Algorithms and Data Structures II.
Lecture Notes:

String Matching

Ásványi Tibor – asvanyi@inf.elte.hu

December 15, 2023

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 Introduction

We have the alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_d\}$ ($1 \leq d < \infty$ integer constant). The elements of the alphabet will be called letters.

Given text $T : \Sigma[n]$, pattern $P : \Sigma[m]$, we search for all the occurrences of $P[0\,..\,m)$ in $T[0\,..\,n)$, provided that $0 < m \leq n$. (These symbols will be used in this way in this chapter.)

We suppose that $P : \Sigma[m]$, $T : \Sigma[n]$ and their lengths, i.e. m and n are fixed where $0 < m \leq n$.

To find the occurrences of $P[0\,..\,m)$ in $T[0\,..\,n)$, we search for those shifts s of P on T where $T[s\,..\,s{+}m) = P[0\,..\,m)$. Clearly, s must be in $0\,..\,n{-}m$.

Definition 1.1 *$s \in 0\,..\,n{-}m$ is a* possible shift *of P on T.*

It is a valid shift*,* **if** *$T[s\,..\,s{+}m) = P[0\,..\,m)$. Otherwise, it is an* invalid shift*.*

Problem 1.2 (String-matching) *Compute the set V of the valid shifts of P on T, i.e.*

$$V = \{\, s \in 0\,..\,n{-}m \mid T[s\,..\,s{+}m) = P[0\,..\,m) \,\}$$

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

As an introduction, consider the following example. We search for pattern $P[0\,..\,4) = BABA$ in text $T[0\,..\,11) = ABABBBABABAB$. (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

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

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 is empty. (Finally, S will contain the valid shifts of P on T, i.e. $S = V$ will be valid.)

