String Matching II

Algorithm : Design & Analysis

[19]

In the last class…

- Simple String Matching
- KMP Flowchart Construction
- Jump at Fail
- KMP Scan

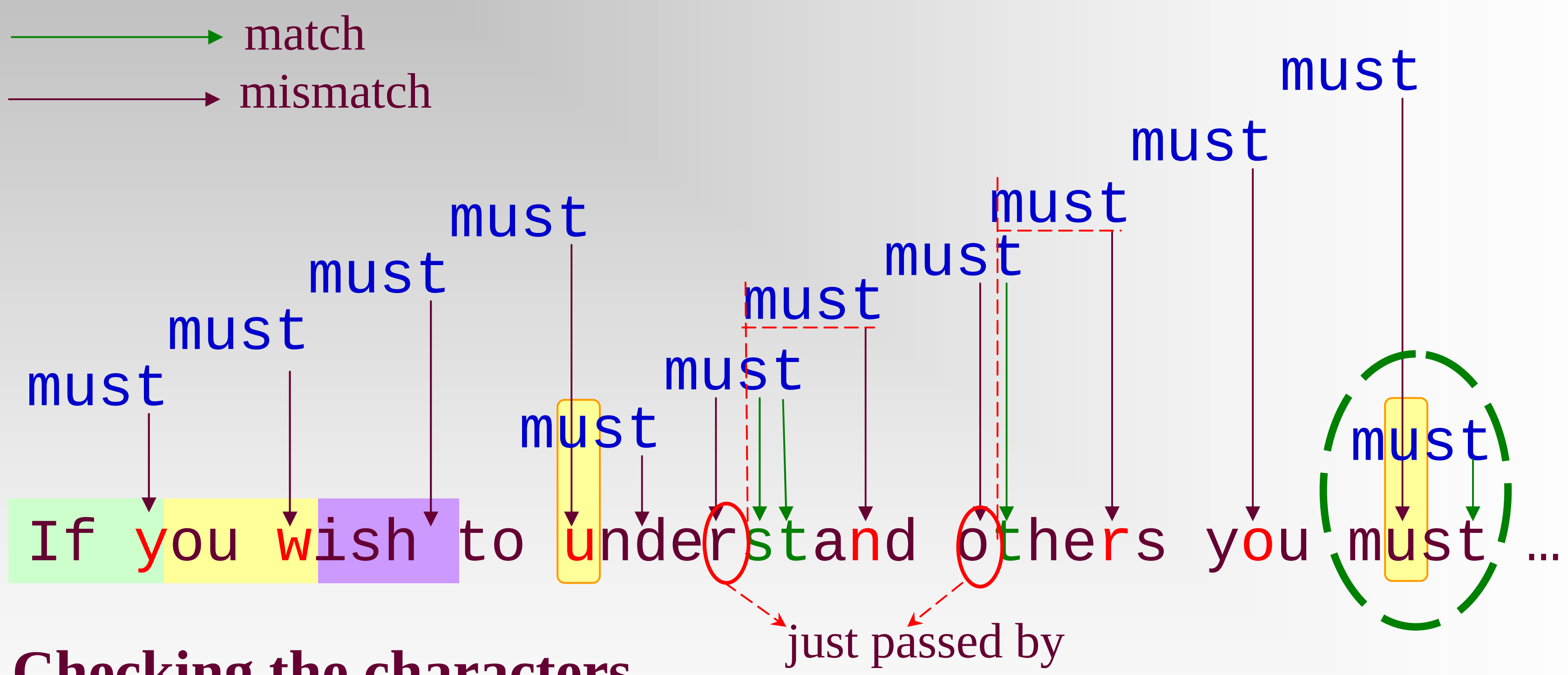
String Matching II

- Boyer-Moore’s heuristics
  - Skipping unnecessary comparison
  - Combining fail match knowledge into jump
- Horspool Algorithm
- Boyer-Moore Algorithm

Skipping over Characters in Text

- Longer pattern contains more information about **impossible** positions in the text.
  - For example: if we know that the pattern doesn’t contain a specific character.
- It doesn’t make the best use of the information by examining characters one by one forward in the text.

