Computer Science for Practicing Engineers

So khớp các chuỗi (Pattern Matching)

TS. Huỳnh Bá Diệu
Email: dieuhb@gmail.com
Phone: 0914146868

So khớp mẫu (Pattern Matching)

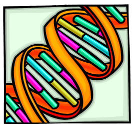
Các thuật toán so khớp chuỗi

1. Brute-force algorithm
2. Boyer-Moore algorithm
3. Knuth-Morris-Pratt algorithm

String

- A string is a sequence of characters
- An alphabet Σ is the set of possible characters for a family of strings
- Example of alphabets:
  - ASCII
  - Unicode
  - {0, 1}
  - {A, C, G, T}
- Let $\boldsymbol{P}$ be a string of size $\boldsymbol{m}$
  - A substring $\boldsymbol{P}[\boldsymbol{i} \,..\, \boldsymbol{j}]$ of $\boldsymbol{P}$ is the subsequence of $\boldsymbol{P}$ consisting of the characters with ranks between $\boldsymbol{i}$ and $\boldsymbol{j}$
  - A prefix of $\boldsymbol{P}$ is a substring of the type $\boldsymbol{P}[0 \,..\, \boldsymbol{i}]$
  - A suffix of $\boldsymbol{P}$ is a substring of the type $\boldsymbol{P}[\boldsymbol{i} \,..\boldsymbol{m} - 1]$
- Given strings $\boldsymbol{T}$ (text) and $\boldsymbol{P}$ (pattern), the pattern matching problem consists of finding a substring of $\boldsymbol{T}$ equal to $\boldsymbol{P}$
- Applications:
  - Text editors
  - Search engines
  - Biological research

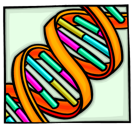
So khớp mẫu (Pattern Matching) Brute-Force Algorithm



- The brute-force pattern matching algorithm compares the pattern $\boldsymbol{P}$ with the text $\boldsymbol{T}$ for each possible shift of $\boldsymbol{P}$ relative to $\boldsymbol{T}$, until either
  - a match is found, or
  - all placements of the pattern have been tried
- Brute-force pattern matching runs in time $\boldsymbol{O}(\boldsymbol{nm})$
- Example of worst case:
  - $\boldsymbol{T} = \boldsymbol{aaa} \ldots \boldsymbol{ah}$
  - $\boldsymbol{P} = \boldsymbol{aaah}$
  - may occur in images and DNA sequences
  - unlikely in English text

```
Algorithm BruteForceMatch(T, P)
    Input text T of size n and pattern
        P of size m
    Output starting index of a
        substring of T equal to P or −1
        if no such substring exists
    for  i ← 0 to n − m
        { test shift i of the pattern }
        j ← 0
        while j < m ∧ T[i + j] = P[j]
            j ← j + 1
        if  j = m
            return  i {match at i}
        else
            break while loop {mismatch}
    return  -1 {no match anywhere}
```

So khớp mẫu (Pattern Matching) Brute-Force Algorithm



```
Algorithm BruteForceMatch(T, P)
    Input text T of size n and pattern
        P of size m
    Output starting index of a
        substring of T equal to P or −1
        if no such substring exists
    for $i \leftarrow 0$ to $n - m$
        { test shift i of the pattern }
        $j \leftarrow 0$
        while $j < m \wedge T[i + j] = P[j]$
            $j \leftarrow j + 1$
        if $j = m$
            return  i {match at i}
        else
            break while loop {mismatch}
    return  -1 {no match anywhere}
```

```
int BruteForceMatch(string T, string P)
{
        int n = T.length(), m= P.length();
        for(int i=0; i<=n-m; i++)
          {
             int j=0;
             while( j < m && T[i + j] == P[j])
j++;
             if(j==m)  return i;
          }
         return -1;
}
string text= "ABABDABACDABABCABAB";
string pat = "ABABCABAB";
```

