Dynamic Programming

By
Prof. Shaik Naseera
Department of CSE
JNTUA College of Engg., Kalikiri

Objectives

- Introduction to Dynamic Programming
- Longest Common Subsequence
- Shortest Path Problems
  - All Pairs Shortest Path
  - Travelling Sales Person Problem
  - Multi-Stage Graphs
- Previous Gate Questions

Introduction

- Dynamic Programming is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions.
- One way to solve problems for which it is not possible to make a sequence of stepwise decisions leading to an optimal decision sequence is to try out all possible decision sequences.
- We could enumerate all decision sequences and then pick out the best.
- Dynamic programming often drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal.
- In dynamic programming an optimal sequence of decisions is arrived at by making explicit appeal to the ***Principle of Optimality.***
- *This principle states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.*

Dynamic Programming

- Well known algorithm design techniques:.
  - Divide-and-conquer algorithms

- Another strategy for designing algorithms is **dynamic programming**.
  - Used when problem breaks down into recurring small subproblems

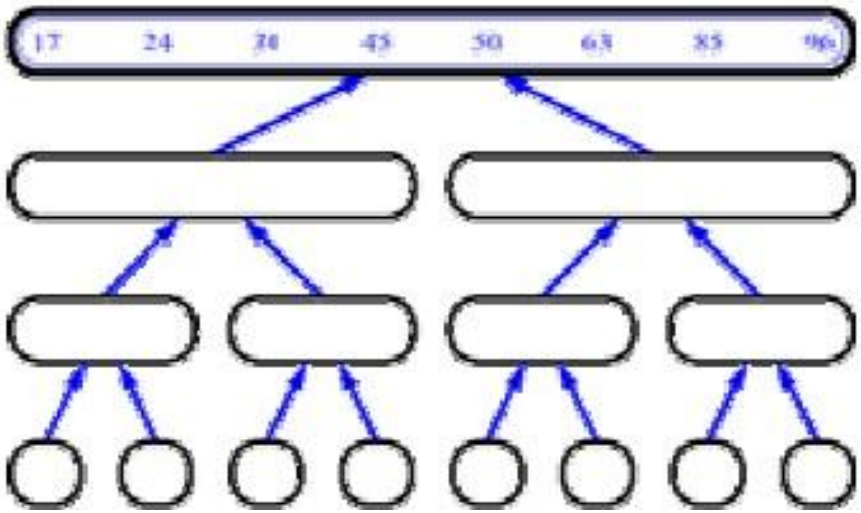
- ***Dynamic programming*** is typically applied to optimization problems. In such problem there can be ***many solutions***. Each solution has a value, and we wish to find *a* ***solution*** with the optimal value.

Divide-and-conquer – Example1

- For example, **MergeSort**

```
Merge-Sort(A, p, r)
    if p < r then
        q←(p+r)/2
        Merge-Sort(A, p, q)
        Merge-Sort(A, q+1, r)
        Merge(A, p, q, r)
```