An Example

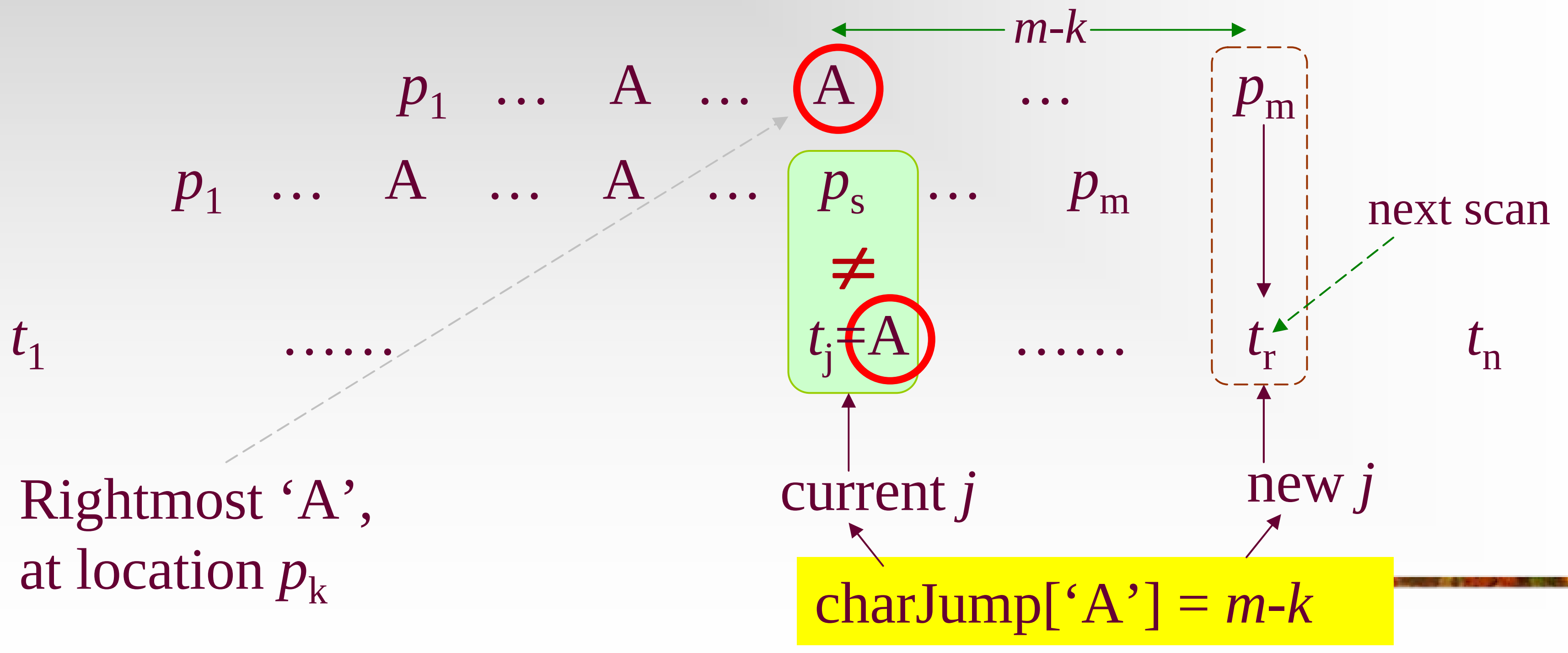


Checking the characters in *P*, in reverse order

The copy of the *P* begins at t_{38}.
Matching is achieved in 18 comparisons

Distance of Jumping Forward

- With the knowledge of *P*, the distance of jumping forward for the pointer of *T* is determined by the character itself, independent of the location in *T*.



m-*k*

p_1 … A … A … p_m

p_1 … A … A … p_s … p_m

≠

next scan

t_1 …… t_j=A …… t_r t_n

Rightmost ‘A’,
at location p_k

current *j*

new *j*

charJump[‘A’] = *m*-*k*

Computing the Jump: Algorithm

Input: Pattern string *P*; *m*, the length of *P*; alphabet size *alpha*=|Σ|

Output: Array *charJump*, indexed 0,…, *alpha*-1, storing the jumping offsets for each char in alphabet.

```
void computeJumps(char[ ] P, int m, int alpha, int[ ] charJump
    char ch;
    int k;
    for (ch=0; ch<alpha; ch++)
        charJump[ch]=m; //For all char no in P, jump by m
    for (k=1; k≤m; k++)
        charJump[$p_k$]=m-k;
```

$\Theta(|\Sigma|+m)$

The increasing order of k ensure that for duplicating symbols in *P*, the jump is computed according to the rightmost

Scan by CharJump: Horspool’s Algorithm

```
int horspoolScan(char[] P, char[] T, int m, int[] charjump)
    int j=m-1, k, match=-1;
    while (endText(T,j) = = false) //up to n loops
        k=0;
        while (k<m and P[m-k-1] = = T[j-k])//up to m loops
            k++;
        if (k= = m) match=j-m; break;
        else j=j+charjump[T[j]];
    return match;
```

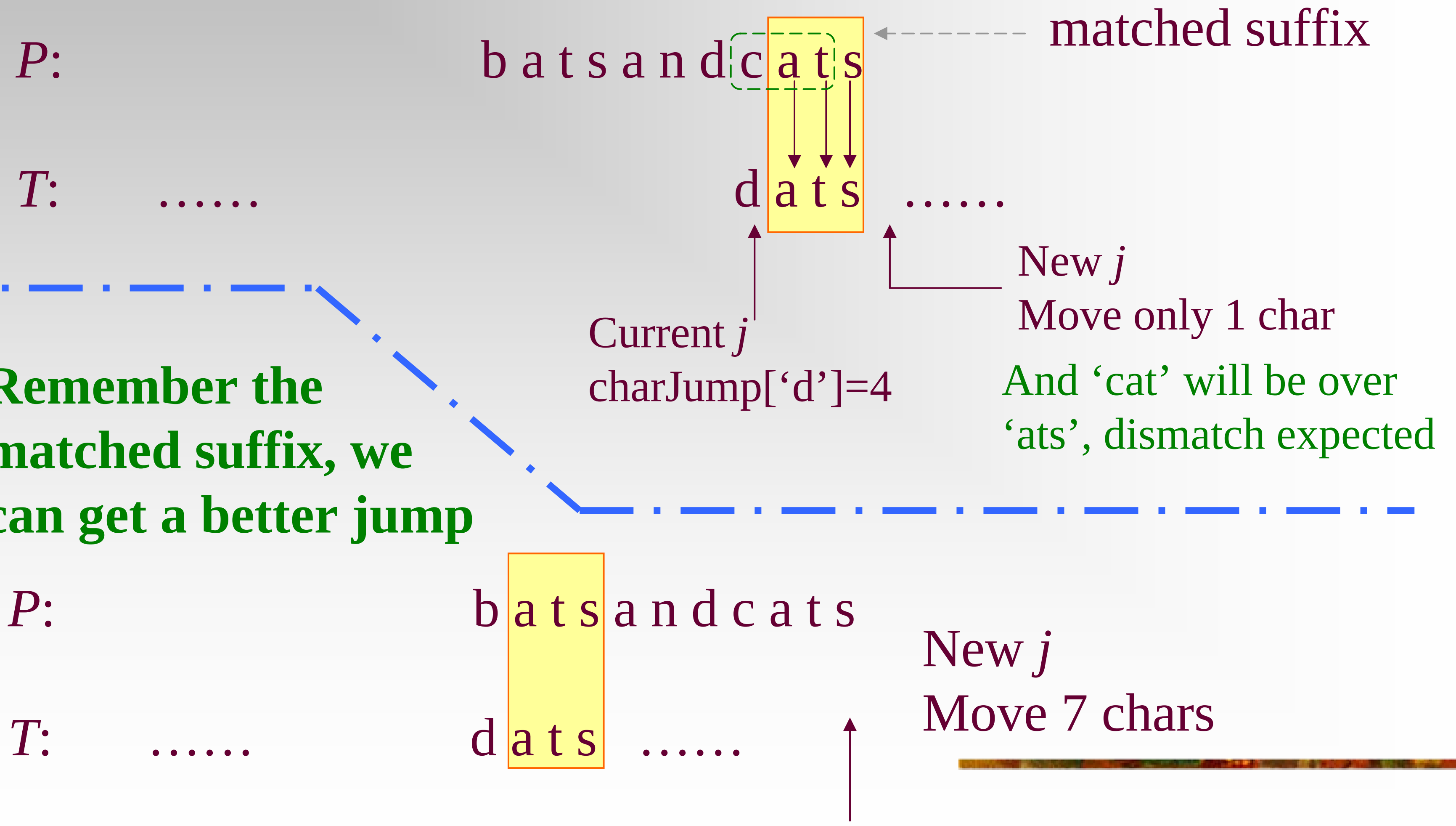
An example:

