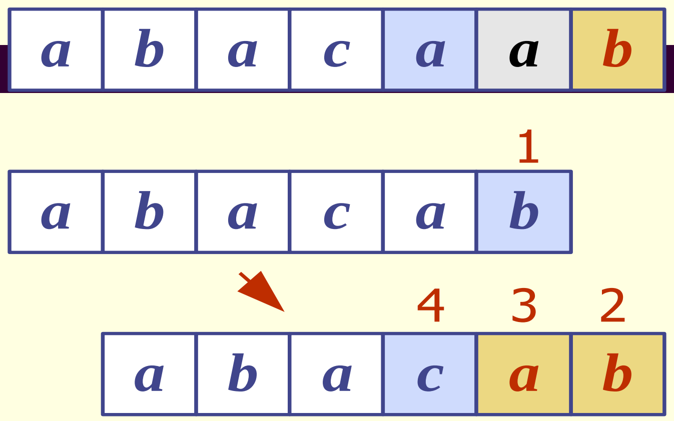
String/Pattern Matching

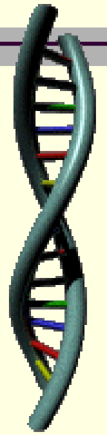


Contents

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* $\quad \Sigma = \{a,g,c,t\}$

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"

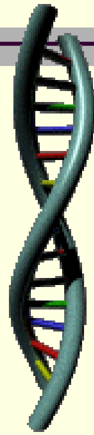
Why String Matching?

- **Applications in Computational Biology**
  - DNA sequence is a long word (or text) over a 4-letter alphabet
  - GTTTGAGTGGTCAGTCTTTTCGTTTCGACGGAGCCCCCAATTAATAAACTCATAAGCAGACCTCAGTTCGCTTAGAGCAGCCGAAA…..
  - 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* $\Sigma = \{a,g,c,t\}$

