Algorithms and Data Structures II. Lecture Notes:

String Matching, Data Compression

Ásványi Tibor – asvanyi@inf.elte.hu

August 27, 2022

Contents

References

[1] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., Introduction to Algorithms (Fourth Edititon), *The MIT Press*, 2022

[2] Cormen, Thomas H., Algorithms Unlocked, *The MIT Press*, 2013.

[3] Ásványi, Tibor Algorithms and Data Structures I. Lecture Notes, `http://aszt.inf.elte.hu/~asvanyi/ds/AlgDs1/`, 2022

1 String Matching ([1] 32)

Given alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_d\}$ ($1 \le d < \infty$ integer constant), text $T : \Sigma[n]$, pattern $P : \Sigma[m]$, we search for all the occurrences of $P[0\,..\,m)$ in $T[0\,..\,n)$, provided that $0 < m \le n$. (These symbols will be used in this way in this chapter.) The elements of the alphabet will be called letters.

Definition 1.1
$s \in 0\,..\,(n-m)$ is a valid shift of P on T, **iff** *$T[s\,..\,s+m) = P[0\,..\,m)$.*

We will compute the set of valid shifts of P on T, i.e. set
$S = \{\, s \in 0\,..\,(n-m) \mid T[s\,..\,s+m) = P[0\,..\,m) \,\}$.

1.1 The naive string-matching (Brute-Force) algorithm

As an introduction, consider the following example. We search for pattern $P[0\,..\,4) = BABA$ in text $T[0\,..\,11) = ABABBABABAB$. (Notation: $\underline{B}$: letter B has been matched successfully against the appropriate letter of the text; $\cancel{B}$: it has been matched unsuccessfully.)

$i =$	0	1	2	3	4	5	6	7	8	9	10
$T[i] =$	A	B	A	B	B	A	B	A	B	A	B
	$\cancel{B}$	A	B	A							
		$\underline{B}$	$\underline{A}$	$\underline{B}$	$\cancel{A}$						
			$\cancel{B}$	A	B	A					
				$\underline{B}$	$\cancel{A}$	B	A				
$s=4$					$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$			
						$\cancel{B}$	A	B	A		
$s=6$							$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	
								$\cancel{B}$	A	B	A

$$S = \{\, 4; 6 \,\}$$

In general, we have a window, i.e $T[s\,..\,s+m)$ on the text. The size of the window is equal to the length of the pattern. We perform the following algorithm.

(0) In the beginning, the window is at the beginning of the text, i.e. $s = 0$, and the set S of valid shifts is empty.

(1) We check, whether we see the pattern, i.e. $P[0\,..\,m)$ in the window.

(2) If so, we add the actual shift s of the window to the set S of valid shifts.

(3) We perform $s := s + 1$, i.e., we slide the window to the right by one.

(4) If the window is still in the text, we go to step (1).

(5) We return the set S of valid shifts.

```
BruteForce($T : \Sigma[n]$ ; $P : \Sigma[m]$ ; $S : \mathbb{N}\{\}$)
    $S := \{\}$
    $s := 0$ to $n - m$
        IF match($T, s, P$) // $T[s\,..\,s{+}m) = P[0\,..\,m)$
            $S := S \cup \{s\}$
        ELSE
            SKIP
```

```
match( $T : \Sigma[]$ ; $s : \mathbb{N}$ ; $P : \Sigma[m]$ ) : $\mathbb{B}$
    // check, whether $T[s\,..\,s{+}m) = P[0\,..\,m)$
    $j := 0$
    $j < m \wedge T[s + j] = P[j]$
        $j++$
    return $j \geq m$
```

Property 1.2 *For the naive string-matching algorithm,*
$MT(n, m) \in \Theta((n{-}m{+}1) * m) \quad \wedge \quad mT(n, m) \in \Theta(n{-}m{+}1)$

Proof. First we take some notes on the number of loop iterations + subroutine calls, to prove both statements.
– The main loop of the Brute-Force algorithm iterates $n{-}m{+}1$ times.
– The loop of equality test $T[s\,..\,s{+}m) = P[0\,..\,m)$ iterates 0 to m times.
– Considering all the iterations of the Brute-Force algorithm, this equality test makes $(n{-}m{+}1) * m$ loop iterations in the worst case (when $P[0\,..\,m)$ occurs "everywhere" in $T[0\,..\,n)$, i.e., both of them have the form "$\sigma\sigma\ldots\sigma$"), and no iteration in the best case (when $P[0] \notin T[0\,..\,n)$).
– There are $1 + (n{-}m{+}1) = n{-}m{+}2$ subroutine calls.