- The **subproblems** are **independent**, **all different.**



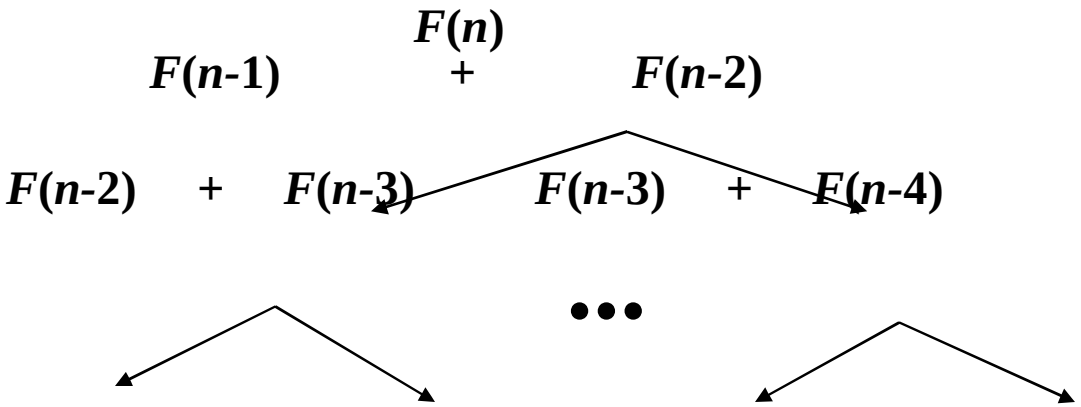
Example 2: Fibonacci numbers

- **Recall definition of Fibonacci numbers:**

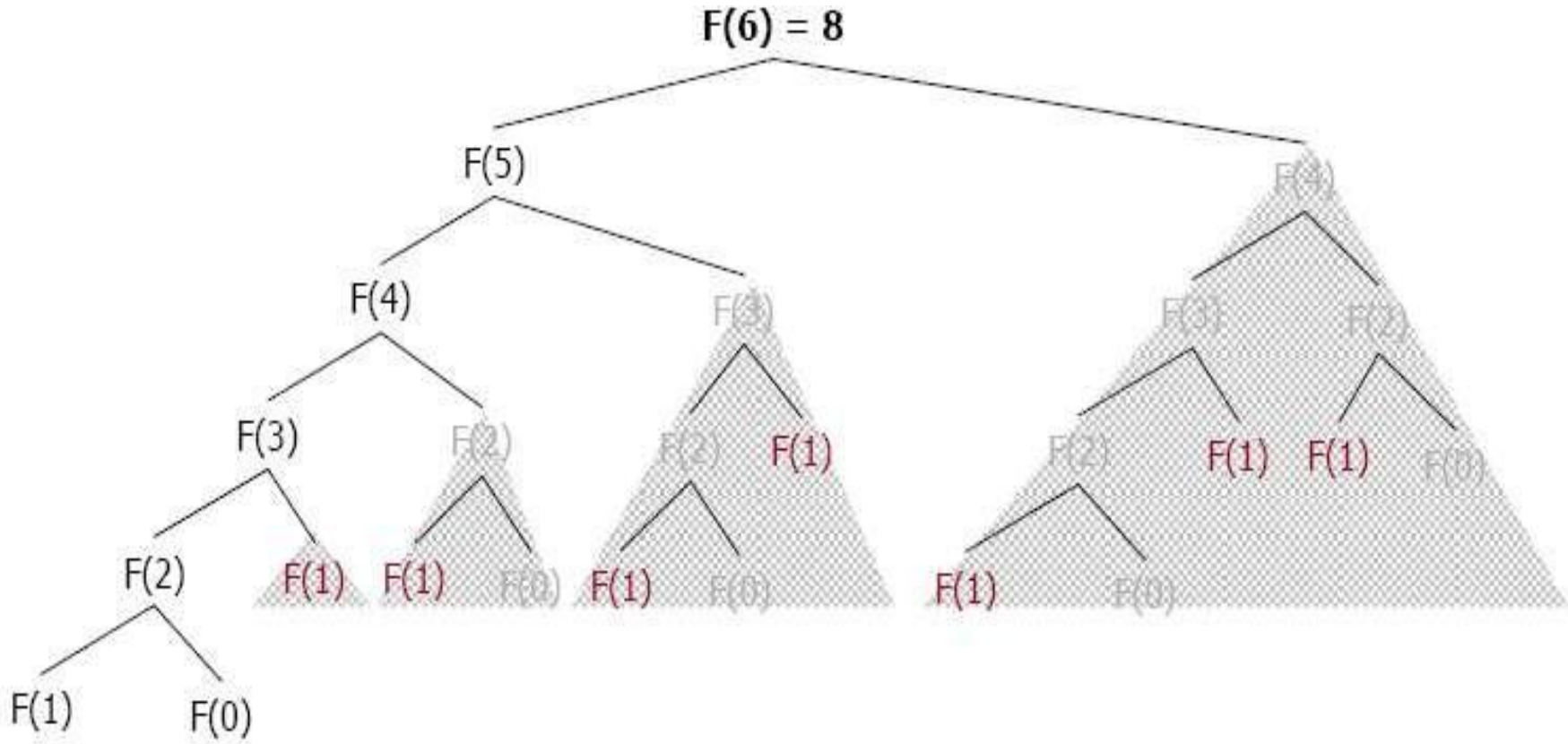
$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$
$$F(0) = 0$$
$$F(1) = 1$$

Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

- **Computing the n^{th} Fibonacci number recursively (top-down):**



Fibonacci Numbers



- We keep calculating the same value over and over!

  Running time complexity becomes exponential

DP Approach: Fibonacci Numbers

- We can calculate *Fn* in ***linear* time** by remembering solutions to the solved subproblems – *dynamic programming*

- Compute solution in a bottom-up fashion

- In this case, only two values need to be remembered at any time

```
Fibonacci(n)
    F0←0
    F1←1
    for i ← 2 to n do
        Fi ← Fi-1 + Fi-2
```

Difference between DP and Divide-and-Conquer

- Using Divide-and-Conquer to solve these problems is **inefficient** because the **same** common **subproblems** have to be solved **many times.**

- DP will solve each of them **once** and **their answers are stored in a table** for future use.

Applications of Dynamic Programming

- Longest common subsequence
- Shortest path problems
  - Multi Stage Graphs
  - All Pair Shortest Path
  - Travelling Sales Person Problem
- 1/0 Knapsack
- Reliability Design
- Matrix chain multiplication
- Optimal Merge portions
- Mathematical optimization
- Optimal Binary Search Tree

Longest Common Subsequence

The **longest common subsequence** (**LCS**) **problem** is the problem of finding the longest subsequence common to all sequences in a set of sequences

Subsequences

- A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.
- In LCS ,we have to find Longest Common Subsequence that is in the same relative order.
- String of length n has 2^n different possible subsequences.
- E.g.—
- Subsequences of “ABCDEFG”.
- “ABC”,”ABG”,”BDF”,”AEG”,'ACEFG”,.....

Common Subsequences

Suppose that X and Y are two sequences over a set S.

X: ABCBDAB

Y: BDCABA

Z: BCBA

We say that Z is a common subsequence of X and Y if and only if

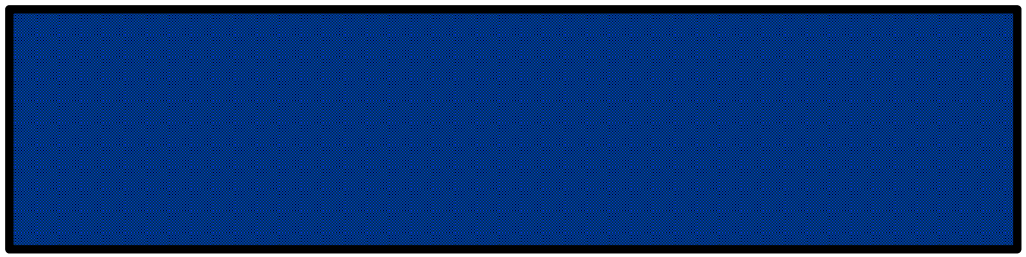
- Z is a subsequence of X
- Z is a subsequence of Y

The Longest Common Subsequence Problem

Given two sequences X and Y over a set S, the longest common subsequence problem asks to find a common subsequence of X and Y that is of maximal length.

