EXACT STRING MATCHING

Eileen Kraemer

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 <- |t|
m <- |p|
i <= 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]*	*t10]*
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]	t10]
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

t[0]	*t[1]*	*t[2]*	*t[3]*	*t[4]*	*t[5]*	*t[6]*	*t[7]*	*t[8]*	*t[9]*	*t10]*
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

t[0]	*t[1]*	*t[2]*	*t[3]*	*t[4]*	*t[5]*	*t[6]*	*t[7]*	*t[8]*	*t[9]*	*t10]*
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

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t10]
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

t[0]	*t[1]*	*t[2]*	*t[3]*	*t[4]*	*t[5]*	*t[6]*	*t[7]*	*t[8]*	*t[9]*	*t10]*
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

t[0]	*t[1]*	*t[2]*	*t[3]*	*t[4]*	*t[5]*	*t[6]*	*t[7]*	*t[8]*	*t[9]*	*t10]*
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

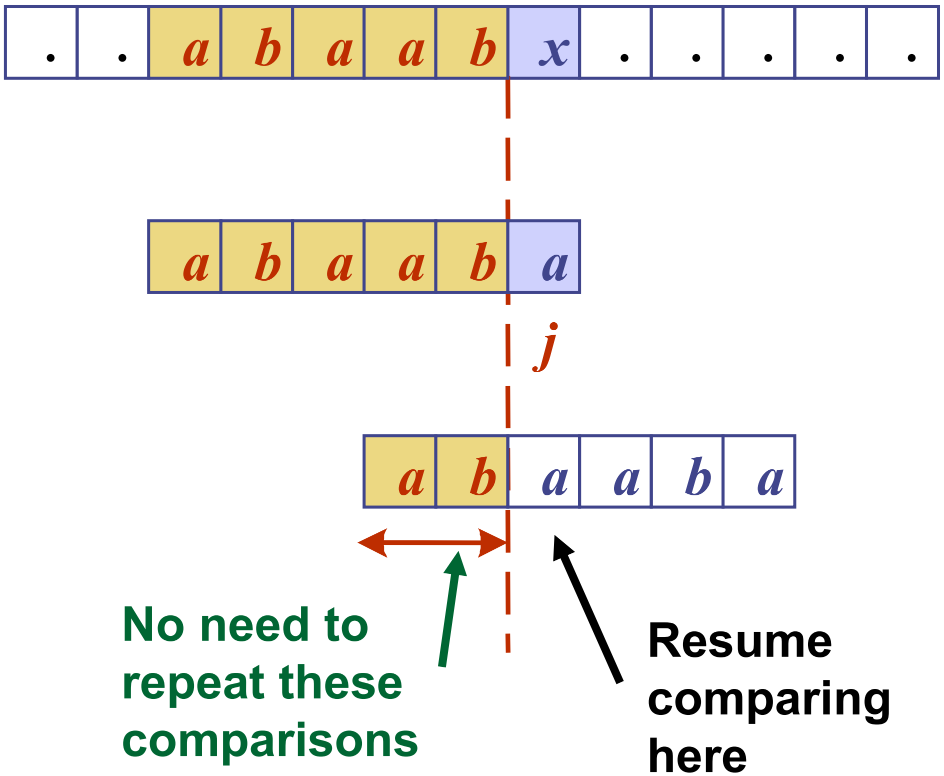
- Worst case:
  - Pattern string always matches completely except for last character
  - Example: search for XXXXXXY in target string of XXXXXXXXXXXXXXXXXXXX
  - Outer loop executed once for every character in target string
