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. \( shft[c] \) is defined for all \( c \in \Sigma \):

- If \( c \) does not occur in \( P \), \( shft[c] = m \).
- Otherwise, \( shft[c] = m - 1 - i \), 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>e</td>
<td>ainainen</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>ainainen</td>
<td>3</td>
</tr>
<tr>
<td>n</td>
<td>ainainen</td>
<td>2</td>
</tr>
<tr>
<td>Σ {a,e,i,n}</td>
<td>—</td>
<td>8</td>
</tr>
</tbody>
</table>

In the integer alphabet model:

- Preprocessing time is \( O(\sigma + m) \).
- In the worst case, the search time is \( O(nm) \).
  - For example, \( P = ba^n1 \) and \( T = a^n \).
- In the best case, the search time is \( O(n/m) \).
  - For example, \( P = ba^n \) and \( T = a^n \).
- In the average case, the search time is \( O(n/\min(m,n)) \).
  - 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.

**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 \( c \in \Sigma \) do \( shft[c] \leftarrow m \)
2. for \( i \leftarrow 0 \) to \( m-2 \) do \( shft[P[i]] \leftarrow m-1-i \)

**Search:**
1. \( j \leftarrow 0 \)
2. while \( j + m \leq n \) do
3. \( i \leftarrow m \)
4. \( i \leftarrow i - 1 \)
5. \( D \leftarrow 2^m - 1 \) if \( D \neq 0 \) do
6. \( D \leftarrow D \) \& \( B[T[i+j+1]] \)
7. \( j \leftarrow j + D \)
8. if \( i = 0 \) then return \( j \)
9. \( j \leftarrow j + shft \)
10. return \( j \)

**BDNM**

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-ai/ kaisen-ainainen | ainaisen-ainainen | ainaisen-ainainen |
| varmasti-ai/ kaisen-ainainen | ainaisen-ainainen | ainaisen-ainainen |

**Example 2.15:** \( P = \text{assisi} \).

**BOM (Backward Oracle Matching):**

- A much simpler deterministic automaton that accepts all suffixes of \( P \), but may also accept some other strings. This can cause shorter shifts but not incorrect behaviour.

**Algorithm 2.16:** BDNM

**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 \( B[P[i]] \leftarrow m-1-i \) \& \( B[P[m-1-i]] \)

**Search:**
1. \( j \leftarrow 0 \)
2. while \( j + m \leq n \) do
3. \( i \leftarrow m \)
4. \( i \leftarrow i - 1 \)
5. if \( i = 0 \) then return \( j \)
6. \( j \leftarrow j + D \)
7. \( D \leftarrow D \) \& \( B[T[j+i]] \)
8. if \( D \neq 0 \) do
9. \( D \leftarrow D \) \& \( B[T[j+i]] \)
10. if \( D \neq 2^m-1 \) then
11. if \( i = 0 \) then return \( j \)
12. \( j \leftarrow j + D \)
13. \( D \leftarrow D \) \& \( B[T[j+i]] \)
14. if \( D \neq 0 \) then return \( j \)
15. return \( j \)
Example 2.17: \( P = \text{assi}, T = \text{apassi} \).

\[
\begin{array}{c|c|c|c|c}
B[c], c \in \{a, i, p, s\} & a & i & p & s \\
\hline
s & 0 & 1 & 0 & 0 \\
\hline
i & 0 & 1 & 0 & 0 \\
\hline
p & 0 & 0 & 0 & 1 \\
\hline
s & 0 & 0 & 0 & 1 \\
\hline
a & 1 & 0 & 0 & 1 \\
\hline
\end{array}
\]

\( D \) when scanning \( \text{apassi} \) backwards:

\[
\begin{array}{c|c|c|c|c|c}
\text{i s s a p a} & 1 & 1 & 0 & 0 & 0 \\
\hline
\text{s a p a} & 1 & 0 & 1 & 0 & 0 \\
\hline
\text{s a p a} & 1 & 0 & 0 & 1 & 0 \\
\hline
\text{a} & 1 & 0 & 0 & 1 & 1 & \Rightarrow \text{occurrence}
\end{array}
\]

\( \Rightarrow \) occurrence 97

\[
\text{D when scanning \text{apassi} backwards}
\]

\[
\begin{array}{c|c|c|c|c|c}
1 & 1 & 1 & 0 & 0 & 0 \\
\hline
s & 0 & 0 & 0 & 1 \\
\hline
i & 0 & 1 & 0 & 0 \\
\hline
a & 1 & 0 & 0 & 0 \\
\hline
\end{array}
\]

\( \Rightarrow \) occurrence

In the integer alphabet model when \( m \leq w \):

- Preprocessing time is \( O(k + m) \).
- In the worst case, the search time is \( O(mn) \).
- In the best case, the search time is \( O(n/m) \).

For example, \( P = a^{2m-1}10 \) and \( T = a^3 \).

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**

- Two of the exact string matching algorithms we have seen, the brute force algorithm and the Karp–Rabin algorithm, use only a constant amount of extra space in addition to the text and the pattern. All the other algorithms use some additional data structures whose size is proportional to the length of the pattern or the alphabet size.