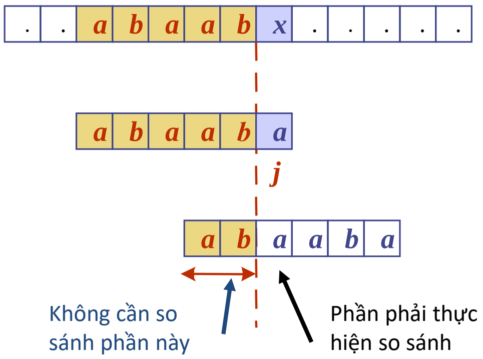
So khớp mẫu (Pattern Matching)

- Boyer–Moore

- Raita algorithm

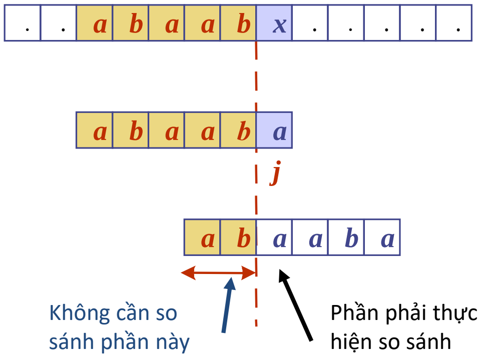
So khớp mẫu (Pattern Matching)
The KMP Algorithm - Motivation

- Thuật toán Knuth-Morris-Pratt’s so sánh mẫu theo thứ tự **left-to-right**, nhưng dịch chuyển các mẫu thông minh hơn phương pháp brute-force.
- **Khi có ký tự không giống nhau, ta có thể dịch vị trí để bắt đầu so sánh về trước nhiều nhất là bao nhiêu để tránh các so sánh không cần thiết???**


. . a b a a b x . . . . .
a b a a b a
j
a b a a b a
Không cần so sánh phần này
Phần phải thực hiện so sánh

So khớp mẫu (Pattern Matching)
The KMP Algorithm - Motivation

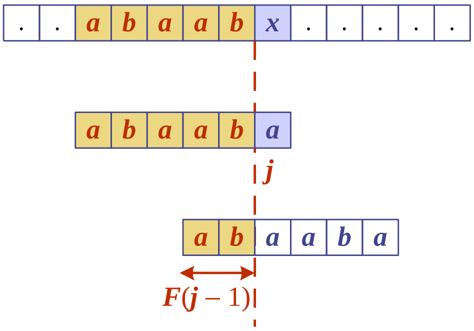
- Thuật toán Knuth-Morris-Pratt’s so sánh mẫu theo thứ tự **left-to-right**, nhưng dịch chuyển các mẫu thông minh hơn phương pháp brute-force.
- Khi có ký tự không giống nhau, ta có thể dịch vị trí để bắt đầu so sánh về trước nhiều nhất là bao nhiêu để tránh các so sánh không cần thiết?
- Answer: Tiền tố lớn nhất của ***P***[0..***j***] ***l à hậu tố của P***[1..***j***] (the largest prefix of ***P***[0..***j***] that is a suffix of ***P***[1..***j***])


. . a b a a b x . . . . .
a b a a b a
j
a b a a b a
Không cần so sánh phần này
Phần phải thực hiện so sánh

So khớp mẫu (Pattern Matching)
KMP Failure Function

- Thuật toán Knuth-Morris-Pratt thực hiện tiền xử lý trên mẫu để tìm các tiền tố (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



So khớp mẫu (Pattern Matching)
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 }
```

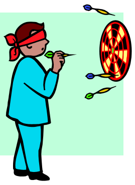
So khớp mẫu (Pattern Matching)
The KMP Algorithm

- 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 $O(m + n)$

