Dynamic Programming

Module 2

Sem IV

AoA Even 2022-23

Introduction

Richard Bellman invented dynamic programming technique in 1950s

It is an algorithm design technique used for solving optimization problems.

Similar to Divide and Conquer technique as it divides original problem into several subproblems.

This technique starts with finding optimal solution for a smallest problem and then gradually enlarges the problem.

In dynamic programming, same subproblems are not solved once and results are stored in a table for later use.
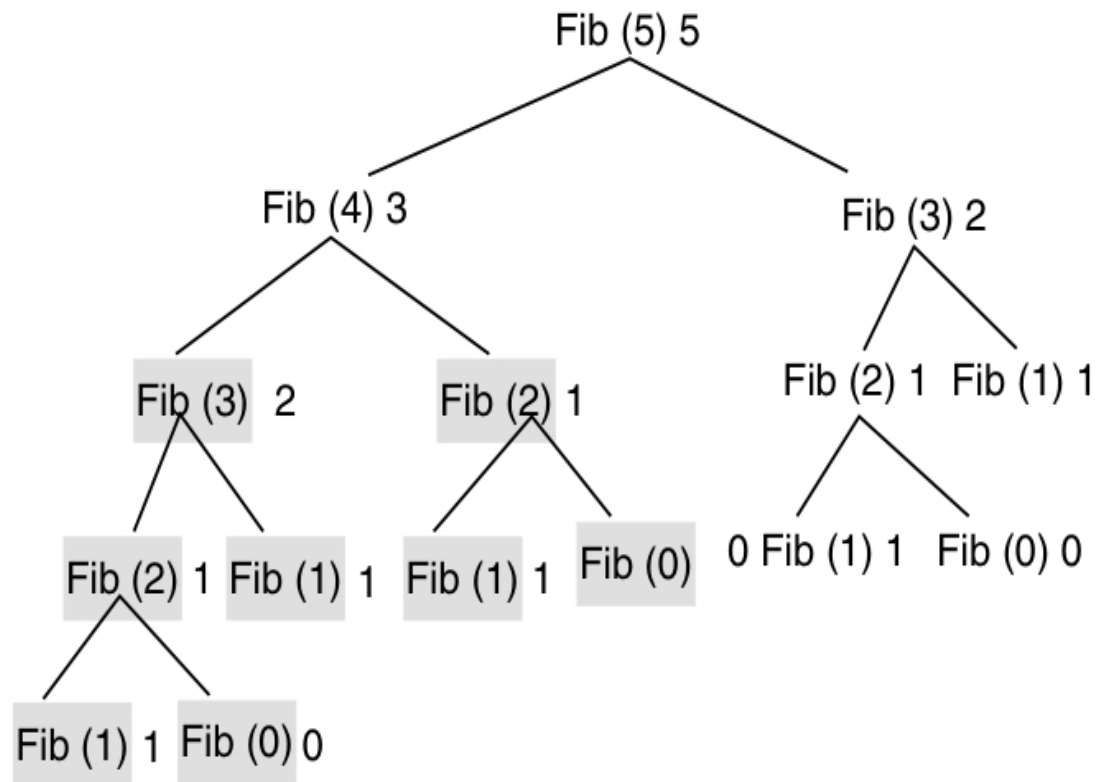
Example

Fibonacci Series:

- Recursive formula for calculating nth element of the Fibonacci series:

$$fib(n) = fib(n-1) + fib(n-2)$$

- When solving using D&C same functions are called again and again.

- Problem can be solved using Dynamic programming by storing the result of intermediate problems in table.

- Here many sub problems such as *fib(2), fib(1)* and *fib(0)*, which are overlapping.

- Here, dynamic programming technique follows bottom-up approach.

Example
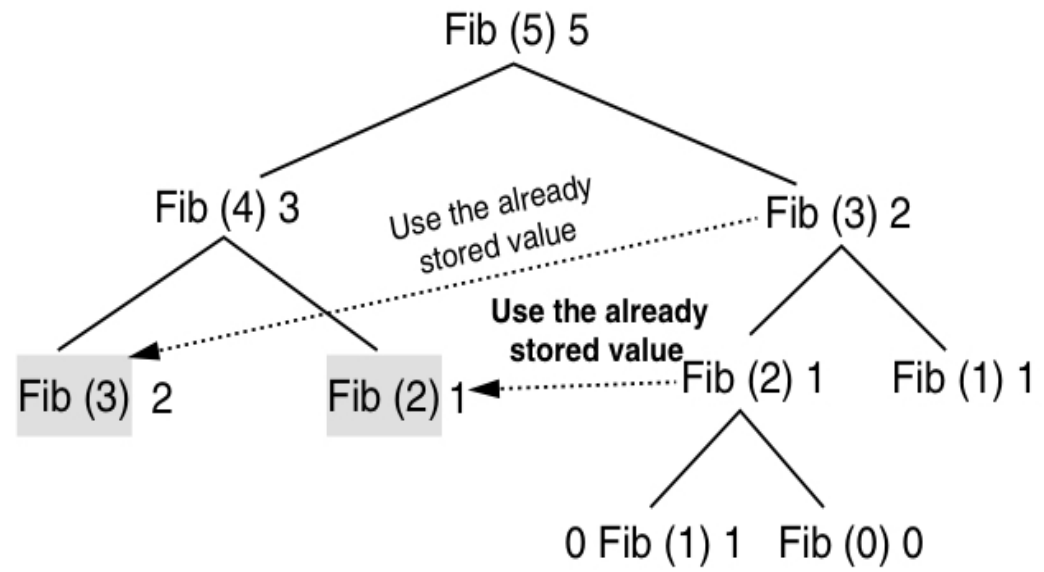

Divide-and-Conquer Technique

Dynamic Programming

(a)

(b)

Example

Fibonacci Series: Recursive Algorithm

```
fib(int n)
 {
       if ((n==0) || (n==1))
            return n;
       else
            return fib(n-1) + fib(n-2);
}
```

Example

Fibonacci Series:

- No. of external nodes in tree of fib(5) = 8 = fib(6)

- So, for value n, number of external nodes = fib(n+1)

- No. of internal nodes of fib(5) = 8 − 1

- So, for fib(n), no. Of internal nodes = fib(n+1)−1

- Hence total nodes in a tree = 2*fib(n+1) − 1

- No. Of recursive calls to fib is one less than number of nodes in tree

  = 2*fib(n+1) − 2

Example

Fibonacci Series:

- For Fibonacci series, f(n) is

$$fib(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

$$= \frac{1}{\sqrt{5}}\left[(1.618)^n - (-0.618)^n\right]$$

- For large values on n, neglecting $(-0.618)^n$, we get

$$fib(n) = \frac{1}{\sqrt{5}}(1.618)^n$$

- Hence, time complexity of recursive algorithm $= 2* \frac{1}{\sqrt{5}}(1.618)^n - 2 \Rightarrow O((1.618)^n)$

Example

Fibonacci Series:

- Dynamic Programming improves time complexity

- In dynamic programming, values are stored in a table for later use.