Search ‘aaaa……aa’ for ‘baaaa’

Note: charjump[‘a’]=1

So, in the worst case: $\Theta(mn)$

Partially Matched Substring



P: batsandcats

matched suffix

T: …… dats ……

Current *j*

charJump[‘d’]=4

New *j*

Move only 1 char

And ‘cat’ will be over ‘ats’, dismatch expected

Remember the matched suffix, we can get a better jump

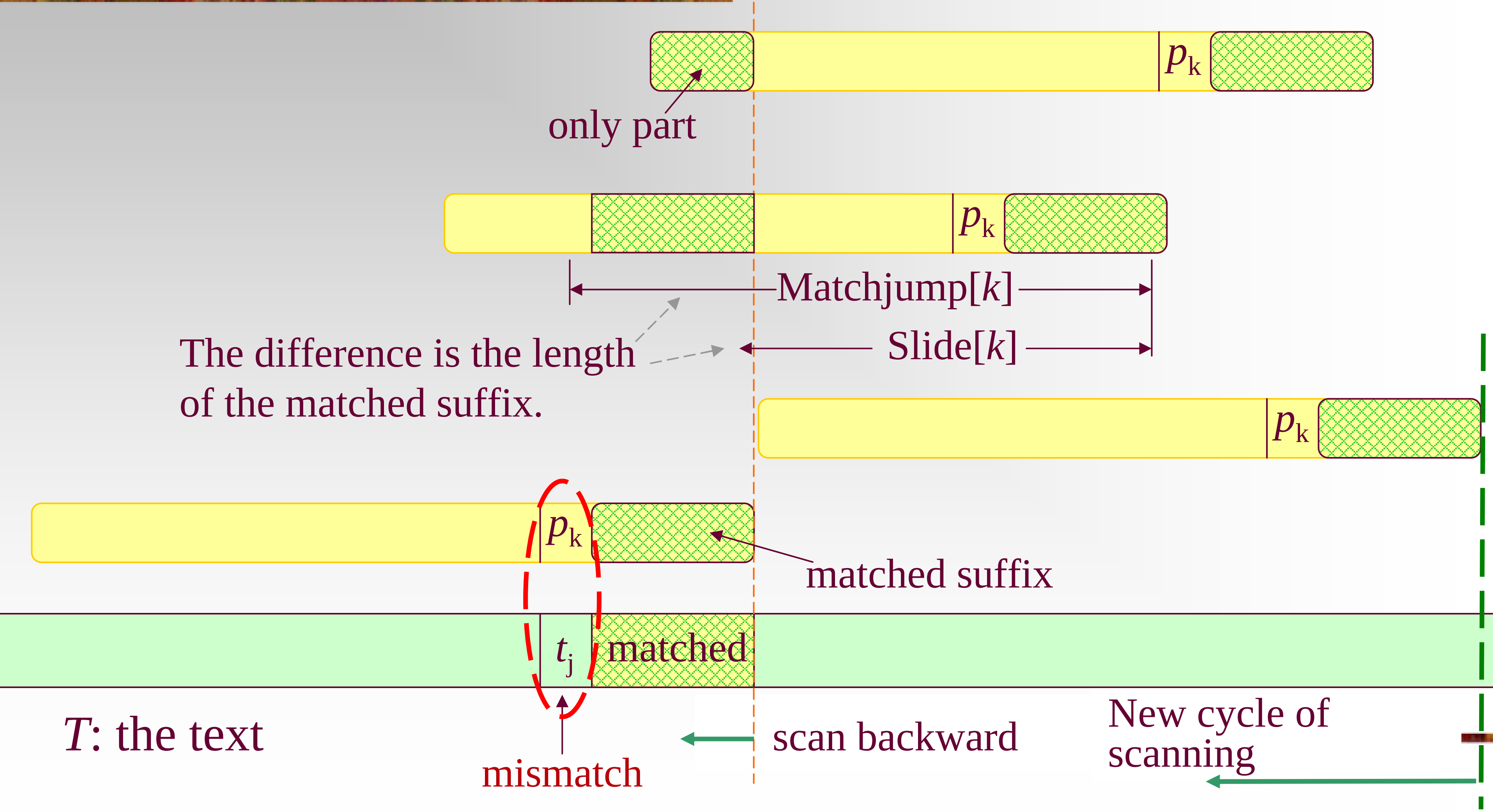
P: batsandcats

T: …… dats ……

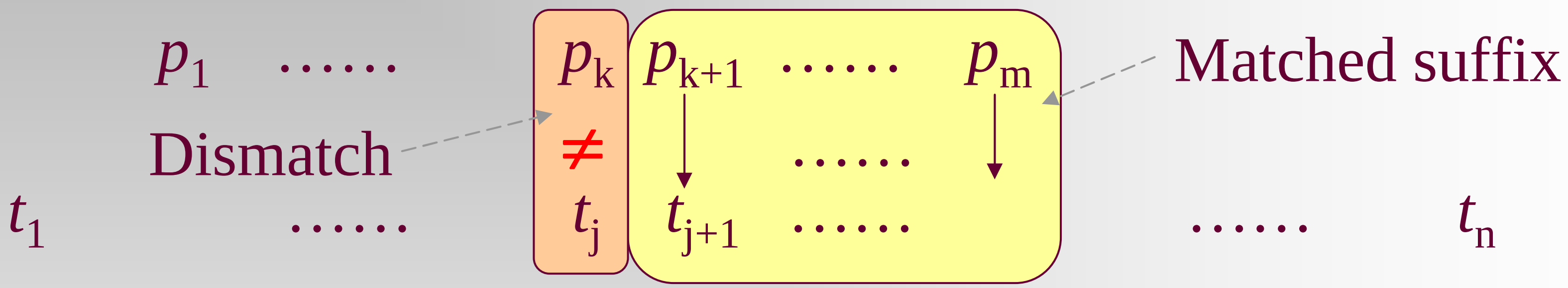
New *j*

Move 7 chars

Basic Idea


p_k
only part
p_k
Matchjump[k]
The difference is the length of the matched suffix.
Slide[k]
p_k
p_k
matched suffix
t_j
matched
T: the text
mismatch
scan backward
New cycle of scanning

