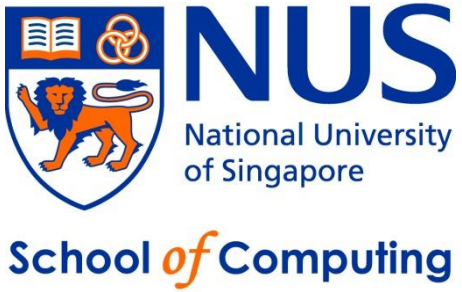
CS2010 – Data Structures and Algorithms II

Lecture 11 –Traveling Salesman + More DP

chongket@comp.nus.edu.sg



Announcements (1)

- Reminder that Online Quiz 2 is tomorrow @ Lab
- Written Quiz 2 is during next lecture, 8 Nov
  - Short quiz of 40~45 mins
  - Open book
  - **Please bring phone/laptop/tablet that will allow you internet access**

Announcements (2)

- PS5 deadline this Friday 11:59pm
- PS6, the last PS will be out this Friday 12 noon
  - Only 2 subtask, A and B
  - Need what you learn this week to solve subtask A. Subtask B can be solved with what you have learned or what you will learn next week
  - Deadline is **Wednesday 15th Nov** 11:59pm

Outline

- DP Algorithms on (Implicit) DAG using the simple TSP (Traveling Salesman Problem) as the underlying theme
  - Higher dimensional DP: Conversion of a general graph into a DAG with an addition of one (or more) extra parameter(s)
- Complete search technique using the classic TSP
  - Complete search on the general graph
  - Can be converted into a DAG ….
- Another Higher dimensional DP Problem
  - LCS (Longest Common Subsequence) between two sequences A and B

Reference: CP3 Section 2.2, 3.5, and 4.7.1

What is the |LIS| of X = {8, 3, 6, 4, 5, 7, 7}?

LIS = Longest Increasing Subsequence

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7

What is the |L**ND**S| of X = {8, 3, 6, 4, 5, 7, 7}?

ND = Non Decreasing

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7

How many distinct paths to go from (0, 0) to (3, 3) if you can only go **down** or **right** at every cell?

1. 6
2. 10
3. 20
4. 40
5. $\propto$

(0, 0)	(0, 1)	(0, 2)	(0, 3)
(1, 0)	(1, 1)	(1, 2)	(1, 3)
(2, 0)	(2, 1)	(2, 2)	(2, 3)
(3, 0)	(3, 1)	(3, 2)	(3, 3)

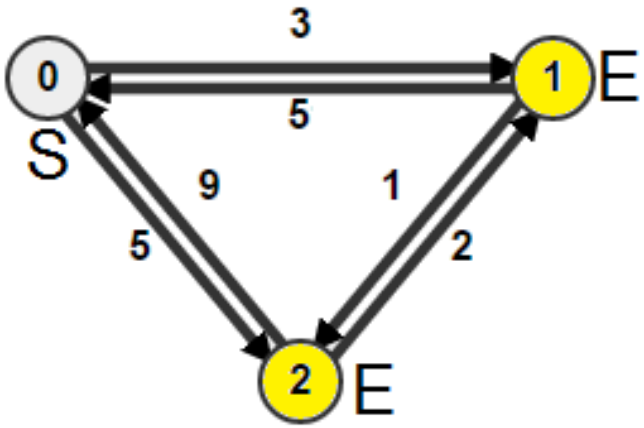
UVa 10702 – Traveling Salesman

TRAVELING SALESMAN PROBLEM (THE SIMPLER EXAMPLE)

Motivating Problem

(UVa 10702 – Traveling Salesman)

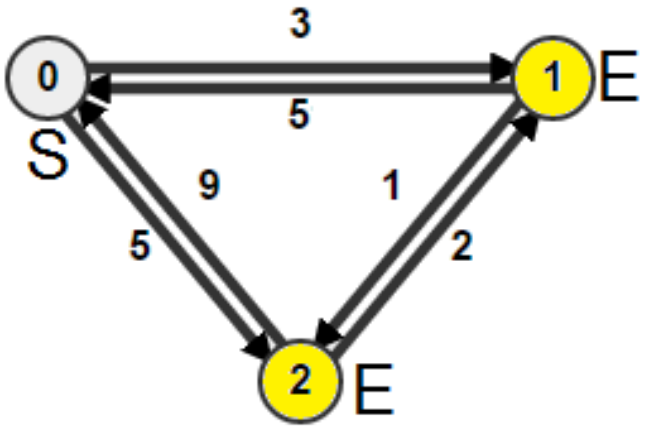
- There are **C** cities
- A salesman starts his sales tour from city **S**
  and can end his sales tour at any city labeled with **E**
- He wants to visit many cities to sell his goods
- Every time he goes from city **U** to city **V**, he gets **profit[U][V]**
  - profit[U][U] (i.e. staying at the same city) is **always 0**… ☹
- What is the maximum profit that he can get?



profit[U][V] is shown as
the weight of edge(U, V)

What is the maximum profit that he can get?

