Substring Search: Brute Force and Knuth-Morris-Pratt Algorithm

Comp Sci 2C03

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Acknowledgments: Brute force based on *Algorithms* by Robert Sedgewick and Kevin Wayne (Chapter 5.3)

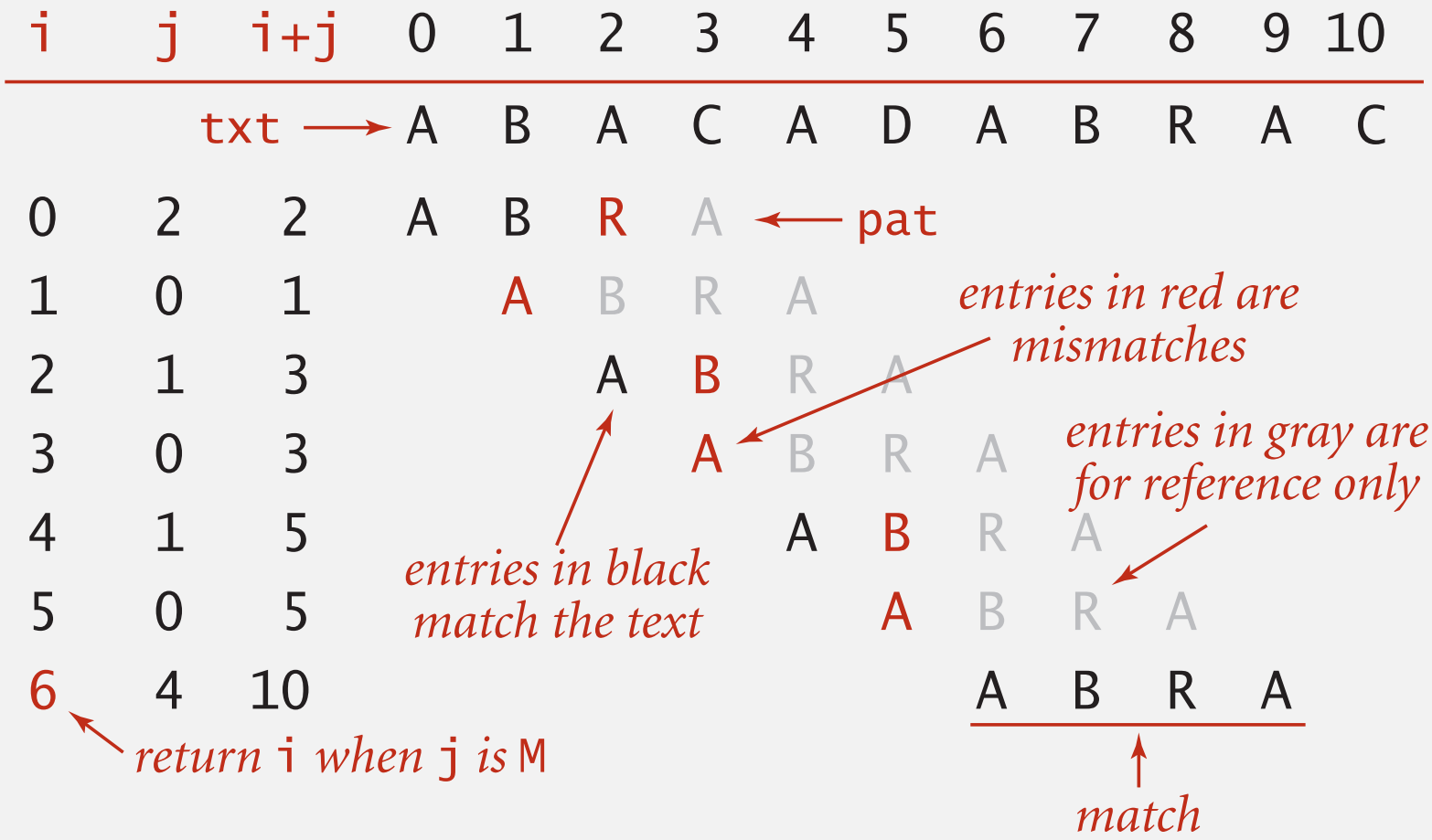
Substring Search (Pattern Matching)

Goal. Find pattern of length M in a text of length N.

typically N >> M

Brute Force

Check for pattern starting at each text position.



