Algorithm 2.10: 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; $s \leftarrow r^{m-1} \bmod q$
$hp \leftarrow 0; ht \leftarrow 0$
for $i \leftarrow 0$ **to** $m-1$ **do** $hp \leftarrow (hp \cdot r + P[i]) \bmod q$ // $hp = H(P)$
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** $P = T[j \dots j+m)$ **then** return j
  $ht \leftarrow ((ht - T[j] \cdot s) \cdot r + T[j+m]) \bmod q$
if $hp = ht$ **then if** $P = T[j \dots j+m)$ **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 $T = T[0 \dots n)$, pattern $P = P[0 \dots m)$
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.

BNDM

Starting the 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 2.14: Horspool shift

```
varmasti-aikaisen-ainainen
ainaisen-ainainen
  ainaisen-ainainen
```

Boyer–Moore shift

```
varmasti-aikaisen-ainainen
ainaisen-ainainen
         ainaisen-ainainen
```

Factor shift

```
varmasti-aikaisen-ainainen
ainaisen-ainainen
                 ainaisen-ainainen
```

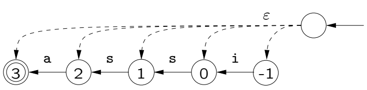

Factor based algorithms use an automaton that accepts suffixes of the reverse pattern P^R (or equivalently reverse prefixes of the pattern P).

- 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 2.15: $P =$ 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 $T[j+i..j+m)$:

- $D.k = 1$ if and only if there is a path from the initial state to state k with the string $(T[j+i..j+m))^R$, and thus $T[j+i..j+m) = P[m-k-1..2m-k-i-1)$.
- 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 after reading at least one character, so the shift brings the right bit to $D.0$.
- Before reading anything $D.(-1) = 1$. This exception is handled by starting the computation with the first shift already performed. Because of this, the shift is done at the end of the loop.

Algorithm 2.16: 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:
 for $c \in \Sigma$ do $B[c] \leftarrow 0$
 for $i \leftarrow 0$ to $m-1$ do $B[P[m-1-i]] \leftarrow B[P[m-1-i]] + 2^i$
Search:
 $j \leftarrow 0$
 while $j + m \leq n$ do
  $i \leftarrow m$; $shift \leftarrow m$
  $D \leftarrow 2^m - 1$ // $D \leftarrow 1^m$
  while $D \neq 0$ do
   // Now $T[j+i..j+m)$ is a pattern factor
   $i \leftarrow i - 1$
   $D \leftarrow D\ \&\ B[T[j+i]]$
   if $D\ \&\ 2^{m-1} \neq 0$ then
    // Now $T[j+i..j+m)$ is a pattern prefix
    if $i = 0$ then return j
    else $shift \leftarrow i$
   $D \leftarrow D << 1$
  $j \leftarrow j + shift$
 return n

Example 2.17: $P =$ 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

		s	a	p	a
i	1	0	0	0	
s	1	1	0	0	
s	1	1	0	0	
a	1	0	1	0	$\Rightarrow$ $shift = 2$

D when scanning apassi backwards

		i	s	s	a	p	a
i	1	1	0	0	0		
s	1	0	1	0	0		
s	1	0	0	1	0		
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, $P = \mathtt{a}^{m-1}\mathtt{b}$ 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 but not in the average case.
- There are also algorithms that are optimal in both cases. They are based on similar techniques, but we will not describe them here.

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 $\ell - \mathit{fail}[\ell]$ positions. In the next lcp computation, MP skips the first $\mathit{fail}[\ell]$ characters (cf. lcp-comparison).
- Crochemore either does the same shift and skip as MP, or a shorter shift than MP 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 $m \geq k \cdot per(S)$.

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 $\mathit{fail}[\ell]$ 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 $c \in \Sigma$,

$$\begin{array}{ll} MS(Sc) = Sc \text{ and } per(Sc) = p & \text{if } c = S[m-p] \\ MS(Sc) = Sc \text{ and } per(Sc) = m+1 & \text{if } c < S[m-p] \\ MS(Sc) \neq Sc & \text{if } c > S[m-p] \end{array}$$

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

$$MS(Sc) = MS(Rc) \qquad \text{if } c > S[m-p]$$

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 $p' = per(P[s'..\ell+1))$.

if $\ell = 0$ then return $(1, 0, 1)$
$i \leftarrow \ell$
while $i < \ell + 1$ do
  // $P[s..i)$ is self-maximal and $p = per(P[s..i))$
  if $P[i] > P[i-p]$ then
    $i \leftarrow i - ((i-s) \bmod p)$
    $s \leftarrow i$
    $p \leftarrow 1$
  else if $P[i] < P[i-p]$ then
    $p \leftarrow i - s + 1$
  $i \leftarrow i + 1$
return $(\ell + 1, s, p)$

As the final piece of the Crochemore algorithm, the following result shows 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.