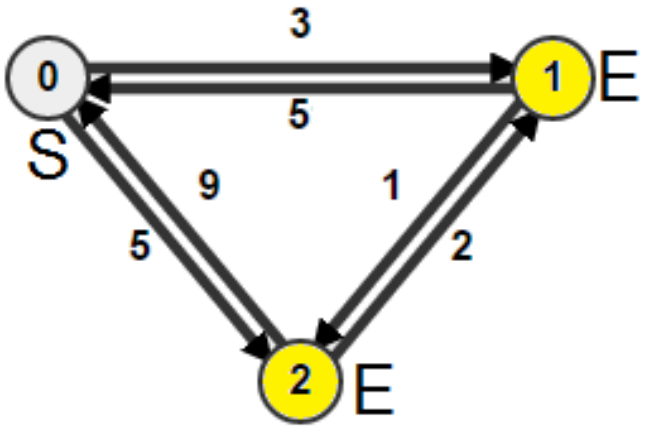
1. 3
2. 5
3. 7
4. 17
5. ∞



Reducing to SS Longest (non simple) Path Problem <u>but on General Graph</u>

This is a problem of finding
the **longest (non simple) path** on **general graph** :O

- General graph has something that does not exist in DAG discussed in previous lecture: **Cycle(s)**
- There are several **positive** weight cycles in this graph, e.g. 0→1→2→0 (weight 13); 0→1→0 (weight 8), etc
  - The salesman can keep re-visiting these cycles to get more $$:O



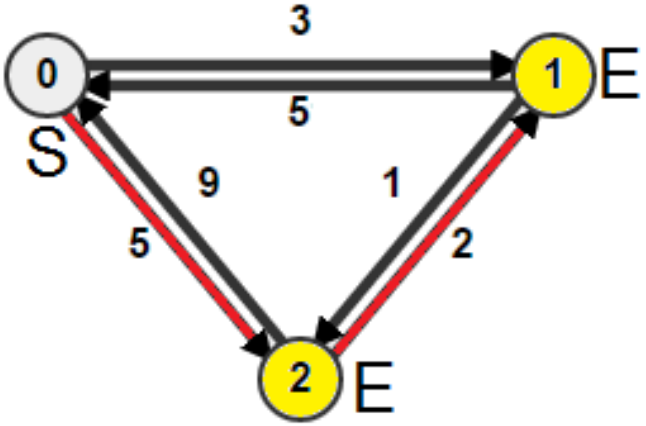
A Note About Longest Path Problem on General Graph

The longest **(non simple)** path from city **S** (vertex 0) to any city with label **E** will have $\propto$ weight

- One can go through any **positive weight cycle** to obtain $\propto$

The longest **(simple)** path from city **S** (vertex 0) to any city with label **E** is: 0→2→1 with weight 5+2 = 7

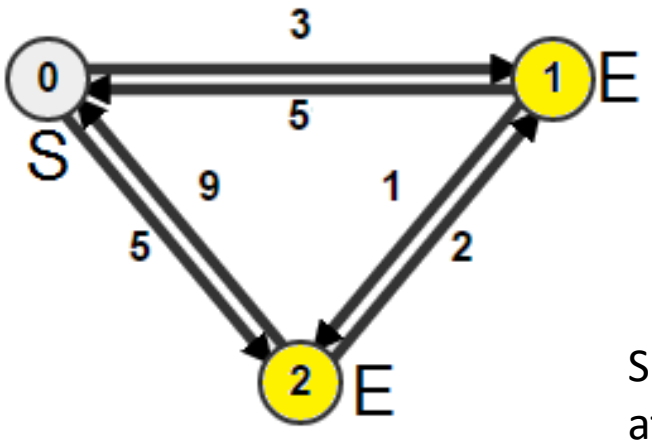
- But this is *hard* and requires "backtracking", as shown later