Z= (B,C,A) Length 3

X: ABCBDAB Y: BDCABA



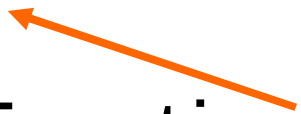
Longest

LCS Notation

Let X and Y be sequences.

We denote by LCS(X, Y) the set of longest common subsequences of X and Y.

LCS(X,Y)



Functional notation,

but not a function

A Poor 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 is of length n, then it has 2^n subsequences
  - This is an exponential-time algorithm!

Dynamic Programming for LCS

Let us try to develop a dynamic programming solution to the LCS problem.

Dynamic Programming for LCS

		a	b	c	d	A	f
	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
c	0	1	1	2	2	2	2
b	0	1	2	2	2	2	2
c	0	1	2	3	3	3	3
f	0	1	2	3	3	3	4

- Example Input strings:
- a b c d a f
- a c b c f
- Solution:
- Longest common subsequence = a b c f
- Length = 4
- Pseudocode:
- If input[i] == input[j]
- T[i][j] = T[i-1][j-1] + 1;
- Else
- T[i][j] = MAX { T[i-1][j], T[i][j-1] }

Solution

		a	b	c	d	a	f
	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
c	0	1	1	2	2	2	2
b	0	1	2	2	2	2	2
c	0	1	2	3	3	3	3
f	0	1	2	3	3	3	4

a b c f

Procedure

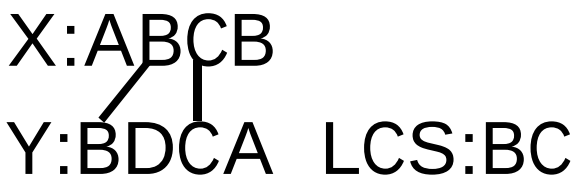
Let X and Y be sequences.

Let c[i,j] be the length of an element in LCS(X_i, Y_j).

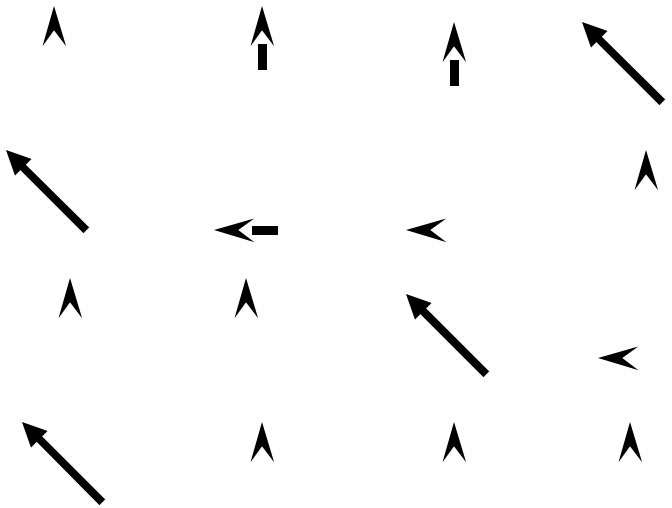
$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i = y_j \\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i =! y_j \end{cases}$$

Dynamic Programming Solution

- Define L[i,j] to be the length of the longest common subsequence of X[0..i] and Y[0..j].
- L[i,j-1] = 0 and L[i-1,j]=0, to indicate that the null part of X or Y has no match with the other.
- Then we can define L[i,j] in the general case as follows:
  1. If xi=yj, then L[i,j] = L[i-1,j-1] + 1 (we can add this match)
  2. If xi≠yj, then L[i,j] = max{L[i-1,j], L[i,j-1]} (we have no match here)



Example



Start at b[m,n]. Follow the arrows. Each diagonal array gives one element of the LCS.

X: pqprqrp Y: qpqrr

		p	q	p	r	q	r	p
	0	0	0	0	0	0	0	0
q	0	0	1	1	1	1	1	1
p	0	1	1	2	2	2	2	2
q	0	1	2	2	2	3	3	3
r	0	1	2	2	3	3	4	4
r	0	1	2	2	3	3	4	4

qprr

ALGORITHM LCS(X,Y)

```
m ← length[X]
n ← length[Y]
for i ← 1 to m do
     c[i,0] ← 0
for j ← 1 to n do
     c[0,j] ← 0
```

LCS(X,Y)

```
for i ← 1 to m do
   for j ← 1 to n do
       if x_i = y_j
          c[i, j] ← c[i-1, j-1]+1
          b[i, j] ← “D”   //diagonal
       else
          if c[i-1, j] ≥ c[i, j-1]
               c[i, j] ← c[i-1, j]
               b[i, j] ← “U”   //up
          else
               c[i, j] ← c[i, j-1]
                b[i, j] ← “L”  //left
return c and b
```

Analysis of LCS Algorithm

- We have two nested loops
  - The outer one iterates m times
  - The inner one iterates n times
  - A constant amount of work is done inside each iteration of the inner loop
  - Thus, the total running time is O(*nm*)

usage

- Biological applications often need to compare the DNA of two (or more) different organisms.
- We can say that two DNA strands are similar if one is a substring of the other.

Shortest Path Problem

-All Pair Shortest Path

-Travelling Sales Person Problem

-Multi-stage Graph

All Pair Shortest Path

- The **all pair shortest path** algorithm is also known as Floyd's algorithm is used to find shortest graph distances between every pair of vertices in a given graph.
- As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to **all** other nodes in the graph.