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

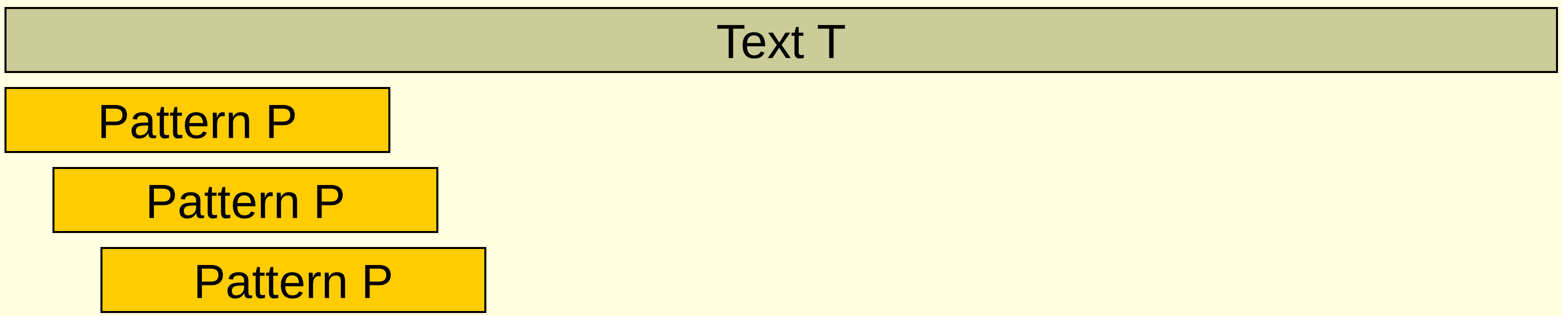
String Matching

```
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

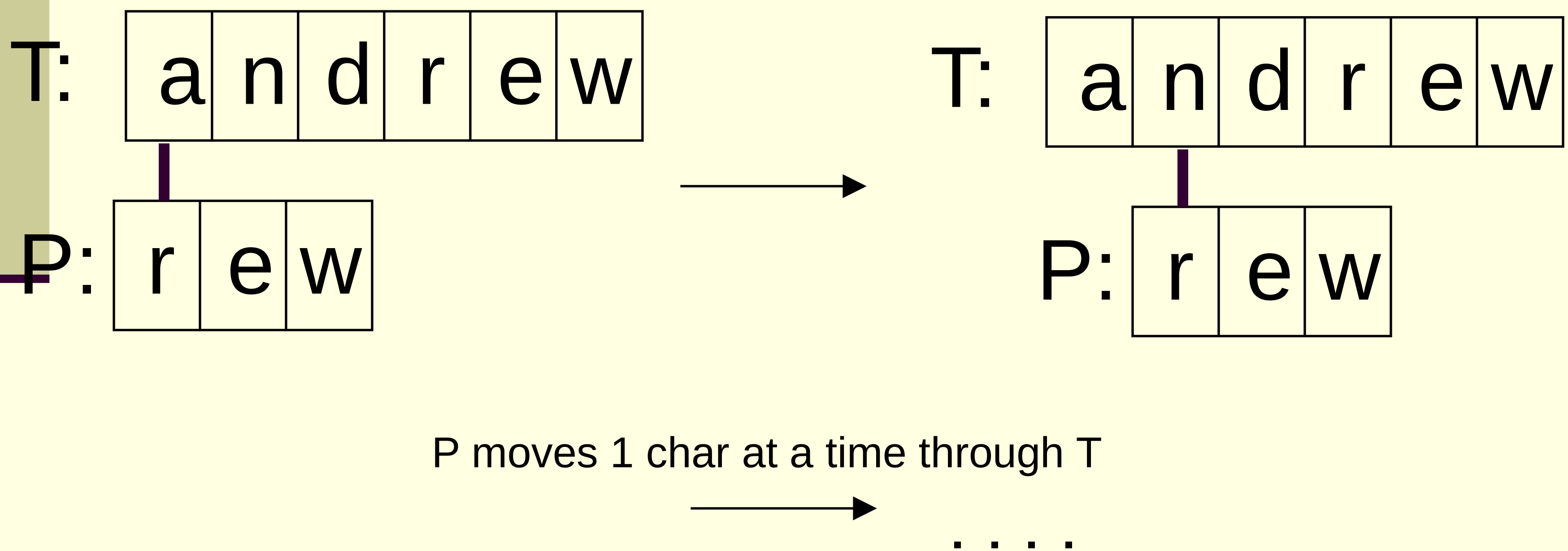


Compare until a match is found. If so return the index where match occurs

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



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