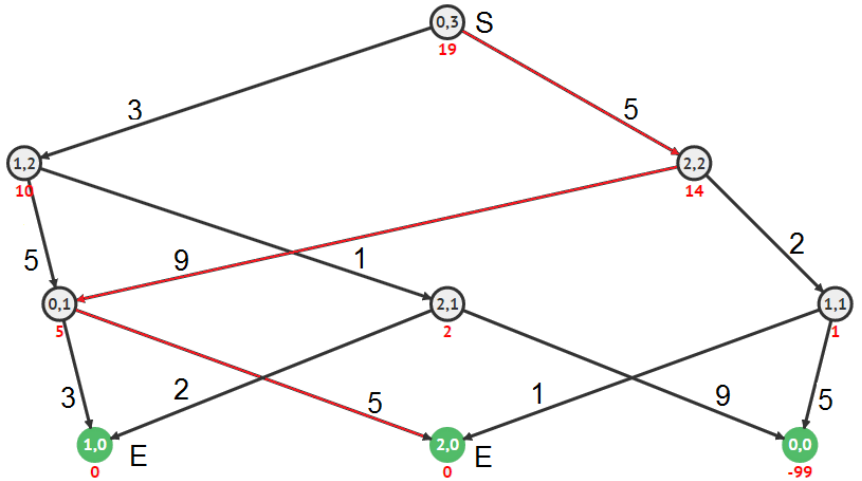
Conversion to a DAG

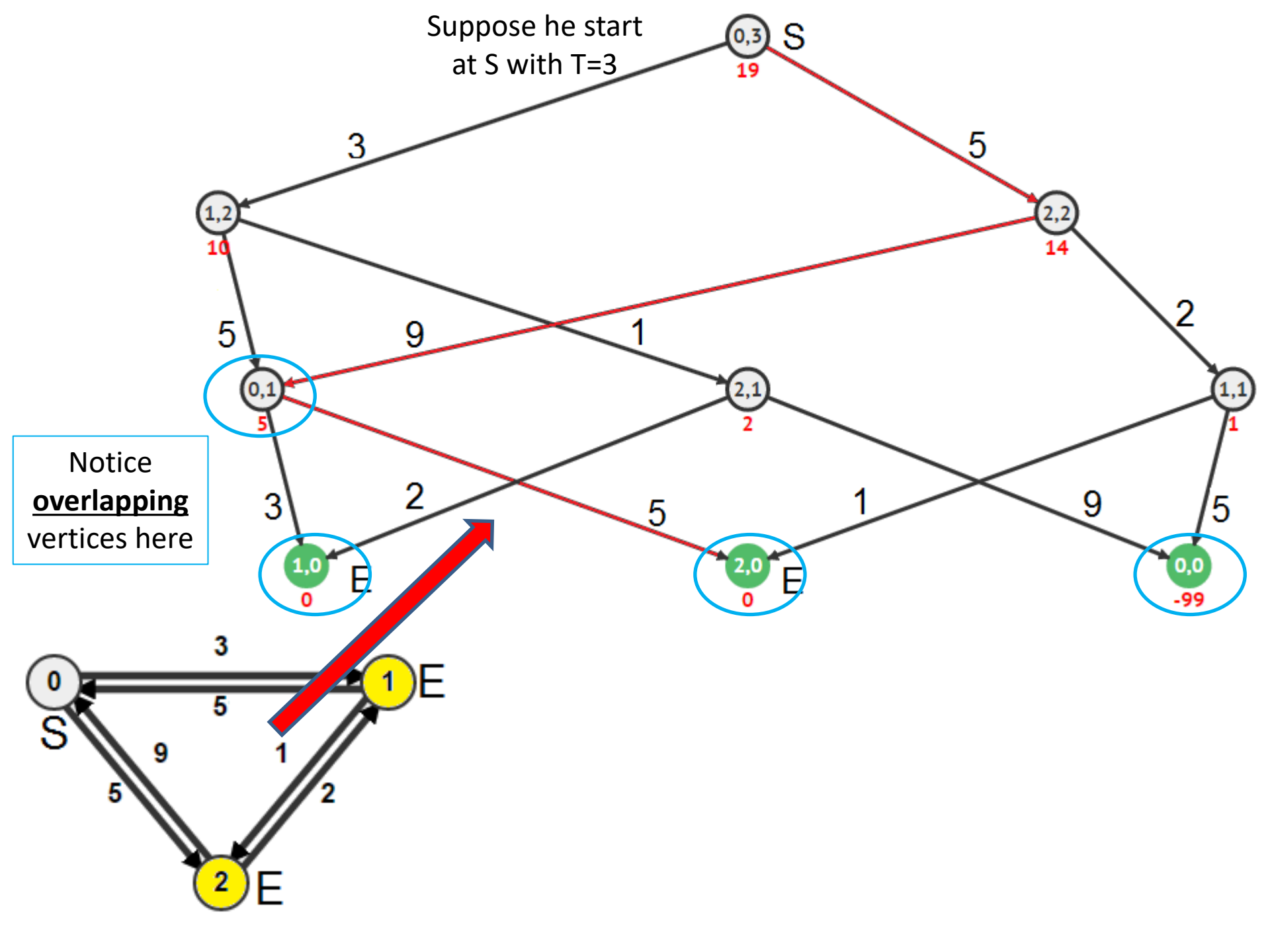
The actual problem (UVa 10702) has an extra condition that converts the general graph into a DAG

- The salesman can only make T inter-city travels :O
  - In a valid tour of the cities, he must arrive at an ending vertex after T step
- **KEY STEP**: Now each vertex has an additional parameter:
  - **num of steps left**
- The graph now looks like this, a DAG:



Suppose he start at S with T=3



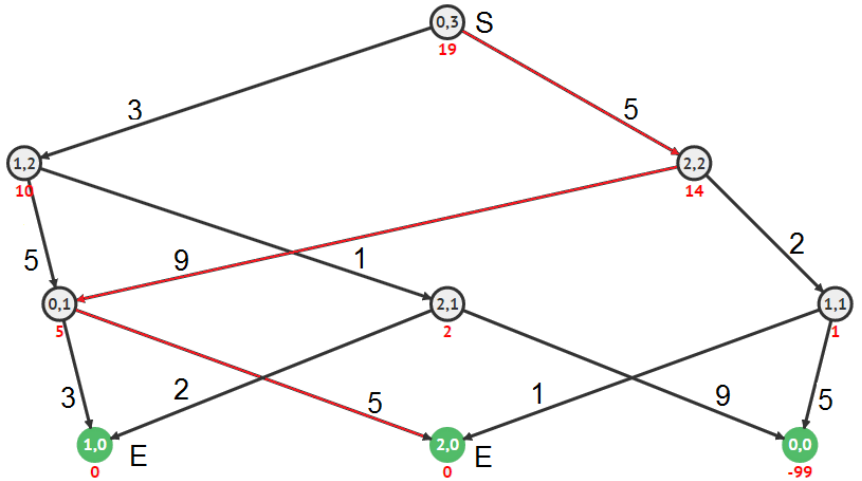

Suppose he start at S with T=3
0,3
S
19
3
5
1,2
10
2,2
14
5
9
1
2
0,1
5
2,1
2
1,1
1
Notice **overlapping** vertices here
3
2
5
1
9
5
1,0
0
E
2,0
0
E
0,0
-99
0
S
1
E
2
E
3
5
9
5
1
2

A Note About Longest Path Problem on Directed Acyclic Graph

There is no longest *non simple* path <u>on DAG</u>

- This is because every paths on DAG are simple, including the longest path ☺

So we can use the term SSLP on DAG, but we usually have to use the term SSL(simple)P on general graph



Graph Solution versus DP Solution

What is the solution for the SSLP on DAG problem?

