NANYANG TECHNOLOGICAL UNIVERSITY

# String Matching

Huang Shell Ying

**Reference:** Introduction to Algorithms. Cormen, T.H., C.E. Leiserson. R.L. Rivest, Chapter 34

Computer Algorithms. Sara Baase & Allen Van Gelder, Chapter 11

The problem: Given a text T of *n* characters and a pattern P of *m* characters, find the first occurrence of P in T.

We may be looking for

- A character string in text;
- A pattern in DNA sequences;
- A piece of coded information representing graphical, audio data, or machine code;
- A sublist in linked list……….

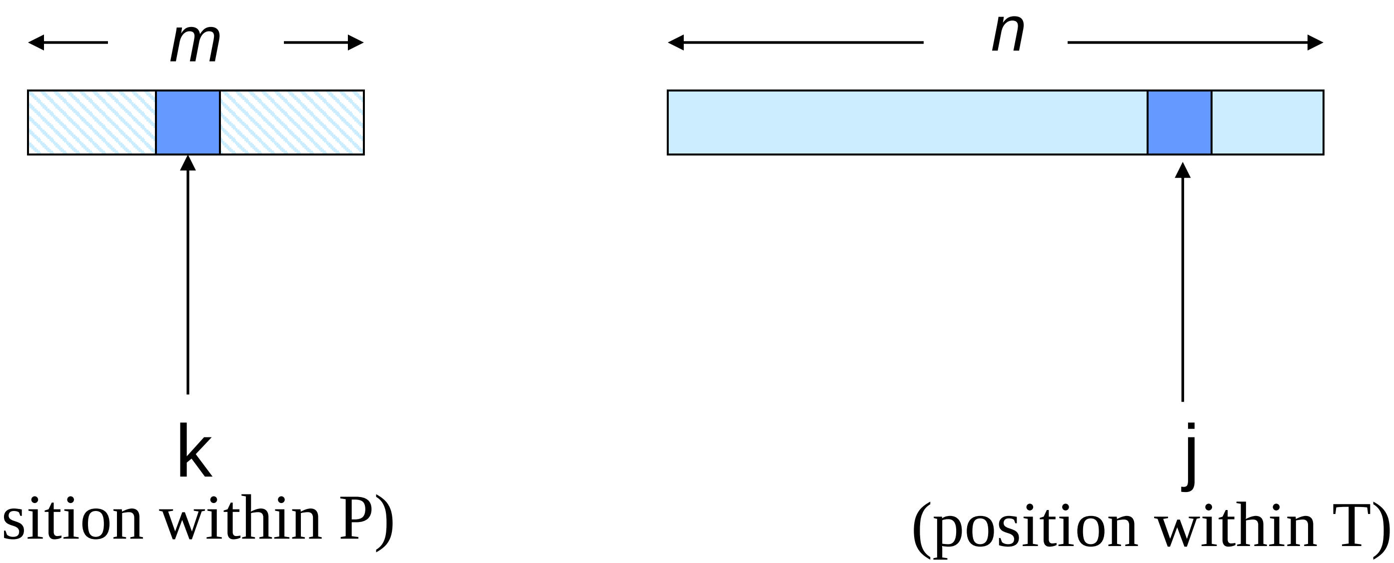
We will study ----

- A straightforward solution
- The Rabin-Karp Algorithm
- The Boyer-Moore Algorithm

Conventions used:

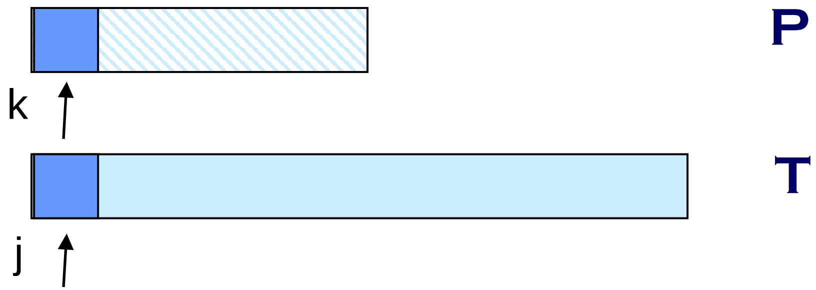
**P = PATTERN**

**T = TEXT**

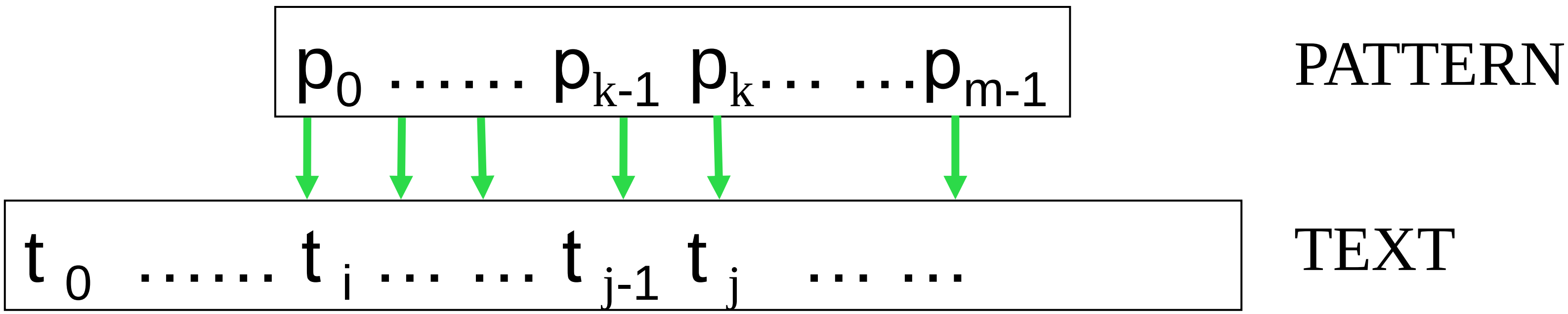


# A straightforward solution

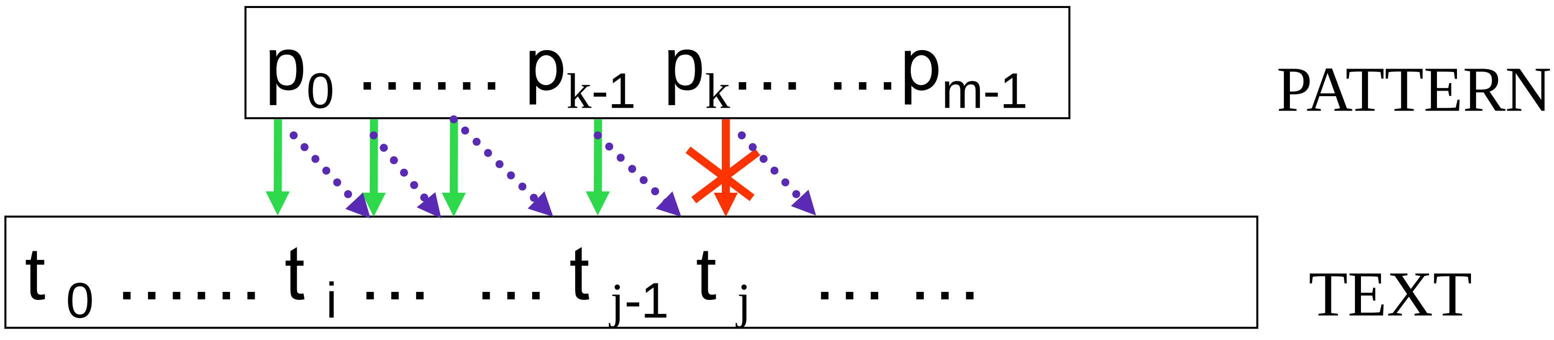
```
int SimpleScan (char [] P, char [] T, int m)
{
    int i, j, k;
   // i is the current guess of where P begins in T;
    // j is the index of the current character in T;
    // k is the index of the current character in P;
  j = k = 0;
  i = 0;
```



```
    while (j < n) {
        if (T[j] == P[k]) {
            j++;
            k++;
            if (k == m)   return i; }
        else {
            j = ++i;
            k = 0;
            if (j > n-m)  break;    }
    return -1;
}
```




- Comparison starts with k=0 and j=i. After a match between P[k] and T[j], j and k are incremented.
- When k reaches m, all characters have been compared and matched. Return i.



When a mismatch happens, shift the pattern right one position: j=++i, k=0.

$p_0$ …… … …$p_{m-1}$

$t_0$ …… $t_i$ $t_{i+1}$ … … … …

After shifting, if n-j<m, there is no enough text left for comparison, then stop comparisons with no match.

$p_0$ …… … …$p_{m-1}$

$t_0$ …… $t_i$ $t_j$ … . …. $t_{n-1}$

# Example

P = ABABC

T = ABABABCCAC

ABABC

ABABABCCAC

ABABC

**Match Successful!**

ABABABCCAC

# Worst case

P = AAAAC

T = AAAAAAAAAAAAA

From the 1st character to the 5th last character of the text T, 5 comparisons are done before a mismatch.

Total is m(n-m+1) comparisons

AAAAC

AAAAAAAAAAAAA

AAAAC

AAAAAAAAAAAAA

....

AAAAC

AAAAAAAAAAAAA

Worst case complexity is O(*mn*) where *m* is the length of the pattern and *n* is the length of the text

# The Rabin-Karp Algorithm

- Outline of the steps of Rabin-Karp Algorithm
  1) Convert the pattern (length *m*) to a number, p
  2) Convert the first *m*-characters (the first text window) to a number, t
  3) If p and t are equal, pattern found and exit
  4) If not end-of-text, shift the text window one character right and convert the string in it to a number t, go to step 3); else pattern not found and exit

**P**: | 0 | 0 | 1 | 0 | 1 |

**m = 5, p = 5**

p = 5

| T: | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

t = 6 12 25 18 4 8 16 1 2 5

**Found!**

To compute the number for the pattern and the number for the first *m*-character text window,

- The set of possible characters is referred to as an alphabet and denoted with sigma Σ. e.g.
  Σ = {0, 1} or Σ = {0, 1, 2, … , 9}
  or Σ = {a, b, c, … , z}
- Let d = |Σ|

- The number p of the pattern and the number t of the first *m*-character text window, are calculated iteratively.

For example, P = “36215”, d = 10

| 3 | 6 | 2 | 1 | 5 |
|---|---|---|---|---|

$p = 3 * 10^4 + 6 * 10^3 + 2 * 10^2 + 1 * 10^1 + 5$

$= (3 * 10^3 + 6 * 10^2 + 2 * 10^1 + 1) * 10 + 5$

$= ((3 * 10^2 + 6 * 10^1 + 2) * 10 + 1) * 10 + 5$

$= (((3 * 10 + 6)*10 + 2) * 10 + 1) * 10 + 5$

- p = P[0]*d^(m-1) + P[1]*d^(m-2) + … + P[m-2]*d + P[m-1]

  = (((P[0]*d + P[1])*d + P[2])*d + … P[m-2])*d + P[m-1]

```
p = P[0];
For j = 1 to m-1
     p = p*d + P[j]
```

The numbers p and t can be computed in $\theta$(m) time.

- To compute the number t after shifting the text window, it can be done in constant time based on the number of the previous text window

  For example:

**36415**

| 3 | 6 | 4 | 1 | 5 | 2 |
|---|---|---|---|---|---|

**i**

$$(36415 - 3 * 10^4) * 10 + 2$$

In general,
new = (old – MSB * $d^{m-1}$) * d + LSB
$d^{m-1}$ is pre-calculated as below

```
dM = 1;
For j = 1 to m-1
    dM = dM*d
// dM = d^(m-1)
```

```
t = (t – T[i]*dM)*d + T[i+m]
// t before this is the number for T[i .. i+m-1]
// t after this is the number for T[i+1 .. i+m]
```

- If the pattern is long (e.g. m = 100), then the resulting number will be enormous. Overflow may occur.