Based on these notes, first we prove that $MT(n, m) \in \Theta((n{-}m{+}1) * m)$.
– In the worst case, there are $(n{-}m{+}1){+}(n{-}m{+}1){*}m{+}(n{-}m{+}2)$ steps, i.e. loop iterations + subroutine calls, which means $MT(n, m) = (n{-}m{+}1){*}(m{+}2){+}1$ steps altogether, where $n \geq m$.
– Thus $MT(n, m) > (n{-}m{+}1){*}m$. Therefore $MT(n, m) \in \Omega((n{-}m{+}1){*}m)$.
– In order to prove $MT(n, m) \in O((n{-}m{+}1){*}m)$, we can solve the following

inequality.
$(n-m+1) * (m + 2) + 1 \leq 2 * (n-m+1) * m$
$(n-m+1) * m + (n-m+1) * 2 + 1 \leq 2 * (n-m+1) * m$
$(n-m+1) * 2 + 1 \leq (n-m+1) * m$
$1 \leq (n-m+1) * (m - 2)$
And this is true if $m \geq 3$. (In this case, $m-2 \geq 1$. Thus $(n-m+1)*(m-2) \geq (n-m+1) * 1 \geq 1$ because $n \geq m$ in general in this topic.)

Finally we prove that $mT(n, m) \in \Theta(n-m+1)$.
– In the best case, there are $(n-m+1)+(n-m+2)$ steps, i.e. loop iterations + subroutine calls, which means $mT(n, m) = 2*(n-m+1)+1$ steps altogether, where $n \geq m$.
– Thus $mT(n, m) > (n-m+1)$. Consequently $mT(n, m) \in \Omega(n-m+1)$.
– In order to prove $mT(n, m) \in O(n-m+1)$, we can solve the following inequality.
$2 * (n-m+1) + 1 \leq 3 * (n-m+1)$
$1 \leq n-m+1$
$0 \leq n-m$
$m \leq n$ which is true in general in this topic. $\square$

Property 1.3 *Provided that on a class of pattern matching problems there is some constant $c \in (0; 1)$ so that $m \leq c * n$, we have the following asymptotic efficiency for the naive algorithm above.*

$$MT(n, m) \in \Theta(n * m) \quad \wedge \quad mT(n, m) \in \Theta(n)$$

Proof. $1*n \geq n-m+1 > n - c*n = (1-c)*n$ where $1-c \in (0; 1)$ constant.
Based on the definition of $\Theta(\cdot)$, $n-m+1 \in \Theta(n)$.
Therefore $(n-m+1) * m \in \Theta(n * m)$.
Considering Property 1.2 and the transitivity of relation $\cdot \in \Theta(\cdot)$, we have
$0 < c < 1 \wedge m \leq c * n \Rightarrow mT(n, m) \in \Theta(n) \wedge MT(n, m) \in \Theta(n * m)$. $\square$

Property 1.4 *Provided that on a class of pattern matching problems there are some constants $0 < \varepsilon \leq c < 1$ so that $\varepsilon * n \leq m \leq c * n$, we have the following worst-case asymptotic efficiency for the naive algorithm above.*

$$MT(n, m) \in \Theta(n^2)$$

Proof. Clearly $\varepsilon * n * n \leq n * m \leq n * n$. Thus $n * m \in \Theta(n^2)$. Considering $MT(n, m) \in \Theta(n * m)$ from Property 1.3, and the transitivity of relation $\cdot \in \Theta(\cdot)$, we have $MT(n, m) \in \Theta(n^2)$ $\square$

1.2 Quick Search

Quick Search is a simplified version of the Boyer-Moore algorithm. Boyer-Moore and its variants are considered extremely efficient string-matching algorithms in usual applications. Quick Search (or some other variant of Boyer-Moore) is often implemented in text editors for the *search* and *substitute* commands.