- We are already familiar with this from the previous lecture
  - SS Longest paths on DAG can be solved with either:
    A. Find topological order and "stretch" edges according to this order/ bottom-up DP (or sometimes called as 'graph way')
    B. Or write a recursive function (top-down DP) with memoization

But this problem is *harder* to be solved with graph way

- The vertices now contain **pair** of information: (vertex_number, steps_left)
  - The number of vertices is not **C**, but now **C*T**
    - Pure graph implementation is slightly more difficult…

DP Solution (1)

Let's solve this problem with top-down DP (recursion + memo)

- We will *not* actually build the (implicit) DAG

Let **get_profit(u, t)** be the maximum profit that the salesman can get when he is at city **u** with **t** number of steps to go:

- if **t** = 0
  - If the salesman can end his tour at city u, i.e. city u has label E;
    - Then **get_profit(u, t) = 0**
  - else if the salesman cannot end his tour at city u;
    - Then **get_profit(u, t) = -INF** (to say that this is a bad choice)
- else (if **t** > 0),
  - **get_profit(u, t) = max(profit[u][v] + get_profit(v, t-1))**
    for all **v** ∈ **[0 .. C-1]** and we skip the case where **v = u** (as we cannot stay)
- What are the dimensions of memo table?
  - Need to save solution to each state (2 parameters)→ city & num steps left
  - Therefore memo table is 2D with size = O(C*T)
  - Or simply look at number of parameters of recursive function and their size

DP Solution (2)

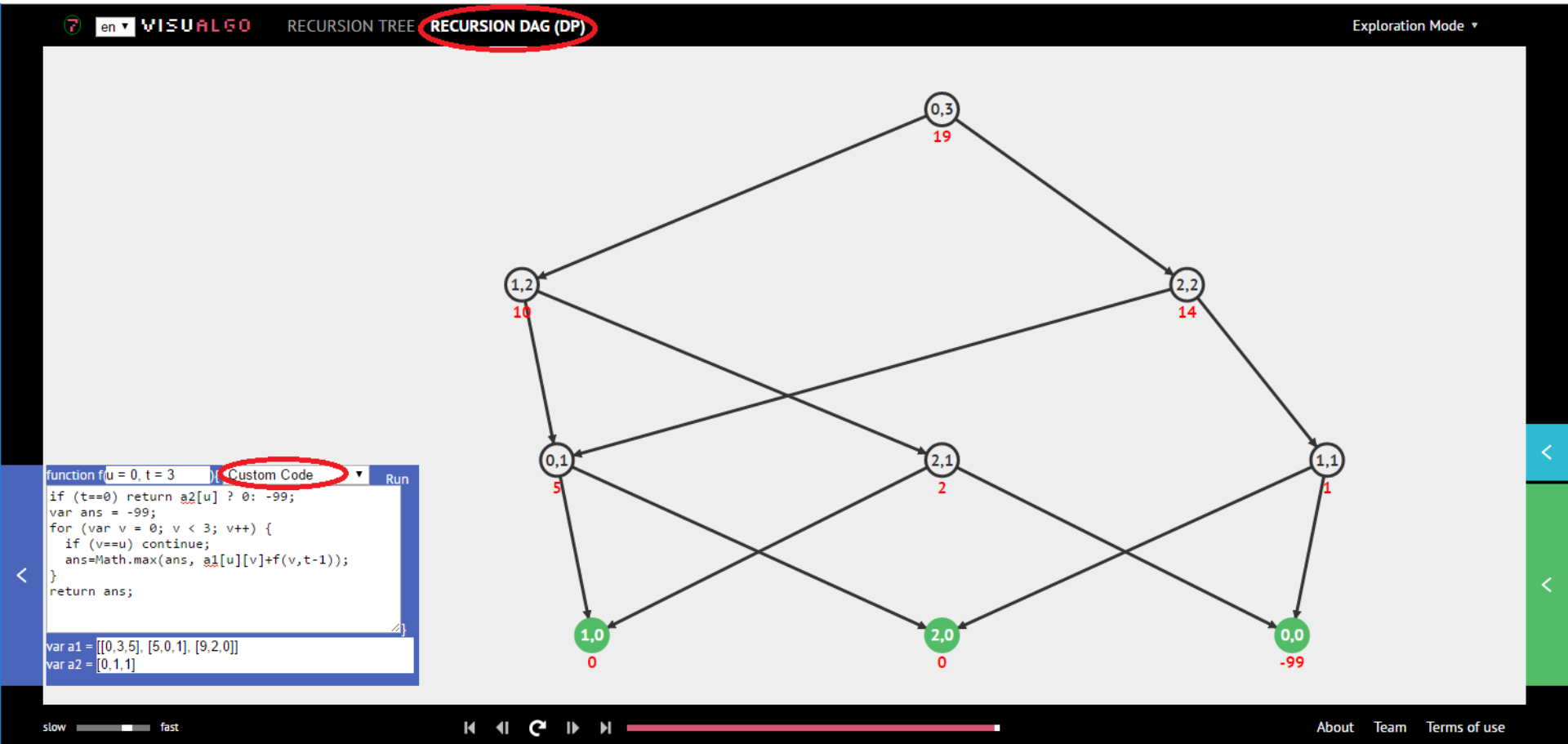
In Java code (see UVa10702.java):

```
private static int get_profit(int u, int t) {
  if (t == 0) // last inter-city travel?
    return canEnd[u] ? 0 : -INF;
  if (memo[u][t] != -1) // computed before?
    return memo[u][t]; // use a 2D array as memo

  memo[u][t] = -INF;
  for (int v = 0; v < C; v++) {
    if (v == u) continue; // we cannot stay
    memo[u][t] = Math.max(memo[u][t],
       profit[u][v] + get_profit(v, t-1));
  }
  return memo[u][t]; // to avoid re-computation
}
```

DP Feature @ VisuAlgo

Try visualgo.net/recursion.html , select custom code and type in the code below and the values for a1, a2, u and t, then press run to view the DP solution



DP Analysis

What is the num of distinct states/space complexity?

- That is, the #vertices in the DAG
  - Answer: O(**C*****T**)

What is the time to compute one distinct state?

- That is, the out-degree of a vertex
  - Answer: O(**C**), scan all **C** vertices but exclude staying at current vertex

What is the overall time complexity?

