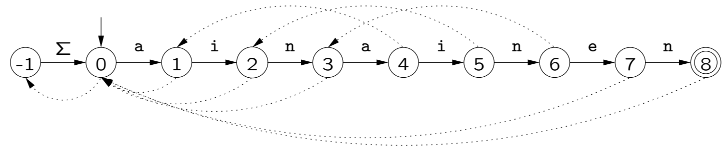
Example: `ai` is the longest proper border of `ainai`.

- 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 1.5: 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 1.6: Morris–Pratt failure function computation
Input: pattern $P = P[0\ldots m)$
Output: array $fail[0..m]$ for P
(1) $i \leftarrow -1; j \leftarrow 0;\ fail[j] = i$
(2) *while* $j < m$ *do*
(3)   *if* $i = -1$ *or* $P[i] = P[j]$ *then* $i \leftarrow i+1; j \leftarrow j+1;\ fail[j] = i$
(4)   *else* $i \leftarrow fail[i]$
(5) output $fail$

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

Theorem 1.7: Algorithms MP and KMP preprocess a pattern in time $\mathcal{O}(m)$ and then search the text in time $\mathcal{O}(n)$.

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

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

When the (K)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. $D \leftarrow (D << 1) + 1$ (the bitwise *shift*). 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 32 or 64. Assume first that $m \le w$. Then each bitvector can be stored in a single integer.

Algorithm 1.8: Shift-And
Input: text $T = T[0 \dots n)$, pattern $P = P[0 \dots m)$
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) $D \leftarrow ((D << 1) + 1) \,\&\, B[T[j]]$
(6) *if* $D \,\&\, 2^{m-1} \neq 0$ *then* return $j - m + 1$ // $D.(m-1) = 1$

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 & on line 5 is replaced by | (bitwise *or*). The advantage is that we don't need that "+1" on line 5.

Example 1.9: $P =$ `assi`, $T =$ `apassi`, bitvectors are columns.

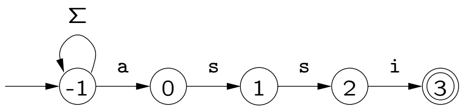
$B[c]$, $c \in \{$`a`,`i`,`p`,`s`$\}$

	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 $m \le w$:

- 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*.

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 text character aligned with the last pattern character. If it doesn't match, move (shift) the pattern forward until there is a match.

Example 1.10: 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 we are past $T[k]$.
- If $P[m-1] = T[k]$, compare the rest in brute force manner. Then shift to the next position, where $T[k]$ matches.

Algorithm 1.11: 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

The length of the shift is determined by the *shift table*. $shift[c]$ 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 1.12: $P =$ `ainainen`.

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

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 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 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 1.13: Horspool shift

`varmasti-aikai`~~`s`~~`en-ainainen`
`ainaisen-ainainen`
`  ainaisen-ainainen`

Boyer–Moore shift

`varmasti-aikai`~~`s`~~`en-ainainen`
`ainaisen-ainainen`
`        ainaisen-ainainen`

Factor shift

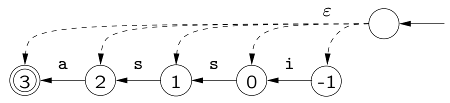
`varmasti-ai`~~`k`~~`aisen-ainainen`
`ainaisen-ainainen`
`               ainaisen-ainainen`

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 1.14: $P = \texttt{assi}$.



- 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 $T[j+i..j+m)$:

- $D.i = 1$ if and only if there is a path from the initial state to state i with the string $(T[j+i..j+m))^R$.
- 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, so the shift brings the right bit to $D.0$. With Shift-And $D.(-1) = 1$ always.
- The exception is that in the beginning before reading anything $D.(-1) = 1$. This is handled by doing the shift at the end of the loop.

Algorithm 1.15: 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* $B[c] := 0$
(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) $\quad i \leftarrow m$; $shift \leftarrow m$
(6) $\quad D \leftarrow 2^m - 1$ $\qquad$ // $D \leftarrow 1^m$
(7) $\quad$ *while* $D \neq 0$ *do*
$\qquad$ // Now $T[j+i..j+m)$ is a pattern factor
(8) $\qquad i \leftarrow i - 1$
(9) $\qquad D \leftarrow D \ \& \ B[T[j+i]]$
(10) $\qquad$ *if* $D \ \& \ 2^{m-1} \neq 0$ *then*
$\qquad\quad$ // Now $T[j+i..j+m)$ is a pattern prefix
(11) $\qquad\quad$ *if* $i = 0$ *then* return j
(12) $\qquad\quad$ *else* $shift \leftarrow i$
(13) $\qquad D \leftarrow D << 1$
(14) $\quad j \leftarrow j + shift$

Example 1.16: $P =$ `assi`, $T =$ `apassi`.

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

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

D when scanning `apas` backwards

	a	p	a	s	
i	0	0	0	1	
s	0	0	1	1	
s	0	0	1	1	
a	0	<u>1</u>	0	1	$\Rightarrow$ $shift = 2$

D when scanning `assi` backwards

		a	s	s	i	
i	0	0	0	1	1	
s	0	0	1	0	1	
s	0	1	0	0	1	
a	<u>1</u>	0	0	0	1	$\Rightarrow$ 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*.
- There are also algorithms that are optimal in *both* cases. They are based on similar techniques, but they will not be described them here.

Karp–Rabin

The Karp–Rabin algorithm uses a *hash function* $H : \Sigma^* \to [0..q) \subset \mathbb{N}$ for strings. It computes $H(P)$ 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 *collision*.

A good hash function has two important properties:

- Collisions are rare.
- Given $H(a\alpha)$, a and b, where $a, b \in \Sigma$ and $\alpha \in \Sigma^*$, we can quickly compute $H(\alpha b)$. This is a called *rolling* or *sliding window* hash function.

The latter property is essential for fast computation of $H(T[j..j+m))$ for all j.

The hash function used by Karp–Rabin is

$$H(c_0c_1c_2 \dots c_{m-1}) = (c_0r^{m-1} + c_1r^{m-2} + \cdots + c_{m-2}r + c_{m-1}) \bmod q$$

This is a rolling hash function:

$$H(\alpha) = (H(a\alpha) - ar^{m-1}) \bmod q$$
$$H(\alpha b) = (H(\alpha) \cdot r + b) \bmod q$$

which follows from the rules of *modulo arithmetic*:

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

The parameters q and r have to be chosen with some care to ensure that collisions are rare.

- The original choice is $r = \sigma$ and q is a large *prime*.
- Another possibility is $q = 2^w$, where w is the machine word size, and r is a small prime ($r = 37$ has been suggested). This is faster in practice, because it avoids slow modulo operations.
- The hash function can be *randomized* by choosing q or r randomly. Furthermore, we can change q or r whenever a collision happens.

Algorithm 1.17: Karp-Rabin

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

```
Choose q and r; s ← r^(m−1) mod q
hp ← 0; ht ← 0
for i ← 0 to m − 1 do hp ← (hp · r + P[i]) mod q      // hp = H(P)
for j ← 0 to m − 1 do ht ← (ht · r + T[j]) mod q
for j ← 0 to n − m − 1 do
    if hp = ht then if P = T[j . . . j + m) then return j
    ht ← ((ht − T[j] · s) · r + T[j + m]) mod q
if hp = ht then if P = T[j . . . j + m) then return j
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, but hashing can be a useful technique in other contexts.

2. 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 2.1: Let $A =$ `Lewensteinn` and $B =$ `Levenshtein`. Then $ed(A, B) = 3$.

The set of edit operations can be described with an *edit sequence* or with an *alignment*:

```
NNSNNNINNNND
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.

Computing Edit Distance

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

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

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

Example 2.3: $A =$ `ballad`, $B =$ `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 2.2. 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 $p + q < i + j$.

The value $ed(A_i, B_j)$ is based on an *optimal edit sequence*. We have three cases depending on what the last edit operation is:

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

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

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

Since the edit sequence is optimal, the correct value is the minimum of the three cases, which agrees with the definition of d_{ij}. $\square$

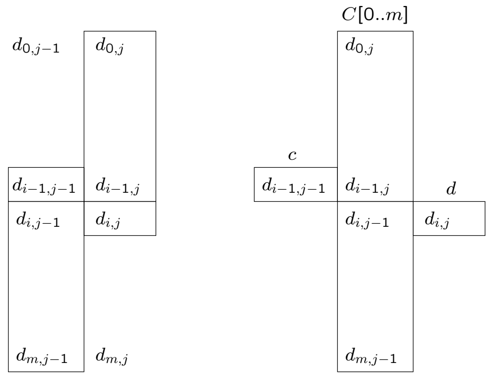
The recurrence gives directly a *dynamic programming* algorithm for computing the edit distance.