In Quick Search, similarly to the naive string-matching algorithm, we have a window ($T[s\,..\,s{+}m)$) with the size of the pattern ($P[0\,..\,m)$). We start with $s = 0$, i.e., we start with the window at the beginning of the text, and we make the $T[s\,..\,s{+}m) = P[0\,..\,m)$ comparisons repeatedly, i.e., we check repeatedly whether we see the pattern in the window. Between two comparisons/checks we slide the window to the right. The naive method always slides the window by one, i.e., it increases the actual shift of the window by one. The speedup of Quick Search (and of many other efficient string-matching algorithms) comes from a typically greater increase of shift s. However, we must ensure that while sliding the window to right, we do not jump over any valid shift, i.e., we find each substring of $T[0\,..\,n)$ which matches $P[0\,..\,m)$.

In these efficient string-matching algorithms, we typically make some preparations before we start the actual search: Based (only) on the pattern $P[0\,..\,m)$, we generate a table. And from this table, after a successful or unsuccessful matching, we can determine in $\Theta(1)$ time, how to go on.

In the case of Quick Search, in this preparation phase, we consider each element of the alphabet Σ. We add a label $shift(\sigma) \in 1\,..\,m{+}1$ to each $\sigma \in \Sigma$.

Let us suppose that $T[s\,..\,s{+}m)$ (the window) has just been matched against $P[0\,..\,m)$; $over := s + m$; $\sigma := T[over]$. This means that $T[over]$ is just over the actual window. Then the $shift(\sigma)$ value shows how much the window should be moved (to the right) above the text so that we have some chance to see the pattern through the window. To decide about the chance, we consider only this $\sigma = T[over]$ character of the text.

We have two cases.

1. Provided that $\sigma \in P[0\,..\,m)$, $shift(\sigma) \in 1\,..\,m$ shows how much the window should be moved above the text, so that the old $T[over]$ can be seen as the rightmost occurrence of σ in $P[0\,..\,m)$. This rightmost occurrence of σ in $P[0\,..\,m)$ corresponds to the smallest movement of the window. (The other occurrences of σ in $P[0\,..\,m)$ corresponds to bigger movements of the window.)

2. Provided that $\sigma \notin P[0\,..\,m)$, $shift(\sigma) = m+1$, i.e., the window jumps over this σ character of the text. (With a smaller movement of the window, we have no chance to see the pattern through it.)

For example, let the alphabet be $\Sigma = \{$A,B,C,D$\}$, and let the pattern be $P[0\,..\,4) =$ CADA. In the following examples, xxxx shows the window's position before sliding it to the right, and the pattern CADA displays the window's position after sliding it to the right.

```
   Text: ...xxxxA......xxxxB......xxxxC......xxxxD...
Pattern:     CADA           CADA      CADA     CADA
```

The appropriate *shift* values are given in the following table.

σ	A	B	C	D
$shift(\sigma)$	1	5	4	2

```
initShift($P : \Sigma[m]$)
$\forall \sigma \in \Sigma$
    $shift(\sigma) := m+1$
$j := 0$ to $m-1$
    $shift(P[j]) := m - j$
```

Considering the size of the alphabet as a constant, we receive

$$T_{\text{initShift}}(m) \in \Theta(m).$$

With the previous pattern $P[0\,..\,4) = CADA$, the illustration of initShift() and that of Quick Search follows.

σ	A	B	C	D
initial $shift(\sigma)$	5	5	5	5
C			4	
A	3			
D				2
A	1			
final $shift(\sigma)$	1	5	4	2

$i=$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$T[i]=$	A	D	A	B	A	B	C	A	D	A	B	C	A	B	A	D	A	C	A	D	A	D	A
	$\cancel{C}$	A	D	A																			
		$\cancel{C}$	A	D	A																		
$s=6$							$\underline{C}$	$\underline{A}$	$\underline{D}$	$\underline{A}$													
												$\underline{C}$	$\underline{A}$	$\cancel{D}$	A								
														$\cancel{C}$	A	D	A						
$s=17$																		$\underline{C}$	$\underline{A}$	$\underline{D}$	$\underline{A}$		
																				$\cancel{C}$	A	D	A

$$S = \{\,6; 17\,\}$$

```
QuickSearch($T : \Sigma[n]$ ; $P : \Sigma[m]$ ; $S : \mathbb{N}\{\}$)
    initShift($P$) ; $S := \{\}$ ; $s := 0$
    $s + m \le n$
        IF match($T, s, P$) // $T[s\,..\,s{+}m) = P[0\,..\,m)$
            $S := S \cup \{s\}$
        ELSE
            SKIP
        IF $s + m < n$
            $s{+}{=}\ shift(T[s+m])$
        ELSE
            break
```

$mT(n,m) \in \Theta\left(\frac{n}{m+1} + m\right)$ (e.g. if $T[0\,..\,n)$ and $P[0\,..\,m)$ are disjunct)
$MT(n,m) \in \Theta((n-m+1)*m)$ (e.g. if $T = \sigma\sigma\ldots\sigma$ és $P = \sigma\ldots\sigma$)

The best-case performance of Quick Search is an order of magnitude better than that of the naive string-matching algorithm. The worst-case performance is a bit worse than that of Brute-Force because of the running time of initShift(P), although this does not influence the asymptotic measure.