Fibonacci Series: Iterative Algorithm

```
fib(int a[],int n)
 {
     int n;
         if ((n==0) || (n==1))
             return n;
         a[0]=0; a[1]=1;
     for(i=2; i<=n; i++)
         a[i] = a[i-1] + a[i-2];
         return a[n];
}
```

The time complexity of Iterative algorithm is O(n), since loop runs n-1 times.

Characteristics of Dynamic Programming

1. Optimal Substructure:

- Problems can be decomposed into several subproblems.

- Optimal solutions to subproblems are found and solution of original problem is expressed in terms of solutions for smaller problems.

- Optimal solution contains optimal subsolution, then problem exhibits optimal substructure.

2. Overlapping Subproblems:

- When recursive algorithm revisits same subproblem over and over, the problem is said to have overlapping subproblem

Characteristics of Dynamic Programming

Principle of Optimality:

- Dynamic Programming works on a principle of optimality.

- Sequence of decision to be made at every stage in dynamic programming starting from smallest subproblem.

- The principle of optimality states that
   "*the optimal solution of a problem is a combination of optimal solutions to some of its subproblems*".

Applications of Dynamic Programming

- Longest Common Subsequence(LCS)

- 0/1 Knapsack Problem

- Travelling Salesman Problem

- Single Source shortest path (Bellman Ford Algorithm)

- All pair shortest path (Floyd Warshall Algorithm)

- Matrix chain Multiplication

- Multistage graphs

Homogeneous Recurrences

- Let $T(n)$ is denoted by t_n

- Consider homogeneous recurrences with constant coefficients i.e., recurrence of form

$$a_0 \, t_n + a_1 \, t_{n-1} + a_2 \, t_{n-2} + \cdots + a_k \, t_{n-k} = 0 \qquad \text{-----} ①$$

- This recurrence relation satisfies following properties:
  - It is linear, because it does not contain term of the form $t_{n-i} \times t_{n-j}$, t_{n-i}^2 etc.
  - It is homogeneous because the linear combination of t_{n-i} is zero.
  - It has constant coefficients since all a_i's are constant.

Homogeneous Recurrences

- For example, consider following recurrence of Fibonacci series

$$t_n = t_{n-1} + t_{n-2}$$

$$t_n - t_{n-1} - t_{n-2} = 0$$

This is homogeneous recurrence with

$$k=2, \; a_o = 1, \; a_1 = 1, \; a_2 = -1$$

Property of Homogeneous Recurrences

- If *f(n)* and *g(n)* are solutions to eq. ① then, their linear combination is also a solution to eq. ①

$$a_0\, f_n + a_1\, f_{n-1} + a_2\, f_{n-2} + \cdots + a_k\, f_{n-k} = 0$$

$$a_0\, g_n + a_1\, g_{n-1} + a_2\, g_{n-2} + \cdots + a_k\, g_{n-k} = 0$$

Proof: Let

$$t_n = cf_n + dg_n \quad \text{be solution of recurrence relation.}$$

substituting t_n in eq. ①

Property of Homogeneous Recurrences

$$t_n = a_0 (cf_n + dg_n) + a_1 (cf_{n-1} + dg_{n-1}) + a_2 (cf_{n-2} + dg_{n-2}) + \cdots + a_k (cf_{n-k} + dg_{n-k})$$

$$= c(a_0 f_n + a_1 f_{n-1} + a_2 f_{n-2} + \cdots + a_k f_{n-k}) +$$

$$d(a_0 g_n + a_1 g_{n-1} + a_2 g_{n-2} + \cdots + a_k g_{n-k})$$

$$= c(0) + d(0)$$

$$= 0$$

Hence, linear combination of solutions to eq. ① is also a solution to eq. ①

Homogeneous Recurrences

To find solution of eq. ① homogeneous recurrence

Let $t_n = x^n$

So, substituting in eq. ①

$$a_0\, x^n + a_1\, x^{n-1} + a_2\, x^{n-2} + \cdots + a_k\, x^{n-k} = 0$$

Multistage Graph

- A Multistage Graph G = {V,E} is a directed weighted graph.

- Vertices V of Graph G are partitioned into k(≥2) disjoint sets

$$I_1, I_2, \ldots I_i \ldots I_k \text{ such that } 1 \leq i \leq k$$

- If *(u,v)* is an edge E then $u \in V_i$ and $v \in Vi + 1$ *for some* $1 \leq i \leq k$ and c[i][j] is the cost of edge (i,j)

- Set $| I_1 | = | I_k | = 1$

Source   Sink
Vertex(S)   Vertex(T)

- There are set of vertices in each stage in a multistage graph

Multistage Graph

- Cost of path from S to T is calculated as the sum of weights along the edge on the path.

- The multistage graph problem is to find minimum cost path from a given source node to destination node.

- The minimum path in multistage graph is obtained by taking the minimum path at each current stage by considering the path length of each vertex obtained in earlier stage.

- The multistage graph problem holds the principle of optimality.

- Application: used to represent resource allocation

Multistage Graph

Use Recurrence Formula:

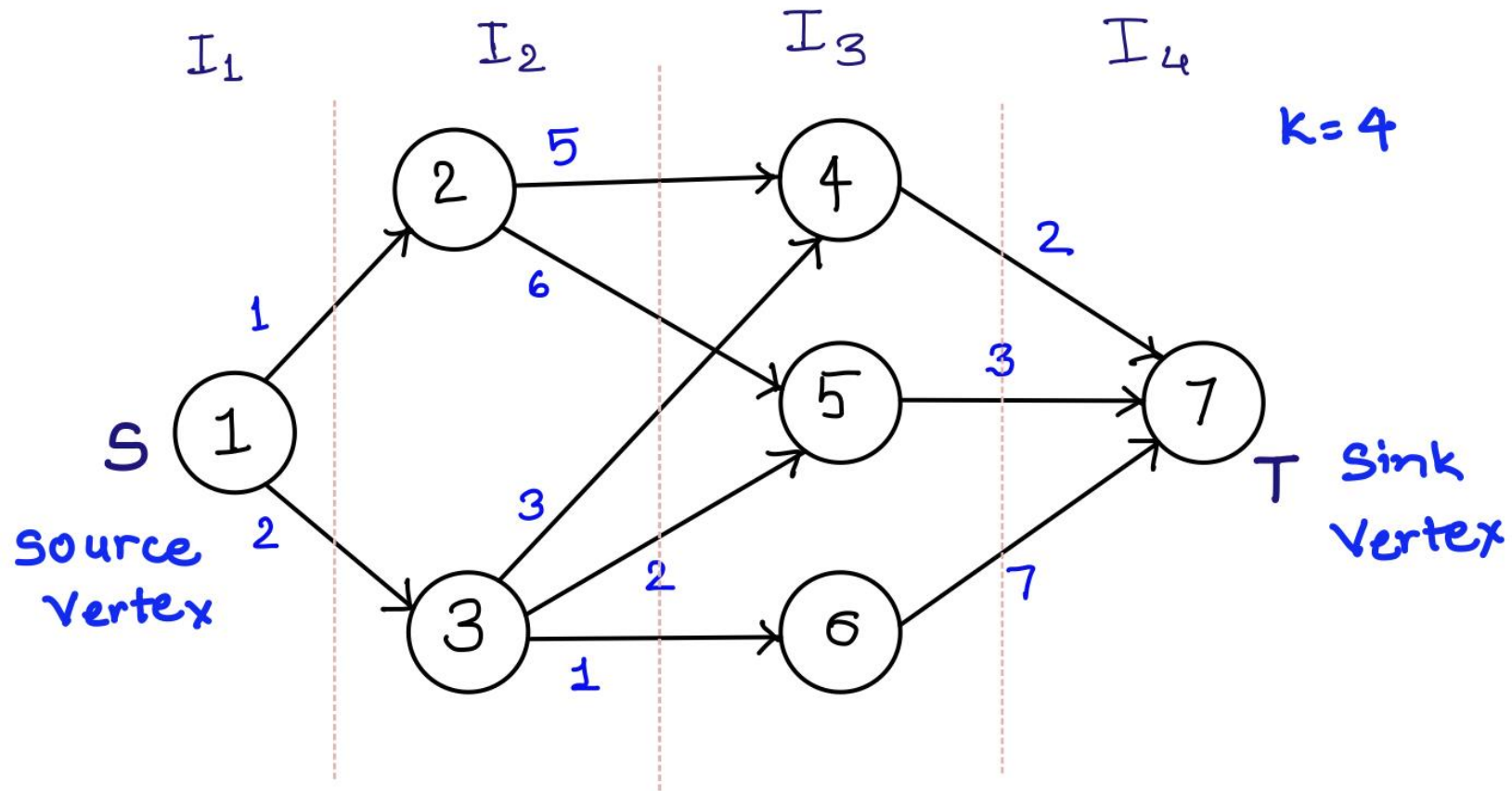- If cost[i][j] is cost of path from vertex j in stage I to destination vertex T, then cost of each vertex on every stage can be calculated using following recurrence formula.

