
CS481/CS583: Bioinformatics Algorithms

Can Alkan

EA509

`calkan@cs.bilkent.edu.tr`

<http://www.cs.bilkent.edu.tr/~calkan/teaching/cs481/>

EXACT STRING MATCHING

The problem of String Matching

Given a string 't', the problem of string matching deals with finding whether a pattern 'p' occurs in 't' and if 'p' does occur then returning position in 't' where 'p' occurs.

Brute force ($O(mn)$)

$n \leftarrow |t|$

$m \leftarrow |p|$

$i \leftarrow 1$

while $i < n$

 if $p = t[i, i+m-1]$

 return i ;

 else

$i = i + 1$;

SimpleStringSearch

$t[0]$ $t[1]$ $t[2]$ $t[3]$ $t[4]$ $t[5]$ $t[6]$ $t[7]$ $t[8]$ $t[9]$ $t[10]$

A	B	C	E	F	G	A	B	C	D	E
---	---	---	---	---	---	---	---	---	---	---

$p[0]$ $p[1]$ $p[2]$ $p[3]$

A	B	C	D
---	---	---	---

Y Y Y N

SimpleStringSearch

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$t[4]$	$t[5]$	$t[6]$	$t[7]$	$t[8]$	$t[9]$	$t[10]$
A	B	C	E	F	G	A	B	C	D	E

$p[0]$	$p[1]$	$p[2]$	$p[3]$
A	B	C	D

N

SimpleStringSearch

<i>t[0]</i>	<i>t[1]</i>	<i>t[2]</i>	<i>t[3]</i>	<i>t[4]</i>	<i>t[5]</i>	<i>t[6]</i>	<i>t[7]</i>	<i>t[8]</i>	<i>t[9]</i>	<i>t[10]</i>
A	B	C	E	F	G	A	B	C	D	E

<i>p[0]</i>	<i>p[1]</i>	<i>p[2]</i>	<i>p[3]</i>
A	B	C	D

N

SimpleStringSearch

<i>t[0]</i>	<i>t[1]</i>	<i>t[2]</i>	<i>t[3]</i>	<i>t[4]</i>	<i>t[5]</i>	<i>t[6]</i>	<i>t[7]</i>	<i>t[8]</i>	<i>t[9]</i>	<i>t[10]</i>
A	B	C	E	F	G	A	B	C	D	E

<i>p[0]</i>	<i>p[1]</i>	<i>p[2]</i>	<i>p[3]</i>
A	B	C	D

N

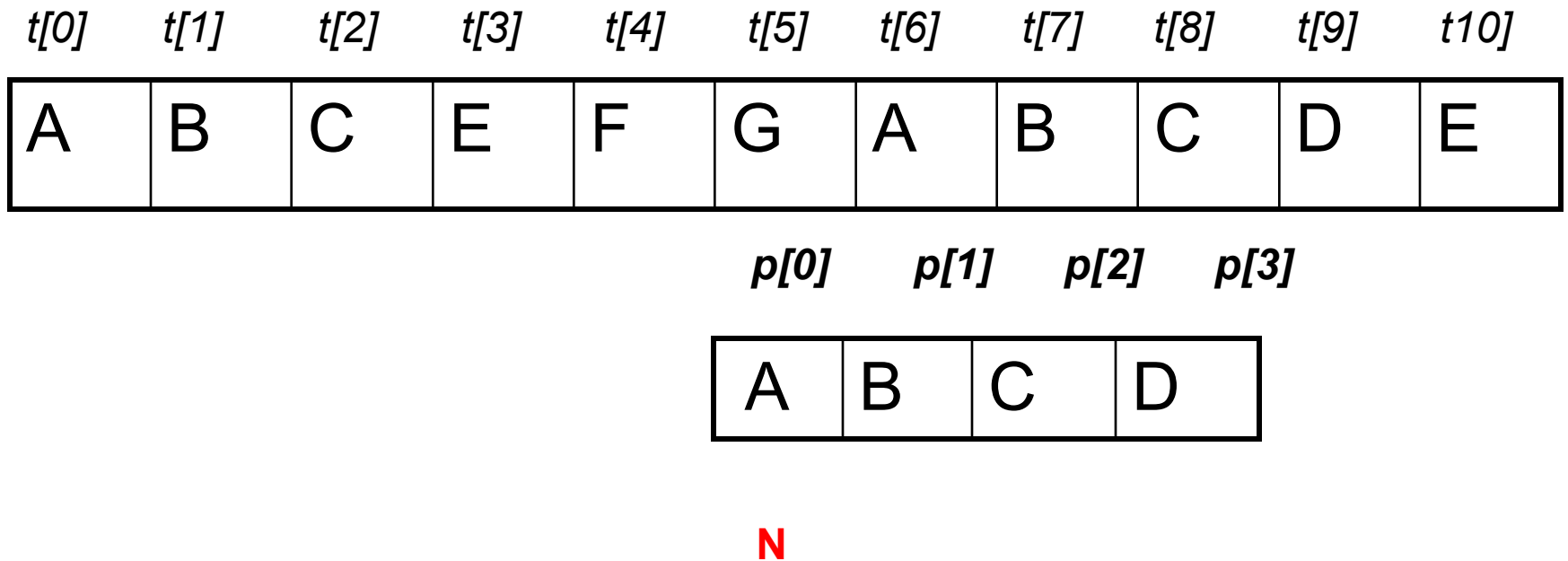
SimpleStringSearch

<i>t[0]</i>	<i>t[1]</i>	<i>t[2]</i>	<i>t[3]</i>	<i>t[4]</i>	<i>t[5]</i>	<i>t[6]</i>	<i>t[7]</i>	<i>t[8]</i>	<i>t[9]</i>	<i>t[10]</i>
A	B	C	E	F	G	A	B	C	D	E

<i>p[0]</i>	<i>p[1]</i>	<i>p[2]</i>	<i>p[3]</i>
A	B	C	D

N

SimpleStringSearch



SimpleStringSearch

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$t[4]$	$t[5]$	$t[6]$	$t[7]$	$t[8]$	$t[9]$	$t[10]$
A	B	C	E	F	G	A	B	C	D	E

$p[0]$	$p[1]$	$p[2]$	$p[3]$
A	B	C	D

Y	Y	Y	Y
---	---	---	---

Straightforward string searching

- Worst case:
 - ❑ Pattern string always matches completely except for last character
 - ❑ Example: search for XXXXXXY in target string of XXXXXXXXXXXXXXXXXXXXXXXX
 - ❑ Outer loop executed once for every character in target string
 - ❑ Inner loop executed once for every character in pattern
 - ❑ $O(mn)$, where $m = |p|$ and $n = |t|$
- OK if patterns are short, but better algorithms exist

Knuth-Morris-Pratt

- $O(m+n)$
 - Key idea:
 - if pattern fails to match, slide pattern to right by as many boxes as possible without permitting a match to go unnoticed
-

SimpleStringSearch

$t[0]$ $t[1]$ $t[2]$ $t[3]$ $t[4]$ $t[5]$ $t[6]$ $t[7]$ $t[8]$ $t[9]$ $t[10]$

A	B	C	E	F	G	A	B	C	D	E
---	---	---	---	---	---	---	---	---	---	---

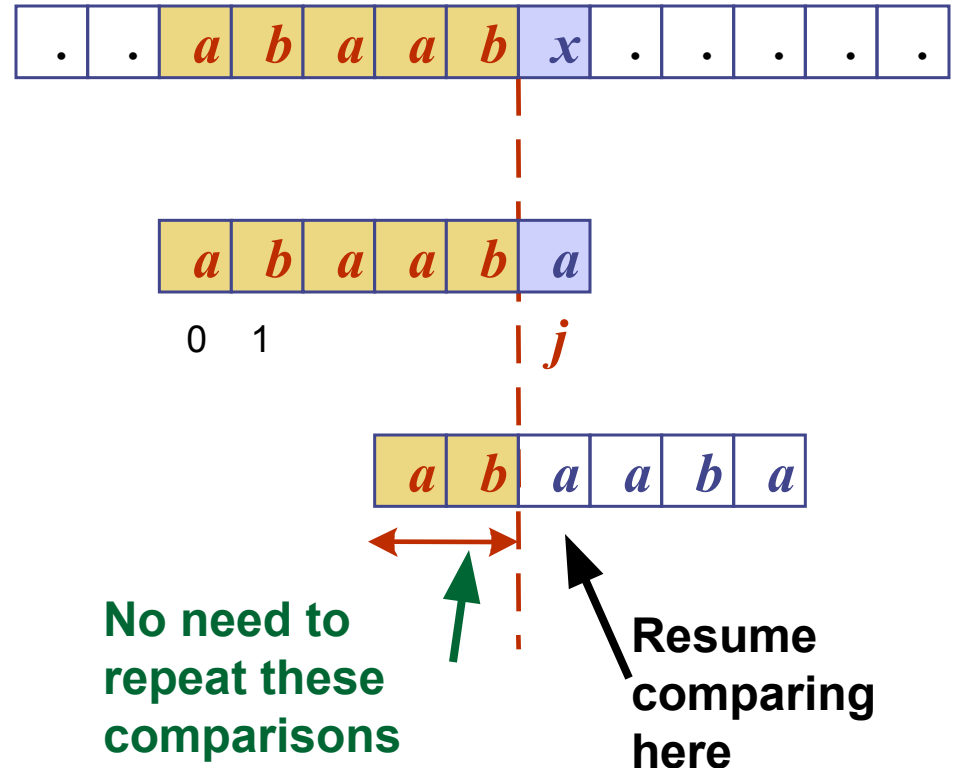
$p[0]$ $p[1]$ $p[2]$ $p[3]$

A	B	C	D
---	---	---	---

Y Y Y N

The KMP Algorithm - Motivation

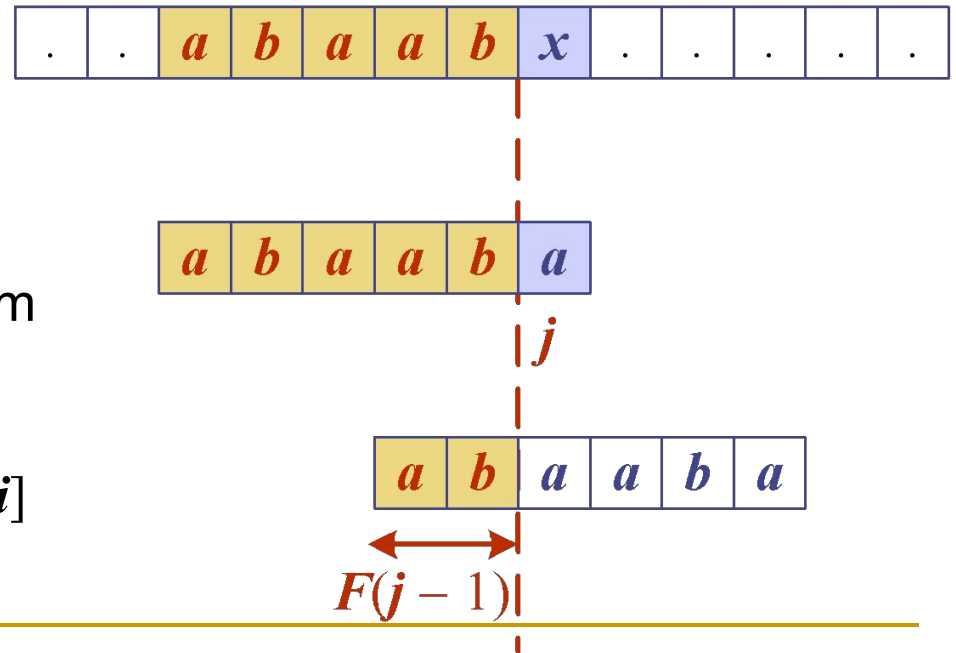
- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[0..j-1]$ that is a suffix of $P[1..j-1]$



