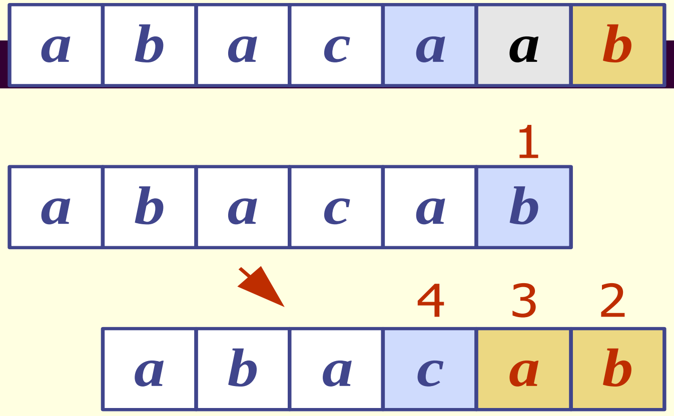
String/Pattern Matching



Contents

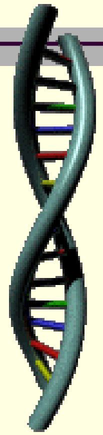
- Introduction
- The naive string matching algorithm
- Rabin Karp algorithm
- Knuth-Morris-Pratt algorithm (KMP)
- Boyer-Moore Algorithm
- Longest common subsequence(LCS)
- Analysis of All problems

1. What is Pattern Matching?

- Definition:
  - given a text string T and a pattern string P, find the pattern inside the text
    - T: “the rain in spain stays mainly on the plain”
    - P: “n th”

- Applications:
  - text editors, Web search engines (e.g. Google), image analysis

Pattern Matching - Example



Input: *P=cagc* $\sum = \{a,g,c,t\}$

1 2 3 4 5 6 7 8 …. 11

T= acagcatcagcagctagca

Output: {2,8,11}

The Problem

- Given a text T and a pattern P, check whether P occurs in T
  - eg: T = {aabbcbbcabbbcbccccabbabbccc}
  - Find all occurrences of pattern P = bbc
- There are ***variations*** of pattern matching
  - Finding “approximate” matchings
  - Finding multiple patterns etc..

String Concepts

- Assume S is a string of size m.
- A *substring* S[i .. j] of S is the string fragment between indexes i and j.
- A *prefix* of S is a substring S[0 .. i]
- A *suffix* of S is a substring S[i .. m-1]
  - i is any index between 0 and m-1

Examples

S

a	n	d	r	e	w
0					5

- Substring S[1..3] == "ndr"
- All possible prefixes of S:
  - "andrew", "andre", "andr", "and", "an”, "a"
- All possible suffixes of S:
  - "andrew", "ndrew", "drew", "rew", "ew", "w"

Why String Matching?

- **Applications in Computational Biology**
  - DNA sequence is a long word (or text) over a 4-letter alphabet
  - GTTTGAGTGGTCAGTCTTTTCGTTTCGACGGAGCCCCCAATT AATAAACTCATAAGCAGACCTCAGTTCGCTTAGAGCAGCCG AAA…..
  - Find a Specific pattern W
- **Finding patterns in documents formed using a large alphabet**
  - Word processing
  - Web searching
  - Desktop search (Google, MSN)
- **Matching strings of bytes containing**
  - Graphical data
  - Machine code
- **grep in unix**
  - grep searches for lines matching a pattern.

Strings

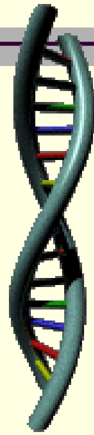
- A string is a sequence of characters
- Examples of strings:
  - Java program
  - HTML document
  - DNA sequence
  - Digitized image
- An alphabet Σ is the set of possible characters for a family of strings
- Example of alphabets:
  - ASCII
  - Unicode
  - {0, 1}
  - {A, C, G, T}
- Let P be a string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

Pattern Matching - Example

Input: $P = cagc$ $\quad \Sigma = \{a, g, c, t\}$

1 2 3 4 5 6 7 8 …. 11

$T =$ acagcatcagcagctagca



Output: {2,8,11}

String Matching

- Text string T[0..N-1]
  T = "**abacaabaccabacabaabb**"
- Pattern string P[0..M-1]
  P = "**abacab**"
- Where is the *first* instance of P in T?
  T[**10..15**] = P[0..5]
- Typically N >>> M

Contents

The Naïve String Matching Algorithm

- The naïve approach tests all the possible placement of Pattern P [1.......m] relative to text T [1......n].
- We try shift s = 0, 1.......n-m, successively and for each shift s. Compare T [s+1.......s+m] to P [1......m].
- The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.......m] = T [s+1.......s+m] for each of the n - m +1 possible value of s.

Algorithm

NAIVE-STRING-MATCHER (T, P)

1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n –m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s

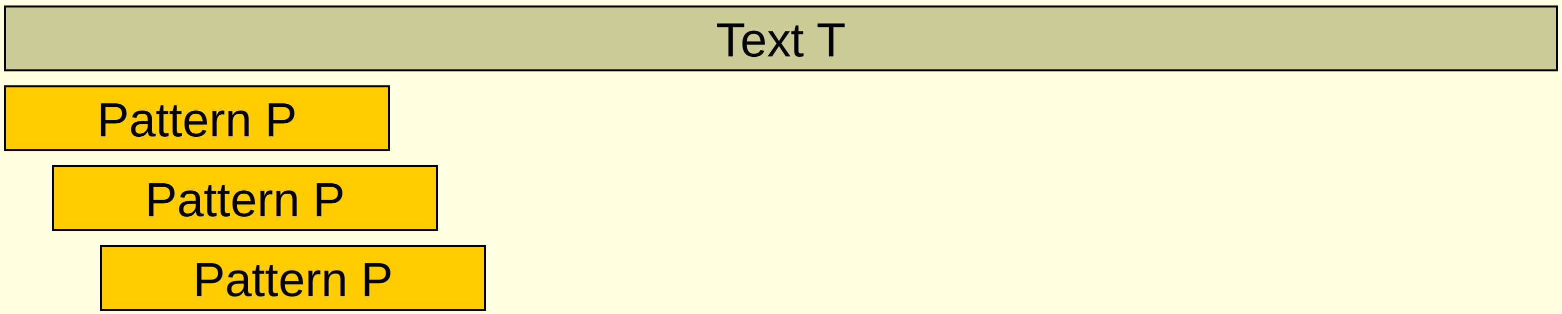
String Matching

```
abacaabaccabacabaabb
abacab
 abacab
  abacab
   abacab
    abacab
     abacab
      abacab
       abacab
        abacab
         abacab
          abacab
```

- **The brute force algorithm**
- **22+6=28** comparisons.

Naïve Algorithm
(or Brute Force)

- Assume |T| = n and |P| = m



Compare until a match is found. If so return the index where match occurs

else return -1

2. The Brute Force Algorithm

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

T:

a	n	d	r	e	w

P:

r	e	w

T:

a	n	d	r	e	w

P:

r	e	w

P moves 1 char at a time through T

. . . .

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

- Brute force pattern matching runs in time O(mn) in the worst case.
- But most searches of ordinary text take O(m+n), which is very quick.

continued

- 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.)

continued

- Example of a worst case:
  - T: "aaaaaaaaaaaaaaaaaaaaaaaaaah"
  - P: "aaah"
- Example of a more average case:
  - T: "a string searching example is standard"
  - P: "store"

A bad case

```
000000000000000001

0000-
 0000-
  0000-
   0000-
    0000-
     0000-
      0000-
       0000-
        0000-
         0000-
          0000-
           0000-
            00001
```

- 60+5 = **65** comparisons are needed

A bad case

```
000000000000000001

0000-
 0000-
  0000-
   0000-
    0000-
     0000-
      0000-
       0000-
        0000-
         0000-
          0000-
           0000-
             00001
```

- 60+5 = **65** comparisons are needed
- **How many of them could be avoided?**

Typical text matching

```
This is a sample sentence

-
 -
  -
   s-
    -
     -
      s-
       -
        -
         -
          s-
           -
            -
             -
              -
               -
                -
                 sente
```

- 20+5=**25** comparisons are needed

  (The match is near the same point in the target string as the previous example.)
- In practice, $0 \le j \le 2$

Contents

- Introduction
- The naive string matching algorithm
- Rabin Karp algorithm
- Knuth-Morris-Pratt algorithm (KMP)
- Boyer-Moore Algorithm
- Longest common subsequence(LCS)
- Analysis of All problems

Rabin-Karp – the idea

- Compare a string's hash values, rather than the strings themselves.
- For efficiency, the hash value of the next position in the text is easily computed from the hash value of the current position.

Rabin-Karp

- The Rabin-Karp string searching algorithm calculates a **hash value** for the pattern, and for each M-character subsequence of text to be compared.
- If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.
- If the hash values are equal, the algorithm will do a Brute Force comparison between the pattern and the M-character sequence.
- In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.

Rabin-Karp Example

- Hash value of “AAAAA” is 37
- Hash value of “AAAAH” is 100

```
1) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH
   37≠100      1 comparison made
2) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
    AAAAH
   37≠100      1 comparison made
3) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
     AAAAH
   37≠100      1 comparison made
4) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
      AAAAH
   37≠100      1 comparison made

    ...

N) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
                          AAAAH
    5 comparisons made       100=100
```

How Rabin-Karp works

- Let characters in both arrays T and P be digits in radix-Σ notation. ($\Sigma = (0,1,...,9)$
- Let p be the value of the characters in P
- Choose a prime number *q* such that fits within a computer word to speed computations.
- Compute (p mod q)
  - The value of p mod q is what we will be using to find all matches of the pattern P in T.

How Rabin-Karp works (continued)

- Compute (T[s+1, .., s+m] mod q) for s = 0 .. n-m
- Test against P only those sequences in T having the same (mod q) value
- (T[s+1, .., s+m] mod q) can be incrementally computed by subtracting the high-order digit, shifting, adding the low-order bit, all in modulo q arithmetic.

Rabin-Karp Algorithm

pattern is M characters long

hash_p=hash value of pattern

hash_t=hash value of first M letters in body of text

```
do
   if (hash_p == hash_t)
         brute force comparison of pattern
         and selected section of text
         hash_t= hash value of next section of text, one
   character over
while (end of text     or
         brute force comparison == true)
```

Rabin-Karp

- Common Rabin-Karp questions:

  "What is the hash function used to calculate values for character sequences?"

  "Isn't it time consuming to hash very one of the M-character sequences in the text body?"

  "Is this going to be on the final?"

- To answer some of these questions, we'll have to get mathematical.

Rabin-Karp Math

- Consider an M-character sequence as an M-digit number in base b, where b is the number of letters in the alphabet. The text subsequence t[i .. i+M-1] is mapped to the number

$$x(i) = t[i] \cdot b^{M-1} + t[i+1] \cdot b^{M-2} + \ldots + t[i+M-1]$$

• Furthermore, given x(i) we can compute x(i+1) for the next subsequence t[i+1 .. i+M] in constant time, as follows:

$$x(i+1) = t[i+1] \cdot b^{M-1} + t[i+2] \cdot b^{M-2} + \ldots + t[i+M]$$

$x(i+1) = x(i) \cdot b$ Shift left one digit

$- t[i] \cdot b^{M}$ Subtract leftmost digit

$+ t[i+M]$ Add new rightmost digit

• In this way, we never explicitly compute a new value. We simply adjust the existing value as we move over one character.

Rabin-Karp Math Example

- Let's say that our alphabet consists of 10 letters.
- our alphabet = a, b, c, d, e, f, g, h, i, j
- Let's say that "a" corresponds to 1, "b" corresponds to 2 and so on.