$$cost[i][j] = c[j][intermediate] + cost[i+1][intermediate]$$

OR

$$cost[i][j] = min\{c[j][intermediate] + cost[i+1][intermediate]\}$$

Multistage Graph: Example

Multistage Graph: Methods to Solve

- **Forward Approach:**
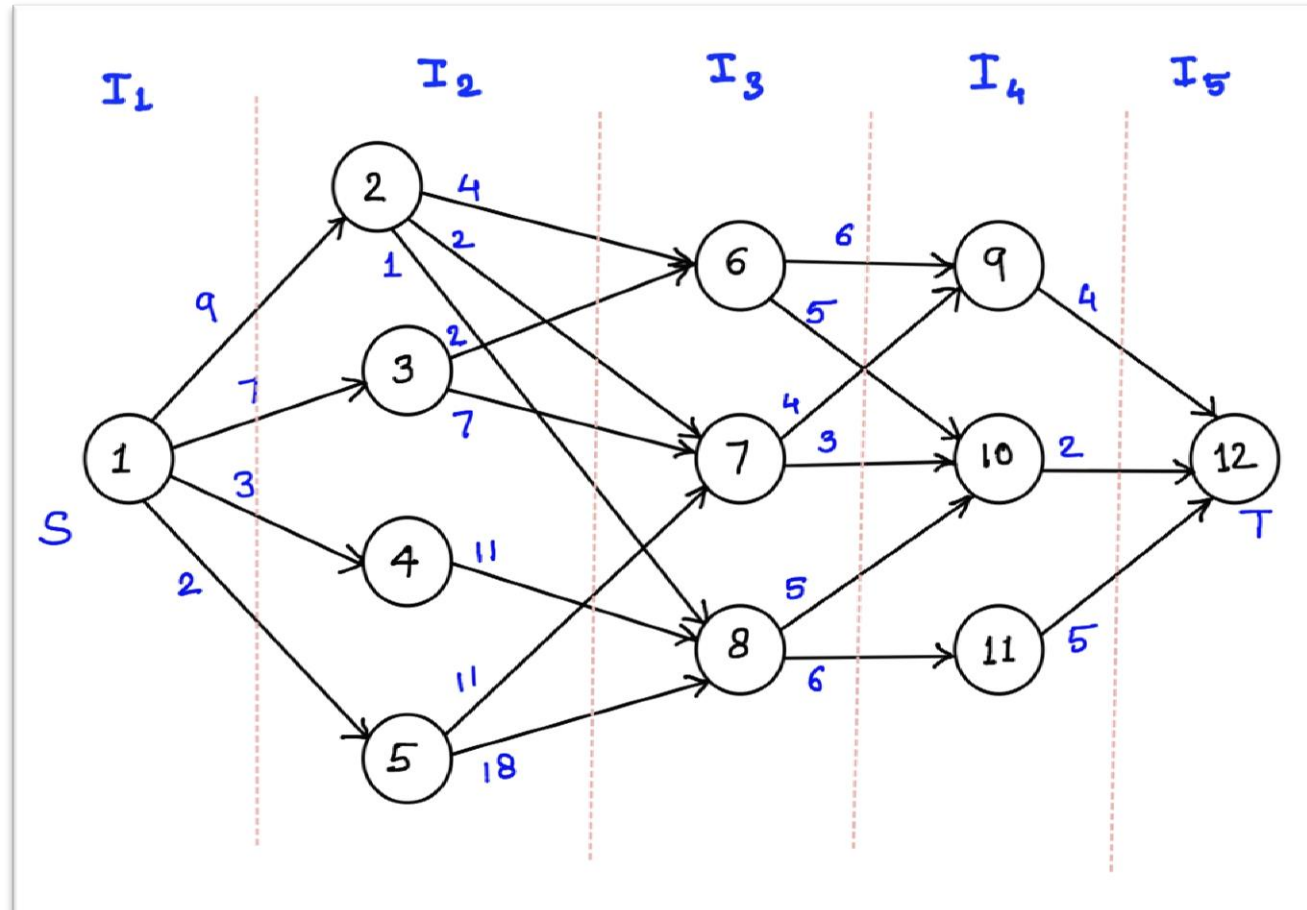 - here we start from destination vertex(i.e. Sink)

$$cost[i][j] = min\{c[j][intermediate] + cost[i+1][intermediate]\}$$

- **Backward Approach:**
  - Here we start from source vertex

$$cost[i][j] = min\{c[intermediate][j] + cost[i-1][intermediate]\}$$

Multistage Graph: (Forward Approach)

Vertex	1	2	3	4	5	6	7	8	9	10	11	12
cost			0									0
d												

Multistage Graph: (Forward Approach)