KMP Failure Function

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself
- The **failure function** $F(j)$ is defined as the length of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F(j-1)$

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



The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the while-loop, either
 - i increases by one, or
 - the shift amount $i - j$ increases by at least one (observe that $F(j-1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP's algorithm runs in optimal time $O(m + n)$

Algorithm *KMPMatch*(T, P)

$F \leftarrow \text{failureFunction}(P)$

$i \leftarrow 0$

$j \leftarrow 0$

while $i < n$

if $T[i] = P[j]$

if $j = m - 1$

return $i - j$ { match }

else

$i \leftarrow i + 1$

$j \leftarrow j + 1$

else

if $j > 0$

$j \leftarrow F[j - 1]$

else

$i \leftarrow i + 1$

$j \leftarrow 0$

return -1 { no match }

Computing the Failure Function

- The failure function can be represented by an array and can be computed in $O(m)$ time
- The construction is similar to the KMP algorithm itself
- At each iteration of the while-loop, either
 - i increases by one, or
 - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$)
- Hence, there are no more than $2m$ iterations of the while-loop

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

j ↓ ↓ i

k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$F(k)$	0					

$m = 6$
 $i = 1$
 $j = 0$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

j ↓ ↓ i

k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$F(k)$	0					

$m = 6$
 $i = 1$
 $j = 0$

Algorithm *failureFunction*(P)

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$      ←  
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

j ↓ ↓ i

k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$F(k)$	0					

$m = 6$
 $i = 1$
 $j = 0$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then ←  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

j ↓ ↓ i

k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$F(k)$	0					

$m = 6$
 $i = 1$
 $j = 0$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

	j			i		
	↓			↓		
k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$F(k)$	0	0				

$m = 6$
 $i = 2$
 $j = 0$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

	j		i			
	↓		↓			
k	0	1	2	3	4	5
$P[k]$	<u>a</u>	b	<u>a</u>	c	a	b
$F(k)$	0	0				

$m = 6$
 $i = 2$
 $j = 0$

Algorithm *failureFunction*(P)

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$  ←  
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```


Example

	j			i		
	\downarrow			\downarrow		
k	0	1	2	3	4	5
$P[k]$	<u>a</u>	b	<u>a</u>	c	a	b
$F(k)$	0	0				

$m = 6$
 $i = 2$
 $j = 0$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

	j			i		
	↓			↓		
k	0	1	2	3	4	5
$P[k]$	<u>a</u>	b	<u>a</u>	c	a	b
$F(k)$	0	0	1			

$m = 6$
 $i = 3$
 $j = 1$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

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

$m = 6$
 $i = 3$
 $j = 1$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$  ←  
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

		j			i	
		↓		↓		
k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$F(k)$	0	0	1			

$m = 6$
 $i = 3$
 $j = 1$

Algorithm *failureFunction*(P)

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then ←  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

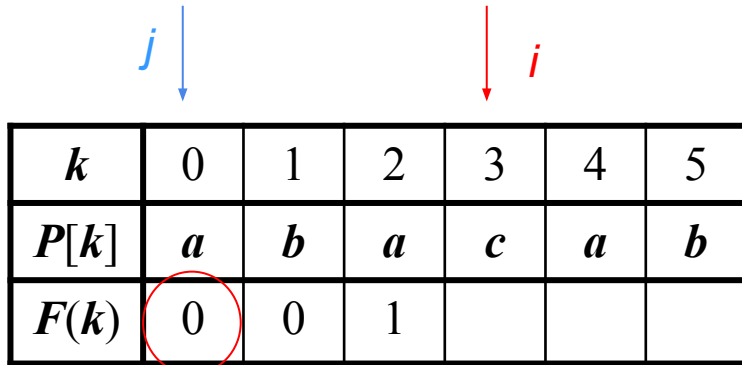
		j			i	
		↓		↓		
k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$F(k)$	0	0	1			

$m = 6$
 $i = 3$
 $j = 1$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example



The diagram illustrates the calculation of the failure function $F(k)$ for the string $P = abacab$. A table shows the indices k from 0 to 5, the corresponding characters $P[k]$, and the values of $F(k)$. A blue arrow labeled j points to the first column, and a red arrow labeled i points to the fourth column. The value $F(0) = 0$ is circled in red, with a red arrow pointing to the text $m = 6, i = 3, j = 0$.

k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$F(k)$	0	0	1			

$m = 6$
 $i = 3$
 $j = 0$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

	j ↓			↓ i		
k	0	1	2	3	4	5
$P[k]$	<u>a</u>	b	a	<u>c</u>	a	b
$F(k)$	0	0	1			

$m = 6$
 $i = 3$
 $j = 0$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

	j				i	
	\downarrow				\downarrow	
k	0	1	2	3	4	5
$P[k]$	<u>a</u>	b	a	<u>c</u>	a	b
$F(k)$	0	0	1	0		

$m = 6$
 $i = 4$
 $j = 0$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```


Example

	j				i	
	↓				↓	
k	0	1	2	3	4	5
$P[k]$	<u>a</u>	b	a	c	<u>a</u>	b
$F(k)$	0	0	1	0		

$m = 6$
 $i = 4$
 $j = 0$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

	<div><div>j</div><div>\downarrow</div></div>						<div><div>i</div><div>\downarrow</div></div>
k	0	1	2	3	4	5	
$P[k]$	a	b	a	c	a	b	
$F(k)$	0	0	1	0	1		

$m = 6$
 $i = 5$
 $j = 1$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

		j								i
		↓								↓
k	0	1	2	3	4	5				
$P[k]$	a	b	a	c	a	b				
$F(k)$	0	0	1	0	1	2				

$m = 6$
 $i = 5$
 $j = 1$


Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

j ↓

k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$F(k)$	0	0	1	0	1	2



$m = 6$
 $i = 6$
 $j = 2$

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

k	0	1	2	3	4	5
$P[k]$	a	b	a	c	a	b
$F(k)$	0	0	1	0	1	2

Algorithm *failureFunction(P)*

```
 $F[0] \leftarrow 0$   
 $i \leftarrow 1$   
 $j \leftarrow 0$   
 $m \leftarrow \text{length}(P)$   
while  $i < m$   
  if  $P[i] = P[j]$   
    {we have matched  $j + 1$  chars}  
     $F[i] \leftarrow j + 1$   
     $i \leftarrow i + 1$   
     $j \leftarrow j + 1$   
  else if  $j > 0$  then  
    {use failure function to shift  $P$ }  
     $j \leftarrow F[j - 1]$   
  else  
     $F[i] \leftarrow 0$  { no match }  
     $i \leftarrow i + 1$ 
```

Example

a b a c a a b a c c a b a c a b a a b b

1 2 3 4 5 6
a b a c a b j=5

7
a b a c a b j=1

8 9 10 11 12
a b a c a b j=4

13
a b a c a b j=0

14 15 16 17 18 19
a b a c a b

j=0
i++

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

```

Algorithm KMPMatch( $T, P$ )
     $F \leftarrow \text{failureFunction}(P)$ 
     $i \leftarrow 0$ 
     $j \leftarrow 0$ 
    while  $i < n$ 
        if  $T[i] = P[j]$ 
            if  $j = m - 1$ 
                return  $i - j$ 
                /* match */
            else
                 $i \leftarrow i + 1$ 
                 $j \leftarrow j + 1$ 
        else
            if  $j > 0$ 
                 $j \leftarrow F[j - 1]$ 
            else
                 $i \leftarrow i + 1$ 
                 $j \leftarrow 0$ 
    return -1 /* no match */
    
```

Example

<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>b</i>
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

1	2	3	4	5	6	
<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>j=5</i>

<i>j</i>	0	1	2	3	4	5
<i>P[j]</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
<i>F(j)</i>	0	0	1	0	1	2

```

Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m - 1
                return i - j
                /* match */
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0
                j ← F[j - 1]
            else
                i ← i + 1
                j ← 0
    return -1 /* no match */
    
```

Example

a b a c a a b a c c a b a c a b a a b b

7
a b a c a b j=1

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

Algorithm *KMPMatch*(T, P)

```
 $F \leftarrow \text{failureFunction}(P)$   
 $i \leftarrow 0$   
 $j \leftarrow 0$   
while  $i < n$   
  if  $T[i] = P[j]$   
    if  $j = m - 1$   
      return  $i - j$   
      /* match */  
    else  
       $i \leftarrow i + 1$   
       $j \leftarrow j + 1$   
  else  
    if  $j > 0$   
       $j \leftarrow F[j - 1]$   
    else  
       $i \leftarrow i + 1$   
       $j \leftarrow 0$   
return  $-1$  /* no match */
```


Example

a b a c a a b a c c a b a c a b a a b b

8 9 10 11 12
a b a c a b j=4

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

```

Algorithm KMPMatch( $T, P$ )
     $F \leftarrow \text{failureFunction}(P)$ 
     $i \leftarrow 0$ 
     $j \leftarrow 0$ 
    while  $i < n$ 
        if  $T[i] = P[j]$ 
            if  $j = m - 1$ 
                return  $i - j$ 
                /* match */
            else
                 $i \leftarrow i + 1$ 
                 $j \leftarrow j + 1$ 
        else
            if  $j > 0$ 
                 $j \leftarrow F[j - 1]$ 
            else
                 $i \leftarrow i + 1$ 
                 $j \leftarrow 0$ 
    return -1 /* no match */
    
```