The hash value for string "cah" would be ...

$$3*100 + 1*10 + 8*1 = 318$$

Rabin-Karp Mods

- If M is large, then the resulting value (~bM) will be enormous. For this reason, we hash the value by taking it **mod** a prime number **q**.
- The **mod** function is particularly useful in this case due to several of its inherent properties:

  [(x mod q) + (y mod q)] mod q = (x+y) mod q

  (x mod q) mod q = x mod q
- For these reasons:

h(i)=((t[i]· bM-1 mod q) +(t[i+1]· bM-2 mod q) + ...
+(t[i+M-1] mod q))mod q

h(i+1) =(h(i) · b mod q	Shift left one digit
-t[i] · b^M mod q	Subtract leftmost digit
+t[i+M] mod q)	Add new rightmost digit
mod q	

A Rabin-Karp example

- Given T = 31415926535 and P = 26
- We choose q = 11
- P mod q = 26 mod 11 = 4

3	1	4	1	5	9	2	6	5	3	5

31 mod 11 = 9 not equal to 4

3	1	4	1	5	9	2	6	5	3	5

14 mod 11 = 3 not equal to 4

3	1	4	1	5	9	2	6	5	3	5

41 mod 11 = 8 not equal to 4

Rabin-Karp example continued

3	1	4	1	5	9	2	6	5	3	5

15 mod 11 = 4 equal to 4 -> spurious hit

3	1	4	1	5	9	2	6	5	3	5

59 mod 11 = 4 equal to 4 -> spurious hit

3	1	4	1	5	9	2	6	5	3	5

92 mod 11 = 4 equal to 4 -> spurious hit

3	1	4	1	5	9	2	6	5	3	5

26 mod 11 = 4 equal to 4 -> an exact match!!

3	1	4	1	5	9	2	6	5	3	5

65 mod 11 = 10 not equal to 4

Rabin-Karp example continued

3	1	4	1	5	9	2	6	5	3	5

53 mod 11 = 9 not equal to 4

3	1	4	1	5	9	2	6	5	3	5

35 mod 11 = 2 not equal to 4

As we can see, when a match is found, further testing is done to insure that a match has indeed been found.

Rabin-Karp Complexity

- If a sufficiently large prime number is used for the *hash function*, the hashed values of two different patterns will usually be distinct.
- If this is the case, searching takes O(N) time, where N is the number of characters in the larger body of text.
- It is always possible to construct a scenario with a worst case complexity of O(MN). This, however, is likely to happen only if the prime number used for hashing is small.

Rabin-Karp Complexity

- The running time of the Rabin-Karp algorithm in the worst-case scenario is O(n-m+1)m but it has a good average-case running time.
- If the expected number of valid shifts is small O(1) and the prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time O(n+m) plus the time to required to process spurious hits.

Analysis

- The running time of the algorithm in the **worst-case** scenario is bad.. But it has a good **average-case** running time.

- O(mn) in worst case
- O(n) if we're more optimistic…
  - Why?
  - How many hits do we expect? (board)

Rabin-Karp Summary

- *Intuition*:
  - If hash codes of two patterns are the same, then patterns “might” be the same
  - If the pattern is length m, compute hash codes of all substrings of length m
  - Leverage previous hash code to compute the next one
- Works well:
  - Multiple pattern search
- But:
  - Computing hash codes may be expensive

Contents

- Introduction
- The naive string matching algorithm
- Rabin Karp algorithm
- Knuth-Morris-Pratt algorithm (KMP)
- Boyer- Moore Algorithm.
- Longest common subsequence(LCS)
- Analysis of All problems

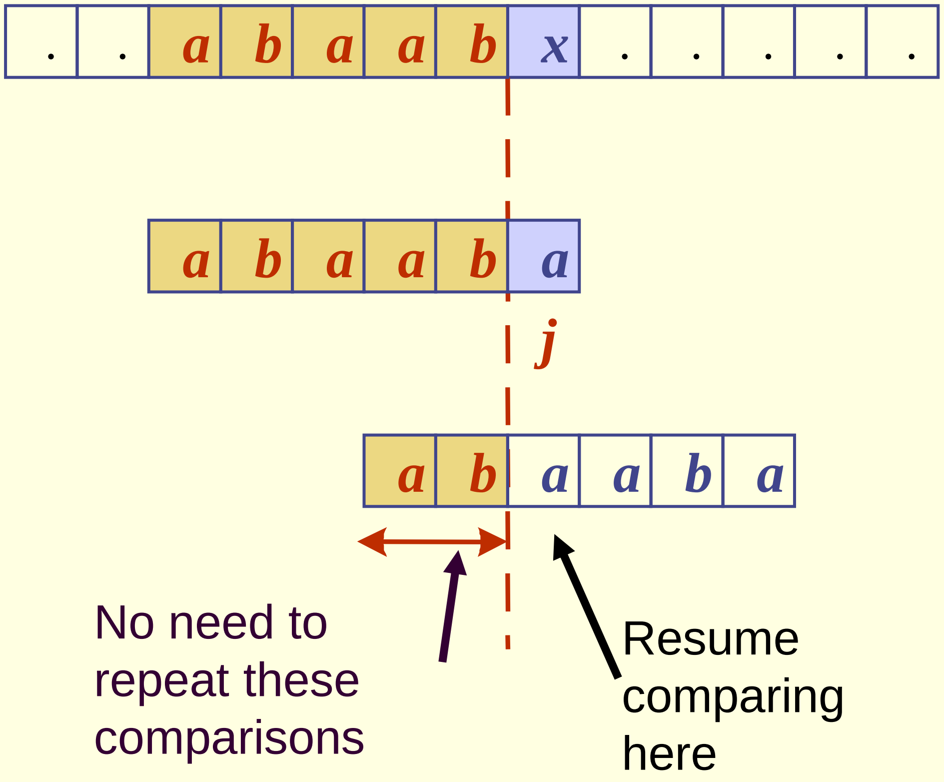
The Knuth-Morris-Pratt Algorithm

Knuth, Morris and Pratt proposed a linear time algorithm for the string matching problemin 1977.

A matching time of O(n) is achieved by avoiding comparisons with elements of ‘S’ that have previously been involved in comparison with some element of the pattern ‘p’ to be matched. i.e., backtracking on the string ‘S’ never occurs

