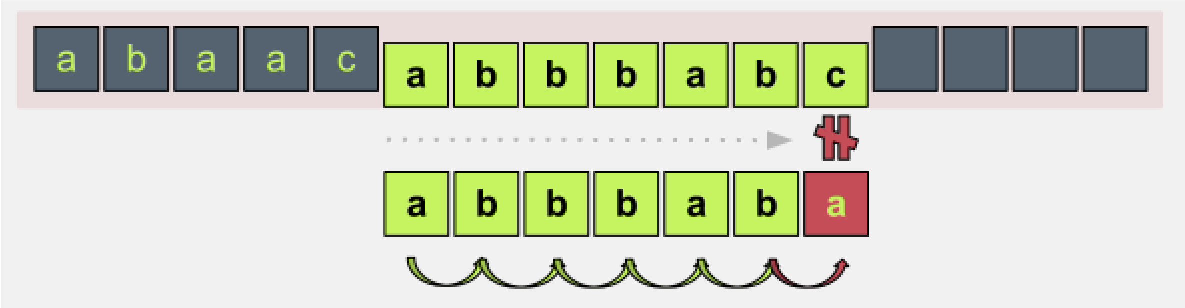


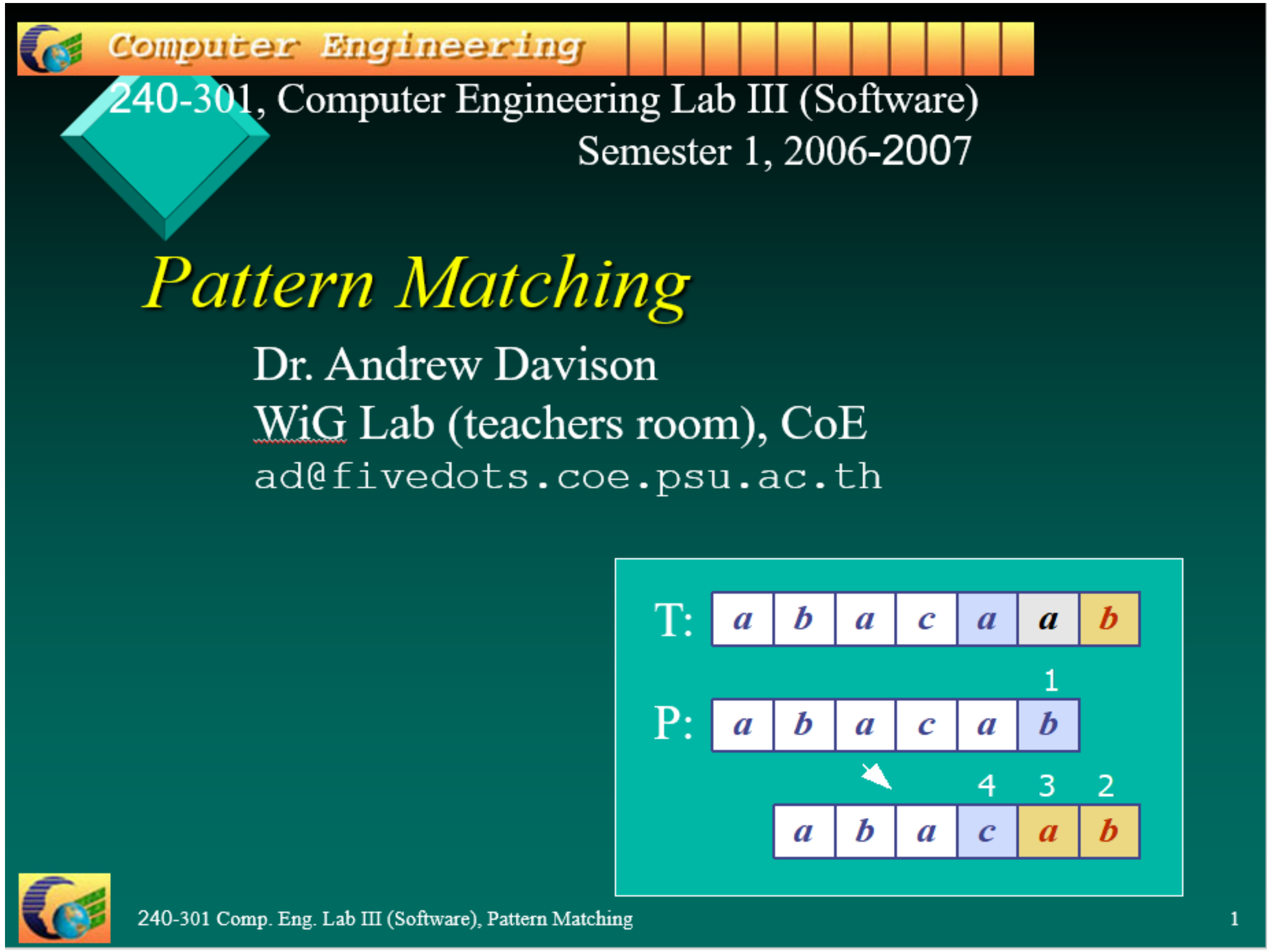
Pencocokan String (*String/Pattern Matching*)

Bahan Kuliah IF2211 Strategi Algoritma

Program Studi Teknik Informatika
STEI-ITB

Referensi untuk slide ini diambil dari:

Dr. Andrew Davison, *Pattern Matching*, WiG Lab (teachers room), CoE (*Updated by*: Dr. Rinaldi Munir, Informatika STEI-ITB)



Overview

1. What is Pattern Matching?
2. The Brute Force Algorithm
3. The Knuth-Morris-Pratt Algorithm
4. The Boyer-Moore Algorithm
5. More Information

1. What is Pattern Matching?

- Definisi: Diberikan:

  1. *T*: teks (*text*), yaitu (*long*) *string* yang panjangnya *n* karakter

  2. *P: pattern*, yaitu *string* dengan panjang *m* karakter (asumsi $m <<< n$) yang akan dicari di dalam teks.

  Carilah (*find* atau *locate*) lokasi pertama di dalam teks yang bersesuaian dengan *pattern*.

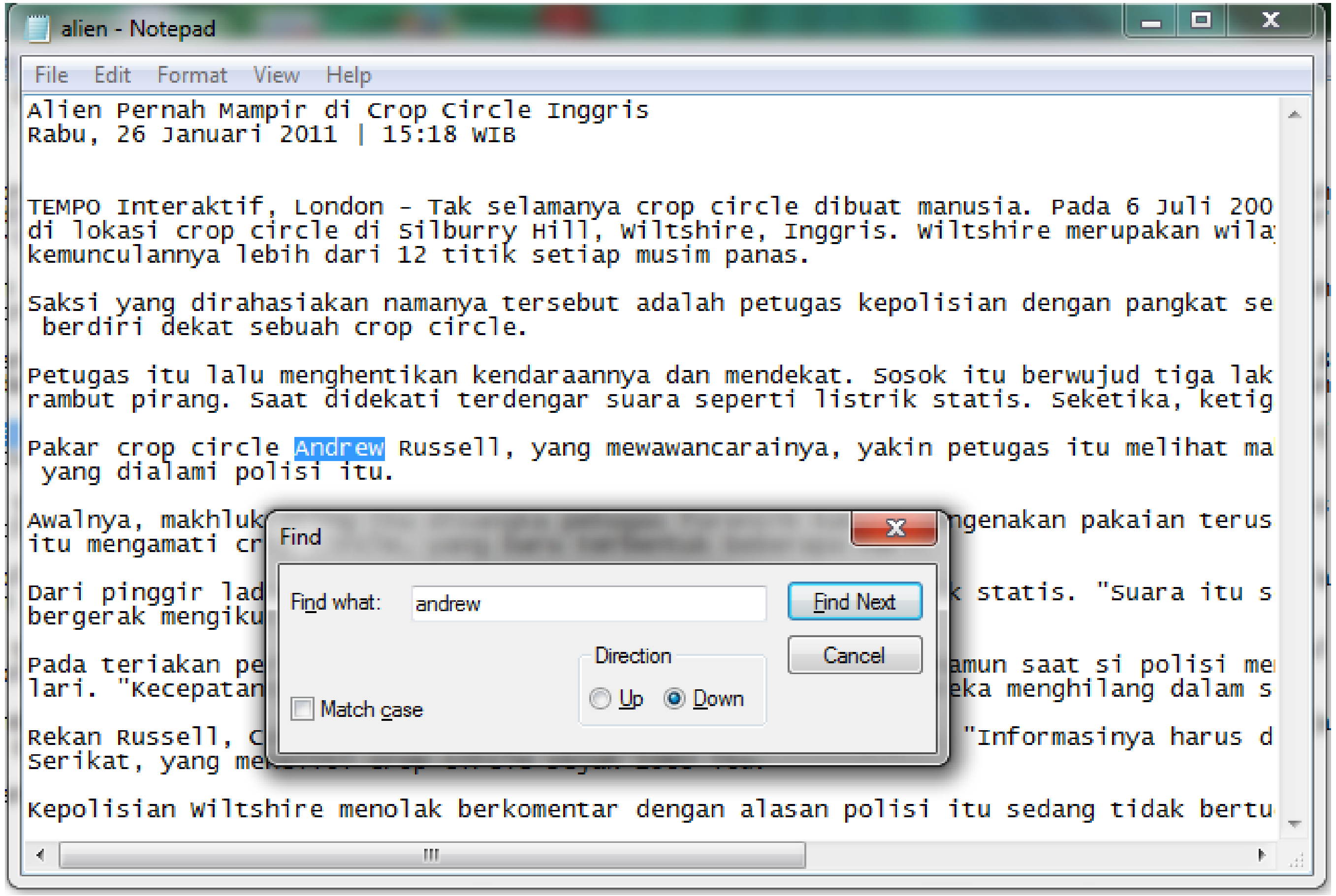
- Contoh:

  T: `the rain in spain stays mainly on the plain`

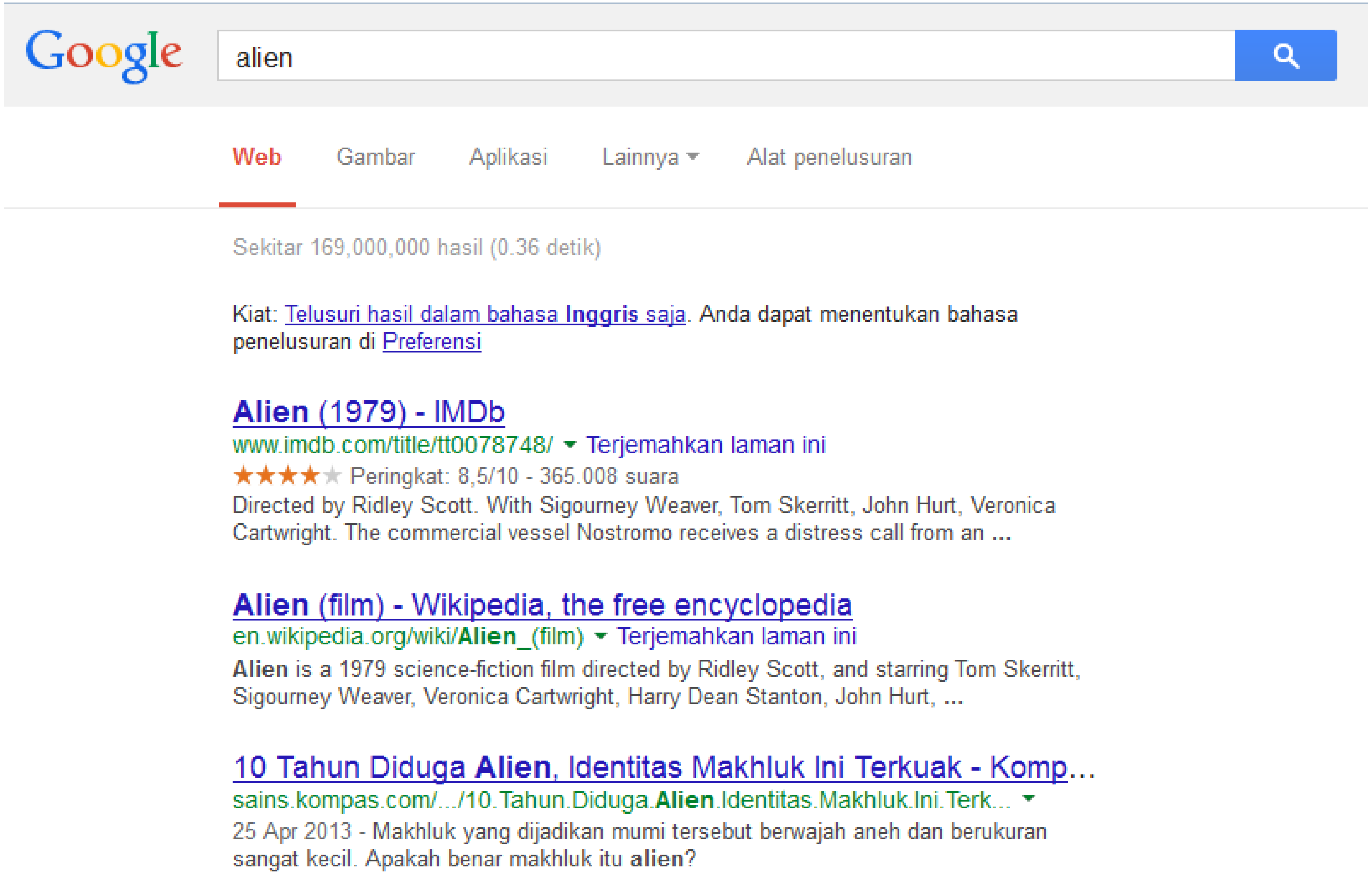
  P: `main`

➢Aplikasi:

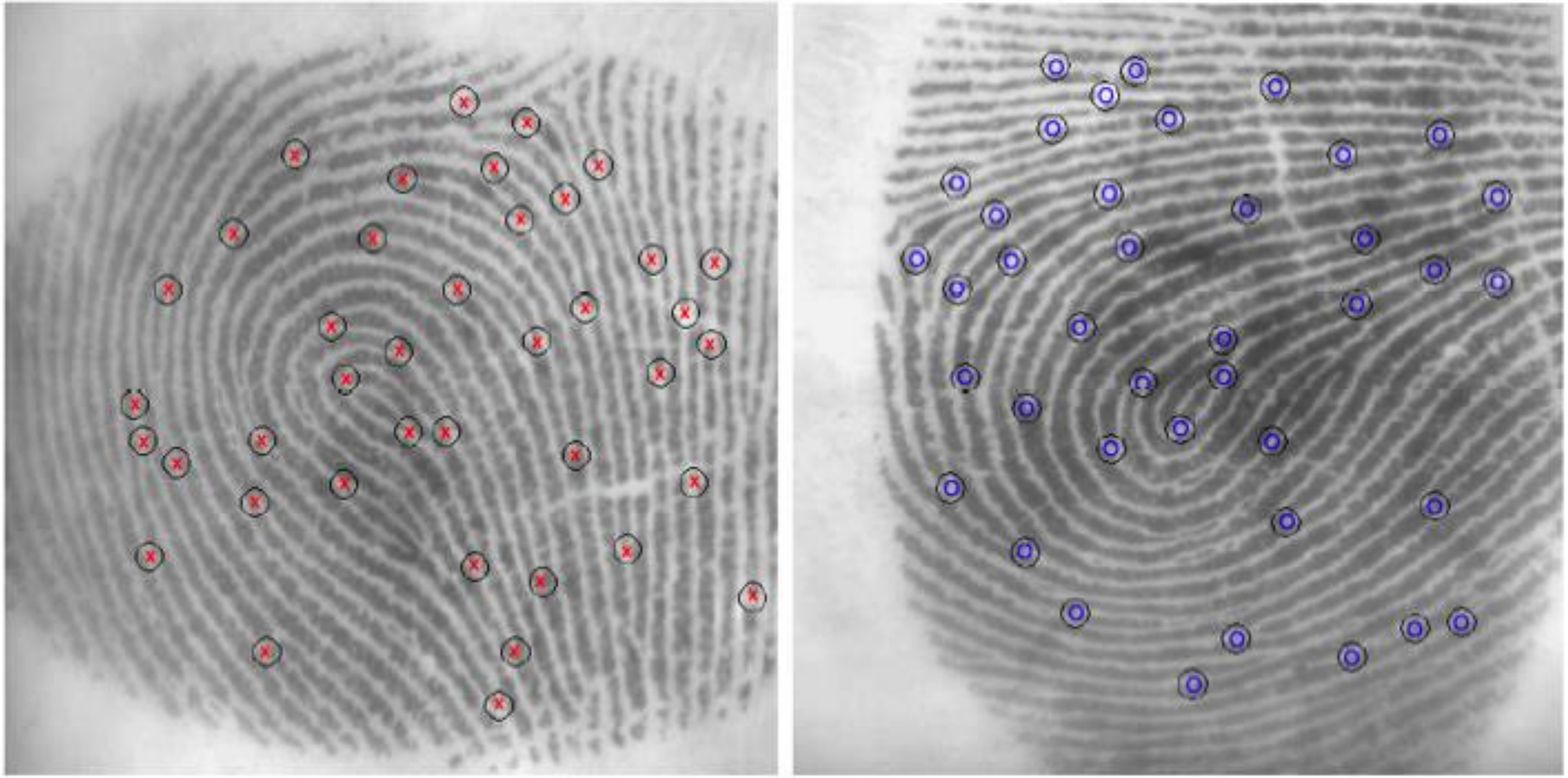
1. Pencarian di dalam Editor Text



2. *Web search engine* (Misal: Google)

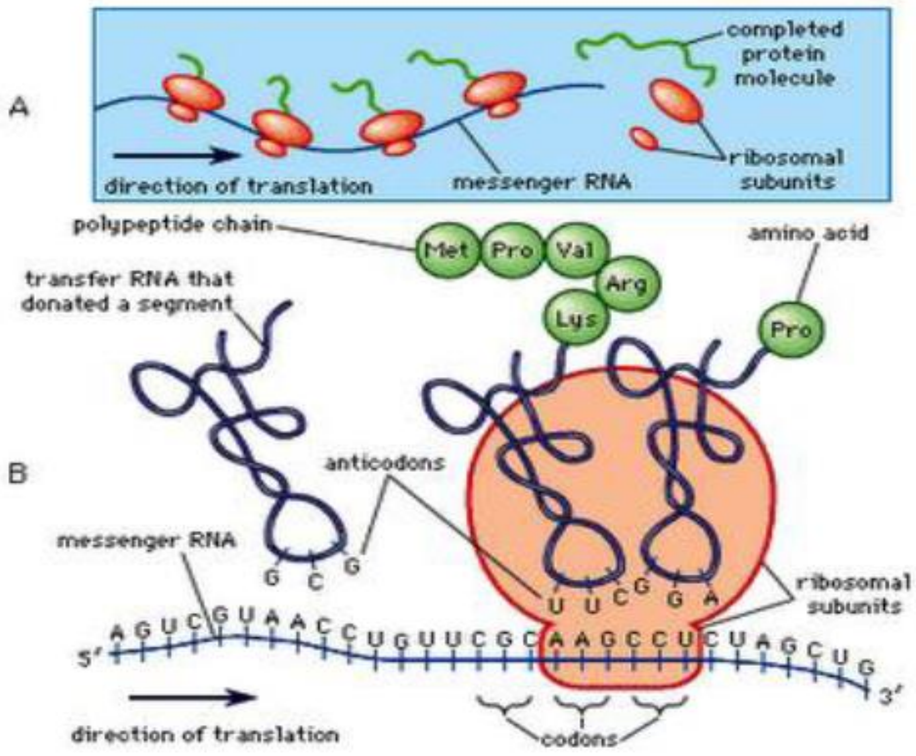


3. Analisis Citra



4. *Bionformatics*

- Pencocokan Rantai Asam Amino pada rantai DNA



© 2006 Encyclopædia Britannica, Inc.