Example

a b a c a a b a c c a b a c a b a a b b

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

j=0

13
a b a c a b

```

Algorithm KMPMatch( $T, P$ )
     $F \leftarrow \text{failureFunction}(P)$ 
     $i \leftarrow 0$ 
     $j \leftarrow 0$ 
    while  $i < n$ 
        if  $T[i] = P[j]$ 
            if  $j = m - 1$ 
                return  $i - j$ 
                /* match */
            else
                 $i \leftarrow i + 1$ 
                 $j \leftarrow j + 1$ 
        else
            if  $j > 0$ 
                 $j \leftarrow F[j - 1]$ 
            else
                 $i \leftarrow i + 1$ 
                 $j \leftarrow 0$ 
    return  $-1$  /* no match */
    
```

Example

a b a c a a b a c c a b a c a b a a b b

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

14 15 16 17 18 19
a b a c a b

j=0
i++

```

Algorithm KMPMatch( $T, P$ )
     $F \leftarrow \text{failureFunction}(P)$ 
     $i \leftarrow 0$ 
     $j \leftarrow 0$ 
    while  $i < n$ 
        if  $T[i] = P[j]$ 
            if  $j = m - 1$ 
                return  $i - j$ 
                /* match */
            else
                 $i \leftarrow i + 1$ 
                 $j \leftarrow j + 1$ 
        else
            if  $j > 0$ 
                 $j \leftarrow F[j - 1]$ 
            else
                 $i \leftarrow i + 1$ 
                 $j \leftarrow 0$ 
    return  $-1$  /* no match */
    
```

The Boyer-Moore Algorithm

- Similar to KMP in that:
 - Pattern compared against target
 - On mismatch, move as far to right as possible
- Different from KMP in that:
 - Compare the patterns from right to left instead of left to right
- Does that make a difference?
 - Yes – much faster on long targets; many characters in target string are never examined at all

Boyer-Moore example

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$t[4]$	$t[5]$	$t[6]$	$t[7]$	$t[8]$	$t[9]$	$t[10]$
A	B	C	E	F	G	A	B	C	D	E

$p[0]$	$p[1]$	$p[2]$	$p[3]$
A	B	C	D

N

There is no E in the pattern : thus the pattern can't match if *any* characters lie under $t[3]$. So, move four boxes to the right.

Boyer-Moore example

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$t[4]$	$t[5]$	$t[6]$	$t[7]$	$t[8]$	$t[9]$	$t[10]$
A	B	C	E	F	G	A	B	C	D	E

$p[0]$	$p[1]$	$p[2]$	$p[3]$
A	B	C	D

N

Again, no match. But there is a B in the pattern. So move two boxes to the right.

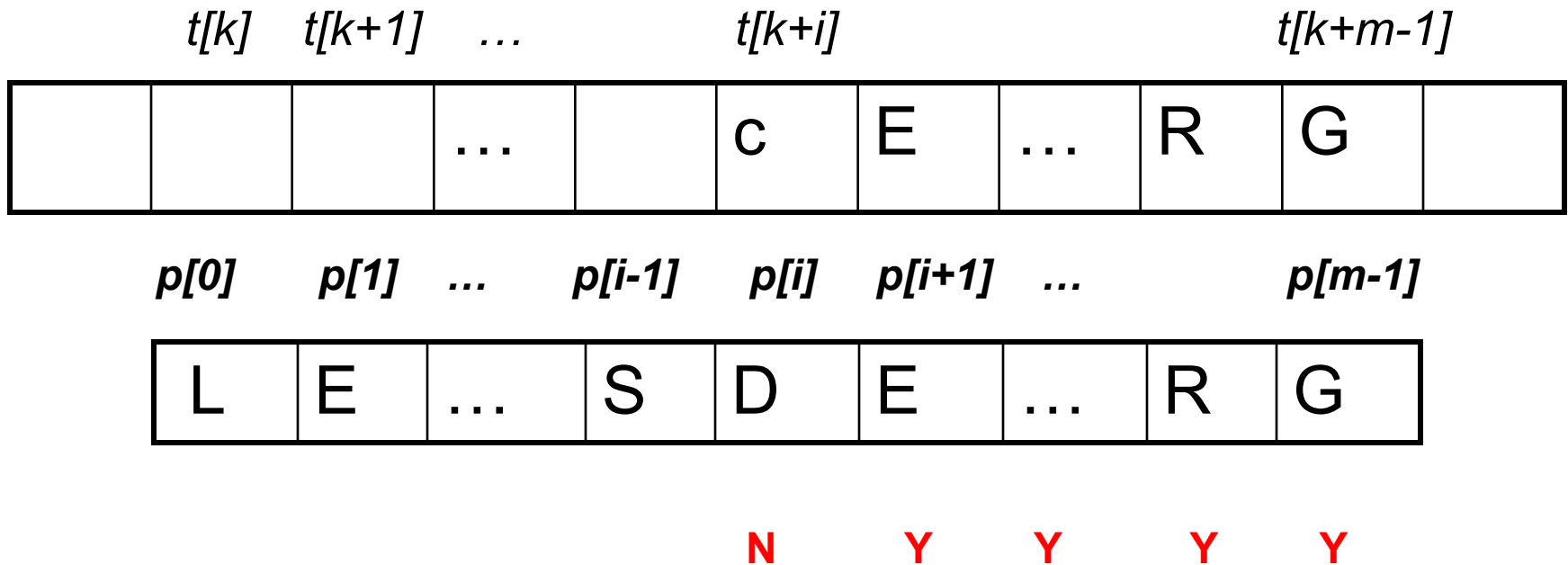
Boyer-Moore example

$t[0]$	$t[1]$	$t[2]$	$t[3]$	$t[4]$	$t[5]$	$t[6]$	$t[7]$	$t[8]$	$t[9]$	$t[10]$
A	B	C	E	F	G	A	B	C	D	E

$p[0]$	$p[1]$	$p[2]$	$p[3]$
A	B	C	D

Y	Y	Y	Y
---	---	---	---

Boyer-Moore : another example



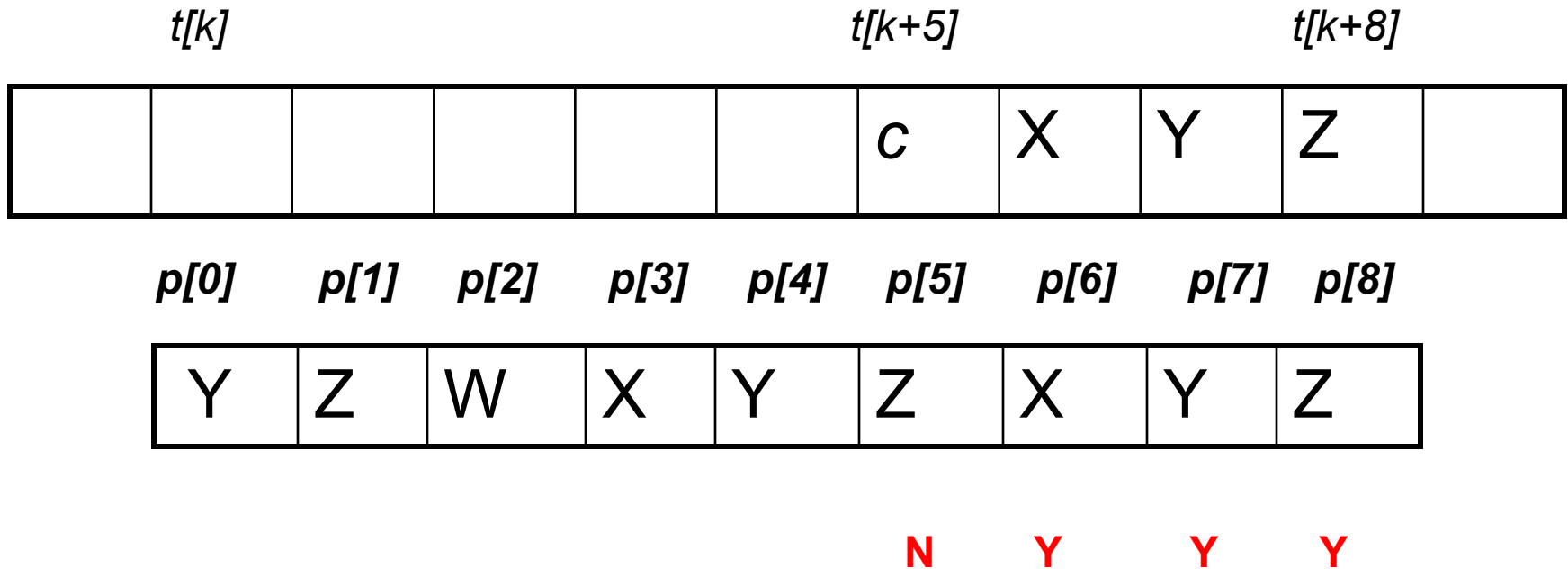
Problem: determine d , the number of boxes that the pattern can be moved to the right.

d should be smallest integer such that $t[k+m-1] = p[m-1-d]$, $t[k+m-2] = p[m-2-d]$, ... $t[k+i] = p[i-d]$

The Boyer-Moore Algorithm

- We said:
 - d should be smallest integer such that:
 - $T[k+m-1] = p[m-1-d]$
 - $T[k+m-2] = p[m-2-d]$
 - $T[k+i] = p[i-d]$
 - Reminder:
 - k = starting index in target string
 - m = length of pattern
 - i = index of mismatch in pattern string
 - Problem: statement is valid only for $d \leq i$
 - Need to ensure that we don't "fall off" the left edge of the pattern