Forward to Match the Suffix

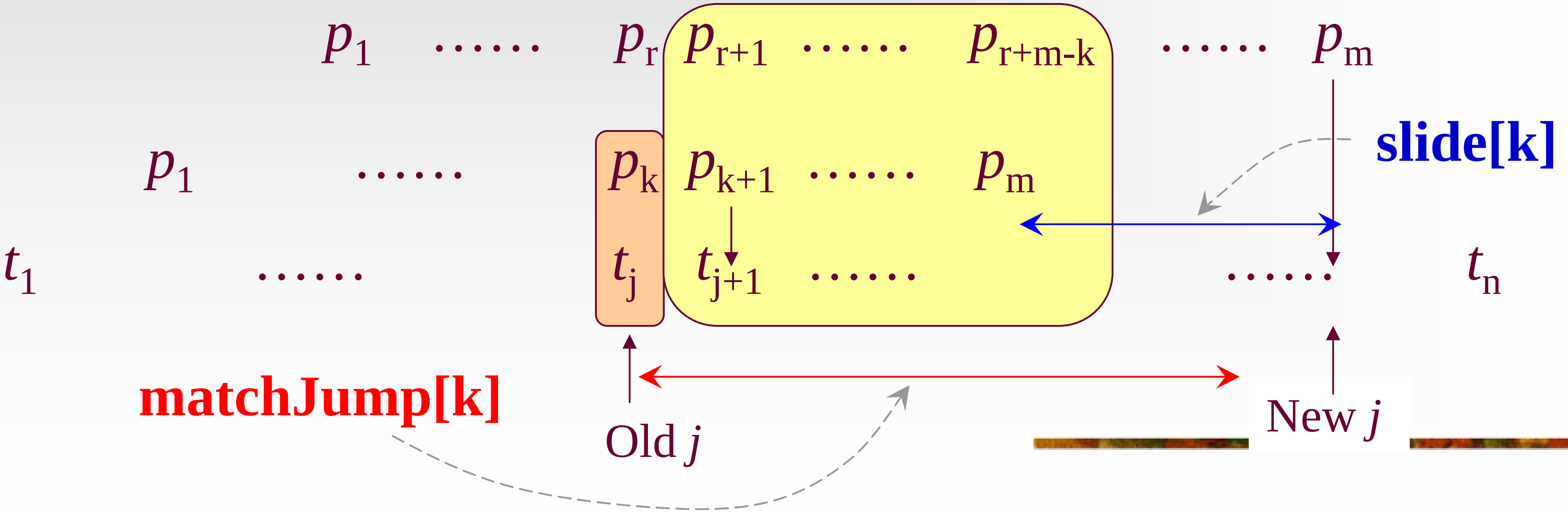


p_1 …… p_k p_{k+1} …… p_m Matched suffix

Dismatch $\neq$ ……

t_1 …… t_j t_{j+1} …… …… t_n

Substring same as the matched suffix occurs in *P*



p_1 …… p_r p_{r+1} …… p_{r+m-k} …… p_m

slide[k]

p_1 …… p_k p_{k+1} …… p_m

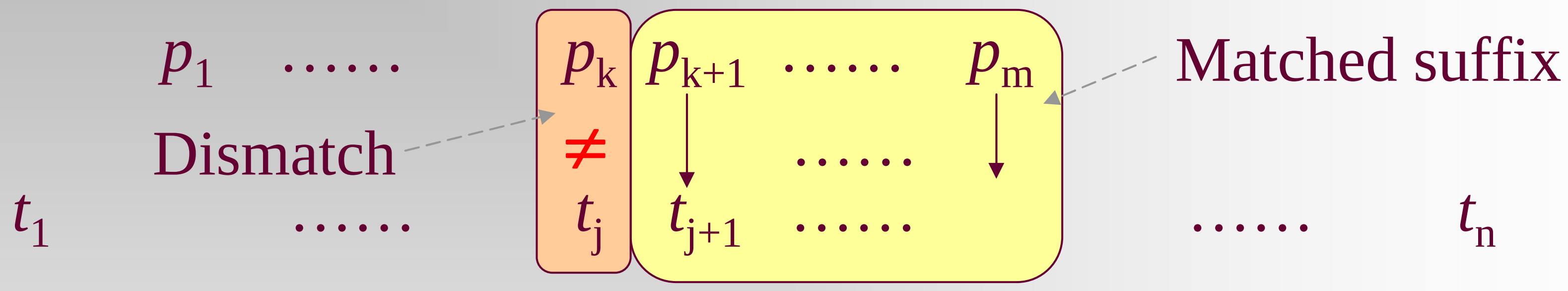
t_1 …… t_j t_{j+1} …… …… t_n

matchJump[k]