```
char *txt = "ABABDABACDABABCABAB";
    char *pat = "ABABCABAB";
```

```
int KMPMatch(string T, string P)
{
    n= T.length();   m= P.length();
    failureFunction(P);
    int i =0, j=0;
    while (i < n)
        if (T[i]==P[j])
            if (j== m - 1) return i-j; //  co giong nhau
           else { i++; j++; }
        else
            if (j>0) j = F[j-1];
            else i = i + 1;
    return -1; //  ko co chuoi con giong nhau
}
```

So khớp mẫu (Pattern Matching)
Computing the Failure Function



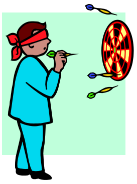
```
Algorithm failureFunction(P)
    $F[0] \leftarrow 0$
    $i \leftarrow 1$
    $j \leftarrow 0$
    while $i < m$
        if $P[i] = P[j]$
            {we have matched $j + 1$ chars}
            $F[i] \leftarrow j + 1$
            $i \leftarrow i + 1$
            $j \leftarrow j + 1$
        else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

```
void  failureFunction(string P)
{
   F[0]  =  0;
   int i   =1,  j=0;
   while  (i  <  m)
      if( P[i]  == P[j])
      {
         F[i]  =   j + 1;
        i  =  i  +  1;    j  =  j  +  1  ;
      }
      else if  ( j  >  0  )j  =  F[j  -  1]  ;
      else {  F[i]  =  0  ;  i   =i  +  1;}
}
```

<u>Tính F cho chuỗi P:</u>

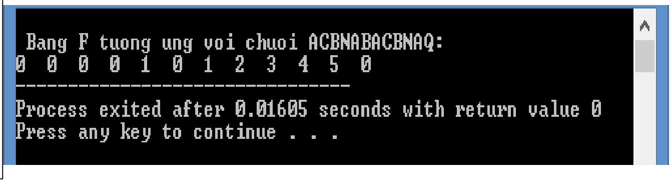
ACBNABACBNAQ

So khớp mẫu (Pattern Matching)
Computing the Failure Function



```
void  failureFunction(string P)
{
     F[0] = 0;
     int i  =1, j=0;
     while (i < m)
     if( P[i] == P[j])
     {
        F[i] =  j + 1;     i = i + 1;    j = j + 1 ;
     }
     else
          if ( j > 0 ) j = F[j - 1];
          else { F[i] = 0 ; i  =i + 1;}
     cout<<"\n Bang F ung voi chuoi "<<P<<":\n";
     for(i=0; i<m; i++)      cout<<F[i]<<"  ";
}
```

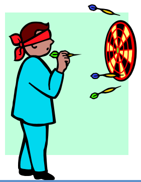
```
string T, P;
int F[100];
int m, n;
int main()
{
  P="ACBNABACBNAQ";
  n= T.length();   m= P.length();
  failureFunction(P);
}
```



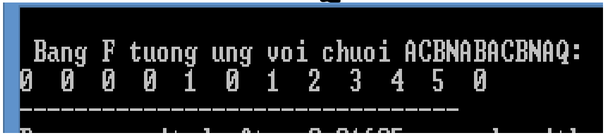
```
 Bang F tuong ung voi chuoi ACBNABACBNAQ:
0  0  0  0  1  0  1  2  3  4  5  0
--------------------------------
Process exited after 0.01605 seconds with return value 0
Press any key to continue . . .
```

13

So khớp mẫu (Pattern Matching)
Computing the Failure Function



"ACBNABACBNAQ"



```
 Bang F tuong ung voi chuoi ACBNABACBNAQ:
0  0  0  0  1  0  1  2  3  4  5  0
--------------------------------
```

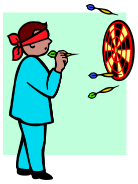
A	C	B	N	A	B	A	C	B	N	A	Q
0	0	0	0	1	0	1	2	3	4	5	0

Cho biết bảng F với chuỗi sau????