Algorithm 2.4: Edit distance
Input: strings $A[1..m]$ and $B[1..n]$
Output: $ed(A, B)$
for $i \leftarrow 0$ *to* m *do* $d_{i0} \leftarrow i$
for $j \leftarrow 1$ *to* n *do* $d_{0j} \leftarrow j$
for $j \leftarrow 1$ *to* n *do*
    for $i \leftarrow 1$ *to* m *do*
        $d_{ij} \leftarrow \min\{d_{i-1,j-1} + \delta(A[i], B[j]), d_{i-1,j} + 1, d_{i,j-1} + 1\}$
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 2.5: Edit distance in $\mathcal{O}(m)$ space
Input: strings $A[1..m]$ and $B[1..n]$
Output: $ed(A, B)$
for $i \leftarrow 0$ *to* m *do* $C[i] \leftarrow i$
for $j \leftarrow 1$ *to* n *do*
    $c \leftarrow C[0]$; $C[0] \leftarrow j$
    for $i \leftarrow 1$ *to* m *do*
        $d \leftarrow \min\{c + \delta(A[i], B[j]), C[i-1] + 1, C[i] + 1\}$
        $c \leftarrow C[i]$
        $C[i] \leftarrow d$
return $C[m]$

- Note that because $ed(A, B) = ed(B, A)$ (exercise), we can assume that $m \le n$.

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

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

Example 2.6: $A = \texttt{ballad}, B = \texttt{handball}$

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

There are 7 paths from $(0, 0)$ to $(6, 8)$ corresponding to, for example, the following edit sequences and alignments:

```
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 2.7: 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*:

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

Theorem 2.8: For all $0 \leq i \leq m$, $0 \leq j \leq n$:

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

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

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

Basis:

$g_{00} = 0 = ed(\epsilon, \epsilon)$

$g_{0j} = 0 = ed(\epsilon, \epsilon) = ed(\epsilon, T(j - 0..j])$ (min at $\ell = 0$)

$g_{i0} = i = ed(P[1..i], \epsilon) = ed(P[1..i], T(0 - 0..0])$ ($0 \leq \ell \leq j = 0$)

Induction step: Essentially the same as in the proof of Theorem 2.2.

Example 2.9: $P = \texttt{match}$, $T = \texttt{remachine}$, $k = 1$

g		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

One occurrence ending at position 6.

Algorithm 2.10: 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 $\mathcal{O}(mn)$.
- The space complexity can be reduced to $\mathcal{O}(m)$ by storing only one column as in Algorithm 2.5.

Ukkonen's Cut-off Heuristic

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

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

Example 2.11: Diagonals -3 and 2.

g		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

More specifically, we have the following property.

Lemma 2.12: 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, $g_{ij} \le g_{i-1,j-1} + \delta(P[i], T[j]) \le g_{i-1,j-1} + 1$. We show that $g_{ij} \ge g_{i-1,j-1}$ by induction on $i + j$.

The induction assumption is that $g_{pq} \ge g_{p-1,q-1}$ 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 $g_{ij} \ge g_{i-1,j-1}$.
2. $g_{ij} = g_{i-1,j} + 1$ and $i > 1$. Then
$$g_{ij} = g_{i-1,j} + 1 \overset{\text{ind. assump.}}{\ge} g_{i-2,j-1} + 1 \overset{\text{definition}}{\ge} g_{i-1,j-1}$$
3. $g_{ij} = g_{i,j-1} + 1$ and $j > 1$. Then
$$g_{ij} = g_{i,j-1} + 1 \overset{\text{ind. assump.}}{\ge} g_{i-1,j-2} + 1 \overset{\text{definition}}{\ge} g_{i-1,j-1}$$
4. $i = 1$. Then $g_{ij} \ge 0 = g_{i-1,j-1}$.

$g_{ij} = g_{i,j-1} + 1$ and $j = 1$ is not possible because $g_{i,1} \le g_{i-1,0} + 1 < g_{i,0} + 1$. $\square$

We can reduce computation using diagonal monotonicity:

- Whenever the value on a column 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.

Example 2.13: $P =$ `match`, $T =$ `remachine`, $k = 1$

g		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					1	1	2	3		
c						1	2	3		
h							1	2		

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

Algorithm 2.14: 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

for $i \leftarrow 0$ *to* m *do* $g_{i0} \leftarrow i$
for $j \leftarrow 1$ *to* n *do* $g_{0j} \leftarrow 0$
$top \leftarrow \min(k+1, m)$
for $j \leftarrow 1$ *to* n *do*
    for $i \leftarrow 1$ *to* top *do*
        $g_{ij} \leftarrow \min\{g_{i-1,j-1} + \delta(A[i], B[j]), g_{i-1,j} + 1, g_{i,j-1} + 1\}$
    while $g_{top,j} > k$ *do* $top \leftarrow top - 1$
    if $top = m$ *then* output j
    else $top \leftarrow top + 1$

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

- The worst case time complexity is still $\mathcal{O}(mn)$.
- 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: $\Delta d_{ij} = g_{ij} - g_{i-1,j}$

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

The proof is left as an exercise.

Example 2.16: '−' 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 $\Delta v^{\text{in}} = \Delta v_{i,j-1}$, $\Delta h^{\text{in}} = \Delta h_{i,j-1}$ 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 computation rule is defined by the following result.

Lemma 2.17: If $Eq = 1$ or $\Delta v^{\text{in}} = -1$ or $\Delta h^{\text{in}} = -1$, then $\Delta d = 0$, $\Delta v^{\text{out}} = -\Delta h^{\text{in}}$ and $\Delta h^{\text{out}} = -\Delta v^{\text{in}}$.
Otherwise $\Delta d = 1$, $\Delta v^{\text{out}} = 1 - \Delta h^{\text{in}}$ and $\Delta h^{\text{out}} = 1 - \Delta v^{\text{in}}$.

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

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

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

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

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 logical operations ($\wedge$, $\vee$, $\neg$).

Now the computation rules can be expressed as follows.

Lemma 2.18:

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

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 2.17,

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

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

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

In the last step, we used the fact that Mh^{in} and Ph^{in} cannot be 1 simultaneously. □

According to Lemma 2.18, 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 bit*parallel* 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

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

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

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

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

Proof. We use induction on i.

Basis $i = 1$: The right-hand side reduces to Eq_{1j}, because $\ell = 1$. By Lemma 2.18, $Xh_{1j} = Eq_{1j} \vee Mh_{0j}$, 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

$$
\begin{aligned}
&\exists \ell \in [1,i] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-1] : Pv_{x,j-1}) \\
&= Eq_{ij} \vee \exists \ell \in [1,i-1] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-1] : Pv_{x,j-1}) \\
&= Eq_{ij} \vee (Pv_{i-1,j-1} \wedge \exists \ell \in [1,i-1] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-2] : Pv_{x,j-1})) \\
&= Eq_{ij} \vee (Pv_{i-1,j-1} \wedge Xh_{i-1,j}) \qquad \text{(ind. assump.)} \\
&= Eq_{ij} \vee Mh_{i-1,j} \qquad \text{(Lemma 2.18)} \\
&= Xh_{ij} \qquad \text{(Lemma 2.18)}
\end{aligned}
$$

□

At first sight, we cannot use Lemma 2.19 to compute even a single bit in constant time, not to mention 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 2.20: Denote $X = Xh_{*j}$, $E = Eq_{*j}$, $P = Pv_{*,j-1}$ ja olkoon $Y = (((E \wedge P) + P) \veebar P) \vee E$. Then $X = Y$.

Proof. By Lemma 2.19, $X[i] = 1$ iff and only if

a) $E[i] = 1$ or

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

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

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

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

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:

$$
\begin{array}{rl}
 & \,i \qquad \ell \\
E[\ell \ldots i] & = \mathtt{00...01} \\
P[\ell \ldots i] & = \mathtt{b1...11} \\
(E \wedge P)[\ell \ldots i] & = \mathtt{00...01} \\
((E \wedge P) + P)[\ell \ldots i] & = \bar{\mathtt{b}}\mathtt{0...0c} \\
(((E \wedge P) + P) \veebar P)[\ell \ldots i] & = \mathtt{11...1}\bar{\mathtt{c}} \\
Y = ((((E \wedge P) + P) \veebar P) \vee E)[\ell \ldots i] & = \mathtt{11...11}
\end{array}
$$

where $\mathtt{b}$ is the unknown bit $P[i]$, $\mathtt{c}$ is the possible carry bit coming from the summation of bits $1 \ldots, \ell - 1$, and $\bar{\mathtt{b}}$ and $\bar{\mathtt{c}}$ 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 $\bar{\mathtt{b}}$ to position i. The difference in this case is that at least one bit $P[k]$, $\ell \le k < i$, is zero, which stops the carry at position k. Thus $((E \wedge P) + P)[i] = \mathtt{b}$ and $Y[i] = 0$.

□

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