  - Inner loop executed once for every character in pattern
  - O(mn), where 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

The KMP Algorithm - Motivation

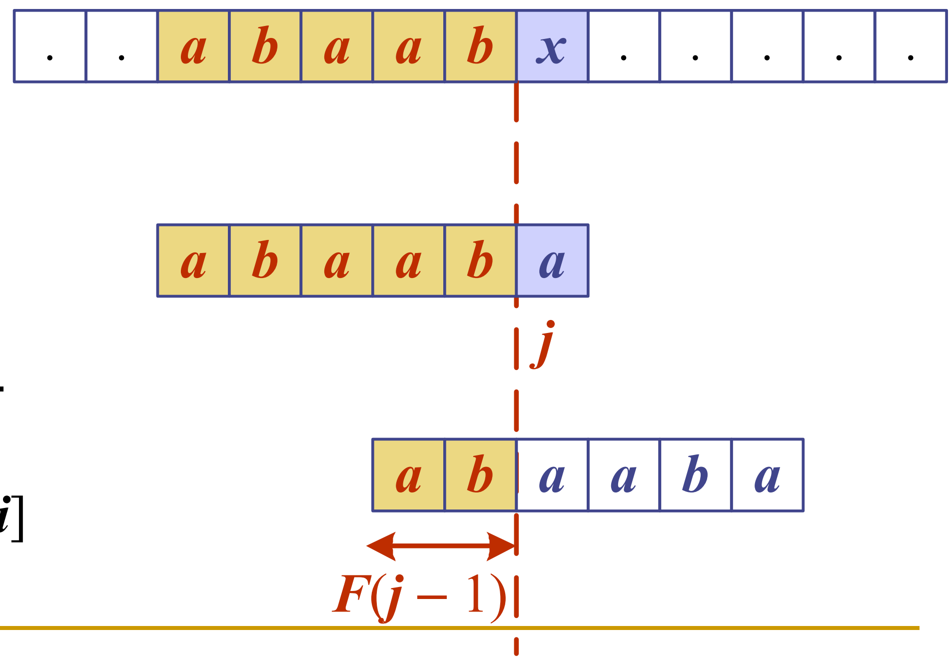
- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix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



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 size 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 $O(m + n)$

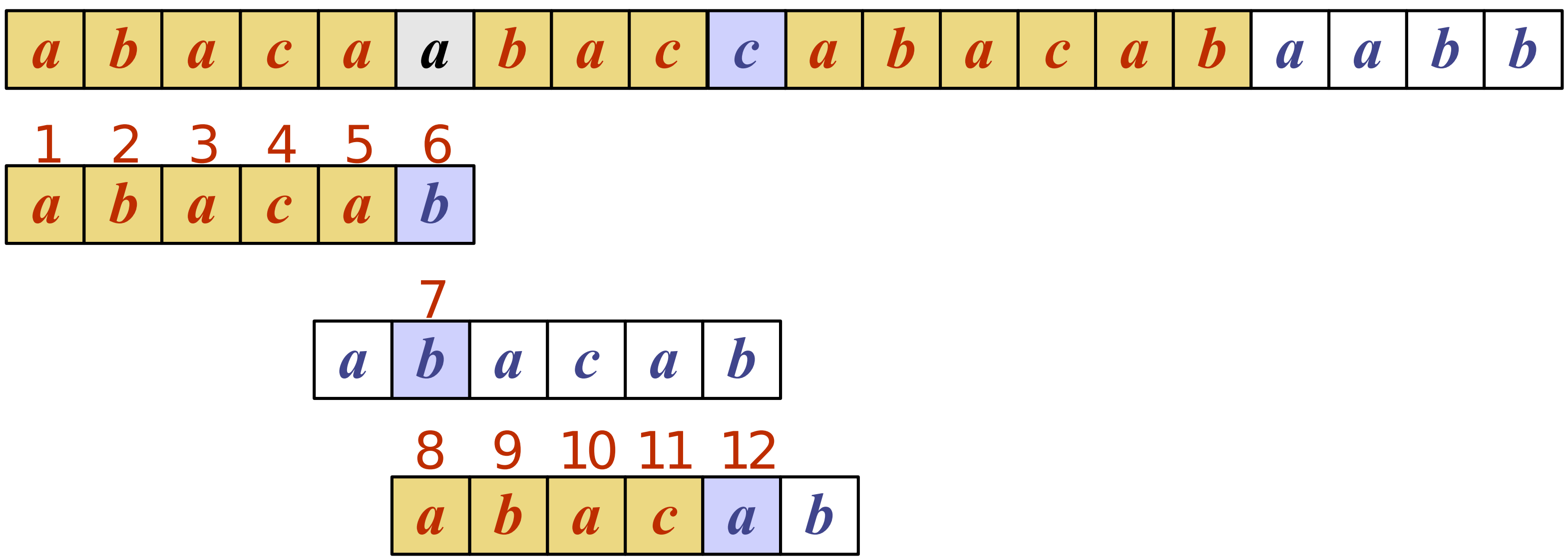
```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m − 1
                return i − j { match }
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0
                j ←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

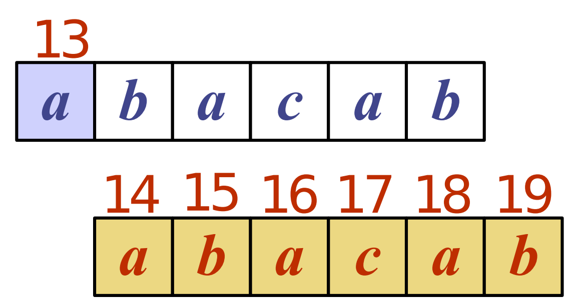
Computing the Failure Function

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

Example





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

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]*	*t10]*
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]	t10]
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]*	*t10]*
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

