CSE 6140/ CX 4140:

Computational Science and Engineering ALGORITHMS

Instructor: Anne Benoit

Visiting Associate Professor, CSE

Based on slides by Bistra Dilkina

Dynamic Programming

1) Show problem has optimal substructure: the optimal solution can be constructed from optimal solutions to subproblems (recurrence relation).

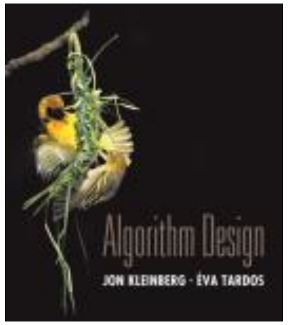
2) Show subproblems are overlapping, i.e. subproblems may be encountered many times but the total number of distinct subproblems is polynomial

3) Construct an algorithm that computes the optimal solution to each subproblem only once, and reuses the stored result all other times

4) Show that time and space complexity is polynomial

[KT 6.1]

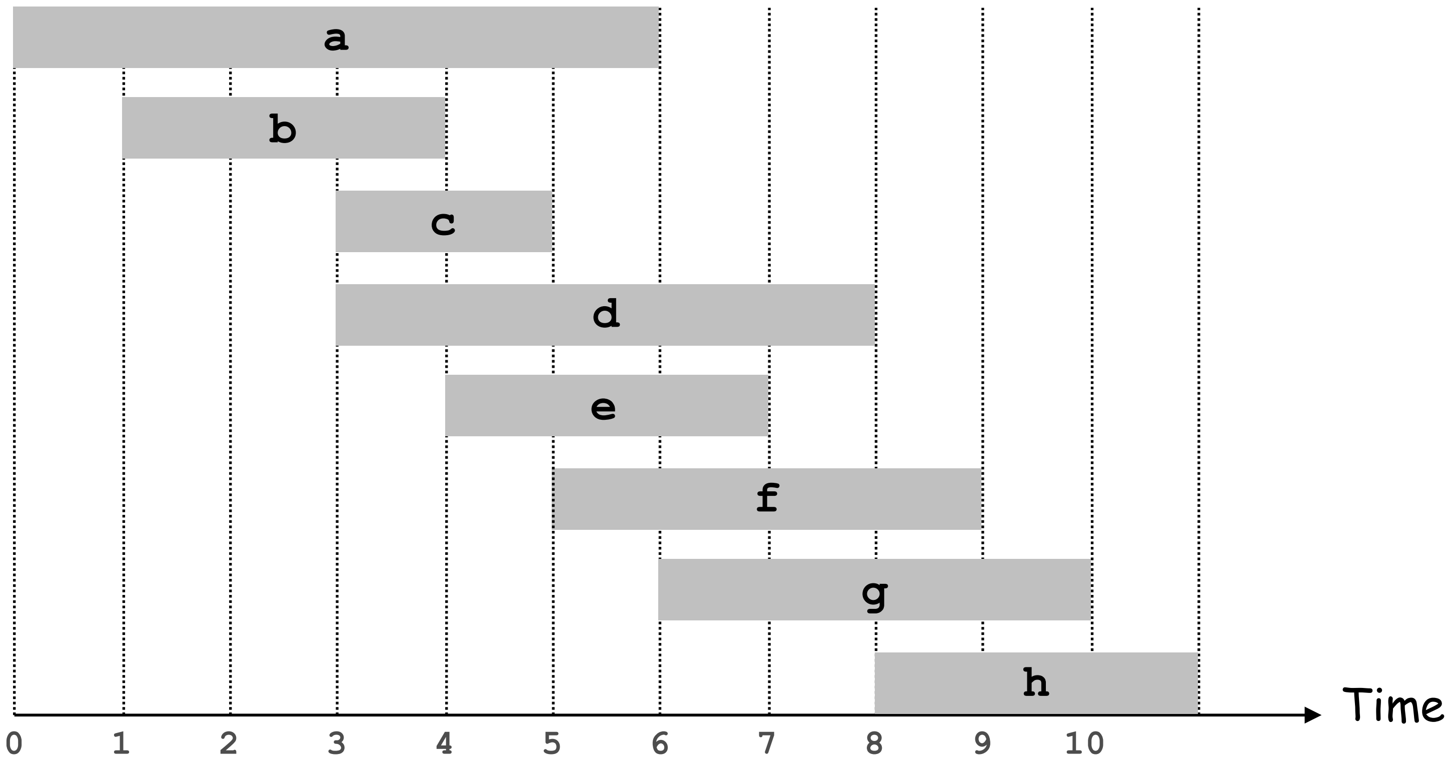
WEIGHTED INTERVAL SCHEDULING



Weighted Interval Scheduling

Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has weight or value v_j .
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

