Algorithm 2.15: BNDM
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 $B[P[m-1-i]] \leftarrow B[P[m-1-i]] + 2^i$
Search:
(3) $j \leftarrow 0$
(4) while $j + m \leq n$ do
(5) $\quad i \leftarrow m$; $shift \leftarrow m$
(6) $\quad D \leftarrow 2^m - 1$ // $D \leftarrow 1^m$
(7) $\quad$ while $D \neq 0$ do
$\quad\quad$ // Now $T[j+i..j+m)$ is a pattern factor
(8) $\quad\quad i \leftarrow i - 1$
(9) $\quad\quad D \leftarrow D$ & $B[T[j+i]]$
(10) $\quad\quad$ if D & $2^{m-1} \neq 0$ then
$\quad\quad\quad$ // Now $T[j+i..j+m)$ is a pattern prefix
(11) $\quad\quad\quad$ if $i = 0$ then return j
(12) $\quad\quad\quad$ else $shift \leftarrow i$
(13) $\quad\quad D \leftarrow D << 1$
(14) $\quad j \leftarrow j + shift$
(15) return n

Example 2.16: $P =$ 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	1	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	1	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, $P =$ a^{m-1}b and $T =$ a^n.
- In the best case, the search time is $\mathcal{O}(n/m)$.
  For example, $P =$ b^m and $T =$ 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 we will not describe them here.

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 $T[j+1..j+1+m) = \alpha T[j+m]$ is computed from the fingerprint for $T[j..j+m) = T[j]\alpha$ in constant time using Lemma 1.38:

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

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

Algorithm 2.17: 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
(1) Choose q and r; $s \leftarrow r^{m-1} \bmod q$
(2) $hp \leftarrow 0$; $ht \leftarrow 0$
(3) for $i \leftarrow 0$ to $m-1$ do $hp \leftarrow (hp \cdot r + P[i]) \bmod q$ // $hp = H(P)$
(4) for $j \leftarrow 0$ to $m-1$ do $ht \leftarrow (ht \cdot r + T[j]) \bmod q$
(5) for $j \leftarrow 0$ to $n-m-1$ do
(6) $\quad$ if $hp = ht$ then if $P = T[j \dots j+m)$ then return j
(7) $\quad ht \leftarrow ((ht - T[j] \cdot s) \cdot r + T[j+m]) \bmod q$
(8) if $hp = ht$ then if $P = T[j \dots j+m)$ then return j
(9) return n

On an integer alphabet:

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

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

Crochemore

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

- When the pattern P is aligned against a text factor $T[j..j+m)$, they compute the longest common prefix $\ell = lcp(P, T[j..j+m))$ and report an occurrence if $\ell = m$. Otherwise, they shift the pattern forward.
- MP shifts the pattern forward by $\ell - fail[\ell]$ positions. In the next lcp computation, MP skips the first $fail[\ell]$ 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 inoptimal but always safe: no occurrence is missed.

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

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

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

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

There is a strong connection between periods and borders.

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

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

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

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

The Crochemore algorithm has two cases:

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

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

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

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

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

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

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

$$MS(Sa) = MS(Ra) \qquad \text{if } a < S[m-p]$$

The proof is omitted.

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

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

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


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 $S[s..m) = MS(S)$ and $p = per(MS(S))$.

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

The algorithm is given on the next slide.

- Time complexity is $\mathcal{O}(n)$. (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.

Algorithm 2.26: Crochemore
Input: strings $T[0..n)$ (text) and $P[0..m)$ (pattern).
Output: position of the first occurrence of P in T

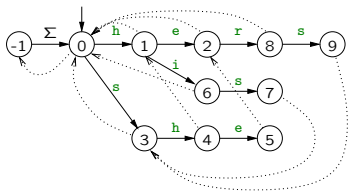
```
(1)  j ← ℓ ← p ← s ← 0
(2)  while j + m ≤ n do
(3)      while j + ℓ < n and ℓ < m and T[j + ℓ] = P[ℓ] do
(4)          (ℓ, s, p) ← Update-MS(P, ℓ, s, p)
         // ℓ = lcp(P, T[j..j + m))
(5)      if ℓ = m then return j
         // MS(P[0..ℓ)) = P[s..ℓ) and p = per(P[s..ℓ))
(6)      if p ≤ ℓ/3 and P[0..s) = P[p..p + s) then
             // per(P[0..ℓ)) = p
(7)          j ← j + p
(8)          ℓ ← ℓ − p
(9)      else    // per(P[0..ℓ)) > ℓ/3
(10)         j ← j + ⌊ℓ/3⌋ + 1
(11)         (ℓ, s, p) ← (0, 0, 0)
(12) return S
```


Aho–Corasick

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

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

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



Algorithm 2.28: Aho–Corasick
Input: text T, pattern set $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$.
Output: all pairs (i, j) such that P_i occurs in T ending at j.

```
(1)  (root, child(), fail(), patterns()) ← Construct-AC-Automaton(P)
(2)  v ← root
(3)  for j ← 0 to n − 1 do
(4)      while child(v, T[j]) = ⊥ do v ← fail(v)
(5)      v ← child(v, T[j])
(6)      for i ∈ patterns(v) do output (i, j)
```

Let S_v denote the string that node v represents.

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

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

```
(1)  Create new node root
(2)  for i ← 1 to k do
(3)      v ← root; j ← 0
(4)      while child(v, P_i[j]) ≠ ⊥ do
(5)          v ← child(v, P_i[j]); j ← j + 1
(6)      while j < |P_i| do
(7)          Create new node u
(8)          child(v, P_i[j]) ← u
(9)          v ← u; j ← j + 1
(10)     patterns(v) ← {i}
(11) (fail(), patterns()) ← Compute-AC-Fail(root, child(), patterns())
(12) return (root, child(), fail(), patterns())
```

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 *patterns*(v) to $\emptyset$ (in addition to initializing $\mathit{child}(v, c)$ to $\perp$ for all $c \in \Sigma$).

Algorithm 2.30: Compute-AC-Fail
Input: AC trie: $root$, *child*() and *patterns*()
Output: AC failure function *fail*() and updated *patterns*()

```
(1)  Create new node fallback
(2)  for c ∈ Σ do child(fallback, c) ← root
(3)  fail(root) ← fallback
(4)  queue ← {root}
(5)  while queue ≠ ∅ do
(6)      u ← popfront(queue)
(7)      for c ∈ Σ such that child(u, c) ≠ ⊥ do
(8)          v ← child(u, c)
(9)          w ← fail(u)
(10)         while child(w, c) = ⊥ do w ← fail(w)
(11)         fail(v) ← child(w, c)
(12)         patterns(v) ← patterns(v) ∪ patterns(fail(v))
(13)         pushback(queue, v)
(14) return (fail(), patterns())
```

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

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

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

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

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

Assuming σ is constant:

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

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

Summary: Exact String Matching

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

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

The algorithms use a wide range of completely different techniques:

- There exists numerous algorithms for exact string matching but almost all them 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.