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.\footnote{The elements \( q \) and \( r \) were both powers of two, \( \log_2(q) \) \text{ bits} \( \log_2(r) \) \text{ bits} \text{ were both powers of two, \( \log_2(q) \) \text{ bits} and \( \log_2(r) \) \text{ bits}}\footnote{More generally, \( q \) and \( r \) should be \text{ co-prime}, i.e., have no common divisors other than 1.}

- Another possibility is that \( q \) is a power of two and \( r \) is a prime number. \( (r = 37 \text{ has been suggested}) \)\footnote{This is faster, because the slow modulo operations can be replaced by bitwise shift operations. If \( q = 2^k \), where \( k \) is the machine word size, the modulo operations can be omitted completely.}

- 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 \( m/n \).

- 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.

### Sets of Strings: Summary

Efficient algorithms and data structures for sets of strings:

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

### 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

### 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:** \( P = \text{aine} \) occurs at position 6 in \( T = \text{karjalainen} \).

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 \( T \).

#### (Knuth–)Morris–Pratt

The Brute force algorithm forgets everything when it shifts the window.\footnote{The MP failure function here. The KMP failure function is left for the exercises.}

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:**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Algorithm 2.4: Knuth–Morris–Pratt</th>
<th>Algorithm 2.2: Brute force</th>
</tr>
</thead>
<tbody>
<tr>
<td>aiainen</td>
<td>aiaiainen (6 comp.)</td>
<td>aiainen (1)</td>
</tr>
<tr>
<td>aiaiainen(1)</td>
<td>aiaiainen (1)</td>
<td>aiaiainen (1)</td>
</tr>
<tr>
<td>aiaiainen(3)</td>
<td>aiaiainen (1)</td>
<td>aiaiainen (1)</td>
</tr>
</tbody>
</table>

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 \( P[0..i] \neq T[j..j] \).

- Now we want to find a new \( i' < i \) such that \( P[0..i'] = T[j..j - i'] \).

  Specifically, we want to match the largest such \( i' \).

- This means that \( P[0..i'] = T[j - i'..j] \).

  In other words, \( P[0..i'] \) is the longest proper border of \( P[0..i] \).

**Example:** \( a \) is the longest proper border of \( aiai \).

- Thus \( fail[i] \) is the length of the longest proper border of \( P[0..i] \).

- \( P[0..i] = \varepsilon \) has no proper border. We set \( fail[0] = -1 \).
Example 2.5: Let \( P = \text{ainainen} \). 

| \( i \) | \( P[\ldots i) \) border \( \text{fail}[i] \) |
|---|---|---|
| 0 | \( \varepsilon \) | 0 |
| 1 | \( a \) | 0 |
| 2 | \( ai \) | 0 |
| 3 | \( ain \) | 0 |
| 4 | \( aina \) | 1 |
| 5 | \( ainai \) | 2 |
| 6 | \( ainaine \) | 3 |
| 7 | \( ainainen \) | 0 |
| 8 | \( aina \) | 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 \( \varepsilon \)-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:

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

Algorithm 2.6: Morris–Pratt failure function computation

Input: pattern \( P = P[0 \ldots m) \)
Output: array \( \text{fail}[m] \)

1. \( i \leftarrow -1 \)
2. While \( j < m \) do
3. \( (\text{if } i = -1 \text{ or } P[i] = P[j] \text{ then } i \leftarrow i+1 ; j \leftarrow j+1) \)
4. \( \text{else } i \leftarrow \text{fail}[i] \)
5. Return \( \text{fail} \)

When the algorithm reads \( \text{fail}[i] \) on line 4, \( \text{fail}[i] \) has already been computed.

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

When \( D \) 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. \( D \leftarrow (D << 1) \) \( \text{ (the bitwise shift and the bitwise or) \; Now } D \) tells
  2. \( \text{while } \exists \text{ prefixes would match if } T[j] \text{ would match every character.} \)

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

Example 2.9: \( P = \text{asssi} \), \( T = \text{apassii} \), bitvectors are columns.

<table>
<thead>
<tr>
<th>( \Sigma )</th>
<th>( a )</th>
<th>( i )</th>
<th>( p )</th>
<th>( s )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( s )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( s )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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

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 \( j \in [0..n-m) \) numerically. Before we compute the hash values of \( P \) 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 fingerprints for \( T[j..j+1..m] \) are computed from the fingerprint for \( T[j..j+m] \) in constant time using Lemma 1.44:

\[
H(T[j..j+1..m]) = (H(T[j..j+m]) - H(T[j..j+m-1])) \cdot r^{m-1} \mod q
\]

A hash function that supports this kind of sliding window computation is known as a rolling hash function.
Algorithm 2.10: Karp-Rabin
Input: text $T = T[0 \ldots n]$, pattern $P = P[0 \ldots m]$.
Output: position of the first occurrence of $P$ in $T$.

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

On an integer alphabet:
- The worst case time complexity is $O(mn)$.
- The average case time complexity is $O(m + n)$.

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

More precisely, suppose we are currently comparing $P$ against $T[j \ldots j + m]$. 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: $P = \text{ainainen}$.

<table>
<thead>
<tr>
<th>$c$</th>
<th>last occ.</th>
<th>$shift$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>ainainen</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>ainainen</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>ainainen</td>
<td>3</td>
</tr>
<tr>
<td>d</td>
<td>ainainen</td>
<td>2</td>
</tr>
<tr>
<td>e</td>
<td>ainainen</td>
<td>7</td>
</tr>
<tr>
<td>$\Sigma \setminus {a, b, c, d}$</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Horspool</th>
</tr>
</thead>
<tbody>
<tr>
<td>ainainen</td>
</tr>
<tr>
<td>ainainen</td>
</tr>
<tr>
<td>ainainen</td>
</tr>
<tr>
<td>ainainen</td>
</tr>
<tr>
<td>ainainen</td>
</tr>
<tr>
<td>ainainen</td>
</tr>
<tr>
<td>ainainen</td>
</tr>
</tbody>
</table>

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

Preprocess:
(1) for $c \in \Sigma$ do $shift[c] \leftarrow m$.
(2) for $i \leftarrow 0$ to $m - 2$ do $shift[P[i]] \leftarrow m - 1 - i$.

Search:
(3) $j \leftarrow 0$.
(4) while $j + m \leq n$ do
(5) if $P[m - 1] = T[j + m - 1]$ then
(6) $i \leftarrow m - 2$.
(7) while $i > 0$ and $P[i] = T[j + i]$ do $i \leftarrow i - 1$.
(8) if $i = -1$ then return $j$.
(9) $j \leftarrow j + shift[T[j + m - 1]]$.
(10) return $n$.

On an integer alphabet:
- Preprocessing time is $O(m + n)$.
- In the worst case, the search time is $O(mn)$.
  For example, $P = ba^{m-1}$ and $T = a^n$.
- In the best case, the search time is $O(n/m)$.
  For example, $P = b^n$ and $T = a^n$.
- In the average case, the search time is $O(n/m \cdot \min(m, \sigma))$.
  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.