Boyer-Moore : another example



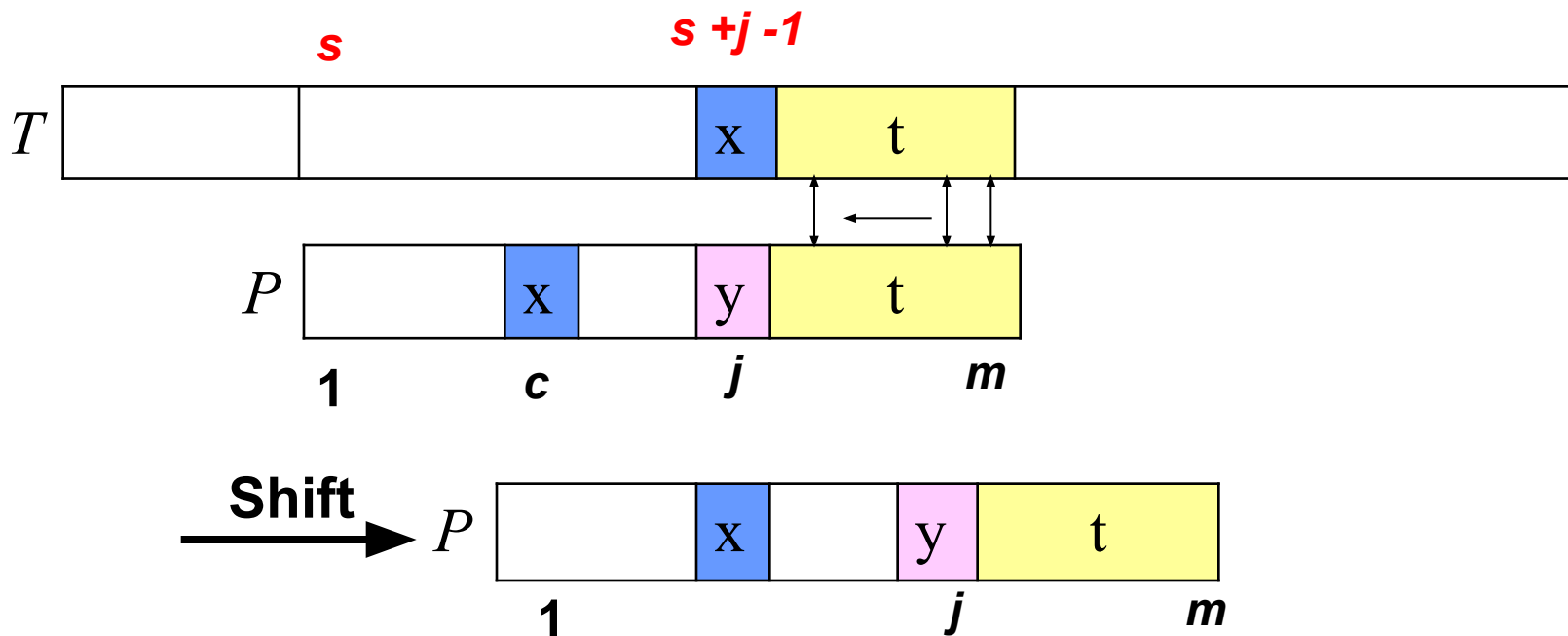
If $c == W$, then d should be 3

If $c == R$, then d should be 7

Bad Character Rule

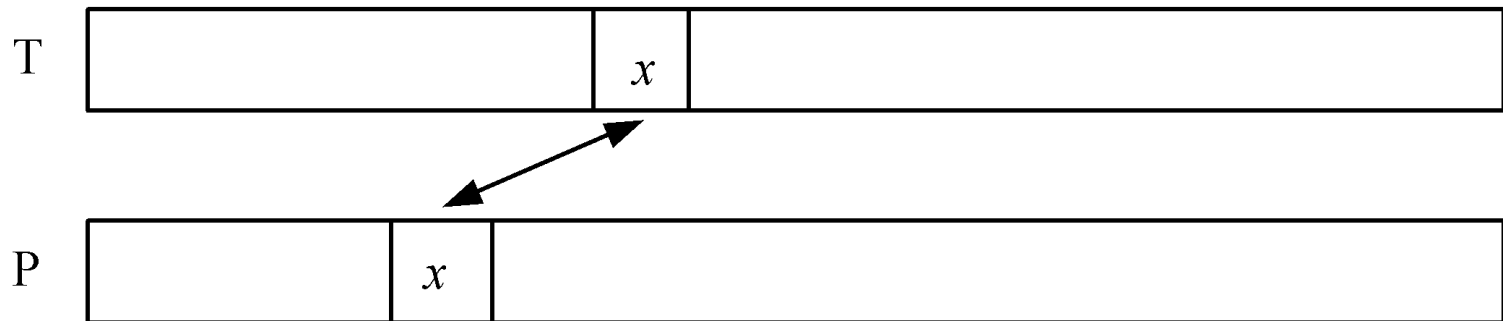
Suppose that P_1 is aligned to T_s now, and we perform a pair-wise comparing between text T and pattern P from right to left. Assume that the first mismatch occurs when comparing T_{s+j-1} with P_j .

Since $T_{s+j-1} \neq P_j$, we move the pattern P to the right such that the largest position c in the left of P_j is equal to T_{s+j-1} . We can shift the pattern at least $(j-c)$ positions right.



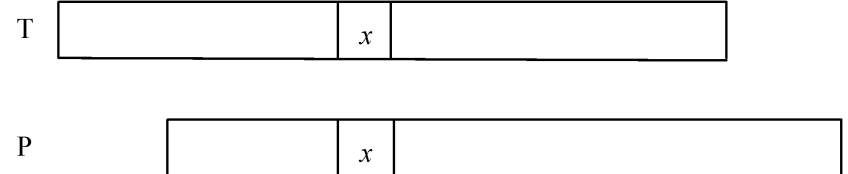
Character Matching Rule

- Bad character rule uses Character Matching Rule.
- For any character x in T , find the nearest x in P which is to the left of x in T .

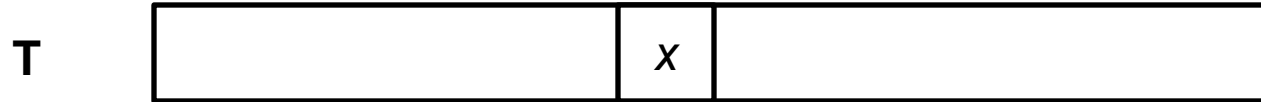


Implication of Character M. Rule

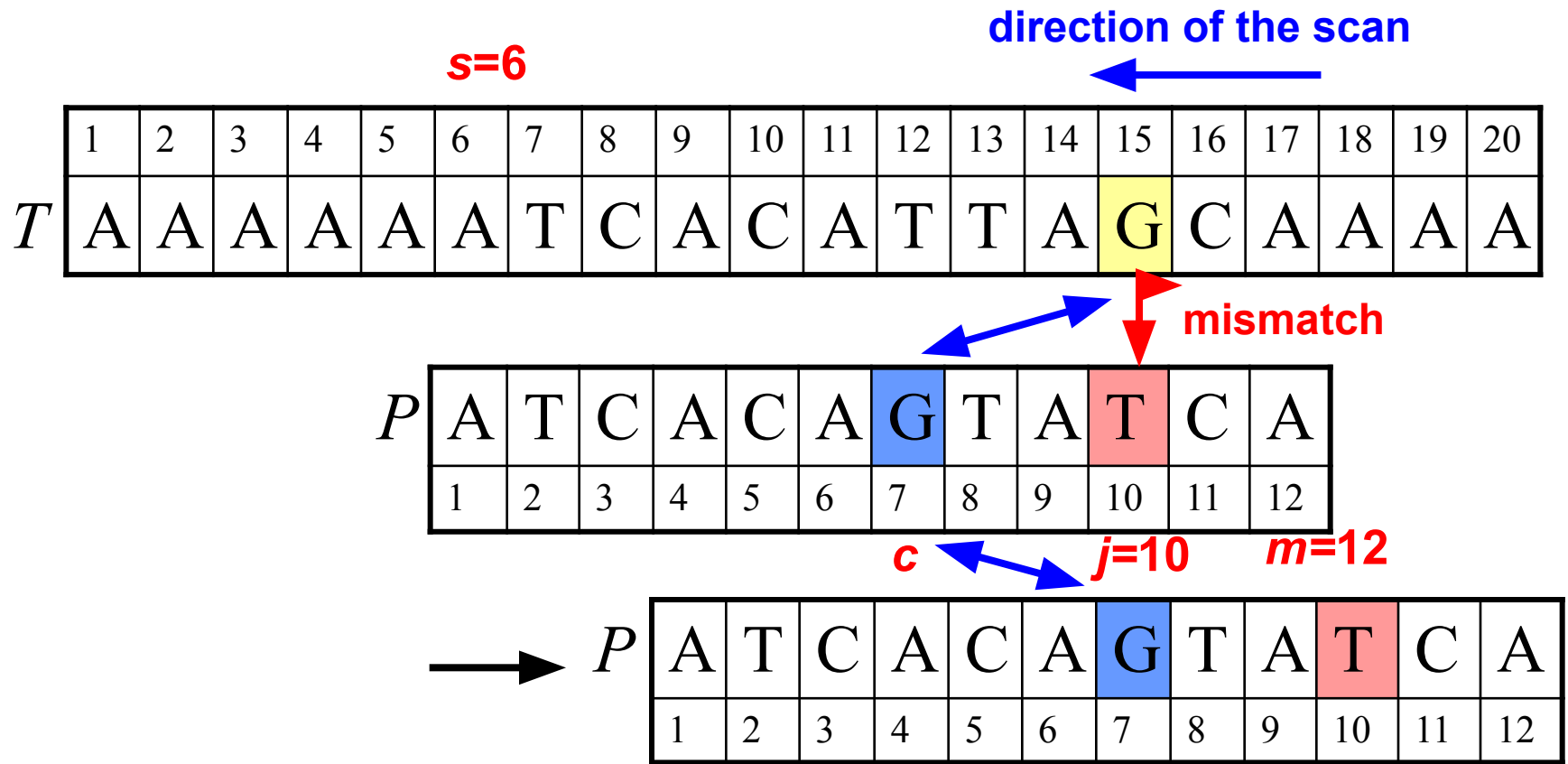
- Case 1. If there is a x in P to the left of T , move P so that the two x 's match.