Fortunately, according to experimental studies, the average performance is much closer to the best case than to the worst case. As a result, in many practical applications, Quick Search is one of the best choices. However, if we want to optimize for the worst case, we need another algorithm, for example, Knuth-Morris-Pratt.

1.3 String Matching in Linear time (Knuth-Morris-Pratt, i.e. KMP algorithm)

As an introduction, consider the following example. We search for the occurrences of pattern $P[0\,..\,8) = BABABBAB$ in text a $T[0\,..\,18) = ABABABABBABABABBAB$. (Notation: The algorithm knows "even without matching" that the unmarked letters at the beginning of the text are the same as the corresponding letters in the text. $\underline{B}$: letter B has been matched successfully against the appropriate letter of the text; a $\cancel{B}$: it has been matched unsuccessfully. [Listen to the explanation at the lecture.])

$i =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$T[i] =$	A	B	A	B	A	B	A	B	B	A	B	A	B	A	B	B	A	B
	$\cancel{B}$																	
		$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\cancel{B}$											
s=3				B	A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$							
									B	A	B	$\underline{A}$	$\underline{B}$	$\cancel{B}$				
s=10											B	A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$
																B	A	B

$$S = \{\,3; 10\,\}$$

Notations 1.5

- *Let ε denote the empty string.*
- *If x and y are two strings, then $x + y$ is their concatenation. (For example, if $x = ABA$ and $y = BA$, then $x + y = ABABA$ and $y + x = BAABA$. If $y = \varepsilon$, then $x + y = x = y + x$.)*
- *If x and y are two strings, then $x \sqsubseteq y$ (x is* possibly full prefix of *y) means that $\exists z$ string, so that $x + z = y$. (For example, if $y = BABA$, then its possibly full prefixes are $\{\varepsilon, B, BA, BAB, BABA\}$.)*
- *If x and y are two strings, then $x \sqsubset y$ (x is* prefix of *y) means that $x \sqsubseteq y \land x \neq y$. (For example, if $y = BABA$, then its prefixes are $\{\varepsilon, B, BA, BAB\}$.)*
- *If x and y are two strings, then $x \sqsupseteq y$ (x is* possibly full suffix of *y) means that $\exists z$ string, so that $z + x = y$. (For example, if $y = BABA$, then its possibly full suffixes are $\{BABA, ABA, BA, A, \varepsilon\}$.)*
- *If x and y are two strings, then $x \sqsupset y$ (x is* suffix of *y) means that $x \sqsupseteq y \land x \neq y$. (For example, if $y = BABA$, then its suffixes are $\{ABA, BA, A, \varepsilon\}$.)*
- *$P_j = P[0\,..\,j)$ ($j \in 0\,..\,m$) string P_j with length j is a possibly full prefix of string P. P_0 is the empty prefix of P. Similarly $T_i = T[0\,..\,i)$. [Similarly $P_0 \sqsupset P_j \quad (j \in 1\,..\,m)$.]*

- $x \sqsubset\!\!\sqsupset y$ — correction: $x \square y$ (x *is* prefix-suffix *of* y) *means that* $x \sqsubset y \wedge x \sqsupset y$. *(For example, if* $y = BABA$, *then its prefix-suffixes are* $\{\varepsilon, BA\}$. *If* $y = BABAB$, *then its prefix-suffixes are* $\{\varepsilon, B, BAB\}$. *If* $y = ABC$, *its only prefix-suffix is* ε.*)*

- $\max_i H$ *is the ith greatest element of set* H $\quad (i \in 1\,..\,|H|)$.
[*Consequently* $\max_1 H = \max H$. *Provided that set* H *is finite,* $\max_{|H|} H = \min H$.]