- For this reason, we <u>hash</u> the value by taking it mod a prime number *q*. This prime number should be large.

  1) **Hash** the pattern to a number, hp
  2) **Hash** the first *m*-character text window to a number, ht
  3) If hp and ht are equal, compare the pattern with the text window. If equal, pattern found and exit
  4) If not end-of-text, shift the text window one character right and **(re)hash** it to a number ht, go to step 3); else pattern not found and exit

Note: if hp = ht, it does not necessarily imply that T[i..i+m-1] = P[0..m-1].
However, if hp $\neq$ ht, definitely T[i..i+m-1] $\neq$ P[0..m-1]

P:

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |

m = 5, q = 13, hp = 5

T:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |

ht =6   4   8   3   1   2

ht = 12

ht = 25%13=12

ht = 18%13=5 compare P and T

ht = 5 compare P and T

**Found!**

- The mod function (% in Java) is particularly useful in this case due to several of its inherent properties:
  - (x+y) mod q = [(x mod q) + (y mod q)] mod q
  - (x mod q) mod q = x mod q
  - xy mod q = [(x mod q)(y mod q)] mod q

  Example:

(3*10+6) mod 13

= ((3*10) mod 13 + 6 mod 13) mod 13

= ( ( (3 mod 13) * (10 mod 13) ) mod 13 + 6 mod 13) mod 13

= (4 + 6) mod 13

= 10

- To calculate hp, the hash value for P[0..m-1], call hash(P, m, base). The hash function is also used to compute the value of the first text window

```
int hash(Txt, m, d)
{
    int h = Txt[0] % q;
    for (int i =  1; i < m; i++)
        h = (h * d + Txt[i] )% q;
    return h;
}
```

E.g. P = "36415", **q = 7**

h = 3

i = 1, h = 1

i = 2, h = 0

i = 3, h = 1

i = 4, h = 1

hash("36415", 5, 10) = 1

Ref:

```
p = P[0];
For j = 1 to m-1
    p = p*d + P[j]
```

The numbers hp and ht can be computed in $\theta(m)$ time.

- After finding ht for T[i .. m-1], ht for T[i+1 .. m] can be calculated by rehash(T, i, m, ht) in constant time $\theta(1)$.

```
int rehash(T, i, m, ht)
{    oldest = (T[i] * dM) % q;    // dM is d^(m-1) mod q
     oldest_removed = ((ht + q) - oldest) % q;
     return (oldest_removed * d + T[i+m]) % q;
}
```

- Compare with no hashing:

```
t = (t – T[i]*dM)*d + T[i+m]
// dM is d^(m-1)
// t before this is the number for T[i .. i+m-1]
// t after this is the number for T[i+1 .. i+m]
```

# Example

```
int rehash(T, i, m, ht)
{    oldest = (T[i] * dM) % q;
     oldest_removed = ((ht + q) - oldest) % q;
     return (oldest_removed * d + T[i+m]) % q;
}
```

ht=1, dM = 4, q = 7

| | i | i+1 | i+2 | i+3 | i+4 | i+5 |
|---|---|---|---|---|---|---|
| T | 3 | 6 | 4 | 1 | 5 | 2 |

Rehash(T, i, 5, 1) =

- Oldest = 5
- Oldest_removed = 3
- Return 32 % 7 = 4

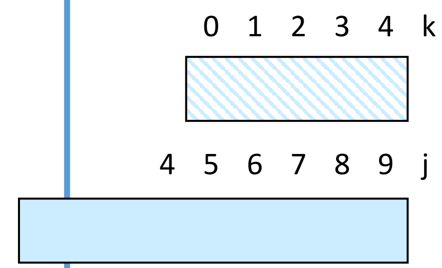
```
// computed once
dM = 1;
For j = 1 to m-1
     dM = dM*d % q
```

```
int RKscan(P, T)
{     m = Length(P);
      n = Length(T);
      dM = 1;
      For j = 1 to m-1     dM = dM*d % q;    // d = |Σ|

      hp = hash(P, m, d);      // θ(m)
      ht = hash(T, m, d);

      for (j = 0; j <= n - m; j++) {
            if (hp == ht && equal_string(P, T, 0, j, m))
                  return j;
            if (j < n-m)   ht = rehash(T, j, m, ht);
      }
      return -1;  // pattern not found
}
```

Maximum
n-m+1
iterations



- The running time of Rabin-Karp algorithm in the worst case is $\theta((n - m + 1)m)$
- However, in many applications, the expected running time is $O(n+m)$ plus the time required to process spurious hits.
  - $O(m)$ time for the 2 hash() calls
  - Close to $O(n)$ time on the for loop
- The number of spurious hits can be kept low by using a large prime number q for the hash functions

# The Boyer-Moore Algorithm

- It is a very efficient algorithm for string searching
- The text being scanned is T with *n* characters
- The pattern we are looking for is P with *m* characters
- Process the text T[1..*n*] from left to right
- Scan the pattern P[1..*m*] **from right to left**
- Preprocessing to generate two tables based on which to slide the pattern as much as possible after a mismatch
- It performs even better with long patterns

```
int BMscan(char[]P char[]T, int m,
                    int[]charJump, int []matchJump )
{ int j;   int k;
  j = m;  k = m;
  while (j <= n) {
        if (k < 1)  return j + 1;    //match found
        if (T[j] == P[k])    {   j--;  k--;  }
        else {     j += max(charJump[T[j]], matchJump[k]);
                   k = m;     }
   }
   return -1;    // match not found
 }
```

charJump and matchJump are the 2 tables generated in a preprocessing step

Assume the 1st character is at P[1] and T[1] respectively

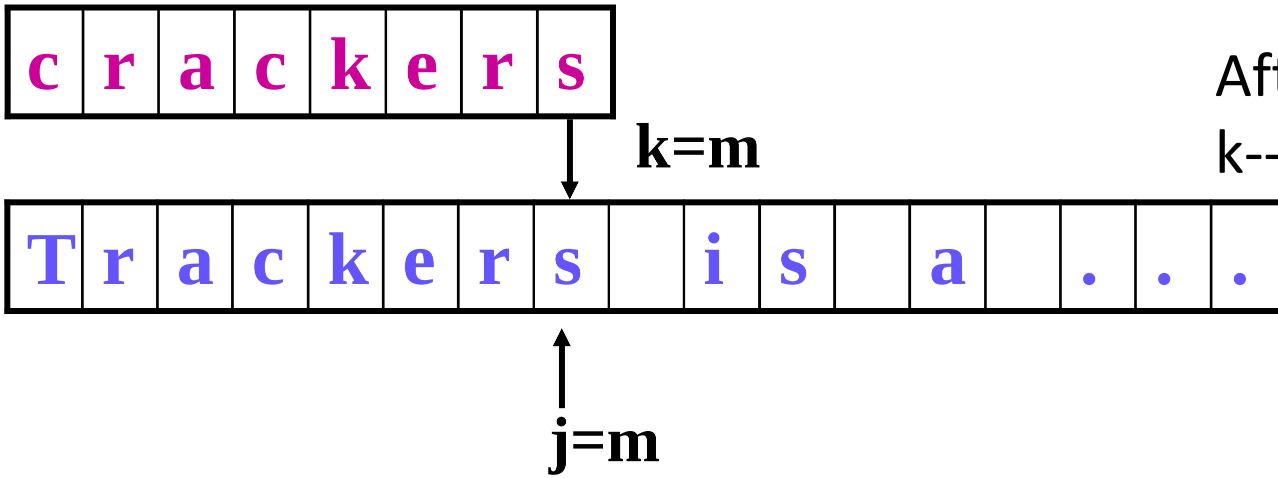
# Examples

To start.