	t[k]	*t[k+1]*	*…*		*t[k+i]*				*t[k+m-1]*	
			…		c	E	…	R	G	

p[0]	***p[1]***	***…***	***p[i-1]***	***p[i]***	***p[i+1]***	***…***		***p[m-1]***
L	E	…	S	D	E	…	R	G
				N	**Y**	**Y**	**Y**	**Y**

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<= i
    - Need to ensure that we don't "fall off" the left edge of the pattern

Boyer-Moore : another example

t[k] *t[k+5]* *t[k+8]*

						c	X	Y	Z	

p[0]	***p[1]***	***p[2]***	***p[3]***	***p[4]***	***p[5]***	***p[6]***	***p[7]***	***p[8]***
Y	Z	W	X	Y	Z	X	Y	Z
					N	**Y**	**Y**	**Y**

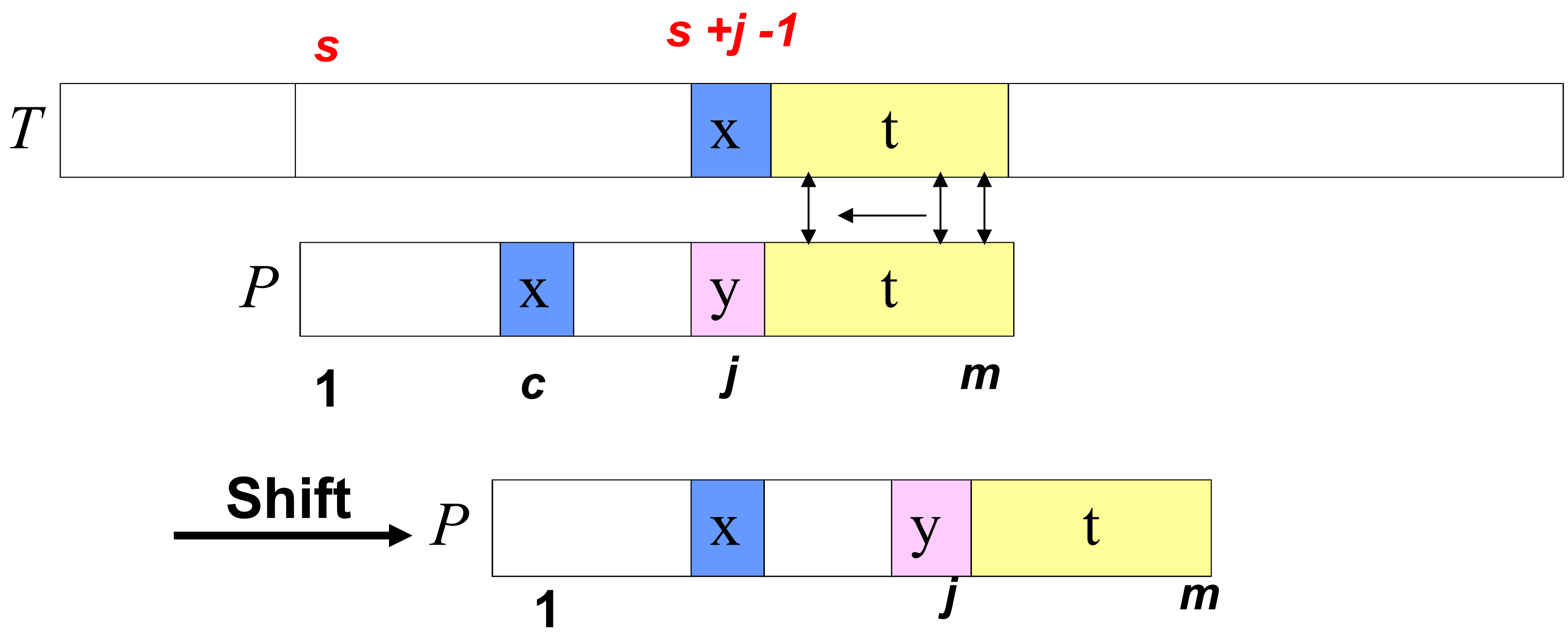
If c == W, then d should be 3

If c == R, then d should be 7

Bad Character Rule

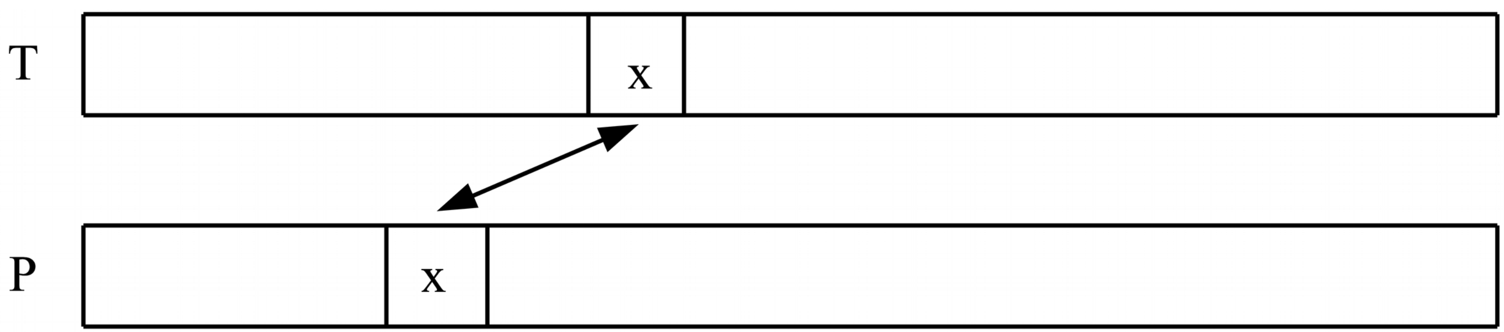
Suppose that P_1 is aligned to T_s now, and we perform a pair-wise comparing between text *T* and pattern *P* from right to left. Assume that the first mismatch occurs when comparing T_{s+j-1} with P_j.

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.



Rule 2-1: Character Matching Rule

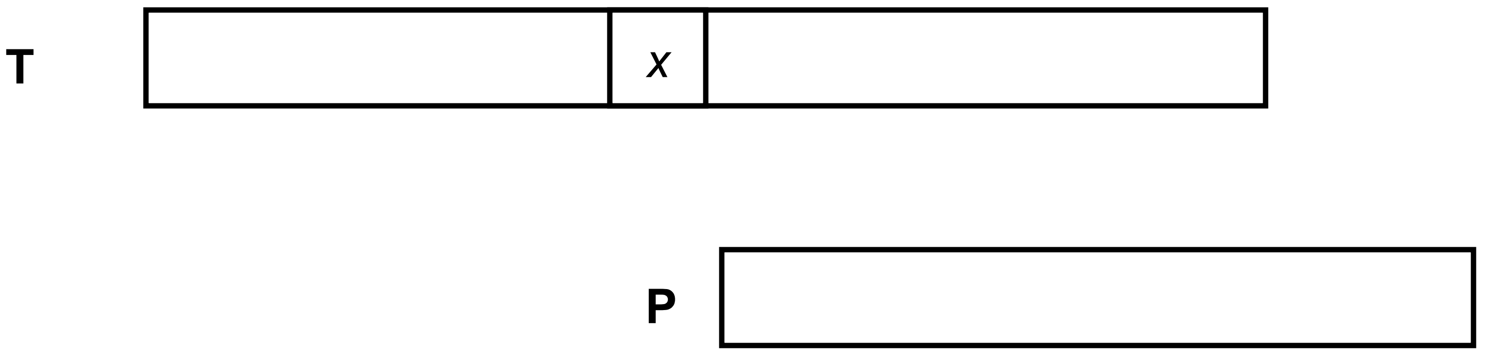
- Bad character rule uses Rule 2-1 (Character Matching Rule).
- For any character *x* in *T*, find the nearest *x* in *P* which is to the left of *x* in *T*.