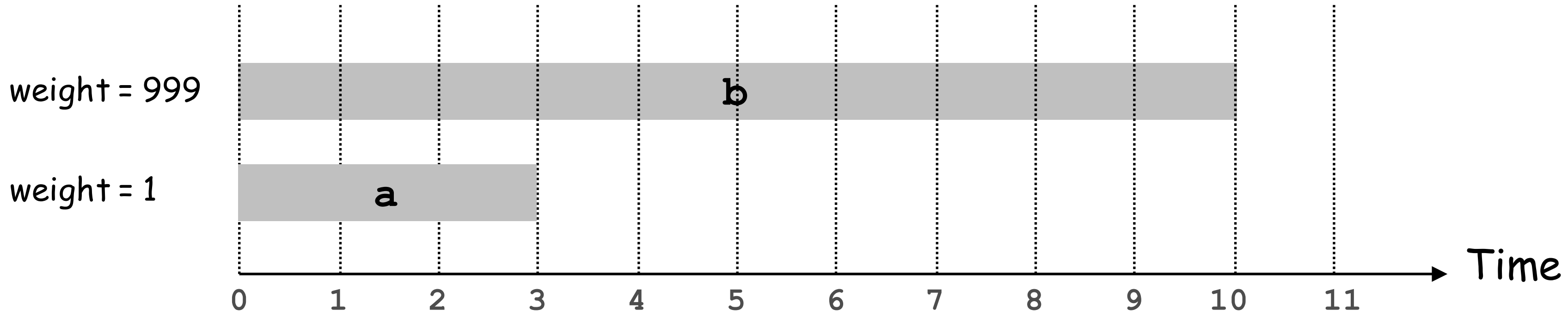


Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all weights are 1.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

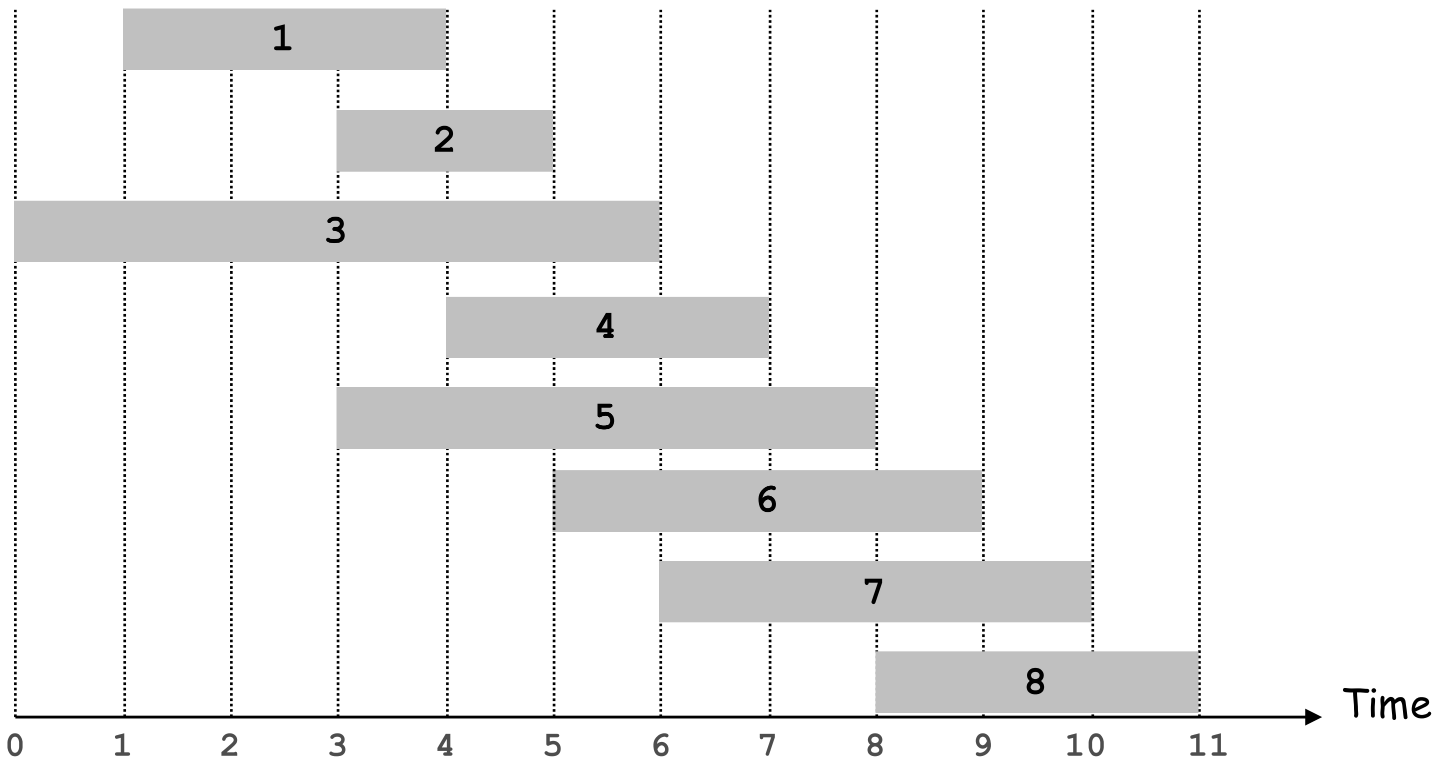
Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.



Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



Dynamic Programming: Binary Choice

Consider an optimal solution O for the jobs {1,..,n}

No matter what O is, what can we say about the job n?

- Either O contains the last job n (Case 1)
- Or O does not contain the last job n (Case 2)

This covers all possible cases for O

Case 1: O contains job n: what can we say about the remaining part of the solution O-{n}?

- O-{n} cannot contain any job that is incompatible with n, i.e. cannot contain any job in p(n) +1,.., n-1, i.e. it only contains jobs in {1,..,p(n)}
- Since O is feasible, O-{n} is a feasible solution for the problem of scheduling {1,…,p(n)}
- More importantly 0-{n} must be an optimal solution for scheduling {1,…,p(n)}. If not, then we could take the optimal solution for {1,..,p(n)} and safely add job n to it, and obtain an overall solution O' better than the given optimal solution O

Dynamic Programming: Binary Choice

Consider an optimal solution O for the jobs {1,..,n}

No matter what O is, we can say that:

- Either O contains the last job n (Case 1)
- Or O does not contain the last job n (Case 2)

This covers all possible cases for O

Case 1: O contains job n

- Contains an optimal solution for scheduling {1,...,p(n)}.

Case 2: O does not contain n

- Then O is a feasible solution for scheduling {1,...,n-1}
- If O is not the optimal solution for {1,..,n-1}, we can replace it with the optimal solution for {1,...,n-1} and obtain a better solution also for scheduling {1,...,n}
- O must contain the optimal solution for scheduling {1,..,n-1}

Finding the optimal solution for {1,..,n} involves looking at optimal solutions for smaller problems of the form {1,…,j}

Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

Case 1: OPT selects job j.

Case 2: OPT does not select job j.

OPTIMAL SUBSTRUCTURE

- Case 1: OPT(j) selects job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j) with value OPT(p(j))
  - collect profit v_j from including j
  - OPT(j) = v(j) + OPT(p(j))
- Case 2: OPT(j) does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1: OPT(j) = OPT(j-1)