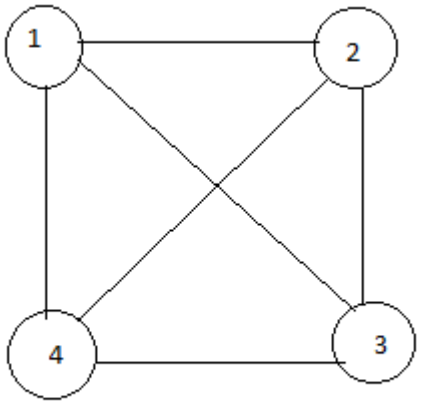
Description

- All Pair Shortest Path Floyd Alg.pptx

Traveling Salesperson Problem

- Let G = (V, E) be a directed graph with edge costs C_{ij}. C_{ij} is defined such that $C_{ij} > 0$ for all i and j and $C_{ij} = \infty$ if $< i,j > \notin E$.
- Let $|V| = n$ and assume $n > 1$.
- A tour of G is a directed cycle that includes every vertex in V.
- The cost of a tour is the sum of the cost of the edges on the tour.
- The traveling salesperson problem is to find a tour of minimum cost.
  - $g(i, S)=\min\{c_{ij}+g(j, S-\{j\})\}$ where $j \in S$

Example



$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

- We need to find the cost of the tour g(1, {2,3,4})

- g(1, {2,3,4}) = min { c12+g(2, {3,4}),
  c13+g(3, {2,4}),
  c14+g(4, {2,3} }
- g(2, {3,4})=min {c23+g (3, {4}), c24+g(4, {3}) }
- g(3, {4})= min {c34+g(4, ∅} = 12+c41=12+8=20
- g(4, {3})=min {c43+g(3, ∅)} = 9+c31=9+6=15
- g(2, {3,4})=min {c23+g (3, {4}), c24+g(4, {3}) }
  =min {9+20, 10+15}
  =min{29, 25} =25

g(4, ∅) = c41=8

- g(3, {2,4})=min {c32+g (2, {4}), c34+g(4, {2}) }
- g(2, {4})= min {c24+g(4, ∅} = 10+8=18
- g(4, {2})=min {c42+g(2, ∅)} = 8+5=13
- g(3, {2,4})=min {c32+g (2, {4}), c34+g(4, {2}) }

  =min {13+18, 12+13}

  =min{31, 25} =25

- g(4, {2,3})=min {c42+g (2, {3}), c43+g(3, {2}) }
- g(2, {3})= min {c23+g(3, ∅} = 9+6=15
- g(3, {2})=min {c32+g(2, ∅)} = 13+5=18
- g(4, {2,3})=min {c42+g (2, {3}), c43+g(3, {2}) }