Implication of Rule 2-1

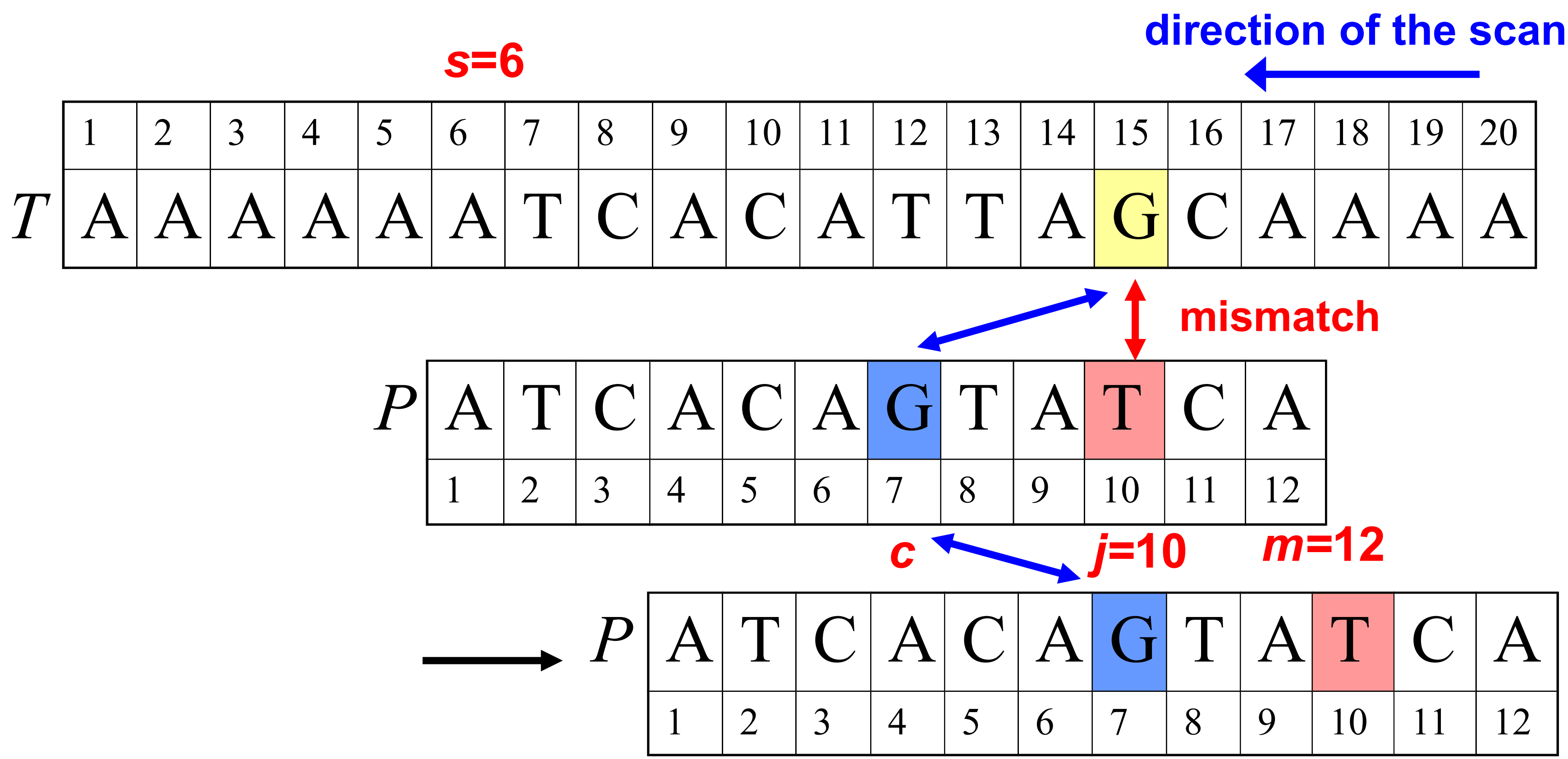
- Case 1. If there is a x in P to the left of T, move P so that the two x's match.

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



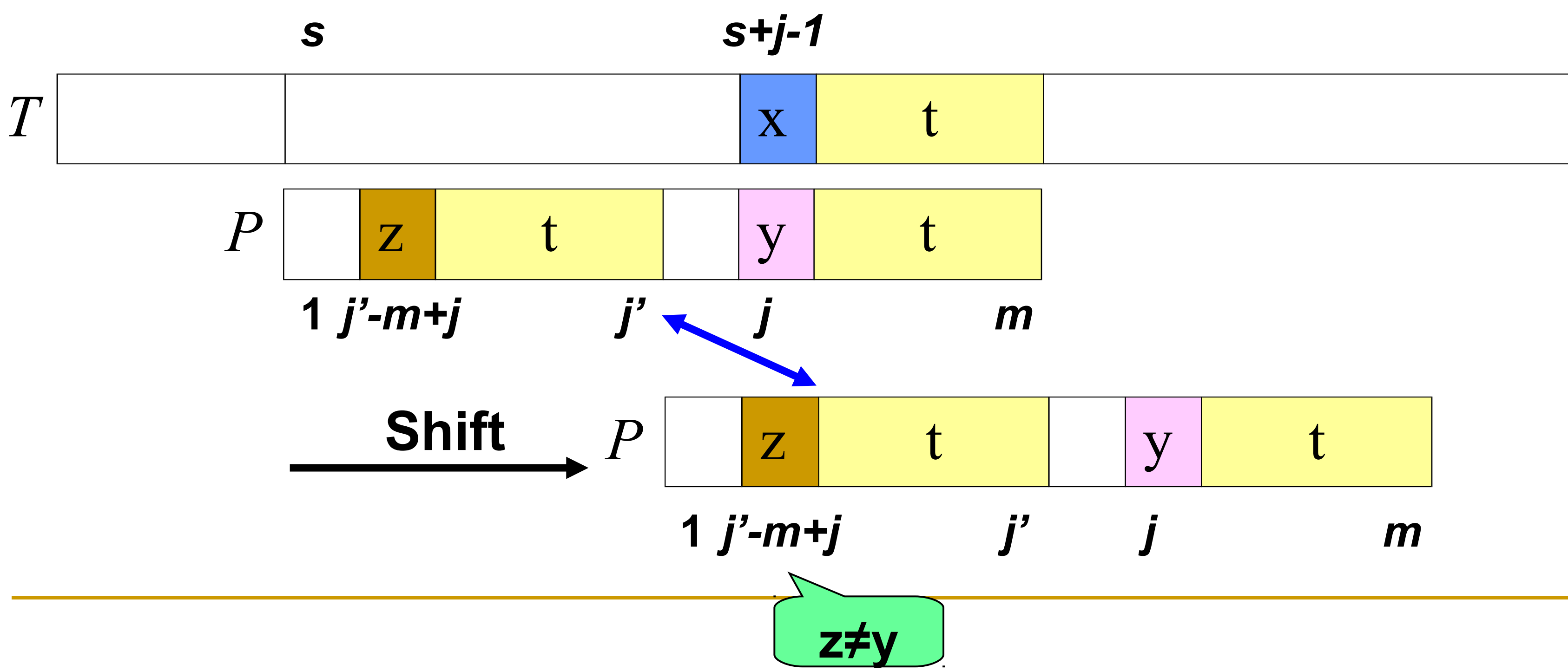
Ex: Suppose that P1 is aligned to T6 now. We compare pairwise between T and P from right to left. Since T16,17 = P11,12 = “CA” and T15 =“G” ≠P10 = “T”. Therefore, we find the rightmost position c=7 in the left of P10 in P such that Pc is equal to “G” and

we can move the window at least (10-7=3) positions.