- Case 2: If no such a x exists in P , move P to the right of x

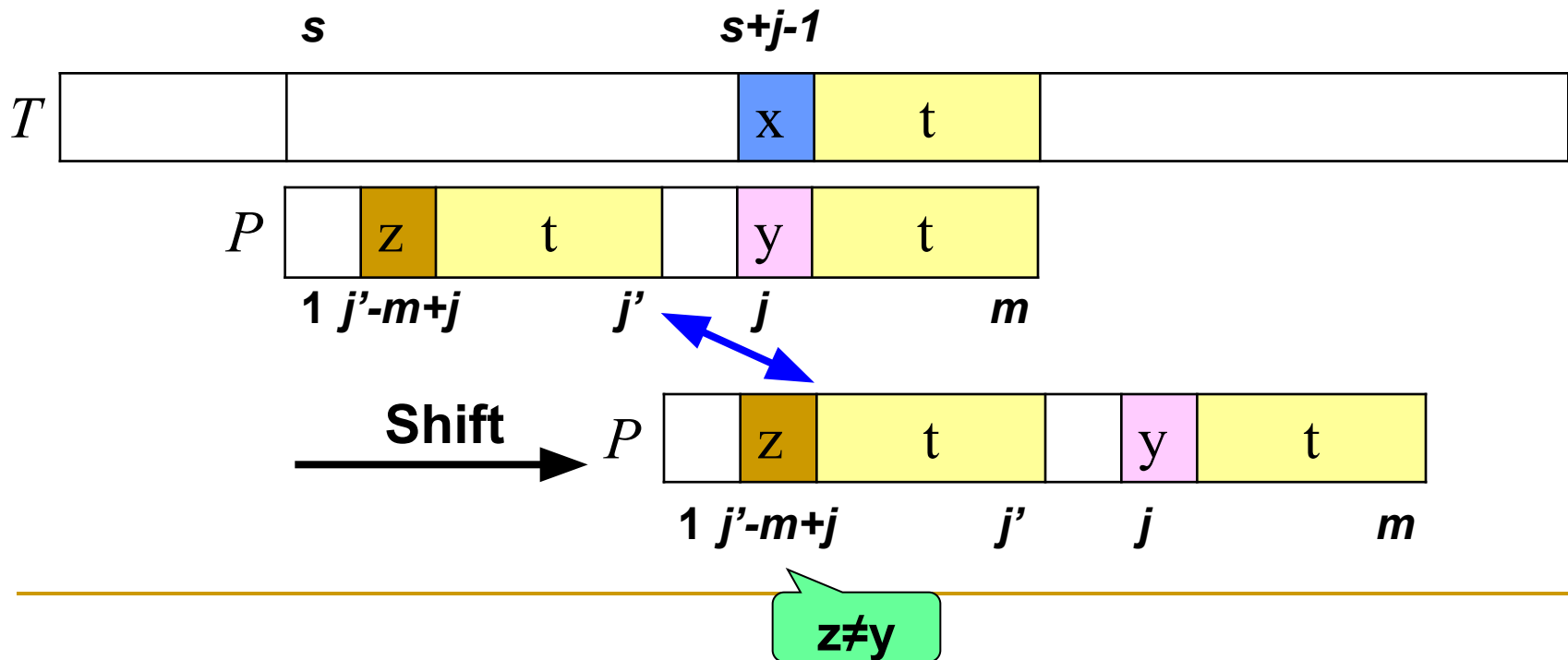


Ex: Suppose that P1 is aligned to T6 now. We compare pairwise between T and P from right to left. Since $T_{16,17} = P_{11,12} = \text{"CA"}$ and $T_{15} = \text{"G"} \neq P_{10} = \text{"T"}$. Therefore, we find the rightmost position $c=7$ in the left of P10 in P such that P_c is equal to "G" and we can move the window at least $(10-7=3)$ positions.



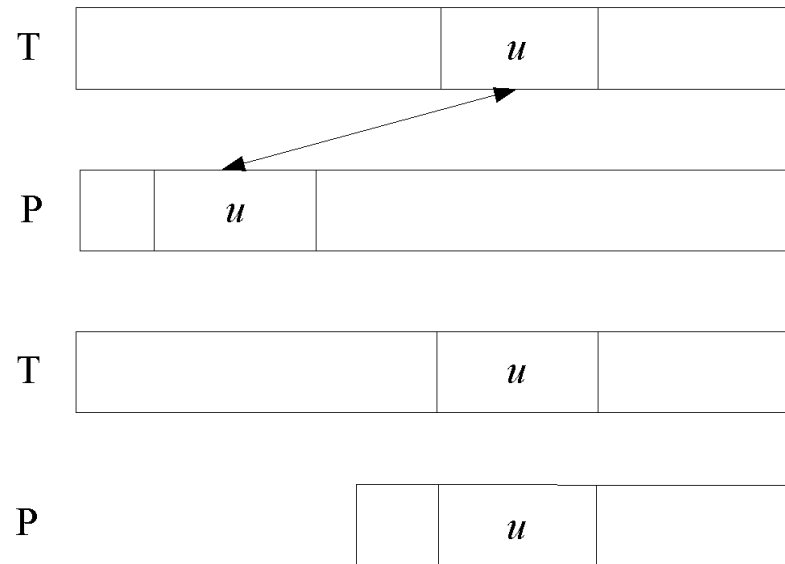
Good Suffix Rule 1

- If a mismatch occurs in T_{s+j-1} , we match T_{s+j-1} with $P_{j'-m+j}$, where j' ($m-j+1 \leq j' < m$) is the **largest position** such that
 - (1) $P_{j+1,m}$ is a suffix of $P_{1,j'}$
 - (2) $P_{j'-(m-j)} \neq P_j$
- We can move the window at least $(m-j')$ position(s).

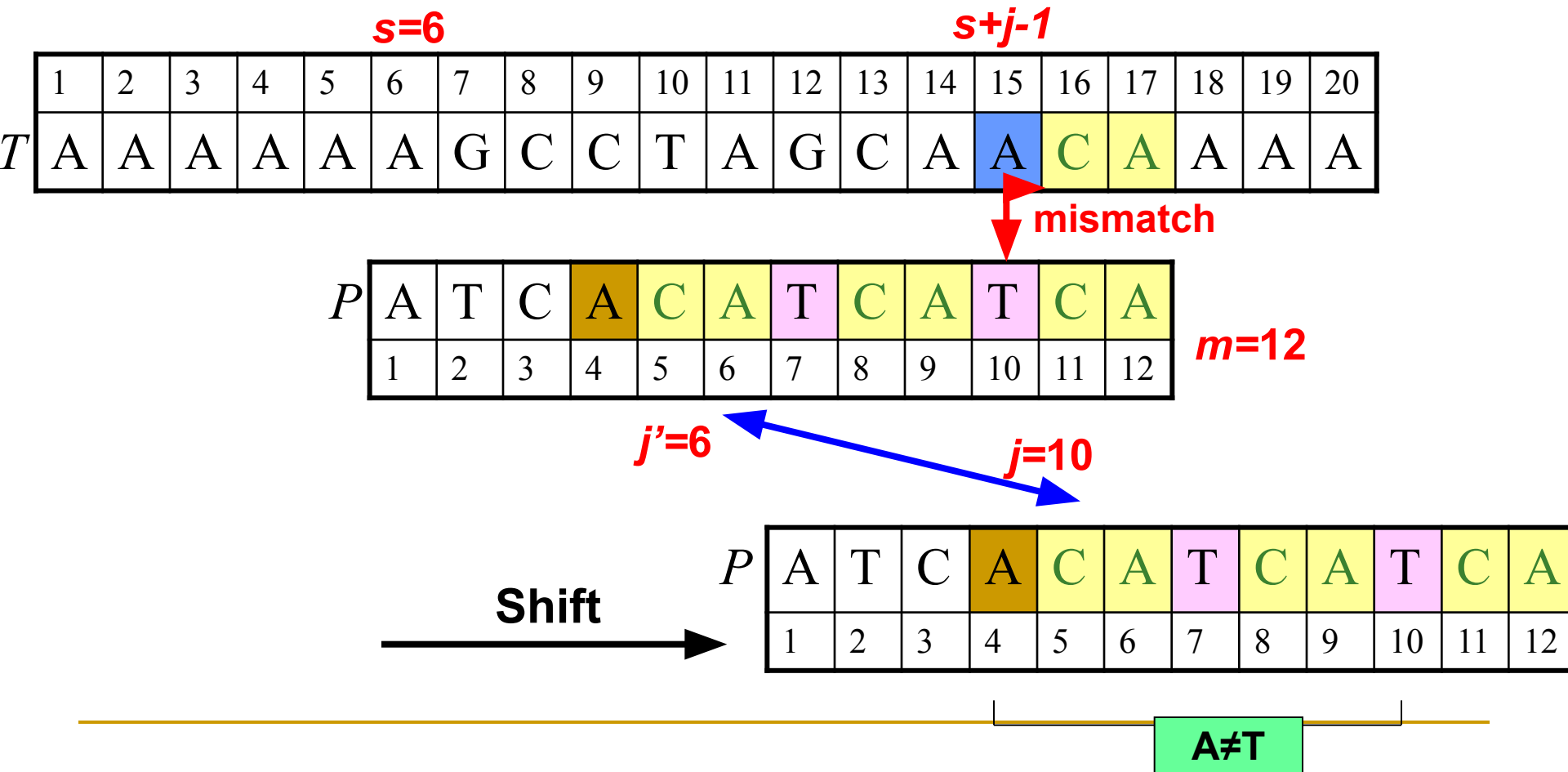


The Substring Matching Rule

- For any substring u in T , find a nearest u in P which is to the left of it. If such a u in P exists, move P ;



Ex: Suppose that $P[1]$ is aligned to $T[6]$ now. We compare pairwise between P and T from right to left. Since $T[16,17] = \text{"CA"} = P[11,12]$ and $T[15] = \text{"A"} \neq P[10] = \text{"T"}$. We find the substring "CA" in the left of $P[10]$ in P such that "CA" is the suffix of $P[1,6]$ and the left character to this substring "CA" in P is not equal to $P[10] = \text{"T"}$. Therefore, we can move the window at least $m-j'$ ($12-6=6$) positions right.