- $H(j) = \{\, h \in 0\,..\,j-1 \mid P_h \sqsupset P_j \,\}$ $\quad (j \in 1\,..\,m)$
(For example, if $P_5 = BABAB$, *then* $H(5) = \{0,1,3\}, H(4) = \{0,2\}, H(3) = \{0,1\}, H(2) = \{0\} = H(1)$.*)*
[$0 \in H(j)$, $\max_1 H(j) = \max H(j)$, $\max_{|H(j)|} H(j) = \min H(j) = 0$.]
[*Equivalent definition:* $H(j) = \{\, |x| : x \square P_j \,\}$ $\quad (j \in 1\,..\,m)$]

- $next(j) = \max H(j)$ $\quad (j \in 1\,..\,m)$
(In the previous example, $next(5) = 3$. *And* $next(1) = 0$ *in general.)*

Properties 1.6 *(Similarly for prefixes)*
$x \sqsupseteq y \wedge y \sqsupseteq z \Rightarrow x \sqsupseteq z \qquad x \sqsupset y \wedge y \sqsupset z \Rightarrow x \sqsupset z$
$x \sqsupseteq y \wedge y \sqsupset z \Rightarrow x \sqsupset z \qquad x \sqsupset y \wedge y \sqsupseteq z \Rightarrow x \sqsupset z$

Property 1.7 *(Similarly for prefixes)* $x \sqsupset z \wedge y \sqsupset z \wedge |x| < |y| \Rightarrow x \sqsupset y$

Property 1.8 *Provided that* $i, j \in 0\,..\,m$, $\quad P_i \sqsupset P_j \iff P_i \square P_j$.

Property 1.9 *Provided that* $0 \le h < j \le m$ *and* $P_j \sqsupseteq T_i$,
$P_h \sqsupset T_i \iff P_h \sqsupset P_j$.

Properties 1.10
$P_h \sqsupseteq T_i \wedge P[h] = T[i] \iff P_{h+1} \sqsupseteq T_{i+1}$
$P_h \sqsupset P_j \wedge P[h] = P[j] \iff P_{h+1} \sqsupset P_{j+1}$
$P_h \square P_j \wedge P[h] = P[j] \iff P_{h+1} \square P_{j+1}$

Property 1.11 $next(j) \in 0\,..\,(j-1) \quad (j \in 1\,..\,m)$

Property 1.12 $next(j+1) \le next(j) + 1 \quad (j \in 1\,..\,m-1)$

Example 1.13

$P[j-1] =$	B	A	B	A	B	B	A	B
$j =$	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
$next(j) =$	*0*	*0*	*1*	*2*	*3*	*1*	*2*	*3*

Proof. [of Property 1.12: $next(j+1) \le next(j) + 1 \quad (j \in 1\,..\,m-1)$]

- If $next(j+1) = 0$, then $next(j+1) = 0 \leq 1 \leq next(j) + 1$.
- If $next(j+1) > 0$, consider $next(j+1) = \max H(j+1)$. Clearly, $next(j+1) \in H(j+1)$. Because $H(j+1) = \{\, h \in 0\,..\,j \mid P_h \sqsupset P_{j+1} \,\}$ we have $P_{(next(j+1)-1)+1} = P_{next(j+1)} \sqsupset P_{j+1}$. Based on property 1.10, $P_{next(j+1)-1} \sqsupset P_j$. Considering $next(j) = \max\{\, h \in 0\,..\,j{-}1 \mid P_h \sqsupset P_j \,\}$ we receive $next(j+1) - 1 \leq next(j)$, and $next(j+1) \leq next(j) + 1$.

□

Lemma 1.14 $next(max_l H(j)) \in H(j) \quad (j \in 1\,..\,m, l \in 1\,..\,|H(j)|{-}1)$

Proof.
$P_{next(\max_l H(j))} \sqsupset P_{\max_l H(j)}$ because $P_{next(i)} \sqsupset P_i \quad (i \in 1\,..\,m)$
$P_{\max_l H(j)} \sqsupset P_j$. Considering the transitivity of relation $\sqsupset$ (1.6) we receive $P_{next(\max_l H(j))} \sqsupset P_j$. As a result, $next(max_l H(j)) \in H(j)$. □

Property 1.15
$\max_{l+1} H(j) = next(\max_l H(j)) \quad (j \in 1\,..\,m, l \in 1\,..\,|H(j)|{-}1)$

Proof.

