Theorem 3.7: Algorithms MP and KMP preprocess a pattern in time $\mathcal{O}(m)$ and then search the text in time $\mathcal{O}(n)$.

Proof. It is sufficient to count the number of comparisons that the algorithms make. After each comparison $P[i] = T[j]$ (or $P[i] = P[j]$), one of the two conditional branches is executed:

then Here j is incremented. Since j never decreases, this branch can be taken at most $n+1$ $(m+1)$ times.

else Here i decreases since $fail[i] < i$. Since i only increases in the then-branch, this branch cannot be taken more often than the then-branch.

□

Shift-And

When the (K)MP algorithm is at position j in the text T, it computes the longest prefix of the pattern $P[0..m)$ that is a suffix of $T[0..j]$. The *Shift-And* algorithm computes *all* prefixes of P that are suffixes of $T[0..j]$.

- The information is stored in a bitvector D of length m, where $D.i = 1$ if $P[0..i] = T[j-i..j]$ and $D.i = 0$ otherwise. ($D.0$ is the least significant bit.)
- When $D.(m-1) = 1$, we have found an occurrence.

The bitvector D is updated at each text position j:

- There are precomputed bitvectors $B[c]$, for all $c \in \Sigma$, where $B[c].i = 1$ if $P[i] = c$ and $B[c].i = 0$ otherwise.
- D is updated in two steps:
  1. $D \leftarrow (D << 1) + 1$ (the bitwise *shift*). Now D tells, which prefixes would match if $T[j]$ would match every character.
  2. $D \leftarrow D \,\&\, B[T[j]]$ (the bitwise *and*). Remove the prefixes where $T[j]$ does not match.

Let w be the *wordsize* of the computer, typically 32 or 64. Assume first that $m \le w$. Then each bitvector can be stored in a single integer.

Algorithm 3.8: Shift-And
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* $B[P[i]] \leftarrow B[P[i]] + 2^i$ // $B[P[i]].i \leftarrow 1$
Search:
(3) $D \leftarrow 0$
(4) *for* $j \leftarrow 0$ *to* $n-1$ *do*
(5) $D \leftarrow ((D << 1) + 1) \,\&\, B[T[j]]$
(6) *if* $D \,\&\, 2^{m-1} \neq 0$ *then* return $j - m + 1$ // $D.(m-1) = 1$

Shift-Or is a minor optimization of Shift-And. It is the same algorithm except the roles of 0's and 1's in the bitvectors have been swapped. Then & on line 5 is replaced by | (bitwise *or*). The advantage is that we don't need that "+1" on line 5.

Example 3.9: $P =$ `assi`, $T =$ `apassi`, bitvectors are columns.

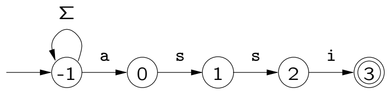
$B[c]$, $c \in \{$`a`,`i`,`p`,`s`$\}$

	a	i	p	s
a	1	0	0	0
s	0	0	0	1
s	0	0	0	1
i	0	1	0	0

D at each step

		a	p	a	s	s	i
a	0	1	0	1	0	0	0
s	0	0	0	0	1	0	0
s	0	0	0	0	0	1	0
i	0	0	0	0	0	0	1

The Shift-And algorithm can also be seen as a *bitparallel simulation* of the *nondeterministic* automaton that accepts a string ending with P.



After processing $T[j]$, $D.i = 1$ if and only if there is a path from the initial state (state -1) to state i with the string $T[0..j]$.

On an integer alphabet when $m \leq w$:

- Preprocessing time is $\mathcal{O}(\sigma + m)$.
- Search time is $\mathcal{O}(n)$.

If $m > w$, we can store the bitvectors in $\lceil m/w \rceil$ machine words and perform each bitvector operation in $\mathcal{O}(\lceil m/w \rceil)$ time.

- Preprocessing time is $\mathcal{O}(\sigma \lceil m/w \rceil + m)$.
- Search time is $\mathcal{O}(n \lceil m/w \rceil)$.

If no pattern prefix longer than w matches a current text suffix, then only the least significant machine word contains 1's. There is no need to update the other words; they will stay 0.

- Then the search time is $\mathcal{O}(n)$ on average.

Algorithms like Shift-And that take advantage of the implicit parallelism in bitvector operations are called *bitparallel*.

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 3.10: Horspool

`ainaisesti-ainainen`
`ainaine`~~`n`~~
`        ainaine`~~`n`~~
`            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 we are 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 3.11: 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:
(1) *for* $c \in \Sigma$ *do* $shift[c] \leftarrow m$
(2) *for* $i \leftarrow 0$ *to* $m-2$ *do* $shift[P[i]] \leftarrow m-1-i$
Search:
(3) $j \leftarrow 0$
(4) *while* $j + m \leq n$ *do*
(5)   *if* $P[m-1] = T[j+m-1]$ *then*
(6) $i \leftarrow m-2$
(7)   *while* $i \geq 0$ *and* $P[i] = T[j+i]$ *do* $i \leftarrow i-1$
(8)   *if* $i = -1$ *then* return j
(9) $j \leftarrow j + shift[T[j+m-1]]$
(10) 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 3.12: $P =$ `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, $P = \mathtt{ba}^{m-1}$ 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 3.13: Horspool shift

`varmasti-aikai`~~`s`~~`en-ainainen`
`ainaisen-ainainen`
`  ainaisen-ainainen`

Boyer–Moore shift

`varmasti-aikai`~~`s`~~`en-ainainen`
`ainaisen-ainainen`
`          ainaisen-ainainen`

Factor shift

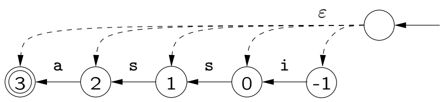
`varmasti-ai`~~`k`~~`aisen-ainainen`
`ainaisen-ainainen`
`                    ainaisen-ainainen`

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 3.14: $P = \mathtt{assi}$.



- 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.

  $\Rightarrow$ If $i = 0$, we found an occurrence.

  $\Rightarrow$ 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.

  $\Rightarrow$ Continue scanning.
- When the automaton can no more reach an accept state:

  $\Rightarrow$ 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 doing the shift at the end of the loop.

Algorithm 3.15: 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) $\quad D \leftarrow 2^m - 1$ $\qquad$ // $D \leftarrow 1^m$
(7) $\quad$ *while* $D \neq 0$ *do*
$\qquad$ // Now $T[j+i..j+m)$ is a pattern factor
(8) $\qquad i \leftarrow i - 1$
(9) $\qquad D \leftarrow D\ \&\ B[T[j+i]]$
(10) $\qquad$ *if* $D\ \&\ 2^{m-1} \neq 0$ *then*
$\qquad\quad$ // Now $T[j+i..j+m)$ is a pattern prefix
(11) $\qquad\quad$ *if* $i = 0$ *then* return j
(12) $\qquad\quad$ *else* $shift \leftarrow i$
(13) $\qquad D \leftarrow D << 1$
(14) $\quad j \leftarrow j + shift$

Example 3.16: $P = \texttt{assi}$, $T = \texttt{apassi}$.

$B[c]$, $c \in \{\texttt{a},\texttt{i},\texttt{p},\texttt{s}\}$

	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, $P = \texttt{a}^{m-1}\texttt{b}$ and $T = \texttt{a}^n$.
- In the best case, the search time is $\mathcal{O}(n/m)$.
  For example, $P = \texttt{b}^m$ and $T = \texttt{a}^n$.
- 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.