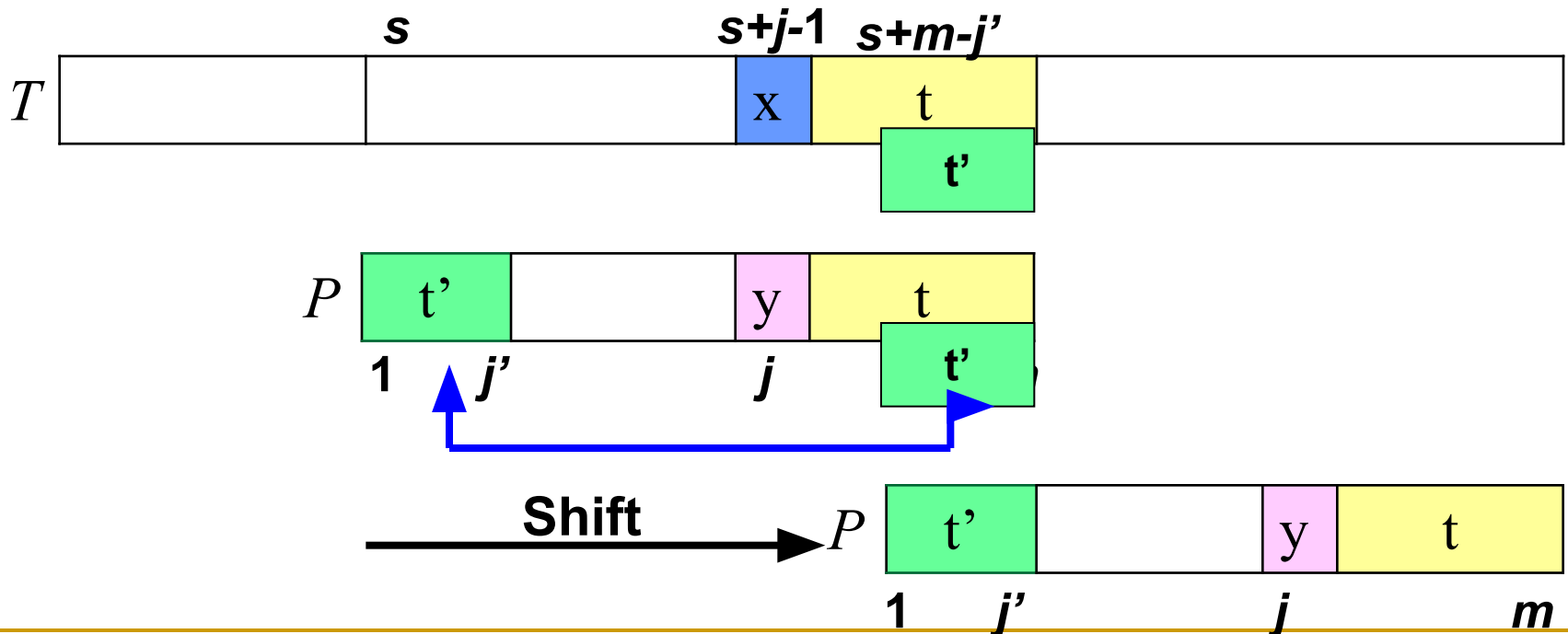


Good Suffix Rule 2

Good Suffix Rule 2 is used only when Good Suffix Rule 1 can not be used. That is, t does not appear in $P[1, j]$. Thus, t is unique in P .

- If a mismatch occurs in T_{s+j-1} , we match $T_{s+m-j'}$ with P_1 , where j' ($1 \leq j' \leq m-j$) is **the largest position** such that

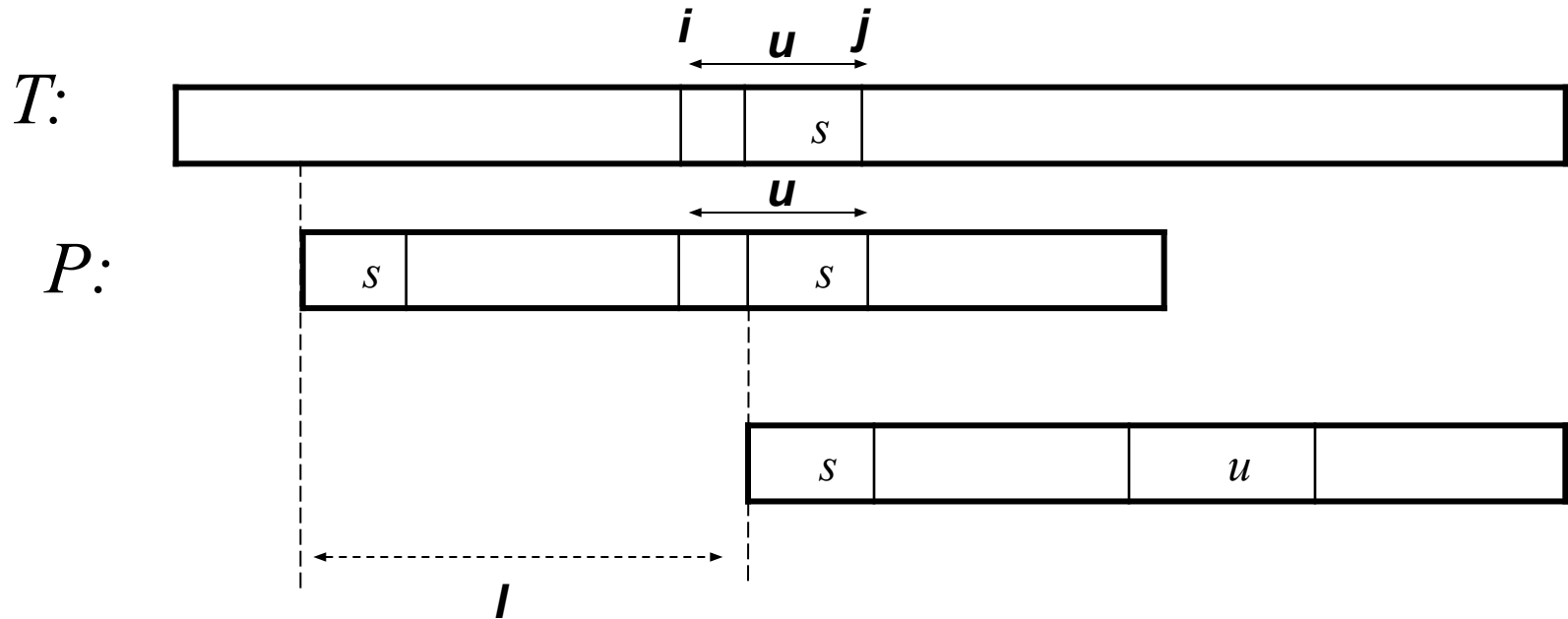
$P_{1,j'}$ is a suffix of $P_{j+1,m}$.



P.S. : t' is suffix of substring t .

Unique Substring Rule

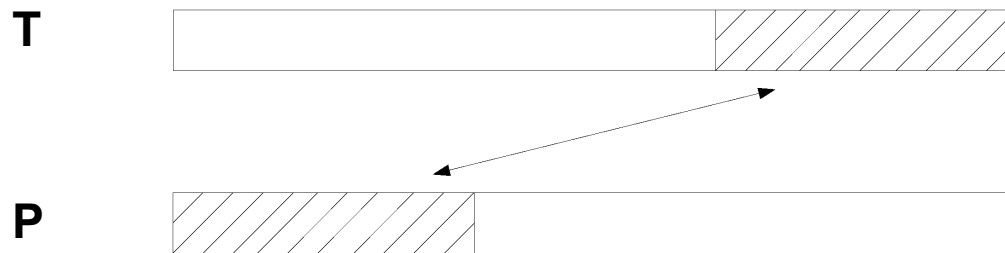
- The substring u appears in P exactly once.
- If the substring s matches with $T_{i,j}$, no matter whether a mismatch occurs in some position of P or not, we can slide the window by l .



The string s is the longest prefix of P which equals to a suffix of u .

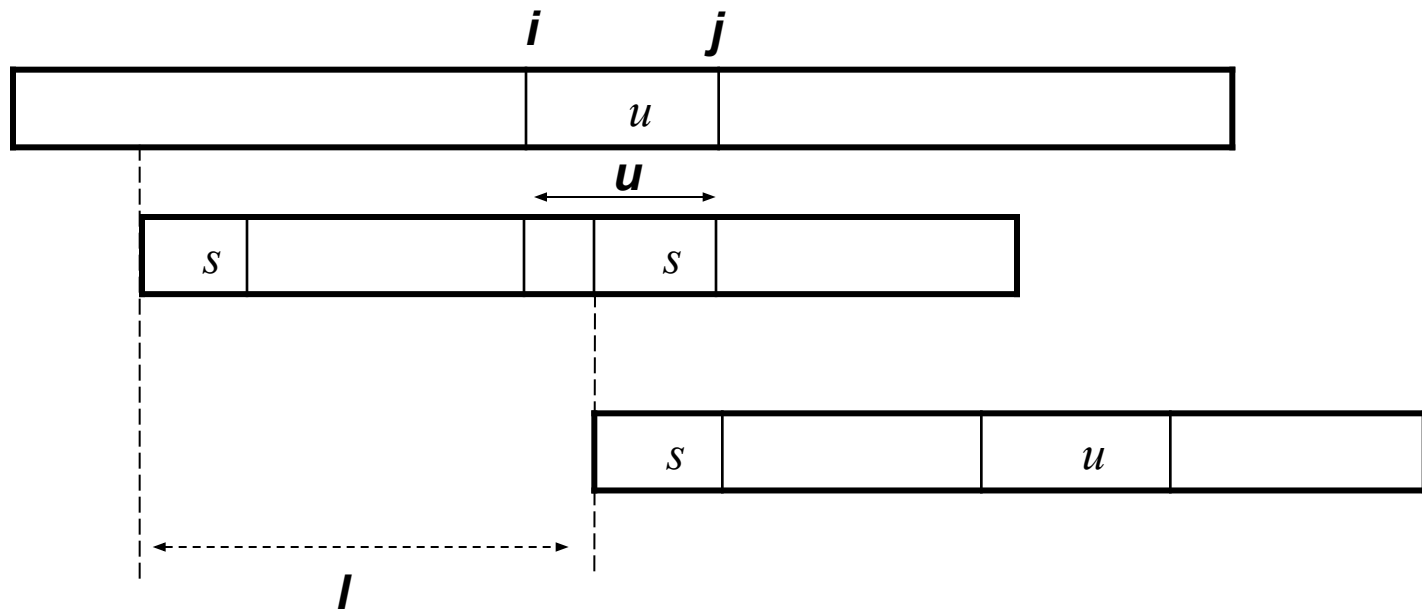
The Suffix to Prefix Rule

- For a window to have any chance to match a pattern, in some way, there must be a suffix of the window which is equal to a prefix of the pattern.