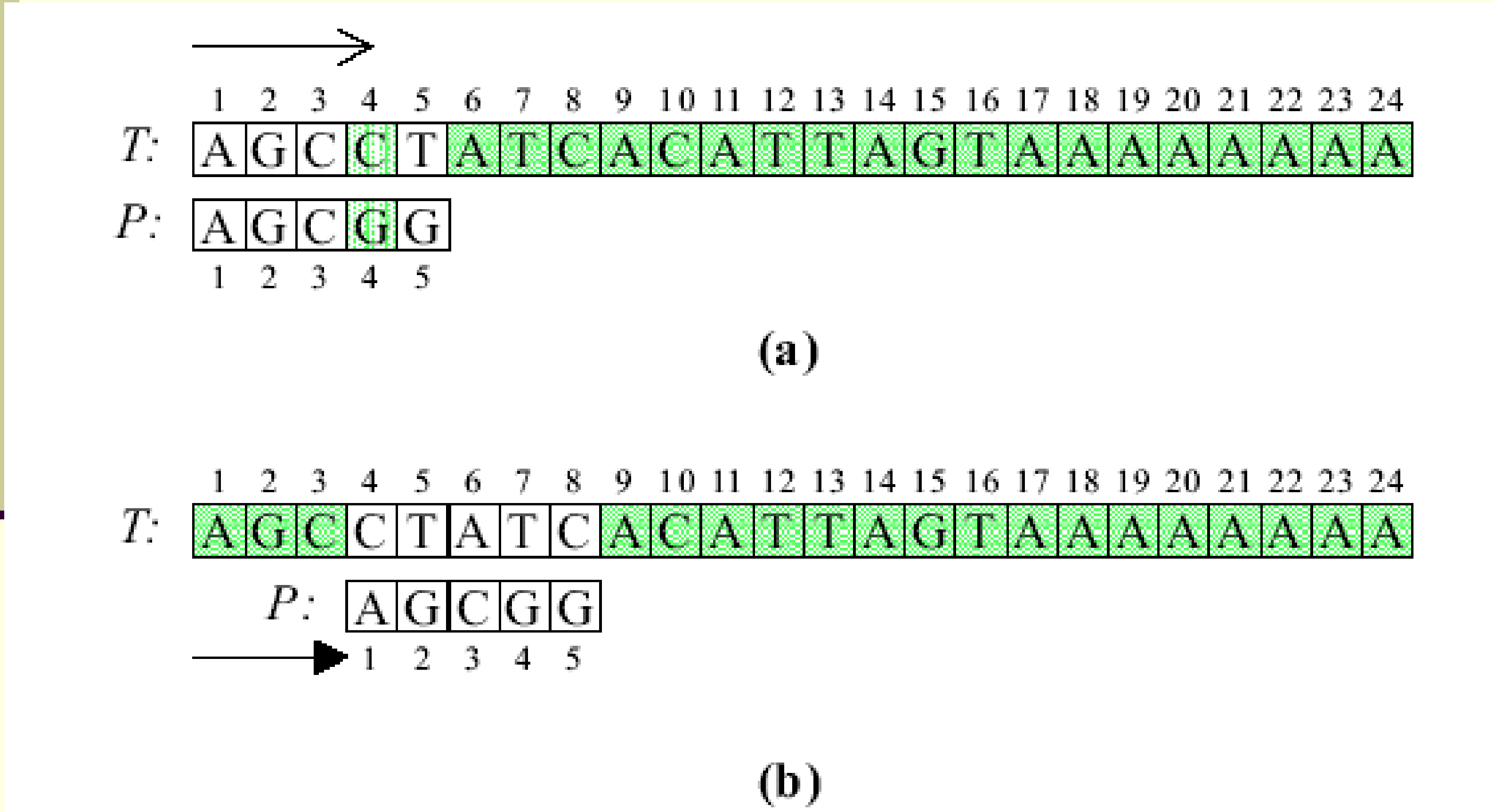
The KMP Algorithm - Motivation

- 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 $P[1..j]$



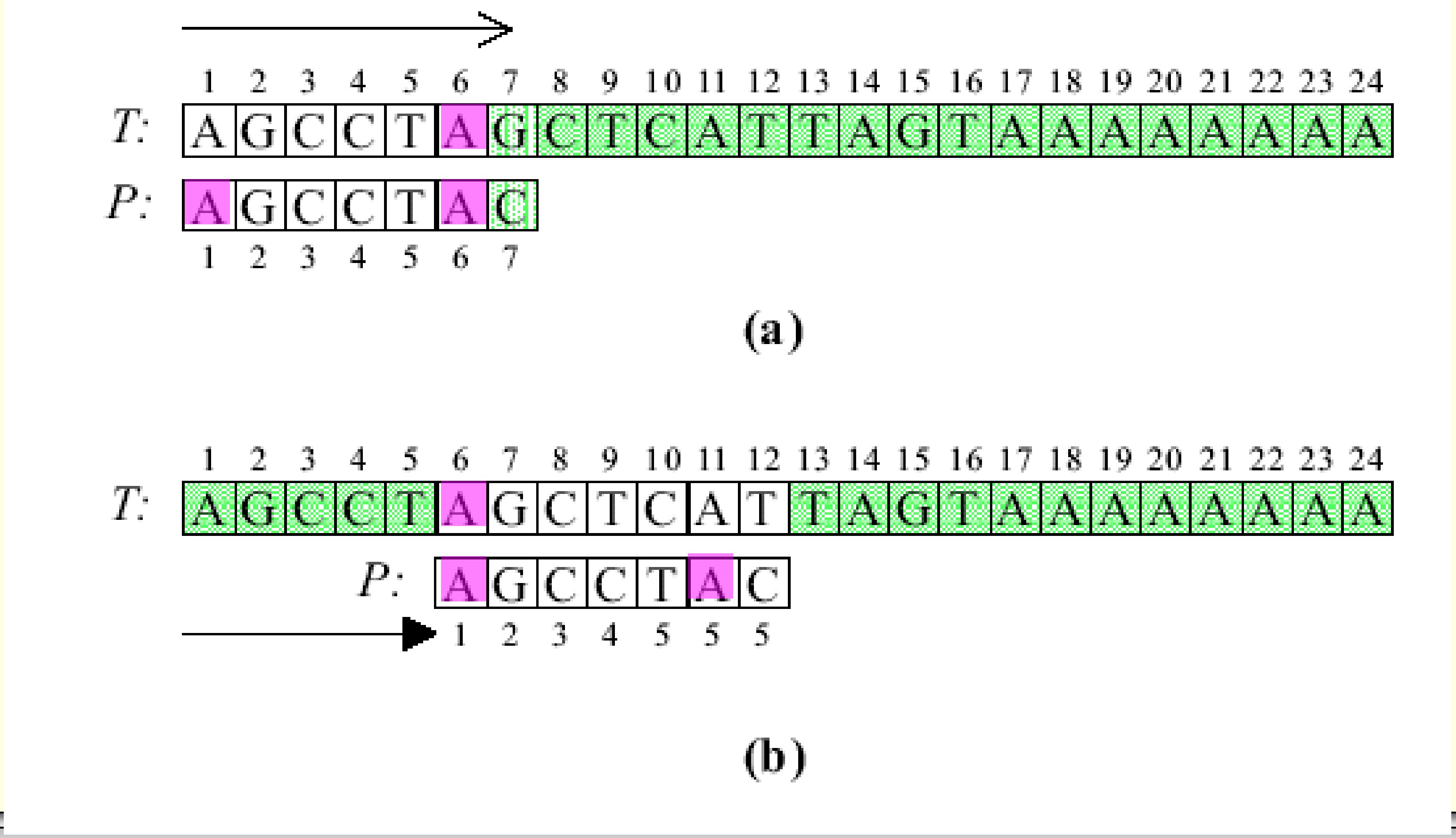
First Case for KMP Algorithm

- The first symbol of P does not appear in P again.
- We can slide to T_4, since $T_4 \neq P_4$ in (a).



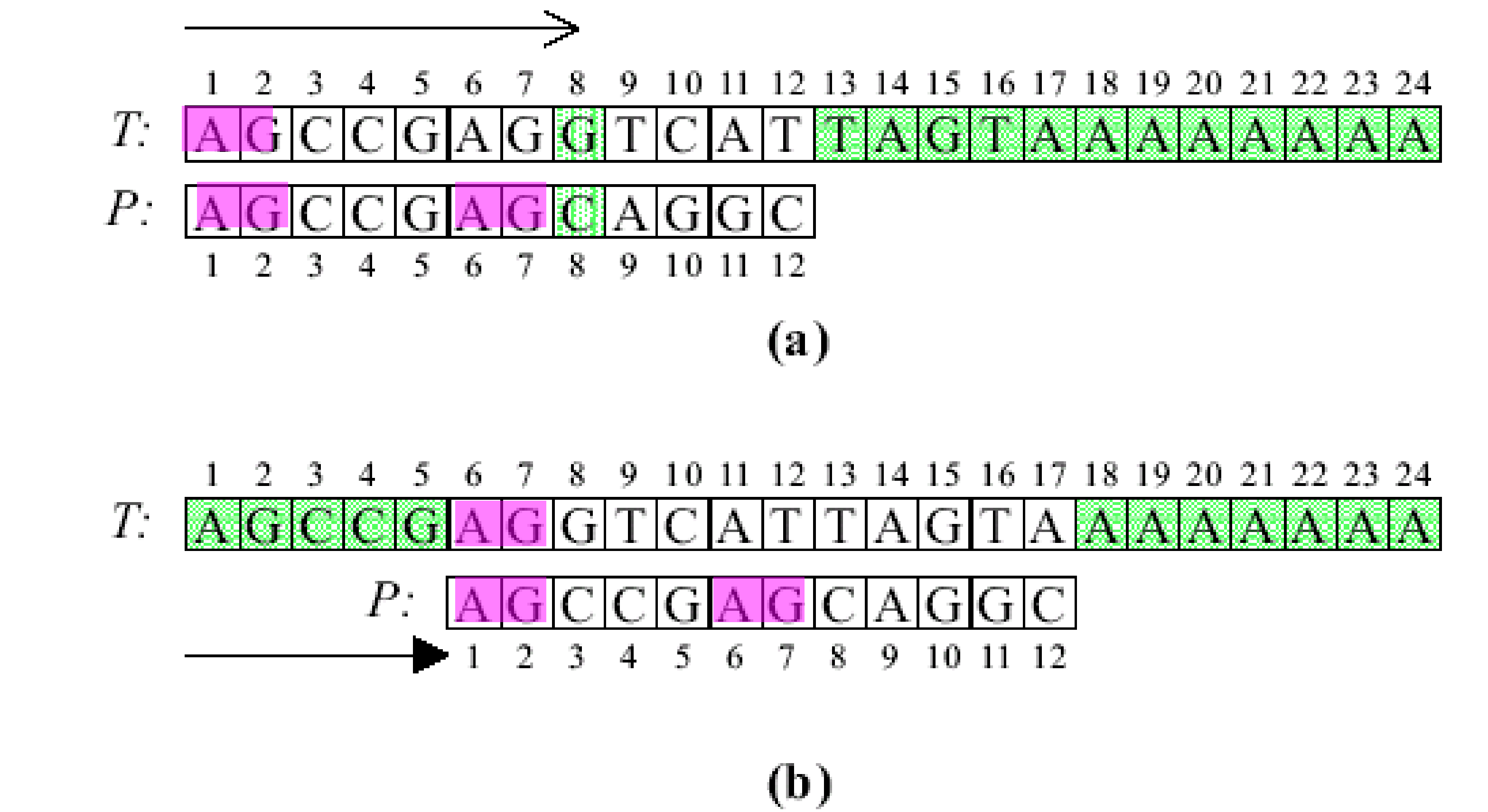
Second Case for KMP Algorithm

- The first symbol of *P* appears in *P* again.
- $T_7 \neq P_7$ in (a). We have to slide to T_6, since $P_6 = P_1 = T_6$.



Third Case for KMP Algorithm

- The prefix of *P* appears in *P* again.
- $T_8 \neq P_8$ in (a). We have to slide to T_6, since $P_{6,7}=P_{1,2}=T_{6,7}$.



The Knuth-Morris-Pratt Algorithm

- The Knuth-Morris-Pratt (KMP) string searching algorithm differs from the brute-force algorithm by keeping track of information gained from previous comparisons.
- A failure function (f) is computed that indicates how much of the last comparison can be reused if it fails.
- Specifically, f is defined to be the longest prefix of the pattern P[0,..,j] that is also a suffix of P[1,..,j]

  -Note: **not** a suffix of P[0,..,j]
- Example:-value of the
- KMP failure function:

j	0	1	2	3	4	5
$P[j]$	a	b	a	b	a	c
$f(j)$	0	0	1	2	3	0

- This shows how much of the beginning of the string matches up to the portion immediately preceding a failed comparison.

  -if the comparison fails at (4), we know the a,b in positions 2,3 is identical to positions 0,1

Components of KMP algorithm

- The prefix function, Π

The prefix function,Π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern ‘p’. In other words, this enables avoiding backtracking on the string ‘S’.

- The KMP Matcher

With string ‘S’, pattern ‘p’ and prefix function ‘Π’ as inputs, finds the occurrence of ‘p’ in ‘S’ and returns the number of shifts of ‘p’ after which occurrence is found.

The Prefix function, Π

Following pseudocode computes the prefix fucnction, Π:

Compute-Prefix-Function (p)

```
m ← length[p]               //'p' pattern to be matched
Π[1] ← 0
k ← 0
    for q ← 2 to m
        do while k > 0 and p[k+1] != p[q]
            do k ← Π[k]
           If p[k+1] = p[q]
            then k ← k +1
           Π[q] ← k
    return Π
```

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
    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: compute Π for the pattern 'p' below:

p							
	a	b	a	b	a	c	a

Initially: m = length[p] = 7
Π[1] = 0
k = 0

Step 1: q = 2, k=0
Π[2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0					

Step 2: q = 3, k = 0,
Π[3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1				

Step 3: q = 4, k = 1
Π[4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
Π	0	0	1	2			

Step 4: $q = 5, k = 2$

$\Pi[5] = 3$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3		

Step 5: $q = 6, k = 3$

$\Pi[6] = 1$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	

Step 6: $q = 7, k = 1$

$\Pi[7] = 1$

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	1

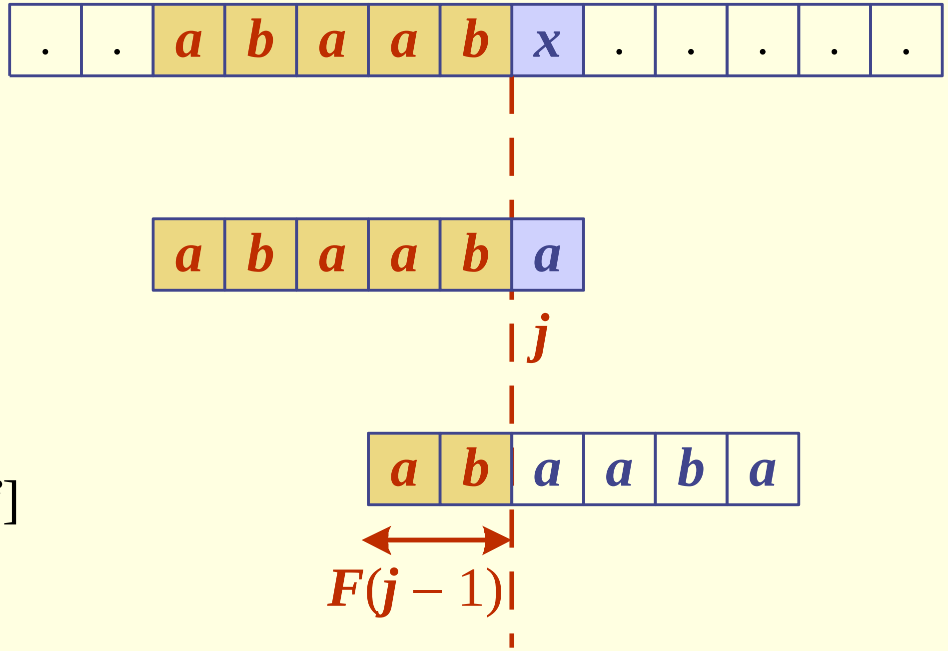
After iterating 6 times, the prefix function computation is complete: →

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	1

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



Building the Table for f

- P = 1010011
- Find self-overlaps

Prefix	Overlap	j	f
1	.	1	0
10	.	2	0
10**1**	**1**	3	1
10**10**	**10**	4	2
10100	.	5	0
10100**1**	**1**	6	1
101001**1**	**1**	7	1

What **f** means

Prefix	Overlap	j	f
1	.	1	0
10	.	2	0
10**1**	**1**	3	1
10**10**	**10**	4	2
10100	.	5	0
10100**1**	**1**	6	1
101001**1**	**1**	7	1

- If **f** is zero, there is no self-match.
  - Set **j=0**
  - Do not change **i**.
    - The next match is **T[i] ? P[0]**

- **f** non-zero implies there is a self-match.