  =min {8+15, 9+18}

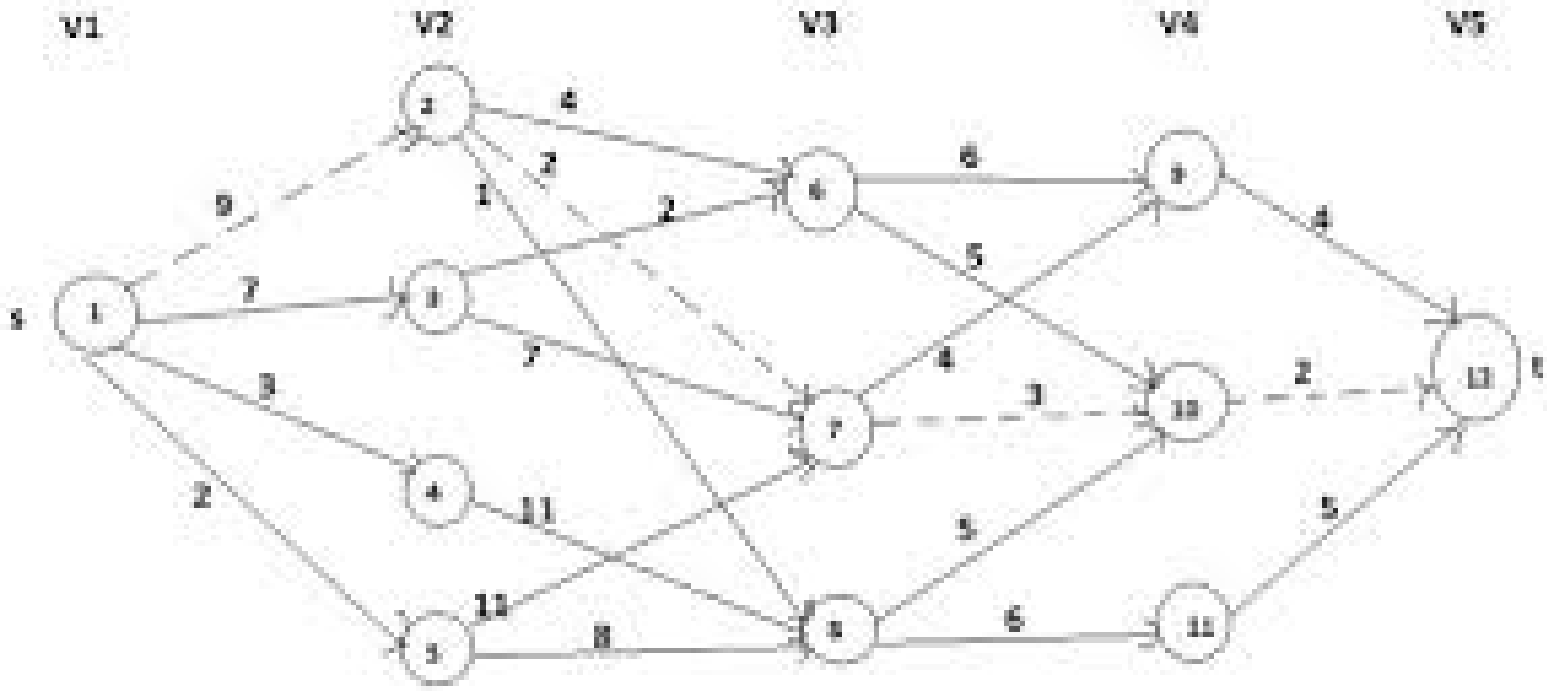
  =min{23, 27} =23

- g(1, {2,3,4}) = min { c12+g(2, {3,4}),
  c13+g(3, {2,4}),
  c14+g(4, {2,3} }
  =min{10+25, 15+25, 20+23}
  =min{35, 40, 43}
  =35

The optimal path is 1-2-4-3-1

$$\begin{bmatrix} 0 & 20 & 30 & 10 \\ 15 & 0 & 16 & 4 \\ 3 & 5 & 0 & 2 \\ 19 & 6 & 18 & 0 \end{bmatrix}$$

Multi Stage Graph



MULTI STAGE GRAPH

Multi-stage Graph

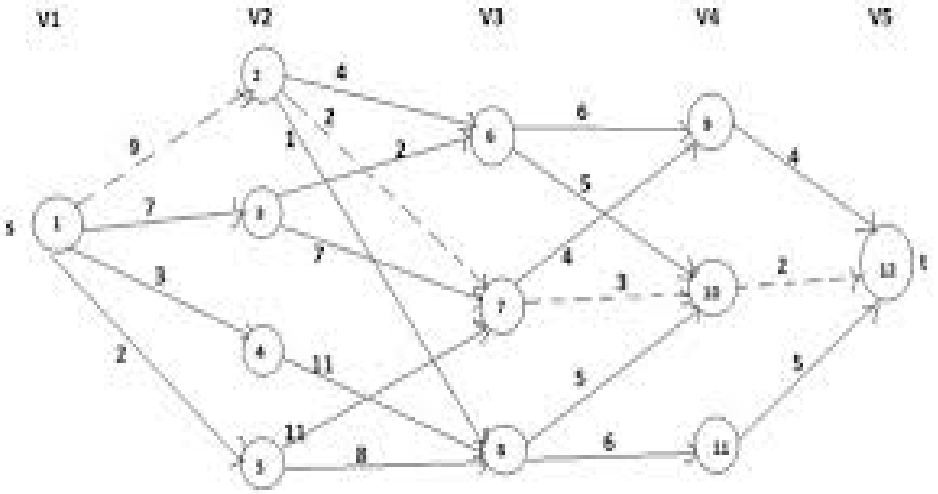
- A multistage graph is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$.
- $\langle u, v \rangle$ is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$ for some i, $1 \leq i \leq k$.
- The sets V_1 and V_k are such that $|V_1| = |V_k| = 1$
- s and t are the vertices in V_1 and V_k respectively.
- The vertex s is the source and t is the sink
- The multi stage graph is to find a minimum cost path from s to t.

Approaches to Multistage Graph

- Forward Approach
- Backward Approach

Forward Approach

- $Cost(i, j)=\min \{ c(j, l) + cost(i+1,l)\}\ l \in V_{i+1}$, $<j,l> \in E$



MULTI STAGE GRAPH

- Cost(5,12)=0;
- Cost(4,9)=4+ Cost(5,12)=4;
- Cost(4,10)=2+ Cost(5,12)=2;
- Cost(4,11)=5+ Cost(5,12)=5;
- Cost(3,6)=min{6+cost(4,9), 5+cost(4,10)}=min{10,7}=7
- Cost(3,7)=min{4+cost(4,9), 3+cost(4,10)}=min{8,5}=5
- Cost(3,8)=min{5+cost(4,10),6+cost(4,11)}=min{7,11}=7
- Cost(2,2)=min{4+cost(3,6),2+cost(3,7),1+cost(3,8)}=min{11,7,8}=7
- Cost(2,3)=min{2+cost(3,6),7+cost(3,7)}=min{9,12}=9
- Cost(2,4)=11+cost(3,8)=18
- Cost(2,5)=min{11+cost(3,7), 8+cost(3,8)}=min{16,15}=15
- Cost(1,1)=min{9+cost(2,2),7+cost(2,3),3+cost(2,4),2+cost(2,5}
  =min{16,16,21,17}=16

Shortest path 1-2-7-10-12

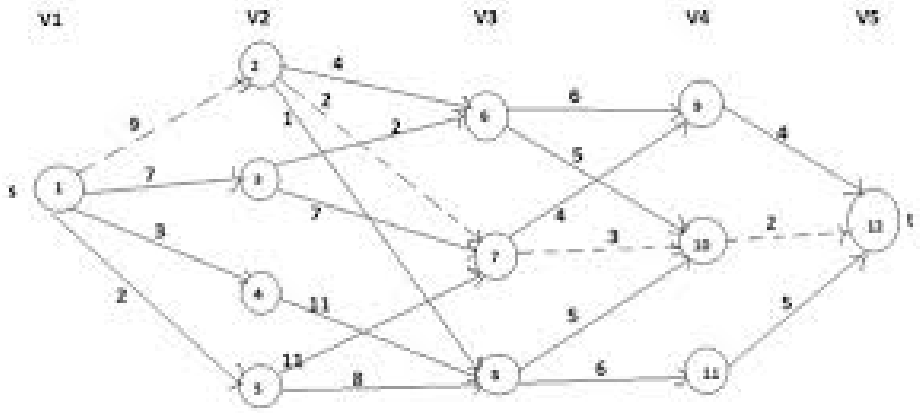
Cost(i, j)=min { c(j, l) +cost (i+1,l)} l $\in V_{i+1}$, <j,l> $\in$ E