Gambar 4. Translasi mRNA menjadi tRNA yang kemudian menjadi rantai protein

```
C:\Users\Septu\Desktop>g++ -o b bf.cpp

C:\Users\Septu\Desktop>b
Masukkan nama file tempat rantai DNA disimpan = t
Masukkan pattern = CGAUCGAUGCUAGUCGAUCGUAGCUAGCUA
rantai DNA yang ingin diperiksa = ACGATGCTAGCTAGC
CTAGCTAGCTGATCGATCGATCGATCGTACGTCAGTCGATCGATCGATG
GCATCGTGATGCGCGCTAGCTAGCTAGCATGCTAGCTAGCTGATCGATC
GATCGATCGATCGATCGATCGATGCTAGCTAGCTATAATCGATCGGCCG
ATGCAGTCAGTCAGTCAGTAGCTGATCTCTGCAGCGCATCGTAGCTAGT
CTAGCTGATCGTAGCTAGTCAGACTGCTAGTCGATCGATGCATGCATGC
TAGCTAGCTAGCTAGCTAGCTAGCTAGACTACGTAGCTAGCTAGCTAGC
CTAGCTAGCTAGTCGATCGATGCTAGCTAGTCAGTCAGTCGATCGATCG
CTACGTCAGTACATCATCTAGGCAGCAGCATGCTGTAGCCTAGCTAGCT
CGATCGATGCATGCTATAGCGCGCGAGTCGTAGCTAGCACACGATGCTA
TCGATGCTAGCTAGCTGATCGATCGATCGATCGTACGTCAGTCGATCGA
GTATATGCATCGTGATGCGCGCTAGCTAGCTAGCATGCTAGCTAGCTGA
TCGATCGATCGATCGATCGATCGATCGATGCTAGCTAGCTATAATCGAT
GCATGCATGCAGTCAGTCAGTCAGTAGCTGATCTCTGCAGCGCATCGTA
TGCATGCTAGCTGATCGTAGCTAGTCAGACTGCTAGTCGATCGATGCAT
GCTAGCTAGCTAGCTAGCTAGCTAGCTAGACTACGTAGCTAGCTA
TGCTAGCTAGCTAGCTAGTCGATCGATGCTAGCTAGTCAGTCAGTCGAT
GCATGACTACGTCAGTACATCATCTAGGCAGCAGCATGCTGTAGCCTAG
TTCGATCGATCGATGCATGCTATAGCGCGCGAGTCGTAGCTAGCACACG
GATCGATCGATGCTAGCTAGCTGATCGATCGATCGATCGTACGTCAGTC
TAGCTAGTATATGCATCGTGATGCGCGCTAGCTAGCTAGCATGCTAGCT
AGCTAGTCGATCGATCGATCGATCGATCGATCGATGCTAGCTAGCTATA
GCGTCAGCATGCATGCAGTCAGTCAGTCAGTAGCTGATCTCTGCAGCGC
GATGCATGCATGCTAGCTGATCGTAGCTAGTCAGACTGCTAGTCGATCG
ATCGTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGACTACGTAGC
CTAGCATGCTAGCTAGCTAGCTAGTCGATCGATGCTAGCTAGTCAGTCA
ACGACTGCATGACTACGTCAGTACATCATCTAGGCAGCAGCATGCTGTA
ATCGTCTTCGATCGATCGATGCATGCTATAGCGCGCGAGTCGTAGCTAG
CTAGTCGATCGATCGATGCTAGCTAGCTGATCGATCGATCGATCGTACG
GCATGCTAGCTAGTATATGCATCGTGATGCGCGCTAGCTAGCTAGCATG
TCAGTCAGCTAGTCGATCGATCGATCGATCGATCGATCGATGCTAGCTA
ACGTCAGCGTCAGCATGCATGCAGTCAGTCAGTCAGTAGCTGATCTCTG
TCGATCGATGCATGCATGCTAGCTGATCGTAGCTAGTCAGACTGCTAGT
CGATCGATCGTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGACTA
AGCTAGCTAGCATGCTAGCTAGCTAGCTAGTCGATCGATGCTAGCTAGT
GTACGTACGACTGCATGACTACGTCAGTACATCATCTAGGCAGCAGCAT
CGATCGATCGTCTTCGATCGATCGATGCATGCTATAGCGCGCGAGTCGT
AGCTAGCTAGTCGATCGATCGATGCTAGCTAGCTGATCGATCGATCGAT
ATCGATGCATGCTAGCTAGTATATGCATCGTGATGCGCGCTAGCTAGCT
AGCTAGTCAGTCAGCTAGTCGATCGATCGATCGATCGATCGATCGATGC
GCAGCTACGTCAGCGTCAGCATGCATGCAGTCAGTCAGTCAGTAGCTGA
AGCTAGTCGATCGATGCATGCATGCTAGCTGATCGTAGCTAGTCAGACT
GTCGATCGATCGATCGTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCT
CTAGCTAGCTAGCTAGCATGCTAGCTAGCTAGCTAGTCGATCGATGCTA
TAGTCAGTACGTACGACTGCATGACTACGTCAGTACATCATCTAGGCAG
ATCGATCGATCGATCGTCTTCGATCGATCGATGCATGCTATAGCGCGCG
rantai kode protein yang ingin dicari = CGAUCGAUG
rantai kode protein ditemukan pada = 13531

Lama operasi = 1954 microsecond

C:\Users\Septu\Desktop>_
```

Sumber: Septu Jamasoka, IF2009

String Concepts

- Assume S is a string of size m.

  $$S = x_0x_1 \ldots x_{m-1}$$

- A *prefix* of S is a substring S[0 .. k]
- A *suffix* of S is a substring S[k .. $m - 1$]
  - k is any index between 0 and $m - 1$

Examples

S

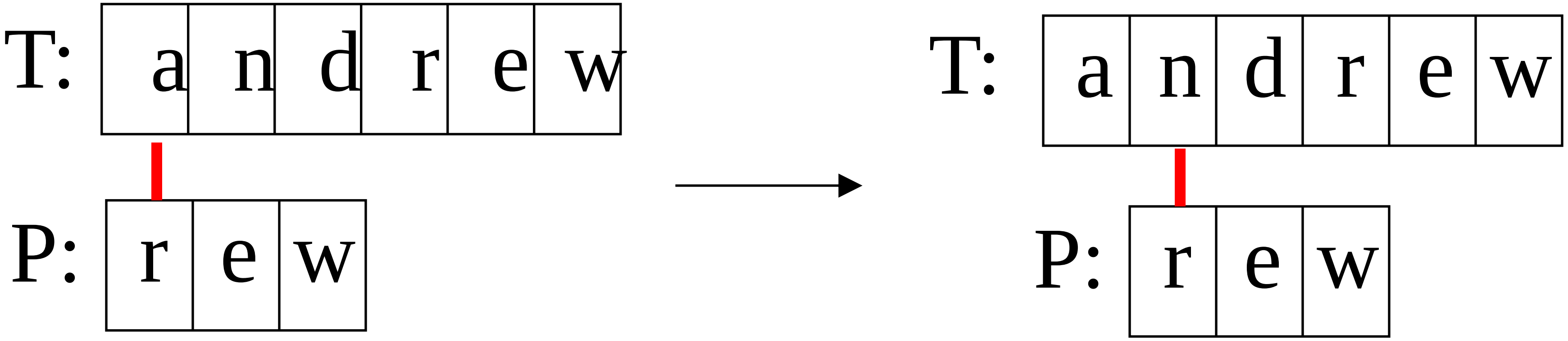
a	n	d	r	e	w
0					5

- All possible prefixes of S:
  - “a", "an", "and", "andr”, "andre“, "andrew“

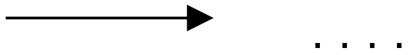
- All possible suffixes of S:
  - ”w”, “ew", “rew", “drew", “ndrew” , "andrew“

2. *The Brute Force Algorithm*

- Check each position in the text T to see if the pattern P starts in that position



P moves 1 char at a time through T



Teks: `NOBODY NOTICED HIM`

Pattern: `NOT`

```
  NOBODY NOTICED HIM
1 NOT
2  NOT
3   NOT
4    NOT
5     NOT
6      NOT
7       NOT
8        NOT
```

Brute Force in Java

Return index where pattern starts, or -1

```
public static int brute(String text,String pattern)
{ int n = text.length();    // n is length of text
  int m = pattern.length(); // m is length of pattern
  int j;
  for(int i=0; i <= (n-m); i++) {
    j = 0;
    while ((j < m) && (text.charAt(i+j)== pattern.charAt(j)))
    {
        j++;
      }
    if (j == m)
        return i;    // match at i
  }
  return -1;    // no match
} // end of brute()
```

Usage

```
public static void main(String args[])
{ if (args.length != 2) {
    System.out.println("Usage: java BruteSearch
                          <text> <pattern>");
    System.exit(0);
  }
  System.out.println("Text: " + args[0]);
  System.out.println("Pattern: " + args[1]);

  int posn = brute(args[0], args[1]);
  if (posn == -1)
    System.out.println("Pattern not found");
  else
    System.out.println("Pattern starts at posn " +  posn);
}
```

Analysis

Worst Case.