- That is, #edges in the DAG = #vertices*outdegree of each vertex
  - Or number of distinct states * time to compute one distinct state
  - Answer: O((**C*****T**) * **C**) = O(**C**^{2}***T**)

Then, we will perform complete search on General Graph

TRAVELING SALESMAN PROBLEM (THE CLASSIC VERSION)

Traveling Salesman Problem (TSP)

The classic TSP is actually "simple" to describe:

- Given a list of **V** cities and their ${}_{\mathbf{V}}C_2$ pairwise distances
  - That is, a **complete weighted graph**, which is a general graph
- Find a **shortest tour** that visits each city **exactly once** except starting city
  - The tour must start and end at the same city
  - Thus the tour will have exactly **V** edges, a **simple** tour

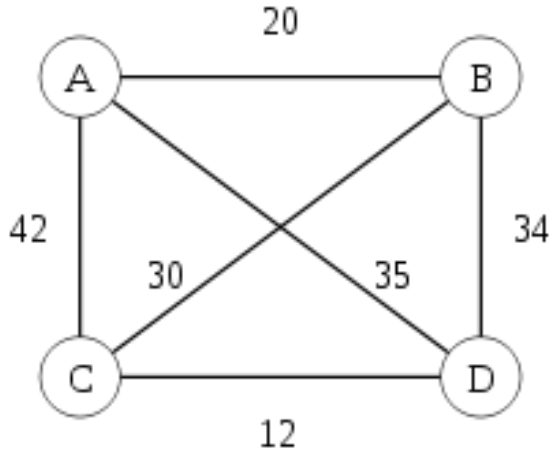
Note that this problem is different from UVa 10702 shown earlier in this lecture

- Take some time to examine the differences

The shortest tour for this TSP instance is …

(you will need some time to compute this)

1. Tour A-B-C-D-A with cost _____
2. Tour A-C-B-D-A with cost _____



How many possible tours are there in a TSP instance of **V** cities?

1. **V** valid tours
2. **V log V** valid tours
3. $\mathbf{V^2}$ valid tours
4. **V!** valid tours

What is the value of “10!” ?

1. 10
2. 100
3. 3628800
4. 10000000
5. 9.332621544394415
   2681699238856267
   e+157

What is the value of “10<u>0</u>!” ?

1. 10
2. 100
3. 3628800
4. 10000000
5. 9.332621544394415
   2681699238856267
   e+157

Brute Force (Naïve) Solution

An O(**V!** * **V**) solution in sketch:

1. Try all **V!** tour **permutations**
2. Compute the cost for each tour – doable in O(**V**)
3. Pick the one with the minimum cost…

But this sketch is too coarse for proper implementation

- 'Simple' question:
  How to generate **V!** permutations of **V** vertices?

Demo: Generating All Permutations (1)

Note: This is usually hard for first timers

The demonstration steps (see TSPslow.java):

1. I start from DFSrec(**s**) (Lecture 06)
   - DFS is O(**V**+**E**), but a complete graph has **V** = **N** and **E** = $\mathbf{N^2}$
   - Thus DFS runs in O(**N**+$\mathbf{N^2}$) = O($\mathbf{N^2}$)
2. I will show how to change DFSrec(**s**) into a backtracking routine that tries all permutations that start and end at a vertex **s**
   - This is an O(**N!**) algorithm, as there are (**N**-1)**!** possible tours

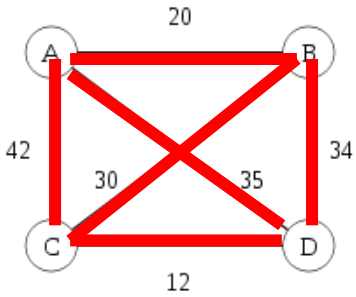
Demo: Generating All Permutations (2)

To do **backtracking** in this complete general graph, we have to use the **visited** flag that is turned on when entering the recursion **and turned off when exiting the recursion (that is the main difference with DFSrec)**

- Backtracking is one example of Complete Search technique

The Only Change…

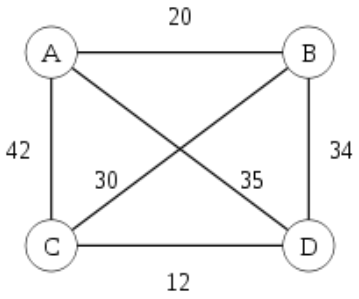
```
private void DFSrec(int u) {
   visited[u] = true; // to avoid cycle
   for (int j = 0; j < AdjList.get(u).size(); j++) {
     IntegerPair v = AdjList.get(u).get(j);
     if (!visited[v.first()])
       DFSrec(v.first());
   }
   visited[u] = false; // let vertex u be reusable later
 }
```