```
for c ∈ Σ do B[c] ← 0^m
for i ← 1 to m do B[P[i]][i] = 1
Pv ← 1^m; Mv ← 0; g ← m
for j ← 1 to n do
    Eq ← B[T[j]]
    Xh ← (((Eq ∧ Pv) + Pv) ⊻ Pv) ∨ Eq
    Ph ← Mv ∨ ¬(Xh ∨ Pv)
    Mh ← Pv ∧ Xh
    Xv ← Eq ∨ Mv
    Pv ← (Mh << 1) ∨ ¬(Xv ∨ (Ph << 1))
    Mv ← (Ph << 1) ∧ Xv
    g ← g + Ph[m] − Mh[m]
    if g ≤ k then output j
```


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

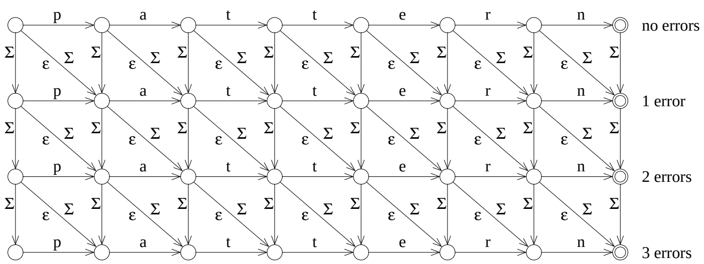
- Pattern preprocessing time is $\mathcal{O}(m + \sigma)$.
- Search time is $\mathcal{O}(n)$.

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 based on bitparallel simulation of a nondeterministic automaton.

Example 2.22: $P = \texttt{pattern}$, $k = 3$



- The algorithm of Wu and Manber uses a bit vector for each row. It can be seen as an extension of Shift-And. The search time complexity is $\mathcal{O}(kn\lceil m/w \rceil)$.
- The algorithm of Baeza-Yates and Navarro uses a bit vector for each diagonal, packed into one long bitvector. The search time complexity is $\mathcal{O}(n\lceil km/w \rceil)$.

Baeza-Yates–Perleberg Filtering Algorithm

A *filtering algorithm* for approximate strings 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*, and it is *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 2.23: Let $P_1 P_2 \ldots P_{k+1} = P$ 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. This is done in $\mathcal{O}(n)$ time using the Aho–Corasick algorithm for multiple exact string matching, which we will see later during this course.

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
$$\mathcal{O}\left(\frac{m^2(k+1)n}{\sigma^r}\right) \leq \mathcal{O}\left(\frac{m^3 n}{\sigma^r}\right) .$$
This is $\mathcal{O}(n)$ if $r \geq 3 \log_\sigma m$.

- The condition $r \geq 3 \log_\sigma m$ is satisfied when $(k+1) \leq m/(3 \log_\sigma m + 1)$.

Theorem 2.24: The average case time complexity of the Baeza-Yates–Perleberg algorithm is $\mathcal{O}(n)$ when $k \leq m/(3 \log_\sigma m + 1) - 1$.

Many variations of the algorithm have been suggested:

- The filtration can be done with a *different multiple exact string matching algorithm*:
  - The first algorithm of this type by Wu and Manber used an extension of the Shift-And algorithm.
  - An extension of BDM achieves $\mathcal{O}(nk(\log_\sigma m)/m)$ average case search time. This is sublinear for small enough k.
  - An extension of the Horspool algorithm is very fast in practice for small k and large σ.

- Using a technique called *hierarchical verification*, the average verification time for a single potential occurrence can be reduced to $\mathcal{O}((m/k)^2)$.

A filtering algorithm by Chang and Marr has average case time complexity $\mathcal{O}(n(k + \log_\sigma m)/m)$, which is optimal.

3. Sorting and Searching Sets of Strings

Sorting algorithms and *search trees* are among the most fundamental algorithms and data structures for sets of objects. They are also closely connected:

- A sorted array is an *implicit* search tree through *binary search*.
- A set can be sorted by inserting the elements in a search tree and then traversing the tree in order.

Another connecting feature is that they are usually based on *order comparisons* between elements, and their time complexity analysis counts the *number of comparisons* needed.

These algorithms and data structures work when the elements of the set are *strings* (pointers to strings, to be precise). However, comparisons are no more constant time operations in general, and the number of comparisons does not tell the whole truth about the time complexity.

In this part, we will see that algorithms and data structures *designed specifically for strings* are often faster.

Sorting Strings

Let us first define an order for strings formally.

Definition 3.1: Let $A[0..m)$ and $B[0..n)$ be two strings on an ordered alphabet Σ. We say that A is *lexicographically* smaller than B, denoted $A < B$, if and only if either

- $m < n$ and $A = B[0..m)$ (i.e., A is a proper prefix of B) or
- $A[0..i) = B[0..i)$ and $A[i] < B[i]$ for some $i \in [0..\min\{m, n\})$.

Determining the order of A and B needs $\Theta(\min\{m, n\})$ *symbol comparisons* in the worst case.

On the other hand, the *expected* number of symbol comparisons for two random strings is $\mathcal{O}(1)$.

$\Omega(n \log n)$ 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*.

What about the *average case* when sorting a set of *random strings*?

The following theorem shows that we cannot achieve $\mathcal{O}(n \log n)$ symbol comparisons in the average case (when $\sigma = n^{o(1)}$).

Theorem 3.2: 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 of Σ. Sorting $\mathcal{R}$ using $\mathcal{A}$ requires $\Omega(n \log n \log_\sigma n)$ symbol comparisons.

- If σ is considered to be a constant, the lower bound is $\Omega(n(\log n)^2)$.
- An intuitive explanation for this result is that the comparisons made by a sorting algorithm are *not random*.

Proof of Theorem 3.2. 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

$$\sum_{\alpha\in\Sigma^k} kn_\alpha \log n_\alpha \ge k(n-\sqrt{n}) \log \frac{n-\sqrt{n}}{\sigma^k} \ge k(n-\sqrt{n})\log(\sqrt{n}-1)$$
$$= \Omega(kn\log n) = \Omega(n \log n \log_\sigma n) \ .$$

Here we have used the facts that $\sigma^k \le \sqrt{n}$, that $\sum_{\alpha\in\Sigma^k} n_\alpha > n - \sigma^k \ge n - \sqrt{n}$, and that $\sum_{\alpha\in\Sigma^k} n_\alpha \log n_\alpha > (n-\sqrt{n})\log((n-\sqrt{n})/\sigma^k)$ (see exercises). □

The preceding lower bound does not hold for algorithms *specialized for sorting strings*. To derive a lower bound for any algorithm based on *symbol comparisons*, we need the following concept.

Definition 3.3: Let S be a string and $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ a set of strings. The *distinguishing prefix* of S in $\mathcal{R}$ is the shortest prefix of S that separates it from the (other) members $\mathcal{R}$. Let $dp_{\mathcal{R}}(S)$ denote the length of the distinguishing prefix, and let $DP(\mathcal{R}) = \sum_{T\in\mathcal{R}} dp_{\mathcal{R}}(T)$ be the total length of distuinguishing prefixes in $\mathcal{R}$.

Example 3.4: Distuingishing prefixes:

```
a akkoselliseen
järjesty kseen
järjestä
m erkkijonot
n ämä
```


Theorem 3.5: 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 $\Omega(DP(\mathcal{R}) + n\log n)$ symbol comparisons.

Proof. The algorithm must examine every symbol in the duistinguishing prefixes. This gives a lower bound $\Omega(DP(\mathcal{R}))$.

On the other hand, the general sorting lower bound $\Omega(n \log n)$ must hold here too.

The result follows from combining the two lower bounds. □

- Note that for a random set of strings $DP(\mathcal{R}) = \mathcal{O}(n \log_\sigma n)$ on average. 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 3.6: TernaryQuicksort(R)

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

if $|R| \le 1$ *then return* R
select a pivot $x \in R$
$R_< \leftarrow \{s \in R \mid s < x\}$
$R_= \leftarrow \{s \in R \mid s = x\}$
$R_> \leftarrow \{s \in R \mid s > x\}$
$R_< \leftarrow$ TernaryQuicksort($R_<$)
$R_> \leftarrow$ TernaryQuicksort($R_>$)
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 (exercise).

The time complexity of both the binary and the ternary quicksort depends on the *selection of the pivot*:

- The optimal choice is the *median* of R, which can be computed in linear worst case time. Then the time complexity is $\mathcal{O}(n \log n)$ in the worst case.
- If we choose the pivot *randomly*, the expected time complexity is $\mathcal{O}(n \log n)$.
- A practical choice is the median of a few elements.

In the following, we assume an optimal pivot selection.

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

Algorithm 3.7: StringQuicksort$(\mathcal{R}, \ell)$

Input: Set $\mathcal{R}$ of strings and the length ℓ of their common prefix.
Output: R in increasing 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 3.8: 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 3.9: String quicksort sorts a set $\mathcal{R}$ of n strings in $\mathcal{O}(DP(\mathcal{R}) + n \log n)$ time.

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

Proof of Theorem 3.9. 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.
- The algorithm never accesses symbols outside the distinguishing prefixes. Thus the total number of symbol comparisons resulting equality is at most $\mathcal{O}(DP(\mathcal{R}))$.

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

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

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 3.10: 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) $\quad tmp \leftarrow C[i]$; $C[i] \leftarrow sum$; $sum \leftarrow sum + tmp$
(6) *for* $i \leftarrow 1$ *to* n *do* // distribute
(7) $\quad J[C[k_i]] \leftarrow k_i$; $C[k_i] \leftarrow C[k_i] + 1$
(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 $\Omega(DP(\mathcal{R}) + n \log n)$ 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 (more on this in the exercises).

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 3.11: LSDRadixSort($\mathcal{R}$)
Input: Set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings of length m over the alphabet $[0..\sigma)$.
Output: $\mathcal{R}$ in increasing lexicographical order.
(1) *for* $\ell \leftarrow m - 1$ *to* 0 *do* CountingSort($\mathcal{R}$,ℓ)
(2) *return* $\mathcal{R}$

- CountingSort($\mathcal{R}$,ℓ) sorts the strings in $\mathcal{R}$ by the symbols at position ℓ using counting sort (with k_i is replaced by $S_i[\ell]$). The time complexity is $\mathcal{O}(|\mathcal{R}| + \sigma)$.
- The *stability* of counting sort is essential.

Example 3.12: $\mathcal{R} = \{\texttt{cat}, \texttt{him}, \texttt{ham}, \texttt{bat}\}$.

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

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

Theorem 3.13: 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 $\mathcal{R}$.

- The weakness of LSD radix sort is that it uses $\Omega(||\mathcal{R}||)$ time even when $DP(\mathcal{R})$ 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 3.14: 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 3.15: MSDRadixSort($\mathcal{R}, \ell$)
Input: Set $\mathcal{R} = \{S_1, S_2, \dots, S_n\}$ of strings over the alphabet $[0..\sigma)$
and the length ℓ of their common prefix.
Output: $\mathcal{R}$ in increasing lexicographical order.

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

- Here CountingSort($\mathcal{R}$,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 3.16: MSD radix sort sorts a set $\mathcal{R}$ of n strings over the alphabet $[0..\sigma)$ in $\mathcal{O}(DP(\mathcal{R}) + n \log \sigma)$ time.

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

- The time excluding the recursive call but including the call to counting sort is $\mathcal{O}(k + \sigma) = \mathcal{O}(k)$. The k symbols accessed here will not be accessed again.
- The algorithm does not access any symbols outside the distinguishing prefixes. Thus the total time spent in this kind of calls is $\mathcal{O}(DP(\mathcal{R}))$.

This still leaves the time spent in the calls to string quicksort. The calls are for sets of size smaller than σ and no string is included two calls. Therefore, the total time over all calls is $\mathcal{O}(DP(\mathcal{R}) + n \log \sigma)$.
$\square$

- There exists a more complicated variant of MSD radix sort with time complexity $\mathcal{O}(DP(\mathcal{R}) + \sigma)$.
- $\Omega(DP(\mathcal{R}))$ is a lower bound for any algorithm that must access symbols one at a time.
- In practice, MSD radix sort is very fast, but it is sensitive to implementation details.

String Mergesort

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

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

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

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

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

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

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

Lemma 3.17: Let A, B and C be strings.

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

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

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

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

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

$\square$

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

Lemma 3.18: Let $A \leq B, B' \leq C$ be strings.

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

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

Proof. We show (a); (b) is symmetric. Assume to the contrary that $B \geq B'$. Then by Lemma 3.17, $lcp(A,B) = \min\{lcp(A,B'), lcp(B',B)\} \leq lcp(A,B')$, which is a contradiction. $\square$

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

Algorithm 3.19: StringMergesort($\mathcal{R}$)
Input: Set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings.
Output: $\mathcal{R}$ sorted and augmented with lcp information.

if $|\mathcal{R}| = 1$ *then return* $\{(S_1, 0)\}$
$k \leftarrow \lfloor n/2 \rfloor$
$\mathcal{P} \leftarrow$ StringMergesort($\{S_1, S_2, \ldots, S_k\}$)
$\mathcal{Q} \leftarrow$ StringMergesort($\{S_{k+1}, S_{k+2}, \ldots, S_n\}$)
return StringMerge($\mathcal{P}, \mathcal{Q}$)

The output is of the form

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

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

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

Algorithm 3.20: 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 $\mathcal{R}$

$\mathcal{R} \leftarrow \emptyset$; $i \leftarrow 1$; $j \leftarrow 1$
while $i \leq m$ *and* $j \leq n$ *do*
    if $k_i > \ell_j$ *then* append (S_i, k_i) to $\mathcal{R}$; $i \leftarrow i + 1$
    else if $\ell_j > k_i$ *then* append (T_j, ℓ_j) to $\mathcal{R}$; $j \leftarrow j + 1$
    else // $k_i = \ell_j$
        $(x, h) \leftarrow$ LcpCompare(S_i, T_j, k_i)
        if $x = "<"$ *then*
            append (S_i, k_i) to $\mathcal{R}$; $i \leftarrow i + 1$
            $\ell_j \leftarrow h$
        else
            append (T_J, ℓ_j) to $\mathcal{R}$; $j \leftarrow j + 1$
            $k_i \leftarrow h$
while $i \leq m$ *do* append (S_i, k_i) to $\mathcal{R}$; $i \leftarrow i + 1$
while $j \leq n$ *do* append (T_J, ℓ_j) to $\mathcal{R}$; $j \leftarrow j + 1$
return $\mathcal{R}$

Lemma 3.21: 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 3.18.
  - $lcp(S_i, T_j) = lcp(X, T_j)$ because by Lemma 3.17 $lcp(X, T_j) = \min\{lcp(X, S_i), lcp(S_i, T_j)\}$.

  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. □

Theorem 3.22: String mergesort sorts a set $\mathcal{R}$ of n strings in $\mathcal{O}(DP(\mathcal{R}) + n \log n)$ time.

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

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

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

Search Trees for Strings

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

Insert and Delete.

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

Range query: Find all objects in a given range.

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

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

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

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

When the objects are strings, operations slow down:

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

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

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

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

The prefix query is a special type of range query.

Trie

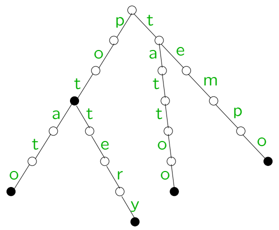
A trie is a *rooted tree* with the following properties:

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

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

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

Example 3.23: $trie(\mathcal{R})$ for $\mathcal{R} = \{\texttt{pot}, \texttt{potato}, \texttt{pottery}, \texttt{tattoo}, \texttt{tempo}\}$.



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

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

There are many implementation options including:

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

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

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

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

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

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

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

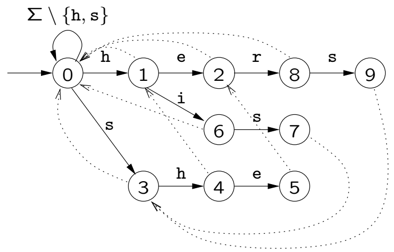
Note that a trie is a complete representation of the strings. There is no need to store the strings elsewhere. Most other data structures hold pointers to the strings, which are stored elsewhere.

Aho–Corasick Algorithm

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



Algorithm 3.25: 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) Construct AC automaton
(2) $v \leftarrow root$
(3) *for* $j \leftarrow 0$ *to* $n-1$ *do*
(4)   *while* $\mathit{child}(v, T[j]) = \perp$ *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)

Let S_v denote the string that node v represents.

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

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

Algorithm 3.26: Aho–Corasick trie construction
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* $\mathit{child}(v, P_i[j]) \neq \perp$ *do*
(5) $v \leftarrow \mathit{child}(v, P_i[j])$; $j \leftarrow j + 1$
(6)   *while* $j < |P_i|$ *do*
(7)   Create new node u
(8) $\mathit{child}(v, P_i[j]) \leftarrow u$
(9) $v \leftarrow u$; $j \leftarrow j + 1$
(10) $\mathit{patterns}(v) \leftarrow \{i\}$

- The creation of a new node v initializes *patterns*(v) to $\emptyset$ and *child*(v, c) to $\perp$ for all $c \in \Sigma$.
- After this algorithm, $i \in$ *patterns*(v) iff v represents P_i.

Algorithm 3.27: Aho–Corasick automaton construction
Input: AC trie: $root$, *child*() and *patterns*()
Output: AC automaton: *fail*() and updated *child*() and *patterns*()
(1) $queue \leftarrow \emptyset$
(2) *for* $c \in \Sigma$ *do*
(3)   *if* $\mathit{child}(root, c) = \perp$ *then* $\mathit{child}(root, c) \leftarrow root$
(4)   *else*
(5) $v \leftarrow \mathit{child}(root, c)$
(6) $fail(v) \leftarrow root$
(7)   pushback$(queue, v)$
(8) *while* $queue \neq \emptyset$ *do*
(9) $u \leftarrow$ popfront$(queue)$
(10)   *for* $c \in \Sigma$ such that $\mathit{child}(u, c) \neq \perp$ *do*
(11) $v \leftarrow \mathit{child}(u, c)$
(12) $w \leftarrow \mathit{fail}(u)$
(13)   *while* $\mathit{child}(w, c) = \perp$ *do* $w \leftarrow \mathit{fail}(w)$
(14) $fail(v) \leftarrow \mathit{child}(w, c)$
(15) $\mathit{patterns}(v) \leftarrow \mathit{patterns}(v) \cup \mathit{patterns}(\mathit{fail}(v))$
(16)   pushback$(queue, v)$

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

Assuming σ is constant:

- The preprocessing time is $\mathcal{O}(m)$, where $m = ||\mathcal{P}||$.
  - The only non-trivial issue is the while-loop on line (13). 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 $w_j = \mathit{fail}(v_j)$ for all j. The depths in the sequence $w_1, w_2, \ldots, w_\ell$ increase by at most one in each step. Every round in the while-loop when computing w_j reduces the depth of w_j by at least one. Therefore, the total number of rounds when computing $w_1, w_2, \ldots, w_\ell$ is at most $\ell = |P_i|$. Thus, the while-loop is executed at most $||\mathcal{P}||$ times during the whole algorithm.
- The search time is $\mathcal{O}(n)$.
- The space complexity is $\mathcal{O}(m)$.

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

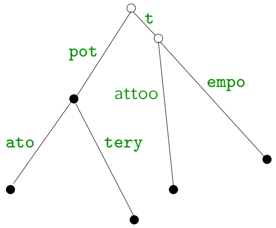
Compact Trie

Tries suffer from a large number nodes, $\Omega(||\mathcal{R}||)$ in the worst case.

- The space requirement is large, since each node needs much more space than a single symbol.
- Traversal of a subtree is slow, which affects prefix and range queries.

The compact trie reduces the number of nodes by replacing *branchless path segments* with a single edge.

Example 3.28: Compact trie for $\mathcal{R} = \{\texttt{pot}, \texttt{potato}, \texttt{pottery}, \texttt{tattoo}, \texttt{tempo}\}$.



- The number of nodes and edges is $\mathcal{O}(|\mathcal{R}|)$.
- The egde labels are factors of the input strings. Thus they can be stored in constant space using pointers to the strings, which are stored separately.
- Any subtree can be traversed in linear time in the the number of leaves in the subtree. Thus a prefix query for a string S can be answered in $\mathcal{O}(|S| + r)$ time, where r is the *number* of strings in the answer.

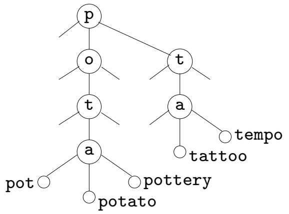
Ternary Tree

The binary tree implementation of a trie supports ordered alphabets but awkwardly. Ternary tree is a simpler data structure based on symbol comparisons.

Ternary tree is like a binary tree except:

- Each internal node has three children: smaller, equal and larger.
- The branching is based on a single symbol at a given position. The position is zero at the root and increases along the middle branches.

Example 3.29: Ternary tree for $\mathcal{R} = \{\texttt{pot}, \texttt{potato}, \texttt{pottery}, \texttt{tattoo}, \texttt{tempo}\}$.



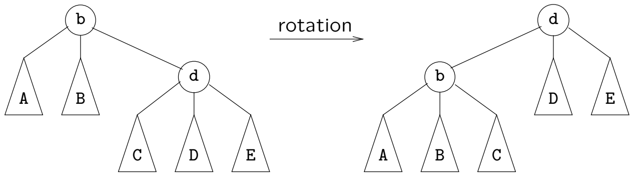
There is an analogy between *sorting algorithms* and *search trees* for strings.

sorting algorithm	search trees
standard binary quicksort	standard binary tree
string quicksort	ternary tree
radix sort	trie

The ternary tree can be seen as the partitioning structure by string quicksort.

A ternary tree 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 trees.



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

In a balanced ternary tree, each step down either

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

Thus, in a ternary tree storing n strings, any downward traversal following a string S takes at most $\mathcal{O}(|S| + \log n)$ time.

For the ternary tree of a string set $\mathcal{R}$ of size n:

- The number of nodes is $\mathcal{O}(DP(\mathcal{R}))$.

- Insertion, deletion, lookup and lcp query for a string S takes $\mathcal{O}(|S| + \log n)$ time.

- Prefix search for a string S takes $\mathcal{O}(|S| + \log n + DP(\mathcal{Q}))$, where $\mathcal{Q}$ is the set of strings given as the result of the query. With some additional data structures, this can be reduced to $\mathcal{O}(|S| + \log n + |\mathcal{Q}|)$

String Binary Search

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

Algorithm 3.30: 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.