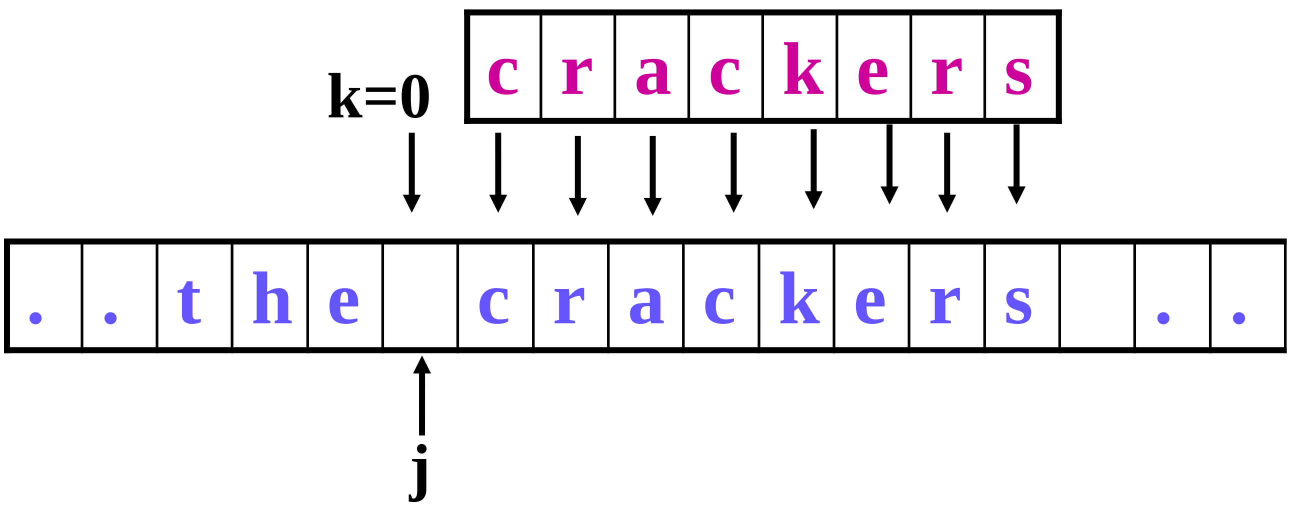
k = m, j = m

After a match,

k--, j--

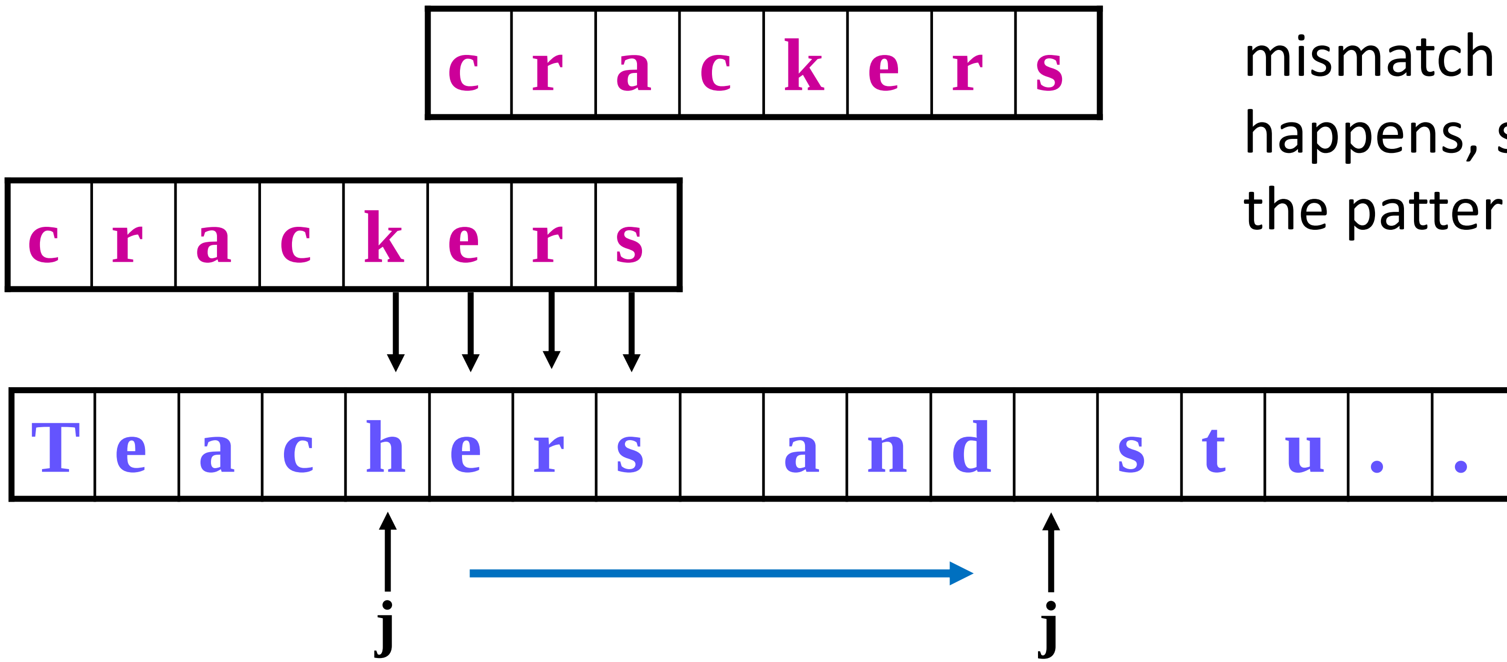



When k reaches 0, all characters have been matched: Return j+1



## Example

When a mismatch happens, shift the pattern

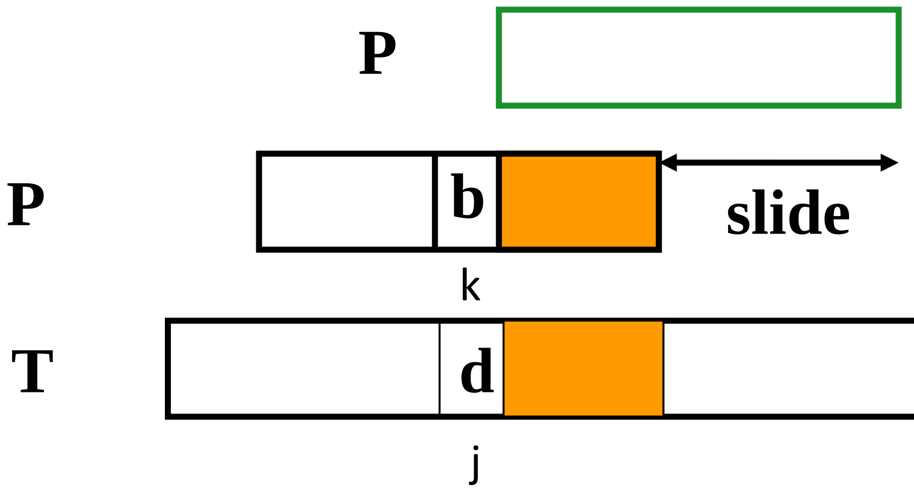



Shift the pattern as much as possible – increment j as much as possible for the next comparison:

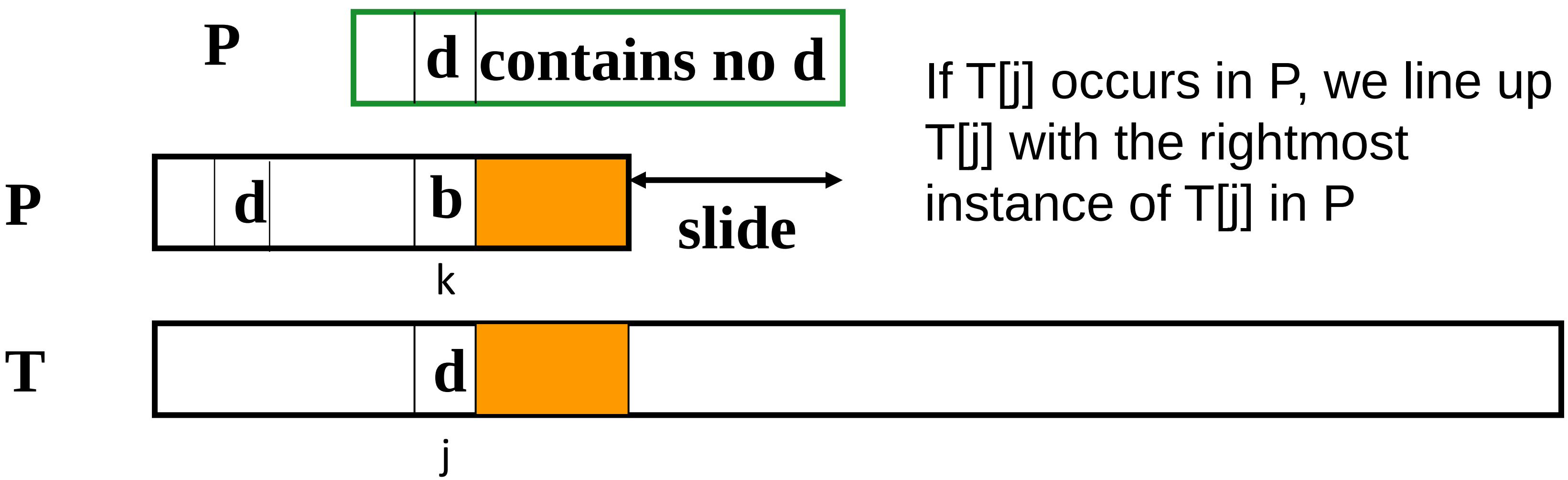
{ j += max(charJump[T[j]], matchJump[k]);

k = m; }

# Preprocessing to compute charJump

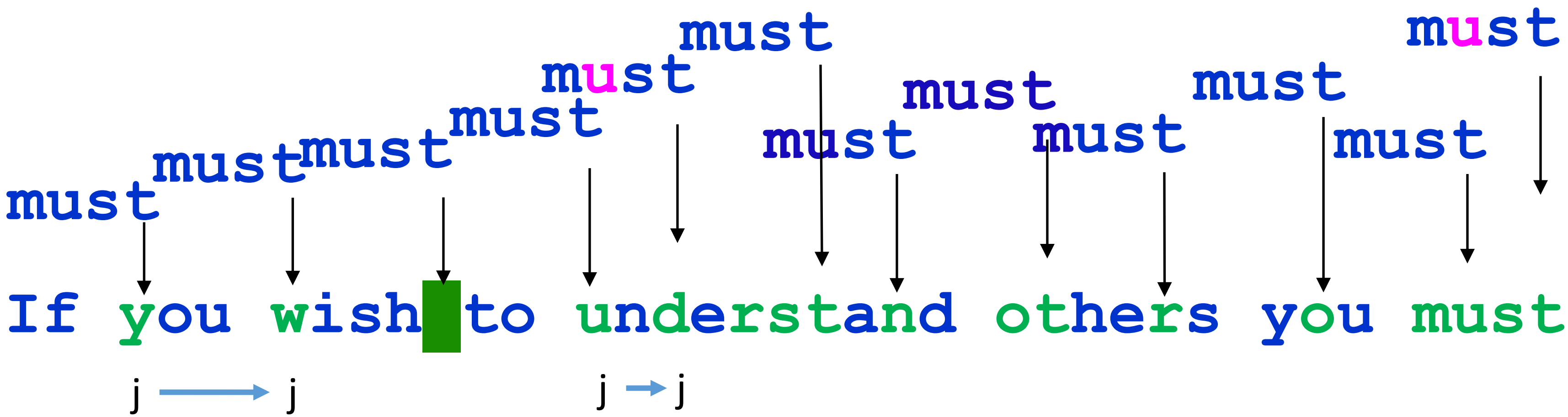



If T[j] does not appear in P at all, we line up P after T[j]