```
00000000000000001

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

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

A bad case

```
00000000000000001

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≤j≤2

Contents

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:

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

$$x(i+1) = x(i) \cdot b \qquad \text{Shift left one digit}$$

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

$$+\ t[i+M] \qquad \text{Add new rightmost digit}$$

- 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

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"

x: A B C B D A B

y: B D C A B A

BCBA = LCS(x, y)

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

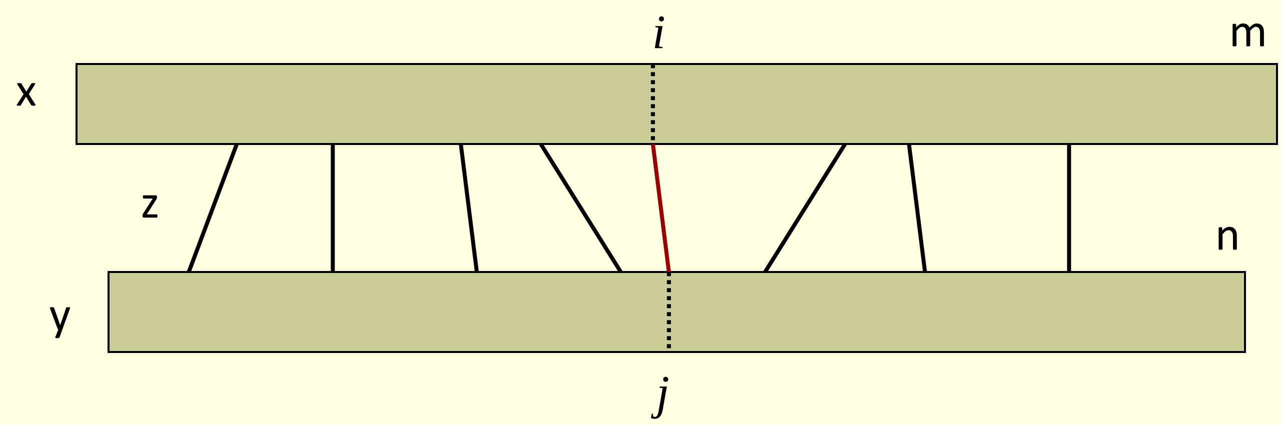
- Key: optimal substructure and overlapping sub-problems
- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.

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($n2^m$).**

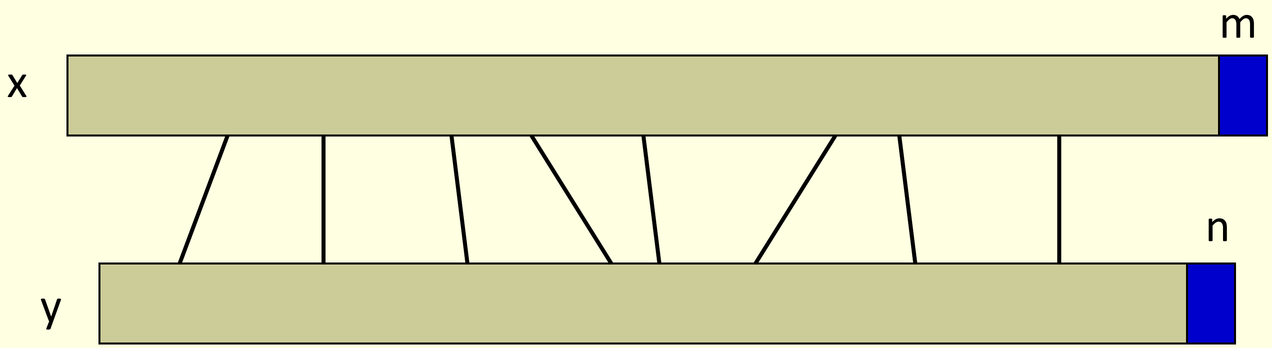
Optimal substructure

- 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.



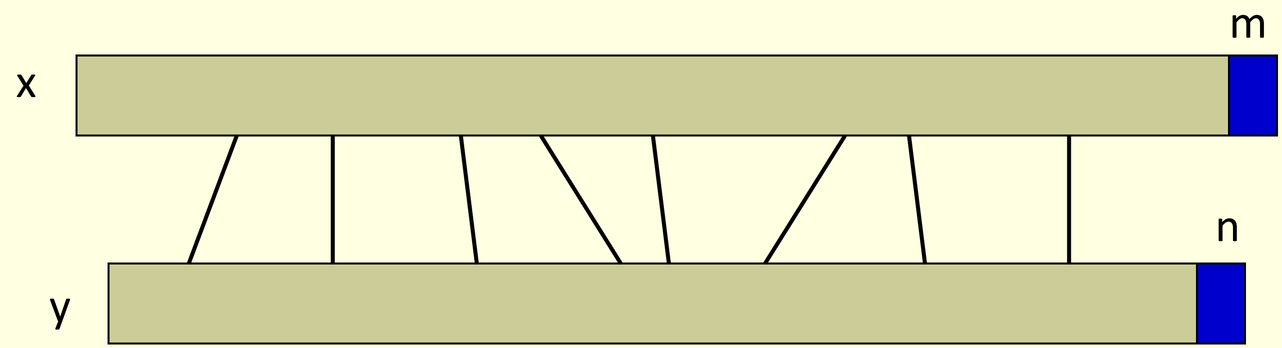
- Subproblems: "find LCS of pairs of *prefixes* of x and y"

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

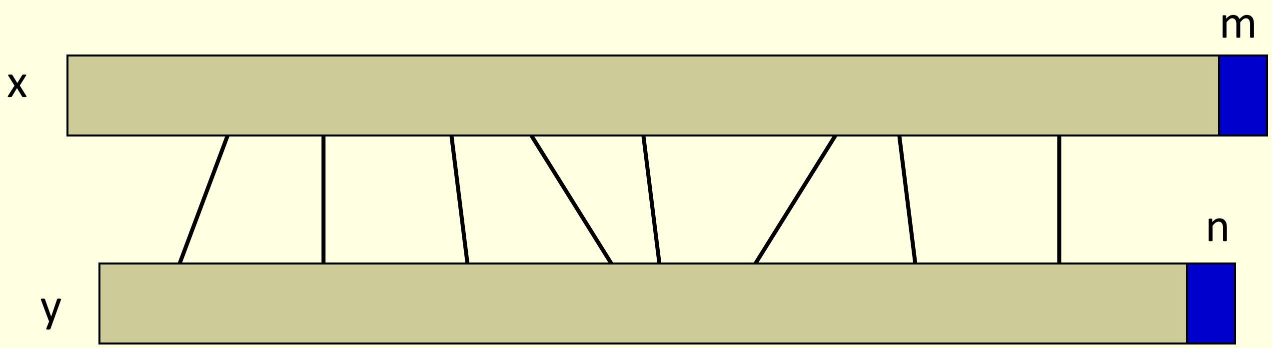


- Case 1: $x[m]=y[n]$ Reduce both sequences by 1 char
  - $LCS(x, y) = LCS(x[1..m-1], y[1..n-1]) \mid\mid x[m]$
- Case 2: $x[m] \neq y[n]$ concatenate
  - $LCS(x, y) = LCS(x[1..m-1], y[1..n])$ or

    $LCS(x[1..m], y[1..n-1])$, 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 $x[m] = y[n]$

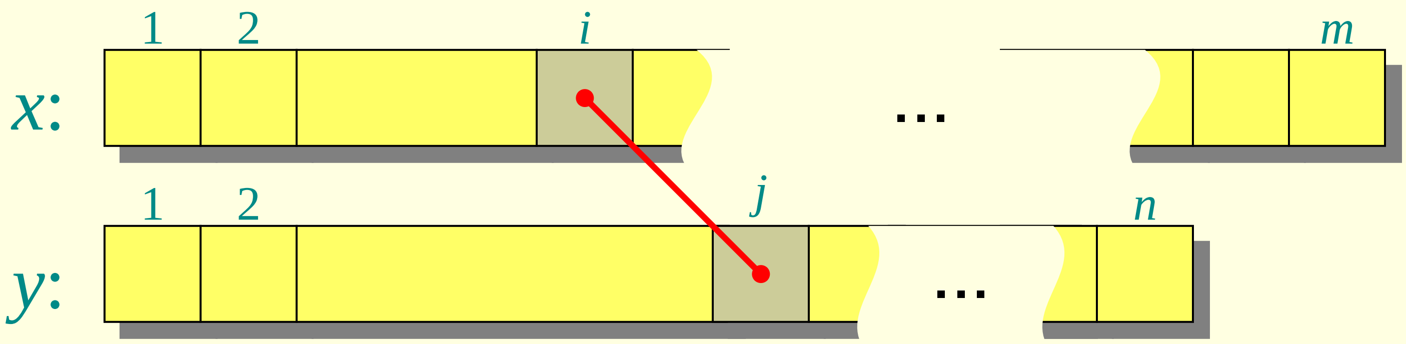