  E.g., **f=2** means **P**[0..1] **= P**[j-2..j-1]
    - Hence must start new comparison at **j-2**, since we know **T**[i-2..i-1] = **P**[0..1]

  In general:
  - Set **j=f[j-1]**
  - Do not change **i**.
    - The next match is **T[i] ? P[f[j-1]]**

Favorable conditions

- P = 1234567
- Find self-overlaps

Prefix	Overlap	j	f
1	.	1	0
12	.	2	0
123	.	3	0
1234	.	4	0
12345	.	5	0
123456	.	6	0
1234567	.	7	0

Mixed conditions

- P = 1231234
- Find self-overlaps

Prefix	Overlap	j	f
1	.	1	0
12	.	2	0
123	.	3	0
123**1**	**1**	4	1
123**12**	**12**	5	2
123**123**	**123**	6	3
1231234	.	7	0

Poor conditions

- P = 1111110
- Find self-overlaps

Prefix	Overlap	j	f
1 .		1	0
1**1**	**1**	2	1
1**11**	**11**	3	2
1**111**	**111**	4	3
1**1111**	**1111**	5	4
1**11111**	**11111**	6	5
1111110 .		7	0

The KMP Matcher

The KMP Matcher, with pattern ‘p’, string ‘S’ and prefix function ‘Π’ as input, finds a match of p in S.

Following pseudocode computes the matching component of KMP algorithm:

KMP-Matcher(S,p)

```
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0                                    //number of characters matched
5 for i ← 1 to n                           //scan S from left to right
6     do while  q > 0 and p[q+1] != S[i]
7              do  q ← Π[q]                //next character does not match
8          if p[q+1] = S[i]
9            then q ← q + 1                //next character matches
10         if q = m                        //is all of p matched?
11           then print "Pattern occurs with shift" i – m
12                q ← Π[ q]                // look for the next match
```

Note: KMP finds every occurrence of a ‘p’ in ‘S’. That is why KMP does not terminate in step 12, rather it searches remainder of ‘S’ for any more occurrences of ‘p’.

<u>Illustration:</u> given a String 'S' and pattern 'p' as follows:

S

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

p

a	b	a	b	a	c	a

Let us execute the KMP algorithm to find whether 'p' occurs in 'S'.

For 'p' the prefix function, Π was computed previously and is as follows:

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	1

Initially: n = size of S = 15;
m = size of p = 7

Step 1: i = 1, q = 0
comparing p[1] with S[1]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

P[1] does not match with S[1]. ‘p’ will be shifted one position to the right.

Step 2: i = 2, q = 0
comparing p[1] with S[2]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p		a	b	a	b	a	c	a

P[1] matches S[2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

Comparing p[2] with S[3]   p[2] does not match with S[3]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Backtracking on p, comparing p[1] and S[3]

Step 4: i = 4, q = 0

comparing p[1] with S[4]   p[1] does not match with S[4]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 5: i = 5, q = 0

comparing p[1] with S[5]   p[1] matches with S[5]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 6: i = 6, q = 1

Comparing p[2] with S[6]   p[2] matches with S[6]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 7: i = 7, q = 2

Comparing p[3] with S[7]   p[3] matches with S[7]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 8: i = 8, q = 3

Comparing p[4] with S[8]   p[4] matches with S[8]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 9: i = 9, q = 4

Comparing p[5] with S[9] p[5] matches with S[9]

S

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p

a	b	a	b	a	c	a

Step 10: i = 10, q = 5

Comparing p[6] with S[10] p[6] doesn’t match with S[10]

S

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p

a	b	a	b	a	c	a

Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

Step 11: i = 11, q = 4

Comparing p[5] with S[11] p[5] matches with S[11]

S

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p

a	b	a	b	a	c	a

Step 12: i = 12, q = 5

Comparing p[6] with S[12]   p[6] matches with S[12]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p							a	b	a	b	a	c	a		

Step 13: i = 13, q = 6

Comparing p[7] with S[13]   p[7] matches with S[13]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p							a	b	a	b	a	c	a		

Pattern ‘p’ has been found to completely occur in string ‘S’. The total number of shifts that took place for the match to be found are: i – m = 13 – 7 = 6 shifts.

Running - time analysis

- Compute-Prefix-Function (Π)

```
1  m ← length[p]                //'p' pattern to be matched
2  Π[1] ← 0
3  k ← 0
4      for q ← 2 to m
5          do while k > 0 and p[k+1] != p[q]
6              do k ← Π[k]
7             If p[k+1] = p[q]
8               then k ← k +1
9             Π[q] ← k
10     return Π
```

In the above pseudocode for computing the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step 1 to step 3 take constant time. Hence the running time of compute prefix function is Θ(m).

- KMP Matcher

```
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0
5 for i ← 1 to n
6    do while  q > 0 and p[q+1] != S[i]
7           do  q ← Π[q]
8     if p[q+1] = S[i]
9        then q ← q + 1
10    if q = m
11      then print "Pattern occurs with shift" i – m
12               q ← Π[ q]
```

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S'. Since step 1 to step 4 take constant time, the running time is dominated by this for loop. Thus running time of matching function is Θ(n).

The KMP Algorithm

- Time Complexity Analysis
- define k = i - j
- In every iteration through the while loop, one of three things happens.
  1) if T[i] = P[j], then i increases by 1, as does j k remains the same.
  2) if T[i] != P[j] and j > 0, then i does not change and k increases by at least 1, since k changes from i - j to i - f(j-1)
  3) if T[i] != P[j] and j = 0, then i increases by 1 and k increases by 1 since j remains the same.
- Thus, each time through the loop, either i or k increases by at least 1, so the greatest possible number of loops is 2n
- This of course assumes that f has already been computed.
- However, f is computed in much the same manner as KMPMatch so the time complexity argument is analogous. KMPFailureFunction is O(m)
- Total Time Complexity: O(n + m)

Example-2

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

Brute Force

```
000000000000000000000000001
00000000000000-
 00000000000000-
  00000000000000-
   00000000000000-
    00000000000000-
     00000000000000-
      00000000000000-
       00000000000000-
        00000000000000-
         00000000000000-
          00000000000000-
           00000000000000-
            00000000000000-
             00000000000000-
```

- A worse case example:

  196 + 14 = 210 comparisons

KMP

```
000000000000000000000000001

00000000000000-
             0-
              0-
               0-
                0-
                 0-
                  0-
                   0-
                    0-
                     0-
                      0-
                       0-
                        0-
                         0-
                          01
```

28+14 = 42 comparisons

Brute Force

```
abcdeabcdeabcedfghijkl

-
 bc-
  -
   -
    -
     -
      bc-
        -
         -
          -
           -
            bcedfg
```

21 comparisons

KMP

```
abcdeabcdeabcedfghijkl

-
 bc-
   -
    -
     -
      bc-
        -
         -
          -
           bcedfg
```

19 comparisons

5 preparation comparisons

Exercises

- Suppose we are given the pattern P = 10010001 and
- text T = 0001001001000010111
- do the following
  - Construct the KMP table for P
  - Trace the KMP algorithm with T