Algorithm

```
Fgraph (G, K, n, P)
  {
      Cost[n]:= 0.0;
      for j:= n-1 to 1 step -1 do
       {
            Let r be a vertex such that <j, r>  is an edge of G and C[j, r] + cost[r]
            is minimum;       //r is the vertex in v_{i+1} and j be the vertex in v_i
            Cost[j] := C[j, r] + Cost[r];   //taking the value of min. cost edge
            d[j] := r;
      }
      P[1]:= 1 ,  P[k]:= n
      for j:= 2 to k-1
      P[j] = d[P[j-1]];
  }
```

Backward Approach

- bcost(i, j)=min{c(l,j)+bcost(i-1,l)}, $l \in v_{i-1}$, $<l,j> \in E$



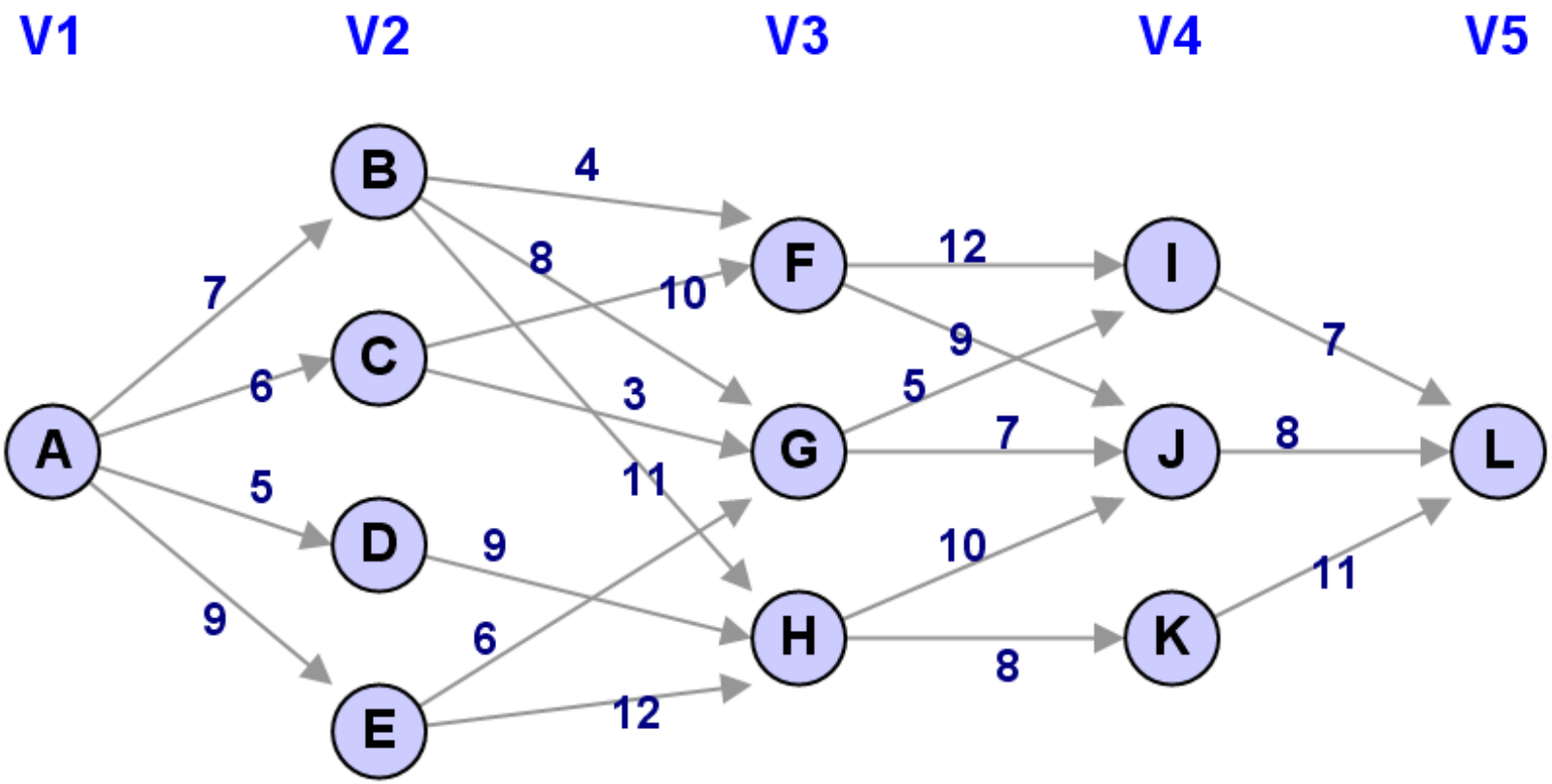
MULTI STAGE GRAPH

- Bcost(1,1)=0;
- Bcost(2,2)=9+bcost(1,1)=9;
- Bcost(2,3)=7++bcost(1,1)=7;
- Bcost(2,4)=3+bcost(1,1)=3;
- Bcost(2,5)=2+bcost(1,1)=2;
- Bcost(3,6)=min{4+bcost(2,2), 2+bcost(2,3)}=min{13,9}=9;
- Bcost(3,7)=11
- Bcost(3,8)=10
- Bcost(4,9)=15
- Bcost(4,10)=14
- Bcost(4,11)=16
- Bcost(5,12)=16

