Algorithm 2.15: 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 \( B[c] \leftarrow 0 \)
(2) for \( i \leftarrow 0 \) to \( m - 1 \) do \( B[P[m - 1 - i]] \leftarrow B[P[m - 1 - i]] + 2^i \)
Search:
(3) \( j \leftarrow 0 \)
(4) while \( j + m \leq n \) do
(5) \( i \leftarrow m \), \( shift \leftarrow m \)
(6) \( D \leftarrow D << 1 \)
(7) \( j \leftarrow j + shift \)
(8) while \( D \neq 0 \) do
(9) \( D \leftarrow D \& B[T[j]] \)
(10) if \( D \& 2^m-1 \neq 0 \) then
(11) // Now \( T[j + i \cdot j + m] \) is a pattern factor
(12) \( i \leftarrow i - 1 \)
(13) \( D \leftarrow D \& B[T[j]] \)
(14) if \( i = 0 \) then return \( j \)
(15) else \( shift \leftarrow i \)
(16) \( j \leftarrow j + shift \)
(17) return \( n \)

Example 2.16: \( P = as, T = asasasi \).
\( B[i], c \in \{ a, i, p, s \} \)
\( D \) when scanning \( as \) backwards
\( a \) \( i \) \( p \) \( s \) \( a \) \( s \) \( a \) \( s \) \( a \) \( = \) \( shift = 2 \)
\( a \) \( s \) \( a \) \( s \) \( a \) \( s \) \( a \) \( s \) \( a \) \( s \) \( a \) \( s \) \( a \) \( s \) \( a \) \( = \) \( occurrence \)

On an integer alphabet when \( m \leq w \):
- Preprocessing time is \( O(r + m) \).
- In the worst case, the search time is \( O(nm) \).
  - For example, \( P = a^m \cdot b \) and \( T = a^w \).
- In the best case, the search time is \( O(n/m) \).
  - For example, \( P = b^n \) and \( T = a^w \).
- In the average case, the search time is \( O(n(\log m)/m) \).
  - This is optimal. It has been proven that any algorithm needs to inspect \( \Omega(n(\log m)/m) \) text characters on average.

When \( m > w \), there are several options:
- Use multi-word bitvectors.
- Use BDM or BOM.

Karp–Rabin
The Karp–Rabin hash function (Definition 1.37) 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 \( T[j..j+1+m] \) is computed from the fingerprint for \( T[j..j+m] \) in constant time using Lemma 1.38:
\[
H(T[j..j+1+m]) = (H(T[j..j]) - H(T[j..j-1+m]) \cdot r^m) \cdot r + H(T[j..j+m]) \mod q
\]
A hash function that supports this kind of sliding window computation is known as a rolling hash function.

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

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

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

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

Algorithm 2.17: 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 \( n - 1 \) do
(4) \( hp \leftarrow (hp \cdot r + P[i]) \mod q \)
(5) \( ht \leftarrow (ht \cdot r + T[i]) \mod q \)
(6) \( if \ hp = ht \ then \ return \ j \)
(7) \( h tờ \lt (h tờ \cdot a) \cdot r + T[j+m] \mod q \)
(8) \( if \ hp = ht \ then \ return \ j \)
(9) return \( n \)

On an integer alphabet:
- The worst case time complexity is \( O(nm) \).
- 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).

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

Example 2.19: The periods of \( S_1 = abababaabababababa = 4, 7, 8 \) and \( S_2 = abcabaabcdabcabcaca = 3, 6, 9, 12, 13 \). \( S_3 \) is 3-periodic but \( S_2 \) is not.

There is a strong connection between periods and borders.

Lemma 2.20: \( p \) is a period of \( S[0..m) \) if and only if \( S \) has a proper border of length \( m-p \).
Proof. Both conditions hold if and only if \( S[0..m-p) = S[p..m) \).

Corollary 2.21: The length of the longest proper border of \( S \) is \( m - \text{per}(S) \).
Recall that $\text{fail}[\ell]$ in MP is the length of the longest proper border of $P[0..\ell)$. Thus the pattern shift by MP is $\ell - \text{fail}[\ell] = \text{per}(P[0..\ell))$ and the icp skip is $\text{fail}[\ell] = \ell - \text{per}(P[0..\ell))$. Thus knowing $\text{per}(P[0..\ell))$ is sufficient to compute MP shift and skip.

The Crochemore algorithm has two cases:

- If $P[0..\ell)$ is 3-periodic, then shift by $\ell / 3$.
- If $P[0..\ell)$ is not 3-periodic, then shift by $\lfloor \ell / 3 \rfloor$ and start the icp comparison from scratch.

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

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

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

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

- $\text{MS}(Sa) = Sa$ and $\text{per}(Sa) = p$ if $a = S[n - p]$.
- $\text{MS}(Sa) = Sa$ and $\text{per}(Sa) = m + 1$ if $a > S[n - p]$.
- $\text{MS}(Sa) \neq Sa$ if $a < S[n - p]$.

Furthermore, let $r = m \mod p$ and $R = S[m - r..m)$. Then $R$ is self-maximal and $\text{MS}(Sa) = \text{MS}(Ra)$ if $a < S[n - p]$.

The proof is omitted.

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

**Lemma 2.25:** Let $S[0..m)$ be a string and let $S[\ell, m) = \text{MS}(S)$ and $p = \text{per}(\text{MS}(S))$.

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

The algorithm is given on the next slide.