Stage 5:

i=5; Vertices(j) = {12}

cost[5][12]=0

Stage 4:

i=4; vertices(j) ={9,10,11}

cost[4][9]= c[9][12]+cost[5][12]

cost[4][9]=(4+0)=4

Similarly, cost[4][10] = 2

cost[4][11]=5

Stage 3:

i=3; Vertices(j) = {6,7,8}

cost[3][6]=min{(c[6][9] + cost [4][9]), (c[6][10] + cost [4][10])}

=min{(6+4),(5+2)} = 7

cost[3][7]= min{(c[7][9] + cost [4][9]), (c[7][10] + cost [4][10])}

=min{(4+4),(3+2)} = 5

cost[3][8]=min{(c[8][10] + cost [4][10]),(c[9][11] + cost [4][11])}

=min{(5+2),(6+5)} = 7

Vertex	1	2	3	4	5	6	7	8	9	10	11	12
cost						7	5	7	4	2	5	0
d						10	10	10	12	12	12	12

Multistage Graph: (Forward Approach)

Stage 2:

 i=2; Vertices(j) = {2, 3, 4, 5}

$cost[2][2]$=min{(c[2][6] + cost [3][6]), (c[2][7] + cost [3][7]), (c[2][8] + cost [3][8]) }

=min{(4+7), (2+5), (1+7)} = min{11,7,8} = 7

$cost[2][3]$= min{(c[3][6] + cost [3][6]), (c[3][7] + cost [3][7])}

=min{(2+7),(7+5)} = min{9,11} = 9

$cost[2][4]$=min{(c[4][8] + cost [3][8]) }

=min{(11+7)} = 18

$cost[2][5]$=min{(c[5][7] + cost [3][7]),(c[5][8] + cost [3][8])}

=min{(11+5),(8+7)} = min{16,15} = 15

Stage 1:

 i=1; Vertices(j) = {1}= Source

$cost[1][1]$ = min{(c[1][2] + cost [2][2]),

(c[1][3] + cost [2][3]),

(c[1][4] + cost [2][4]),

(c[1][5] + cost [2][5])}

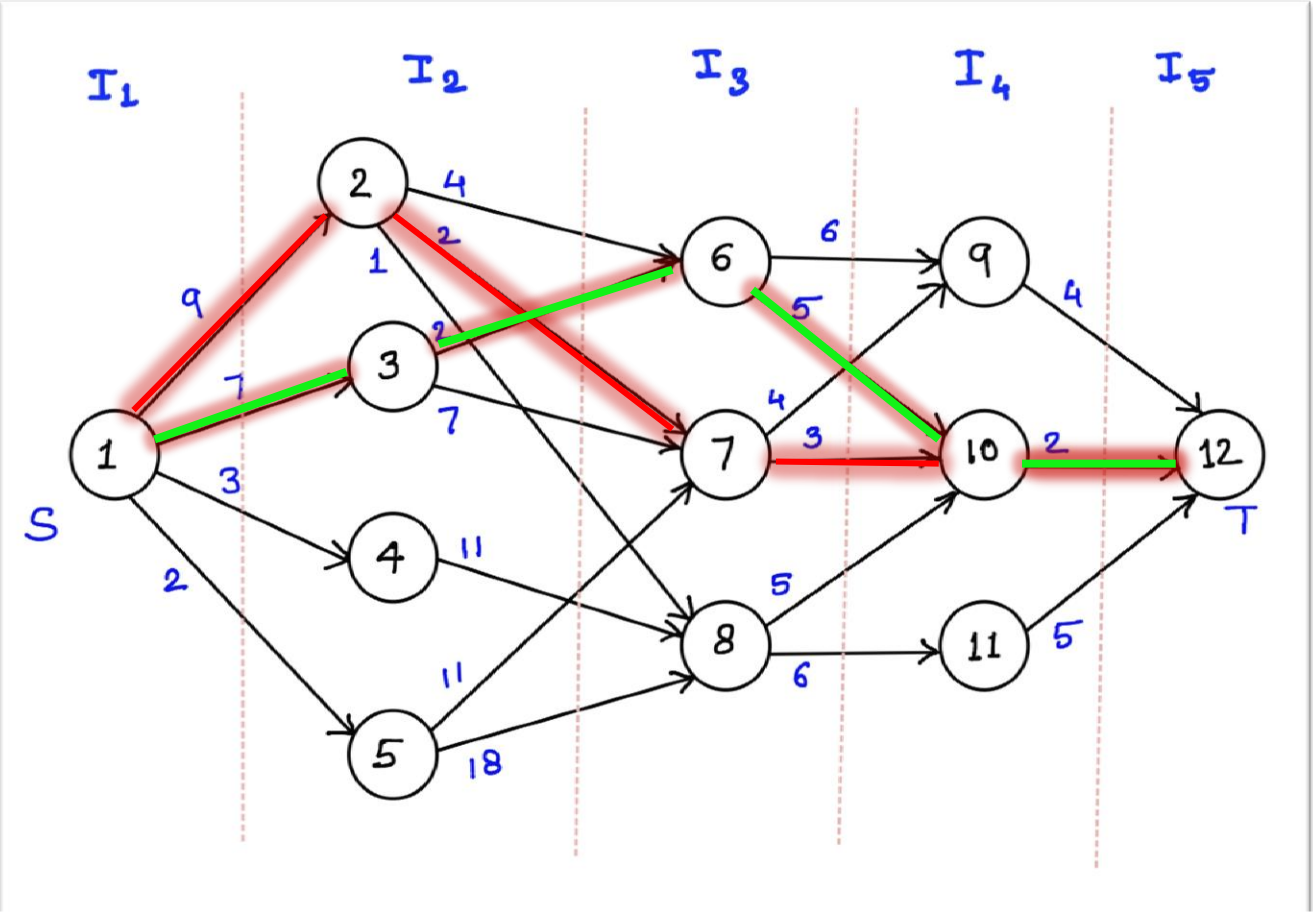= min{(9+7), (7+9), (3+18) +(2+15)}

= min{16, 16, 21, 17} = 16

Vertex	1	2	3	4	5	6	7	8	9	10	11	12
cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2 or 3	7	6	8	8	10	10	10	12	12	12	12

Multistage Graph: (Forward Approach)

Shortest Path 1:
1 – 2 – 7 – 10 – 12

Shortest Path 2:
1 – 3 – 6 – 10 – 12

Vertex	1	2	3	4	5	6	7	8	9	10	11	12
cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2 or 3	7	6	8	8	10	10	10	12	12	12	12

Multistage Graph: (Backward Approach)

Vertex	1	2	3	4	5	6	7	8	9	10
cost										
d										

Algorithm: (Forward Approach)

Algorithm 1 `MultiGraphforward(G, k, n, p)`

Input: The input is a k stage graph G = {V, E} with |V| = n vertices, E is a set of edges and `C[i][j]` is the cost of edge `(i, j)`. `C[n][n]` is the cost matrix of the given graph G. `Path[1...k]` is the array of minimum cost path.