```
left ← 0; right ← n          // final answer is in the range [left..right]
while left < right do
    mid ← ⌈(left + right)/2⌉
    if k_mid < x then left ← mid
    else right ← mid − 1
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

- $\mathcal{O}(m + \log n \log_\sigma n)$ 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 3.31: Let $A \leq B, B' \leq C$ be strings. Then $lcp(B, B') \geq lcp(A, C)$.

Proof. Let $B_{min} = \min\{B, B'\}$ and $B_{max} = \max\{B, B'\}$. By Lemma 3.17,

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

$\square$

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 $S_{right+1}$, and we know that $S_{left} \leq P, S_{mid} \leq S_{right+1}$.
- If we are using LcpCompare, we know $lcp(S_{left}, P)$ and $lcp(P, S_{right+1})$.

By Lemmas 3.17 and 3.31,

$$lcp(P, S_{mid}) \geq lcp(S_{left}, S_{right+1}) = \min\{lcp(S_{left}, P), lcp(P, S_{right+1})\}$$

Thus we can skip $\min\{lcp(S_{left}, P), lcp(P, S_{right+1})\}$ first characters when comparing P and S_{mid}.

Algorithm 3.32: 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.

$left \leftarrow 0$; $right \leftarrow n$
$llcp \leftarrow 0$; $rlcp \leftarrow 0$
while $left < right$ *do*
    $mid \leftarrow \lceil (left + right)/2 \rceil$
    $mlcp \leftarrow \min\{llcp, rlcp\}$
    $(x, mlcp) \leftarrow \text{LcpCompare}(S_{mid}, P, mlcp)$
    if $x =$ "$<$" *then* $left \leftarrow mid$; $llcp \leftarrow mclp$
    else $right \leftarrow mid - 1$; $rlcp \leftarrow mclp$
return $left$

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

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 $S_{right+1}$, and we know $lcp(S_{left}, P)$ and $lcp(P, S_{right+1})$.

The values $left$ and $right$ depend only on mid. In particular, they do not depend on P. Thus, we can precompute and store the values

$$LLCP[mid] = lcp(S_{left}, S_{mid})$$
$$RLCP[mid] = lcp(S_{mid}, S_{right+1})$$

Now we know all lcp values between P, S_{left}, S_{mid}, $S_{right+1}$ except $lcp(P, S_{mid})$. The following lemma shows how to utilize this.

Lemma 3.33: Let $A \leq B, B' \leq C$ be strings.
(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 directly from Lemma 3.18. Then by Lemma 3.17, $lcp(A, B') = \min\{lcp(A, B), lcp(B, B')\}$. Since $lcp(A, B') < lcp(A, B)$, we must have $lcp(A, B') = lcp(B, B')$.

In case (e), we use Lemma 3.17:

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

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

Algorithm 3.34: 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) $left \leftarrow 0$; $right \leftarrow n$
(2) $llcp \leftarrow 0$; $rlcp \leftarrow 0$
(3) *while* $left < right$ *do*
(4) $\quad mid \leftarrow \lceil (left + right)/2 \rceil$
(5) $\quad$ *if* $LLCP[mid] > llcp$ *then* $left \leftarrow mid$
(6) $\quad$ *else if* $RLCP[mid] > rlcp$ *then* $right \leftarrow mid - 1$
(7) $\quad$ *else if* $llcp > LLCP[mid]$ *then* $right \leftarrow mid - 1$; $rlcp \leftarrow LLCP[mid]$
(8) $\quad$ *else if* $rlcp > RLCP[mid]$ *then* $left \leftarrow mid$; $llcp \leftarrow RLCP[mid]$
(9) $\quad$ *else*
(10) $\quad\quad mlcp \leftarrow \max\{llcp, rlcp\}$
(11) $\quad\quad (x, mlcp) \leftarrow \text{LcpCompare}(S_{mid}, P, mlcp)$
(12) $\quad\quad$ *if* $x =$ "<" *then* $left \leftarrow mid$; $llcp \leftarrow mclp$
(13) $\quad\quad$ *else* $right \leftarrow mid - 1$; $rlcp \leftarrow mclp$
(14) *return* $left$

Theorem 3.35: An ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ can be preprocessed in $\mathcal{O}(DP(\mathcal{R}))$ 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 $\mathcal{O}(dp_{\mathcal{R}}(S_{mid}))$ time. Thus the arrays $LLCP$ and $RLCP$ can be computed in $\mathcal{O}(DP(\mathcal{R}))$ time and stored in $\mathcal{O}(n)$ space.