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

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

(3) We perform $s := s + 1$, i.e., 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, equal to the set V of valid shifts.

```
BruteForce($T : \Sigma[n]$ ; $P : \Sigma[m]$ ; $S : \mathbb{N}\{\}$)
    $S := \{\}$
    $s := 0$ to $n - m$
        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 2.1 *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.
$2 * (n{-}m{+}1) + 1 \leq 3 * (n{-}m{+}1)$
$1 \leq n{-}m{+}1$
$0 \leq n{-}m$
$m \leq n$ is generally true in this topic. □

Property 2.2 *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 2.1 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 2.3 *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 2.2, and the transitivity of relation $\cdot \in \Theta(\cdot)$, we have $MT(n,m) \in \Theta(n^2)$ $\square$

3 Quick Search

Quick Search is a simplified version of the Boyer-Moore algorithm. Boyer-Moore and its variants are considered highly 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. Quick Search's speedup (and the speedup 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 the 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 prepare before we start the actual search: Based (only) on the pattern $P[0\,..\,m)$, we generate a table. This table shows how to go on in $\Theta(1)$ time after a successful or unsuccessful matching.

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 window movement, we cannot 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 it 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($P$) ; $S := \{\}$ ; $s := 0$
$s + m \le n$
    match($T, s, P$) // $T[s\,..\,s{+}m) = P[0\,..\,m)$
        $S := S \cup \{s\}$
    SKIP
    $s + m < n$
        $s {+}{=} shift(T[s+m])$
    **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 Brute-Force's 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 the worst case. As a result, in many practical applications, Quick Search is one of the best choices. However, if we want to optimise for the worst case, we need another algorithm, for example, Knuth-Morris-Pratt.

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

4.1 Notations and basic notions

- $\mathbb{N} = \{\, i \in \mathbb{Z} \mid i \geq 0 \,\}$ $\quad$ $i\,..\,k = \{\, j \in \mathbb{N} \mid i \leq j \leq k \,\}$
- $[i\,..\,k) = \{\, j \in \mathbb{N} \mid i \leq j < k \,\}$ $\quad$ $(i\,..\,k) = \{\, j \in \mathbb{N} \mid i < j < k \,\}$
- $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_d\}$ is the alphabet where $d \in \mathbb{N} \wedge d > 0$
- $T : \Sigma[n]$ is the text which is an array of letters (string) indexed from 0 to $n-1$
- $P : \Sigma[m]$ is the pattern $(0 < m \leq n)$. We search for the occurrences of P in T
- Let ε denote the empty string.
- $P_{:j} = P[0\,..\,j) = P[0\,..\,j-1]$ and similarly for T. $\quad P_{:0} = \varepsilon$
- String $x + y$ is the concatenation of strings x and y. (For example, if $x = ABA$ and $y = BA$, then $x + y = ABABA$ and $y + x = BAABA$. If $y = \varepsilon$, then $x + y = x = y + x$.)
- $x \sqsupseteq y$ (string x is a *suffix* of string y) if $\exists z$ string: $z + x = y$ $(P_{:0} \sqsupseteq P_{:j}.)$ (For example, if $y = BABA$, then its suffixes are $\{BABA, ABA, BA, A, \varepsilon\}$.)
- $x \sqsupset y$ (x is a *proper suffix* of y) if $x \sqsupseteq y \wedge x \neq y$ $\quad (P_{:0} \sqsupset P_{:j}$ if $j > 0)$ (For example, if $y = BABA$, then its proper suffixes are $\{ABA, BA, A, \varepsilon\}$.)