- Time complexity is $O(n)$. (Proof omitted.)
- It uses only a constant number of integer variables in addition to the strings $P$ and $T$.
- Works on ordered alphabet.

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

### Aho–Corasick

Given a text $T$ and a set $P = \{P_1, P_2, \ldots, P_k\}$ 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 $\text{trie}(P)$ as an automaton and augments it with a failure function similar to the Morris-Pratt failure function.

**Example 2.27:** Aho–Corasick automaton for $P = \{\text{ba, aba, biais, bers}\}$.

1. **Algorithm 2.24:** Update-MS($P[0..\ell), s, p$).

   **Input:** a string $P[0..\ell)$ and integers $s, p$, such that $M S(P[0..\ell]) = P[s..\ell]$ and $p = \text{per}(P[0..\ell))$.

   **Output:** a triple $(\ell + 1, s, p)$ such that $M S(P[0..\ell + 1)) = P[\ell + 1..\ell + p]$ and $p' = \text{per}(P'[0..\ell + 1))$.

   1. If $\ell = 0$ then return $(0, 1, 0)$.
   2. $i \leftarrow \ell$
   3. while $i < \ell + 1$ do
   4. if $P[i] \neq P[\ell]$ then
   5. $i \leftarrow i + s - (i - s \mod p)$
   6. $s \leftarrow 1$
   7. $p \leftarrow 1$
   8. else if $P[i] < P[\ell]$ then
   9. $i \leftarrow i + s + 1$
   10. $s \leftarrow \lfloor s / 2 \rfloor$
   11. return $(\ell + 1, s, p)$

   The proof is omitted.

### Algorithm 2.26: Crochemore

**Input:** position of the first occurrence of $P$ in $T$.

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

### Algorithm 2.28: Aho–Corasick

**Input:** text $T$, pattern set $P = \{P_1, P_2, \ldots, P_k\}$.

1. $S_i$ denotes the string that node $v$ represents.
2. $\text{root}(v)$ is the root and $\text{child}(v)$ is the function of the trie.
3. $\text{fail}(v) = u$ such that $S_i$ is the longest proper suffix of $S_i$ represented by any trie node $u$.
4. $\text{patterns}(v)$ is the set of pattern indices $i$ such that $P_i$ is a suffix of $S_i$.

Each node, the algorithm computes the node $v$ such that $S_i$ is the longest suffix of $T[0..j]$ represented by any node.

**Algorithm 2.29:** Construct-AC-Automaton

**Input:** pattern set $P = \{P_1, P_2, \ldots, P_k\}$.

1. Create new node $\text{root}$
2. $v \leftarrow \text{root}$
3. while $v \neq \text{root}$ do
4. while $\text{child}(v, P[j]) \neq \text{null}$ do
5. $v \leftarrow \text{child}(v, P[j])$
6. while $j < |P|$ do
7. Create new node $u$
8. $\text{child}(v, P[j]) = u$
9. $v \leftarrow u$
10. $j \leftarrow j + 1$
11. $\text{patterns}(v) \leftarrow \{j\}$
12. $\text{Create-AC-Automaton}(v)$

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

- Line (10) marks $v$ as a representative of $P_i$.
- The creation of a new node $v$ initializes $\text{patterns}(v)$ to $\emptyset$ (in addition to initializing $\text{child}(v, c)$ to $\text{null}$ for all $c \in \Sigma$).
Algorithm 2.30: Compute-AC-Fail

Input: AC trie: root, child() and patterns()
Output: AC failure function fail() and patterns()

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

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

Assuming \( \sigma \) is constant:

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

The analysis when \( \sigma \) is not constant is left as an exercise.

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

- The nodes that represent suffixes of \( Sv \) that are exactly \( \text{fail}^*(v) = \{ u | \text{fail}(u) = \text{fail}(\text{fail}(v)), \ldots, \text{root} \} \).
- Let \( u = \text{parent}(v) \) and child(\( u, c \)) = \( v \). Then \( Sv = Sv' \) and a string \( S \) is a suffix of \( S_v \) iff \( S_v \) is suffix of \( S_v' \). Thus for any node \( w \)
  - If \( w \in \text{fail}(v) \), then parent(\( \text{fail}(v) \)) \in \text{fail}^*(u).
  - If \( w \in \text{fail}(u) \) and child(\( w, c \)) \( \neq \) \( \perp \), then child(\( w, c \)) \in \text{fail}^*(v).
- Therefore, \( \text{fail}(v) = \text{child}(w, c) \), where \( w \) is the first node in \( \text{fail}(u) \) other than \( u \) such that child(\( w, c \)) \( \neq \) \( \perp \).

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

\[
\text{patterns}(v) = \{ i | P_i \text{ is a suffix of } Sv \} = \{ i | P_i = Sv' \text{ and } w \in \text{fail}(v) \} = \{ i | P_i = Sv' \text{ or } \text{patterns}(\text{fail}(v)) \}
\]

Summary: Exact String Matching

Exact string matching is a fundamental problem in stringology. We have seen several different algorithms for solving the problem. The properties of the algorithms vary with respect to worst case time complexity, average case time complexity, type of alphabet (ordered/integer) and even space complexity. The algorithms use a wide range of completely different techniques:

- There exists numerous algorithms for exact string matching but almost all are based on these techniques.
- Many of the techniques can be adapted to other problems. All of the techniques have some uses in practice too.