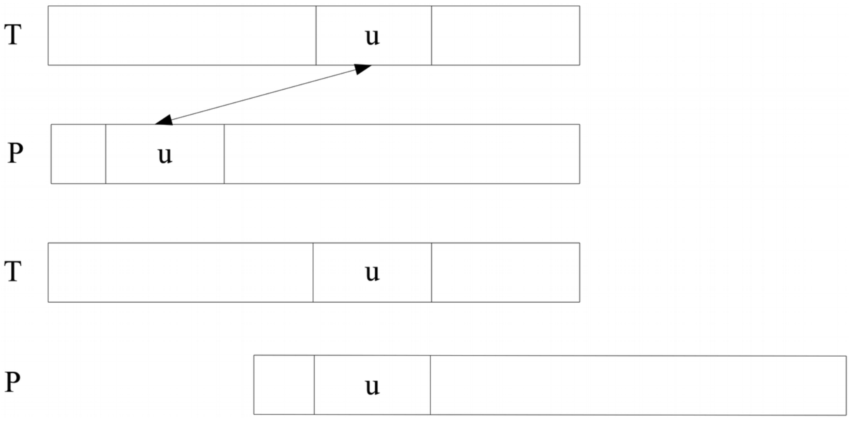
Good Suffix Rule 1

- If a mismatch occurs in T_{s+j-1}, we match T_{s+j-1} with $P_{j'-m+j}$, where j' ($m-j+1 \leqq j' < m$) 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).

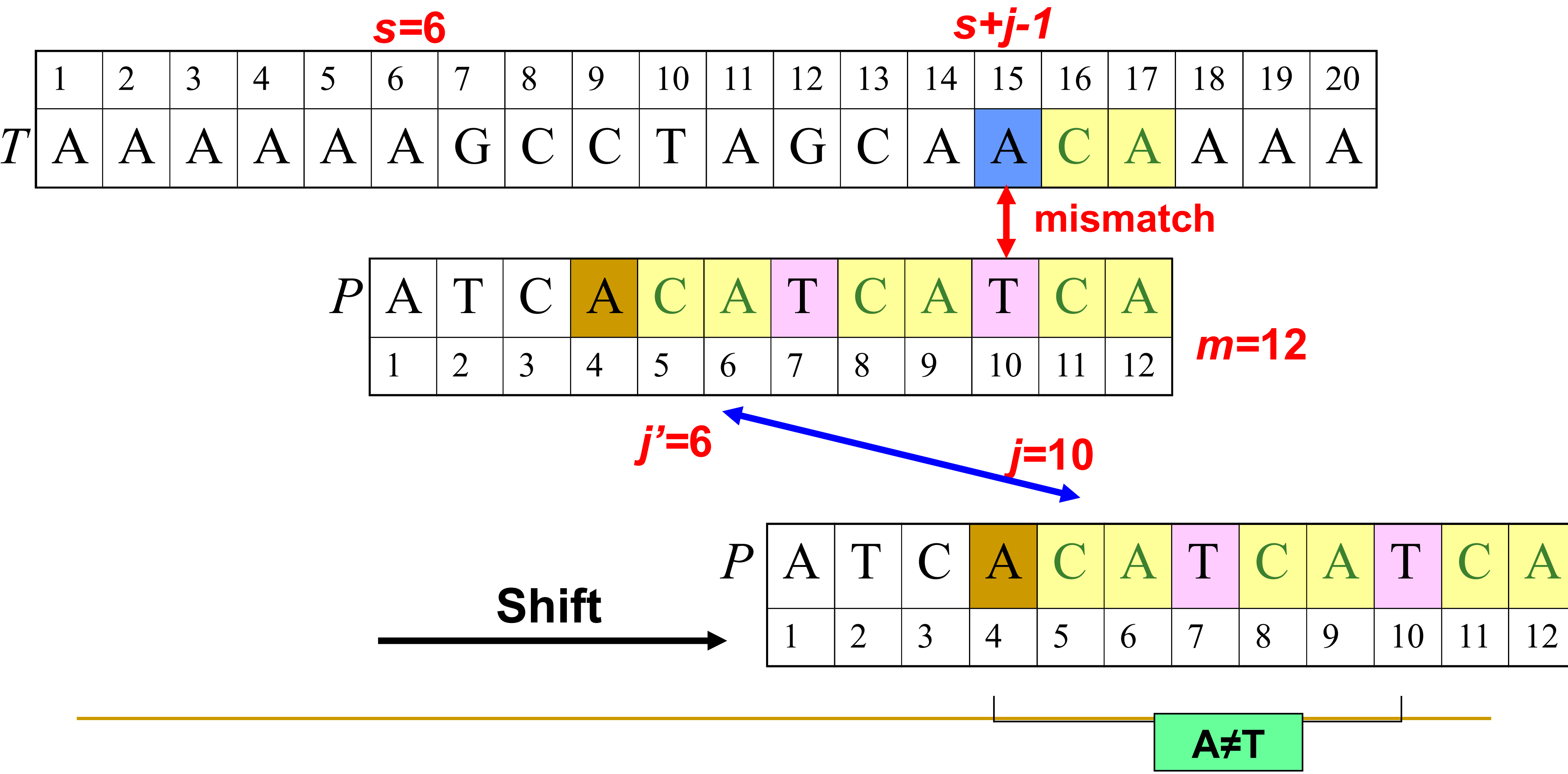


Rule 2: The Substring Matching Rule

- 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 P1 is aligned to T6 now. We compare pair-wise between P and T from right to left. Since T16,17 = “CA” = P11,12 and T15 =“A” ≠P10 = “T”. We find the substring “CA” in the left of P10 in P such that “CA” is the suffix of P1,6 and the left character to this substring “CA” in P is not equal to P10 = “T”. Therefore, we can move the window at least m-j’ (12-6=6) positions right.