- $x \sqsubseteq y$ (x is a *prefix* of y) if $\exists z$ string: $x + z = y$ $\quad (P_{:i} \sqsubseteq P_{:j}$ if $i \leq j)$ (For example, if $y = BABA$, then its prefixes are $\{\varepsilon, B, BA, BAB, BABA\}$.)

- $x \sqsubset y$ (x is a *proper prefix* of y) if $x \sqsubseteq y \land x \neq y$ $\quad (P_{:i} \sqsubset P_{:j}$ if $i < j)$ $\quad$ (For example, if $y = BABA$, then its proper prefixes are $\{\varepsilon, B, BA, BAB\}$.)
- $\pi/1 : \mathbb{N}[m]$ is an array of natural numbers. It is indexed from 1 to m.

The following trivial properties of strings will be useful.

Lemma 4.1 *(Transitivity of the suffix relation)*
$x \sqsupset y \land y \sqsupseteq z \Rightarrow x \sqsupset z$.

Lemma 4.2 *(Overlapping-suffix lemma)*
Suppose that x, y and z are strings such that $x \sqsupseteq z$ and $y \sqsupseteq z$.
$|x| \leq |y| \Rightarrow x \sqsupseteq y$. $\quad |x| < |y| \Rightarrow x \sqsupset y$. $\quad |x| = |y| \Rightarrow x = y$.

Lemma 4.3 *(Suffix-extension lemma)*
$P_{:j} \sqsupseteq T_{:i} \land P[j] = T[i] \iff P_{:j+1} \sqsupseteq T_{:i+1}$.
$P_{:i} \sqsupset P_{:j} \land P[i] = P[j] \iff P_{:i+1} \sqsupset P_{:j+1}$.

4.2 Introduction to the KMP algorithm

We suppose that $P : \Sigma[m]$, $T : \Sigma[n]$ and their lengths, i.e. m and n are fixed where $0 < m \leq n$. We search for those shifts s of P on T where $T[s\,..\,s{+}m) = P[0\,..\,m)$. Clearly, s must be in $0\,..\,n{-}m$.

Definition 4.4 *$s \in 0\,..\,n{-}m$ is a* possible shift *of P on T.*
It is a valid shift*,* **if** *$T[s\,..\,s{+}m) = P[0\,..\,m)$. Otherwise, it is an* invalid shift*.*

Problem 4.5 (String-matching) *Compute the set V of the valid shifts of P on T, i.e.*

$$V = \{\, s \in 0\,..\,n{-}m \mid T[s\,..\,s{+}m) = P[0\,..\,m) \,\}$$

The naive string-matching (Brute-Force) algorithm checks each possible shift in order and collects the valid shifts with maximal (i.e. worst-case) time complexity $\Theta((n{-}m{+}1) * m)$, which is $\Theta(n^2)$ if $m = \lfloor n/2 \rfloor$. (See Section 2 and [1].)

More advanced methods – like the different versions of the Boyer-Moore, Rabin-Karp, and KMP [1] algorithms – use information gained about the pattern and the text. They do not check each possible shift of P on T but often make a jump in T.

We prefer KMP because it runs in $\Theta(n)$ time on all the possible inputs, and it never backtracks on T, making it easy to implement on sequential files. On the contrary, the Brute-force and Quick Search algorithms run in $\Theta(n^2)$ time in the worst case. And they sometimes 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.

KMP is traditionally introduced as a highly efficient simulation of *string matching with finite automata* [1]. Here, we avoid these automata and start with analysing P and T, i.e. the strings.

As an introductory example, consider the following one.

Example 4.6 *In this example, we suppose there is a longer text T, but we consider only $T[i-5\,..\,i+2] = BABABABB$ here. The pattern is $P = P_{:6} = BABABB$. The actual shift is $i-5$. The successfully matched characters are underlined. The unsuccessfully matched character is crossed out.*

...	T_{i-5}	T_{i-4}	T_{i-3}	T_{i-2}	T_{i-1}	T_i	T_{i+1}	T_{i+2}
...	B	A	B	A	B	A	B	B
$P =$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\cancel{B}$		
			B	A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$

In the third line of the table, we successfully matched $P_{:5}$ to $T[i-5\,..\,i)$ but $P[5] \neq T[i]$. Consequently, $i-5$ is not a valid shift.