Algorithm

- Algorithm BGraph (G, K, n, p)
- // some function as FGraph
- {
- Bcost [1]:= 0.0;
- for j:= 2 to n do
- {
- // compute bcost[j].
- Let r be such that <r, j> is an edge of G and c[r,j]+ bcost[r] is minimum; //r is a vertex in v_{i-1} and j is a vertex in $v_{i;}$
- Bcost[j]=c[r,j]+ bcost[r];
- d [j]:= r;
- }
- // find a minimum cost path
- p[1] := 1;
- p[k] := n;
- for j := k-1 to 2 do
- p[j]:= d[p[j+1]];
- }

Multi Stage Graph



Gate Questions

Q. No. 1

GATE CSE 2016 Set 2

The Floyd-Warshall algorithm for all-pair shortest paths computation is based on

A Greedy paradigm.

B Divide-and-Conquer paradigm.

C Dynamic Programming paradigm.

D neither Greedy nor Divide-and-Conquer nor Dynamic Programming paradigm.

GATE CSE 2015 Set 1

Q. No. 2

Match the following:

List 1

(P) Prim's algorithm for minimum spanning tree

(Q) Floyd-Warshall algorithm for all pairs shortest paths

(R) Mergesort

(S) Hamiltonian circuit

List 2

(i) Backtracking

(ii) Greedy method

(iii) Dynamic programming

(iv) Divide and conquer

A P - iii, Q - ii, R - iv, S - i

B P - i, Q - ii, R - iv, S - iii

C P - ii, Q - iii, R - iv, S - i

D P - ii, Q - i, R - iii, S - iv

GATE CSE 2011

Q. No. 3

An algorithm to find the length of the longest monotonically increasing sequence of numbers in an array A[0:n−1] is given below.

Let L_i, denote the length of the longest monotonically increasing sequence starting at index i in the array. Initialize $L_{n-1}=1$.

For all i such that $0 \leq i \leq n-2$

$$L_i = \begin{cases} 1 + L_{i+1} & \text{if A[i] < A[i+1]} \\ 1 & \text{Otherwise} \end{cases}$$

Finally, the length of the longest monotonically increasing sequence is max(L_0, L_1, ...,L_{n-1})

Which of the following statements is TRUE?

A. The algorithm uses dynamic programming paradigm

B. The algorithm has a linear complexity and uses branch and bound paradigm

C. The algorithm has a non-linear polynomial complexity and uses branch and bound paradigm

D. The algorithm uses divide and conquer paradigm

Q. No. 4

GATE CSE 1998

Which one of the following algorithm design techniques is used in finding all pairs of shortest distances in a graph?

A Dynamic programming

B Backtracking

C Greedy

D Divide and Conquer

GATE CSE 2015 Set 2

Q. No. 5

Given below are some algorithms, and some algorithm design paradigms.

GROUP 1	GROUP 2
1. Dijkstra's Shortest Path	i. Divide and Conquer
2. Floyd-Warshall algorithm to compute all pairs shortest path	ii. Dynamic Programming
3. Binary search on a sorted array	iii. Greedy design
4. Backtracking search on a graph	iv. Depth-first search
	v. Breadth-first search

A $1-i,\ 2-iii,\ 3-i,\ 4-v.$

B $1-iii,\ 2-iii,\ 3-i,\ 4-v.$

C $1-iii,\ 2-ii,\ 3-i,\ 4-iv.$

D $1-iii,\ 2-ii,\ 3-i,\ 4-v.$

Match the above algorithms on the left to the corresponding design paradigm they follow.

GATE CSE 2009

Q. No. 6

A sub-sequence of a given sequence is just the given sequence with some elements (possibly none or all) left out. We are given two sequences X[m] and Y[n] of lengths m and n, respectively with indexes of X and Y starting from 0.

We wish to find the length of the longest common sub-sequence (LCS) of X[m] and Y[n] as l(m,n), where an incomplete recursive definition for the function l(i,j) to compute the length of the LCS of X[m] and Y[n] is given below:

```
l(i,j)  = 0, if either i = 0 or j = 0
        = expr1, if i,j > 0 and X[i-1] = Y[j-1]
        = expr2, if i,j > 0 and X[i-1] ≠ Y[j-1]
```

The value of l(i, j) could be obtained by dynamic programming based on the correct recursive definition of l(i, j) of the form given above, using an array L[M, N], where M = m+1 and N = n + 1, such that L[i, j] = l(i, j).

Which one of the following statements would be TRUE regarding the dynamic programming solution for the recursive definition of l(i, j)?

A All elements of L should be initialized to 0 for the values of I(i, j) to be properly computed.

B The values of I(i,j) may be computed in a row major order or column major order of L[M, N].

C The values of I(i, j) cannot be computed in either row major order or column major order of L[M, N].

D L[p, q] needs to be computed before L[r, s] if either p < r or q < s.

GATE CSE 2009

Q. No.7

A sub-sequence of a given sequence is just the given sequence with some elements (possibly none or all) left out. We are given two sequences X[m] and Y[n] of lengths m and n, respectively with indexes of X and Y starting from 0.