Contents

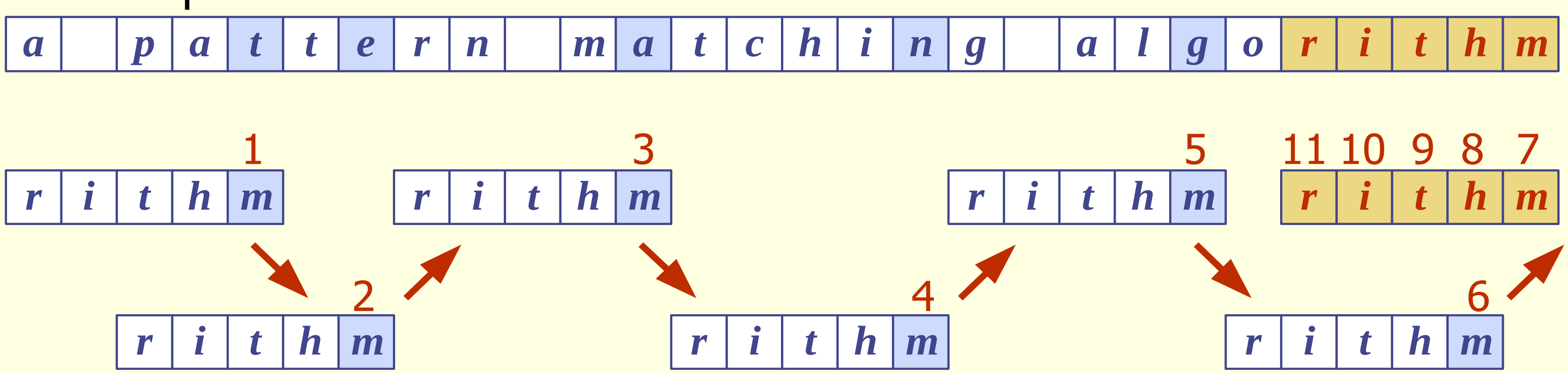
Boyer-Moore Heuristics

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

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

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

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



Last-Occurrence Function

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

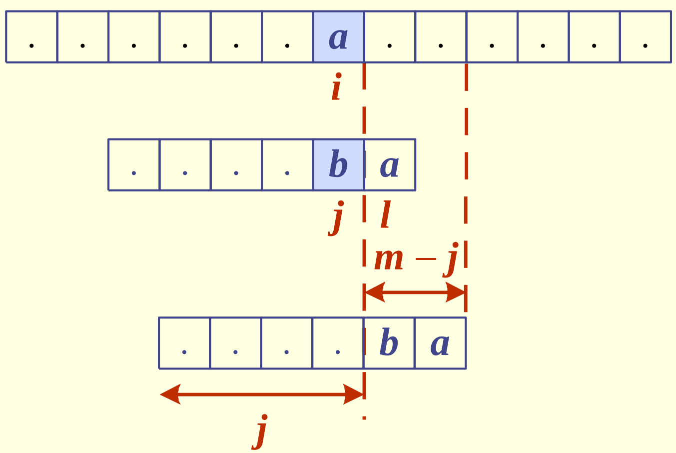
c	a	b	c	d
$L(c)$	4	5	3	-1

- 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

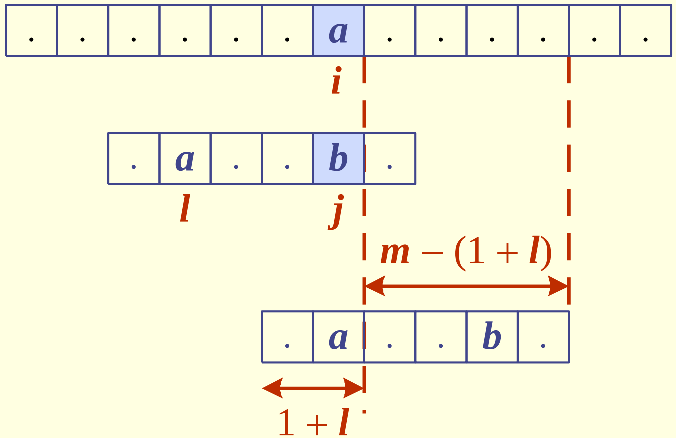
The Boyer-Moore Algorithm

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

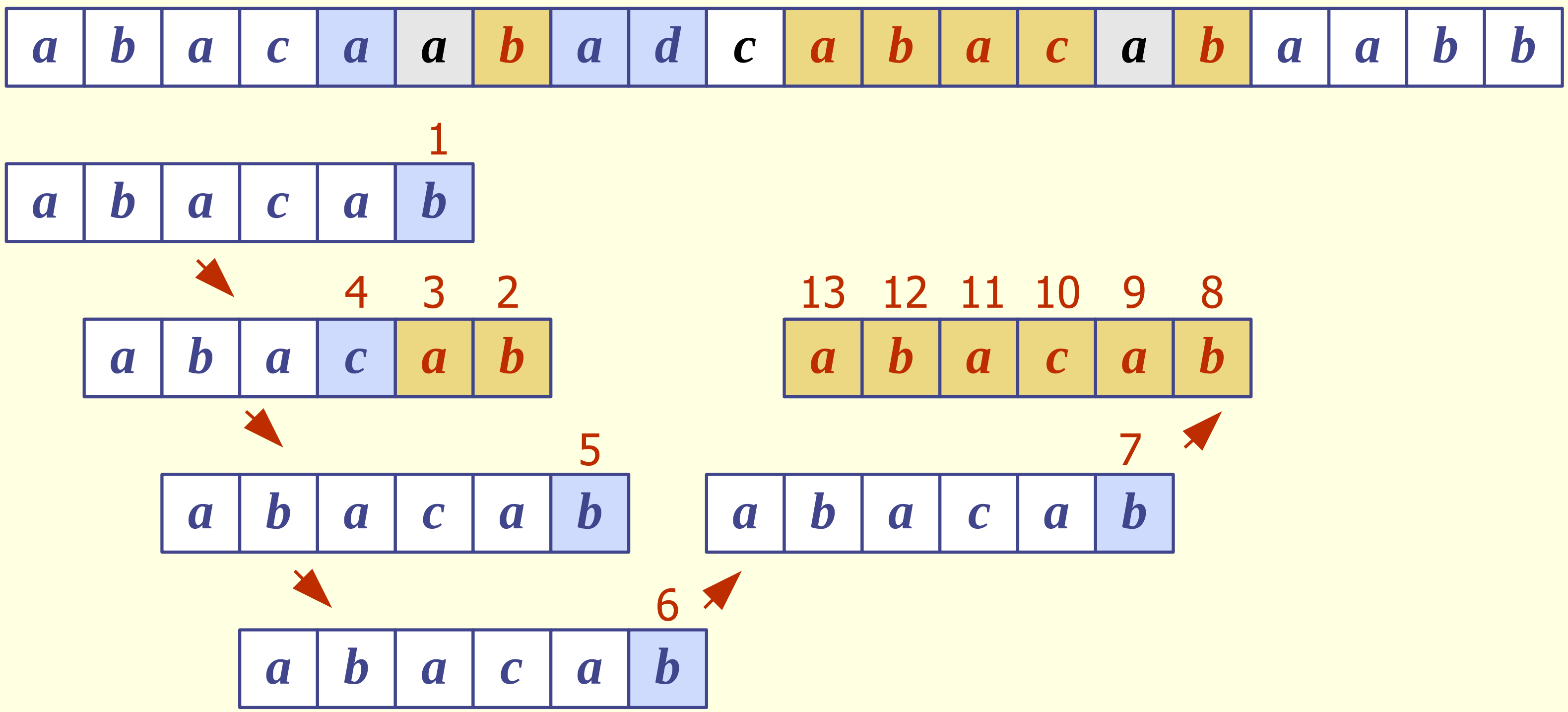
Case 1: $j \le 1 + l$



Case 2: $1 + l \le j$



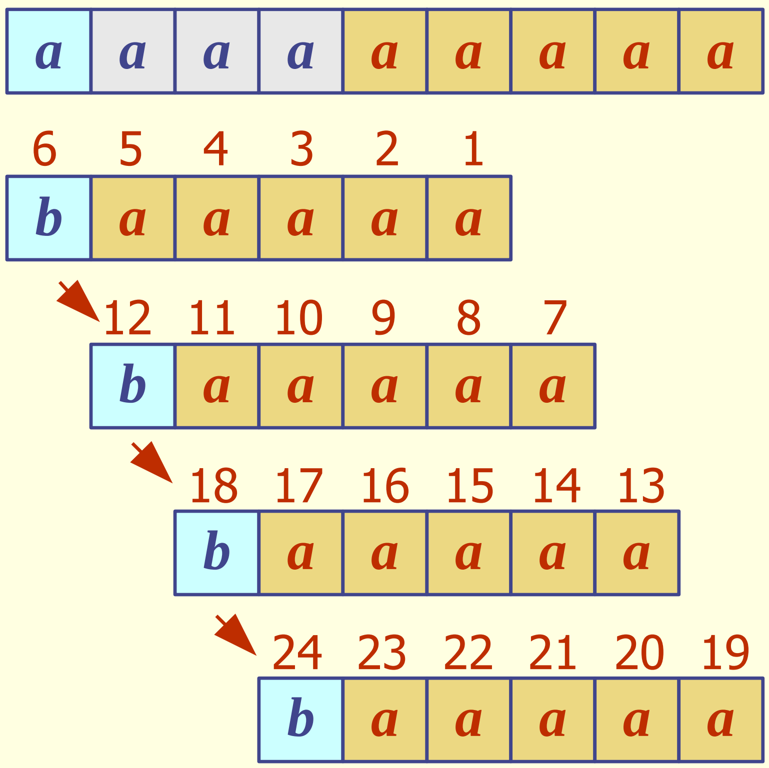
Example



a b a c a a b a d c a b a c a b a a b b
1
a b a c a b
4 3 2
a b a c a b
13 12 11 10 9 8
a b a c a b
5
a b a c a b
7
a b a c a b
6
a b a c a b

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



The Boyer-Moore Algorithm

- The B-M algorithm takes a 'backward' approach: the pattern string (P) is aligned with the start of the text string (T), and then compares the characters of a pattern from right to left, beginning with rightmost character.
- If a character is compared that is not within the pattern, no match can be found by analysing any further aspects at this position so the pattern can be changed entirely past the mismatching character.
- For deciding the possible shifts, B-M algorithm uses two pre-processing strategies simultaneously. Whenever a mismatch occurs, the algorithm calculates a variation using both approaches and selects the more significant shift thus, if make use of the most effective strategy for each case.

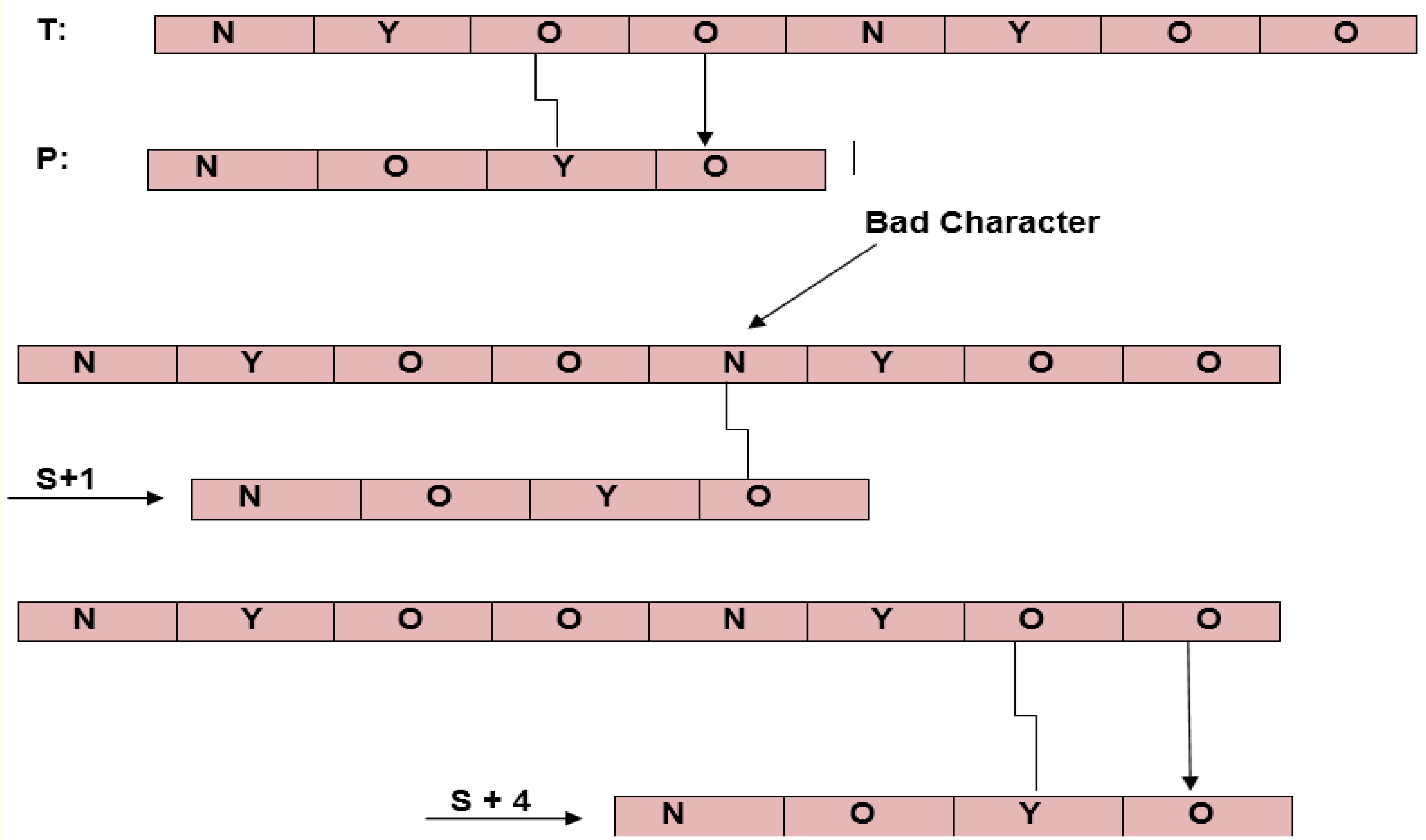
- The two strategies are called heuristics of B - M as they are used to reduce the search. They are:
  - Bad Character Heuristics
  - Good Suffix Heuristics