Thus we make a minimal further shift *of P on T so that the $P_{:k}$ ($0 \leq k < 5$) which is still against $T[i-k\,..\,i)$ matches it, i.e. $P_{:k} \sqsupset T_{:i}$. (See the last line of the table.) This* minimal further shift *is ≤ 5, because $P_{:0} \sqsupset T_{:i}$. And with* this shift, *we do not jump over any possibly valid shift. Actually $k = 3$. Then we successfully match $P[3]$ to $T[i]$, $P[4]$ to $T[i+1]$ and $P[5]$ to $T[i+2]$. Thus, $i-3$ is a valid shift. The bigger possible shifts would jump over valid shift $i-3$.*

Understanding the previous example, the question remains: How do we efficiently determine the value of k above? In the the following argument, $j = 5$, but it can be applied to any $j \in 1\,..\,m$ where $i-j$ is the actual shift, $P_{:j} \sqsupseteq T_{:i}$ and $(P[j] \neq T[i] \lor j = m)$.

Unquestionably, a greater further shift corresponds to a smaller k, and a smaller further shift corresponds to a greater k. And k corresponds to the *minimal further shift* of P on T so that $P_{:k} \sqsupset T_{:i}$. Thus k is the greatest h so that $P_{:h} \sqsupset T_{:i}$ and $0 \leq h < j$. Moreover, $P_{:h} \sqsupset T_{:i}$ is equivalent to $P_{:h} \sqsupset P_{:j}$ because $P_{:j} \sqsupseteq T_{:i}$ and $0 \leq h < j$.

As a result, k depends only on $P_{:j}$. Because P is fixed, k depends only on j. In other words, we need the longest $P_{:h}$ so that $P_{:h} \sqsubset P_{:j} \wedge P_{:h} \sqsupset P_{:j}$. Its length is defined by the prefix function π.

Definition 4.7 $\pi(j) = \max\{\, h \in 0\,..\,j{-}1 \mid P_{:h} \sqsupset P_{:j} \,\} \qquad (j \in 1\,..\,m)$

Section 4.4 gives an efficient calculation of this function. Still, before it, we consider another example run of the KMP algorithm. And in Section 4.3, we analyse the main procedure of the KMP algorithm. In the case of manual calculation, we compute the π prefix function's values more quickly if we know its following properties.

Property 4.8 $j \in 1\,..\,m \Rightarrow \pi(j) \in 0\,..\,j-1$

Proof. According to definition 4.7, $\pi(j)$ is the maximum of a subset of $0\,..\,j-1$. □

Lemma 4.9 $j \in [1\,..\,m) \Rightarrow \pi(j+1) \leq \pi(j)+1$

Proof.
– If $\pi(j+1) = 0 \Rightarrow \pi(j+1) = 0 \leq 0+1 \leq \pi(j)+1$ because $\pi(j) \geq 0$ by the Property 4.8.
– If $\pi(j+1) > 0 \Rightarrow$ with Definition 4.7, $P_{:(\pi(j+1)-1)+1} = P_{:\pi(j+1)} \sqsupset P_{:j+1} \Rightarrow$ with Lemma 4.3, $P_{:\pi(j+1)-1} \sqsupset P_{:j} \Rightarrow$ again with Definition 4.7,
$\pi(j+1) - 1 \leq \pi(j) \Rightarrow \pi(j+1) \leq \pi(j)+1$. □

Example 4.10 *We calculate the π function on the pattern $P_{:8} = P[0\,..\,8) = BABABBAB$, according to the Definition 4.7, the Property 4.8 and the Lemma 4.9.*

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

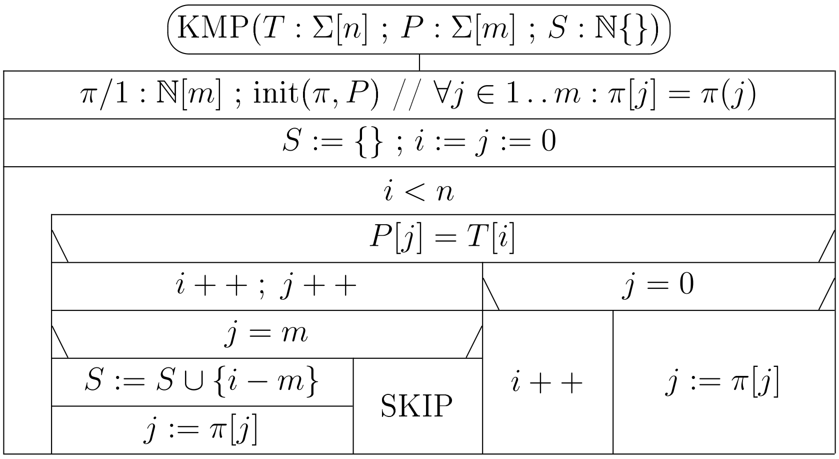
Example 4.11 *We search for the occurrences*
– of the pattern $P_{:8} = P[0\,..\,8) = BABABBAB$
– in the text $T_{:18} = T[0\,..\,18) = ABABABABBABABABBAB$.
(Notation: The algorithm knows "even without matching" that the unmarked letters at the beginning of the pattern are the same as the corresponding letters in the text. $\underline{B}$: letter B has been matched successfully against the appropriate text letter; a $\cancel{B}$: it has been matched unsuccessfully.)

$i =$	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

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

4.3 The KMP algorithm

In this section, first, we transform the intuitive approach of the KMP algorithm of the Introduction into the following structure diagram. Next, we analyse it.