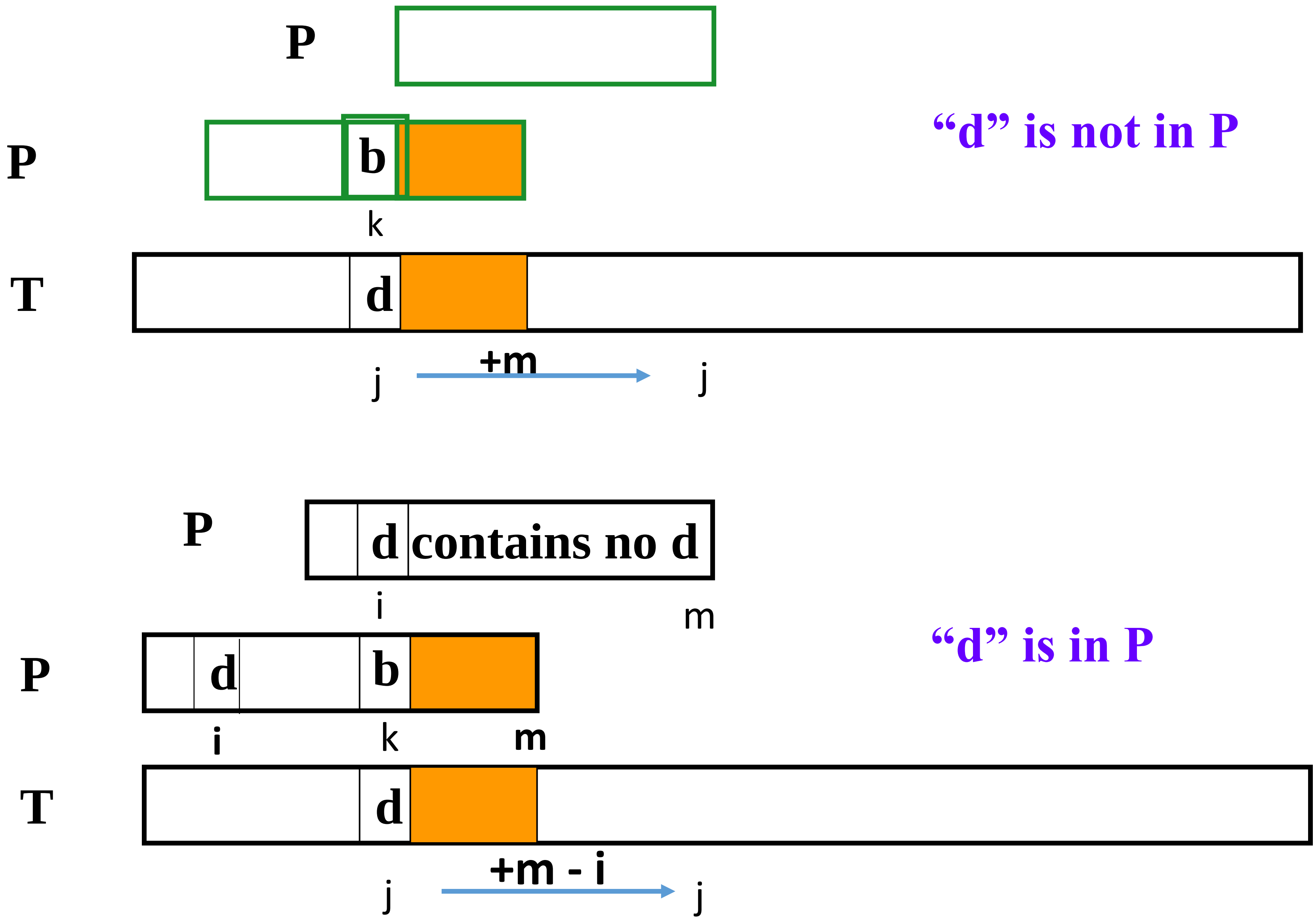



If T[j] occurs in P, we line up T[j] with the rightmost instance of T[j] in P

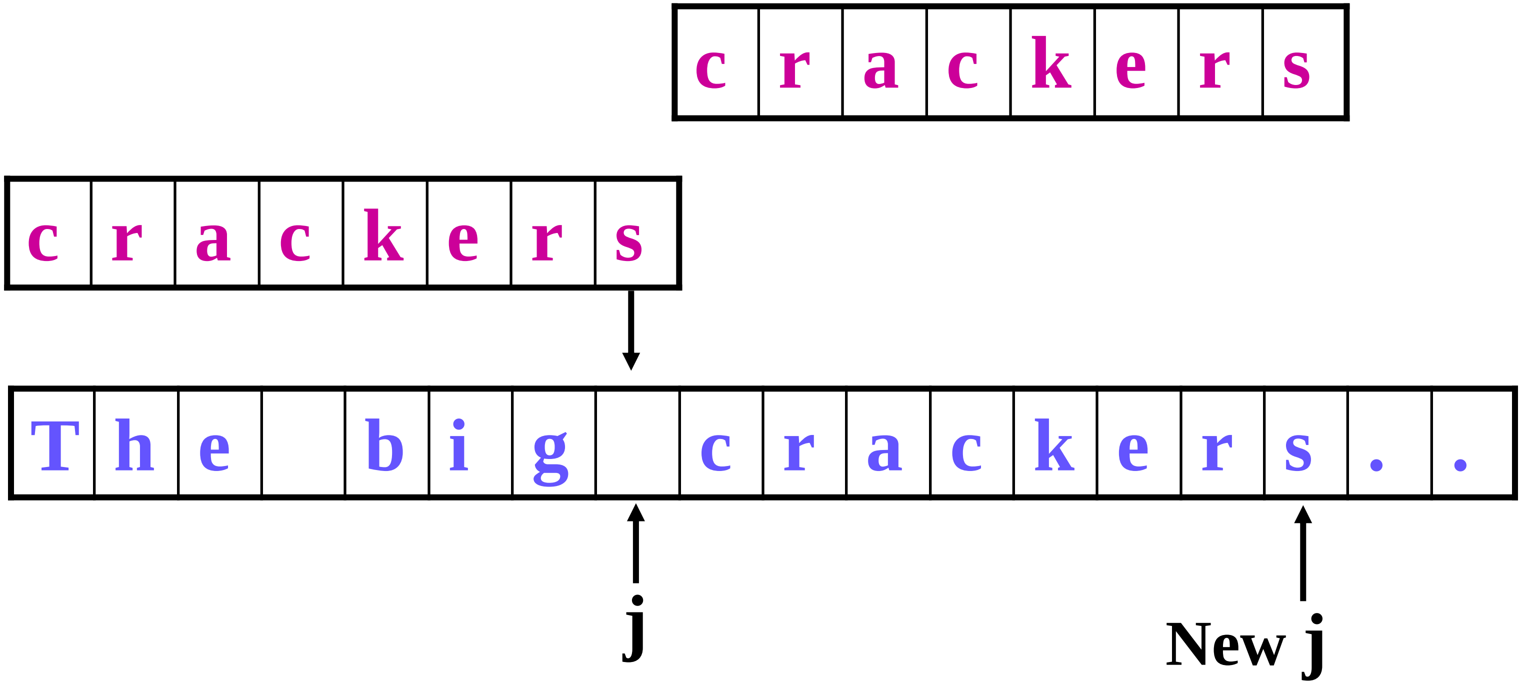
# Example




- Many of the *n* characters in the text are never compared – sublinear complexity
- We need to calculate how the text index j should be incremented to begin the next right-to-left scan of the pattern

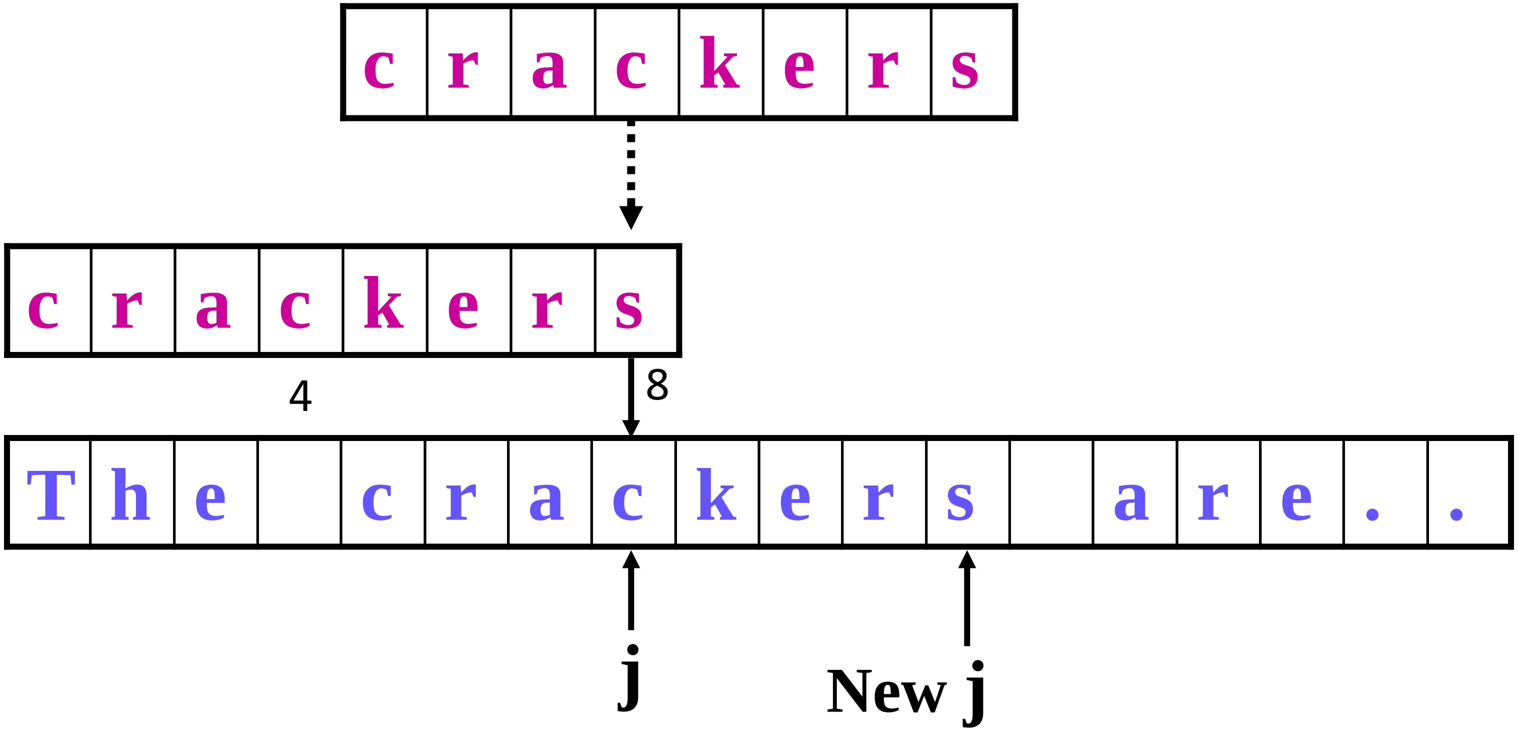

P
P
b
k
T
d
+m
j
j
"d" is not in P
P
d
contains no d
i
m
P
d
b
i
k
m
T
d
+m - i
j
j
"d" is in P

# Example




To line up P after T[j], e.g. ‘ ’, P is slid 8 places to the right:
j = j + 8

# Example




To line up T[j], e.g. ‘c’, with the rightmost ‘c’ in P, P is slid 4 places to the right: j = j + 8 - 4

- Computing the jumps for all the characters:

```
void computeJumps(char [] P, int m,
                  int alpha, int [] charJump)
{   char ch;  int k;
    for (ch = 0; ch < alpha; ch++)
        charJump[ch] = m;
    for (k = 1; k <= m; k++)
            charJump[ P[k] ] = m - k;
}
```

Number of characters in character set

Position from the end

Complexity is O(|Σ| + m)

Notice that if a character appears more than once, we take the right-most occurrence.

| c | r | a | c | k | e | r | s |
|---|---|---|---|---|---|---|---|

First:  charJump['c'] = 8-1
Then: charJump['c'] = 8-4

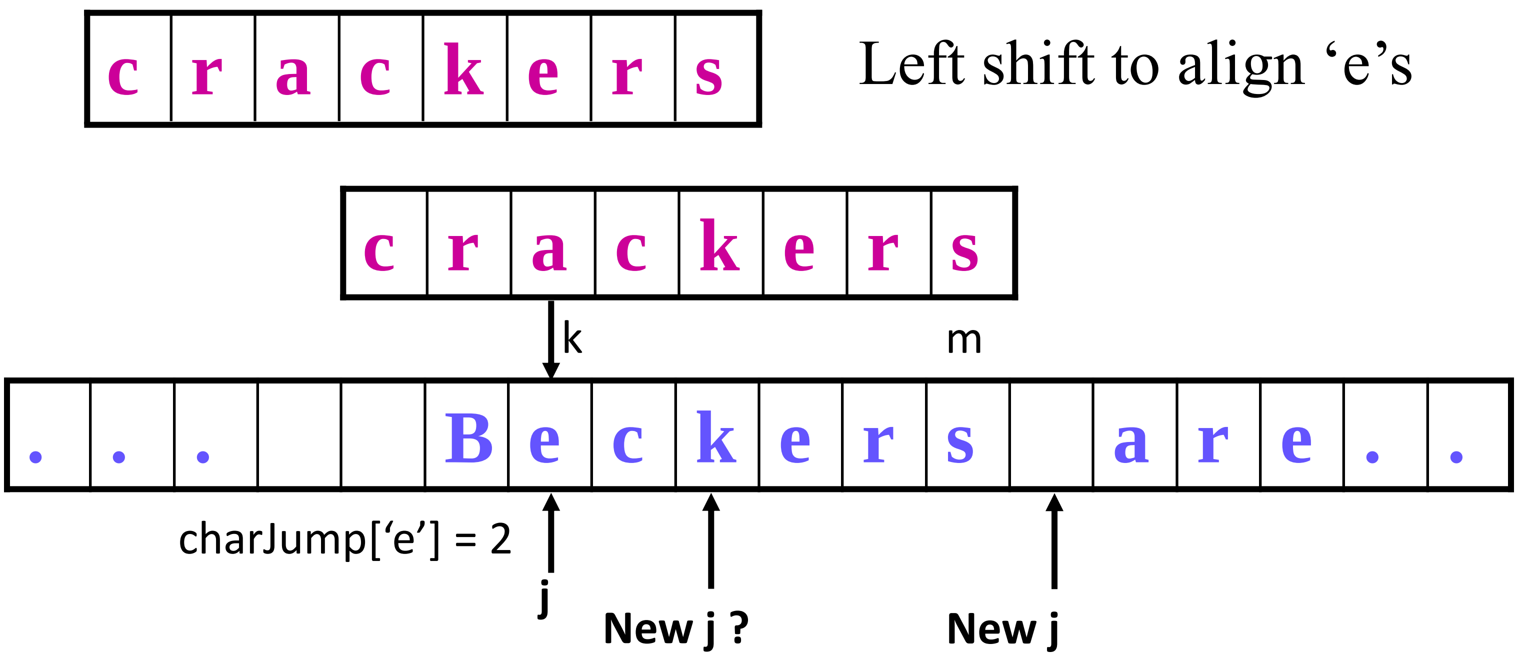
E.g. for

| r | a | t | s | a | n | d | c | a | t | s |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

$m = 11$,

| charJump | 2 | 11 | 3 | 4 | … | 5 | … | 10 | 0 | 1 | … |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | a | b | c | d | … | n | … | r | s | t | … |