1. Bad Character Heuristics

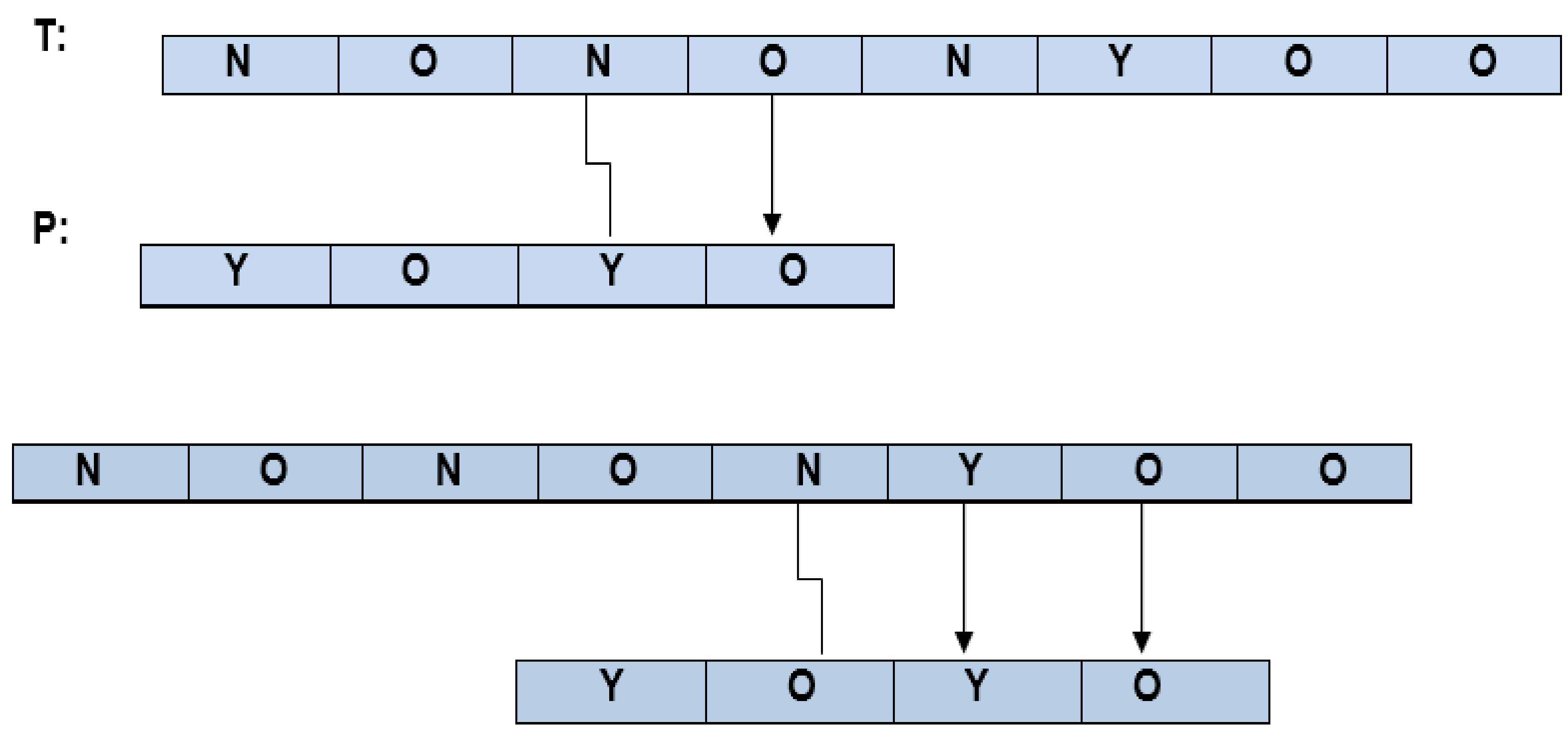
- This Heuristics has two implications:
- Suppose there is a character in a text in which does not occur in a pattern at all. When a mismatch happens at this character (called as bad character), the whole pattern can be changed, begin matching form substring next to this 'bad character.'
- On the other hand, it might be that a bad character is present in the pattern, in this case, align the nature of the pattern with a bad character in the text.
- Thus in any case shift may be higher than one.

Example1:

Let Text T = <nyoo nyoo> and pattern P = <noyo>

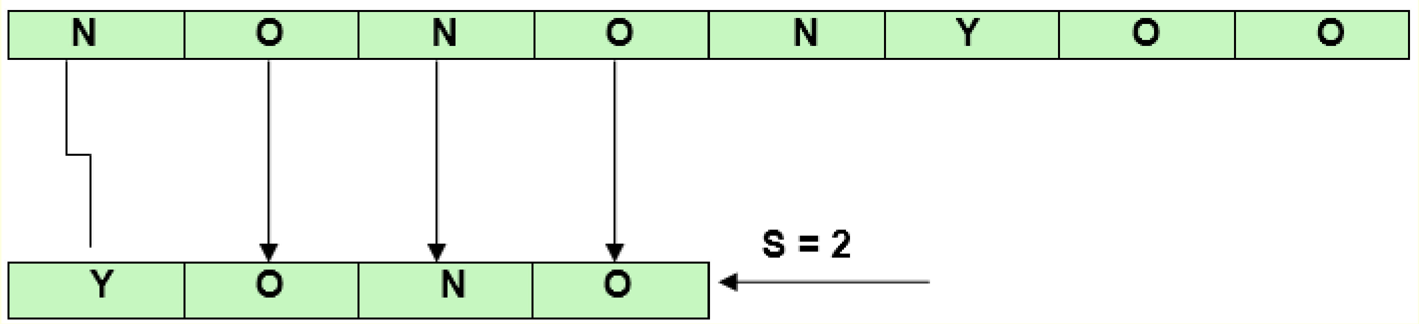


Example2: If a bad character doesn't exist the pattern then.



Problem in Bad-Character Heuristics:

- In some cases, Bad-Character Heuristics produces some negative shifts.



- This means that we need some extra information to produce a shift on encountering a bad character. This information is about the last position of every aspect in the pattern and also the set of characters used in a pattern (often called the alphabet ∑of a pattern).

COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, Σ)

1.for each character $a \in \Sigma$

2. do λ [a] = 0

3. for $j \leftarrow 1$ to m

4. do $\lambda [P[j]] \leftarrow j$

5. Return λ

2. Good Suffix Heuristics:

- A good suffix is a suffix that has matched successfully. After a mismatch which has a negative shift in bad character heuristics, look if a substring of pattern matched till bad character has a good suffix in it, if it is so then we have an onward jump equal to the length of suffix found.

COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)

1. Π ← COMPUTE-PREFIX-FUNCTION (P)
2. P'← reverse (P)
3. Π'← COMPUTE-PREFIX-FUNCTION (P')
4. for j ← 0 to m
5. do ɣ [j] ← m - Π [m]
6. for l ← 1 to m
7. do j ← m - Π' [L]
8. If ɣ [j] > l - Π' [L]
9. then ɣ [j] ← 1 - Π'[L]
10. Return ɣ

```
BOYER-MOORE-MATCHER (T, P, Σ)
n ←length [T]
m ←length [P]
λ← COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, Σ )
γ← COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)
s ←0
While s ≤ n - m
do j ← m
While j > 0 and P [j] = T [s + j]
do j ←j-1
If j = 0
then print "Pattern occurs at shift" s
s ← s + γ[0]
else s ← s + max (γ [j], j -  λ[T[s+j]])
```

Contents

- Introduction
- The naive string matching algorithm
- Rabin Karp algorithm
- Knuth-Morris-Pratt algorithm (KMP)
- Boyer-Moore Algorithm
- Longest common subsequence(LCS)
- Analysis of All problems

Common subsequence

- A subsequence of a string is the string with zero or more chars left out
- A common subsequence of two strings:
  - A subsequence of both strings
  - Ex: x = {A B C B D A B }, y = {B D C A B A}
  - {B C} and {A A} are both common subsequences of x and y

Longest Common Subsequence

- Given two sequences $x[1 \,.\,.\, m]$ and $y[1 \,.\,.\, n]$, find a longest subsequence common to them both.

"a" *not* "the"

$$\left.\begin{array}{llllllll} x: & A & B & C & B & D & A & B \\ y: & B & D & C & A & B & A & \end{array}\right\} \begin{array}{l} \text{BCBA} = \\ \text{LCS}(x, y) \end{array}$$

functional notation, but not a function

Longest Common Subsequence Problem

- A Longest Common Subsequence LCS of two strings S1 and S2 is a longest string the can be obtained from S1 and from S2 by deleting elements.
- For example, S1 = “thoughtful” and S2 = “shuffle” have an LCS: “hufl”.
- Useful in spelling correction, document comparison, etc.

Brute-force LCS algorithm

Check every subsequence of $x[1 \,..\, m]$ to see if it is also a subsequence of $y[1 \,..\, n]$.

Analysis

- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).
- Hence, the runtime would be exponential !

Towards a better algorithm: a DP strategy

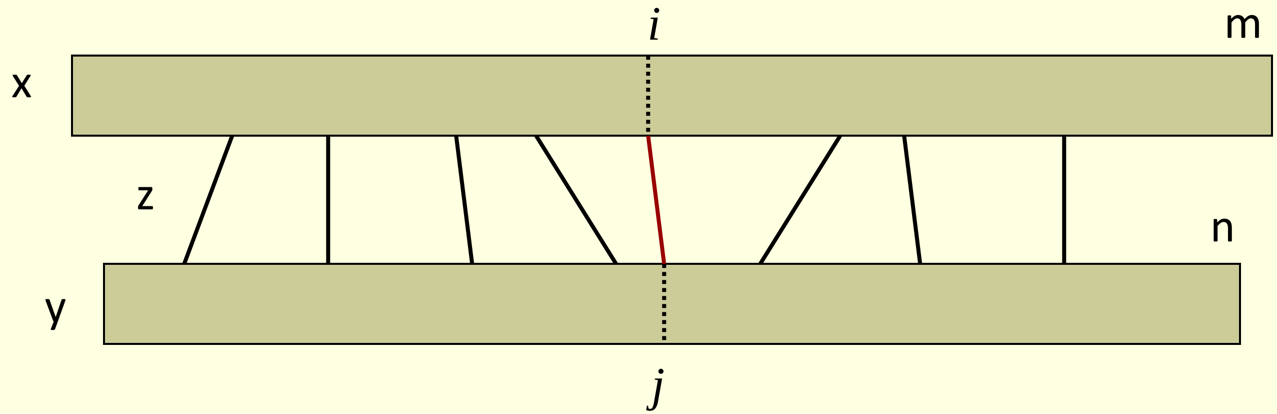
- Key: optimal substructure and overlapping sub-problems
- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.

Brute force solution

- **Solution: For every subsequence of x, check if it is a subsequence of y.**
- **Analysis :**
  - **There are 2^m subsequences of x.**
  - **Each check takes O(n) time, since we scan y for first element, and then scan for second element, etc.**
  - **The worst case running time is $O(n2^m)$.**

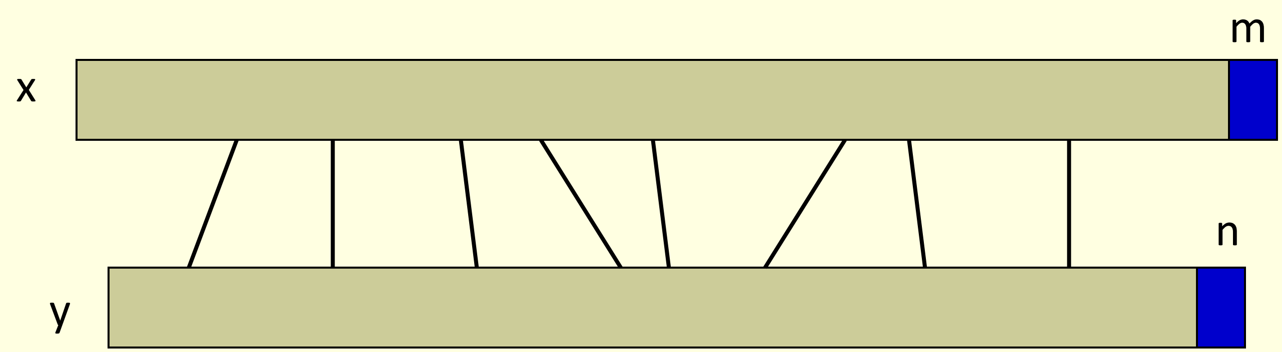
Optimal substructure

- Notice that the LCS problem has *optimal substructure*: parts of the final solution are solutions of subproblems.
  - If z = LCS(x, y), then any prefix of z is an LCS of a prefix of x and a prefix of y.