Brute Force: Java

Check for pattern starting at each text position.

i	j	i + j	0	1	2	3	4	5	6	7	8	9	10
			A	B	A	C	A	D	A	B	R	A	C
4	3	7					A	D	A	C	R		
5	0	5						A	D	A	C	R	

```
public static int search(String pat, String txt)
{
   int M = pat.length();
   int N = txt.length();
   for (int i = 0; i <= N - M; i++)
   {
      int j;
      for (j = 0; j < M; j++)
         if (txt.charAt(i+j) != pat.charAt(j))
            break;
      if (j == M) return i;
   }
```

← index in text where pattern starts

Brute Force: Worst Case

Brute-force algorithm can be slow if text and pattern are repetitive.

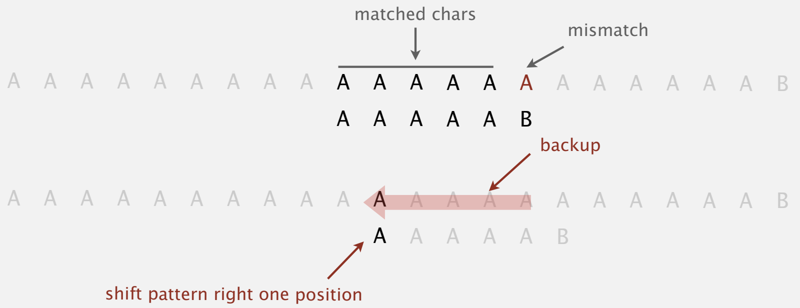
i	j	i+j	0	1	2	3	4	5	6	7	8	9
		txt →	A	A	A	A	A	A	A	A	A	B
0	4	4	A	A	A	A	B	← pat				
1	4	5		A	A	A	A	B				
2	4	6			A	A	A	A	B			
3	4	7				A	A	A	A	B		
4	4	8					A	A	A	A	B	
5	5	10						A	A	A	A	B

match

Worst case. $\sim MN$ char compares.

Backup

Brute-force algorithm needs backup for every mismatch.



Approach 1. Maintain buffer of last M characters.

Knuth-Morris-Pratt Algorithm: Example I

- It employs a clever method to always avoid backup and will run in $O(N)$.
- Let S = 'babcbabcabcaabcabcabcacabc' and P = 'abcabcacab'.
- To find P in S we slide P along S from left to right, looking at the characters that are opposite one another.
- Initially, we try the following configuration:

```
S    babcbabcabcaabcabcabcacabc
P    abcabcacab
     ↑
```

Knuth-Morris-Pratt Algorithm: Example II

- Initially, we try the following configuration:

```
S    b a b c b a b c a b c a a b c a b c a b c a c a b c
P    a b c a b c a c a b
     ↑
```

- We check the characters of P from left to right. The arrows show the comparisons carried out before we find a character that does not match.
- In this case there is only one comparison. After this failure we try

```
S    b a b c b a b c a b c a a b c a b c a b c a c a b c
P      a b c a b c a c a b
       ↑↑↑↑
```

Knuth-Morris-Pratt Algorithm: Example III

- There is only one comparison. After this failure we try

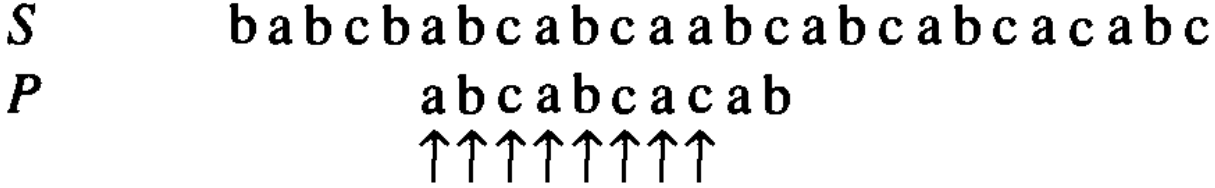
```
S    b a b c b a b c a b c a a b c a b c a b c a c a b c
P      a b c a b c a c a b
       ↑ ↑ ↑ ↑
```

- This time the first three characters of P are the same as the characters opposite them in S, but the fourth does not match.
- Up to now, we have proceeded exactly as in the naive algorithm. However we now know that the last four characters examined in S are abc$x$ where $x \neq$ "a".
- Without making any more comparisons with S, we can conclude that it is *useless* to slide P one, two, or three characters along : such an alignment cannot be correct.
- So let us try sliding P four characters along.