We wish to find the length of the longest common sub-sequence (LCS) of X[m] and Y[n] as l(m,n), where an incomplete recursive definition for the function l(i,j) to compute the length of the LCS of X[m] and Y[n] is given below:

```
l(i,j)  = 0, if either i = 0 or j = 0
        = expr1, if i,j > 0 and X[i-1] = Y[j-1]
        = expr2, if i,j > 0 and X[i-1] ≠ Y[j-1]
```

Which one of the following options is correct?

Which one of the following options is correct?

A. expr1 = l(i − 1, j) + 1

B. expr1 = l(i, j − 1)

C. expr2 = max(l(i − 1, j), l(i, j − 1))

D. expr2 = max(l(i − 1,j − 1),l (i, j))

Q. No. 8

GATE CSE 2014 Set 2

Consider two strings A = "qpqrr" and B = "pqprqrp". Let x be the length of the longest common subsequence (not necessarily contiguous) between A and B and let y be the number of such longest common subsequences between A and B. Then x + 10y = ___.

A) 33 B) 23 C)43 D)34

Answer

Correct Answer is **34**

The LCS is of length 4. There are Y=3 LCS of length X=4 "qprr", "pqrr" and qpqr


Thank You

ALL PAIRS SHORTEST PATH

What is All Pairs Shortest Path Problem?

The all-pairs shortest path problem is the determination of the shortest distance between every pair of vertices in a given graph. We have to calculate the minimum cost to find the shortest path.

Procedure to find the all pairs shortest path:

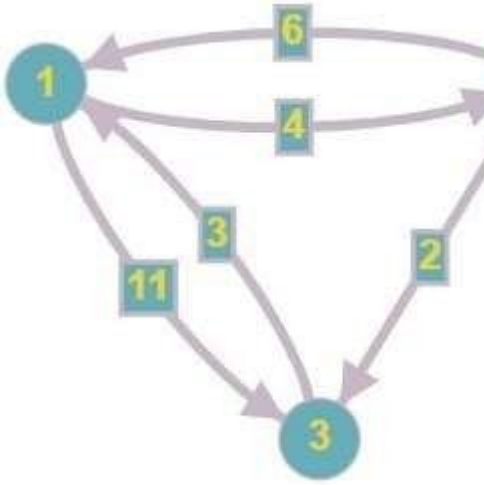
- First we consider “G” as a directed graph
- The cost of the graph is the length or cost of each edges and cost(i,i)=0
- If there is an edge between i and j then cost(i,j)=cost of the edge from i to j and if there is no edge then cost(i, j)= ∞
- Need to calculate the shortest path/ cost between any two nodes using intermediary nodes.
- The following equation is used to calculate the minimum
- cost $A^k(i,j)=\min\{A^{k-1}(i,j), A^{k-1}(i,k)+A^{k-1}(k,j)\}$

Algorithm of All Pairs Shortest Path

```
Algorithm AllPaths(cost, A, n)
  for i :=1to n do
        for j :=1to n do
              A[i, j]:=cost[i,j];
       for k :=1to n do
              for i :=1to n do
                    for j :=1to n do
                    A[i,j]:=min(A[i,j],A[i,k]+A[k,j];
```

Example:

Here, A^0=Cost= $\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$



When we calculate A^1 will omit column 1 and row 1 and calculate cost for rest of the 4 element.

$A^1(2,3)=\min\{A^{1-1}(2,3), A^{1-1}(2,1)+ A^{1-1}(1,3)\}$

$=\min\{2,17\} =2$

$A^1(3,2)=\min\{A^{1-1}(3,2), A^{1-1}(3,1)+ A^{1-1}(1,2)\}$

$=\min\{\infty,7\} =7$

$A^1=\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$

Now we will calculate A^2

$A^2(1,3)=\min\{A^{2-1}(1,3), A^{2-1}(1,2)+ A^{2-1}(2,3)\}$

$=\min\{11,6\}$

$=6$

$A^2(3,1)=\min\{A^{2-1}(3,1), A^{2-1}(3,2)+ A^{2-1}(2,1)\}$

$=\min\{3,13\}$

$=3$

$$A^2=\begin{bmatrix}0 & 4 & 6\\ 6 & 0 & 2\\ 3 & 7 & 0\end{bmatrix}$$

Now we will calculate A^3

$A^3(1,2)=\min\{A^{3-1}(1,2), A^{3-1}(1,3)+ A^{3-1}(3,2)\}$

$=\min\{4,13\}$

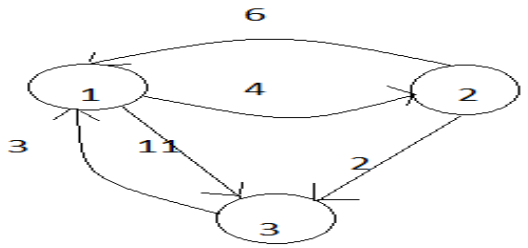
$=4$

$A^3(2,1)=\min\{A^{3-1}(2,1), A^{3-1}(2,3)+ A^{3-1}(3,1)\}$

$=\min\{6,5\}$

$=5$

$$A^3=\begin{bmatrix}0 & 4 & 6\\ 5 & 0 & 2\\ 3 & 7 & 0\end{bmatrix}$$



A^0	1	2	3
1	0	4	11
2	6	0	2
3	3		0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

Algorithm

```
lgorithm AllPaths(cost, A, n)

  for i := 1 to n do
      for   j:= 1 to n do
            A[i, j]:=cost[i, j];

     for  k := 1 to n do
      for  i := 1 to n do
       for   j:= 1 to n do
            A[i, j]:= min (A[i, j], A[i, k]+A[k, j]);
```

Applications of All Pairs Shortest Path

- Road Networking
- Network Routing
- Flight Reservations
- Driving Directions