- On the other hand, the brute force algorithm and the Karp–Rabin algorithm have a worst case running time of \( O(mn) \).

- Remarkably, there exists algorithms with \( O(n) \) worst case time that need only a constant amount of extra space. One of them is the Crochemore algorithm.

- We will only outline the main ideas of the algorithm here without detailed proofs.

All the linear time, constant extra space algorithms are based on the periodicity properties of the pattern.

**Definition 2.18:** Let \( S[.m] \) 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 \( m \geq k \cdot \text{per}(S) \).

**Example 2.19:** The periods of \( S_1 = \text{abababaa} \) are \( 4, 7, 8 \) and \( 9 \). The periods of \( S_2 = \text{abababcabaca} \) 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 \( S[.m] \) if and only if \( S[0..m - p] = S[p..m] \).

**Proof.** Both conditions hold if and only if \( S[0..m - p] = S[p..m] \). \( \Box \)

**Corollary 2.21:** The length of the longest proper border of \( S \) is \( m - \text{per}(S) \).

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 \( \ell = \text{lcp}(P[.\ell], T[j..j + m]) \) and report an occurrence if \( \ell = m \). Otherwise, they shift the pattern forward.

- MP shifts the pattern forward by \( \ell - \text{fail}(\ell) \) positions. 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 \( \text{per}(P[0..\ell]) \).

- In the next lcp-computation, MP skips the first \( \text{fail}(\ell) = \ell - \text{per}(P[0..\ell]) \) characters (cf. lcp-comparison).

- Thus knowing \( \text{per}(P[0..\ell]) \) is sufficient to emulate MP shift and skip.

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

**Example 2.23:** \( MS(\text{abababaa}) = \text{baabaa} \) and \( MS(\text{abababcabaca}) = \text{cabcabcabaca} \).

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

**Lemma 2.24:** Let \( S[.m] \) be a self-maximal string and let \( p = \text{per}(S) \). For any \( e \in \Sigma \),

\[
\begin{align*}
MS(Se) &= Sc \quad \text{if } e = S[m - p] \\
MS(Sc) &= Se \quad \text{if } e < S[m - p] \\
MS(Se) &\neq Sc \quad \text{if } e > S[m - p]
\end{align*}
\]

Furthermore, let \( r = m \mod p \) and \( R = S[m - r..m] \). Then \( R \) is self-maximal and

\[
MS(Sc) = MS(Rc) \quad \text{if } e > S[m - p]
\]
Crochemore’s algorithm computes the maximal suffix and its period for \( P[0..\ell) \) incrementally using Lemma 2.24. The following algorithm updates the maximal suffix information when the match is extended by one character.

**Algorithm 2.25:** Update-MS\((P, i, s, p)\)
Input: a string \( P \) and integers \( i, s, p \) such that \( MS(P[0..i)) = P[s..i) \) and \( p = \text{per}(P[s..i)) \).
Output: a triple \((\ell + 1, s, p)\) such that
\[
MS(P[0..\ell + 1)) = P[s..\ell + 1) \) and \( p = \text{per}(P[s..\ell + 1)) \).
(1) \( \ell = 0 \) then return \((1, 0, 1)\).
(2) \( i \leftarrow \ell \).
(3) while \( i < \ell + 1 \) do
   (4) if \( P[i] > P[\ell] \) then
      (5) \( i \leftarrow i - (i - \ell) \mod p \).
      (6) \( s \leftarrow s - 1 \).
      (7) \( p \leftarrow 1 \).
   (8) else if \( P[i] < P[\ell] \) then
      (9) \( p \leftarrow i - s + 1 \).
      (10) \( i \leftarrow i + 1 \).
   (11) return \((\ell + 1, s, p)\).

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

**Example 2.29:** Aho–Corasick automaton for \( P = \{ba, aba, his, hers\} \).

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 2.31:** Aho–Corasick
Input: text \( T \) and a set \( P = \{P_1, P_2, \ldots, P_h\} \) 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.

**Algorithm 2.32:** Construct-AC-Automaton
Input: pattern set \( P = \{P_1, P_2, \ldots, P_h\} \).
Output: AC automaton: \((\text{root}, \text{child}(\cdot), \text{fail}(\cdot), \text{patterns}())\).
(1) \( \text{root} \leftarrow \text{root}(\cdot), \text{child}(\cdot), \text{fail}(\cdot), \text{patterns}()) \).
(2) \( \text{fail}(\cdot), \text{patterns}() \) ← Compute-AC-Fail\((\text{root}, \text{child}(\cdot), \text{patterns}())\).
(3) return \((\text{root}, \text{child}(\cdot), \text{fail}(\cdot), \text{patterns}())\).

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 2.26:** Let \( S[0..m) \) be a string and \( S[0..s) = MS(S) \) and \( p = \text{per}(MS(S)) \).
- \( S \) 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 \( \text{per}(S) = p \).