The main while loop in Algorithm 3.7.7 is executed $\mathcal{O}(\log n)$ times and everything except LcpCompare on line (11) needs constant time.

If a given LcpCompare call performs $1 + t$ symbol comparisons, $mclp$ increases by t on line (11). Then on lines (12)–(13), either $llcp$ or $rlcp$ increase 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 $\mathcal{O}(|P|)$. □

Binary search can be seen as a search on an *implicit* binary search tree, where the middle element is the root, the middle elements of the first and second half are the children of the root, etc.. The string binary search technique can be extended for arbitrary *binary search trees*.

- Let S_v be the string stored at a node v in a binary search tree. Let $S_<$ and $S_>$ be the closest lexicographically smaller and larger strings stored at *ancestors* of v.

- The comparison of a query string P and the string S_v is done the same way as the comparison of P and S_{mid} in string binary search. The roles of S_{left} and $S_{right+1}$ are taken by $S_<$ and $S_>$.

- If each node v stores the values $lcp(S_<, S_v)$ and $lcp(S_v, S_>)$, then a search in a balanced search tree can be executed in $\mathcal{O}(|P| + \log n)$ time. Other operations including insertions and deletions take $\mathcal{O}(|P| + \log n)$ time too.

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 a *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 $T_{[0..n]}$.

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 $DP(T_{[0..n]}) = \Theta(n^2)$, for example, when $T = \mathtt{a}^n$ or $T = XX$ for any string X.

- Trie with $||T_{[0..n]}||$ nodes and ternary tree with $DP(T_{[0..n]})$ nodes would be too large.
- Compact trie with $\mathcal{O}(n)$ nodes and an ordered array with $n + 1$ entries have linear size.
- Binary search tree with $\mathcal{O}(n)$ nodes would be an option too, but an ordered array is a better choice for a static text. We do not cover the case of dynamic, changing text on this course: it a non-trivial problem because changing a single symbol can affect a large number of suffixes.

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

Suffix Tree

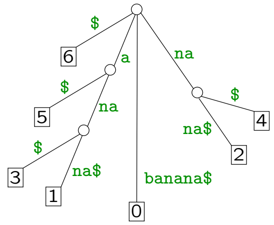
The suffix tree of a text T is the compact trie of the set $T_{[0..n]}$ of all suffixes of T.

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: $T = \mathtt{banana\$}$.



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 $\Theta(DP(T_{[0..n]}))$. This is later modified 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 in Algorithm 3.26 (AC trie construction) except it has been modified to work on a compact trie instead of a trie.

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 the string S_u represented by 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 *pair* (u, d) with $\mathit{depth}(\mathit{parent}(u)) < d \leq \mathit{depth}(u)$ represents a position on the edge $(\mathit{parent}(u), u)$ corresponding to the string $S_{(u,d)} = T[\mathit{start}(u) \ldots \mathit{start}(u) + d)$.

Note that the positions (u, d) correspond to nodes in the uncompact trie.

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*

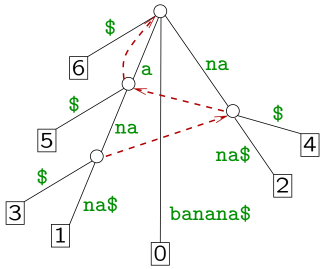
create new node $root$; $\mathit{depth}(root) \leftarrow 0$
$u \leftarrow root$; $d \leftarrow 0$
for $i \leftarrow 0$ *to* n *do* // insert suffix T_i
  while $d = \mathit{depth}(u)$ *and* $\mathit{child}(u, T[i+d]) \neq \perp$ *do*
    $u \leftarrow \mathit{child}(u, T[i+d])$; $d \leftarrow d+1$
    while $d < \mathit{depth}(u)$ *and* $T[\mathit{start}(u) + d] = T[i+d]$ *do* $d \leftarrow d+1$
  if $d < \mathit{depth}(u)$ *then* // we are in the middle of an edge
    create new node v
    $\mathit{start}(v) \leftarrow i$; $\mathit{depth}(v) \leftarrow d$
    $p \leftarrow \mathit{parent}(u)$
    $\mathit{child}(v, T[\mathit{start}(u) + d]) \leftarrow u$; $\mathit{parent}(u) \leftarrow v$
    $\mathit{child}(p, T[i + \mathit{depth}(p)]) \leftarrow v$; $\mathit{parent}(v) \leftarrow p$
    $u \leftarrow v$
  create new leaf w // w represents suffix T_i
  $\mathit{start}(w) \leftarrow i$; $\mathit{depth}(w) \leftarrow n - i + 1$
  $\mathit{child}(u, T[i+d]) \leftarrow w$; $\mathit{parent}(w) \leftarrow u$
  $u \leftarrow root$; $d \leftarrow 0$

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 $T = \texttt{banana\$}$ 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}.

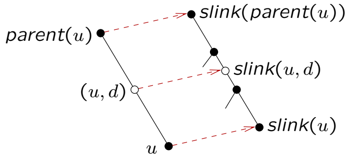
If u is an internal node, then it has two child edges with labels starting with different symbols, say a and b, which means that $T[i..j)a$ and $T[i..j)b$ occur somewhere in T. Then, $T[i+1..j)a$ and $T[i+1..j)b$ occur in T too, and thus there must be a branching node v representing $T[i+1..j)$. $\square$

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

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 position represented by a pair (u, d):

slink(u, d)
(1) $v \leftarrow$ *slink*(*parent*(u))
(2) *while* *depth*(v) $< d - 1$ *do*
(3) $\quad v \leftarrow$ *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) $v \leftarrow$ *slink*(*parent*(u))
(2) *while* *depth*(v) < *depth*(u) − 1 *do*
(3) $\quad v \leftarrow$ *child*(v, T[*start*(u) + *depth*(v) + 1])
(4) *if* *depth*(v) > *depth*(u) − 1 *then*
(5) $\quad$ create new node w
(6) $\quad$ *start*(w) ← *start*(u) + 1; *depth*(w) ← *depth*(u) − 1; *slink*(w) ← $\perp$
(7) $\quad p \leftarrow$ *parent*(v)
(8) $\quad$ *child*(w, T[*start*(v) + *depth*(w)]) ← v; *parent*(v) ← w
(9) $\quad$ *child*(p, T[*start*(w) + *depth*(p)]) ← w; *parent*(w) ← p
(10) $\quad v \leftarrow w$
(11) *slink*(u) ← v

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 lines (4)–(10) is not necessary 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 position on the path labelled T_{j+1}. However, it may be that it does not become a branching node until the insertion of T_{i+1}.

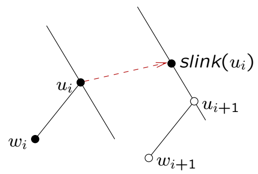
- In such a case, ComputeSlink(u_i) creates *slink*(u_i) a moment before it would otherwise be created by the brute force construction.

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) $u \leftarrow root$; $d \leftarrow 0$
(3) *for* $i \leftarrow 0$ *to* n *do* // insert suffix T_i
(4) *while* $d = \mathit{depth}(u)$ *and* $\mathit{child}(u, T[i+d]) \neq \perp$ *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* // we are in the middle of an edge
(8) create new node v
(9) $\mathit{start}(v) \leftarrow i$; $\mathit{depth}(v) \leftarrow d$; $\mathit{slink}(v) \leftarrow \perp$
(10) $p \leftarrow \mathit{parent}(u)$
(11) $\mathit{child}(v, T[\mathit{start}(u) + d]) \leftarrow u$; $\mathit{parent}(u) \leftarrow v$
(12) $\mathit{child}(p, T[i + \mathit{depth}(p)]) \leftarrow v$; $\mathit{parent}(v) \leftarrow p$
(13) $u \leftarrow v$
(14) create new leaf w // w represents suffix T_i
(15) $\mathit{start}(w) \leftarrow i$; $\mathit{depth}(w) \leftarrow n - i + 1$
(16) $\mathit{child}(u, T[i + d]) \leftarrow w$; $\mathit{parent}(w) \leftarrow u$
(17) *if* $\mathit{slink}(u) = \perp$ *then* ComputeSlink(u)
(18) $u \leftarrow \mathit{slink}(u)$; $d \leftarrow d - 1$

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 (18) 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 (2)–(3) 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 (2)–(3) 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.