optimal substructure

RECURRENCE RELATION

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Brute Force

Brute force algorithm.

```
Input: n, $s_1,…,s_n$ , $f_1,…,f_n$ , $v_1,…,v_n$

  Sort jobs by finish times so that $f_1 \le f_2 \le ... \le f_n$.

  Compute p(1), p(2), …, p(n)

  Call Compute-Opt(n)



Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max($v_j$ + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

Proof this algorithm is correct

With the optimal substructure analysis we proved that:

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Claim. The algorithm Compute-Opt(j) computes correctly the optimal value for each j=1,..,n.

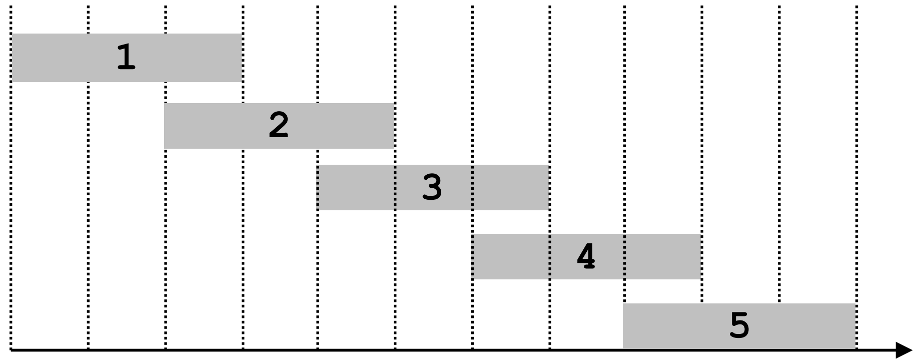
Proof. By induction on j

1) True for j=0, OPT(0) = 0
2) Assume true for all i < j
3) By induction we know OPT(j-1) and OPT(p(j)) are computed correctly
   Hence, `Compute-Opt(j) = max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1)) = max(v_j + OPT(p(j)), OPT(j-1)) = OPT(j)`

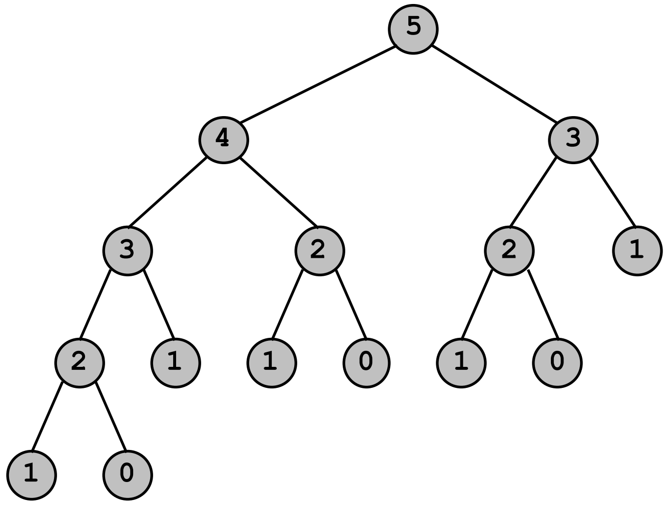
Weighted Interval Scheduling: Brute Force

Ex. What if each job is incompatible with only one earlier job, i.e. p(j) = j-2. T(n) = T(n-1) + T(n-2) + O(1) grows like Fibonacci sequence -> T(n) in $O(2^n)$.

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.



p(1) = 0, p(j) = j-2



Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, $s_1,…,s_n$ , $f_1,…,f_n$ , $v_1,…,v_n$

 Sort jobs by finish times so that $f_1 \le f_2 \le ... \le f_n$.
 Compute p(1), p(2), …, p(n)

 for j = 1 to n
    M[j] = empty      ← global array
 M[0] = 0
 M-Compute-Opt(n)


M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max($v_j$ + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

What have we done so far?

1. We showed optimal substructure property for the problem
2. Derived a recurrence relation based on the optimal substructure (with overlapping subproblems)
3. Showed total number of distinct subproblems is polynomial and designed a DP Algorithm that implements the recurrence relation and caches explored subproblems to avoid repeated work
4. Analyze Space and Time of our algorithm

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes O(n log n) time.

- Sort by finish time: O(n log n).
- Computing p(·) : O(n log n) via sorting by start time.

- `M-Compute-Opt(j)`: each invocation takes O(1) time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two recursive calls

- The running time is bound by (a constant $\times$ the number of recursive calls)
- Progress measure Φ = # nonempty entries of `M[]`.
  - initially $\Phi = 0$, throughout $\Phi \le n$.
  - (ii) increases Φ by 1 $\Rightarrow$ at most 2n recursive calls.

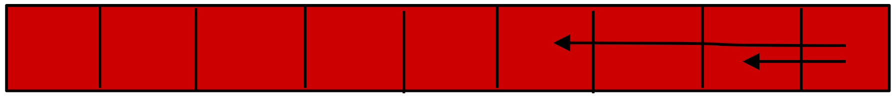
- Overall running time of `M-Compute-Opt(n)` is O(n). ▪

Remark. The overall algorithm takes O(n) if jobs are pre-sorted by start and finish times when given as input.

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

M[]



