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 text character aligned with the last pattern character. If it doesn't match, move (shift) the pattern forward until there is a match.

Example 2.10: 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 brute force manner. Then shift to the next position, where $T[k]$ matches.

Algorithm 2.11: Horspool
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 $shift[c] \leftarrow m$
  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

The length of the shift is determined by the shift table. $shift[c]$ 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

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 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 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 2.13: Horspool shift

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

Boyer–Moore shift

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

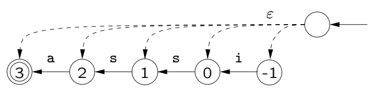
Factor shift

```
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 2.14: $P =$ `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 $T[j+i..j+m)$:

- $D.i = 1$ if and only if there is a path from the initial state to state i with the string $(T[j+i..j+m))^R$.
- 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, so the shift brings the right bit to $D.0$. With Shift-And $D.(-1) = 1$ always.
- The exception is that in the beginning before reading anything $D.(-1) = 1$. This 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 2.15: 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) $\quad i \leftarrow m$; $shift \leftarrow m$
(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	<u>1</u>	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	<u>1</u>	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.
- There are also algorithms that are optimal in both cases. They are based on similar techniques, but we will not describe them here.