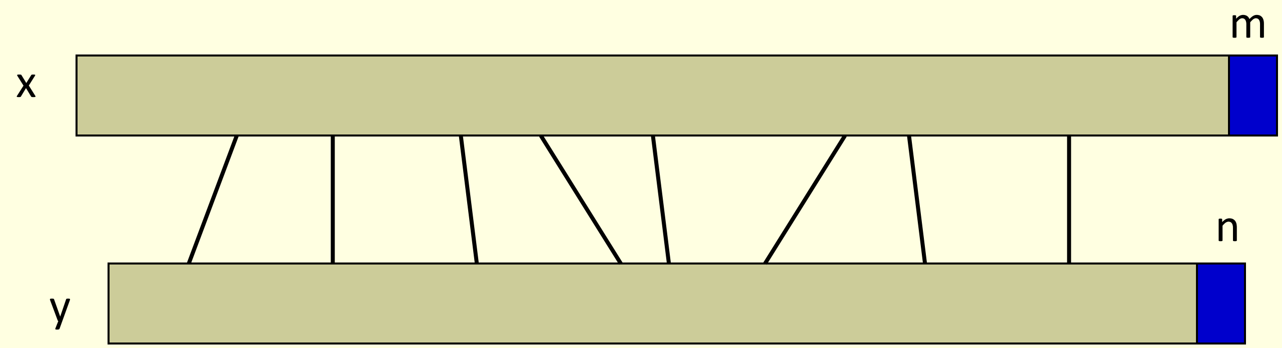
- Subproblems: "find LCS of pairs of *prefixes* of x and y"

Recursive thinking



- Case 1: $x[m]=y[n]$. There is **an** optimal LCS that matches $x[m]$ with $y[n]$. → Find out LCS $(x[1..m-1], y[1..n-1])$
- Case 2: $x[m]\neq y[n]$. At most one of them is in LCS
  - Case 2.1: $x[m]$ *not in LCS* → Find out LCS $(x[1..m-1], y[1..n])$
  - Case 2.2: $y[n]$ *not in LCS* → Find out LCS $(x[1..m], y[1..n-1])$

Recursive thinking



- Case 1: $x[m]=y[n]$
  - $LCS(x, y) = LCS(x[1..m-1], y[1..n-1]) \,||\, x[m]$

Reduce both sequences by 1 char

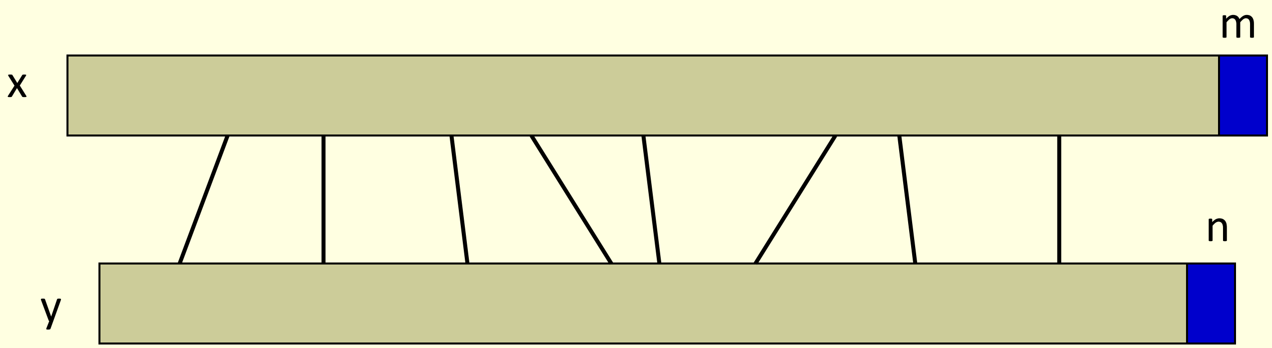
concatenate

- Case 2: $x[m] \neq y[n]$
  - $LCS(x, y) = LCS(x[1..m-1], y[1..n])$ or

    $LCS(x[1..m], y[1..n-1])$, whichever is longer

Reduce either sequence by 1 char

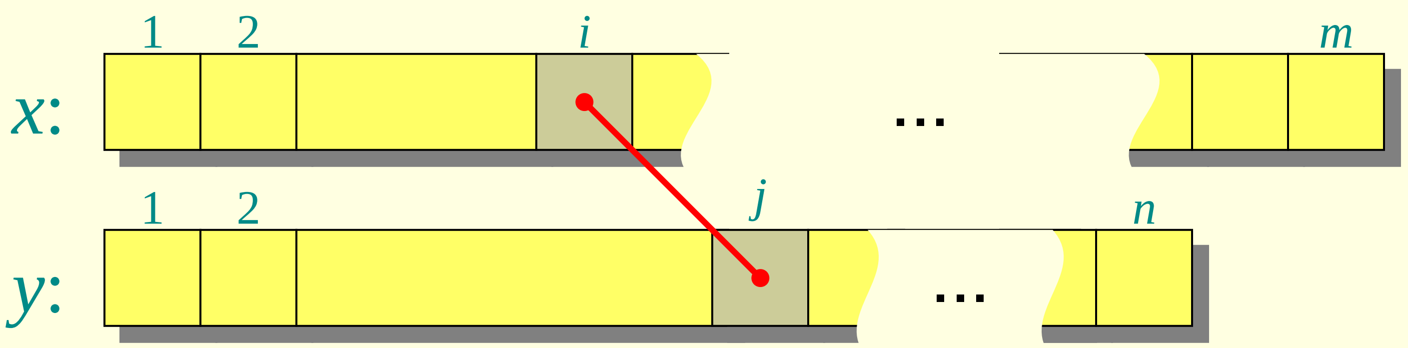
Finding length of LCS



- Let *c[i, j]* be the length of LCS(*x[1..i], y[1..j]*)
  => *c[m, n]* is the length of LCS(*x, y*)
- If $x[m] = y[n]$
  $c[m, n] = c[m\text{-}1, n\text{-}1] + 1$
- If $x[m] \,!\!= y[n]$
  $c[m, n] = max \{ c[m\text{-}1, n], c[m, n\text{-}1] \}$

Generalize: recursive formulation

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$



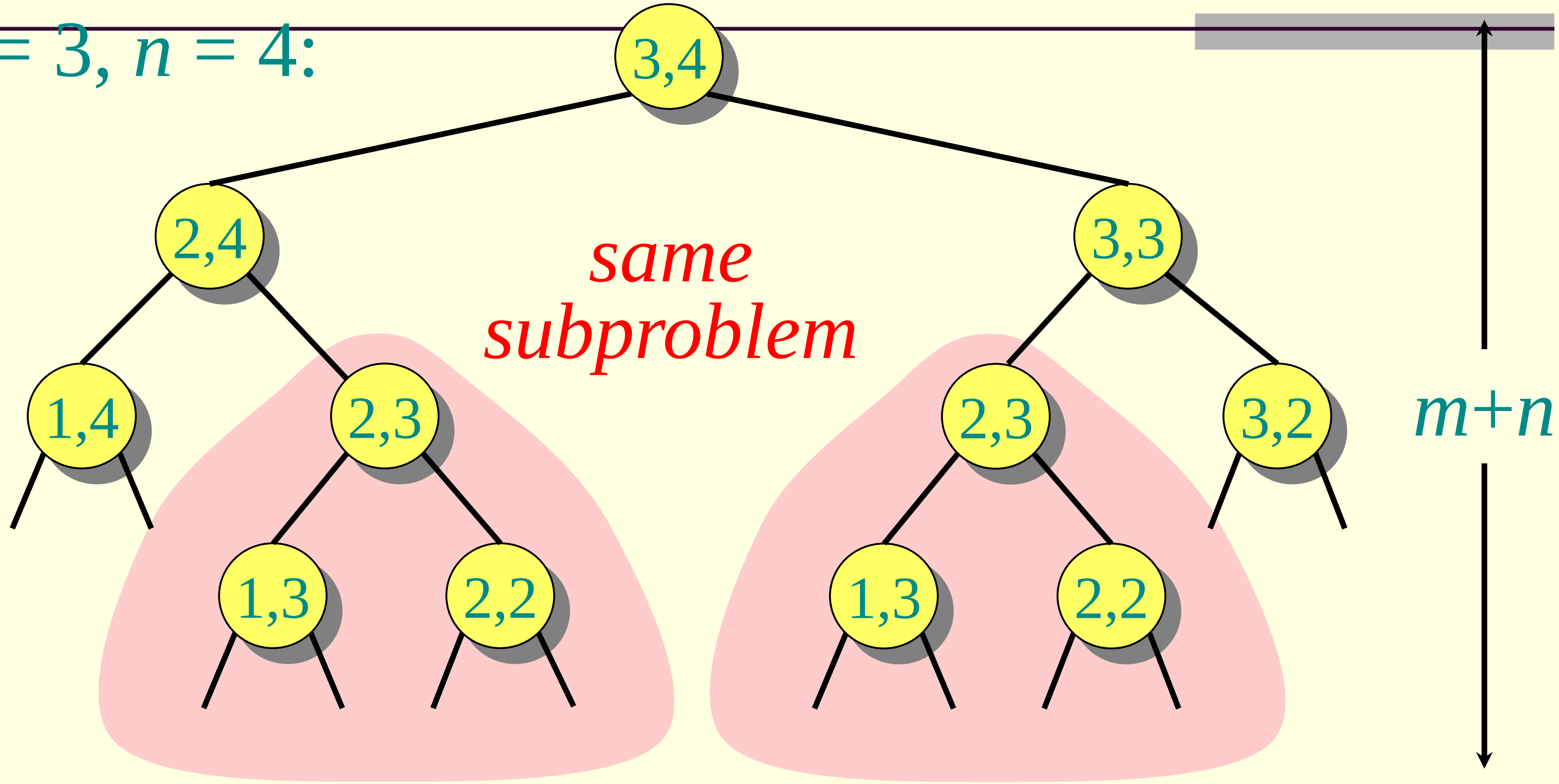
Recursive algorithm for LCS

LCS(x, y, i, j)
  if $x[i] = y[j]$
    then $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$
    else $c[i, j] \leftarrow \max\{LCS(x, y, i-1, j)$,
      LCS$(x, y, i, j-1)\}$

Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion tree

$m = 3, n = 4$:



Height $= m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!

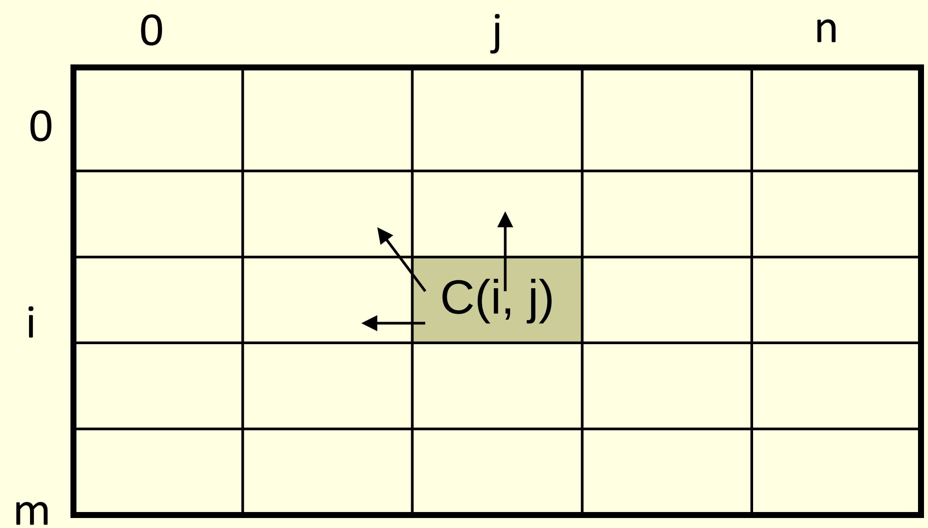
Dynamic Programming

- Analyze the problem in terms of a number of smaller subproblems.
- Solve the subproblems and keep their answers in a table.
- Each subproblem's answer is easily computed from the answers to its own subproblems.