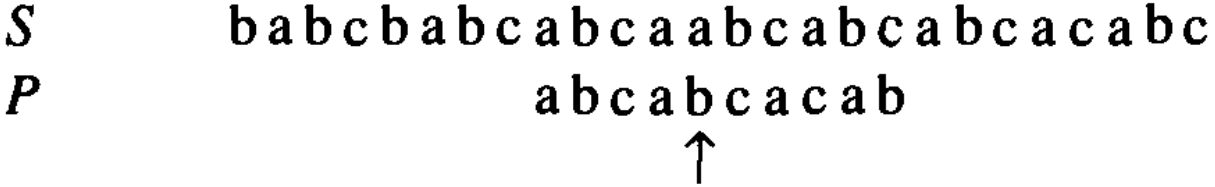
```
S    b a b c b a b c a b c a a b c a b c a b c a c a b c
P            a b c a b c a c a b
             ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
```

Knuth-Morris-Pratt Algorithm: Example IV

- So let us try sliding P four characters along.



- Following this mismatch, we know that the last eight characters examined in S are abcabca$x$ where $x \neq$ "c".
- Sliding P one or two places along cannot be right ; however moving it three places might work.



Knuth-Morris-Pratt Algorithm: Example V

- Sliding P one or two places along cannot be right ; however moving it three places might work.

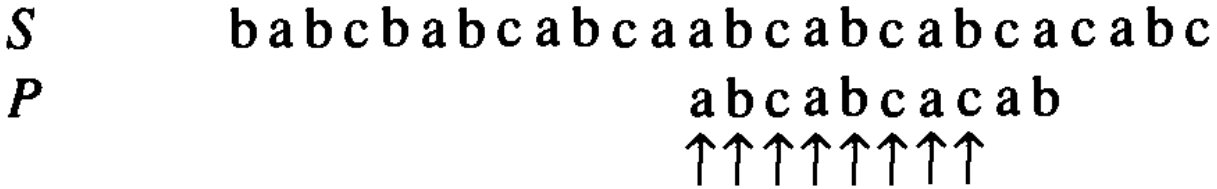
```
S    b a b c b a b c a b c a a b c a b c a b c a c a b c
P              a b c a b c a c a b
               ↑
```

- There is no need to recheck the first four characters of P : we chose the movement of P in such a way as to ensure that they necessarily match.
- It suffices to start checking at the current position of the pointer. In this case we have a second mismatch in the same position.
- This time, sliding P four places along might work. (A three-place movement is not enough: we know that the last characters examined in S are ax, where x is not a "b".)

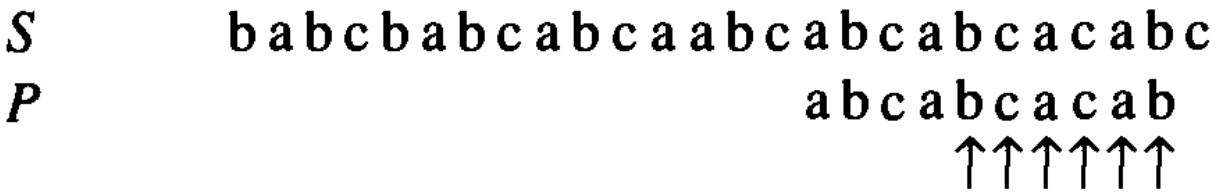
```
S    b a b c b a b c a b c a a b c a b c a b c a c a b c
P                    a b c a b c a c a b
                     ↑↑↑↑↑↑↑↑↑
```

Knuth-Morris-Pratt Algorithm: Example VI

- This time, sliding P four places along might work. (A three-place movement is not enough: we know that the last characters examined in S are ax, where x is not a "b".)