**Example 2.27:**
- \( S[0..9) = \text{aababaaba} \): \( MS(S[0..9)) = S[3..9) = \text{baabaab} \), \( s = 3 \), \( p = \text{per}(baabaab) = 4 \leq 9/3 \), and thus \( S[0..9) \) is not 3-periodic.
- \( S[0..9) = \text{baabababbbaaab} \): \( MS(S[0..9)) = S[4..9) = \text{baabaab} \), \( s = 4 \), \( p = \text{per}(baabaab) = 4 \leq 9/3 \), \( S[0..4) = \text{baab} \neq \text{baab} = S[3..7) \), and thus \( S[0..9) \) is not 3-periodic.
- \( S[1..13) = \text{abcabcabca}\): \( MS(S[1..13) = S[3..13) = \text{abcabcabca} \), \( s = 3 \), \( p = \text{per}(abcabcabca) = 3 \), \( S[0..3) = \text{abc} \neq S[3..6) \), and thus \( S[1..13) \) is 3-periodic and \( \text{per}(S[1..13)) = p = 3 \).

In the general alphabet model:
- The time complexity is \( O(n) \).
- 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.

Let \( S_v \) denote the string represented by a node \( v \) in the trie. The components of the AC automaton are:
- \( \text{root} \) is the root and \( \text{child}(\cdot) \) the child function of the trie.
- \( \text{fail}(\cdot) \) is such that \( S_v \) is the longest proper suffix of \( S_v \) represented by any trie node \( u \).
- \( \text{patterns}(\cdot) \) is the set of pattern indices \( i \) such that \( P_i \) is a suffix of \( S_v \).

**Algorithm 2.33:** Construct-AC-Trie
Input: pattern set \( P = \{P_1, P_2, \ldots, P_h\} \).
Output: AC trie: \((\text{root}, \text{child}(\cdot), \text{fail}(\cdot), \text{patterns}(\cdot))\).
(1) Create new node \( \text{root} \).
(2) for \( v \leftarrow 1 \) 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 \( j < |P| \) do
      (7) Create new node \( u \).
      (8) \( \text{child}(v, P[j]) \leftarrow u \).
      (9) \( v \leftarrow v; j \leftarrow j + 1 \).
(10) \text{patterns}(v) \leftarrow \{i\}.
(11) return \((\text{root}, \text{child}(\cdot), \text{fail}(\cdot), \text{patterns}(\cdot))\).

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 \( \text{patterns}(v) \) to \( \emptyset \).

As an alternative to initializing \( \text{child}(v, c) \) to \( \bot \) for all \( c \in \Sigma \).
Algorithm 2.34: Compute-AC-Fail
Input: AC trie: root, child() and patterns()
Output: AC failure function fail() and patterns()

1. Create new node failback
2. for \( c \in \Sigma \) do child(failback, \( c \)) \( \leftarrow \) root
3. fail(root) \( \leftarrow \) failback
4. queue \( \leftarrow \) {root}
5. while queue \( \neq \emptyset \) do
   (6) \( u \leftarrow \text{popfront}(\text{queue}) \)
   (7) for \( c \in \Sigma \) such that child(\( u \), \( c \)) \( \neq \emptyset \) do
      \( v \leftarrow \text{child}(u, c) \)
      \( w \leftarrow \text{fail}(u) \)
      while child(\( w \), \( c \)) \( = \emptyset \) do \( w \leftarrow \text{fail}(w) \)
      fail(\( v \)) \( \leftarrow \) child(\( w \), \( c \))
      patterns(\( v \)) \( \leftarrow \) patterns(\( v \)) \( \cup \) patterns(\( \text{fail}(w) \))
   (11) pushback(queue, \( v \))
   (12) \( \text{return} \) (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.

In the constant alphabet model:
- The search time is \( O(n) \).
- The space complexity is \( O(m) \), where \( m = ||P|| \).
- The implementation of patterns() requires care (exercise).
- The preprocessing time is \( O(m) \), where \( m = ||P|| \).
- The only non-trivial issue is the whole-loop on line (10) in Compute-AC-Fail.
- Let root, \( v_2 \), \( v_3 \), ..., \( v_l \) be the nodes on the path from root to a node representing a pattern \( P \).
- Let \( w_j = \text{fail}(v_j) \) for all \( j \).
- Let depth(\( v \)) be the depth of a node \( v \) (depth(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(\( w_j \)) \( \leq \) depth(\( w_{j-1} \)) \( + \) 1 \(- t_j \) after line (10), where \( t_j \) is the number of rounds in the whole-loop.
- Thus, the total number of rounds in the whole-loop when processing the nodes \( v_2 \), \( v_3 \), ..., \( v_l \) is at most \( l = ||P|| \), and thus over the whole algorithm at most \( ||P|| \).

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, alphabet model, and even space complexity.

The algorithms use a wide range of completely different techniques:
- There exists numerous algorithms for exact string matching (see study group) but most of them use variations or combinations of the techniques we have seen.
- Many of the techniques can be adapted to other problems. All of the techniques have some uses in practice.

Selected Literature

- Survey
- Knuth–Morris–Pratt
- Shift-Or / Shift-And