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

Note 4.12 *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 quickly, even if this text is in a sequential file.*

We will prove in Section 4.4 that init(π, P) collects the values of the prefix function π into the $\pi/1$ array in $\Theta(m)$ time. And the postcondition of the init(π, P) call is $\forall j \in 1\,..\,m : \pi[j] = \pi(j)$. (See the structure diagram of init(π, P) in Section 4.4.) Our analysis of the KMP algorithm is based on Invariant 4.14 where $i - j$ is the actual shift.

4.3.1 The partial correctness of the procedure KMP(T, P, S)

The following lemma will be appropriate where $i - j$ is the actual shift.

Lemma 4.13
$j \in 1\,..\,m \wedge P_{:j} \sqsupseteq T_{:i} \Rightarrow$ there is no valid shift in $(i-j\,..\,i-\pi(j))$.

Proof. Assume indirectly that $k \in (\pi(j)\,..\,j)$ and $i - k$ is a valid shift. This means
$T[i-k\,..\,i-k+m) = P[0\,..\,m)$. Clearly, $k < j \leq m$, therefore $k < m$. Thus $T[i-k\,..\,i) = P[0\,..\,k)$, i.e. $P_{:k} \sqsupseteq T_{:i}$. And $P_{:j} \sqsupseteq T_{:i} \wedge k < j$. As a result, $P_{:k} \sqsupset P_{:j}$ because of the Overlapping-suffix lemma (4.2). But $k > \pi(j)$. For this reason, $P_{:k} \not\sqsupset P_{:j}$ follows from Definition 4.7 of the π function. □

Theorem 4.14
Statement (Inv) *is an invariant of the loop of the procedure KMP(T, P, S).*
(Inv) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i \leq n \wedge j < m \ \wedge S = V \cap [0\,..\,i-j)$.

Proof. Immediately before the first loop iteration, we perform the $S := \{\}$; $i := j := 0$ initialisations. Thus, (Inv) holds because
$i = j = 0 \wedge P_{:0} \sqsupseteq T_{:0} \wedge 0 \leq 0 \leq 0 \leq n \wedge 0 < m \wedge S = \{\} = V \cap [0\,..\,0)$.

We prove that each iteration of the loop keeps (Inv). The postcondition of the init(π, P) call, i.e. $(\forall j \in 1\,..\,m : \pi[j] = \pi(j))$ is implicitly added to each statement.

Supposing that $i < n$, we enter the loop and
(Inv1) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \ \wedge S = V \cap [0\,..\,i-j) \quad$ stands.

- If $P[j] = T[i]$, then
  $P_{:j+1} \sqsupseteq T_{:i+1}$ according to the Suffix-extension lemma (4.3). After increasing i and j we have
  (Inv2) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \ \wedge S = V \cap [0\,..\,i-j)$.

  1. If $j = m$, then $P_{:m} \sqsupseteq T_{:i}$, i.e. $P[0\,..\,m) = T[i-m\,..\,i)$. This means $i-m$ is a valid shift. Adding it to S, we have the following statement.
     (Inv3) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \ \wedge S = V \cap [0\,..\,i-j]$.
     Because $j \in 1\,..\,m \wedge P_{:j} \sqsupseteq T_{:i}$, considering Lemma 4.13, we receive that there is no valid shift in the interval $(i-j\,..\,i-\pi(j))$. Thus
     $P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \ \wedge S = V \cap [0\,..\,i-\pi(j))$.
     Consider that $P_{:\pi(j)} \sqsupset P_{:j} \sqsupseteq T_{:i}$. Based on the transitivity of the suffix relation (Lemma 4.1) and $\pi[j] = \pi(j)$:
     $P_{:\pi[j]} \sqsupset T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \ \wedge S = V \cap [0\,..\,i-\pi[j])$.

Because $\pi[j] = \pi(j) \in [0\,..\,j)$, after the $j := \pi[j]$ assignment already $0 \leq j < i \wedge j < m$ stands. Consequently,
$P_{:j} \sqsupset T_{:i} \wedge 0 \leq j < i \leq n \wedge j < m \;\wedge S = V \cap [0\,..\,i{-}j)$ holds. At the end of the first program branch, this directly implies (Inv).

2. Provided that $j \neq m$, (Inv2) implies $j < m$. Thus, (Inv) holds at the end of the second program branch.