- Jumlah perbandingan: $m(n - m + 1) = O(mn)$

- Contoh:
  - T: `aaaaaaaaaaaaaaaaaaaaaaaaaah`
  - P: `aaah`

Best case

- Kompleksitas kasus terbaik adalah *O*(*n*).

- Terjadi bila karakter pertama *pattern P* tidak pernah sama dengan karakter teks *T* yang dicocokkan

- Jumlah perbandingan maksimal *n* kali:
- Contoh:

T: `String ini berakhir dengan zzz`

P: `zzz`

Average Case

- But most searches of ordinary text take O(m+n), which is very quick.

- Example of a more average case:
  - T: `a string searching example is standard`
  - P: `store`

- The brute force algorithm is fast when the alphabet of the text is large
  - e.g. A..Z, a..z, 1..9, etc.

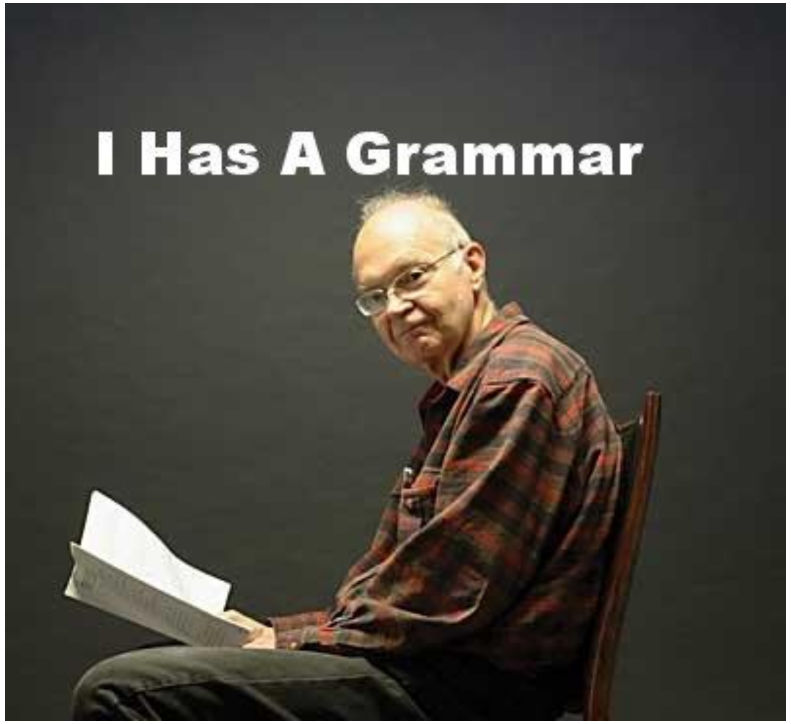
- It is slower when the alphabet is small
  - e.g. 0, 1 (as in binary files, image files, etc.)

2. *The KMP Algorithm*

- The Knuth-Morris-Pratt (KMP) algorithm looks for the pattern in the text in a *left-to-right* order (like the brute force algorithm).

- But it shifts the pattern more intelligently than the brute force algorithm.

Donald E. Knuth

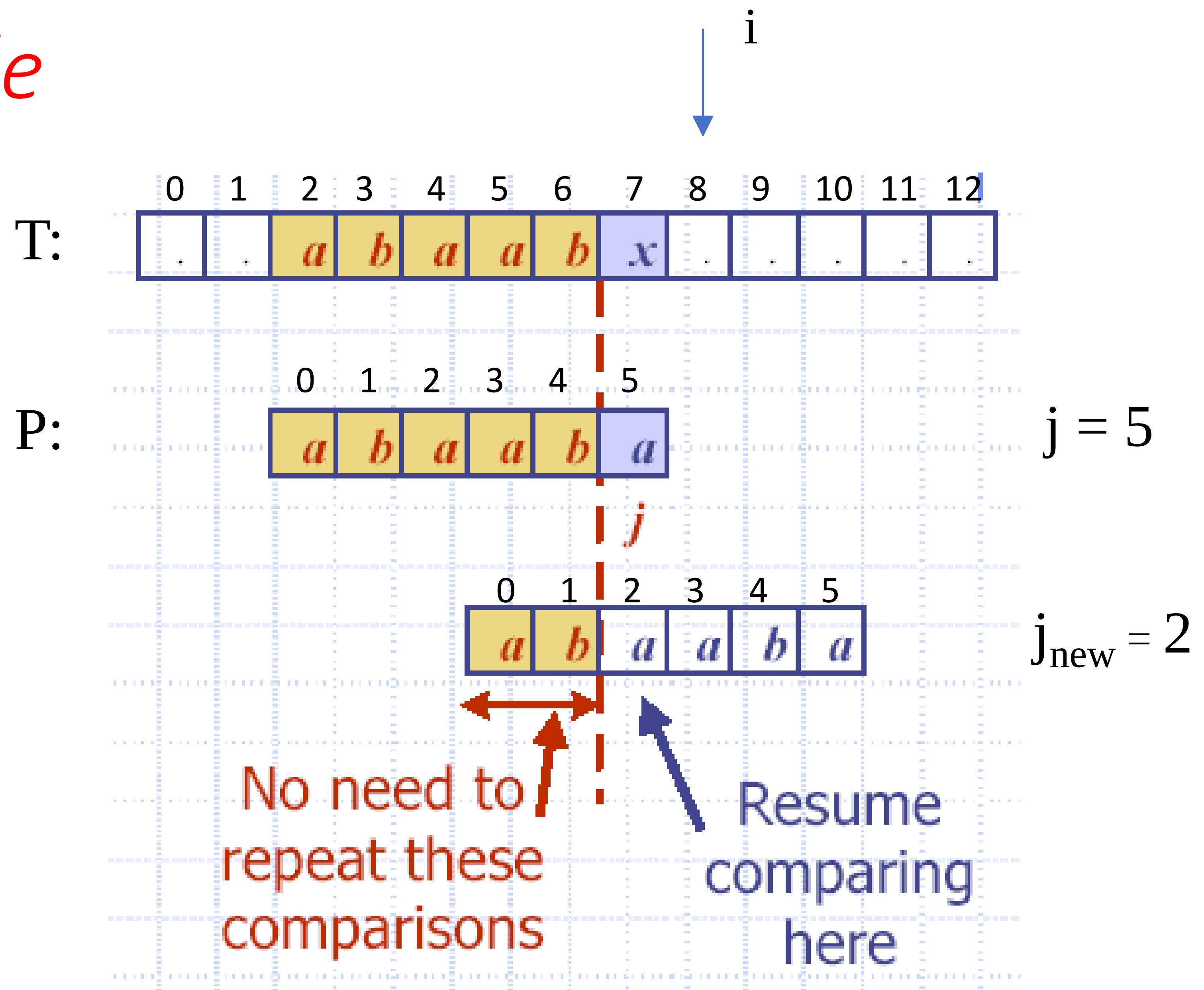


Donald Ervin Knuth (born January 10, 1938) is a computer scientist and Professor Emeritus at Stanford University. He is the author of the seminal multi-volume work *The Art of Computer Programming*.[3] Knuth has been called the "father" of the analysis of algorithms. He contributed to the development of the rigorous analysis of the computational complexity of algorithms and systematized formal mathematical techniques for it. In the process he also popularized the asymptotic notation.

- If a mismatch occurs between the text and pattern P at P[j], i.e T[i] $\neq$ P[j], what is the *most* we can shift the pattern to avoid *wasteful comparisons*?

- *Answer*: the largest prefix of P[0 .. j-1] that is a suffix of P[1 .. j-1]

Example



i
0 1 2 3 4 5 6 7 8 9 10 11 12
T:
. . a b a a b x . . . . .
0 1 2 3 4 5
P:
a b a a b a
j
j = 5
0 1 2 3 4 5
a b a a b a
jnew = 2
No need to repeat these comparisons
Resume comparing here

Why

- Find largest prefix (start) of:

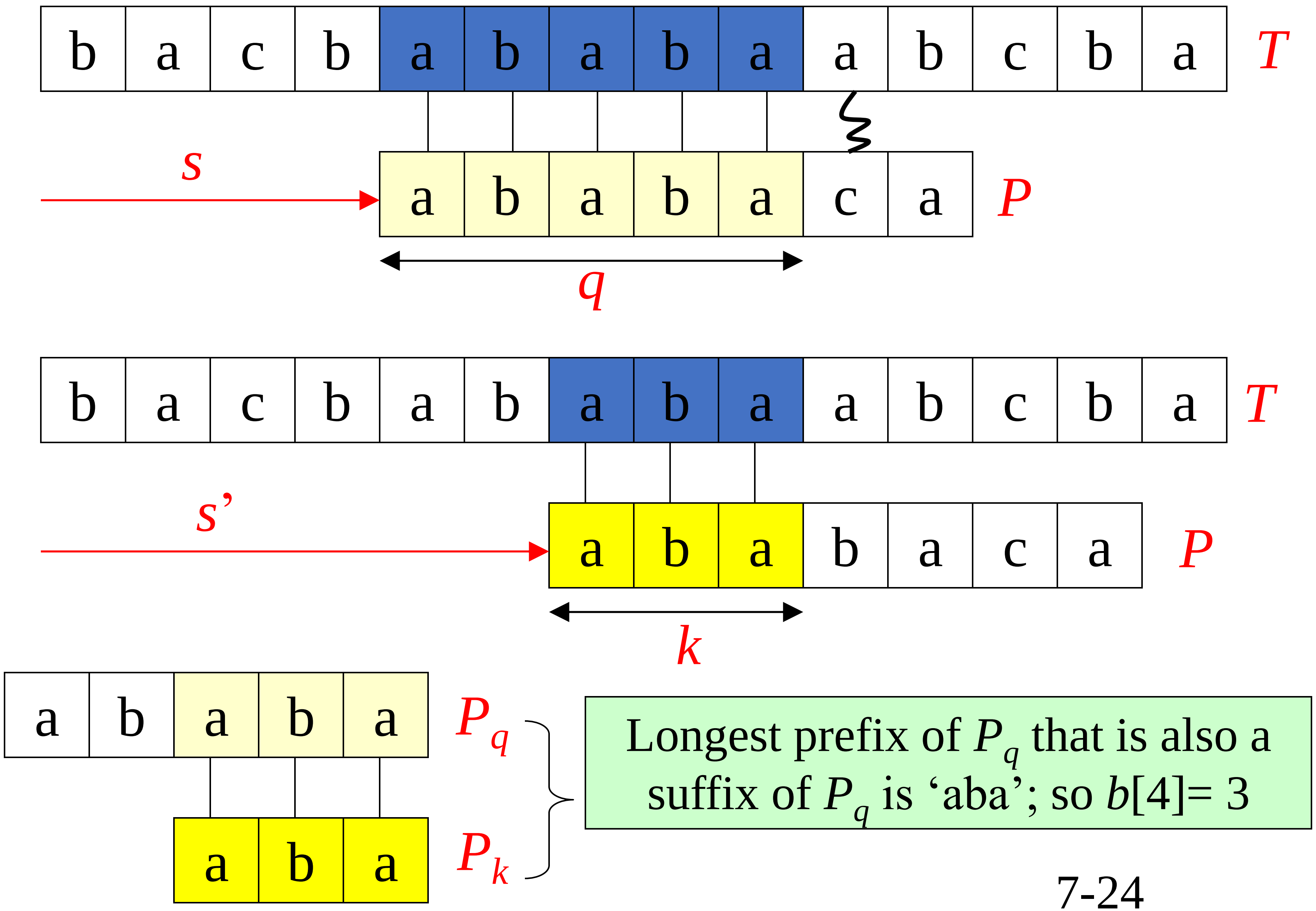
  abaab (P[0..4])

  which is suffix (end) of:

  abaab (P[1.. 4])
- Answer: ab → panjang = 2
- Set *j* = 2 // the new j value to begin comparison
- Jumlah pergeseran:

  s = length(abbab) – length(ab)

  = 5 – 2 = 3



Longest prefix of P_q that is also a suffix of P_q is 'aba'; so $b[4]= 3$

Fungsi Pinggiran KMP (KMP Border Function)

- KMP preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself.

- j = mismatch position in $P[]$
- k = position before the mismatch ($k = j - 1$).

- The *border function* $b(k)$ is defined as the *size* of the largest prefix of P[0..k] that is also a suffix of P[1..k].
- The other name: *failure function* (disingkat: *fail*)

Border Function Example

(k = j-1)

- P: abaaba

  j: 012345

j	0	1	2	3	4	5
P[j]	a	b	a	a	b	a
k	–	0	1	2	3	4
b(k)	–	0	0	1	1	2

b(k) is the size of the largest border.

- In code, *b*() is represented by an array, like the table.

Why is b(4) == 2?

P: "abaaba"

- *b*(4) means
  - find the size of the largest prefix of P[0..4] that is also a suffix of P[1..4]
  -
  - find the size largest prefix of "abaab" that is also a suffix of "baab“
  - find the size of "ab"

  == 2

- Contoh lain: P = `ababababca`

  j = `0123456789`

(k = j-1)

j	0	1	2	3	4	5	6	7	8	9
P [*j*]	**a**	**b**	**a**	**b**	**a**	**b**	**a**	**b**	**c**	**a**
k	-	0	1	2	3	4	5	6	7	8
b[*k*]	-	0	0	1	2	3	4	5	6	0

Using the Border Function

- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm.
  - if a mismatch occurs at P[j] (i.e. P[j] != T[i]), then

```
k = j-1;
j = b(k);     // obtain the new j
```

KMP in Java

Return index where pattern starts, or -1

```
public static int kmpMatch(String text,
                           String pattern)
  {
    int n = text.length();
    int m = pattern.length();

    int fail[] = computeFail(pattern);

    int i=0;
    int j=0;
   :
```

```
while (i < n) {
     if (pattern.charAt(j) == text.charAt(i)) {
       if (j == m - 1)
         return i - m + 1; // match
       i++;
       j++;
     }
     else if (j > 0)
      j = fail[j-1];
     else
       i++;
   }
   return -1; // no match
 } // end of kmpMatch()
```

```
public static int[] computeFail(String pattern)
  {
    int fail[] = new int[pattern.length()];
    fail[0] = 0;

    int m = pattern.length();
    int j = 0;
    int i = 1;
  :
```

```
while (i < m) {
     if  (pattern.charAt(j) ==
          pattern.charAt(i)) {   //j+1 chars match
       fail[i] = j + 1;
       i++;
       j++;
     }
     else if (j > 0) // j follows matching prefix
       j = fail[j-1];
     else {     // no match
       fail[i] = 0;
       i++;
     }
   }
   return fail;
 } // end of computeFail()
```

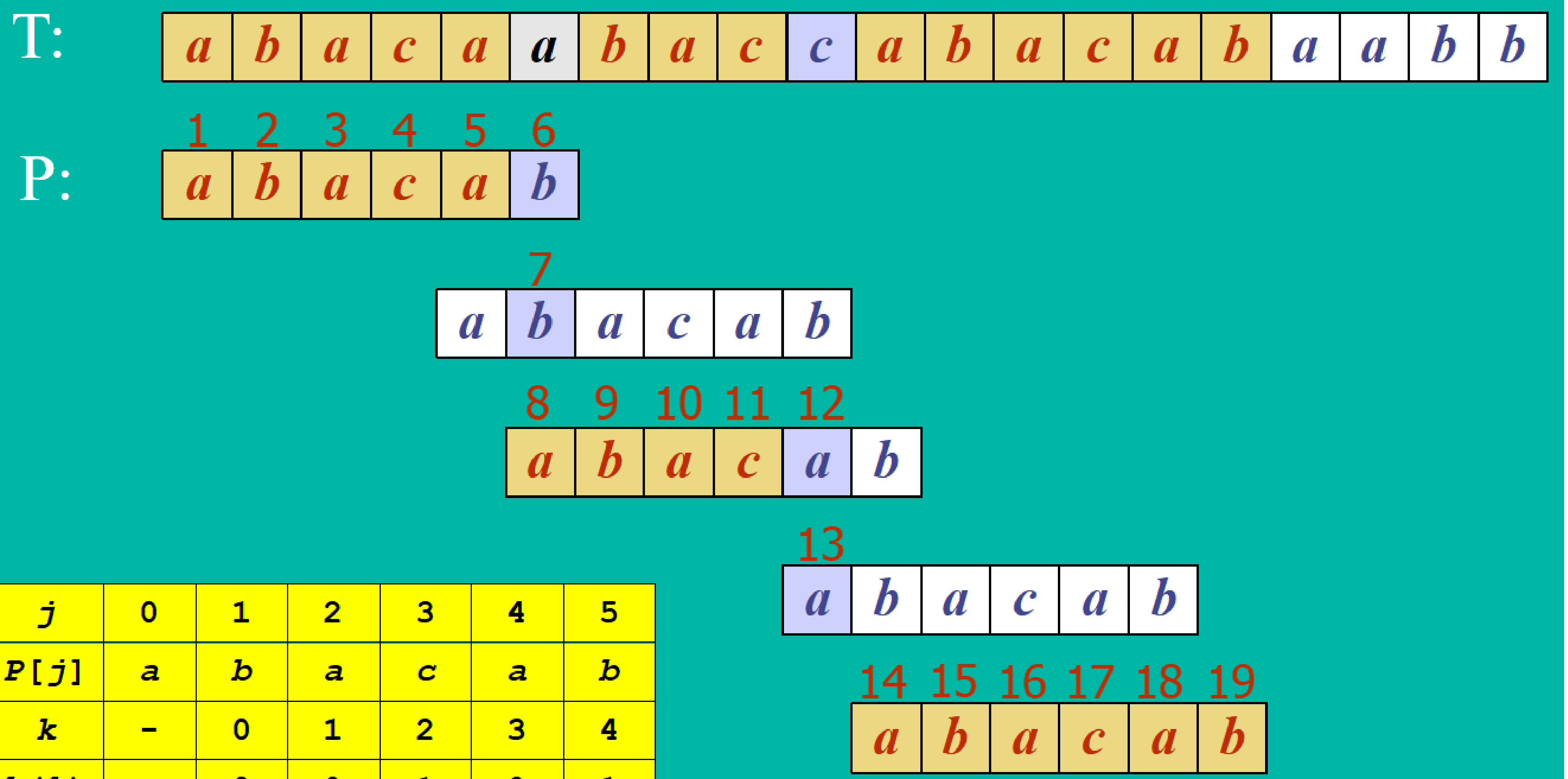
Similar code to kmpMatch()

Usage

```
public static void main(String args[])
  { if (args.length != 2) {
      System.out.println("Usage: java KmpSearch
                               <text> <pattern>");
      System.exit(0);
    }
    System.out.println("Text: " + args[0]);
    System.out.println("Pattern: " + args[1]);

    int posn = kmpMatch(args[0], args[1]);
    if (posn == -1)
      System.out.println("Pattern not found");
    else
      System.out.println("Pattern starts at posn "
                                          + posn);
}
```

Example



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

Jumlah perbandingan karakter: 19 kali

Why is b(4) == 1?