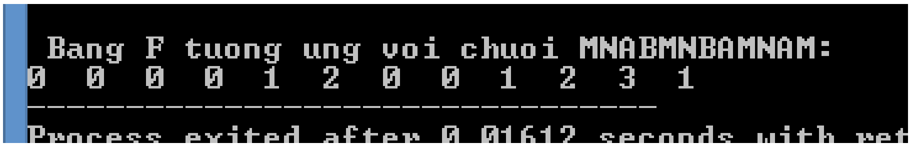
M	N	A	B	M	N	B	A	M	N	A	M

14

So khớp mẫu (Pattern Matching)
Computing the Failure Function

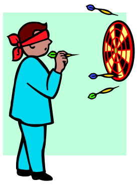


M	N	A	B	M	N	B	A	M	N	A	M



```
 Bang F tuong ung voi chuoi MNABMNBAMNAM:
0  0  0  0  1  2  0  0  1  2  3  1
--------------------------------
Process exited after 0.01612 seconds with ret
```

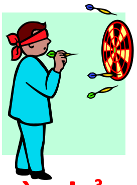
So khớp mẫu (Pattern Matching)
Computing the Failure Function



- 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)
    $F[0] \leftarrow 0$
    $i \leftarrow 1$
    $j \leftarrow 0$
    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 }
            $i \leftarrow i + 1$
```

So khớp mẫu (Pattern Matching)
Computing the Failure Function



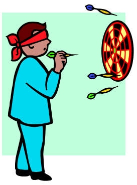
Thực hiện tính các bước so sánh cho hai chuỗi sau dựa vào bảng F???

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

17

So khớp mẫu (Pattern Matching)
Computing the Failure Function



Các bước thực hiện so khớp:

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														
					7														
				a	b	a	c	a	b										
					8	9	10	11	12										
					a	b	a	c	a	b									
									13										
									a	b	a	c	a	b					
										14	15	16	17	18	19				
										a	b	a	c	a	b				

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

18

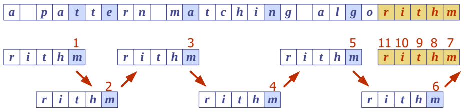
So khớp mẫu (Pattern Matching)
Boyer-Moore Heuristics

- The Boyer-Moore’s pattern matching algorithm is based on two heuristics

  Looking-glass heuristic (right-to-left matching): Compare $\boldsymbol{P}$ with a subsequence of $\boldsymbol{T}$ moving backwards