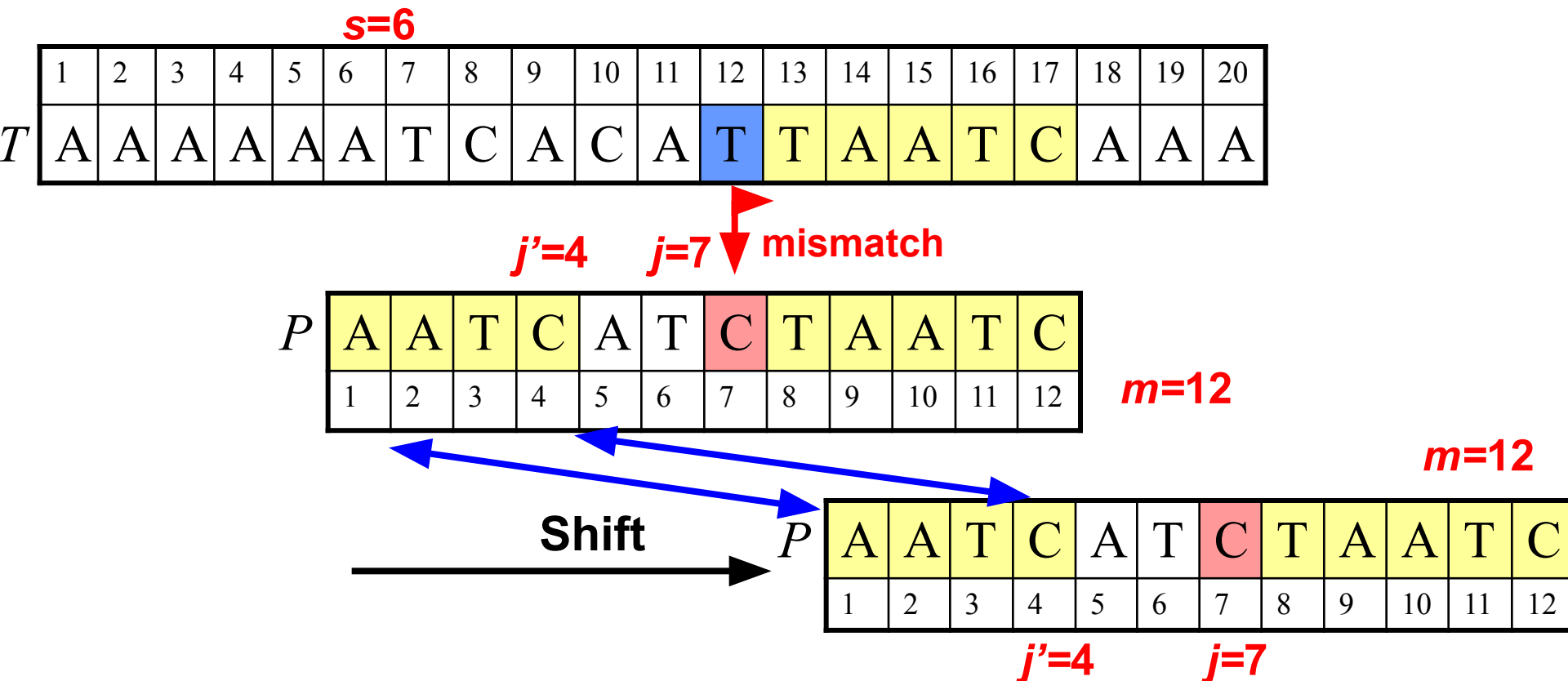


The Suffix to Prefix Rule

- Note that the above rule also uses Rule 1.
- It should also be noted that the unique substring is the shorter and the more right-sided the better.
- A short u guarantees a short (or even empty) s which is desirable.



Ex: Suppose that P_1 is aligned to T_6 now. We compare pairwise between P and T from right to left. Since $T_{12} \neq P_7$ and there is no substring $P_{8,12}$ in left of P_8 to exactly match $T_{13,17}$. We find a longest suffix “AATC” of substring $T_{13,17}$, the longest suffix is also prefix of P . We shift the window such that the last character of prefix substring to match the last character of the suffix substring. Therefore, we can shift at least $12-4=8$ positions.



- Let $B(a)$ be the rightmost position of a in $P[1..j]$. The function will be used for applying *bad character rule*.

j	1	2	3	4	5	6	7	8	9	10	11	12	Σ
P	A	T	C	A	C	A	T	C	A	T	C	A	$B[12]$
													...
													$B[10]$
													...

A	C	G	T
12	11	0	10
9	8	0	10

- We can move our pattern right at least $j - B[j](T_{s+j-1})$ position by above B function.

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T	A	G	C	T	A	G	C	C	T	G	C	A	C	G	T	A	C	A

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A

Move at least
 $10 - B[10](G) = 10$ positions

Let $G_s(j)$ be the largest number of shifts by *good suffix rule* when a mismatch occurs for comparing P_j with some character in T .

- $gs_1(j)$ be the largest k such that $P_{j+1,m}$ is a suffix of $P_{1,k}$ and $P_{k-m+j} \neq P_j$ where $m-j+1 \leq k < m$; 0 if there is no such k .
(gs_1 is for Good Suffix Rule 1)
- $gs_2(j)$ be the largest k such that $P_{1,k}$ is a suffix of $P_{j+1,m}$ where $1 \leq k \leq m-j$; 0 if there is no such k .
(gs_2 is for Good Suffix Rule 2.)
- **$Gs(j) = m - \max\{gs_1, gs_2\}$** , if $j = m$, $Gs(j)=1$.

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
gs_1	0	0	0	0	0	0	9	0	0	6	1	0
gs_2	4	4	4	4	4	4	4	4	1	1	1	0
Gs	8	8	8	8	8	8	3	8	11	6	11	1

$$gs_1(7)=9$$

$\because P_{8,12}$ is a suffix of $P_{1,9}$ and $P_4 \neq P_7$

$$gs_2(7)=4$$

$\because P_{1,4}$ is a suffix of $P_{8,12}$

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
gs_1	0	0	0	0	0	0	9	0	0	6	1	0
gs_2	4	4	4	4	4	4	4	4	1	1	1	0
G_s	8	8	8	8	8	8	3	8	11	6	11	1

$s=6$

$s+j-1$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	A	A	A	A	A	A	G	C	C	T	A	G	C	A	A	C	A	A	A	A

mismatch

P	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

$m=12$

$j'=6$

$j=10$

Shift

P	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

Time Complexity

- Use bad character or good suffix rule
 - The one that skips more
- The preprocessing phase in $O(m+\Sigma)$ complexity
- If you are searching for ALL matches, worst case:
 - $O(mn)$ when P is in T at all positions
 - $T=AAAAAAAAAAAAA; P=AAAA$
 - $O(m+n)$ when P is not in T

BRUTE FORCE – EXAMPLE #2

Brute force

T = ABABABCABABABCABABAC
P = ABABAC

X

Comparisons: 6

T = ABABABCABABABCABABAC
P = ABABAC

X

Comparisons: 1

T = ABABABCABABABCABABAC
P = ABABAC

X

Comparisons: 5

T = ABABABCABABABCABABAC
P = ABABAC

X

Comparisons: 1

T = ABABABCABABABCABABAC
P = ABABAC

X

Comparisons: 3

Brute force

T = ABABABCABABABCABABAC

P = ABABAC

X

Comparisons: 1

T = ABABABCABABABCABABAC

P = ABABAC

X

Comparisons: 1

T = ABABABCABABABCABABAC

P = ABABAC

X

Comparisons: 6

T = ABABABCABABABCABABAC

P = ABABAC

X

Comparisons: 1

T = ABABABCABABABCABABAC

P = ABABAC

X

Comparisons: 5

Brute force

T = ABABABCABABABCABABAC

P = ABABAC

x

Comparisons: 1

T = ABABABCABABABCABABAC

P = ABABAC

x

Comparisons: 3

T = ABABABCABABABCABABAC

P = ABABAC

x

Comparisons: 1

T = ABABABCABABABCABABAC

P = ABABAC

x

Comparisons: 1

T = ABABABCABABABCABABAC

P = ABABAC

match

Comparisons: 6

Total comparisons: 41

KMP – EXAMPLE #2

Knuth-Morris-Pratt

T = ABABABCABABABCABABAC

P = ABABAC

Reminder: $F(j)$ is defined as the size of
the largest prefix of
 $P[0..j]$ that is also a suffix of $P[1..j]$

	A	B	A	B	A	C
F	0	0	1	2	3	0

Algorithm *failureFunction*(P)

$F[0] \leftarrow 0$

$i \leftarrow 1$

$j \leftarrow 0$

$m \leftarrow \text{length}(P)$

while $i < m$

if $P[i] = P[j]$

{we have matched $j + 1$ chars}

$F[i] \leftarrow j + 1$

$i \leftarrow i + 1$

$j \leftarrow j + 1$

else if $j > 0$ **then**

{use failure function to shift P }

$j \leftarrow F[j - 1]$

else

$F[i] \leftarrow 0$ { no match }

$i \leftarrow i + 1$

Knuth-Morris-Pratt

	0	1	2	3	4	5
	A	B	A	B	A	C
F	0	0	1	2	3	0

T = ABABABCABABABCABABAC

P = ABABAC

x

j = 5, i = 5

New j = F[4] = 3 (shift 5-3 = 2)

New i = 5 (no change)

T = ABABABCABABABCABABAC
P = ABABAC

i=5
j=3

Algorithm *KMPMatch*(T, P)

F ← *failureFunction*(P)

i ← 0

j ← 0

while *i* < *n*

if *T*[*i*] = *P*[*j*]

if *j* = *m* - 1

return *i* - *j* { match }

else

i ← *i* + 1

j ← *j* + 1

else

if *j* > 0

j ← *F*[*j* - 1]

else

i ← *i* + 1

j ← 0

return -1 { no match }

Comparisons = 6

Knuth-Morris-Pratt

	0	1	2	3	4	5
	A	B	A	B	A	C
F	0	0	1	2	3	0

T = ABABABCABABABCABABAC

P = ABABAC

x

j = 4, i = 6

New j = F[3] = 2 (shift 4-2 = 2)

New i = 6 (no change)

T = ABABABCABABABCABABAC

P = ABABAC

i=6

j=2

Algorithm *KMPMatch*(T, P)

$F \leftarrow \text{failureFunction}(P)$

$i \leftarrow 0$

$j \leftarrow 0$

while $i < n$

if $T[i] = P[j]$

if $j = m - 1$

return $i - j$ { match }

else

$i \leftarrow i + 1$

$j \leftarrow j + 1$

else

if $j > 0$

$j \leftarrow F[j - 1]$

else

$i \leftarrow i + 1$

$j \leftarrow 0$

return -1 { no match }

Comparisons = 6+2 = 8

Knuth-Morris-Pratt

	0	1	2	3	4	5
	A	B	A	B	A	C
F	0	0	1	2	3	0

T = ABABABCABABABCABABAC

P = ABABAC

x

j = 2, i = 6

New j = F[1] = 0 (shift 2-0 = 2)

New i = 6 (no change)

T = ABABABCABABABCABABAC
P = ABABAC

i=6
j=0

Algorithm *KMPMatch*(T, P)