P: "abacab"

- b(4) means
  - find the size of the largest prefix of P[0..4] that is also a suffix of P[1..4]

  = find the size largest prefix of "abaca" that is also a suffix of "baca”

  = find the size of "a“

  = 1

Kompleksitas Waktu KMP

- Menghitung fungsi pinggiran : $O(m)$,
- Pencarian *string* : $O(n)$
- Kompleksitas waktu algoritma KMP adalah $O(m+n)$.
  - sangat cepat dibandingkan *brute force*

KMP Advantages

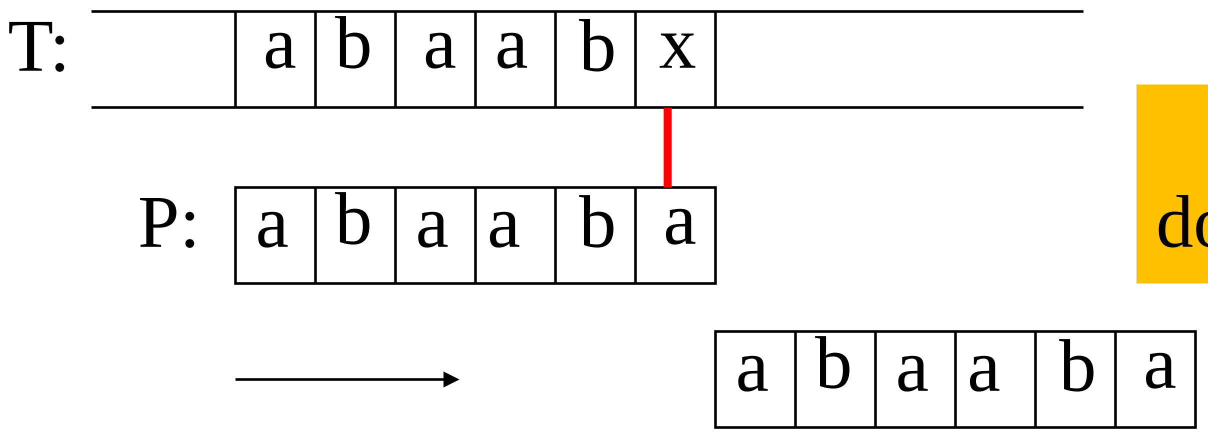
- The algorithm never needs to move backwards in the input text, T
  - this makes the algorithm good for processing very large files that are read in from external devices or through a network stream

KMP Disadvantages

- KMP doesn't work so well as the size of the alphabet increases
  - more chance of a mismatch (more possible mismatches)
  - mismatches tend to occur early in the pattern, but KMP is faster when the mismatches occur later

KMP Extensions

- The basic algorithm doesn't take into account the letter in the text that caused the mismatch.



Basic KMP does **not** do this.

Latihan

Diberikan sebuah *text*: `abacaabacabacababa` dan *pattern*: `acabaca`

a) Hitung fungsi pinggiran
b) Gambarkan proses pencocokan *string* dengan algoritma KMP sampai *pattern* ditemukan
c) Berapa jumlah perbandingan karakter yang terjadi?

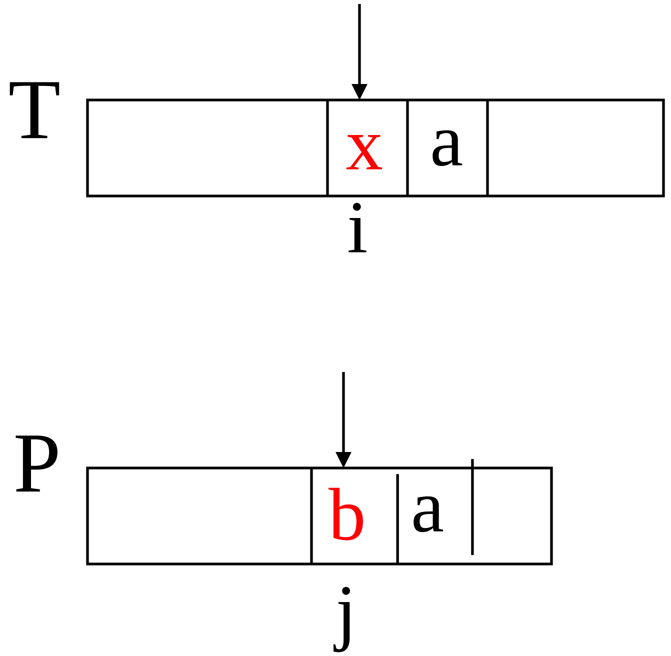
3. The Boyer-Moore Algorithm

- The Boyer-Moore pattern matching algorithm is based on two techniques.

- 1. The *looking-glass* technique
  - find P in T by moving *backwards* through P, starting at its end

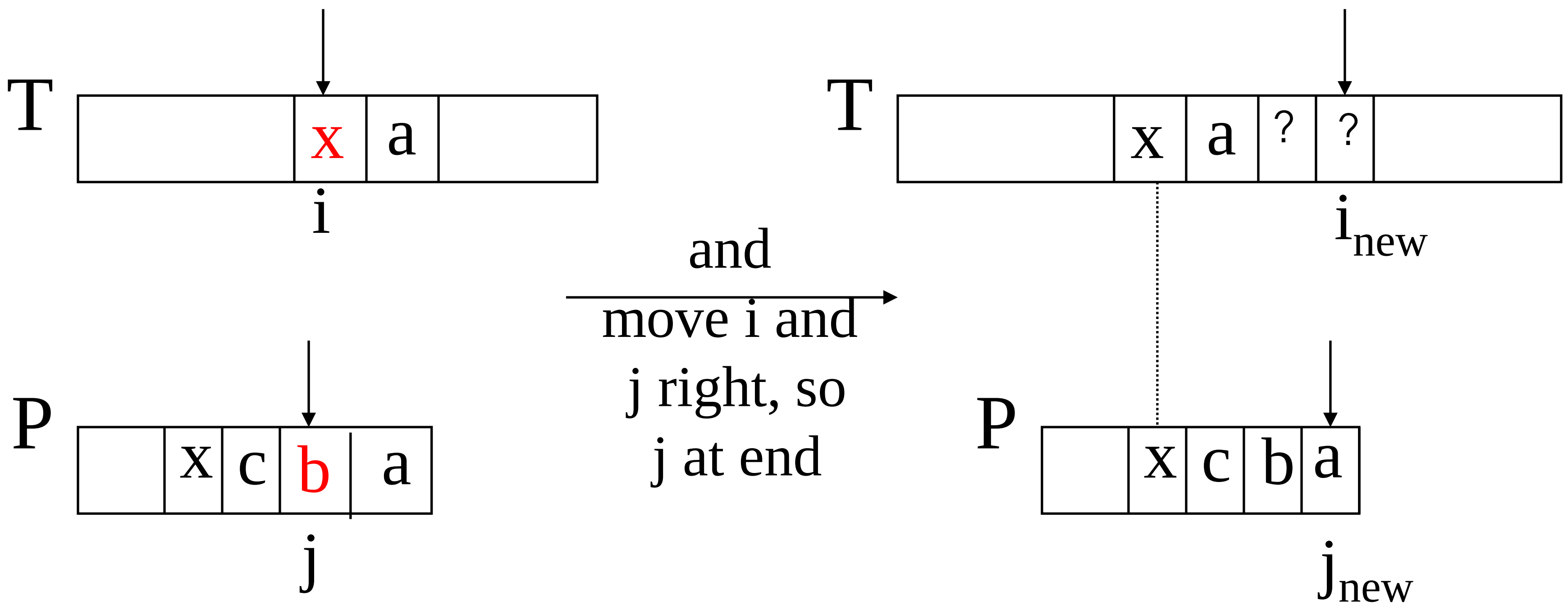
- 2. The *character-jump* technique
  - when a mismatch occurs at T[i] == x
  - the character in pattern P[j] is not the same as T[i]

- There are 3 possible cases, tried in order.