Sometimes this heuristic fails, for example,



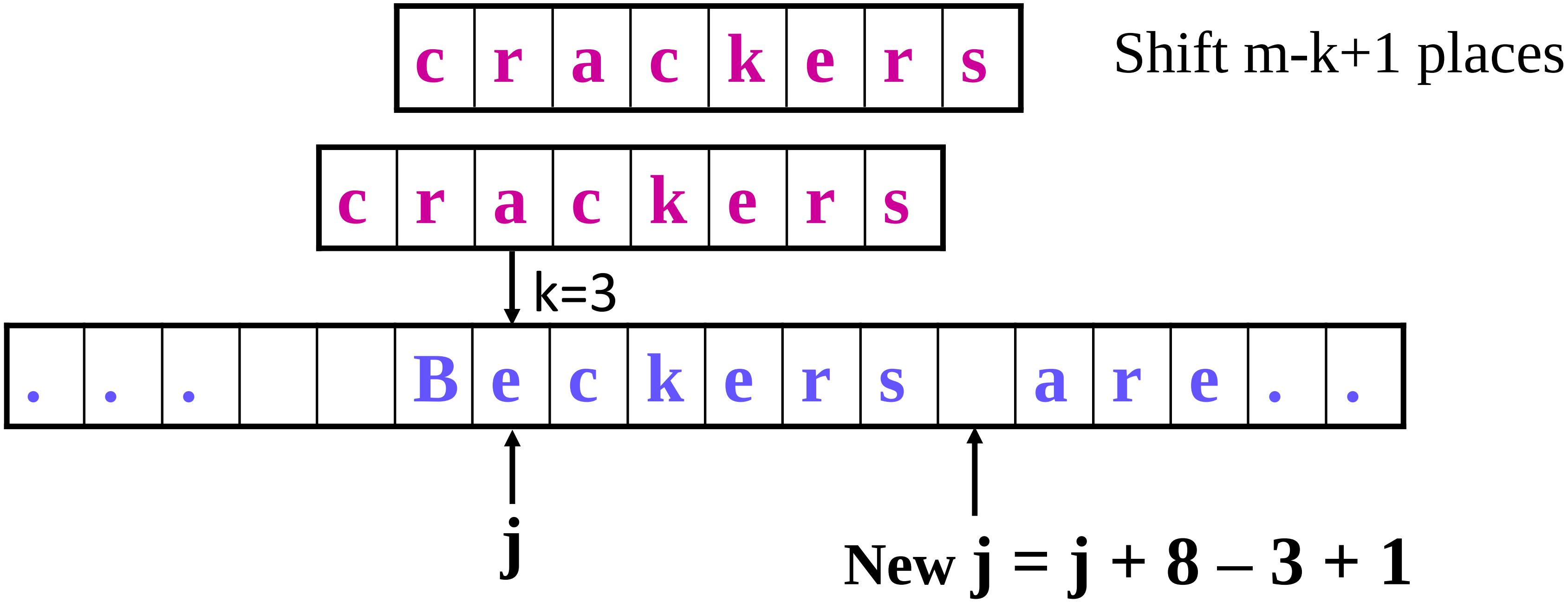
# Simplified Boyer-Moore (using charJump only)

```
int simpleBMscan(char[]P char[]T, int m,  int[]charJump)
{ int j;   int k;
  j = m;  k = m;
  while (j <= n) {
        if (k < 1)  return j + 1;    //match found
        if (T[j] == P[k])    {   j--;  k--;  }
        else {     j += max(charJump[T[j]], m-k+1);
                   k = m;      }
   }
   return -1;    // match not found
 }
```

E.g.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| c | r | a | c | k | e | r | s |

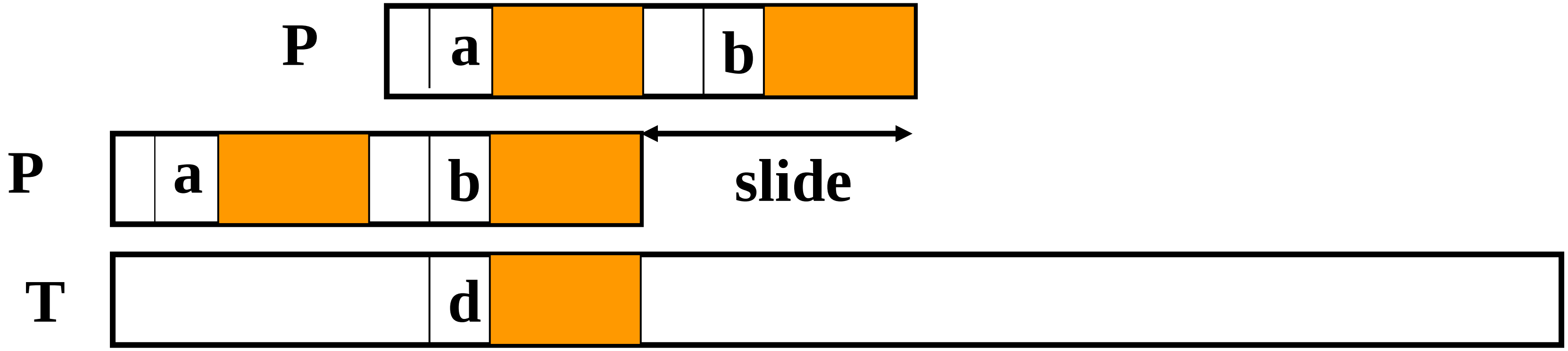
| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| charJump | 5 | 8 | 4 | 8 | 2 | … | 3 | … | 1 | 0 | … |
| | a | b | c | d | e | … | k | … | r | s | … |



# Preporocessing to compute matchJump

This heuristic tries to derive the maximum shift from the structure of the pattern. It is defined for each of the characters in P.

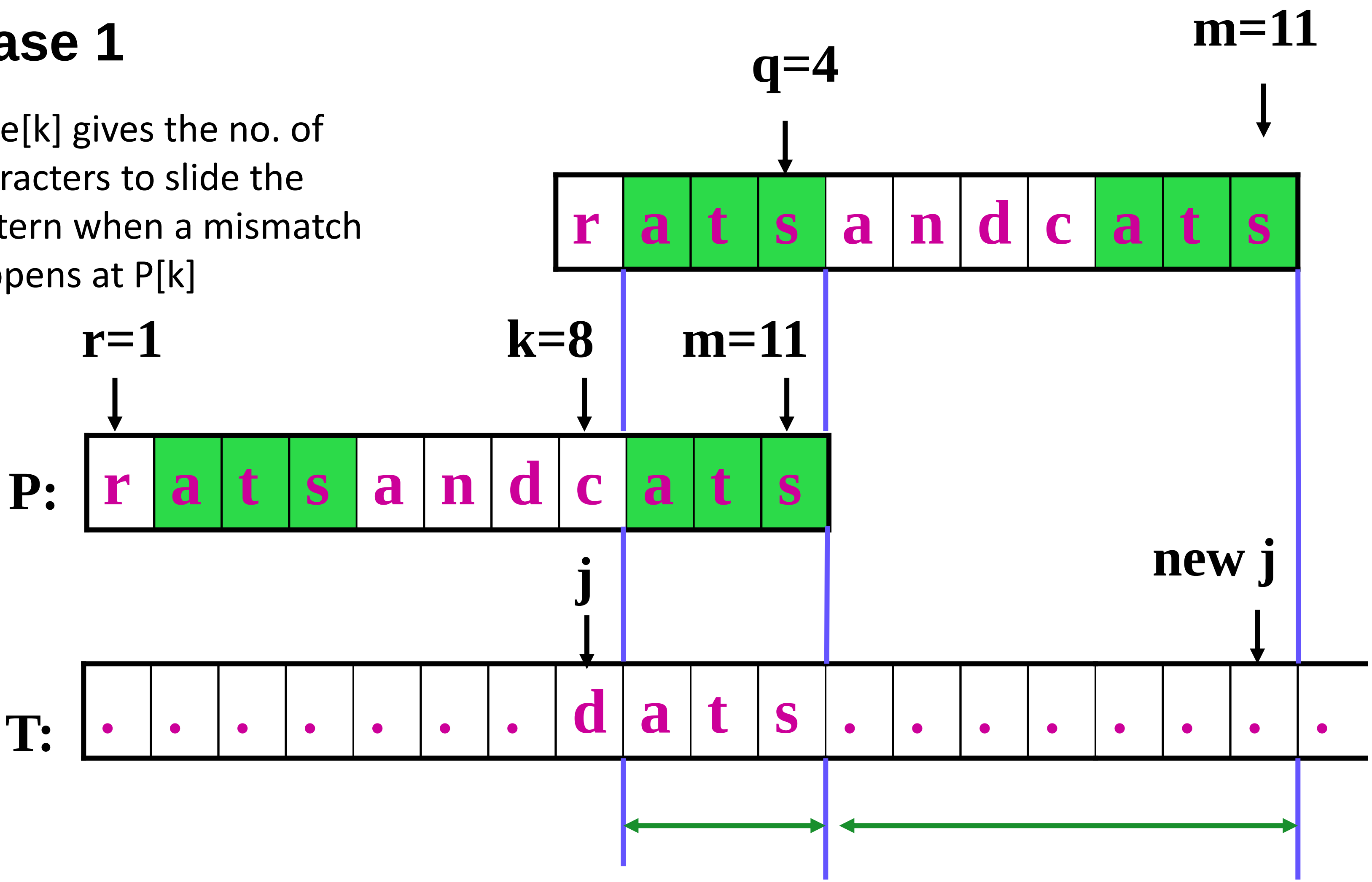
**Case 1**: The matching suffix occurs earlier in the pattern, but preceded by a different character




We line up the earlier occurrence of the suffix in P with the matched substring in T

## Case 1

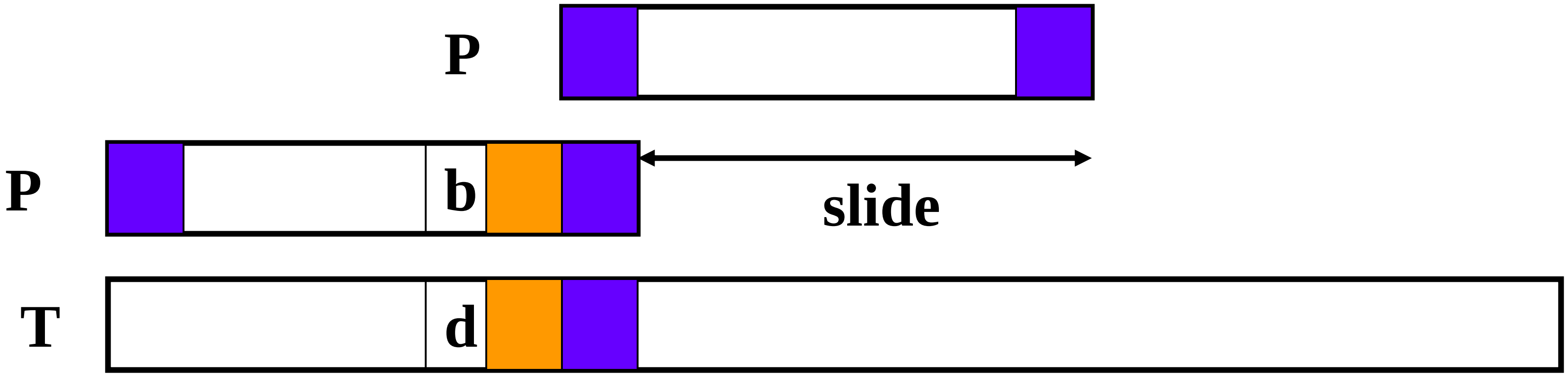
Slide[k] gives the no. of characters to slide the pattern when a mismatch happens at P[k]