```
Input: n, $s_1,…,s_n$ , $f_1,…,f_n$ , $v_1,…,v_n$

Sort jobs by finish times so that $f_1 \le f_2 \le ... \le f_n$.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max($v_j$ + M[p(j)], M[j-1])
}
```

Dynamic Programming

Top-down DP = Memoization

- Design a *recursive* algorithm
- Store result for each subproblem when you first compute it
- Check for existing result for a subproblem, before doing any extra work

Bottom-up DP = Iterative DP

- Determine dependency between a problem and its subproblems
- Determine an order in which to compute subproblems so that you always have what you need already available
- Fill in the table of results in the determined order (*using FOR loops*)

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithm computes optimal value.
What if we want the solution itself?
A. Do some post-processing.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if ($v_j$ + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

- # of recursive calls $\leq$ n $\Rightarrow$ O(n).

LONGEST COMMON SUBSEQUENCE [CLRS 15.4]

Longest Common Subsequence (LCS)

- Given two strings/sequences
- Ex: X= {A B C B D A B }, Y= {B D C A B A}
- find Longest Common Subsequence (a sequence of letters that appears in both X and Y but not necessarily contiguously):
- Subsequence: BCBA
- X = A B C B D A B
- Y = B D C A B A

- Applications:
  - compare files – maximize number of matched common lines, the rest marked as changes
  - comparison of two DNA strings – used as a similarity measure, also equivalent to minimizing insert/delete mutations that can transform one DNA string into the other

Naive Approach to the LCS Problem

- A Brute-force solution:
  - Enumerate all subsequences of X
  - Test which ones are also subsequences of Y
  - Pick the longest one.
- Analysis:
  - if |X| = m, |Y| = n
  - there are 2^m subsequences of X
  - we must compare each with Y (n comparisons)
  - So the running time of the brute-force algorithm is $O(n\ 2^m)$
  - Exponential time!

LCS Notation

- Let X and Y be sequences.
- We denote by LCS(X, Y) the set of longest common subsequences of X and Y (set of optimal solutions).

Prefix

- Let $X = < x_1,x_2,\ldots,x_m>$ be a sequence.
- We denote by X_i the subsequence
- $X_i = < x_1,x_2,\ldots,x_i>$
- and call it the i^{th} prefix of X.

Optimal Substructure Theorem

Given two strings/sequences:

$X = < x_1,x_2,\ldots,x_m>$

$Y = < y_1,y_2,\ldots,y_n>$

Let $Z = < z_1,z_2,\ldots,z_k>$ be any optimal solution, a LCS of X and Y.

a) If $x_m = y_n$ then $x_m = y_n = z_k$
   - Z_{k-1} is in LCS(X_{m-1} , Y_{n-1})

b) Else $x_m \neq y_n$
   b) Either $x_m \neq z_k$
      - Z is in LCS(X_{m-1} , Y)

   c) Or $x_m = z_k$ and hence $y_n \neq z_k$
      - Z is in LCS(X, Y_{n-1})

Overlapping Subproblems

- Algorithm idea
- If $x_m = y_n$ then find solution to LCS(X_{m-1} , Y_{n-1}) and append x_m
- If $x_m \neq y_n$ then find a solution for each of the two subproblems LCS(X_{m-1} , Y_n) and LCS(X_m , Y_{n-1}), and choose the longer one

- Overlap: notice that LCS(X_{m-1} , Y_{n-1}) can appear as a subproblem when solving LCS(X_{m-1} , Y_n) and LCS(X_m , Y_{n-1})
- Small number of distinct subproblems: only $m \times n$ possible scenarios for the prefixes

Recursive Solution

- Let X and Y be sequences with lengths m and n.
- Let c[i,j] be the length of the solution to LCS(X_i, Y_j).
- What is the base case?
- c[i,j] =

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

- Looking for c[m,n]

Solving LCS with Recursion

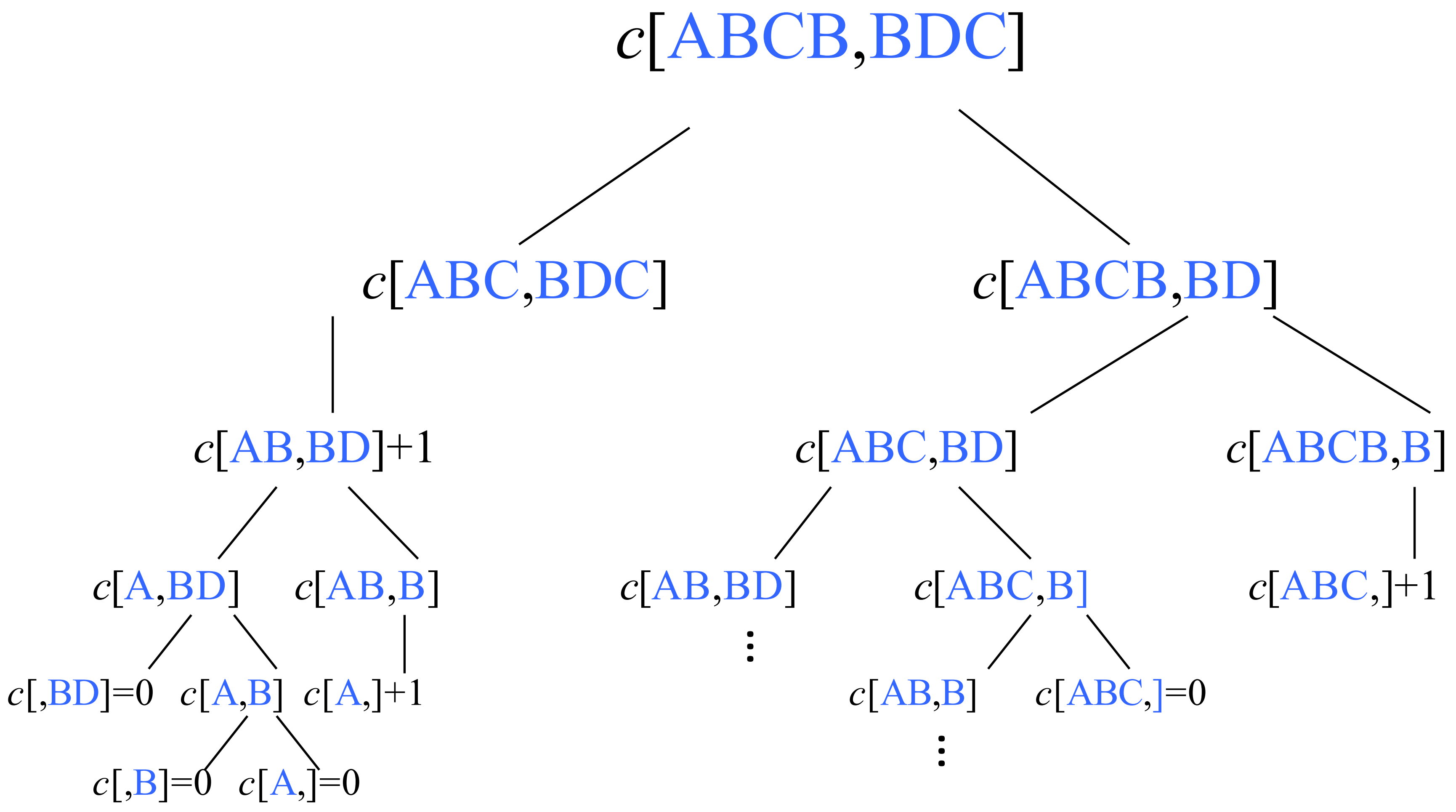
$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$