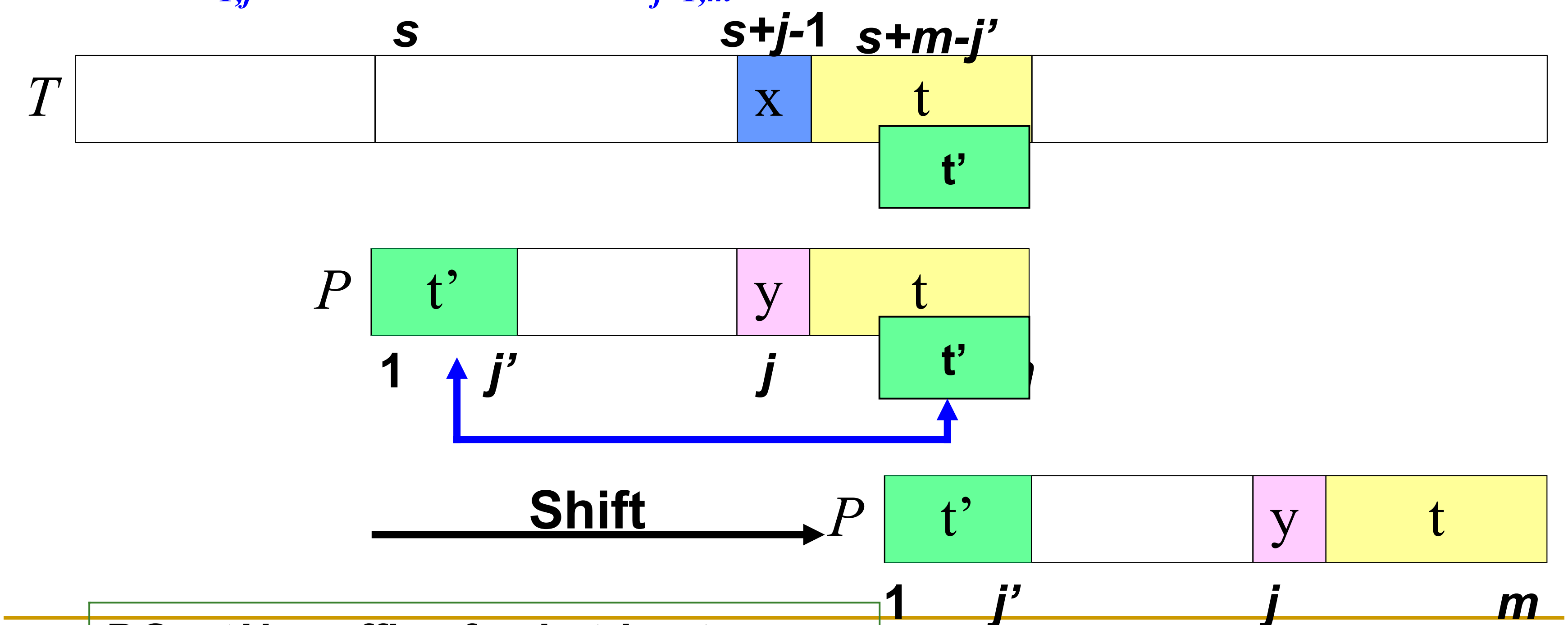


Good Suffix Rule 2

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' $(1 \leqq j' \leqq m-j)$ is **the largest position** such that

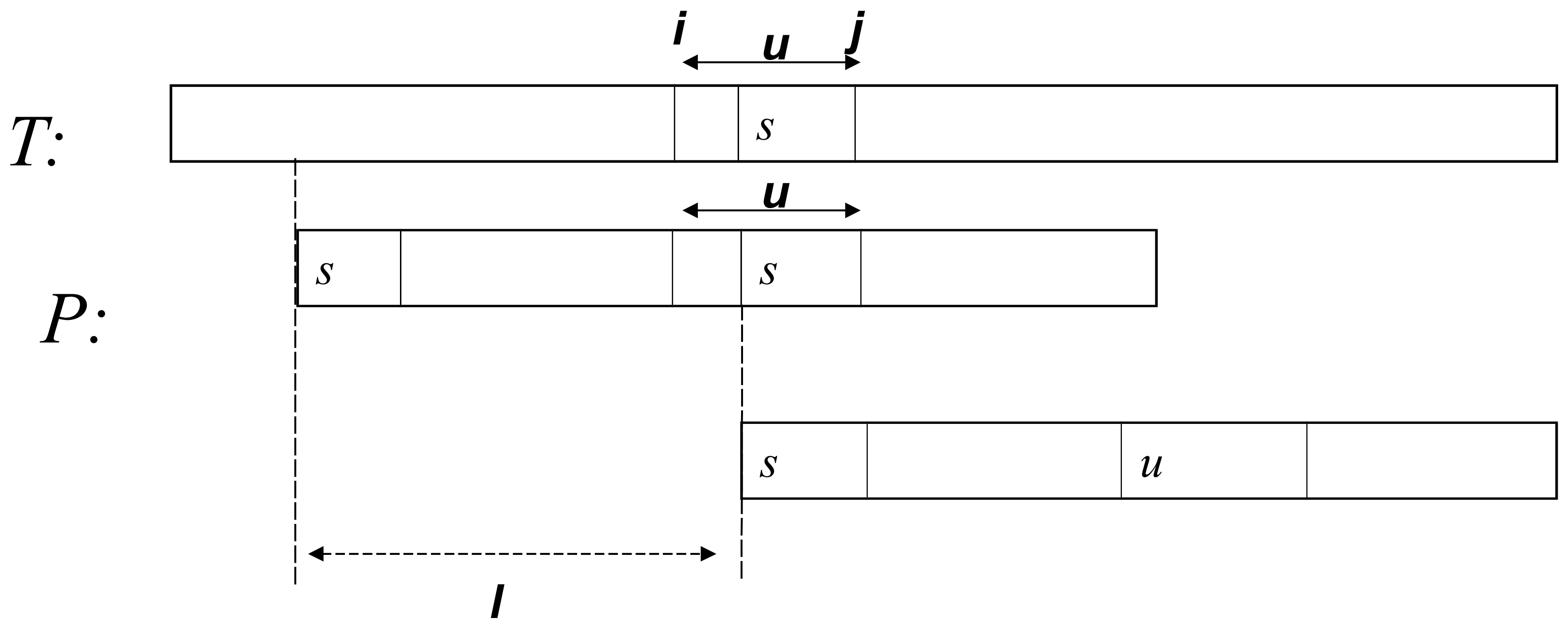
  $P_{1,j'}$ is a suffix of $P_{j+1,m}$.