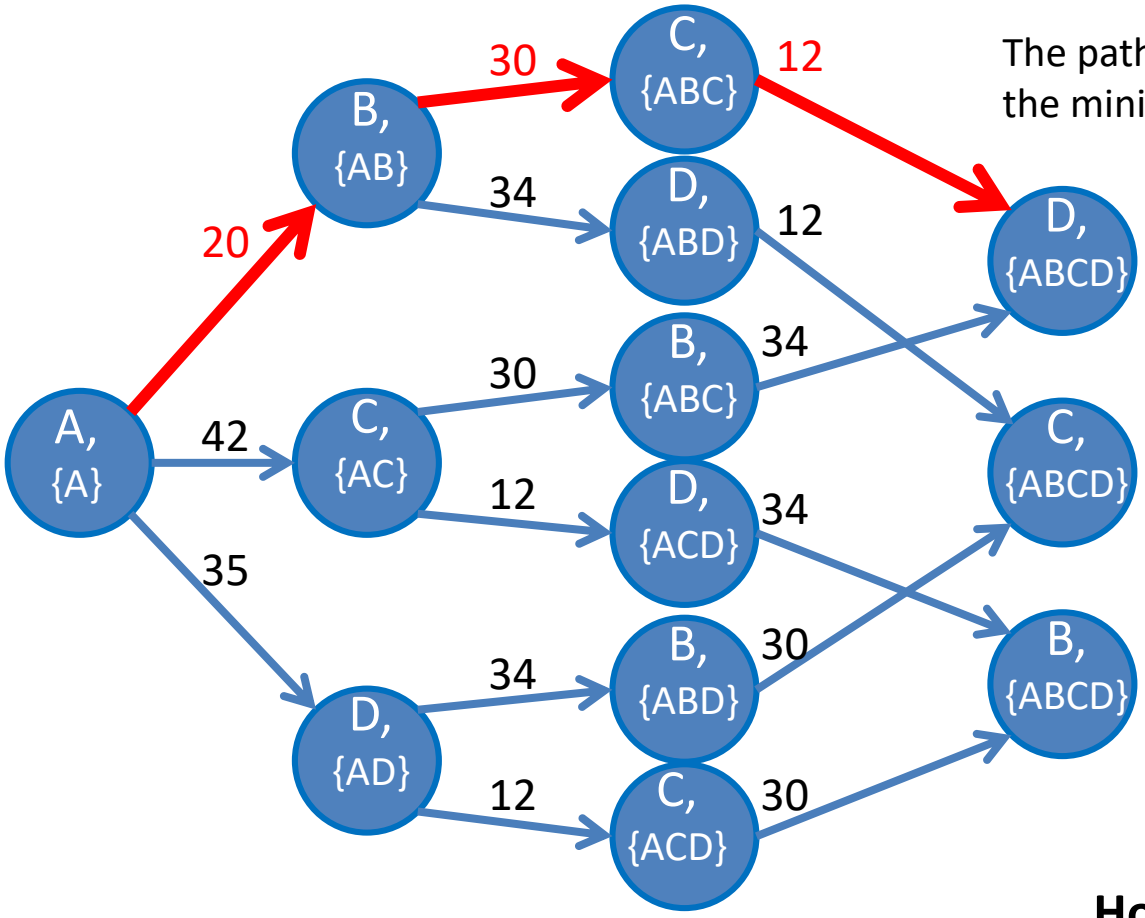
TSP Complete Search (1)

Other stuff you need to code to find shortest tour

- Keep track of the current path
- Once a possible permutation is generated (path has v vertices)
  - compute the cost of the edges in the permutation
  - Including weight of edge from last vertex in path back to source vertex
  - Update the minimum tour cost found so far



Convert into a DAG !



The path in red (it is a tour actually) has the minimum total weight of 97

35
(goes back from D to A)

42
(goes back from C to A)

20
(goes back from B to A)

How to get this DAG?

TSP DP Solution ?

- Since the graph can be converted into a DAG there is a DP solution
- Take CS4234 to learn how to solve TSP using DP solution ☺

Larger TSP Instances

N	N!	Remarks
9	362,880	
10	3,628,800	10 x (PS6 A+B)
11	39,916,800	11 x
12	479,001,600	12 x
13	6,227,020,800	13 x
14	87,178,291,200	14 x
15	1,307,674,368,000	15 x
16	20,922,789,888,000	16 x
17	355,687,428,096,000	17 x
20 :O	$\sim 2.4 * 10^{18}$	
30 :O	$\sim 2.6 * 10^{32}$	

LONGEST COMMON SUBSEQUENCE

(ANOTHER HIGHER DIMENSIONAL DP)

Longest Common Subsequence (LCS) (1)

- Given two strings x and y, find the longest common subsequence (LCS) and print its length
- Example:
  - x = A**BC**BD**AB**
  - y = **B**D**CAB**C
  - "BCAB" is the longest subsequence found in both sequences, so the answer is 4