**matchJump[k] = m-k + Slide[k]**

**Slide[k] = m - q (P[r] ≠ P[k])**

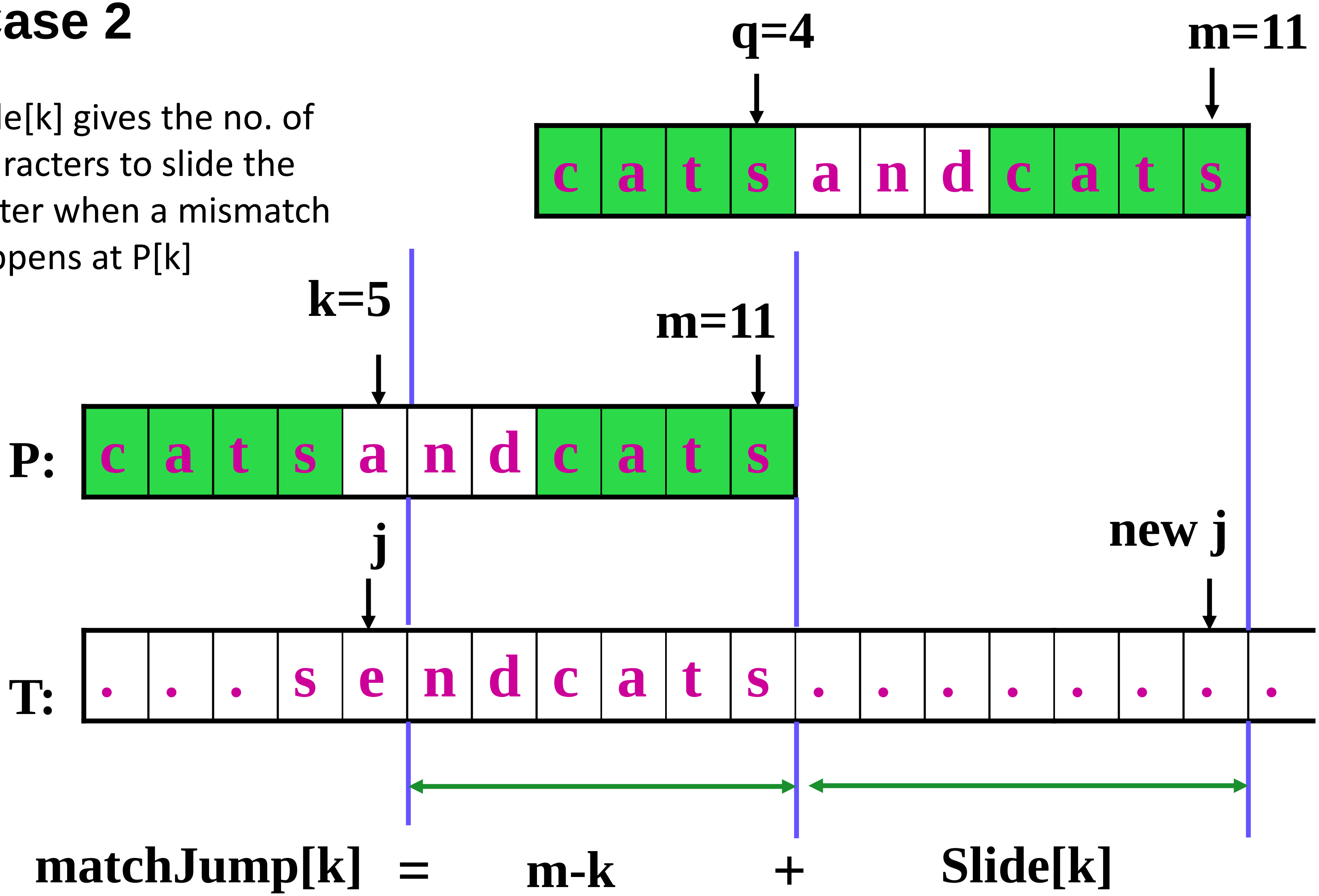
**Case 2**: Only part of the matching suffix occurs at the beginning of the pattern (a prefix).




We line up the prefix in P with part of the matched substring in T

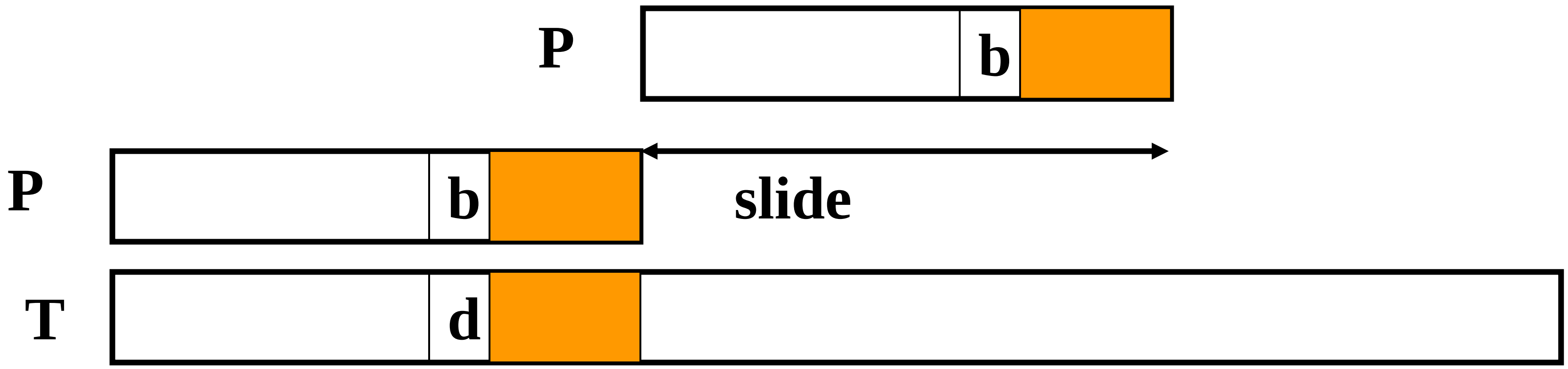
## Case 2

Slide[k] gives the no. of characters to slide the patter when a mismatch happens at P[k]




**Slide[k] = m - q**

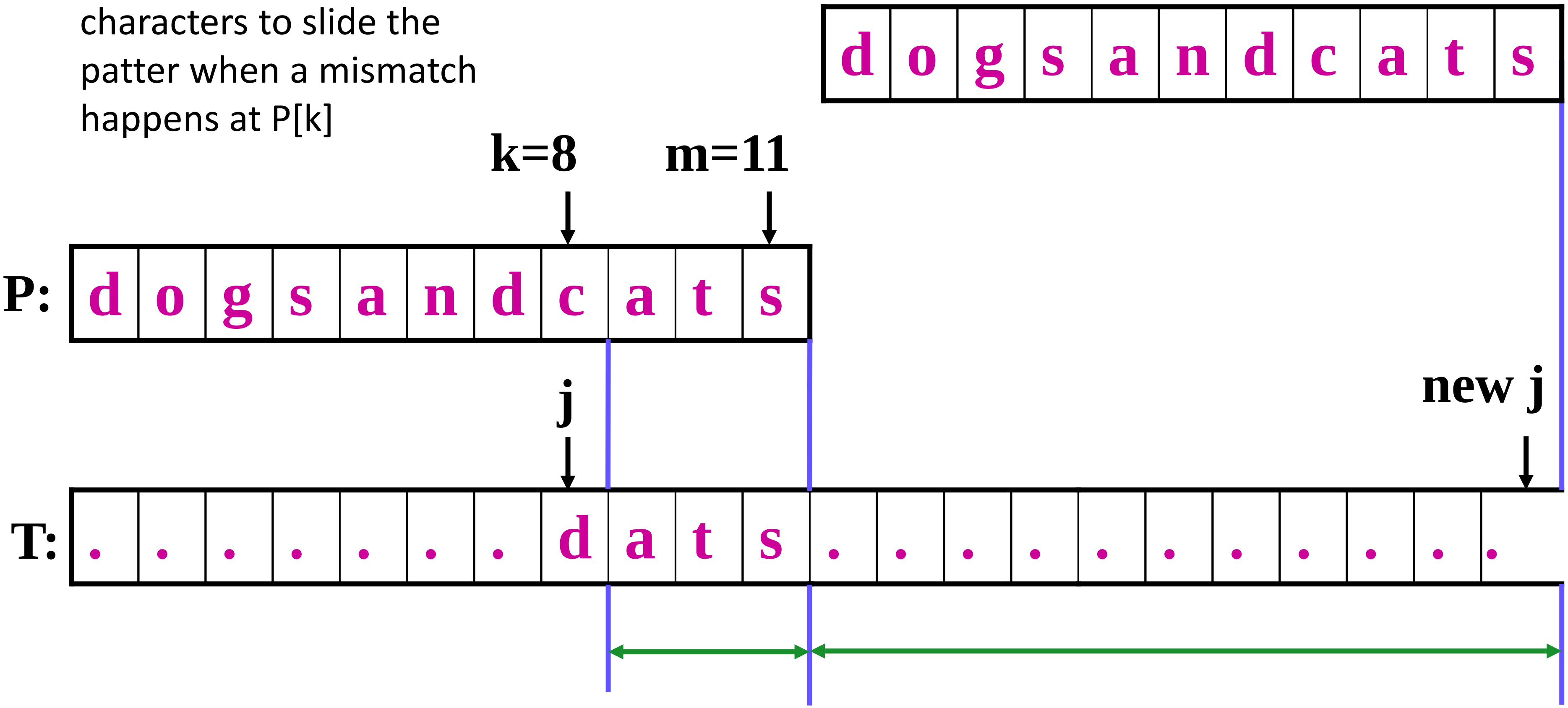
**Case 3**: There is no other occurrence of the matching suffix in the pattern. (Case 1 and Case 2 do not happen)




We line up P after the matched substring in T

# Case 3

Slide[k] gives the no. of characters to slide the patter when a mismatch happens at P[k]

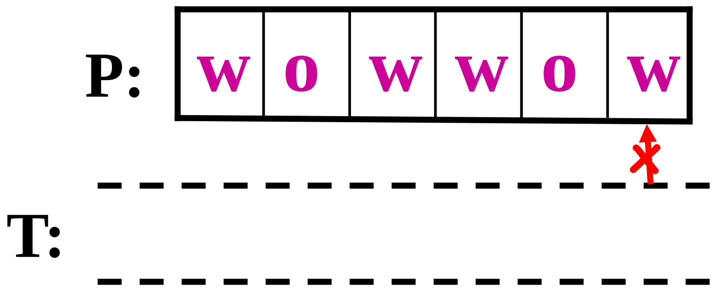



**matchJump[k] = m-k + Slide[k]**

**Slide[k] = m (= m – q where q is 0)**

# Example: computing matchJump

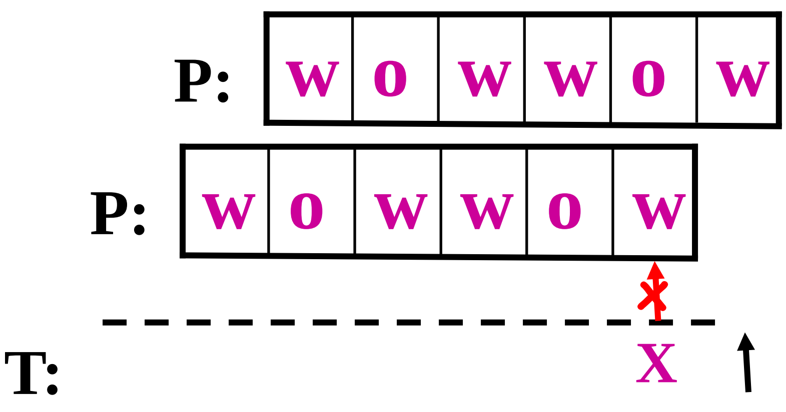
Slide[m] = 1

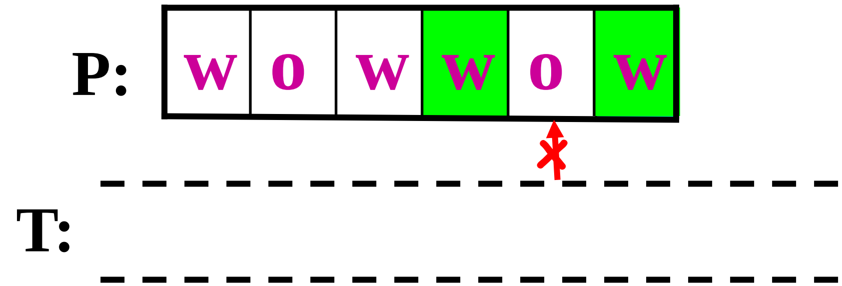



Matched = 0    (m-k)