```
c($i$, $j$):
    if $i$=0 OR $j$=0: return 0                  // empty input sequence
    else if $x_i$=$y_j$: f = c($i$-1, $j$-1)+1       // match
    else f = max{$c$($i$, $j$-1), $c$($i$-1, $j$)}      // no match
    return f

return c($m$,$n$)
```

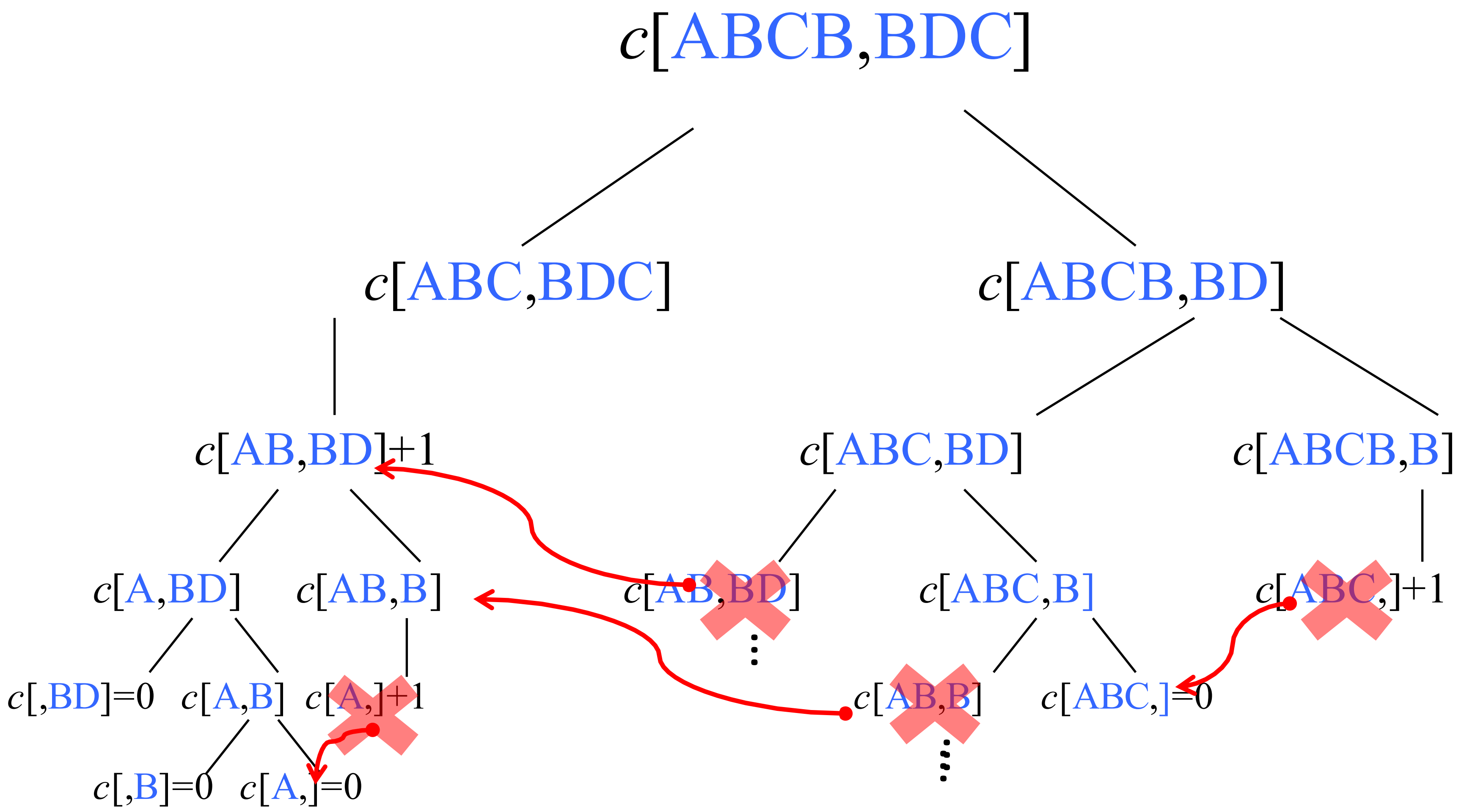
Solving LCS with Recursion

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



Solving LCS with Recursion

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



Solving LCS with Recursion+Memoization

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

```
memo = { }
c($i$, $j$):
    if ($i$, $j$) in memo: return memo[$i$, $j$]
    else if $i$=0 OR $j$=0: return 0
    else if $x_i$=$y_j$: f = c($i$-1, $j$-1)+1
    else f = max{$c$($i$, $j$-1), $c$($i$-1, $j$)}