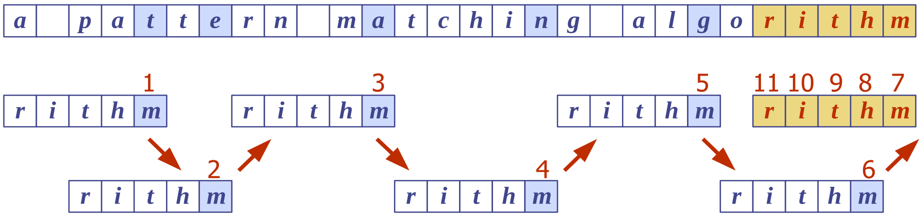
  Character-jump heuristic (bad character shift rule): When a mismatch occurs at $\boldsymbol{T}[\boldsymbol{i}] = \boldsymbol{c}$

  - If $\boldsymbol{P}$ contains $\boldsymbol{c}$, shift $\boldsymbol{P}$ to align the last occurrence of $\boldsymbol{c}$ in $\boldsymbol{P}$ with $\boldsymbol{T}[\boldsymbol{i}]$
  - Else, shift $\boldsymbol{P}$ to align $\boldsymbol{P}[0]$ with $\boldsymbol{T}[\boldsymbol{i} + 1]$

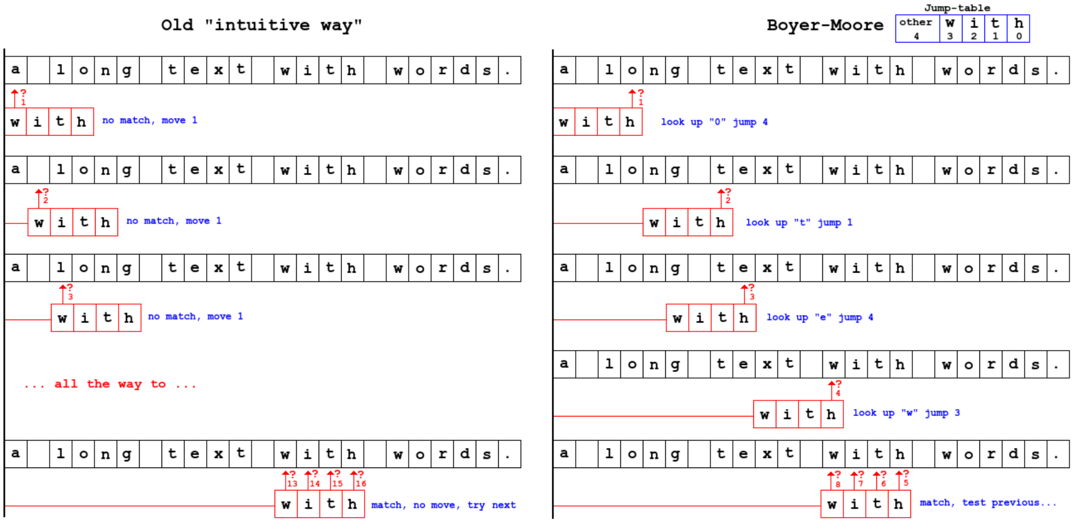


So khớp mẫu (Pattern Matching)
Boyer-Moore Heuristics

Example



So khớp mẫu (Pattern Matching)
Boyer-Moore Heuristics



So khớp mẫu (Pattern Matching)
Last-Occurrence Function

- Boyer-Moore's algorithm preprocesses the pattern $\boldsymbol{P}$ and the alphabet Σ to build the last-occurrence function $\boldsymbol{L}$ mapping Σ to integers, where $\boldsymbol{L}(\boldsymbol{c})$ is defined as
  - the largest index $\boldsymbol{i}$ such that $\boldsymbol{P}[\boldsymbol{i}] = \boldsymbol{c}$ or
  - -1 if no such index exists

Example:

- $\Sigma = \{a, b, c, d\}$
- $\boldsymbol{P} = \boldsymbol{abacab}$

c	a	b	c	d
$L(c)$	4	5	3	-1

So khớp mẫu (Pattern Matching)
Last-Occurrence Function

Example:

- $\Sigma = \{a, b, c, d\}$
- $P = abacab$

c	a	b	c	d
$L(c)$	4	5	3	–1

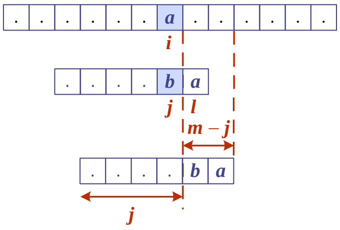
The last-occurrence function can be represented by an array indexed by the numeric codes of the characters

The last-occurrence function can be computed in time $O(m + s)$, where m is the size of P and s is the size of Σ

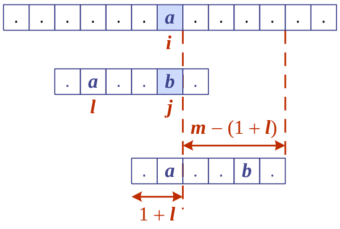
So khớp mẫu (Pattern Matching)
The Boyer-Moore Algorithm