Old *j*

New *j*

Partial Match for the Suffix



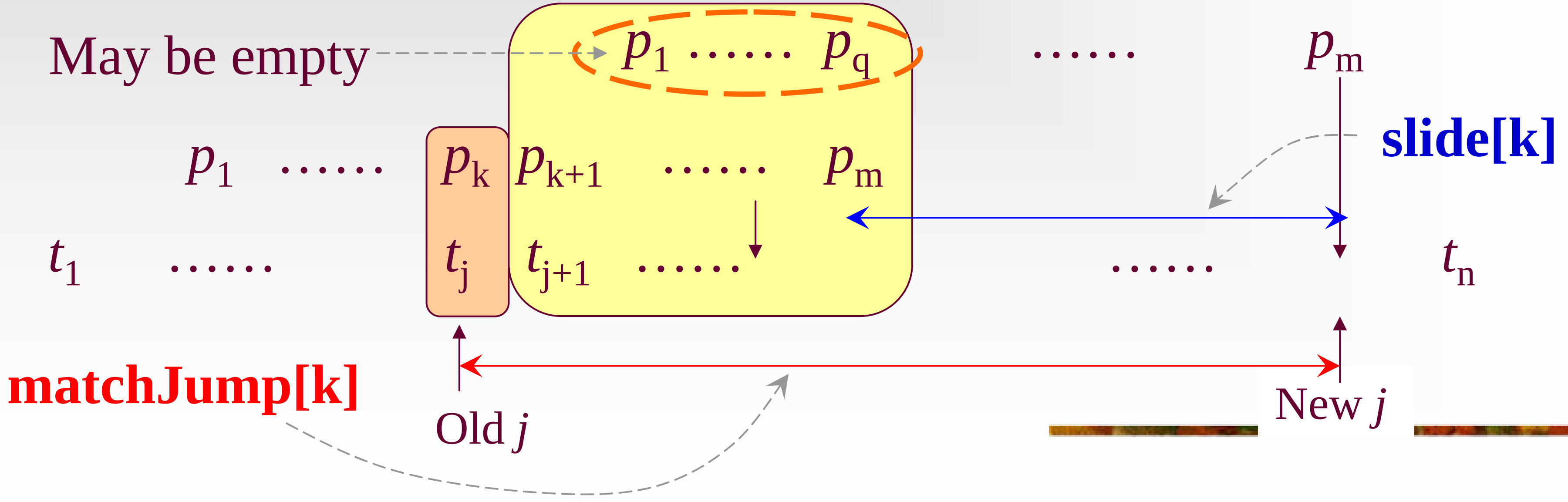
p_1 …… p_k p_{k+1} …… p_m

Matched suffix

Dismatch $\neq$ ……

t_1 …… t_j t_{j+1} …… …… t_n

No entire substring same as the matched suffix occurs in *P*



May be empty

p_1 …… p_q …… p_m

slide[k]

p_1 …… p_k p_{k+1} …… p_m

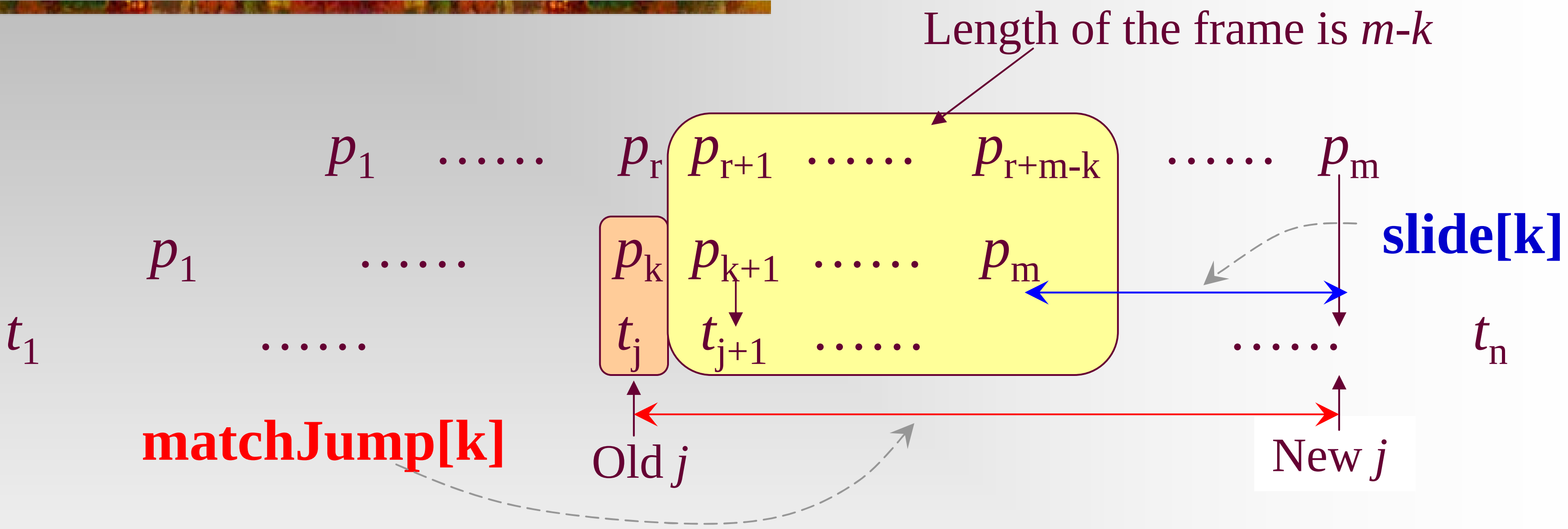
t_1 …… t_j t_{j+1} …… …… t_n

matchJump[k]