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.

Text Statistics

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

- The *number of distinct factors* in the text is exactly the number of nodes in the (uncompact) trie. Using the suffix tree, this number can be computed as the total length of the edges plus one (root/empty string). The time complexity is $\mathcal{O}(n)$ 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 three of the string $S\pounds T\$$, where $\pounds$ 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 position, 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 $T[0..n)$ with respect to a string S is an array $MS[0..n)$, where $MS[i]$ is a pair (ℓ_i, p_i) such that

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

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

- If before reading $T[i]$ we are at the node v in the automaton, then $T[i-d..d) = S[j..j+d)$, where $j = \mathit{start}(v)$ and $d = \mathit{depth}(v)$. If reading $T[i]$ causes a failure transition, then $MS[i-d] = (d, j)$.

From the matching statistics, we can easily compute the longest common factor of S and T. 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 the lcp of any two suffixes 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 and T^R_{n-i-1}, which can be computed in constant time. Then $T[i-k..i+k]$ 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.

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 $i \in [0..n]$.

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 $T = \texttt{banana\$}$.

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

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 see that the suffix array can be constructed in the same asymptotic time it takes to *sort the characters* of the text.

As with suffix trees, *exact string matching* in T can be performed by *prefix search* on the suffix array. The answer can be conveniently given as a *contiguous range* in the suffix array containing the suffixes. The range can be found using string binary search.

- If we have the additional arrays $LLCP$ and $RLCP$, the result range 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.

LCP Array

Efficient string binary search uses the arrays $LLCP$ and $RLCP$. For many applications, the suffix array is augmented with a different lcp array $LCP[1..n]$. For all i,

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

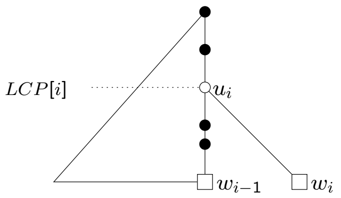
This is the same as the lcp information in the output of StringMergesort.

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$`

The suffix tree can be easily constructed from the suffix and LCP arrays in linear time.

- Insert the suffixes into the tree in lexicographical order.
- The leaf w_i representing the suffix T_i is inserted as the rightmost leaf. The parent u_i of w_i is along the rightmost path in the tree, and the depth of u_i is $LCP[i]$. If there is no node at that depth, a new node is inserted.



- Keep the nodes on the rightmost path on a *stack* with the deepest node on top. The node u_i or the edge, where u_i is inserted, is found by removing nodes from the stack until the right depth has been reached. Note that the removed nodes are no more on the rightmost path after the insertion of w_i.

The suffix tree can be replaced by the suffix and LCP arrays in many applications. 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

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

  This follows from the suffix tree construction on the previous slide and the formula we saw earlier for the suffix tree.

- Matching statistics of T with respect to S can be computed in linear time using the generalized suffix array of S and T (i.e., suffix array of $S£T\$$) and its LCP array.

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 4.9: 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 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. RMQ preprocessing is much simpler than LCA preprocessing.

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

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 by a suffix array range corresponding to 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.

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

Proof. Let T_j be the lexicographic predecessor of T_i, i.e., $T_j < T_i$ and there are no other suffixes between them in the lexicographical order.

- Then $LCP[SA^{-1}[i]] = lcp(T_i, T_j) = \ell$.

- If $\ell > 0$, then for some symbol c, $T_i = cT_{i+1}$ and $T_j = cT_{j+1}$. Thus $T_{j+1} < T_{i+1}$ and $lcp(T_{i+1}, T_{j+1}) = \ell - 1$.

- Then $LCP[[SA^{-1}[i+1]] \geq lcp(T_{i+1}, T_{j+1}) = \ell - 1$.

$\square$

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

Algorithm 4.11: 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) $\quad k \leftarrow SA^{-1}[i]$ $\quad$ // $i = SA[k]$
(4) $\quad j \leftarrow SA[k-1]$
(5) $\quad$ *while* $T[i+\ell] = T[j+\ell]$ *do* $\ell \leftarrow \ell + 1$
(6) $\quad LCP[k] \leftarrow \ell$
(7) $\quad$ *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.

Suffix Array Construction

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

- Using standard sorting or string sorting the time complexity is $\Omega(DP(T_{[0..n]}))$.
- 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 size alphabet.

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

In fact, possibly the fastest way to construct a suffix *tree* is to first construct the suffix array and the LCP array, and then the suffix tree using the algorithm we saw earlier.

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 $T_{[0..n]}^\ell$ to sort the set $T_{[0..n]}^{2\ell}$. The key idea is assigning *order preserving names* for the factors in $T_{[0..n]}^\ell$. For $i \in [0..n]$, let N_i^ℓ be an integer in the range $[0..n]$ such that, for all $i, j \in [0..n]$:

$$N_i^\ell \leq N_j^\ell \text{ if and only if } T_i^\ell \leq T_j^\ell \ .$$

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

$$N_i^\ell = |\{j \in [0, n] \mid T_j^\ell < T_i^\ell\}| \ .$$

Example 4.12: Prefix doubling for $T = \texttt{banana\$}$.

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 4.13: 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 implementation 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 assume that the text is like T' during the suffix array construction. After the construction, we can use either T or T' as the text depending on what we want to do.

Recursive Suffix Array Construction

Let us now describe a linear time algorithm 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 algorithm is:

0. Divide the suffixes into two subsets $A \subset [0..n]$ and $\bar{A} = [0..n] \setminus A$.
1. Sort the set T_A. This is done by a reduction to the suffix array construction of a string of length $|A|$, which is done *recursively*.
2. Sort the set $T_{\bar{A}}$ using the order of T_A.
3. Merge the two sorted sets T_A and $T_{\bar{A}}$.

The set A can be chosen so that

- $|A| \leq \alpha n$ for a constant $\alpha < 1$.
- Excluding the recursive call, all steps can be done in linear time.

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

The set A must be chosen so that:

1. Sorting T_A can be reduced to suffix array construction on a text of length $|A|$.
2. Given sorted T_A the suffix array of T is easy to construct.

There are a few different options. Here we use the simplest one.

Step 0: Select A.

- For $k \in \{0, 1, 2\}$, define $C_k = \{i \in [0..n] \mid i \bmod 3 = k\}$.
- Let $A = C_1 \cup C_2$.

Example 4.14:

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	$

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

Step 1: Sort T_A.

- For $k \in \{1, 2\}$, Construct the strings $R_k = (T_k^3, T_{k+3}^3, T_{k+6}^3, \ldots, T_{\max C_k}^3)$ whose characters are factors of length 3 in the original text, and let $R = R_1 R_2$.
- Replace each factor T_i^3 in R with a lexicographic name $N_i^3 \in [1..|R|]$. 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 $SA_{R'}^{-1}$ of R'. This is done recursively unless all symbols in R' are unique, in which case $SA_{R'}^{-1} = R'$.
- From $SA_{R'}^{-1}$, we get lexicographic names for suffixes in T_A.
  For $i \in A$, let $N[i] = SA_{R'}^{-1}[j]$, where j is the position of T_i^3 in R.
  For $i \in \bar{A}$, let $N[i] = \perp$. Also let $N[n+1] = N[n+2] = 0$.

Example 4.15:

R	abb	ada	bba	do$	bba	dab	bad	o$	
R'	1	2	4	7	4	6	3	8	0
$SA_{R'}^{-1}$	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: Sort $T_{\bar{A}}$.

- For each $i \in \bar{A}$, 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 $N[i+1] \ne \perp$.

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

Example 4.16:

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 $(\$, 0) < (\mathtt{a}, 5) < (\mathtt{a}, 7) < (\mathtt{b}, 2) < (\mathtt{y}, 1)$.

Step 3: Merge T_A and $T_{\bar{A}}$.

- Use comparison based merging algorithm needing $\mathcal{O}(n)$ comparisons.

- To compare $T_i \in T_A$ and $T_j \in T_{\bar{A}}$, we have two cases:
$$i \in C_1 : T_i \le T_j \iff (T[i], N[i+1]) \le (T[j], N[j+1])$$
$$i \in C_2 : T_i \le T_j \iff (T[i], T[i+1], N[i+2]) \le (T[j], T[j+1], N[j+2])$$
Note that $N[i+1] \ne \perp$ in the first case and $N[i+2] \ne \perp$ in the second case.

Example 4.17:

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 4.18: The suffix array of a string $T[0..n)$ can be constructed in $\mathcal{O}(n)$ time plus the time needed to sort the characters of T.

- There are a few other suffix array construction algorithms and one suffix tree construction algorithm (Farach's) with the same time complexity.

- All of them have a similar recursive structure, where the problem is reduced to suffix array construction on a shorter string that represents a subset of all suffixes.