    memo[$i$, $j$]=f
    return f
return c($m$,$n$)
```

Solving LCS with Recursion+Memoization

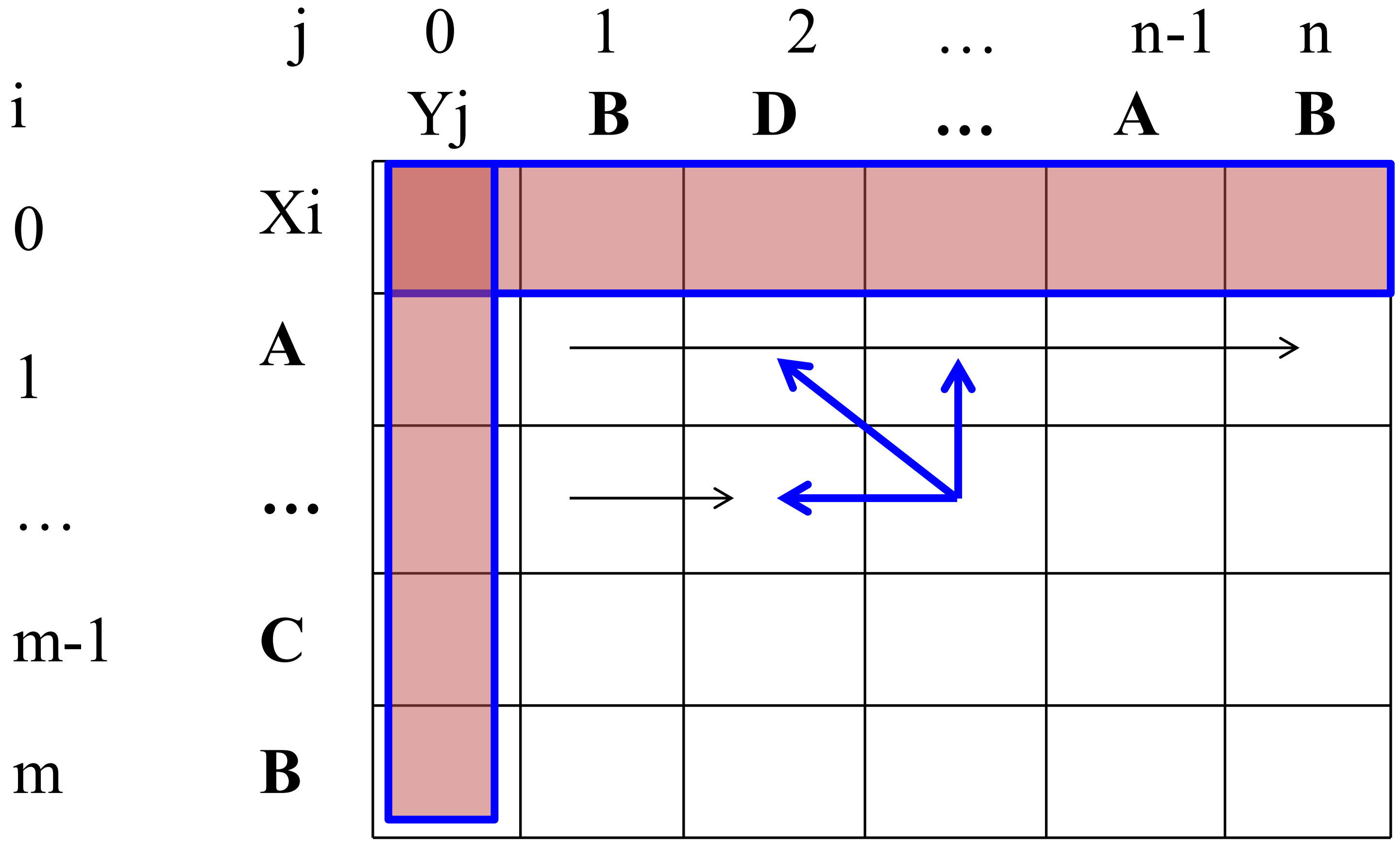
- Each subproblem c[i,j] is computed only once, at most two recursive calls per c[i,j] filled, and each call takes constant time
- There are at most m×n subproblems -> Time and space is O(m×n)

- This is an example of Top-down Dynamic Programming
  - Also known as Memoization

- How about bottom-up DP? No recursion!
  - Start with smallest problems first, and store results
  - Every time you solve a problem, you already have the solutions of the needed subproblems

LCS subproblem order

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



Allocate array c[m+1,n+1]

LCS Length Algorithm