Old j

New j

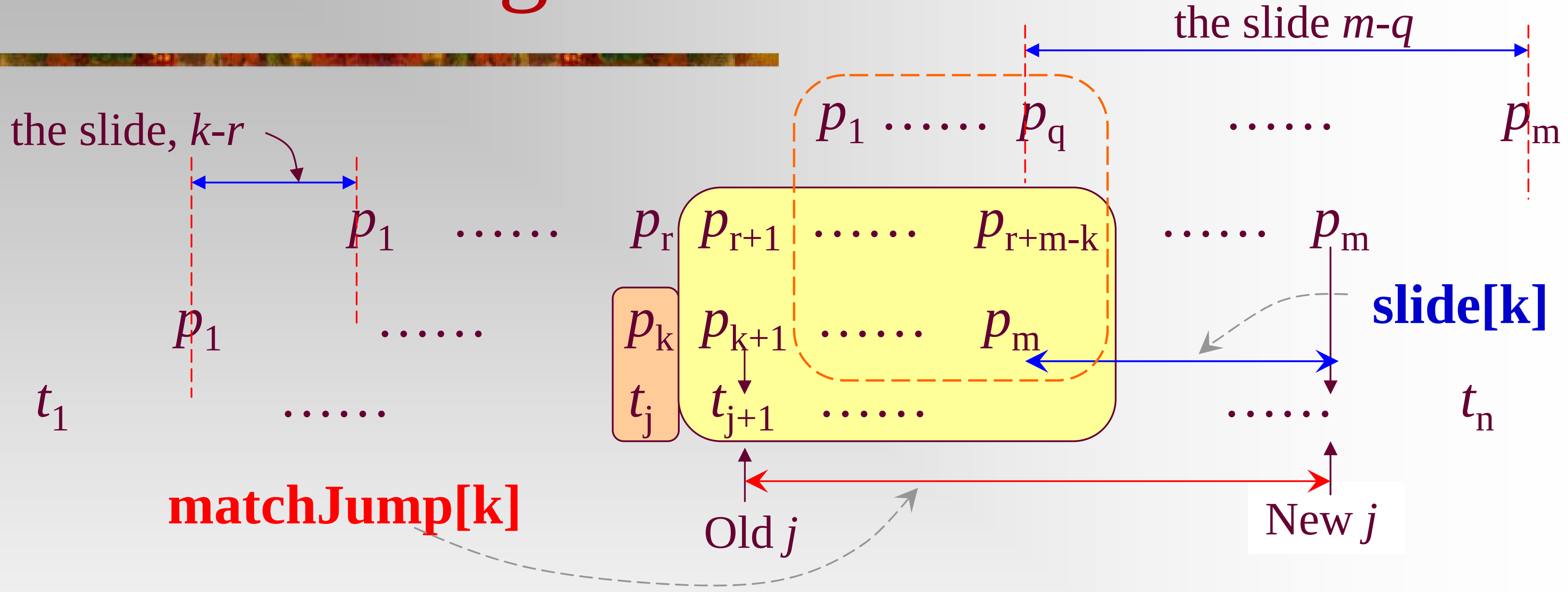
matchjump and *slide*


Length of the frame is *m*-*k*
p_1 …… p_r p_{r+1} …… p_{r+m-k} …… p_m
slide[k]
p_1 …… p_k p_{k+1} …… p_m
t_1 …… t_j t_{j+1} …… …… t_n
matchJump[k]
Old *j*
New *j*

- **slide[k]**: the distance *P* slides forward after dismatch at p_k, with *m*-*k* chars matched to the right
- **matchjump[k]**: the distance *j*, the pointer of *P*, jumps, that is:

matchjump[k]=slide[k]+m-k

Determining the *slide*



the slide *m-q*
p_1 …… p_q …… p_m
the slide, *k-r*
p_1 …… p_r p_{r+1} …… p_{r+m-k} …… p_m
slide[k]
p_1 …… p_k p_{k+1} …… p_m
t_1 …… t_j t_{j+1} …… …… t_n
matchJump[k]
Old *j*
New *j*

•Let r(r<k) be the largest index, such that p_{r+1} starts a largest substring matching the matched suffix of P, and $p_r \neq p_k$, then slide[k]=k-r

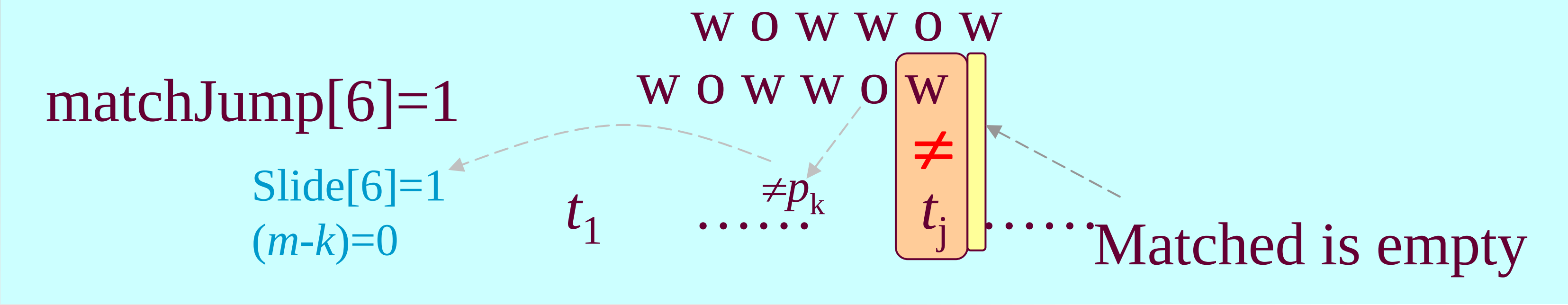
• If the r not found, the longest prefix of P, of length q, matching the matched suffix of P will be lined up. Then slide[k]=m-q.

$p_r = p_k$ is senseless since p_k is a mismatch

Computing *matchJump*: Example

P = “ w o w w o w ”