Output: The minimum cost path array P

1. `Cost[n]=0;`
2. `For (j=n-1; j>=2; j--)`
3. `{`
4. `Cost[j]=C[j][r]+Cost[r]`

    `// Here r be a vertex such that <j, r> is an edge of G`

5. `d[j]=r`

    `// and(j, r)  ∈ E and C[j][r] + Cost[r]is minimum`

6. `}`
7. `Path[1]=1`
8. `Path[k]=n`
9. `For (j= 2; j<=k-1; j++)`
10. `{`
11. `Path[j]=d[Path[j-1]]`
12. `}`
13. `Stop`

Algorithm: (Backward Approach)

Algorithm 2 `MultiGraphbackward(G, k, n, p)`

Input: The input is a k stage graph G = {V, E} with |V| = n vertices, E is a set of edges and `C[i][j]` is the cost of edge `(i, j)`. `C[n][n]` is the cost matrix of the given graph G. `Path[1...k]` is the array of minimum cost path.
Output: The minimum cost path array P

1. `BCost[n]=0;`
2. `For (j=2; j<=n; j++)`
3. `{`
4. `BCost[j]= BCost[r]+ C[r][j]`
   `// Here r be a vertex such that <r, j> is an edge of G`
5. `Intermediate[j]=r`
   `// (j, r) ∈ E and BCost[r]+C[r][j] is minimum`
6. `}`
7. `Path[1]=1`
8. `Path[k]=n`
9. `For (j=k-1; j>=2 ; j--)`
10. `{`
11. `Path[j]= Intermediate [Path[j+1]]`
12. `}`
13. `Stop`

0/1 Knapsack Problem

- The objective of knapsack problem is to fill the knapsack with items to maximize the total value/profit, subject to its capacity.
- It is known as 0/1 knapsack problem , because we put one item into the knapsack or we will not include that item into the knapsack.
- Knapsack problem shows both overlapping subproblems and optimal substructure.
- The recursive definition for the optimum is given as follows:

$$S[k, u] = \begin{cases} 0 & \text{if } k=0 \text{ or } u=0 \\ S[k-1, u] & \text{if } u < w_k \\ \max\{S[k-1, u], S[k-1, u-w_k] + v_k\} & \text{if } u \geq w_k \end{cases}$$

0/1 Knapsack Problem: Example

Item (k)	1	2	3	4
Weight(w_k)	2	3	4	5
Value(v_k)	4	8	9	11

W = 5

Weight(u) →

Item(k)

	0	1	2	3	4	5
0						
1						
2						
3						
4						

0/1 Knapsack Problem: Example

Weight(u) → Item(k)	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

Step 1: k = 0; u = {0…5}
k=0 , u = 0
S[k,u] = 0 for u={1,2,3,4,5}

0/1 Knapsack Problem: Example

Step 2: k = 1; u = {0...5}
$k=1$, $w_1=2$, $u = 0$;
 $u<w_1$,
$S[k,u] = S[k-1,u]$
$S[1,0] = S[0,0] = 0$

Step 2: k = 1; u = {0...5}
$k=1$, $w_1=2$, $u = 1$;
 $u<w_1$,
$S[k,u] = S[k-1,u]$
$S[1,1] = S[0,1] = 0$

Step 2: k = 1; u = {0...5}
$k=1$, $w_1=2$, $u = 2$;
 $u=w_1$,
$S[k,u] = \max\{S[k-1,u], S[k-1, u-w_k] + v_k\}$
$S[1,2] = \max\{S[0,2], S[0, 0] + 4\} = \max\{0,0+4\} = 4$

Step 2: k = 1; u = {0...5}
$k=1$, $w_1=2$, $u = 3$;
 $u>w_1$,
$S[k,u] = \max\{S[k-1,u], S[k-1, u-w_k] + v_k\}$
$S[1,3] = \max\{S[0,3], S[0,1] + 4\} = \max\{0,0+4\} = 4$

Step 2: k = 1; u = {0...5}
$k=1$, $w_1=2$, $u = 4$;
 $u>w_1$,
$S[k,u] = \max\{S[k-1,u], S[k-1, u-w_k] + v_k\}$
$S[1,4] = \max\{S[0,4], S[0,2] + 4\} = \max\{0,0+4\} = 4$

Step 2: k = 1; u = {0...5}
$k=1$, $w_1=2$, $u = 5$;
 $u>w_1$,
$S[k,u] = \max\{S[k-1,u], S[k-1, u-w_k] + v_k\}$
$S[1,5] = \max\{S[0,5], S[0,3] + 4\} = \max\{0,0+4\} = 4$

0/1 Knapsack Problem: Example

Weight(u) → Item(k)	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	4	4	4	4
2						
3						
4						

0/1 Knapsack Problem: Example

Step 3: k = 2; u = {0…5}
k=2, w_2=3, u = 0;
 $u<w_2$,
S[k,u] = S[k-1,u]
S[2,0] = S[1,0] = 0

Step 3: k = 2; u = {0…5}
k=2, w_2=3, u = 3;
 $u=w_1$,
S[k,u] = max{S[k-1,u], S[k-1, u-w_k] + v_k}
S[2,3] = max{S[1,3], S[1,0] + 8} = max{4,0+8} = 8

Step 3: k = 2; u = {0…5}
k=2, w_2=3, u = 1;
 $u<w_2$,
S[k,u] = S[k-1,u]
S[2,1] = S[1,1] = 0

Step 3: k = 2; u = {0…5}
k=2, w_2=3, u = 4;
 $u>w_1$,
S[k,u] = max{S[k-1,u], S[k-1, u-w_k] + v_k}
S[2,4] = max{S[1,4], S[1,1] + 8} = max{4,0+8} = 8

Step 3: k = 2; u = {0…5}
k=2, w_2=3, u = 2;
 $u<w_2$,
S[k,u] = S[k-1,u]
S[2,2] = S[1,2]= 4

Step 3: k = 2; u = {0…5}
k=2, w_2=3, u = 5;
 $u>w_1$,
S[k,u] = max{S[k-1,u], S[k-1, u-w_k] + v_k}
S[2,5] = max{S[1,5], S[1,2] + 8} = max{4,4+8} = 12

0/1 Knapsack Problem: Example

Weight(u) → Item(k)	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	4	4	4	4
2	0	0	4	8	8	12
3						
4						

0/1 Knapsack Problem: Example

Step 4: k = 3; u = {0...5}
k=3, w_3=4, u = 0;
 u<w_3,
S[k,u] = S[k-1,u]
S[3,0] = S[2,0] = 0

Step 4: k = 3; u = {0...5}
k=3, w_3=4, u = 3;
 u<w_3,
S[k,u] = S[k-1,u]
S[3,3] = S[2,3] = 8

Step 4: k = 3; u = {0...5}
k=3, w_3=4, u = 1;
 u<w_3,
S[k,u] = S[k-1,u]
S[3,1] = S[2,1] = 0

Step 4: k = 3; u = {0...5}
k=3, w_3=4, u = 4;
 u=w_3,
S[k,u] = max{S[k-1,u], S[k-1, u-w_k] + v_k}
S[3,4] = max{S[2,4], S[2,0] + 8} = max{8,0+9} = 9

Step 4: k = 3; u = {0...5}
k=3, w_3=4, u = 2;
 u<w_3,
S[k,u] = S[k-1,u]
S[3,2] = S[2,2]= 4