- First we prove that $\max_{l+1} H(j) \leq next(\max_l H(j))$.
  Clearly $P_{\max_{l+1} H(j)} \sqsupset P_j \wedge P_{\max_l H(j)} \sqsupset P_j \wedge \max_{l+1} H(j) < \max_l H(j)$. Considering 1.7, i.e. $x \sqsupset z \wedge y \sqsupset z \wedge |x| < |y| \Rightarrow x \sqsupset y$, we have $P_{\max_{l+1} H(j)} \sqsupset P_{\max_l H(j)}$. Thus $P_{\max_{l+1} H(j)} \,\square\, P_{\max_l H(j)}$. Considering the definition of function $next$, $P_{next(\max_l H(j))} \,\square\, P_{\max_l H(j)}$, and this is the longest one among the strings with this property. Consequently $\max_{l+1} H(j) \leq next(\max_l H(j))$.
- On the other hand, $next(\max_l H(j)) \leq \max_{l+1} H(j)$, because from the definition of function $next$, $next(\max_l H(j)) < \max_l H(j)$, and according to Lemma 1.14, $next(\max_l H(j)) \in H(j)$.

□

Now we define array $next/1 : \mathbb{N}[n]$. The following procedure, init$(next, P)$ initializes it so that it contains the values of function $next$, i.e., $\forall j \in 1\,..\,m :$ $next[j] = next(j)$, or in short: $next[1\,..\,m] = next(1\,..\,m)$. Procedure init$(next, P)$ fills array $next/1 : \mathbb{N}[n]$ based on array $P : \Sigma[m]$.

```
init($next/1 : \mathbb{N}[m]$ ; $P : \Sigma[m]$)
$next[1] := i := 0$ ; $j := 1$
$j < m$
    $P[i] = P[j]$
        $i{+}{+}$
        $j{+}{+}$
        $next[j] := i$
    
        $i = 0$
            $j{+}{+}$
            $next[j] := 0$
        
            $i := next[i]$
```

Invariant of the loop:
$0 \le i < j \le m \wedge$
$next[1\,..\,j] = next(1\,..\,j) \wedge$
$P_i \sqsupset P_j \wedge$
$[j < m \rightarrow$
$(\forall l \in i{+}2\,..\,j : P_l \not\sqsupset P_{j+1})]$

Exercise 1.16 *Prove that the invariant of the loop of procedure* init$(next, P)$ *is correct. Why does this invariant imply that* $next[1\,..\,m] = next(1\,..\,m)$ *is satisfied when the procedure returns?*

Property 1.17 *For* init$(next/1{:}\mathbb{N}[m];\ P{:}\Sigma[m])$: $MT(m), mT(m) \in \Theta(m)$.

Proof. It is enough to see that the loop of procedure init$(next, P)$ iterates minimum m–1 times, and maximum $2m$–2 times. Remember that according to the invariant of this loop $0 \le i < j \le m$.

- Before the first iteration $j = 1$, and each iteration increases j at most by one. Thus it needs at least m–1 iteration to achieve $j = m$, to finish the loop and the procedure as well.
- $t(i, j) := 2j - i$. Clearly, $0 \le i < j \le m$ implies $t(i, j) \in 2\,..\,2m$. Before the first iteration $t(i, j) = 2$. And $t(i, j)$ strictly increases with each iteration (on any branch of the core of the loop). Thus the loop stops after at most $2m-2$ iterations.

□

The illustration of procedure init$(next, P)$ on pattern $ABABBABA$:
(We start a new line at the beginning of a branch of the loop.)

			0	1	2	3	4	5	6	7
i	j	$next[j]$	A	B	A	B	B	A	B	A
0	1	0		$\cancel{A}$						
0	2	0			$\underline{A}$					
1	3	1			A	$\underline{B}$				
2	4	2			A	B	$\cancel{A}$			
0	4	2					$\cancel{A}$			
0	5	0						$\underline{A}$		
1	6	1						A	$\underline{B}$	
2	7	2						A	B	$\underline{A}$
3	8	3								

The result:

$P[j-1] =$	A	B	A	B	B	A	B	A
$j =$	1	2	3	4	5	6	7	8
$next[j] =$	0	0	1	2	0	1	2	3

Based on the properties of prefixes and suffixes which we have seen above, the algorithm Knuth-Morris-Pratt (KMP) solves the string matching problem in linear, i.e. $\Theta(n)$ time where n is the length of the text.