- In case of $P[j] \neq T[i]$, the Suffix-extension lemma (4.3) implies $P_{:j+1} \not\sqsupseteq T_{:i+1}$, i.e. $P[0\,..\,j] \neq T[i-j\,..\,i]$. Based on (Inv1), $j < m$. Thus, $P[0\,..\,m) \neq T[i-j\,..\,i-j+m)$. Consequently, $i{-}j$ is an invalid shift. Comparing this to (Inv1), i.e.
$P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \;\wedge S = V \cap [0\,..\,i{-}j)$, we have
(Inv4) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \;\wedge S = V \cap [0\,..\,i{-}j]$.

3. If $j = 0$, then considering (Inv4) we receive
$P_{:0} \sqsupseteq T_{:i} \wedge 0 = j \leq i < n \wedge j < m \;\wedge S = V \cap [0\,..\,i{-}j]$.
After performing $i++$,
$P_{:0} \sqsupseteq T_{:i} \wedge 0 = j \leq i \leq n \wedge j < m \;\wedge S = V \cap [0\,..\,i{-}j)$ stands.
Therefore, (Inv) holds at the end of the third program branch:
(Inv) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i \leq n \wedge j < m \;\wedge S = V \cap [0\,..\,i{-}j)$.
4. Provided that $j \neq 0$, then taking (Inv4) into account,
$P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i < n \wedge j < m \;\wedge S = V \cap [0\,..\,i{-}j]$
is obtained. This has the direct consequence
(Inv3) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \;\wedge S = V \cap [0\,..\,i{-}j]$.
We have already seen in the examination of the first program branch that in the case of (Inv3), after performing assignment $j := \pi[j]$, the invariant (Inv) holds. Finally, (Inv) also stands at the end of the last program branch.

□

Theorem 4.15 *If the KMP algorithm terminates, it solves Problem 4.5 of string-matching, i.e. $S = V$ holds when it returns.*

Proof. Let us consider Theorem 4.14. The (Inv) invariant of KMP's loop with the loop's termination condition, i.e. $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i \leq n \wedge j < m \wedge S = V \cap [0\,..\,i{-}j)$ with $i \geq n$ implies that $i = n \wedge j < m \wedge S = V \cap [0\,..\,n{-}j)$ holds when the loop of KMP becomes completed. Furthermore, $j < m \Rightarrow [0\,..\,n{-}j) \supset [0\,..\,n{-}m) \supseteq \{\, s \in 0\,..\,n{-}m \mid T[s\,..\,s{+}m) = P[0\,..\,m) \,\} = V \Rightarrow [0\,..\,n{-}j) \supset V$. Thus $S = V \cap [0\,..\,n{-}j) = V$. Consequently, $S = V$ holds when the procedure KMP returns. □

4.3.2 The termination of the procedure KMP(T, P, S)

First, we prove that the loop iterates at least n times. Before the first iteration, $i = 0$. Each iteration increases i by 1 or 0. And the loop terminates with $i = n$ according to the $i < n$ condition and the $0 \le i \le n$ invariant. Thus, there are at least n iterations before the loop terminates.

Second, we prove that the loop iterates at most $2n$ times. Let the termination function be $2i - j$ where $0 \le j \le i \le n$ [see the (Inv) invariant in Theorem 4.14]. Thus $2i - j \in 0\,..\,2n$. Before the loop, $2i - j = 0$, and each iteration increases $2i - j$. Consequently, there are at most $2n$ iterations before the loop terminates.

The loop of KMP runs in $\Theta(n)$ time because n is at least the number of the iterations of the KMP loop, which is at most $2n$.

Remember that we will prove in Section 4.4 that the $\textsc{init}(\pi, P)$ call terminates in $\Theta(m)$ time.

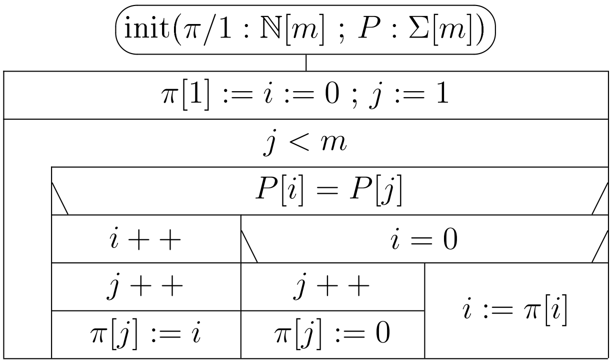
As a result, the time complexity of the KMP(T, P, S) procedure is $\Theta(n) + \Theta(m) = \Theta(n)$ because $n \ge m \ge 0$.

4.4 Initializing the prefix array

The following lemma will be appropriate:

Lemma 4.16 $P_{:i} \sqsupset P_{:j} \wedge 0 < i < j < m \wedge \pi(j+1) \le i \Rightarrow \pi(j+1) \le \pi(i) + 1$

Proof. If $\pi(j+1) = 0$ then $\pi(j+1) < 0 + 1 \le \pi(i) + 1$ because $\pi(i) \ge 0$ by definition. Provided that $\pi(j+1) > 0$, $k := \pi(j+1) - 1$. Thus $i > k \ge 0$ and $k + 1 = \pi(j+1)$. By the definition of the π function, $P_{:k+1} \sqsupset P_{:j+1}$. Consequently $P_{:k} \sqsupset P_{:j}$. Considering $P_{:i} \sqsupset P_{:j}$ and $k < i$, we have $P_{:k} \sqsupset P_{:i}$, consequently $k \le \pi(i)$. Therefore $\pi(j+1) = k + 1 \le \pi(i) + 1$. □



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

Invariant of the loop:
$P_{:i} \sqsupset P_{:j} \wedge$
$[\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,]$
$\wedge\, 0 \le i < j \le m \,\wedge$
$(j < m \rightarrow \pi(j+1) \le i + 1)$

4.4.1 The partial correctness of the procedure init(π, P)

Theorem 4.17
Statement (inv) *is an invariant of the loop of the procedure* init(π, P).
(inv) $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
$0 \leq i < j \leq m \wedge (\,j < m \rightarrow \pi(j+1) \leq i+1\,)$.

Proof.

Because of the initialisations $\pi[1] := i := 0$; $j := 1$, immediately before the first iteration of the loop, (inv) corresponds to the following statement: $P_{:0} \sqsupset P_{:1} \wedge (\,\forall k \in 1\,..\,1 : \pi[k] = \pi(k) = \pi(1) = 0\,) \wedge 0 \leq 0 < 1 \leq m \wedge (\,1 < m \rightarrow \pi(2) \leq 1\,)$. To prove the elements of this formula, $P_{:0} = \varepsilon$ is a proper suffix of any nonempty string; according to Property 4.8, $\pi(1) = 0$; the size m of the pattern P is not zero; and finally, because of Lemma 4.9, $\pi(1+1) \leq \pi(1) + 1 = 1$, provided that $m > 1$.

Still, we have to prove that the iterations of the loop keep the (inv) invariant, i.e. provided that (inv) holds before an iteration of the loop, it will also stand at the end of any branch of the loop's body. When we enter into the body of the loop, (inv) and the loop's condition $(j < m)$ implies

(inv1) $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
$0 \leq i < j < m \wedge \pi(j+1) \leq i+1$.

1. If $P[i] = P[j]$, according to Lemma 4.3 we have $P_{:i+1} \sqsupset P_{:j+1}$ because $P_{:i} \sqsupset P_{:j}$ [see (inv1)]. Based on the definition of the π prefix function (4.7), $P_{:i+1} \sqsupset P_{:j+1}$ implies $\pi(j+1) \geq i+1$. But $\pi(j+1) \leq i+1$ is found in (inv1). Consequently, $\pi(j+1) = i+1$. Performing the assignments $i++; j++; \pi[j] := i$,

   $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge 0 < i < j \leq m \wedge \pi(j) = i.$

   Provided that $j < m$, $\pi(j) = i$ and Lemma 4.9 implies $\pi(j+1) \leq \pi(j) + 1 = i + 1$. Therefore, at the end of the first branch of the loop's body, (inv) holds.

2-3. If $P[i] \neq P[j]$, then $P_{:i+1} \not\sqsupset P_{:j+1}$ because of Lemma 4.3. Thus, $\pi(j+1) \neq i+1$, according to Definition 4.7. In (inv1) we have $\pi(j+1) \leq i+1$. Consequently, $\pi(j+1) \leq i$.

   Comparing this to (inv1), we receive that (inv2) stands before the internal if-statement:

   (inv2) $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
   $0 \leq i < j < m \wedge \pi(j+1) \leq i$.

2. Provided that $i = 0$, consider $\pi(j+1) \leq i$ from (inv2). We have $\pi(j+1) = 0$ because the π function is non-negative.

   Comparing this to (inv2), after the assignments $j++; \; \pi[j] := 0$ we receive

   $P_{:i} \sqsupset P_{:j} \wedge (\, \forall k \in 1 \mathinner{..} j : \pi[k] = \pi(k) \,) \wedge 0 = i < j \leq m \wedge \pi(j) = i.$