← Direction of computing



w o w w o w

w o w w o w

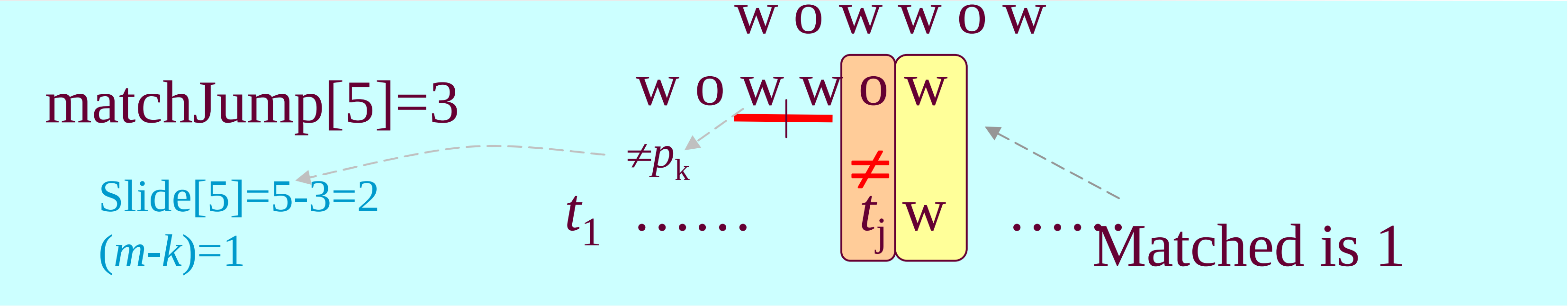
matchJump[6]=1

Slide[6]=1

(m-k)=0

t_1 …… $\neq p_k$ ≠ t_j ……

Matched is empty



w o w w o w

w o w w o w

matchJump[5]=3

Slide[5]=5-3=2

(m-k)=1

$\neq p_k$

t_1 …… ≠ t_j w ……

Matched is 1

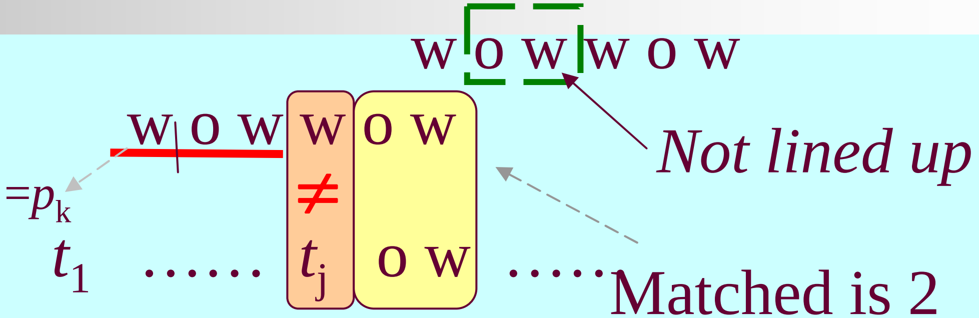
Computing *matchJump*: Example

P = " w o w w o w "

Direction of computing

matchJump[4]=7

No found, but
a prefix of length 1,
so, Slide[4] = m-1=5



w o w w o w

w o w w o w

$=p_k$

≠

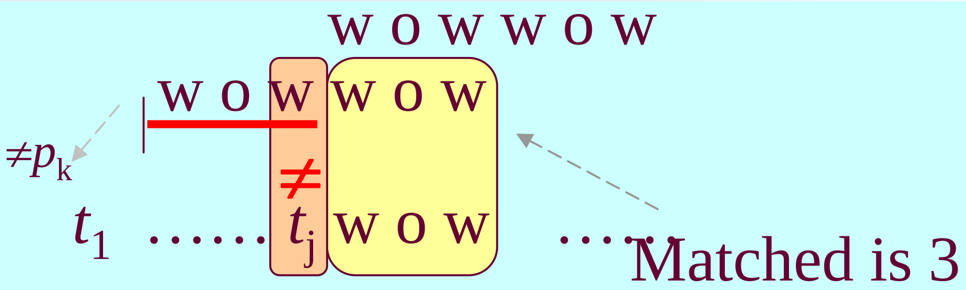
t_1 …… t_j o w ……

Not lined up

Matched is 2

matchJump[3]=6

Slide[3]=3-0=3
(m-k)=3



w o w w o w

w o w w o w

$\neq p_k$

≠

t_1 …… t_j w o w ……

Matched is 3

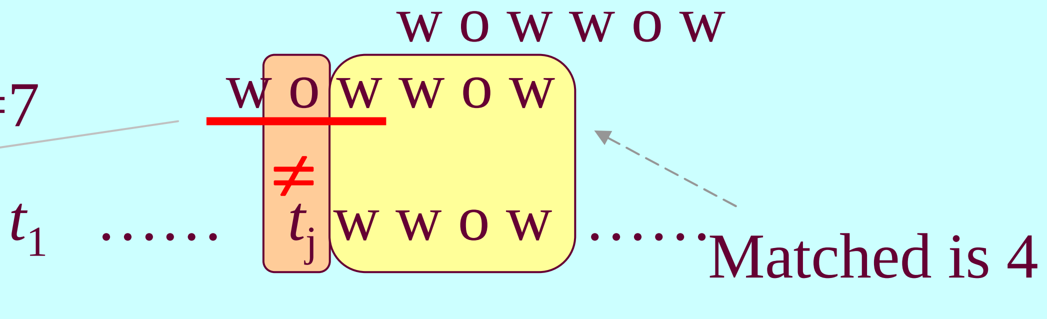
Computing *matchJump*: Example

P = " w o w w o w "

Direction of computing

matchJump[2]=7

No found, but
a prefix of length 3,
so, Slide[2] = *m*-3=3



w o w w o w

~~w o w~~ w o w

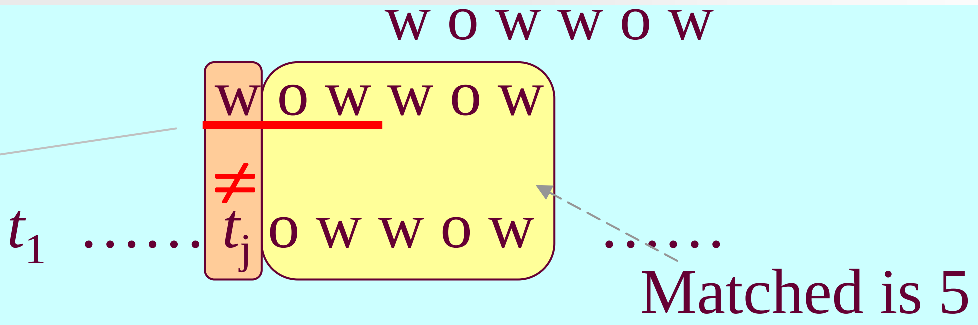
≠

t_1 ...... t_j w w o w ......

