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

- The original choice is $r = \sigma$ and q is a large prime.
- Another possibility is that q is a power of two and r is a small prime ($r = 37$ has been suggested). This is faster in practice, because the slow modulo operations can be replaced by bitwise shift operations. If $q = 2^w$, where w is the machine word size, the modulo operations can be omitted completely.
- If q and r were both powers of two, then only the last $\lceil (\log q)/\log r \rceil$ characters of the string would affect the hash value. More generally, q and r should be coprime, i.e, have no common divisors other than 1.
- The hash function can be randomized by choosing q or r randomly. For example, if q is a prime and r is chosen uniformly at random from $[0..q)$, the probability that two strings of length m collide is at most m/q.
- A random choice over a set of possibilities has the additional advantage that we can change the choice if the first choice leads to too many collisions.

Automata

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

A trie is a special type of an automaton.

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

Automata are much more powerful than tries in representing languages:

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

Automata support set inclusion testing but not other trie operations:

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

Sets of Strings: Summary

Efficient algorithms and data structures for sets of strings:

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

Lower bounds:

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

The central role of longest common prefixes:

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

2. Exact String Matching

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

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

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

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

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

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

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

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

```
i ← 0; j ← 0
while i < m and j < n do
    if P[i] = T[j] then i ← i + 1; j ← j + 1
    else j ← j − i + 1; i ← 0
if i = m then return j − m else return n
```

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

(Knuth–)Morris–Pratt

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

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

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

Example 2.3:

Brute force	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 2.4: 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

```
compute fail[0..m]
i ← 0; j ← 0
while i < m and j < n do
    if i = −1 or P[i] = T[j] then i ← i + 1; j ← j + 1
    else i ← fail[i]
if i = m then return j − m else return n
```

- $fail[i] = -1$ means that there is no more pattern positions to compare against this text positions and we should move to the next text position.
- $fail[m]$ is never needed here, but if we wanted to find all occurrences, it would tell how to continue after a full match.

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

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

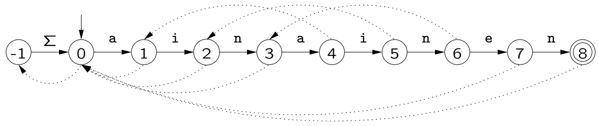
Example: `ai` is the longest proper border of `ainai`.

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

Example 2.5: Let $P = \texttt{ainainen}$.

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

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



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

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

Algorithm 2.6: Morris–Pratt failure function computation
Input: pattern $P = P[0 \dots m)$
Output: array $fail[0..m]$ for P
(1) $i \leftarrow -1; j \leftarrow 0;\ fail[j] \leftarrow i$
(2) **while** $j < m$ **do**
(3)   **if** $i = -1$ **or** $P[i] = P[j]$ **then** $i \leftarrow i+1; j \leftarrow j+1;\ fail[j] \leftarrow i$
(4)   **else** $i \leftarrow fail[i]$
(5) **return** $fail$

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

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

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

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

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

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

□

Shift-And (Shift-Or)

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

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

The bitvector D is updated at each text position j:

- There are precomputed bitvectors $B[c]$, for all $c \in \Sigma$, where $B[c].i = 1$ if $P[i] = c$ and $B[c].i = 0$ otherwise.
- D is updated in two steps:
  1. $D \leftarrow (D << 1) \mid 1$ (the bitwise shift and the bitwise or). Now D tells, which prefixes would match if $T[j]$ would match every character.
  2. $D \leftarrow D \,\&\, B[T[j]]$ (the bitwise and). Remove the prefixes where $T[j]$ does not match.

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

Algorithm 2.8: Shift-And
Input: text $T = T[0 \dots n)$, 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) $D \leftarrow ((D << 1) \mid 1) \,\&\, B[T[j]]$
(6)   **if** $D \,\&\, 2^{m-1} \neq 0$ **then return** $j - m + 1$ // $D.(m-1) = 1$
(7) **return** n

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

Example 2.9: $P = \texttt{assi}$, $T = \texttt{apassi}$, bitvectors are columns.

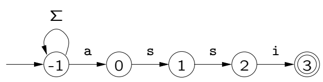
$B[c]$, $c \in \{\texttt{a},\texttt{i},\texttt{p},\texttt{s}\}$

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

D at each step

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

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



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

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

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

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

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

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

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

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

Karp–Rabin

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

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

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

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

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

Algorithm 2.10: Karp-Rabin

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

Choose q and r; $s \leftarrow r^{m-1} \bmod q$
$hp \leftarrow 0; ht \leftarrow 0$
for $i \leftarrow 0$ to $m-1$ do $hp \leftarrow (hp \cdot r + P[i]) \bmod q$ $\quad // \ hp = H(P)$
for $j \leftarrow 0$ to $m-1$ do $ht \leftarrow (ht \cdot r + T[j]) \bmod q$
for $j \leftarrow 0$ to $n-m-1$ do
    if $hp = ht$ then if $P = T[j \ldots j+m)$ then return j
    $ht \leftarrow ((ht - T[j] \cdot s) \cdot r + T[j+m]) \bmod q$
if $hp = ht$ then if $P = T[j \ldots j+m)$ 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 for a single pattern, but can be for multiple patterns (exercise).

Horspool

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

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

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

Example 2.11: Horspool

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


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

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

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

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

Example 2.12: $P =$ ainainen.

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

Algorithm 2.13: Horspool

Input: text $T = T[0 \ldots n)$, pattern $P = P[0 \ldots m)$
Output: position of the first occurrence of P in T
Preprocess:
for $c \in \Sigma$ do $shift[c] \leftarrow m$
for $i \leftarrow 0$ to $m-2$ do $shift[P[i]] \leftarrow m-1-i$
Search:
$j \leftarrow 0$
while $j + m \leq n$ do
    if $P[m-1] = T[j+m-1]$ then
        $i \leftarrow m-2$
        while $i \geq 0$ and $P[i] = T[j+i]$ do $i \leftarrow i-1$
        if $i = -1$ then return j
    $j \leftarrow j + shift[T[j+m-1]]$
return n

On an integer alphabet:

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

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