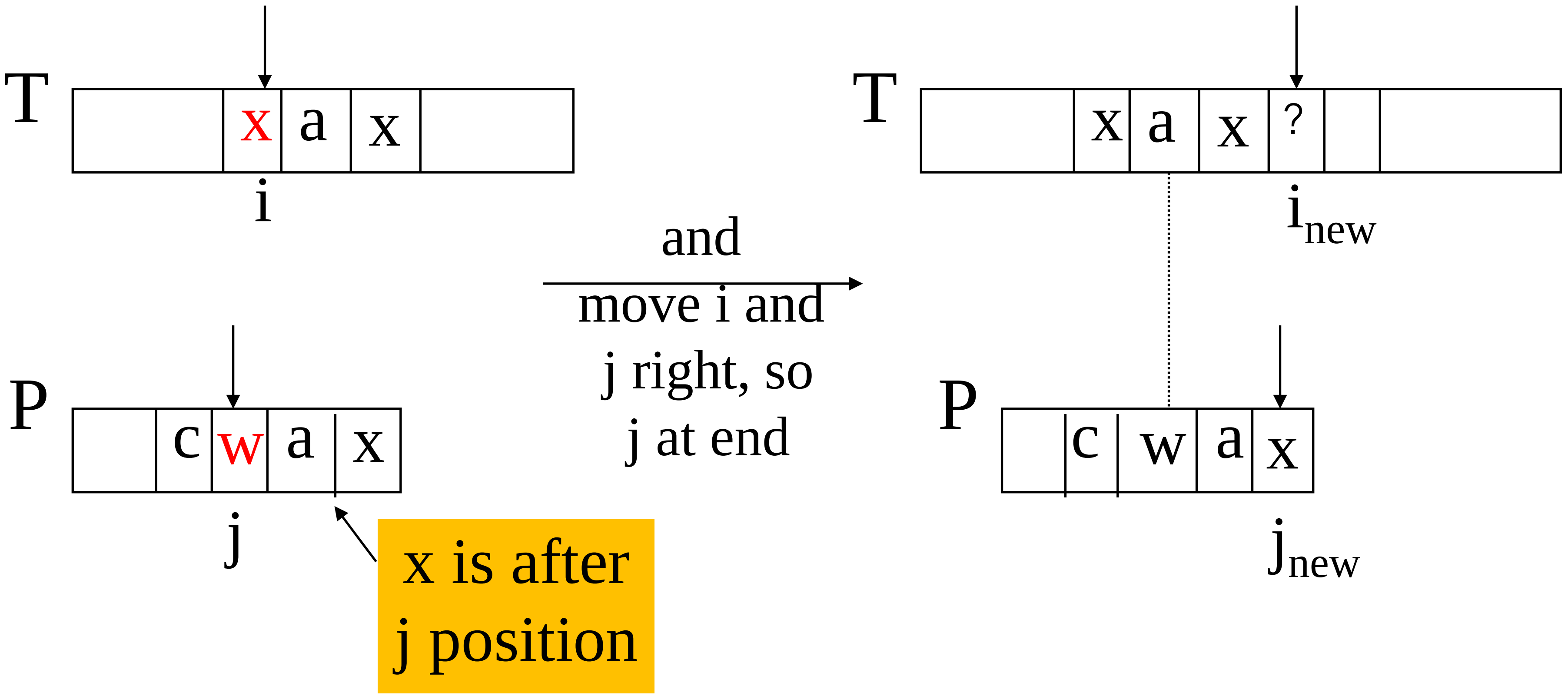
Case 1

- If P contains x somewhere, then try to *shift P* right to align the last occurrence of x in P with T[i].



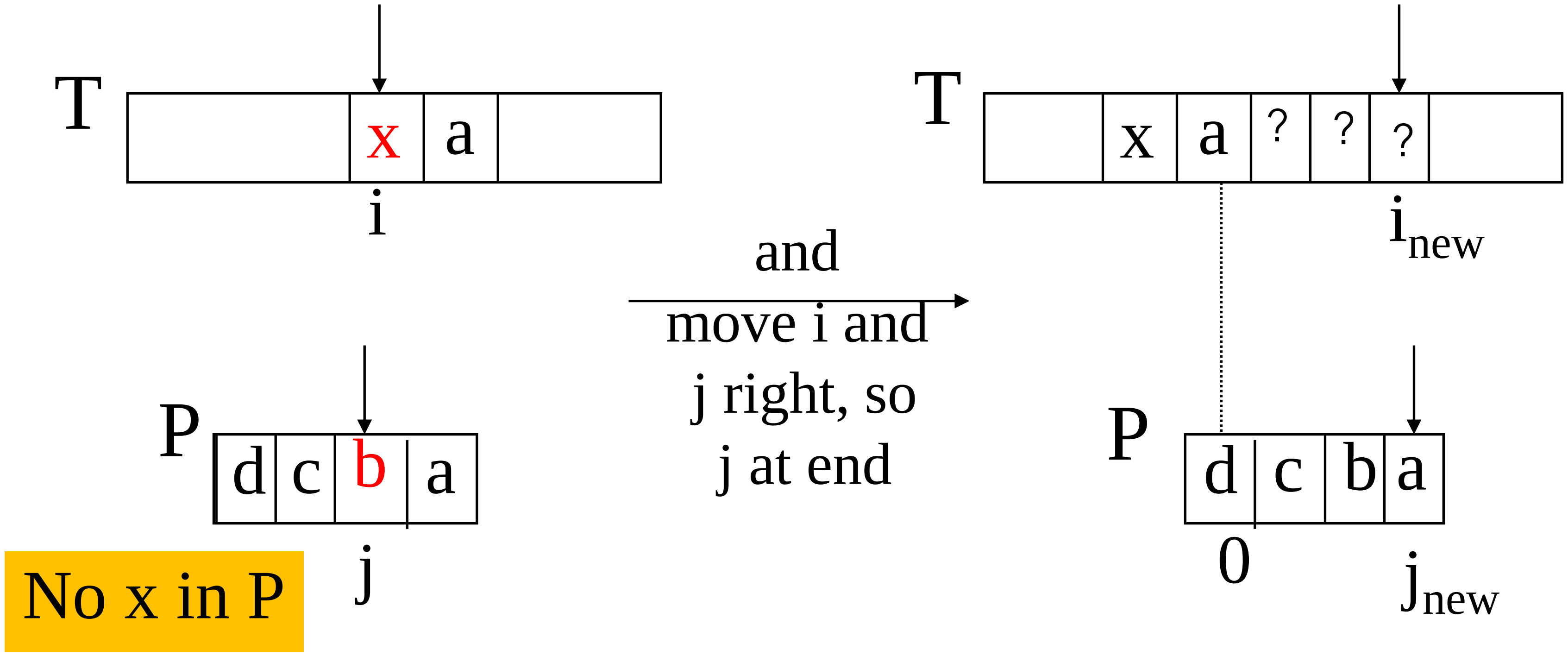
Case 2

- If P contains x somewhere, but a shift right to the last occurrence is *not* possible, then *shift P* right by 1 character to T[i+1].

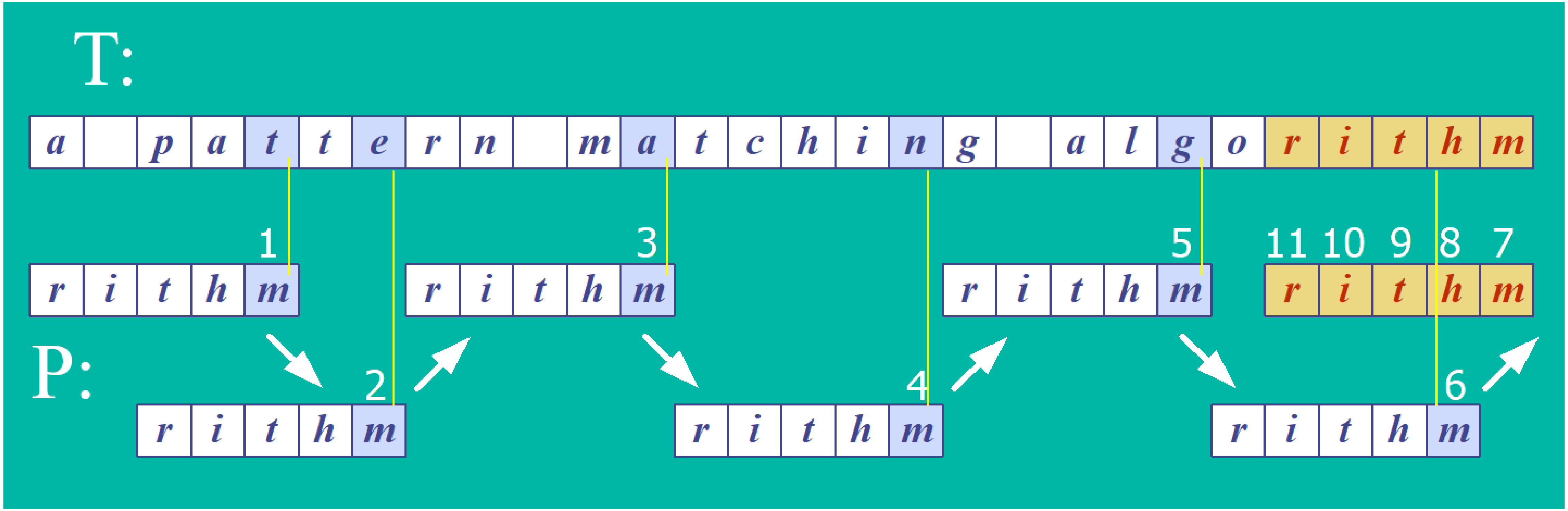


Case 3

- If cases 1 and 2 do not apply, then *shift* P to align P[0] with T[i+1].



