Algorithm 2.10: Karp-Rabin
Input: text T = T[0...n), pattern P = P[0...m)
Output: position of the first occurrence of P in T

(1) Choose q and r; s ← r·m mod q
(2) hp ← 0; ht ← 0
(3) for i ← 0 to m − 1 do hp ← (hp · r + P[i]) mod q // hp = H(P)
(4) for j ← 0 to m − 1 do ht ← (ht · r + T[j]) mod q
(5) if j < m then if P[j] = T[i + j] then return j
(6) if hp = ht then if P[j..j+m) = T[i..i+m) then return j
(7) return n

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

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..j+m).
Start by comparing P[0..m) to T[0..k], where k = m − 1.

• If P[m−1] = T[k], shift the pattern until the pattern character aligned with
the last character of P is past the current character of T.
• 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 pattern character.

The length of the shift is determined by the shift table that is precomputed
for the pattern. shift[c] is defined for all c ∈ Σ:

• 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.

<table>
<thead>
<tr>
<th>c</th>
<th>last occ.</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>n</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>i</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>n</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>i</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>n</td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

On an integer alphabet:
• Preprocessing time is O(σ+ m).
• In the worst case, the search time is O(mn).
For example, P = ainainen, ainainen, ainainen.

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

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

Preprocess:
(1) for c ∈ Σ do shift[c] ← m
(2) for i ← 0 to m − 2 do shift[P[i]] ← m − 1 − i
Search:
(3) j ← 0
(4) while j + m ≤ n do
(5) if P[m−1] = T[j..j+m] then
(6) i ← m − 2
(7) while i ≥ 0 and P[i] = T[j+i] do i ← i − 1
(8) if j = i then return j
(9) j ← j + shift[T[j+m−1]]
(10) return n

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
variasti-aiainen-ainainen
ainainen-ainainen
ainainen-ainainen
ainainsen-ainainen
ainainsen-ainainen
ainainsen-ainainen
ainainsen-ainainen
ainainsen-ainainen

Boyer–Moore shift
Factor shift
variasti-aiainen-ainainen
ainainen-ainainen
ainainsen-ainainen
ainainsen-ainainen
ainainsen-ainainen
ainainsen-ainainen
ainainsen-ainainen
ainainsen-ainainen

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 ← j + shift.

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 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
ainaine/ n
ainaine/ n
ainainen

Example 2.15: P = assi.

BOM (Backward Oracle Matching) uses a much simpler deterministic
automaton that accepts all suffixes of Pm but may also accept some
other strings. This can cause shorter shifts but incorrect behaviour.
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 \(T[i..j+m]\):

- \(D.k = 1\) if and only if there is a path from the initial state to state \(k\) with the string \((T[i..j+m], i+m)\)
- \(T[i..j+m] = P[m-k-1..2m-k-1]\)
- If \(D(m-1) = 1\), then \(T[i..j+m]\) 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 \(P[m - 1 - i] = P^m[i] = c\)

The update when reading \(T[i..j]\) is familiar: \(D = \{D < < 1\} \& B[T[i..j]]\)

- Note that there is no ‘\(\cdots 1\)”. This is because \(D(\cdots 1) = 0\) always after reading at least one character, so the shift brings the right bit to \(D.0\).
- Before reading anything \(D(\cdots 1) = 1\). 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 \(T = T[0..n]\), pattern \(P = P[0..m]\)
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. \(\text{shift} \leftarrow m\)
6. \(D \leftarrow 2^m - 1\)
7. while \(D \neq 0\) do
   - if \(N_{T}[i..j+m]\) is a pattern factor
     - \(i \leftarrow i + 1\)
   - else \(\text{shift} \leftarrow i\)
   - \(D \leftarrow D \& B[T[i..j]]\)
8. if \(D \& 2^{m-1} \neq 0\) then \(D \leftarrow D < < 1\)
9. \(D \leftarrow D - c\)
10. \(j \leftarrow j + \text{shift}\)
11. return \(n\)

On an integer alphabet when \(m \leq w\):

- Preprocessing time is \(O(\sigma + m)\).
- In the worst case, the search time is \(O(mn)\).
- For example, \(P = a^n b^n\) and \(T = a^n\).
- In the best case, the search time is \(O(n/m)\).
- For example, \(P = a^n\) and \(T = a^n\).
- In the average case, the search time is \(O(n(\log_2 m)/m)\).

This is optimal! It has been proven that any algorithm needs to inspect \(\Omega(n(\log_2 m)/m)\) 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.

**Crochemore**

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

- When the pattern \(P\) is aligned against a text factor \(T[i..j+m]\), they compute the longest common prefix \(\ell = \text{lcp}(P, T[i..j+m])\) and report an occurrence if \(\ell = m\). Otherwise, they shift the pattern forward.
- MP shifts the pattern forward by \(\ell - \text{fail}[\ell]\) positions. In the next lcp computation, MP skips the first \(\text{fail}[\ell]\) 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 \(S[i..i+p] = S[i..i+p] \forall i \in [0..m-p]\). The smallest period of \(S\) is denoted \(\text{per}(S)\). \(S\) is \(k\)-periodic if \(m \geq k \cdot \text{per}(S)\).

**Example 2.19:** The periods of \(S_1 = \text{asabaabaa}\) are 4, 7, and 9. The periods of \(S_2 = aabacabaca\) 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 \(m - \text{per}(S)\).
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 $p = \perm(S)$. For any $c \in \Sigma$, $MS(S_c) = S_c$ and $\perm(S_c) = p$ if $c = S[m - p] \backslash S[m - p]$.

Furthermore, let $r = m \mod p$ and $R = S[m - r..m)$. Then $R$ is self-maximal and $MS(S_c) = MS(R_c)$ if $c > S[m - p]$.