   If $j < m$, Lemma 4.9 and $\pi(j) = i$ implies $\pi(j+1) \leq \pi(j) + 1 = i + 1$. Therefore, at the end of the second branch of the loop's body, (inv) also holds.

3. Provided that $i \neq 0$, then (inv2) implies (inv3):

   (inv3) $\quad P_{:i} \sqsupset P_{:j} \wedge (\, \forall k \in 1 \mathinner{..} j : \pi[k] = \pi(k) \,) \wedge$
   $\qquad\qquad 0 < i < j < m \wedge \pi(j+1) \leq i.$

   Thus, we can apply Lemma 4.16, and we have $\pi(j+1) \leq \pi(i)+1$. On the other hand, from (inv3) we receive $\pi[i] = \pi(i)$. Comparing this to Definition 4.7, we receive $P_{:\pi[i]} \sqsupset P_{:i}$. In (inv3), $P_{:i} \sqsupset P_{:j}$ is found. With Lemma 4.1, $P_{:\pi[i]} \sqsupset P_{:j}$ follows.

   Consider (inv3) and $\pi(j+1) \leq \pi(i) + 1$. After the assignment $i := \pi[i]$, we receive

   $P_{:i} \sqsupset P_{:j} \wedge (\, \forall k \in 1 \mathinner{..} j : \pi[k] = \pi(k) \,) \wedge$
   $0 \leq i < j < m \wedge \pi(j+1) \leq i + 1.$

   Therefore, at the end of the last branch of the loop's body, (inv) also holds.

□

Corollary 4.18 *Let us consider Theorem 4.17. Invariant* (inv) *of the loop and the negation of the loop's condition, i.e.* $j \geq m$*, implies* $j = m$*. As a result, procedure* $\mathrm{init}(\pi/1 : \mathbb{N}[m] \,;\, P : \Sigma[m])$ *has the following post-conditions:*

$$\forall k \in 1 \mathinner{..} m : \pi[k] = \pi(k)$$

.

4.4.2 The termination of the procedure init(π, P)

First, we prove that the loop iterates at least $m - 1$ times. Before the first iteration, $j = 1$. Each iteration increases j by 1 or 0. And the loop terminates with $j = m$ according to the $j < m$ condition and the $0 \leq j \leq m$ invariant. Thus, there are at least $m - 1$ iterations before the loop terminates.

Second, we prove that the loop iterates at most $2m - 2$ times. Let the termination function be $2j - i$ where $0 \leq i < j \leq m$. Thus $2j - i \in 2 \mathinner{..} 2m$.

Before the loop, $2j-i=2$, and each iteration increases $2j-i$. Consequently, the loop iterates not more than $2m-2$ times.

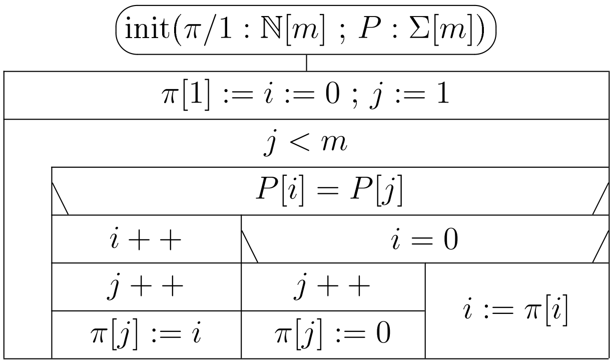
As a result, the time complexity of the init(π, P) procedure is $\Theta(m)$.

4.5 Summary

In this section, we found *relatively* simple and short mathematical proof of the correctness and efficiency of the KMP algorithm. (Compare it to the evidence in [1].) It is based on
(1) the properties of strings,
(2) the appropriate invariant properties of the loops of the algorithm and
(3) the suitable termination functions of these loops.

4.6 An illustration of the KMP algorithm

1st Example:
The illustration of the init(π, P) procedure on the pattern $ABABBBABA$:
(We start a new line at the beginning of each of the three program branches.)



```
init(π/1 : ℕ[m] ; P : Σ[m])
  $\pi[1] := i := 0 ; j := 1$
  $j < m$
    $P[i] = P[j]$
      $i++$
      $j++$
      $\pi[j] := i$
    $i = 0$
      $j++$
      $\pi[j] := 0$
    else
      $i := \pi[i]$
```

			0	1	2	3	4	5	6	7
i	j	$\pi[j]$	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
$\pi[j] =$	0	0	1	2	0	1	2	3

2nd Example:
We search the pattern $P[0\,..\,8) = ABABBABA$
in the text $T[0\,..\,17) = ABABABBABABBABABA$.

The $\pi/1 : \mathbb{N}[8]$ array for the sample has already been calculated.

$P[j-1] =$	A	B	A	B	B	A	B	A
$j =$	1	2	3	4	5	6	7	8
$\pi[j] =$	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	$\underline{A}$

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

5 Acknowledgments

Thanks to our faculty leaders for financial support, my colleagues for the encouragement and my students for the questions.