Step 4: k = 3; u = {0...5}
k=3, w_3=4, u = 5;
 u>w_3,
S[k,u] = max{S[k-1,u], S[k-1, u-w_k] + v_k}
S[3,5] = max{S[2,5], S[2,1] +11 } = max{12,0+11} = 12

0/1 Knapsack Problem: Example

Weight(u) → Item(k)	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	4	4	4	4
2	0	0	4	8	8	12
3	0	0	4	8	9	12
4						

0/1 Knapsack Problem: Example

Step 5: k = 4; u = {0...5}
k=4, w_4=5, u = 0;
 u<w_4,
S[k,u] = S[k-1,u]
S[4,0] = S[3,0] = 0

Step 5: k = 4; u = {0...5}
k=4, w_4=5, u = 3;
 u<w_4,
S[k,u] = S[k-1,u]
S[4,3] = S[3,3] = 8

Step 5: k = 4; u = {0...5}
k=4, w_4=5, u = 1;
 u<w_4,
S[k,u] = S[k-1,u]
S[4,1] = S[3,1] = 0

Step 5: k = 4; u = {0...5}
k=4, w_4=5, u = 4;
 u<w_4,
S[k,u] = S[k-1,u]
S[4,4] = S[3,4] = 9

Step 5: k =4; u = {0...5}
k=4, w_4=5, u = 2;
 u<w_4,
S[k,u] = S[k-1,u]
S[4,2] = S[3,2]= 4

Step 5: k = 4; u = {0...5}
k=4, w_4=5, u = 5;
 u=w_4,
S[k,u] = max{S[k-1,u], S[k-1, u-w_k] + v_k}
S[4,5] = max{S[3,5], S[3,0] +11 } = max{12,0+11} = 12

0/1 Knapsack Problem: Example

Weight(u) → Item(k)	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	4	4	4	4
2	0	0	4	8	8	12
3	0	0	4	8	9	12
4	0	0	4	8	9	12

optimal profit of the given knapsack problem is 12

0/1 Knapsack Problem

To find the items of the optimal cost:

1	`if S[k][u] ≠ S[k - 1][u],`	`//S[k][u] and S[k - 1][u] are different`
2	`then, kth item is selected i.e.`	`//k=n and u=W(Capacity of Knapsack)`
3	`print k and set u = u–wk`	
4	`k = k–1`	
5	`if S[k][u] = S[k - 1][u] then,`	
6	`set k = k–1`	

0/1 Knapsack Problem: Example

Weight(u) → Item(k)	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	4	4	4	4
2	0	0	4	8	8	12
3	0	0	4	8	9	12
4	0	0	4	8	9	12

optimal profit

$I = \{\ 0\ ,\ 0\ ,\ 0\ ,\ 0\ \}$

$I = \{\ 0\ ,\ 1\ ,\ 0\ ,\ 0\ \}$

$I = \{\ 1\ ,\ 1\ ,\ 0\ ,\ 0\ \}$

S[1][2]≠ S[0][2]
So, k=1-1=0 & u=u-w_2 =2-2=0

S[2][4]≠ S[1][4]
So, k=2-1=1 & u=u-w_2 =5-3=2

S[3][4]= S[2][4]
So, k=3-1=2

S[4][4]= S[3][4]
So, k=4-1=3

0/1 Knapsack Problem: Algorithm

Algorithm 5.1: knapsack($w[1..n]$, $v[1..n]$, W)

1. For $u = 0$ to W
2. $S[0, u] = 0$
3. Endfor
4. For $i = 0$ to n
5. $S[i, 0] = 0$
6. Endfor
7. For $i = 0$ to n
8.   For $u = 0$ to W
9.   If($w[i] \leq u$) then   // item fits in knapsack
10.   If($v[i] + S[i-1, u-w[i]] > S[i-1, u]$) then
11. $S[i, u] = v[i] + S[i-1, u-w[i]]$
12.   Else $S[i, u] = S[i-1, u]$
13.   Else $S[i, u] = S[i-1, u]$
14.   Endfor
15. Endfor
16. Print $S[n][W]$ // max value of items included in the knapsack

0/1 Knapsack Problem: Algorithm

```
17      // Find knapsack items
18.     i = n,  k = W
19.     While( i > 0 and k > 0)
20.           If( S[i][k] ≠ S[i-1][k] )
21.                 Print i              // item i is in the knapsack
22.                 k = k - w[i]
23.           Endif
24.           i = i - 1
25.     Endwhile
```

All Pair Shortest Path

Floyd-Warshall Algorithm:

- Floyd's Algorithm is for finding shortest path between every pair of vertices of a graph.

- Works for both directed and undirected graph.

- The graph may contain negative edges, but it should not contain negative cycles.

- Floyd's algorithm requires a weighted graph

- Floyd's algorithm computes distance matrix of weighted graph.

- It takes dynamic programming approach and runs in $O(n^3)$ times.

All Pair Shortest Path

Floyd-Warshall Algorithm:

▪ Distance matrix of a weighted graph is computed with n vertices through a series of $n \times n$ matrices

$$D^0, D^1, D^2, D^3, \ldots\ldots D^k, \ldots\ldots D^{n-1}$$

where,

D^0 is weighted matrix with no intermediate vertex in its path between source i to destination j.

D^k is weighted matrix with k as intermediate vertex between i and j, k vary from $1 \ to \ k-1$, and

D^n is last matrix contains length of shortest paths among all paths that can use all vertices as intermediate
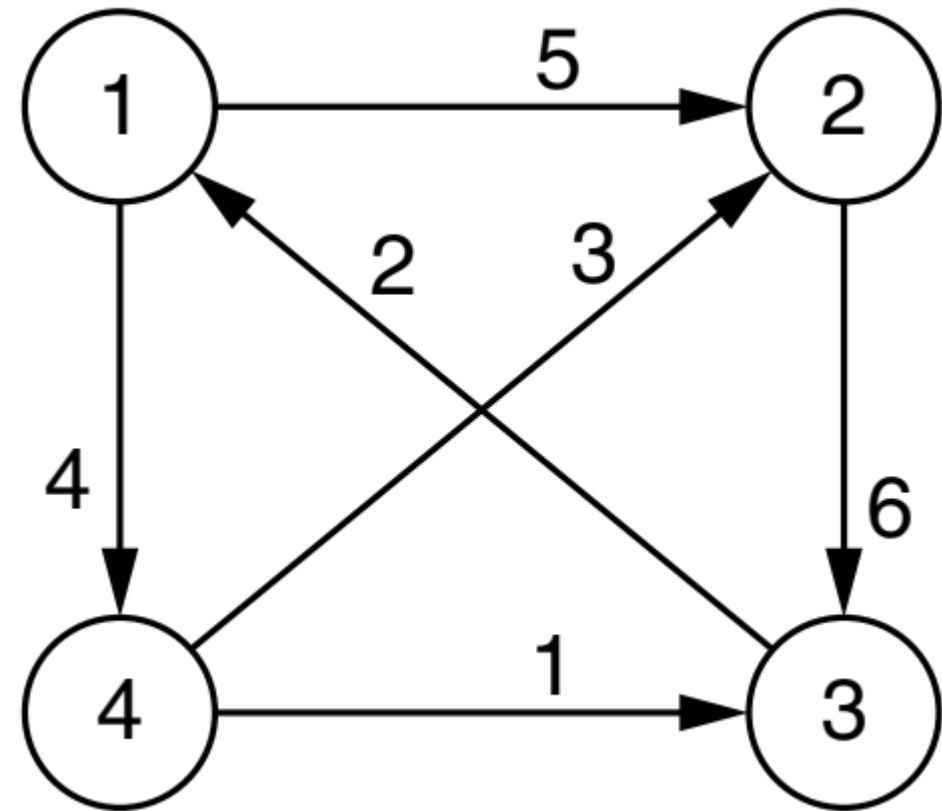
All Pair Shortest Path: Algorithm