Longest Common Subsequence (LCS) (2) – Define problem to be solved

*Define problem in a way that is amenable to DP

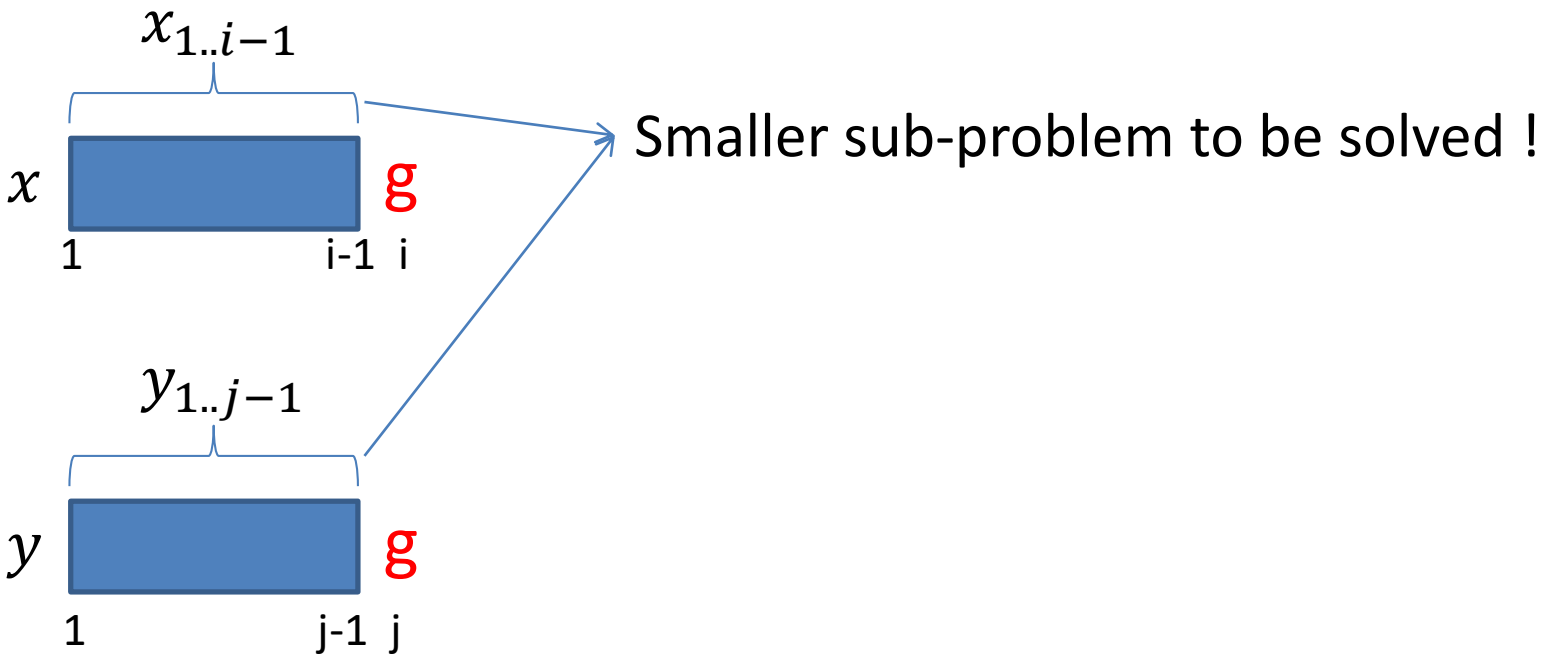
- Let a sequence x of length n: $x = < x_1, x_2, \ldots, x_{n-1}, x_n >$
- Let $x_{1..i}$ be a prefix of x of length i: $x_{1..i} = < x_1, x_2, \ldots, x_i >$, $0 \leq i \leq n$
  - $i = 0$ represents the empty prefix
- Let $D_{i,j}$ be the length of the LCS of $x_{1..i}$ and $y_{1..j}$ where $0 \leq i \leq length(x)$ and $0 \leq j \leq length(y)$
  - We need two parameters i and j to define a state (2 dimensional DP)

- LCS of two sequences x and y is then LCS of $x_{1..length(x)}$ and $y_{1..length(y)}$ or $D_{length(x),length(y)}$
  - Optimal Substructure → solution to longer prefixes should depend on solution to shorter prefixes
  - Write a recurrence for solving $D_{i,j}$

Longest Common Subsequence (LCS) (3)

Define the recurrence

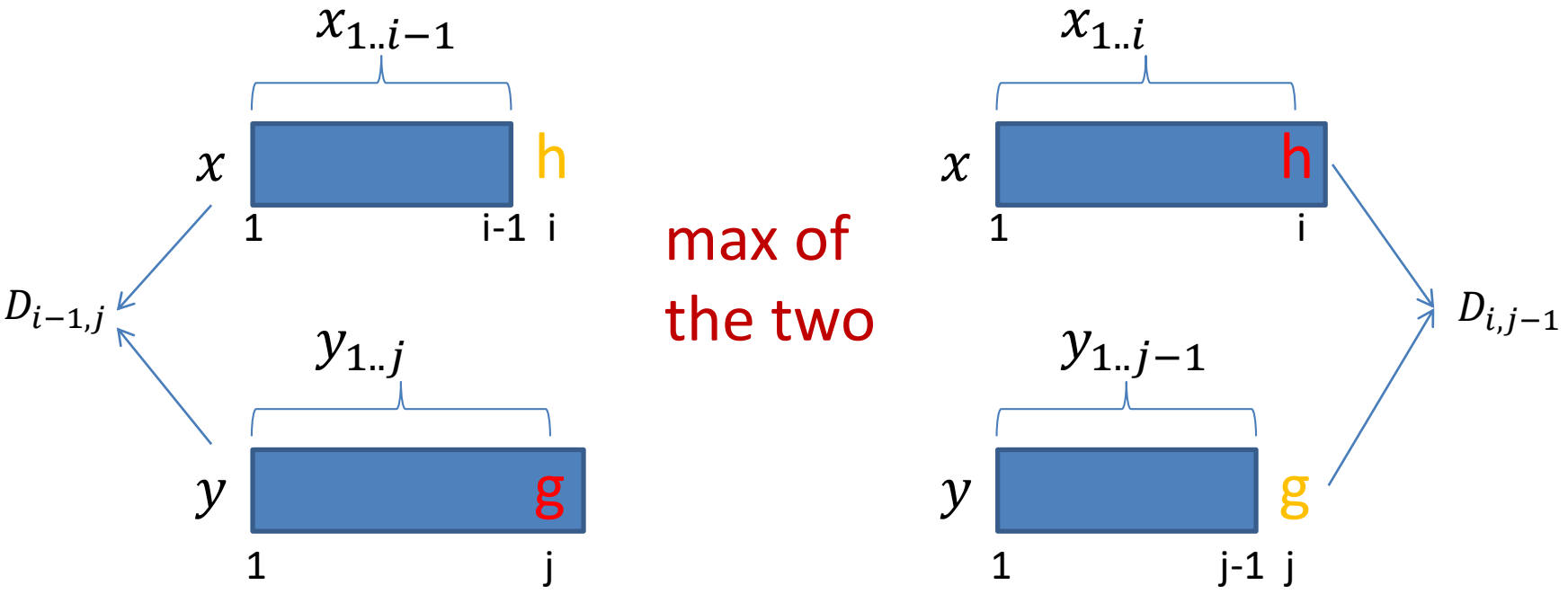
- To compute $D_{i,j}$, we look at the characters x_i and y_j
- If $x_i = y_j$, they both contribute to the LCS
  - $D_{i,j} = D_{i-1,j-1} + 1$



Longest Common Subsequence (LCS) (4)