Boyer-Moore Example (1)



Jumlah perbandingan karakter: 11 kali

Last Occurrence Function

- Boyer-Moore's algorithm preprocesses the pattern P and the alphabet A to build a last occurrence function L()
  - L() maps all the letters in A to integers

- L(x) is defined as: // x is a letter in A
  - the largest index i such that P[i] == x, or
  - -1 if no such index exists

L() Example

- A = {a, b, c, d}
- P: "abacab"

P

a	b	a	c	a	b
0	1	2	3	4	5

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

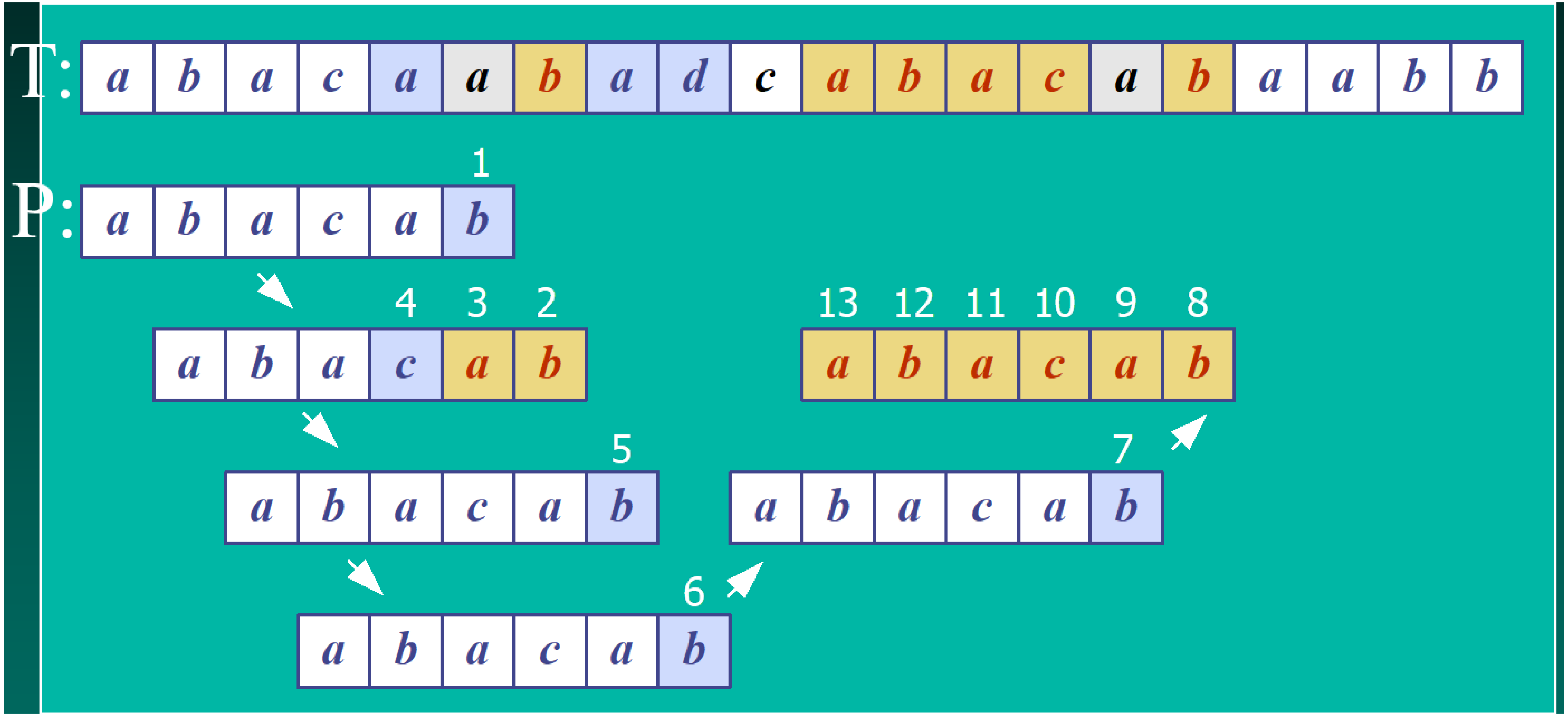
L() stores indexes into P[]

Note

- In Boyer-Moore code, L() is calculated when the pattern P is read in.

- Usually L() is stored as an array
  - something like the table in the previous slide

Boyer-Moore Example (2)



Jumlah perbandingan karakter: 13 kali

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

Boyer-Moore in Java

Return index where pattern starts, or -1

```
public static int bmMatch(String text,
                          String pattern)
 {
   int last[] = buildLast(pattern);
   int n = text.length();
   int m = pattern.length();
   int i = m-1;

   if (i > n-1)
     return -1; // no match if pattern is
                // longer than text
  :
```

```
int j = m-1;
   do {
     if (pattern.charAt(j) == text.charAt(i))
       if (j == 0)
         return i; // match
       else { // looking-glass technique
         i--;
         j--;
       }
     else { // character jump technique
        int lo = last[text.charAt(i)];  //last occ
        i = i + m - Math.min(j, 1+lo);
        j = m - 1;
     }
   } while (i <= n-1);

   return -1; // no match
 } // end of bmMatch()
```

```java
public static int[] buildLast(String pattern)
  /* Return array storing index of last
    occurrence of each ASCII char in pattern. */
  {
    int last[] = new int[128]; // ASCII char set

    for(int i=0; i < 128; i++)
      last[i] = -1; // initialize array

    for (int i = 0; i < pattern.length(); i++)
      last[pattern.charAt(i)] = i;

    return last;
  } // end of buildLast()
```

Usage

```
public static void main(String args[])
  { if (args.length != 2) {
       System.out.println("Usage: java BmSearch
                                 <text> <pattern>");
       System.exit(0);
    }
    System.out.println("Text: " + args[0]);
    System.out.println("Pattern: " + args[1]);

    int posn = bmMatch(args[0], args[1]);
    if (posn == -1)
       System.out.println("Pattern not found");
    else
       System.out.println("Pattern starts at posn "
                                          + posn);
}
```

Analysis

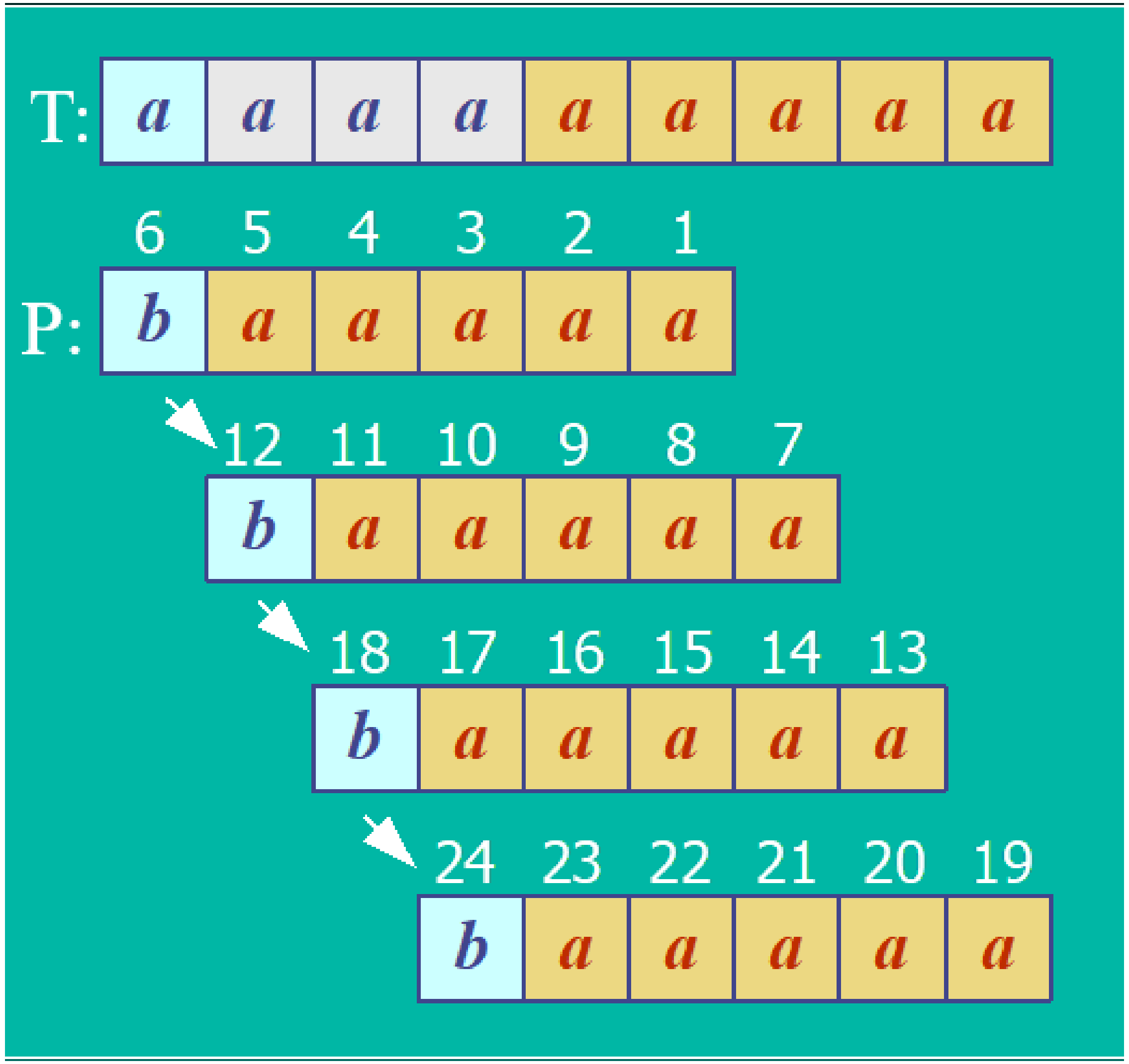
- Boyer-Moore worst case running time is O(nm + A)

- But, Boyer-Moore is fast when the alphabet (A) is large, slow when the alphabet is small.
  - e.g. good for English text, poor for binary

- Boyer-Moore is *significantly faster than brute force* for searching English text.

Worst Case Example

- T: "aaaaa...a"
- P: "baaaaa"



Jumlah perbandingan karakter: 24 kali

5. More Information

- *Algorithms in C++*
  Robert Sedgewick
  Addison-Wesley, 1992
  - chapter 19, String Searching

This book is
in the CoE library.

- Online Animated Algorithms:
  - `http://www.ics.uci.edu/~goodrich/dsa/11strings/demos/pattern/`
  - `http://www-sr.informatik.uni-tuebingen.de/~buehler/BM/BM1.html`
  - `http://www-igm.univ-mlv.fr/~lecroq/string/`

SELAMAT BELAJAR