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

Rule 3-1: Unique Substring Rule

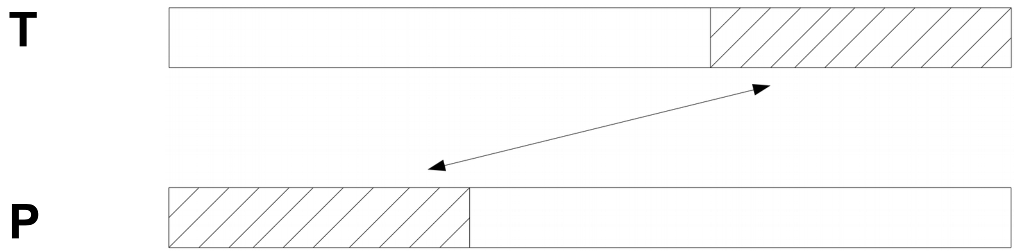
- The substring u appears in P exactly once.
- If the substring s matches with $T_{i,j}$, no matter whether a mismatch occurs in some position of P or not, we can slide the window by l.



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

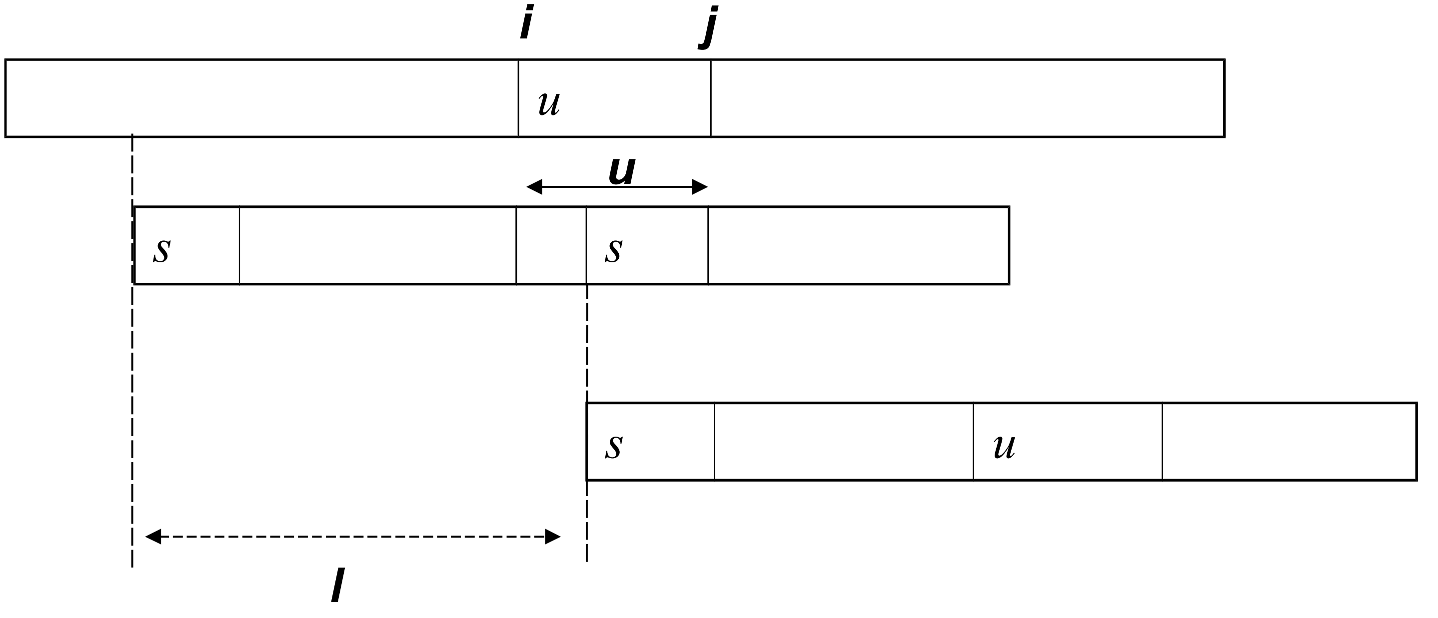
Rule 1: The Suffix to Prefix Rule

- 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.

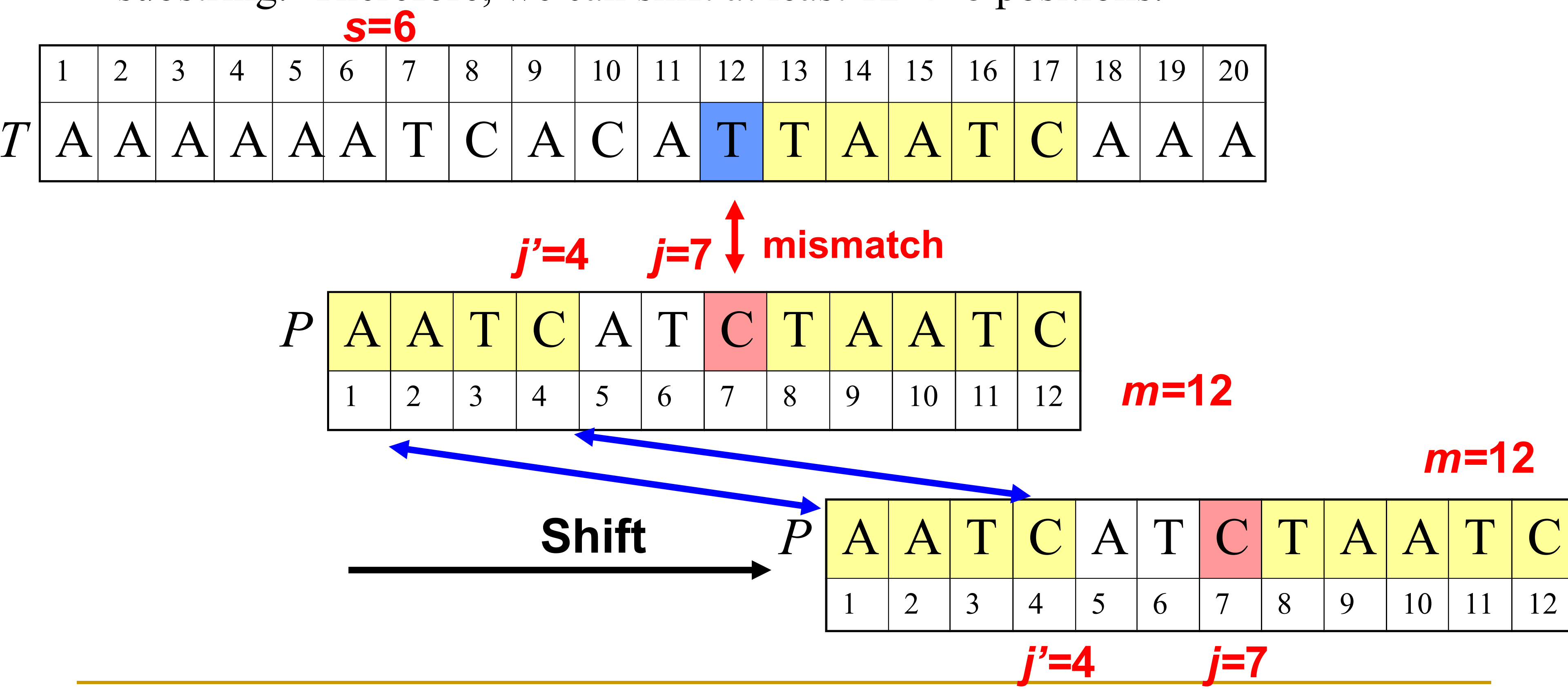


Rule 1: The Suffix to Prefix Rule

- 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 pair-wise 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
P	A	T	C	A	C	A	T	C	A	T	C	A

Σ	A	C	G	T
B[12]	12	11	0	10
…				
B[10]	9	8	0	10
…				

- We can move our pattern right at least $\boldsymbol{j\text{-}B[j](T_{s+j-1})}$ 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 *Gs*(*j*) be the largest number of shifts by *good suffix rule* when a mismatch occurs for comparing P_j with some character in *T*.