- Yet again we have a mismatch, and this time a three-place movement is necessary.



- We complete the verification starting at the current position time the correspondence between the target string and the pattern is complete.

KMP: Failure Function I

- The main idea of the KMP algorithm is to preprocess the pattern string P so as to compute a failure function f that indicates the proper shift of P so that, to the largest extent possible, we can reuse previously performed comparisons.
- Specifically, the **failure function** $f(j)$ is defined as the *length* of the *longest prefix* of P that is a *suffix* of $P[1..j]$ (note that we did not put $P[0..j]$ here).
- **Prefix**: let $x = a_1 \dots a_n$ be a string. The strings λ (empty string), $a_1, a_1a_2, \dots, a_1 \dots a_j, \dots, a_1 \dots a_n$ are **prefixes** of x.
- **Suffix**: let $x = a_1 \dots a_n$ be a string. The strings λ (empty string), $a_n, a_{n_1}a_n, \dots, a_j \dots a_n, \dots, a_1 \dots a_n$ are **suffixes** of x.
- Consider $x = abacab$.
  Prefixes $\lambda, a, ab, aba, abac, abaca, abacab$.
  Suffixes: $\lambda, b, ab, cab, acab, bacab, abacab$.
  The longest prefix of x that is a suffix of x is ab.

KMP: Failure Function II

- The **failure function** $f(j)$ is defined as the *length* of the *longest prefix* of P that is a *suffix* of $P[1..j]$ (note that we did not put $P[0..j]$ here).
- We also use the convention that $f(0) = 0$.
- The importance of this failure function is that it 'encodes' repeated substrings inside the pattern itself.
- Consider the pattern string $P = abacab$.
- The KMP failure function $f(j)$ for the string P is as shown in the following table:

j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
f(j)	0	0	1	0	1	2

KMP: Failure Function: Example I

- $P = abacab$

j	0	1	2	3	4	5
$P[j]$	a					
$f(j)$	0					

$\Rightarrow$

j	0	1	2	3	4	5
$P[j]$	a	b				
$f(j)$	0	0				

$\Rightarrow$

j	0	1	2	3	4	5
$P[j]$	a	b	a			
$f(j)$	0	0	1			

$\Rightarrow$

j	0	1	2	3	4	5
$P[j]$	a	b	a	c		
$f(j)$	0	0	1	0		

$\Rightarrow$

j	0	1	2	3	4	5
$P[j]$	a	b	a	c	a	
$f(j)$	0	0	1	0	1	

$\Rightarrow$

j	0	1	2	3	4	5
$P[j]$	a	b	a	c	a	b
$f(j)$	0	0	1	0	1	2

KMP: Failure Function: Example II

- $P = abcabcacab$

j	0	1	2	3	4	5	6	7	8	9
$P[j]$	a									
$f(j)$	0									

$\Rightarrow$

j	0	1	2	3	4	5	6	7	8	9
$P[j]$	a	b								
$f(j)$	0	0								

$\Rightarrow$

j	0	1	2	3	4	5	6	7	8	9
$P[j]$	a	b	c							
$f(j)$	0	0	0							

$\Rightarrow$

j	0	1	2	3	4	5	6	7	8	9
$P[j]$	a	b	c	a						
$f(j)$	0	0	0	1						

$\Rightarrow$

j	0	1	2	3	4	5	6	7	8	9
$P[j]$	a	b	c	a	b					
$f(j)$	0	0	0	1	2					

$\Rightarrow$

- $P = abcabcacab$

j	0	1	2	3	4	5	6	7	8	9
$P[j]$	a	b	c	a	b					
$f(j)$	0	0	0	1	2					

$\Rightarrow$

j	0	1	2	3	4	5	6	7	8	9
$P[j]$	a	b	c	a	b	c				
$f(j)$	0	0	0	1	2	3				

$\Rightarrow$

j	0	1	2	3	4	5	6	7	8	9
$P[j]$	a	b	c	a	b	c	a			
$f(j)$	0	0	0	1	2	3	1			

$\Rightarrow$

j	0	1	2	3	4	5	6	7	8	9
$P[j]$	a	b	c	a	b	c	a	c		
$f(j)$	0	0	0	1	2	3	1	0		

$\Rightarrow$

j	0	1	2	3	4	5	6	7	8	9
$P[j]$	a	b	c	a	b	c	a	c	a	
$f(j)$	0	0	0	1	2	3	1	0	1	

$\Rightarrow$

j	0	1	2	3	4	5	6	7	8	9
$P[j]$	a	b	c	a	b	c	a	c	a	b
$f(j)$	0	0	0	1	2	3	1	0	1	2

The Knuth-Morris-Pratt Algorithm: Pseudocode

The KMP pattern matching algorithm, shown below, incrementally processes the text string T comparing it to the pattern string P.

Algorithm KMPMatch(T, P):

Input: Strings T (text) with n characters and P (pattern) with m characters

Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T

$f \leftarrow$ KMPFailureFunction(P) // construct the failure function f for P
$i \leftarrow 0$
$j \leftarrow 0$
while $i < n$ **do**
  if $P[j] = T[i]$ **then**
    if $j = m - 1$ **then**
      return $i - m + 1$ // a match!
    $i \leftarrow i + 1$
    $j \leftarrow j + 1$
  else if $j > 0$ // no match, but we have advanced in P **then**
    $j \leftarrow f(j - 1)$ // j indexes just after prefix of P that must match
  else
    $i \leftarrow i + 1$
return "There is no substring of T matching P."

Intuition Behind the KMP Algorithm I

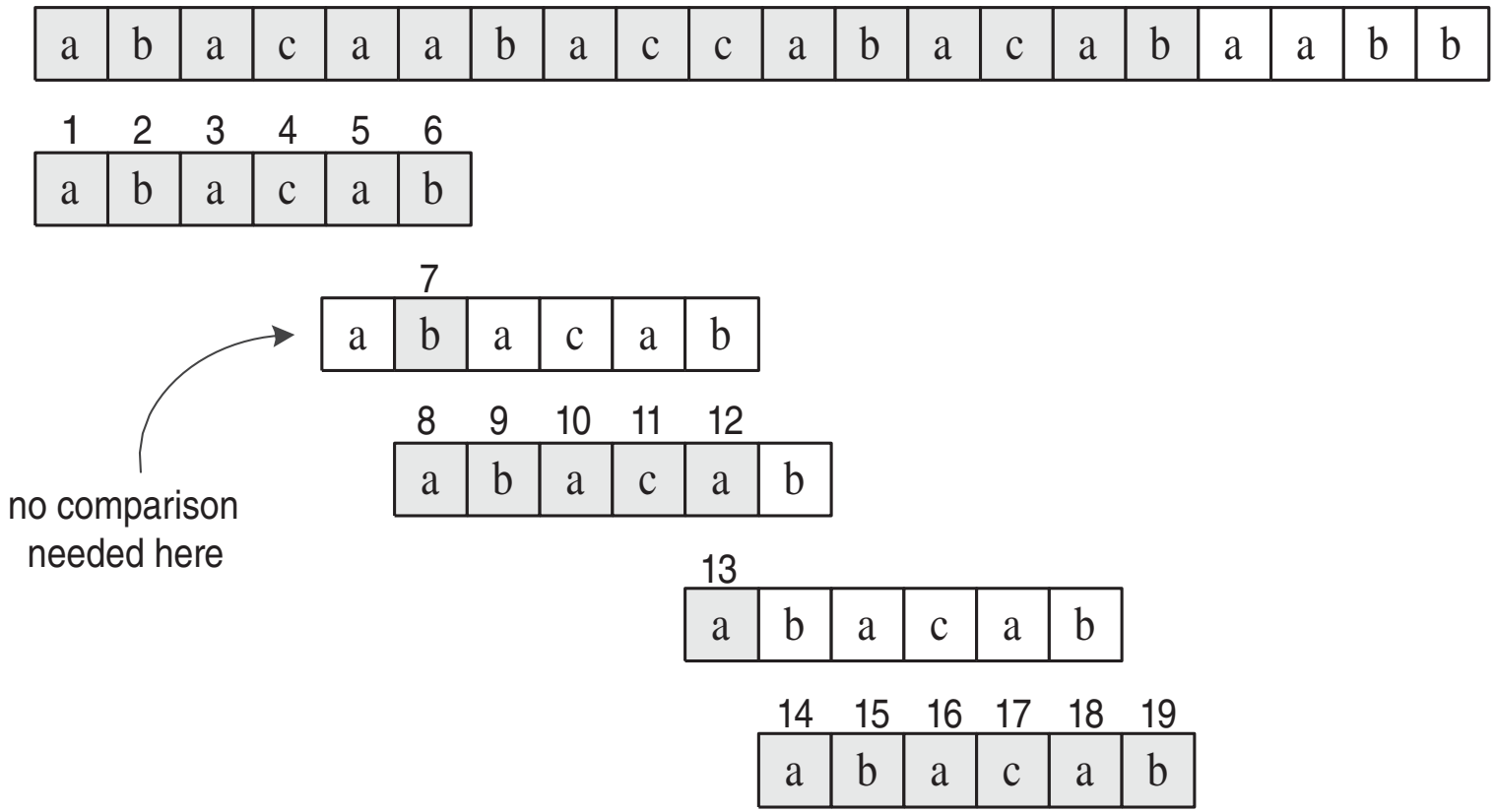
- During the execution of the KMP algorithm, each time there is a match, we increment the current indices.
- On the other hand, if there is a mismatch and we have previously made progress in P, then we consult the failure function to determine the new index in P where we need to continue checking P against T.
- Otherwise (there was a mismatch and we are at the beginning of P), we simply increment the index for T (and keep the index variable for P at its beginning).
- We repeat this process until we find a match of P in T or the index for T reaches n, the length of T (indicating that we did not find the pattern P in T).