```
LCS-Length(X, Y)
0. m = length(X)  // get the # of symbols in X
1. n  = length(Y) // get the # of symbols in Y
2. allocate matrix c of size (m+1)x(n+1)
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 $X_i$ (rows)
6.     for j = 1 to n                       // for all $Y_j$ (columns)
7.             if ( Xi == Yj )
8.                     c[i,j] = c[i-1,j-1] + 1 // match
9.             else c[i,j] = max( c[i-1,j], c[i,j-1] )
10. return c
```

Analysis of LCS Algorithm

- We have two nested loops
  - The outer one iterates n times
  - The inner one iterates m times
  - A constant amount of work is done inside each iteration of the inner loop
  - Thus, the total running time is O(nm)
- Answer is contained in c[m,n] (and the actual subsequence can be recovered from the c table) (Don’t forget to specify this in your problem answers)

LCS Example

- Inputs:
- X = ABCB
- Y = BDCAB

What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB

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

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

LCS Example (0)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

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

ABCB
BDCAB

LCS Example (1)

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**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
```

ABCB
BDCAB

LCS Example (2)

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**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 $c[i,j] = \max( c[i-1,j], c[i,j-1] )$
```

ABCB

BDCAB

LCS Example (3)

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (4)

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (5)

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	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])

ABCB
BDCAB

LCS Example (6)

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	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 $c[i,j] = \max( c[i-1,j], c[i,j-1] )$
```

ABCB
BDCAB

LCS Example (7)

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB

BDCAB

LCS Example (8)

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	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 $c[i,j] = \max( c[i-1,j], c[i,j-1] )$
```

ABCB
BDCAB

LCS Example (10)

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (11)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**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])

ABCB
BDCAB

LCS Example (12)

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**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**	$\rightarrow$ **2**	$\rightarrow\downarrow$ **2**
4	**B**	**0**					

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

ABCB
BDCAB

LCS Example (13)

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (14)

		j	0	1	2	3	4	5
	i		Yj	B	D	C	A	B
	0	Xi	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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (15)

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	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 $c[i,j] = \max( c[i-1,j], c[i,j-1] )$
```

ABCB
BDCAB