First we initialize array $next/1 : \mathbb{N}[m]$ with procedure init$(next, P)$. Next, with the help of this, we determine the valid shifts of the pattern:

```
KMP($T : \Sigma[n]$ ; $P : \Sigma[m]$ ; $S : \mathbb{N}\{\}$)
  $next/1 : \mathbb{N}[m]$ ; init$(next, P)$
  $S := \{\}$ ; $i := j := 0$
  while $i < n$
    if $P[j] = T[i]$
      $i++$ ; $j++$
      if $j = m$
        $S := S \cup \{i - m\}$
        $j := next[m]$
      else
        SKIP
    else
      if $j = 0$
        $i++$
      else
        $j := next[j]$
```

Before discussing KMP in general, we explain this procedure through the following example. We search for the occurrences, i.e. valid shifts of pattern $P[0..8) = ABABBABA$ (see the computation of its $next$ array above) in text $T[0..17) = ABABABBABABBABABA$.

$P[j-1] =$	A	B	A	B	B	A	B	A
$j =$	1	2	3	4	5	6	7	8
$next[j] =$	0	0	1	2	0	1	2	3

The search:

$i =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$T[i] =$	A	B	A	B	A	B	B	A	B	A	B	B	A	B	A	B	A
	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\cancel{B}$												
$s=2$			A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$							
$s=7$								A	B	A	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$		
													A	B	A	$\underline{B}$	$\cancel{B}$
															A	B	$\underline{A}$

$$S = \{2; 7\}$$

$s=0$ We start with $s=0$. We find that $P[0]=T[0],\ldots,P[3]=T[3]$, but $P[4]\neq T[4]$. The longest prefix of $P_8=P[0\,..\,8)$ which matches the beginning of the text is $P_4=P[0\,..\,4)$. The problem: where to shift P so that we have a chance to find a valid shift, but we do not jump over a solution. $P_4=T[0\,..\,4)$. Therefore, if we find a prefix of P_4 which is also a suffix of it, it will also be a suffix of $T[0\,..\,4)$. We know that $next[4]=2$, i.e., $P_2 \Box P_4$, and this is the longest one.

$s=2$ – If we shift P, so that $P[0\,..\,2)$ is under initial position of $P[2\,..\,4)$ which is equal to $T[2\,..\,4)$, then $P[0\,..\,2)$ automatically matches with $T[2\,..\,4)$, and we can go on with comparisons $P[2]=T[4], P[3]=T[5],\ldots,P[7]=T[9]$. We have found a valid shift ($s=2$) here.
– One may say that also $P_0 \Box P_4$, and we could go on with comparison $P[0]=T[4]$, but in this case, we would have jumped over a solution. The smallest shift corresponds to the longest prefix-suffix, and we have to use it, to avoid jumping over a possible solution.

Therefore we selected the longest prefix-suffix of P_4, went on with $P[2]=T[4]\ldots$ and finally found valid shift $s=2$. (Note that the length of the longest prefix-suffix is most often different from the valid shift found.) Now we find that $next[8]=3$ which means that the longest prefix-suffix of P_8 is P_3. Thus we shift P so that $P[0\,..\,3)$ is under the end of the previous position of $P[0\,..\,8)=T[2\,..\,10)$ because this shift corresponds to the smallest one where we have chance to find an occurrence of the pattern.

$s=7$ Again, automatically $P_3 \sqsupset T_{10}$. Thus we go on with checking $P[3]=T[10]$, and so on. We are lucky again because we find another valid shift $s=7$. Again we find that $next[8]=3$ which means that the longest prefix-suffix of P_8 is P_3. Thus we shift P so that $P[0\,..\,3)$ is under the end of the previous position of $P[0\,..\,8)=T[7\,..\,15)$ because this shift corresponds to the smallest one where we have chance to find an occurrence of the pattern.

$s=12$ Now the end of the pattern is over the text, but the KMP algorithm does not check this (to reduce the overall running time despite a longer "end of the game"). Still it finds $P[3]=T[15]$ and $P[4]\neq T[16]$. We have found that $P_4 \sqsupset T_{16}$. Because $next[4]=2$ which means that P_2 is the longest prefix-suffix of P_4, we can shift pattern P so that P_2 is under the previous position of $P[2\,..\,4)=T[14\,..\,16)$.