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

$j \leftarrow \ell \leftarrow p \leftarrow s \leftarrow 0$
while $j + m \leq n$ do
  while $j + \ell < n$ and $\ell < m$ and $T[j+\ell] = P[\ell]$ do
    $(\ell, s, p) \leftarrow$ Update-MS(P, ℓ, s, p)
  // $\ell = lcp(P, T[j..j+m))$
  if $\ell = m$ then return j
  // $MS(P[0..\ell)) = P[s..\ell)$ and $p = per(P[s..\ell))$
  if $p \leq \ell/3$ and $P[0..s) = P[p..p+s)$ then
    // $per(P[0..\ell)) = p$
    $j \leftarrow j + p$
    $\ell \leftarrow \ell - p$
  else // $per(P[0..\ell)) > \ell/3$
    $j \leftarrow j + \lfloor \ell/3 \rfloor + 1$
    $(\ell, s, p) \leftarrow (0, 0, 0)$
return n

For ordered alphabet:

- The time complexity is $\mathcal{O}(n)$.
- The algorithm uses only a constant number of integer variables in addition to the strings P and T.

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.

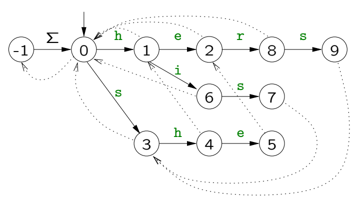
There are also other linear time, constant extra space algorithms. All of them are based on string periodicity in some way.

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 $\mathcal{P} = \{\texttt{he}, \texttt{she}, \texttt{his}, \texttt{hers}\}$.



Let S_v denote the string represented by a node v in the trie. The components of the AC automaton are:

- $root$ is the root and *child*() the child function of the trie.
- *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.

Example 2.28: For the automaton in Example 2.27, *patterns*$(2) = \{1\}$ ($\{\texttt{he}\}$), *patterns*$(5) = \{1, 2\}$ ($\{\texttt{he}, \texttt{she}\}$), *patterns*$(7) = \{3\}$ ($\{\texttt{his}\}$), *patterns*$(9) = \{4\}$ ($\{\texttt{hers}\}$), and *patterns*$(v) = \emptyset$ for all other nodes v.

At each stage of the matching, 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: 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.

$(root$, *child*(), *fail*(), *patterns*()$) \leftarrow$ Construct-AC-Automaton($\mathcal{P}$)
$v \leftarrow root$
for $j \leftarrow 0$ to $n - 1$ do
  while *child*$(v, T[j]) = \perp$ do $v \leftarrow$ *fail*(v)
  $v \leftarrow$ *child*$(v, T[j])$
  for $i \in$ *patterns*(v) do output (i, j)

The construction of the automaton is done in two phases: the trie construction and the failure links computation.

Algorithm 2.30: Construct-AC-Automaton
Input: pattern set $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$.
Output: AC automaton: $root$, *child*(), *fail*() and *patterns*().

$(root$, *child*(), *patterns*()$) \leftarrow$ Construct-AC-Trie($\mathcal{P}$)
(*fail*(), *patterns*()$) \leftarrow$ Compute-AC-Fail($root$, *child*(), *patterns*())
return $(root$, *child*(), *fail*(), *patterns*()$)$

Algorithm 2.31: Construct-AC-Trie
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 ← 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) return (root, child(), patterns())
```

Lines (3)–(10) perform 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 *child*(v, c) to $\perp$ for all $c \in \Sigma$).

Algorithm 2.32: 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):

- Let *fail*$^*(v) = \{v, \mathit{fail}(v), \mathit{fail}(\mathit{fail}(v)), \ldots, root\}$. These nodes are exactly the trie nodes that represent suffixes of S_v.
- 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 $w \in \mathit{fail}^*(v) \setminus \{root\}$, then $\mathit{parent}(w) \in \mathit{fail}^*(u)$.
  - If $w \in \mathit{fail}^*(u)$ and $\mathit{child}(w, c) \neq \perp$, then $\mathit{child}(w, c) \in \mathit{fail}^*(v)$.
- Therefore, $\mathit{fail}(v) = \mathit{child}(w, c)$, where w is the first node in *fail*$^*(u)$ other than u such that $\mathit{child}(w, c) \neq \perp$, or $\mathit{fail}(v) = root$ if no such node w exists.

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 $m = ||\mathcal{P}||$.
  - The 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 most of them use variations or combinations of the techniques we have seen (study groups).
- Many of the techniques can be adapted to other problems. All of the techniques have some uses in practice.