Slide[6] = 1

**matchJump[6] =   1**

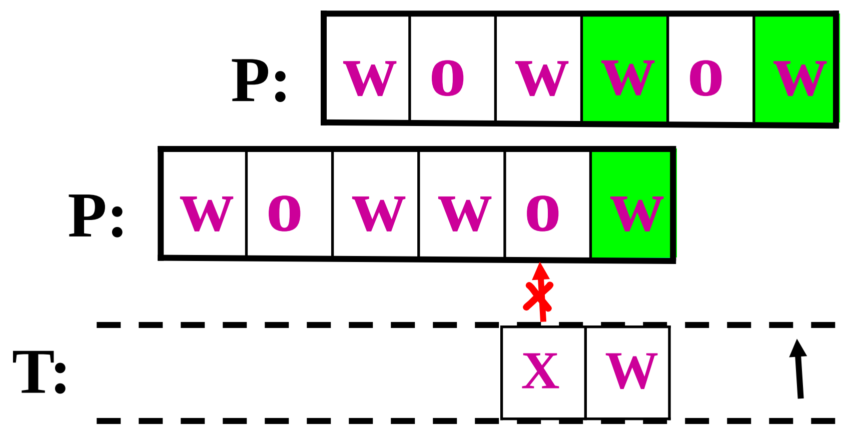


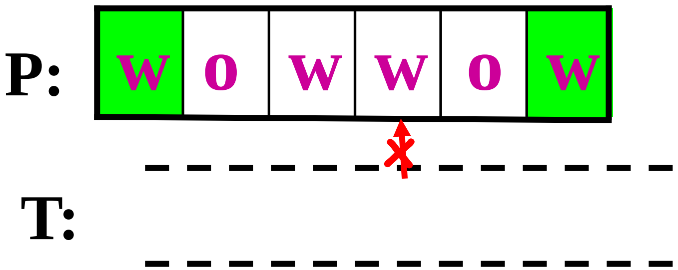




Matched = 1   (m-k)

Slide[5] = 2   (m-q)

**matchJump[5] =   3**

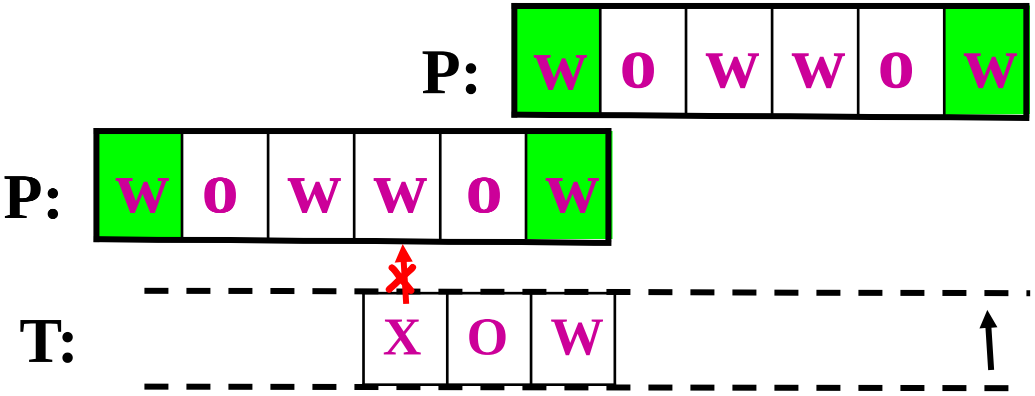


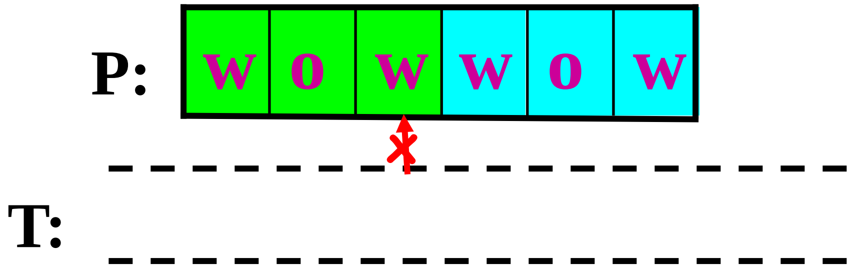



Matched = 2   (m-k)

Slide[4] = 5   (m-q)

**matchJump[4] =   7**

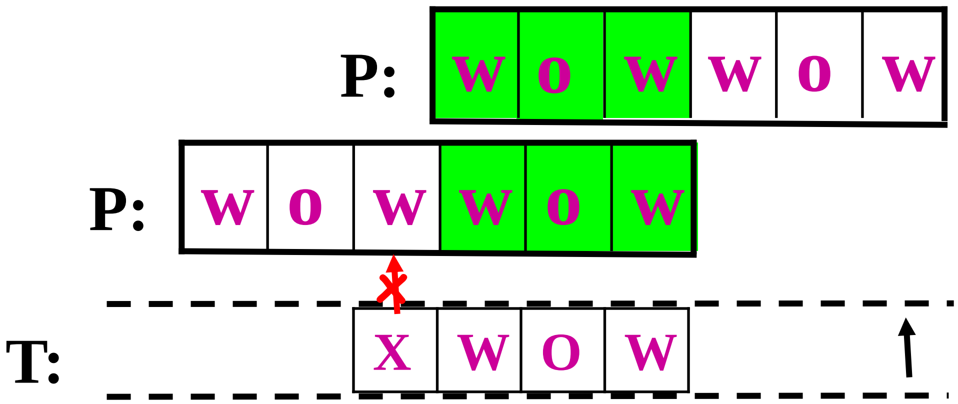


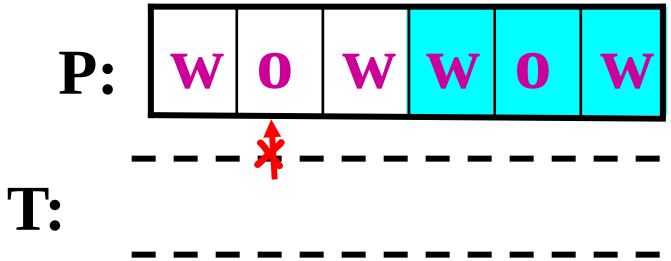



Matched = 3   (m-k)

Slide[3] = 3   (m-q)

**matchJump[3] =   6**

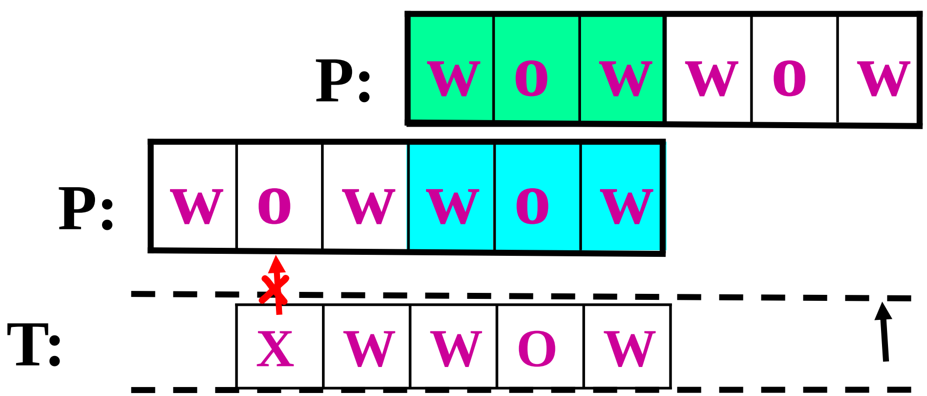


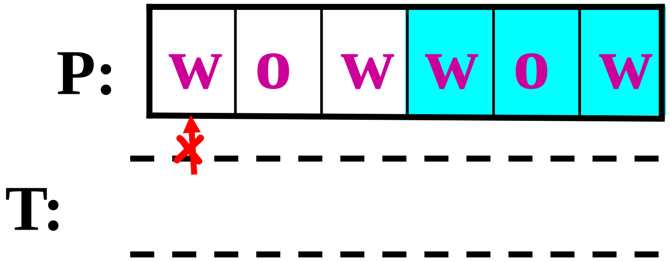



Matched = 4 (m-k)

Slide[2] = 3 (m-q)

**matchJump[2] = 7**



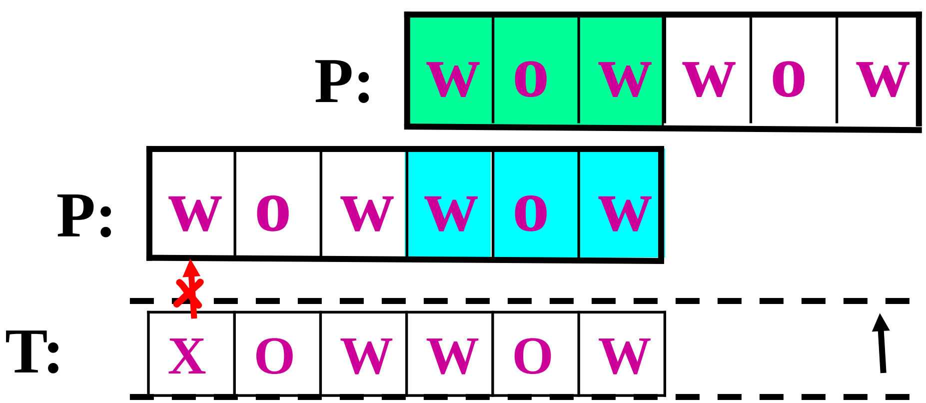



Matched = 5 (m-k)

Slide[1] = 3 (m-q)

**matchJump[1] = 8**

| | | | | | | |
|---|---|---|---|---|---|---|
| **matchJump** | 8 | 7 | 6 | 7 | 3 | 1 |



# BMscan example:  Pattern is WOWWOW

charjump['O'] = 1, charjump['W'] = 0, charjump[X] = 6,

**matchJump**

| | | | | | |
|---|---|---|---|---|---|
| 8 | 7 | 6 | 7 | 3 | 1 |

Match found after 18 comparisons

**WOWWOW**

**WOWWOW**

**WOWWOW**

**WOWWOW**

P = **WOWWOW**

T = **LOWOWNOWLWOWWOMMOWWOWWOW**

## Another example

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| R | A | T | S | C | A | T | S |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| R | A | T | S | C | A | T | S |

Matched = m-k = 0, Slide[8] = 1

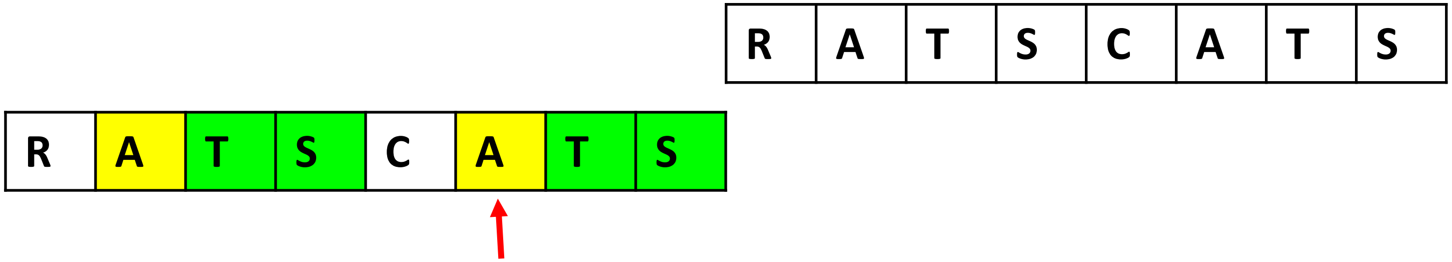
matchJump[8] = 1

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| R | A | T | S | C | A | T | S |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| R | A | T | S | C | A | T | S |