$s=14$ We have found that $P_2 \sqsupset T_{16}$. Thus we check $P[2]=T[16]$. This is true, but with the next check, i.e. $P[3]=T[17]$ we would over-index the text, so we stop.

After all, we received that the set of valid shifts is $S = \{2; 7\}$.

In general, we search for the valid shifts of P_m in T_n. We can start from P_0 and T_0 because $P_0 \sqsupseteq T_0$. In general, we have $P_j \sqsupseteq T_i$ for some i and j.

When the KMP algorithm finds that $P_j \sqsupseteq T_i \wedge j = m$, then it has found a valid shift which is $s = i - m$. When it finds that $P_j \sqsupseteq T_i \wedge j < m \wedge P[j] \neq T[i]$, then it has not found a valid shift.

But in both cases – provided that $j > 0$ –, in order to find a new valid shift, it determines the longest prefix-suffix of P_j which is $P_{next[j]}$. This is useful because $P_{next[j]} \sqsupseteq T_i$ and moving $P_{next[j]}$ under the end of T_i, this is the smallest shift with which we have chance to find a new valid shift. To make this movement of the pattern, we perform the assignment statement $j := next[j]$, and then $P_j \sqsupseteq T_i$ is satisfied again.

Provided that $i < n$, we can go on with checking $P[j] = T[i]$. (1) If $P[j] = T[i]$, then $P_{j+1} \sqsupseteq T_{i+1}$, and we increment i and j. Then $P_j \sqsupseteq T_i$ is satisfied again, and so on. (2) If $P[j] \neq T[i]$, again we have to determine the longest prefix-suffix of P_j, and so on.

Still there is the case when $j = 0 \wedge P[j] \neq T[i]$. Then we make the minimal possible shift, i.e., we increment i and go on.

The whole process can continue while $i < n$ which means that $T[i]$ exists.

Now we verify that $MT(n), mT(n) \in \Theta(n)$ for procedure KMP().

```
KMP($T : \Sigma[n]$ ; $P : \Sigma[m]$ ; $S : \mathbb{N}\{\}$)
  $next/1 : \mathbb{N}[m]$ ; init($next, P$)
  $S := \{\}$ ; $i := j := 0$
  $i < n$
    $P[j] = T[i]$
      $i{+}{+}$ ; $j{+}{+}$
      $j = m$
        $S := S \cup \{i - m\}$
        $j := next[m]$
      else
        SKIP
    else
      $j = 0$
        $i{+}{+}$
      else
        $j := next[j]$
```

Property 1.18 *A trivial invariant of the loop of algorithm KMP:*

$$i \in 0\,..\,n \wedge j \in [0\,..\,m) \wedge j \leq i$$

To prove that the time complexity of procedure KMP() is linear, it is enough to verify that the running time of its main loop is $\Theta(n)$, because $m \in 1\,..\,n$, consequently the $\Theta(m)$ time needed for the init$(next, P)$ call and other initialization do not modify this asymptotic order. And to verify the $\Theta(n)$ operational complexity of this main loop, it is sufficient to prove that it performs at least n and at most $2n$ iterations. For this purpose, we can use invariant 1.18: $i \in 0\,..\,n \wedge j \in 0\,..\,(m-1) \wedge j \leq i$.

- Before the first iteration, $i = 0$. Because each iteration increases i by at most one, and the condition of the loop is $i < n$, we need at least n iterations to finish the loop.
- To prove the upper bound $2n$ of the number of iterations, $t(i,j) := 2i - j$. Based on invariant $i \in 0\,..\,n \wedge j \in 0\,..\,(m-1) \wedge j \leq i$, we have $t(i,j) \in 0\,..\,2n$. Before the first iteration, $t(i,j) = 0$. Because $t(i,j) = 2i - j$ strictly increases on each of the four branches of the body of the loop, and $t(i,j) \leq 2n$, the loop stops after at most $2n$ iterations.

One can see that variable i never decreases during the run of procedure KMP() (i.e., we never backtrack on the text when we search it for the pattern). Consequently, the algorithm Knuth-Morris-Pratt can be implemented easily, even if this text is in a sequential file.

This is not the case with the Brute-force and Quick Search algorithms. In these algorithms, sometimes we have to backtrack even $m-2$ characters on the text. Provided that the text is in a sequential file, this means that during the run of the implementations of these algorithms, the last $m-1$ characters of the text must be stored in a buffer.