Moreover, let $v = \max(\{v \mid S[v] \in \perm(S)\})$ and $v = \max(\{v \mid S[v] \in \perm(S)\})$.

Algorithm 2.24: Update-MS($P, \ell, s, p$).
Input: a string $P$ and integers $\ell, s, p$ such that $MS(P[0..\ell)) = P[s..\ell)$ and $p = \perm(P[s..\ell))$.

Output: a tuple $((\ell + 1, \ell + 1))$ such that $MS(P[0..\ell + 1)) = P[s..\ell + 1)$ and $p = \perm(P[s..\ell + 1))$.

(1) If $\ell = 0$ then return $(0, 0)$.
(2) $i \leftarrow \ell$
(3) While $i < \ell + 1$ do
    (4) // If $P[v] > P[i]$ then
        (5) \( i \leftarrow i + 1 \)
        (6) $s \leftarrow i$
        (7) $p \leftarrow 1$
    (8) Else if $P[v] < P[i]$ then
        (9) $p \leftarrow i + 1$
    (10) $i \leftarrow i + 1$
(11) return $(\ell + 1, \ell + 1)$

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) $j \leftarrow \ell \leftarrow s \leftarrow 0$
(2) While $j + m \leq n$ do
    (3) While $j + \ell < n$ and $\ell < m$ and $T[j + \ell] = P[\ell]$ do
        (4) // $t = \max(\{t \mid S[t] \in \perm(S)\})$
            (5) // $P[t] = \max(\{t \mid S[t] \in \perm(S)\})$
        (6) If $p \leq \ell / 3$ and $P[0..\ell] = P[p..p + s)$ then
            (7) $j \leftarrow j + p$
            (8) $\ell \leftarrow \ell - p$
            (9) Else if $P[0..\ell] > \ell / 3$ then
                (10) $j \leftarrow j + \ell / 3 + 1$
                (11) // $P[0..\ell] = P[0..\ell]$
                (12) return $n$

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 $P = \{ba, she, bis, bers\}$.

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 $P = \{P_1, P_2, \ldots, P_k\}$.
Output: all pairs $(i, j)$ such that $P$ occurs in $T$ ending at $j$.

(1) $(\text{root, child}(v), \text{fail}(v), \text{patterns}(v)) \leftarrow \text{Construct-AC-Automaton}(P)$
(2) $v \leftarrow \text{root}$
(3) For $j \leftarrow 0$ to $n - 1$ do
    (4) While $\text{child}(v, T[j]) = \perp$ do $v \leftarrow \text{fail}(v)$
    (5) $v \leftarrow \text{child}(v, T[j])$
    (6) For $i \in \text{patterns}(v)$ 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 $P = \{P_1, P_2, \ldots, P_k\}$.
Output: AC automaton $(\text{root, child}, \text{fail}(v)$ and $\text{patterns}(v)$).

(1) $(\text{root, child}(v), \text{patterns}(v)) \leftarrow \text{Construct-AC-Trie}(P)$
(2) $(\text{fail}(v), \text{patterns}(v)) \leftarrow \text{Compute-AC-Fail(root, child), patterns})$
(3) return $(\text{root, child}, \text{fail}(v), \text{patterns}(v))$.
Algorithm 2.31: Construct-AC-Trie
Input: pattern set \( P = \{P_1, P_2, \ldots, P_k\} \).
Output: AC trie: root, child() and patterns().

1. Create new node root
2. for \( i = 1 \) to \( k \) do
   3. \( v \leftarrow \text{root}; \ j \leftarrow 0 \)
   4. while \( \text{child}(v, P[j]) \neq \bot \) do
      5. \( v \leftarrow \text{child}(v, P[j]); \ j \leftarrow j + 1 \)
   6. while \( j < |P_i| \) do
      7. Create new node \( u \)
      8. \( \text{child}(v, P[j]) \leftarrow u \)
      9. \( v \leftarrow w; \ j \leftarrow j + 1 \)
   10. \( \text{patterns}(v) \leftarrow \{i\} \)
11. return (root, child(), patterns())

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

- The creation of a new node \( v \) initializes \( \text{patterns}(v) \) to \( \emptyset \) (in addition to initializing \( \text{child}(v, c) \) to \( \bot \) for all \( c \in \Sigma \)).

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

- Let \( \text{fail}^*(v) = \{v, \text{fail}(v), \text{fail}^*(v), \ldots, \text{root}\} \). These nodes are exactly the trie nodes that represent suffixes of \( S_v \).
- Let \( u = \text{parent}(v) \) and \( \text{child}(u, c) = v \). Then \( S_v = S_vc \) and a string \( S \) is a suffix of \( S_v \) iff \( S \) is suffix of \( S_v \). Thus for any node \( w \):
  - if \( w \in \text{fail}^*(v) \setminus \{\text{root}\} \), then \( \text{parent}(w) \in \text{fail}(u) \).
  - if \( w \in \text{fail}(u) \) and \( \text{child}(u, c) \neq \bot \), then \( \text{child}(w, c) \in \text{fail}(v) \).

Therefore, \( \text{fail}(v) = \text{child}(w, c) \), where \( w \) is the first node in \( \text{fail}(u) \) other than \( u \) such that \( \text{child}(w, c) \neq \bot \), or \( \text{fail}(v) = \text{root} \) if no such node \( w \) exists.

\( \text{patterns}(v) \) is correctly computed on line (12):

\[
\text{patterns}(v) = \{i \mid P_i \text{ is a suffix of } S_v\}
= \{i \mid P_i = S_v \text{ and } w \in \text{fail}(v)\}
= \{i \mid P_i = S_v \} \cup \text{patterns}(%(\text{fail}(v)))
\]

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 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.