```
Algorithm BoyerMooreMatch(T, P, Σ)
    $L \leftarrow lastOccurenceFunction(P, \Sigma)$
    $i \leftarrow m - 1$
    $j \leftarrow m - 1$
    repeat
        if $T[i] = P[j]$
            if $j = 0$
                return $i$  { match at $i$ }
            else
                $i \leftarrow i - 1$
                $j \leftarrow j - 1$
        else
            { character-jump }
            $l \leftarrow L[T[i]]$
            $i \leftarrow i + m - \min(j, 1 + l)$
            $j \leftarrow m - 1$
    until $i > n - 1$
    return $-1$ { no match }
```

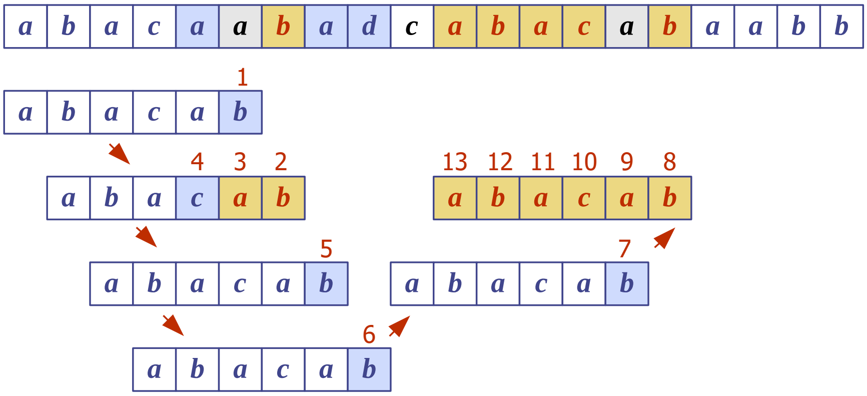
Case 1: $j \leq 1 + l$



Case 2: $1 + l \leq j$

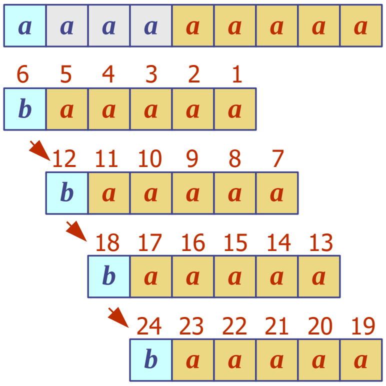


So khớp mẫu (Pattern Matching)
The Boyer-Moore Algorithm



So khớp mẫu (Pattern Matching) : Analysis

- Boyer-Moore’s algorithm runs in time $O(nm + s)$
- Example of worst case:
  - $T = aaa \ldots a$
  - $P = baaa$
- The worst case may occur in images and DNA sequences but is unlikely in English text
- Boyer-Moore’s algorithm is significantly faster than the brute-force algorithm on English text



Các nội dung cần đọc trước [review FC2]

- Danh sách liên kết (Đơn, Đôi, Vòng)
- Ngăn xếp (Stack)
- Hàng đợi (Queue)
- Cây (Tree)
- Bảng băm (Hash Table)

Tài liệu đọc thêm về các thuật toán so khớp chuỗi

http://www-igm.univ-mlv.fr/~lecroq/string/node22.html

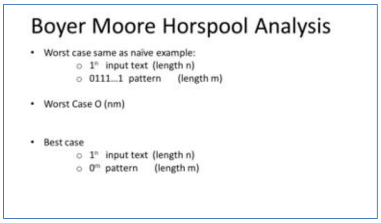
https://www.topcoder.com/community/competitive-programming/tutorials/introduction-to-string-searching-algorithms/

https://study.com/academy/lesson/string-searching-algorithms-methods-types.html

Link YouTube

https://www.youtube.com/watch?v=V5-7GzOfADQ

https://www.youtube.com/watch?v=PHXAOKQk2dw



Boyer Moore Horspool Analysis

- Worst case same as naive example:
  - 1^n input text (length n)
  - 0111…1 pattern (length m)
- Worst Case O (nm)
- Best case
  - 1^n input text (length n)
  - 0^m pattern (length m)

29