Define the recurrence

- If $x_i \neq y_j$, then either x_i or y_j does not contribute the LCS.
- Try dropping each one and get the max.
  - $D_{i,j} = max(D_{i-1,j}, D_{i,j-1})$



Longest Common Subsequence (LCS) (4)

- Base cases: i or j = 0, $D_{0,j} = D_{i,0} = 0$
- Recursive Case:
  - If $x_i = y_j, D_{i,j} = D_{i-1,j-1} + 1$
  - If $x_i \neq y_j, D_{i,j} = max(D_{i-1,j}, D_{i,j-1})$

- To implement this recurrence need a 2D array for D
  - size is length(x)*length(y)
- Fill D left to right, top to bottom (topological ordering)
- Final solution is in $D[length(x)][length(y)]$

Longest Common Subsequence (LCS) (5)

- Implementation – Iterative version (bottom up)

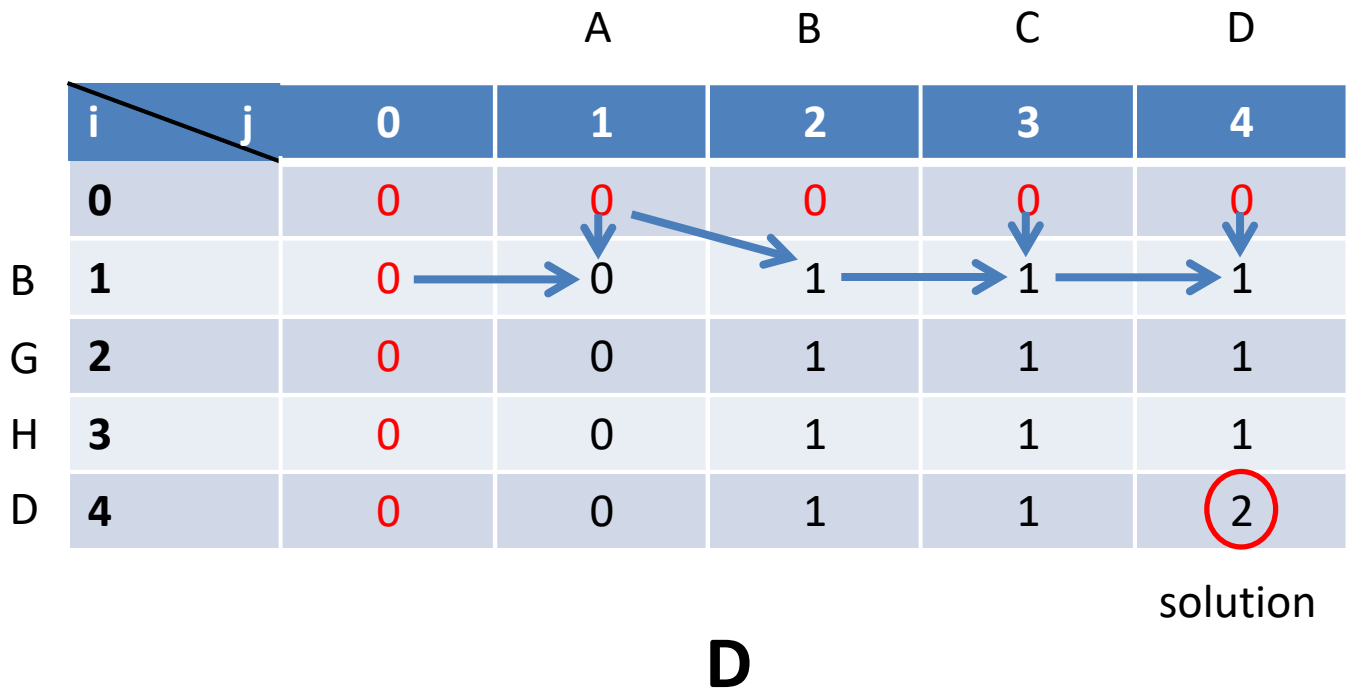
```
for(i = 0; i <= n; i++)
    D[i][0] = 0;
for(j = 0; j <= m; j++)
    D[0][j] = 0;
for(i = 1; i <= n; i++) {
    for(j = 1; j <= m; j++) {
        if(x[i] == y[j])
            D[i][j] = D[i-1][j-1] + 1;
        else
            D[i][j] = max(D[i-1][j], D[i][j-1]);
    }
}
```

Init all the base cases to 0

Left to right, top to bottom is correct order since subproblem solutions are found to left, up or upper left of current cell

Longest Common Subsequence (LCS) (6)

- Example: x = **B**GH**D**, y = A**B**C**D**



DP Analysis

What is the num of distinct states/space complexity?

- Answer: O(length(x)*length(y)) -> size of D

What is the time to compute one distinct state?

- In recursive case, at most look at the solution to 2 previous states (imagine each vertex representing a state as 2 incoming edges)
- Answer: O(**1**)

What is the overall time complexity?

- number of distinct states * time to compute one distinct state
  = O((length(x)*length(y)) * O(1)
  = O((length(x)*length(y))

Summary (1)

By definition, a general graph has cycle(s)

- We cannot write a recursive formula in a cyclic structure…
  - Therefore we cannot use DP technique on general graph :O…

We have seen how to convert *some* general graphs into DAGs by introducing one extra parameter

- Now we can write recursive formulas and use DP technique
- Analysis of the space and time complexity of *basic* DP is *easy*
- Note: Some of these problems are better solved as top-down DP (written as recursive functions), where we do not explicitly build the DAG…
  - Note that harder DP problems may require us to introduce more than one extra parameters…

Summary (2)

We have also modelled another non-graph problem (LCS) into higher dimensional DP problem

- Define the problem appropriately
- Write the recurrences
- Use bottom-up DP without explicitly building the DAG