DP Algorithm

- Key: find out the correct order to solve the sub-problems
- Total number of sub-problems: *m* * *n*

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$



DP Algorithm

LCS-Length(X, Y)

```
1. m = length(X)  // get the # of symbols in X
2. n  = length(Y) // get the # of symbols in Y
3. for i = 1 to m      c[i,0] = 0      // special case: Y[0]
4. for j = 1 to n      c[0,j] = 0      // special case: X[0]
5. for i = 1 to m                      // for all X[i]
6.      for j = 1 to n                           // for all Y[j]
7.           if ( X[i] == Y[j])
8.                c[i,j] = c[i-1,j-1] + 1
9.           else c[i,j] = max( c[i-1,j], c[i,j-1] )
10. return c
```

LCS Example

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the LCS of X and Y?

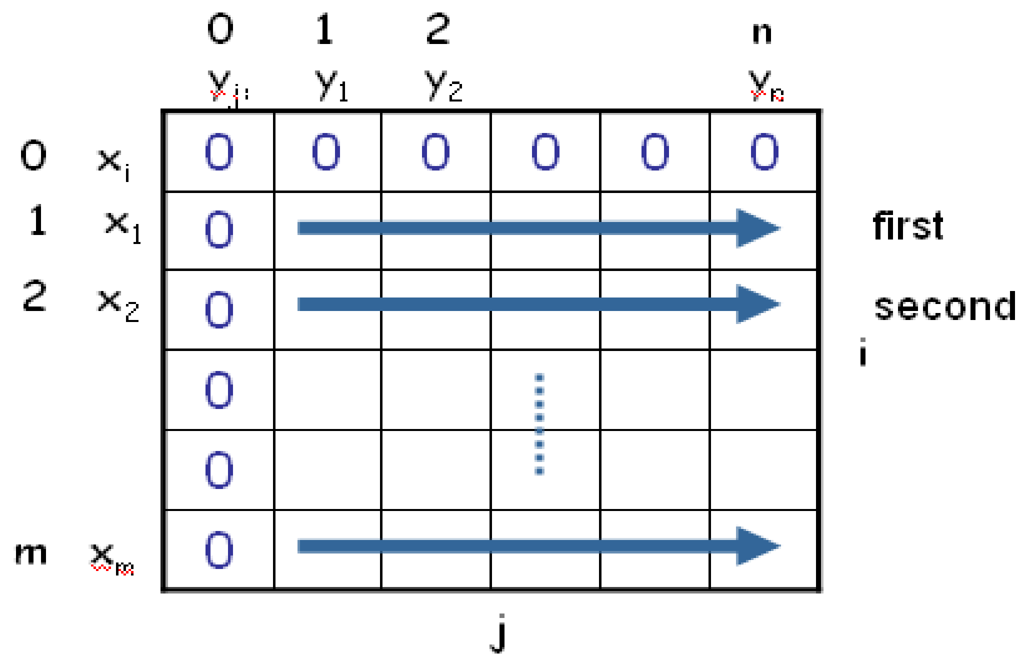
LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

Computing the Length of the LCS

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$



LCS Example (0)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Y[j]		**B**	**D**	**C**	**A**	**B**
0	X[i]							
1	**A**							
2	**B**							
3	**C**							
4	**B**							

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,6]

LCS Example (1)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m   c[i,0] = 0
for j = 1 to n   c[0,j] = 0

LCS Example (2)

ABCB
BDCAB

		j	0	1	2	3	4	5
i		Y[j]		**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**				
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (3)

ABCB
BDCAB

	i		j	0	1	2	3	4	5
				Y[j]	**B**	**D**	**C**	**A**	**B**
	0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
	1	**A**		**0**	**0**	**0**	**0**		
	2	**B**		**0**					
	3	**C**		**0**					
	4	**B**		**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (4)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (5)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	$\rightarrow$ **1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (6)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**				
3	**C**		**0**					
4	**B**		**0**					

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (7)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (8)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**					
4	**B**		**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (9)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**			
4	**B**	**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (10)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (11)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (12)

ABCB

BDCAB

		j	0	1	2	3	4	5
i			Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**				

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (13)

ABCB
BDCAB

i	X[i]	j: 0 Y[j]	1 B	2 D	3 C	4 A	5 B
0		0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (14)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(m*n)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

How to find actual LCS

- The algorithm just found the *length* of LCS, but not LCS itself.
- How to find the actual LCS?
- For each c[i,j] we know how it was acquired:

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- A match happens only when the first equation is taken
- So we can start from *c[m,n]* and go backwards, remember *x[i]* whenever *c[i,j] = c[i-1, j-1]+1*.

2	2
2	3

For example, here
c[i,j] = c[i-1,j-1] +1 = 2+1=3

Finding LCS

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1** ←	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2** ←	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

Time for trace back: $O(m+n)$.

Finding LCS (2)

	j	0	1	2	3	4	5
i		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

LCS (reversed order): **B C B**

LCS (straight order): **B C B**

(this string turned out to be a palindrome)

Compute Length of an LCS

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
      do c[i, 0] ← 0
for j ← 0 to n
      do c[0, j] ← 0
for i ← 1 to m
      do for j ← 1 to n
               do if x_i = y_j
                     then c[i, j] ← c[i − 1, j − 1] + 1
                          b[i, j] ← "↖"
                     else if c[i − 1, j] ≥ c[i, j − 1]
                             then c[i, j] ← c[i − 1, j]
                                  b[i, j] ← "↑"
                             else c[i, j] ← c[i, j − 1]
                                  b[i, j] ← "←"
return c and b
```

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A		↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	←	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	←	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑

c table

(represent b table)

source: 91.503 textbook Cormen, et al.

Construct an LCS

```
PRINT-LCS(b, X, i, j)
if i = 0 or j = 0
    then return
if b[i, j] = "↖"
    then PRINT-LCS(b, X, i − 1, j − 1)
         print x_i
elseif b[i, j] = "↑"
    then PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

For the b table in Figure 15.6, this procedure prints "$BCBA$." The procedure takes time $O(m+n)$, since at least one of i and j is decremented in each stage of the recursion.

```
LCS-Length(X, Y)   // dynamic programming solution
    m = X.length()
    n = Y.length()
    for i = 1 to m do  c[i,0] = 0
    for j = 0 to n do  c[0,j] = 0                         O(nm)
    for i = 1 to m do  // row
          for j = 1 to n do // cloumn
                if x_i = =y_i then
                          c[i,j] = c[i-1,j-1] + 1
                          b[i,j] ="↖"
                else if c[i-1, j] ≥ c[i,j-1] then
                          c[i,j] = c[i-1,j]
                          b[i,j] = "^"
                else      c[i,j] = c[i,j-1]
                          b[i,j] = "<"
```

	j	0	1	2	3	4	5	6
i		y_j	**B**	**D**	**C**	**A**	**B**	**A**
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**						
2	**B**	**0**						
3	**C**	**0**						
4	**B**	**0**						
5	**D**	**0**						
6	**A**	**0**						
7	**B**	**0**						

First Optimal-LCS initializes row 0 and column 0

	j	0	1	2	3	4	5	6
i		***y_j***	***B***	***D***	***C***	***A***	***B***	***A***
0	***x_i***	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	***A***	**0**	**^ 0**	**^ 0**	**^ 0**	**↖ 1**	**< 1**	**1**
2	***B***	**0**	**^ 1**	**< 1**	**< 1**	**^ 1**	**↖ 2**	**< 2**
3	***C***	**0**	**^ 1**	**^ 1**	**↖ 2**			
4	***B***	**0**						
5	***D***	**0**						
6	***A***	**0**						
7	***B***	**0**						

Next each c[*i*, *j*] is computed, row by row, starting at c[1,1].

If $x_i == y_j$ then c[i, j] = c[i-1, j-1]+1 and b[i, j] = ↖

	j	0	1	2	3	4	5	6
i		y_j	*B*	*D*	*C*	*A*	*B*	*A*
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	*A*	**0**	**^ 0**	**^ 0**	**^ 0**	**↖ 1**	**< 1**	**1**
2	*B*	**0**	**^ 1**	**< 1**	**< 1**	**^ 1**	**↖ 2**	**< 2**
3	*C*	**0**	**^ 1**	**^ 1**	**↖ 2**	**< 2**		
4	*B*	**0**						
5	*D*	**0**						
6	*A*	**0**						
7	*B*	**0**						

If x_i <> y_j then c[i, j] = max(c[i-1, j], c[i, j-1]) and b[i, j] points to the larger value

	j	0	1	2	3	4	5	6
i		y_j	**B**	**D**	**C**	**A**	**B**	**A**
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**^ 0**	**^ 0**	**^ 0**	**↖ 1**	**< 1**	**1**
2	**B**	**0**	**^ 1**	**< 1**	**< 1**	**^ 1**	**↖ 2**	**< 2**
3	**C**	**0**	**^ 1**	**^ 1**	**↖ 2**	**< 2**	**^ 2**	
4	**B**	**0**						
5	**D**	**0**						
6	**A**	**0**						
7	**B**	**0**						

if c[i-1, j] == c[i, j-1]
then b[i,j] points up

	j	0	1	2	3	4	5	6
i		y_j	**B**	**D**	**C**	**A**	**B**	**A**
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**^ 0**	**^ 0**	**^ 0**	**↖ 1**	**< 1**	**↖ 1**
2	**B**	**0**	**^ 1**	**< 1**	**< 1**	**^ 1**	**↖ 2**	**< 2**
3	**C**	**0**	**^ 1**	**^ 1**	**↖ 2**	**< 2**	**^ 2**	**^ 2**
4	**B**	**0**	**↖ 1**	**^ 1**	**^ 2**	**^ 2**	**↖ 3**	**< 3**
5	**D**	**0**	**^ 1**	**↖ 2**	**^ 2**	**^ 2**	**^ 3**	**^ 3**
6	**A**	**0**	**^ 1**	**^ 2**	**^ 2**	**↖ 3**	**^ 3**	**↖ 4**
7	**B**	**0**	**↖ 1**	**^ 2**	**^ 2**	**^ 3**	**↖ 4**	**^ 4**

To construct the LCS, start in the bottom right-hand corner and follow the arrows. A ↖ indicates a matching character.

	j	0	1	2	3	4	5	6
i		y_j	***B***	***D***	***C***	***A***	***B***	***A***
0	x_i	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	***A***	**0**	**^ 0**	**^ 0**	**^ 0**	**↖ 1**	**< 1**	**↖ 1**
2	***B***	**0**	**^ 1**	**< 1**	**< 1**	**^ 1**	**↖ 2**	**< 2**
3	***C***	**0**	**^ 1**	**^ 1**	**↖ 2**	**< 2**	**^ 2**	**^ 2**
4	***B***	**0**	**↖ 1**	**^ 1**	**^ 2**	**^ 2**	**↖ 3**	**< 3**
5	***D***	**0**	**^ 1**	**↖ 2**	**^ 2**	**^ 2**	**^ 3**	**^ 3**
6	***A***	**0**	**^ 1**	**^ 2**	**^ 2**	**↖ 3**	**^ 3**	**↖ 4**
7	***B***	**0**	**↖ 1**	**^ 2**	**^ 2**	**^ 3**	**↖ 4**	**^ 4**

LCS: **B C B A**

Constructing an LCS

Print-LCS(b,X,i,j)

if i = 0 or j = 0 then

return

if b[i,j] = “↖” then

Print-LCS(b, X, i-1, j-1)

print x_i

else if b[i,j] = “^” then

Print-LCS(b, X, i-1, j)

else Print-LCS(b, X, i, j-1)