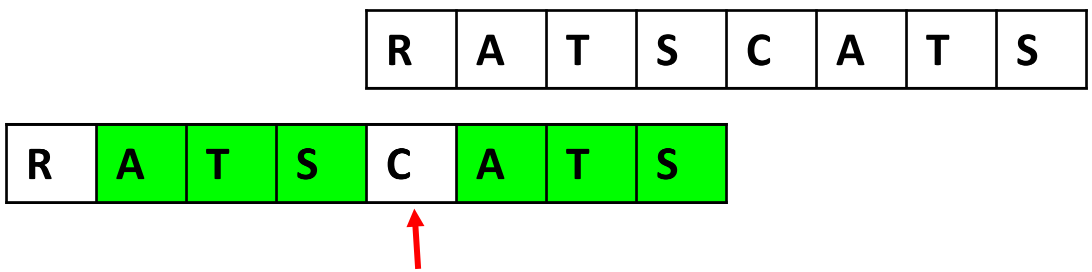
Matched = m-k = 1, Slide[7] = m = 8

matchJump[7] = 9



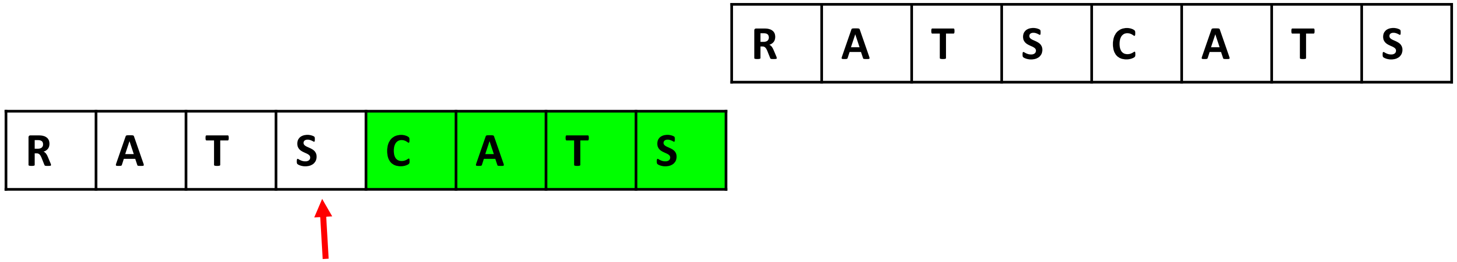

Matched = m-k = 2, Slide[6] = m = 8

matchJump[6] = 10

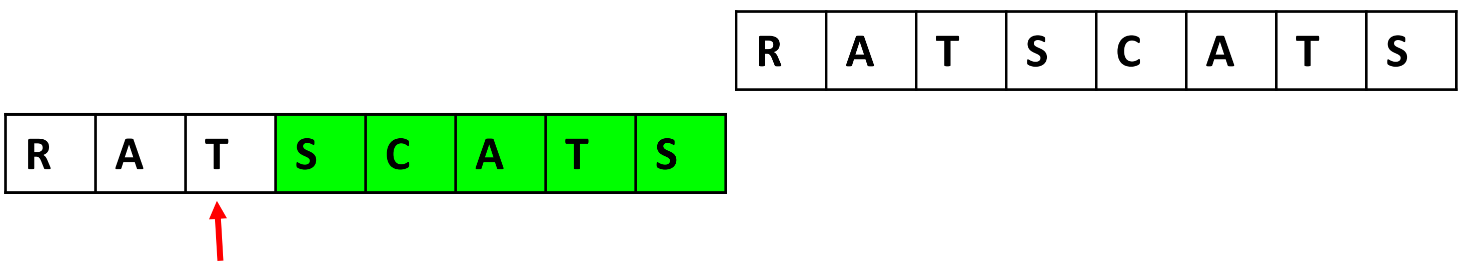



Matched = m-k = 3, Slide[5] = m-q = 4

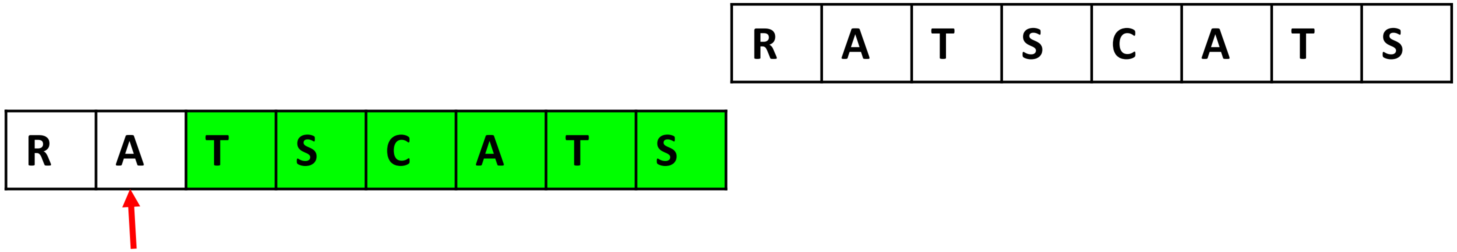
matchJump[5] = 7



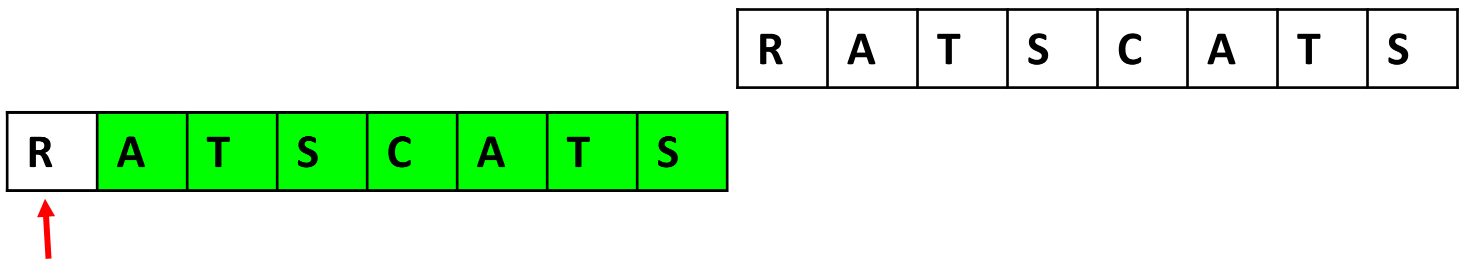

Matched = m-k = 4, Slide[4] = m = 8
matchJump[4] = 12




Matched = m-k = 5, Slide[3] = m = 8
matchJump[3] = 13



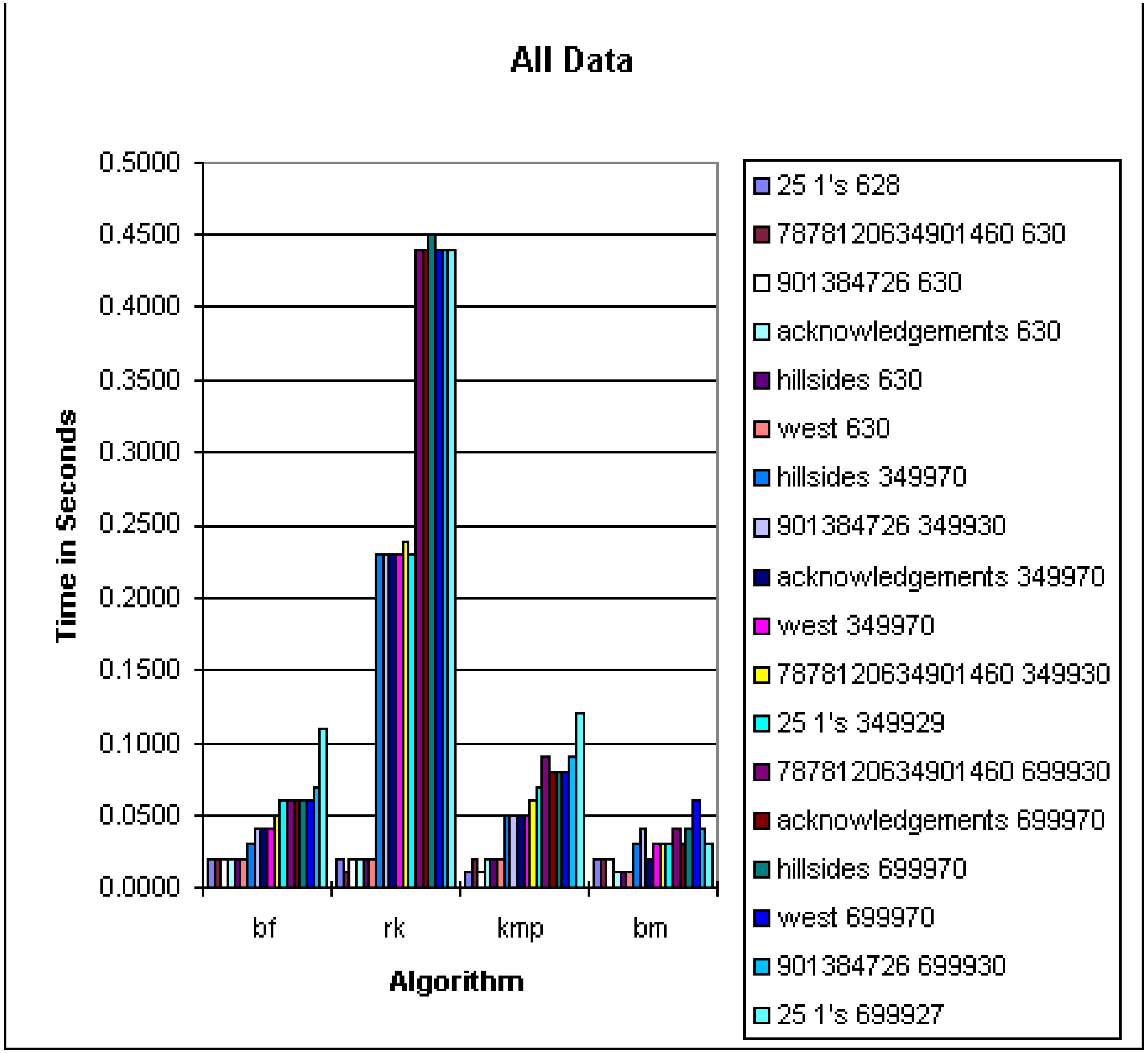

Matched = m-k = 6, Slide[2] = m = 8
matchJump[2] = 14




Matched = m-k = 7, Slide[1] = m = 8
matchJump[1] = 15

| | R | A | T | S | C | A | T | S |
|---|---|---|---|---|---|---|---|---|
| matchJump | **15** | **14** | **13** | **12** | **7** | **10** | **9** | **1** |

- Brute-Force Algorithm (bf)
- Rabin-Karp Algorithm (rk)
- Knuth-Morris-Pratt Algorithm (kmp)
- Boyer-Moore Algorithm (bm)



- Brute Force behaved better than we expected
  - because worst case is not common. Worst case would occur when the pattern and the text produced a near match.
- Rabin-Karp behaved much worse
  - Rabin-Karp has several function calls. These are expensive, timewise.
  - Any division, including mod, is time expensive.
  - The conversion from character values to numeric values takes time.

- Boyer-Moore algorithm is considered the most efficient string-matching algorithm in usual applications, for example, in text editors.
- Moore says the algorithm has the peculiar property that, roughly speaking, the longer the pattern is, the faster the algorithm goes.
- The payoff is not as for binary strings or for very short patterns.
- For binary strings Knuth-Morris-Pratt algorithm is recommended.
- For the very shortest patterns, the brute force algorithm may be better.

- What else do we learn from the BM algorithm?
  - Designing algorithms to solve problems often needs insights into a problem’s structure – analyse the problem carefully before thinking about its solution