$F \leftarrow \text{failureFunction}(P)$

$i \leftarrow 0$

$j \leftarrow 0$

while $i < n$

if $T[i] = P[j]$

if $j = m - 1$

return $i - j$ { match }

else

$i \leftarrow i + 1$

$j \leftarrow j + 1$

else

if $j > 0$

$j \leftarrow F[j - 1]$

else

$i \leftarrow i + 1$

$j \leftarrow 0$

return -1 { no match }

Comparisons = 8 + 1 = 9

Knuth-Morris-Pratt

	0	1	2	3	4	5
	A	B	A	B	A	C
F	0	0	1	2	3	0

T = ABABABCABABABCABABAC

P = ABABAC

x

j = 0, i = 6

New j = 0

New i = 7 (shift 1)

T = ABABABCABABABCABABAC

P = ABABAC

i=7

j=0

Algorithm *KMPMatch*(T, P)

$F \leftarrow \text{failureFunction}(P)$

$i \leftarrow 0$

$j \leftarrow 0$

while $i < n$

if $T[i] = P[j]$

if $j = m - 1$

return $i - j$ { match }

else

$i \leftarrow i + 1$

$j \leftarrow j + 1$

else

if $j > 0$

$j \leftarrow F[j - 1]$

else

$i \leftarrow i + 1$

$j \leftarrow 0$

return -1 { no match }

Comparisons = 9+1 = 10

Knuth-Morris-Pratt

	0	1	2	3	4	5
	A	B	A	B	A	C
F	0	0	1	2	3	0

T = ABABABCABABABCABABAC

P = ABABAC

x

j = 5, i = 12

New j = F[4] = 3 (shift 5-3=2)

New i = 12 (no change)

T = ABABABCABABABCABABAC

P = ABABAC

i=12

j=3

Algorithm *KMPMatch*(T, P)

$F \leftarrow \text{failureFunction}(P)$

$i \leftarrow 0$

$j \leftarrow 0$

while $i < n$

if $T[i] = P[j]$

if $j = m - 1$

return $i - j$ { match }

else

$i \leftarrow i + 1$

$j \leftarrow j + 1$

else

if $j > 0$

$j \leftarrow F[j - 1]$

else

$i \leftarrow i + 1$

$j \leftarrow 0$

return -1 { no match }

Comparisons = 10+6=16

Knuth-Morris-Pratt

	0	1	2	3	4	5
	A	B	A	B	A	C
F	0	0	1	2	3	0

T = ABABABCABABABCABABAC

P = ABABAC

x

j = 4, i = 13

New j = F[3] = 2 (shift 4-2=2)

New i = 13 (no change)

T = ABABABCABABABCABABAC
P = ABABAC

i=13

j=2

Algorithm *KMPMatch*(T, P)

$F \leftarrow \text{failureFunction}(P)$

$i \leftarrow 0$

$j \leftarrow 0$

while $i < n$

if $T[i] = P[j]$

if $j = m - 1$

return $i - j$ { match }

else

$i \leftarrow i + 1$

$j \leftarrow j + 1$

else

if $j > 0$

$j \leftarrow F[j - 1]$

else

$i \leftarrow i + 1$

$j \leftarrow 0$

return -1 { no match }

Comparisons = 16+2=18

Knuth-Morris-Pratt

	0	1	2	3	4	5
	A	B	A	B	A	C
F	0	0	1	2	3	0

T = ABABABCABABABCABABAC

P = ABABAC

x

j = 2, i = 13

New j = F[1] = 0 (shift 2-0=0)

New i = 13 (no change)

T = ABABABCABABABCABABAC

P = ABABAC

i=13

j=0

Algorithm *KMPMatch*(T, P)

$F \leftarrow \text{failureFunction}(P)$

$i \leftarrow 0$

$j \leftarrow 0$

while $i < n$

if $T[i] = P[j]$

if $j = m - 1$

return $i - j$ { match }

else

$i \leftarrow i + 1$

$j \leftarrow j + 1$

else

if $j > 0$

$j \leftarrow F[j - 1]$

else

$i \leftarrow i + 1$

$j \leftarrow 0$

return -1 { no match }

Comparisons = 18+1 = 19

Knuth-Morris-Pratt

	0	1	2	3	4	5
	A	B	A	B	A	C
F	0	0	1	2	3	0

T = ABABABCABABABCABABAC

P = ABABAC

x

j = 0, i = 13

New j = 0 (no shift)

New i = 14 (shift by 1)

T = ABABABCABABABCABABAC

P = ABABAC

i=14
j=0

Algorithm *KMPMatch*(T, P)

$F \leftarrow \text{failureFunction}(P)$

$i \leftarrow 0$

$j \leftarrow 0$

while $i < n$

if $T[i] = P[j]$

if $j = m - 1$

return $i - j$ { match }

else

$i \leftarrow i + 1$

$j \leftarrow j + 1$

else

if $j > 0$

$j \leftarrow F[j - 1]$

else

$i \leftarrow i + 1$

$j \leftarrow 0$

return -1 { no match }

Comparisons = 19+1+6=26

Brute force: 41

BOYER-MOORE – EXAMPLE

#2

Boyer-Moore

T = ABABABCABABABCABABAC
P = ABABAC

Comparison: 1

x

T = ABABA**B**CABABABCABABAC
P = ABA**B**AC

Bad character rule

T = ABABABCABABABCABABAC
P = ABABAC

Good suffix rule
Note: no suffix matches in
the previous step!!!

Pick bad character rule shift:

T = ABABABCABABABCABABAC
P = ABABAC

Boyer-Moore

T = ABABABCABABABCABABAC

P = ABABAC

Comparison: 1

x

T = ABABABCABABABCABABAC

P = ABABAC

Bad character rule

T = ABABABCABABABCABABAC

P = ABABAC

Good suffix rule

Note: no suffix matches in the previous step!!!

Pick either shift:

T = ABABABCABABABCABABAC

P = ABABAC

Boyer-Moore

T = ABABABCABABABCABABAC
P = ABABAC

Comparison: 1

x

T = ABABABCABABABCABABAC
P = ABABAC

Bad character rule

T = ABABABCABABABCABABAC
P = ABABAC

Good suffix rule
Note: no suffix matches in
the previous step!!!

Pick bad character rule shift:

T = ABABABCABABABCABABAC
P = ABABAC

Boyer-Moore

T = ABABABCABABABCABABAC

P = ABABAC

Comparison: 1

X

T = ABABABCABA**B**ABCABABAC

P = ABAB**B**AC

Bad character rule

T = ABABABCABABABCABABAC

P = ABABAC

Good suffix rule

Note: no suffix matches in the previous step!!!

Pick bad character rule shift:

T = ABABABCABABABCABABAC

P = ABABAC

Boyer-Moore

T = ABABABCABABABCABABAC
P = ABABAC

Comparison: 1

x

T = ABABABCABABA**B**CABABAC
P = ABAB**B**AC

Bad character rule

T = ABABABCABABABCABABAC
P = ABABAC

Good suffix rule
Note: no suffix matches in
the previous step!!!

Pick bad character rule shift:

T = ABABABCABABABCABABAC
P = ABABAC

Boyer-Moore

T = ABABABCABABABCABABAC
P = ABABAC

Comparison: 1

x

T = ABABABCABABABCABABAC
P = ABABAC

Bad character rule

T = ABABABCABABABCABABAC
P = ABABAC

Good suffix rule

Pick either:

T = ABABABCABABABCABABAC
P = ABABAC

Boyer-Moore

T = ABABABCABABABCABABAC
P = ABABAC

Comparison: 1

x

T = ABABABCABABABCABABAC
P = ABABAC

Bad character rule

T = ABABABCABABABCABABAC
P = ABABAC

Good suffix rule

Pick bad character rule shift:

T = ABABABCABABABCABABAC
P = ABABAC

Boyer-Moore

T = ABABABCABABABCABABAC
P = ABABAC

Comparison: 1

x

T = ABABABCABABABCABA**B**AC
P = ABABA**B**AC

Bad character rule

T = ABABABCABABABCABABAC
P = ABABAC

Good suffix rule

Pick bad character rule shift and match:

T = ABABABCABABABCABABAC
P = ABABAC

Comparison: 6

Total comparisons: 14 -- KMP: 26 Brute Force: 41

BOYER-MOORE – EXAMPLE

#3

Boyer-Moore

T = ABABABCABABABCBCBAB
P = ABCBAB

x

Comparison: 4

T = ABABABCABABABCBCBAB
P = ABCBAB

Bad character rule

T = ABABABCABABABCBCBAB
P = ABCBAB

Good suffix rule 2
Suffix (BAB) = prefix(P) = AB

Pick good suffix rule 2:

T = ABABABCABABABCBCBAB
P = ABCBAB

Boyer-Moore

T = ABABABCABABABCBCBAB
P = ABCBAB

Comparison: 1

x

T = ABABABCABABABCBCBAB
P = ABCBAB

Bad character rule

T = ABABABCABABABCBCBAB
P = ABCBAB

Good suffix rule
No suffix match in previous
step

Pick either:

T = ABABABCABABABCBCBAB
P = ABCBAB

Boyer-Moore

T = ABABABCABABABCBCBAB
P = ABCBAB

Comparison: 4

x

T = ABABABCABABABCBCBAB
P = ABCBAB

Bad character rule

T = ABABABCABABABCBCBAB
P = ABCBAB

Good suffix rule 2
Suffix (BAB) = prefix(P) = AB

Pick good suffix rule 2:

T = ABABABCABABABCBCBAB
P = ABCBAB

Boyer-Moore

T = ABABABCABABABCBCBAB
P = ABCBAB

Comparison: 1

x

T = ABABABCABABABC**A**BCBAB
P = ABCB**A**B

Bad character rule

T = ABABABCABABABCBCBAB
P = ABCBAB

Good suffix rule
No suffix match in previous
step

Pick either:

T = ABABABCABABABCBCBAB
P = ABCBAB

Boyer-Moore

T = ABABABCABABABC**AB**CBAB
P = ABCBAB

Comparison: 3

X

T = ABABABCABABAB**C**ABCBAB
P = ABC**C**BAB

Bad character rule

T = ABABABCABABABC**AB**CBAB
P = **AB**CBAB

Good suffix rule 1 or 2
Suffix (AB) = prefix(P) = AB

Pick good suffix rule and match:

T = ABABABCABABABCABCBAB
P = ABCBAB

Comparison: 6

Brute force & KMP for this example: exercise

Total comparisons: 19

Demonstration

- Knuth-Morris-Pratt

- <https://cmps-people.ok.ubc.ca/ylucet/DS/KnuthMorrisPratt.html>

- Boyer-Moore

- <https://dwnusbaum.github.io/boyer-moore-demo/>
-

Z-Algorithm