Intuition Behind the KMP Algorithm II

- The main part of the KMP algorithm is the while-loop, which performs a comparison between a character in T and a character in P each iteration.
- Depending upon the outcome of this comparison, the algorithm either moves on to the next characters in T and P, consults the failure function for a new candidate character in P, or starts over with the next index in T.
- The correctness of this algorithm follows from the definition of the failure function.
- The skipped comparisons are actually unnecessary, for the failure function guarantees that all the ignored comparisons are redundant—they would involve comparing characters we already know match.

An illustration of the KMP pattern matching algorithm

j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
f(j)	0	0	1	0	1	2



The algorithm performs 19 character comparisons, which are indicated with numerical labels.

Constructing the KMP Failure Function

- We compare the pattern to itself.
- Each time we have two characters that match, we set $f(i) = j + 1$.
- Note that since we have $i > j$ throughout the execution of the algorithm, $f(j-1)$ is always defined when we need to use it.
- We want time complexity $O(m)$, where $m = \mathit{length}(P)$

Algorithm KMPFailureFunction(P):

Input: String P (pattern) with m characters

Output: The failure function f for P, which maps j to the length of the longest prefix of P that is a suffix of $P[1..j]$

$i \leftarrow 1$
$j \leftarrow 0$
$f(0) \leftarrow 0$
while $i < m$ **do**
    if $P[j] = P[i]$ **then**
        // we have matched $j + 1$ characters
        $f(i) \leftarrow j + 1$
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
    else if $j > 0$ **then**
        // j indexes just after a prefix of P that must match
        $j \leftarrow f(j - 1)$
    else
        // we have no match here
        $f(i) \leftarrow 0$
        $i \leftarrow i + 1$

Note how the algorithm uses the previous values of the failure function to efficiently compute new values.

Analysis of the KMP Algorithm

Theorem

The Knuth-Morris-Pratt algorithm performs pattern matching on a text string of length n and a pattern string of length m in $O(n+m)$ time.

- The running time analysis of the KMP algorithm may seem a little surprising at first, for it states that, in time proportional to that needed just to read the strings T and P separately, we can find the first occurrence of P in T.
- Also, it should be noted that the running time of the KMP algorithm does not depend on the size of the alphabet.
- The intuition behind the worst-case efficiency of the KMP algorithm comes from our being able to get the most out of each comparison that we do, and by our not performing comparisons we know to be redundant.
- The KMP algorithm is best suited for strings from small-size alphabets, such as DNA sequences.