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 $T[n] = \$$. 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 4.19: The BWT of $T = \mathtt{banana\$}$ is $L = \mathtt{annb\$aa}$.

F						L
$	b	a	n	a	n	a
a	$	b	a	n	a	n
a	n	a	$	b	a	n
a	n	a	n	a	$	b
b	a	n	a	n	a	$
n	a	$	b	a	n	a
n	a	n	a	$	b	a

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*.

Inverse BWT

Let $\mathcal{M}'$ be the matrix obtained by rotating $\mathcal{M}$ one step to the right.

Example 4.20:

$$\begin{array}{c} \mathcal{M} \\ \begin{array}{cccccc|c|} \$ & b & a & n & a & n & a \\ a & \$ & b & a & n & a & n \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \\ b & a & n & a & n & a & \$ \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \end{array} \end{array} \xrightarrow{\text{rotate}} \begin{array}{c} \mathcal{M}' \\ \begin{array}{|c|cccccc} a & \$ & b & a & n & a & n \\ n & a & \$ & b & a & n & a \\ n & a & n & a & \$ & b & a \\ b & a & n & a & n & a & \$ \\ \$ & b & a & n & a & n & a \\ a & n & a & \$ & b & a & n \\ a & n & a & n & a & \$ & b \end{array} \end{array}$$

- The rows of $\mathcal{M}'$ are the rotations of T in a different order.

- In $\mathcal{M}'$ without the first column, the rows are sorted lexicographically. If we sort the rows of $\mathcal{M}'$ *stably* by the first column, we obtain $\mathcal{M}$.

This cycle $\mathcal{M} \xrightarrow{\text{rotate}} \mathcal{M}' \xrightarrow{\text{sort}} \mathcal{M}$ is the key to inverse BWT.

- In the cycle, each column moves one step to the right and is permuted. The permutation is fully determined by the last column of $\mathcal{M}$, i.e., the BWT.

- By repeating the cycle, we can reconstruct $\mathcal{M}$ from the BWT.

- To reconstruct T, we do not need to compute the whole matrix just one row.

Example 4.21:

```
------a          a------        $-----a          a$-----        $b----a
------n          n------        a-----n          na-----        a$----n
------n  rotate  n------  sort  a-----n  rotate  na-----  sort  an----n
------b   -->    b------  -->   a-----b   -->    ba-----  -->   an----b
------$          $------        b-----$          $b-----        ba----$
------a          a------        n-----a          an-----        na----a
------a          a------        n-----a          an-----        na----a

          $ba---a          $ban--a          $bana-a          $banana
          a$b---n          a$ba--n          a$ban-n          a$banan
rotate    ana---n  rotate  ana$--n  rotate  ana$b-n  rotate  ana$ban
& sort    ana---b  & sort  anan--b  & sort  anana-b  & sort  anana$b
 -->      ban---$   -->    bana--$   -->    banan-$   -->    banana$
          na$---a          na$b--a          na$ba-a          na$bana
          nan---a          nana--a          nana$-a          nana$ba
```


The permutation that transforms $\mathcal{M}'$ into $\mathcal{M}$ is called the *LF-mapping*.

- LF-mapping is the permutation that stably sorts the BWT L, i.e., $F[LF[i]] = L[i]$. Thus it is easy to compute from L.
- Given the LF-mapping, we can easily follow a row through the permutations.

Algorithm 4.22: Inverse BWT
Input: BWT $L[0..n]$
Output: text $T[0..n]$
Compute LF-mapping:
(1) *for* $i \leftarrow 0$ *to* n *do* $R[i] = (L[i], i)$
(2) sort R (stably by first element)
(3) *for* $i \leftarrow 0$ *to* n *do*
(4) $\quad (\cdot, j) \leftarrow R[i]$; $LF[j] \leftarrow i$
Reconstruct text:
(5) $j \leftarrow$ position of \$ in L
(6) *for* $i \leftarrow n$ *downto* 0 *do*
(7) $\quad T[i] \leftarrow L[j]$
(8) $\quad j \leftarrow LF[j]$
(9) *return* T

The time complexity is dominated by the stable sorting.

On Burrows-Wheeler Compression

The basic principle of text compression is that, the more frequently a factor occurs, the shorter its encoding should be.

Let c be a symbol and w a string such that the factor cw occurs frequently in the text.

- The occurrences of cw may be distributed all over the text, so recognizing cw as a frequently occurring factor is not easy. It requires some large, global data structures.
- In the BWT, the high frequency of cw means that c is frequent in that part of the BWT that corresponds to the rows of the matrix $\mathcal{M}$ beginning with w. This is easy to recognize using local data structures.

This localizing effect makes compressing the BWT much easier than compressing the original text.

We will not go deeper into text compression on this course.

Example 4.23: A part of the BWT of a reversed english text corresponding to rows beginning with `ht`:

```
oreeereoeeieeeeaooeeeeeeaereeeeeeeeeeeeereeeeeeeeeeeaaeeaeeeeeeee
eaeeeeeeeeaeieeeeeeeeereeeeeeeeeeeeeeeeeeeeeeeeaeeieeeeeeeaaieee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeaeieeeeeeeeeeeeeeeeeeeeeeeeeeeeeeaee
eeeeeeeeeeeeeeeeeereeeeeeeeeeieaeeeeieeeeaeeeeeeeeeieeeeeeeee
eeeieeeeeeeeioaaeeaoereeeeeeeeeeeaaeaaeeeeieeeeeeeieeeeeeeeeaeeee
eeaeeeeeeereeeaeeeeeieeeeeeeeeiieee. e  eeeeiiiiii e          ,
i   o       oo e  eiiiiee,er  ,  ,     ,  . iii
```

and some of those symbols in context:

```
t raise themselves, and the hunter, thankful and r
ery night it flew round the glass mountain keeping
agon, but as soon as he threw an apple at it the b
f animals, were resting themselves.  "Halloa, comr
ple below to life.  All those who have perished on
 that the czar gave him the beautiful Princess Mil
ng of guns was heard in the distance.  The czar an
cked magician put me in this jar, sealed it with t
o acted as messenger in the golden castle flew pas
u have only to say, 'Go there, I know not where; b
```


Backward Search

Let $P[0..m)$ be a pattern and let $[b..e)$ be the suffix array range corresponding to suffixes that begin with P, i.e., $SA[b..e)$ contains the starting positions of P in the text T. Earlier we noted that $[b..e)$ can be found by *binary search* on the suffix array.

Backward search is a different technique for finding this range. It is based on the observation that $[b..e)$ is also the range of rows in the matrix $\mathcal{M}$ beginning with P.

Let $[b_i, e_i)$ be the range for the pattern suffix P_i. The backward search will first compute $[b_{m-1}, e_{m-1})$, then $[b_{m-2}, e_{m-2})$, etc. until it obtains $[b_0, e_0) = [b, e)$. Hence the name backward search.

Backward search uses the following data structures:

- An array $C[0..\sigma)$, where $C[c] = \left|\{i \in [0..n] \mid L[i] < c\}\right|$. In other words, $C[c]$ is the number of occurrences of symbols that are smaller than c.
- The function $\mathit{rank}_L : \Sigma \times [0..n+1] \to [0..n]$:

  $$\mathit{rank}_L(c, j) = \left|\{i \mid i < j \text{ and } L[i] = c\}\right| .$$

  In other words, $\mathit{rank}_L(c, j)$ is the number of occurrences of c in L before position i.

Given b_{i+1}, we can now compute b_i as follows. Computing e_i from e_{i+1} is similar.

- $C[P[i]]$ is the number of rows beginning with a symbol smaller than $P[i]$. Thus $b \geq C[P[i]]$.
- $\mathit{rank}_L(P[i], b_{i+1})$ is the number of rows that are lexicographically smaller than P_{i+1} and contain $P[i]$ at the last column. Rotating these rows one step to the right, we obtain the rotations of T that begin with $P[i]$ and are lexicographically smaller than P_i.
- Thus $b_i = C[P[i]] + \mathit{rank}_L(P[i], b_{i+1})$.

Algorithm 4.24: Backward Search
Input: array C, function *rank*$_L$, pattern P
Output: suffix array range $[b..e)$ containing starting positions of P

(1) $b \leftarrow 0$; $e \leftarrow n + 1$
(2) *for* $i \leftarrow m - 1$ *downto* 0 *do*
(3) $\quad c \leftarrow P[i]$
(4) $\quad b \leftarrow C[c] + \mathit{rank}_L(c, b)$
(5) $\quad e \leftarrow C[c] + \mathit{rank}_L(c, e)$
(6) *return* $[b..e)$

- The array C requires an integer alphabet that is not too large.

- The trivial implementation of the function *rank*$_L$ as an array requires $\Theta(\sigma n)$ space, which is often too much. There are much more space efficient (but slower) implementations. There are even implementations with a size that is close to the size of the *compressed text*.