Matched is 4

matchJump[1]=8

No found, but
a prefix of length 3,
so, Slide[1] = *m*-3=3



w o w w o w

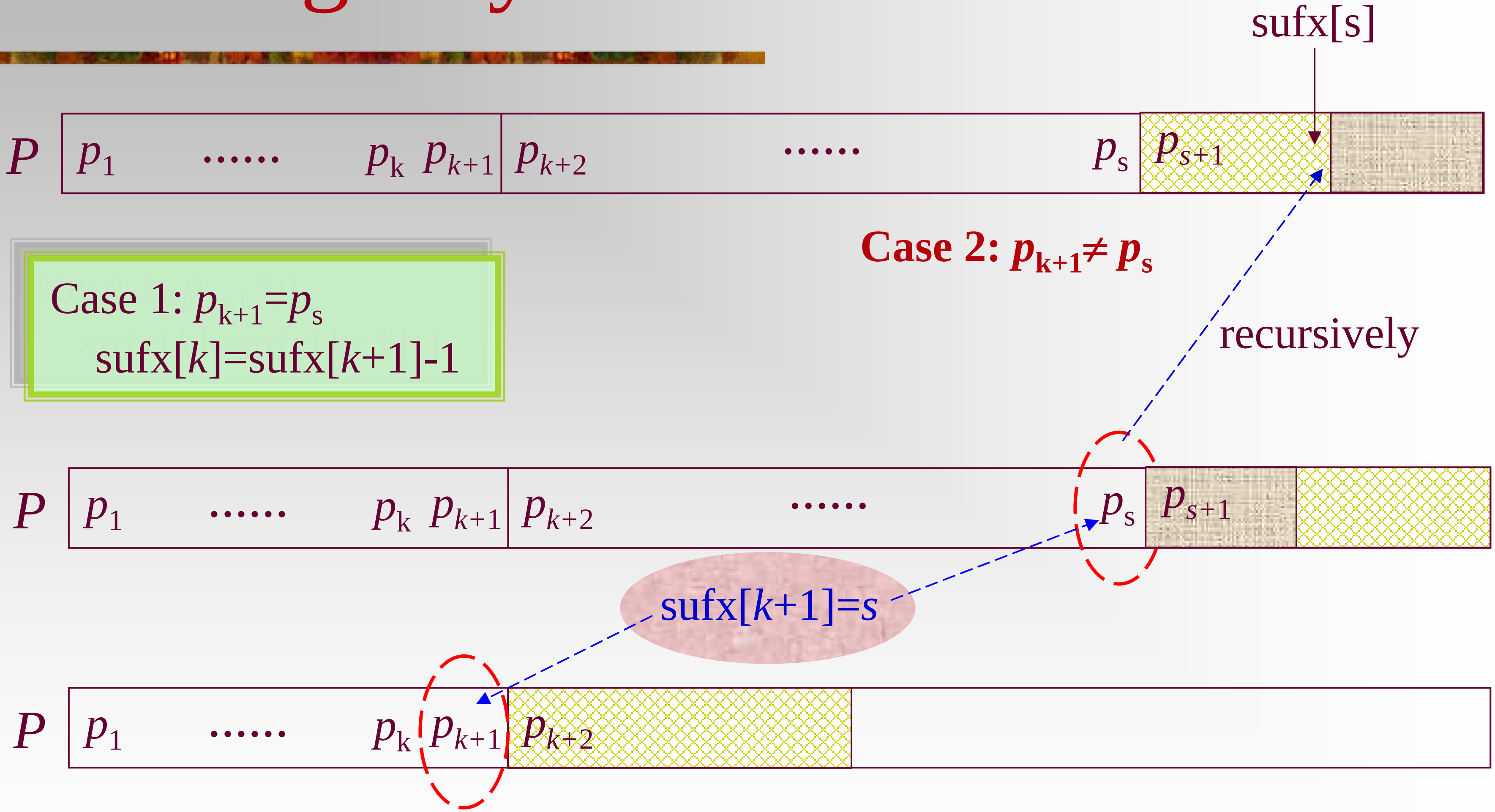
~~w o w~~ w o w

≠

t_1 ...... t_j o w w o w ......

Matched is 5

Finding *r* by Recursion



sufx[s]

P p_1 …… p_k p_{k+1} p_{k+2} …… p_s p_{s+1}

Case 1: $p_{k+1}=p_s$
sufx[k]=sufx[k+1]-1

Case 2: $p_{k+1} \neq p_s$

recursively

P p_1 …… p_k p_{k+1} p_{k+2} …… p_s p_{s+1}

sufx[k+1]=s

P p_1 …… p_k p_{k+1} p_{k+2}

Computing the slides: the Algorithm

```
for (k=1; k≤m; k++) matchjump[k]=m+1;
    sufx[m]=m+1;
    for (k=m-1; k≥0; k--)
        s=sufix[k+1]
        while (s≤m)
            if ($p_{k+1}$= = $p_s$) break;
            matchjump[s] = min (matchjump[s], s-(k+1));
            s = sufx[s];
        sufx[k]=s-1;
```

initialized as impossible values

Remember:
slide[*k*]=*k*-*r*
here: *k* is *s*,
and *r* is
k+1

Computing the *matchjump*: Whole Procedure

```
void computeMatchjumps(char[] P, int m, int[] matchjump)
   int k,r,s,low,shift;
   int[] sufx = new int[m+1]

   <computing slides: as the precedure in the frame afore>

   low=1; shift=sufx[0];
   while (shift≤m)
      for (k=low; k≤shift; k++)
         matchjump[k] = min(matchjump[k], shift);
      low=shift+1; shift=sufx[shift];
```

computing slides for sufix matched shorter prefix

```
   for (k=1; k≤m; k++)
      matchjump[k]+=(m-k);
   return
```

turn into matchjump by adding *m-k*

Boyer-Moore Scan Algorithm

```
int boyerMooreScan(char[] P, char[] T, int[] charjump, int[] matchjump)
    int match, j, k;

    match=-1;
    j=m; k=m; // first comparison location
    while (endText(T,j) ==false)
        if (k<1)
            match = j+1 //success
            break;
        if ($t_j$ = = $p_k$ ) j--; k--;
        else
            j+=max(charjump[$t_j$], matchjump[k]);
            k=m;
    return match;
```

scan from right to left

take the better of the two heuristics

Home Assignment

- pp.508-
  - 11.16
  - 11.19
  - 11.20
  - 11.25