  $c[m, n] = c[m-1, n-1] + 1$

- If $x[m] \mathrel{!}= y[n]$

  $c[m, n] = max \{ c[m-1, n], c[m, n-1] \}$

Generalize: recursive formulation

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



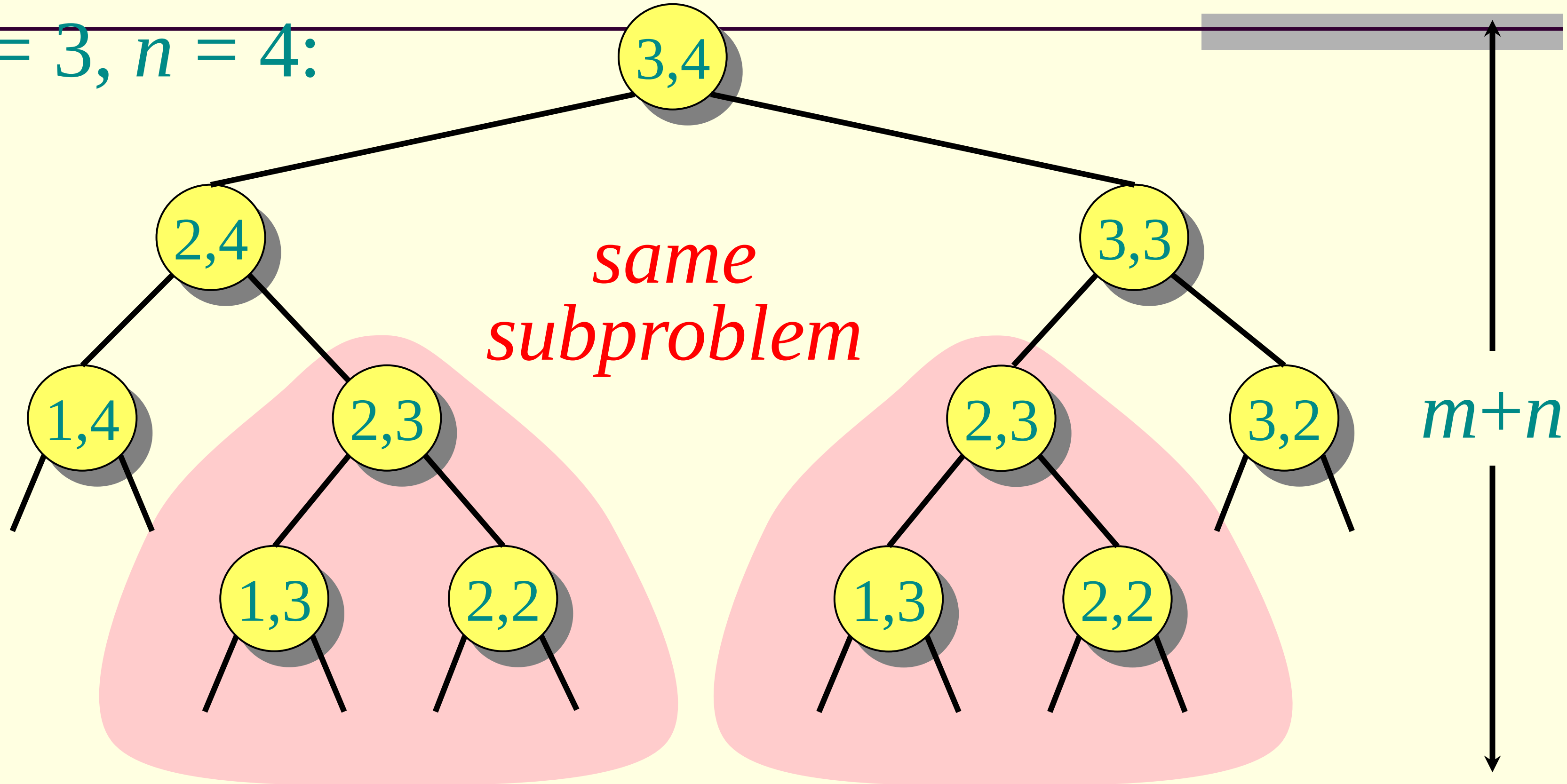
Recursive algorithm for LCS

$$
\begin{aligned}
&\text{LCS}(x, y, i, j)\\
&\quad \textbf{if } x[i] = y[j]\\
&\qquad \textbf{then } c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1\\
&\qquad \textbf{else } c[i, j] \leftarrow \max\{\text{LCS}(x, y, i-1, j),\\
&\qquad\qquad\qquad\qquad\qquad \text{LCS}(x, y, i, j-1)\}
\end{aligned}
$$

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!

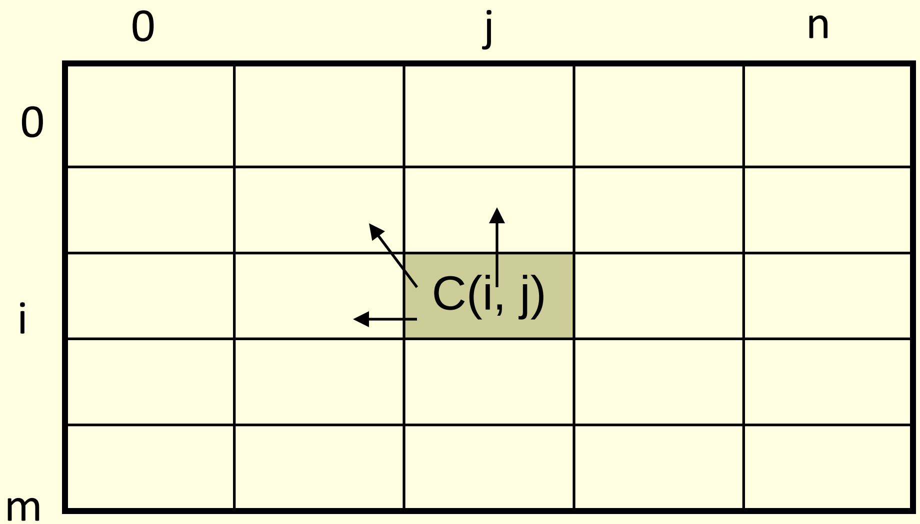
Dynamic Programming

- 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?

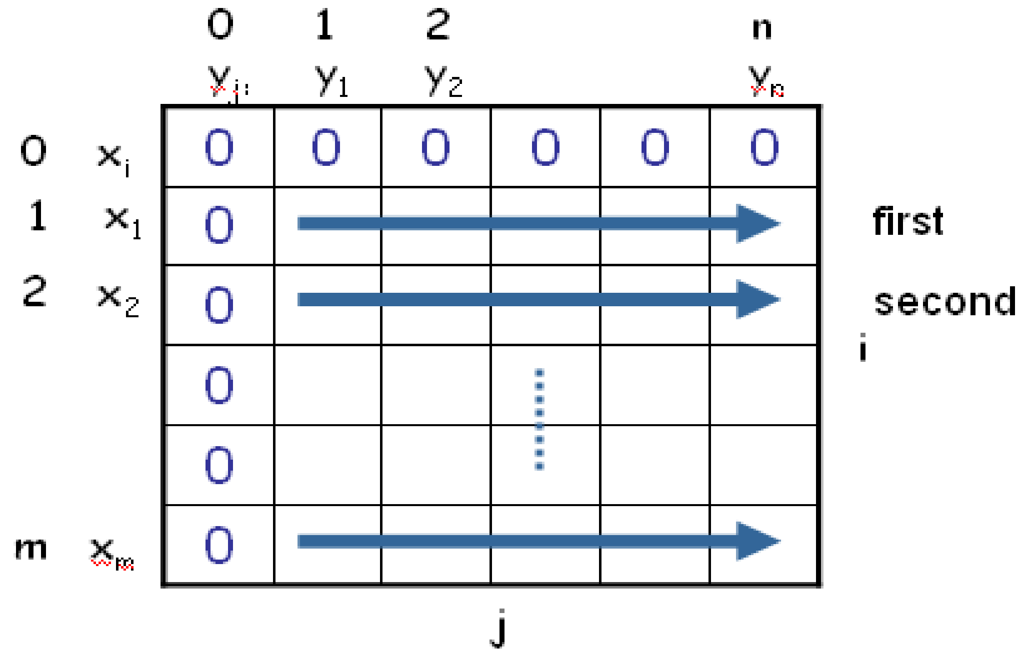
LCS(X, Y) = BCB

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

	j	0	1	2	3	4	5
i		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

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**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

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

	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**		
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**	**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

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**	**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

		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**		

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	↑ 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

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)