- $\mathbf{gs_1(j)}$ 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 $m-j+1 \leqq k<m$; 0 if there is no such k.

(*gs_1* is for Good Suffix Rule 1)

- $\mathbf{gs_2(j)}$ be the largest k such that $P_{1,k}$ **is a suffix of** $P_{j+1,m}$, where $1 \leqq k \leqq m-j$; 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

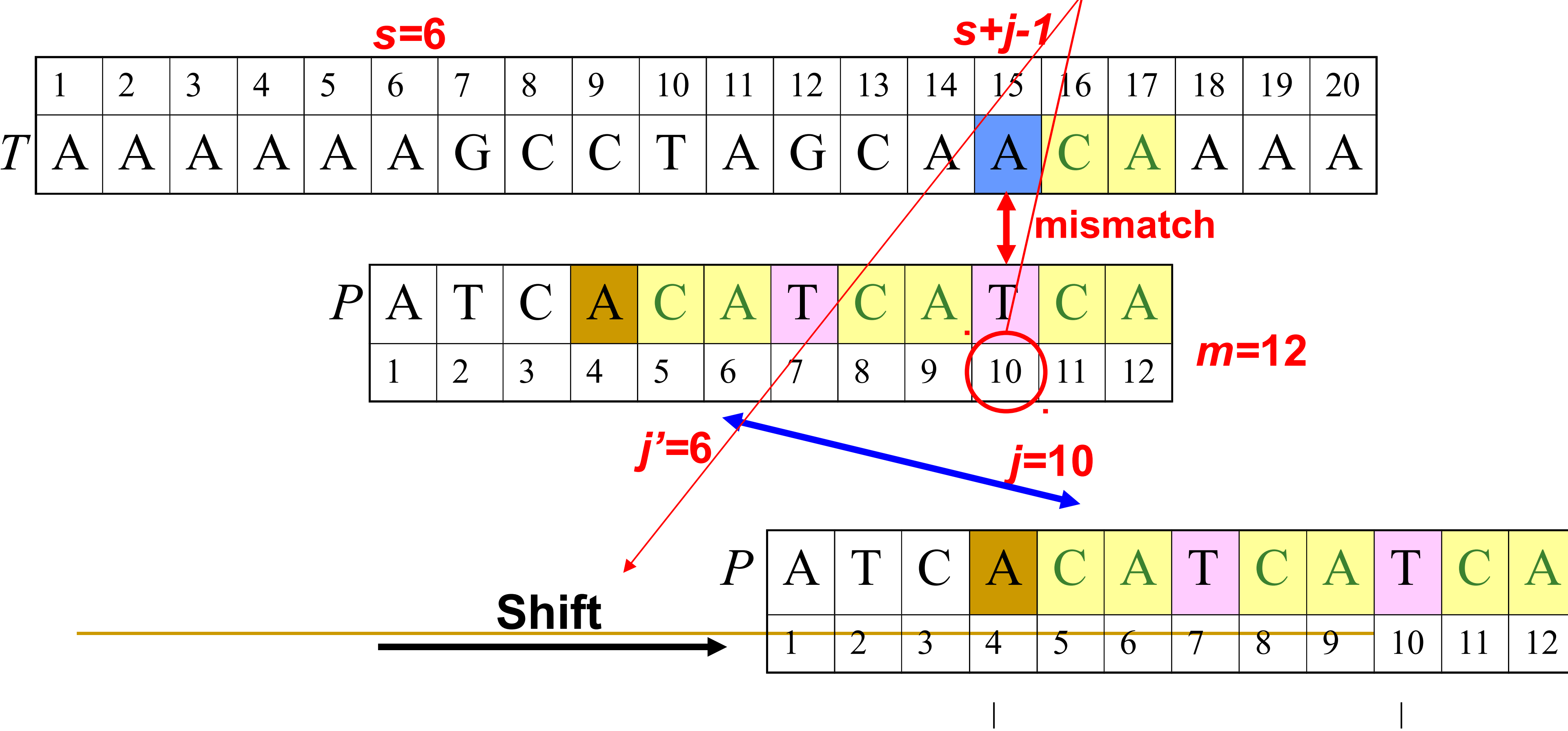
$gs_1(7)=9$

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

$gs_2(7)=4$

∵ $P_{1,4}$ 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
Gs	8	8	8	8	8	8	3	8	11	6	11	1



Time Complexity

- The preprocessing phase in O(m+Σ) complexity
- If you are searching for ALL matches, worst case:
  - O(mn) when P is in T
    - T=AAAAAAAAAAAA; P=AAAA
  - O(m+n) when P is not in T