Floyd-Warshall Algorithm:

Recursive formula:

$$D^k[i][j] = min\{D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][j]\}$$

Here, old values of D^{k-1} is overwritten by new values of D^k

Algorithm 5.5: All-pairsShortestPath ($W[1..n]$, n)

1.   Initialize matrix D with weight matrix W.
2.   For $k = 1$ to n
3.   For $i = 1$ to n
4.   For $j = 1$ to n
5. $D[i][j] = min\{ D[i][j], D[i][k] + C[k][j] \}$
6.   Endfor
7.   Endfor
8.   Endfor

All Pair Shortest Path: Example

Floyd-Warshall Algorithm:

Recursive formula:

$$D^k[i][j] = min\{D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][j]\}$$

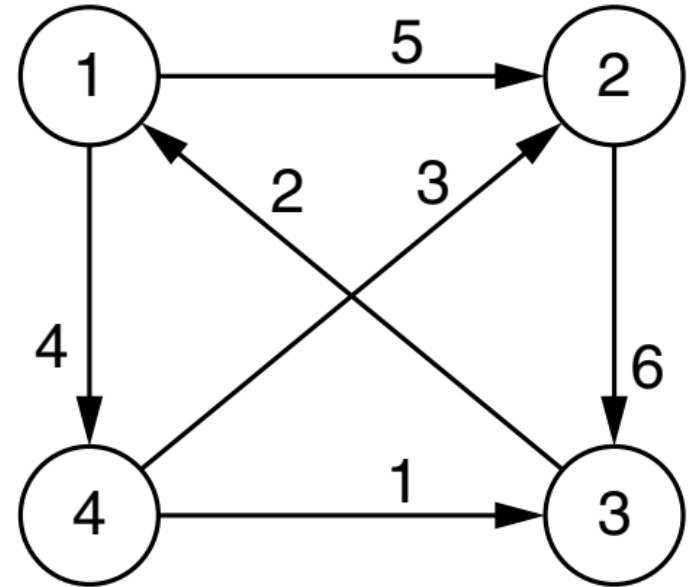Here, old values of D^{k-1} is overwritten by new values of D^k

All Pair Shortest Path: Example

Floyd-Warshall Algorithm:

Path matrix P is also initialized to NULL (0) entries.

$$
P^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array}
\left[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
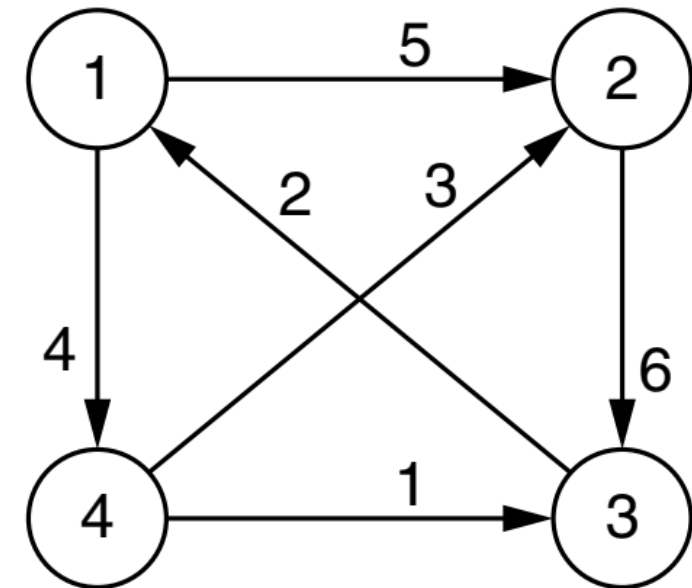0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{array}
\right]
$$

All Pair Shortest Path: Example

Floyd-Warshall Algorithm:

D^1 : Node 1 as intermediate node

$$P^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$D^1 = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \begin{bmatrix} d_1^{(1)}{}_1 & d_1^{(1)}{}_2 & d_1^{(1)}{}_3 & d_1^{(1)}{}_4 \\ d_2^{(1)}{}_1 & d_2^{(1)}{}_2 & d_2^{(1)}{}_3 & d_2^{(1)}{}_4 \\ d_3^{(1)}{}_1 & d_3^{(1)}{}_2 & d_3^{(1)}{}_3 & d_3^{(1)}{}_4 \\ d_4^{(1)}{}_1 & d_4^{(1)}{}_2 & d_4^{(1)}{}_3 & d_4^{(1)}{}_4 \end{bmatrix} = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \begin{bmatrix} 0 & 5 & \infty & 4 \\ \infty & 0 & 6 & \infty \\ 2 & 7 & 0 & 6 \\ \infty & 3 & 1 & 0 \end{bmatrix}$$

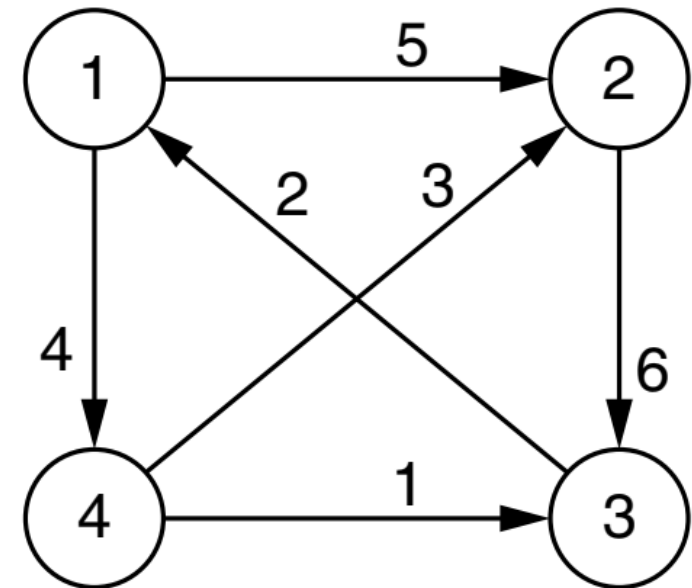# All Pair Shortest Path: Example

Floyd-Warshall Algorithm:

D^2 : Node 2 as intermediate node

$$P^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \begin{bmatrix} 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \begin{bmatrix} d_1^{(1,2)}{}_1 & d_1^{(1,2)}{}_2 & d_1^{(1,2)}{}_3 & d_1^{(1,2)}{}_4 \\ d_2^{(1,2)}{}_1 & d_2^{(1,2)}{}_2 & d_2^{(1,2)}{}_3 & d_2^{(1,2)}{}_4 \\ d_3^{(1,2)}{}_1 & d_3^{(1,2)}{}_2 & d_3^{(1,2)}{}_3 & d_3^{(1,2)}{}_4 \\ d_4^{(1,2)}{}_1 & d_4^{(1,2)}{}_2 & d_4^{(1,2)}{}_3 & d_4^{(1,2)}{}_4 \end{bmatrix} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \begin{bmatrix} 0 & 5 & 11 & 4 \\ \infty & 0 & 6 & \infty \\ 2 & 7 & 0 & 6 \\ \infty & 3 & 1 & 0 \end{bmatrix}$$

All Pair Shortest Path: Example
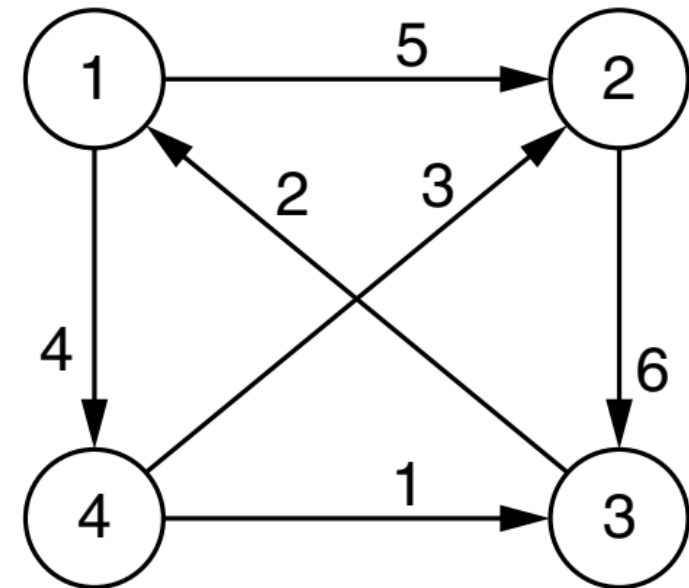
Floyd-Warshall Algorithm:

D^3 : Node 3 as intermediate node

$$P^3 = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 0 & 0 & \mathbf{2} & 0 \\ \mathbf{3} & 0 & 0 & \mathbf{3} \\ 0 & \mathbf{1} & 0 & \mathbf{1} \\ \mathbf{3} & 0 & 0 & 0 \end{bmatrix}$$

$$D^3 = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} d_1^{(1,2,3)}{}_1 & d_1^{(1,2,3)}{}_2 & d_1^{(1,2,3)}{}_3 & d_1^{(1,2,3)}{}_4 \\ d_2^{(1,2,3)}{}_1 & d_2^{(1,2,3)}{}_2 & d_2^{(1,2,3)}{}_3 & d_2^{(1,2,3)}{}_4 \\ d_3^{(1,2,3)}{}_1 & d_3^{(1,2,3)}{}_2 & d_3^{(1,2,3)}{}_3 & d_3^{(1,2,3)}{}_4 \\ d_4^{(1,2,3)}{}_1 & d_4^{(1,2,3)}{}_2 & d_4^{(1,2,3)}{}_3 & d_4^{(1,2,3)}{}_4 \end{bmatrix} = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 0 & 5 & 11 & 4 \\ \mathbf{8} & 0 & 6 & \mathbf{12} \\ 2 & 7 & 0 & 6 \\ \mathbf{3} & 3 & 1 & 0 \end{bmatrix}$$

All Pair Shortest Path: Example

D^4 : Node 4 as intermediate node

$$
P^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}
\begin{array}{cccc} 1 & 2 & 3 & 4 \end{array}
\begin{bmatrix}
0 & 0 & \mathbf{4} & 0 \\
\mathbf{3} & 0 & 0 & \mathbf{3} \\
0 & 1 & 0 & 1 \\
\mathbf{3} & 0 & 0 & 0
\end{bmatrix}
$$

$$
D^4 = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array}
\begin{bmatrix}
d_1^{(1,2,3,4)}{}_1 & d_1^{(1,2,3,4)}{}_2 & d_1^{(1,2,3,4)}{}_3 & d_1^{(1,2,3,4)}{}_4 \\
d_2^{(1,2,3,4)}{}_1 & d_2^{(1,2,3,4)}{}_2 & d_2^{(1,2,3,4)}{}_3 & d_2^{(1,2,3,4)}{}_4 \\
d_3^{(1,2,3,4)}{}_1 & d_3^{(1,2,3,4)}{}_2 & d_3^{(1,2,3,4)}{}_3 & d_3^{(1,2,3,4)}{}_4 \\
d_4^{(1,2,3,4)}{}_1 & d_4^{(1,2,3,4)}{}_2 & d_4^{(1,2,3,4)}{}_3 & d_4^{(1,2,3,4)}{}_4
\end{bmatrix}
=
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array}
\begin{bmatrix}
0 & 5 & \mathbf{5} & 4 \\
8 & 0 & 6 & 12 \\
2 & 7 & 0 & 6 \\
3 & 3 & 1 & 0
\end{bmatrix}
$$
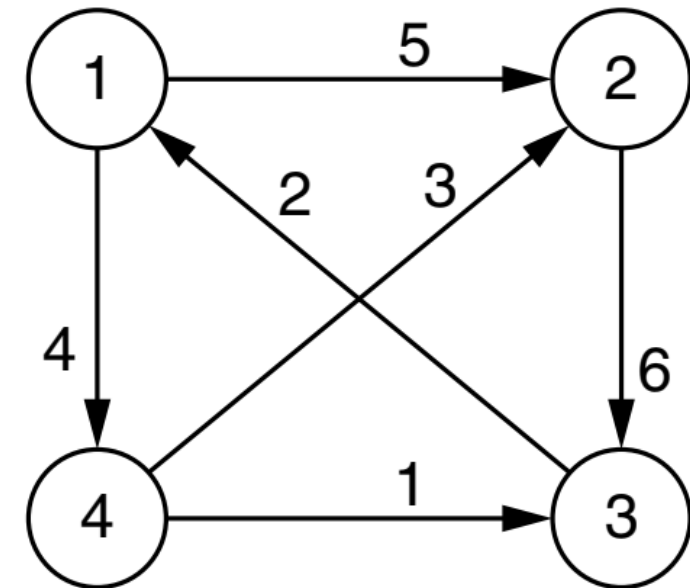
Single Source Shortest Path

Bellman Ford Algorithm:

- If graph has negative edge cost, then Dijkstra's Algorithm does not work.

- Bellman ford proposed an algorithm, which is combination of Dijkstra's algorithm and unweighted algorithms.

- Slower than Dijkstra's but more versatile.

- If there exists negative cycle in a graph, then there will be no shortest path.

- Relaxation Formula:
```
If (d[v]> d[u]+cost[u,v])
     then, d[v]= d[u]+cost[u,v]
```

Single Source Shortest Path

Bellman Ford Algorithm: Example
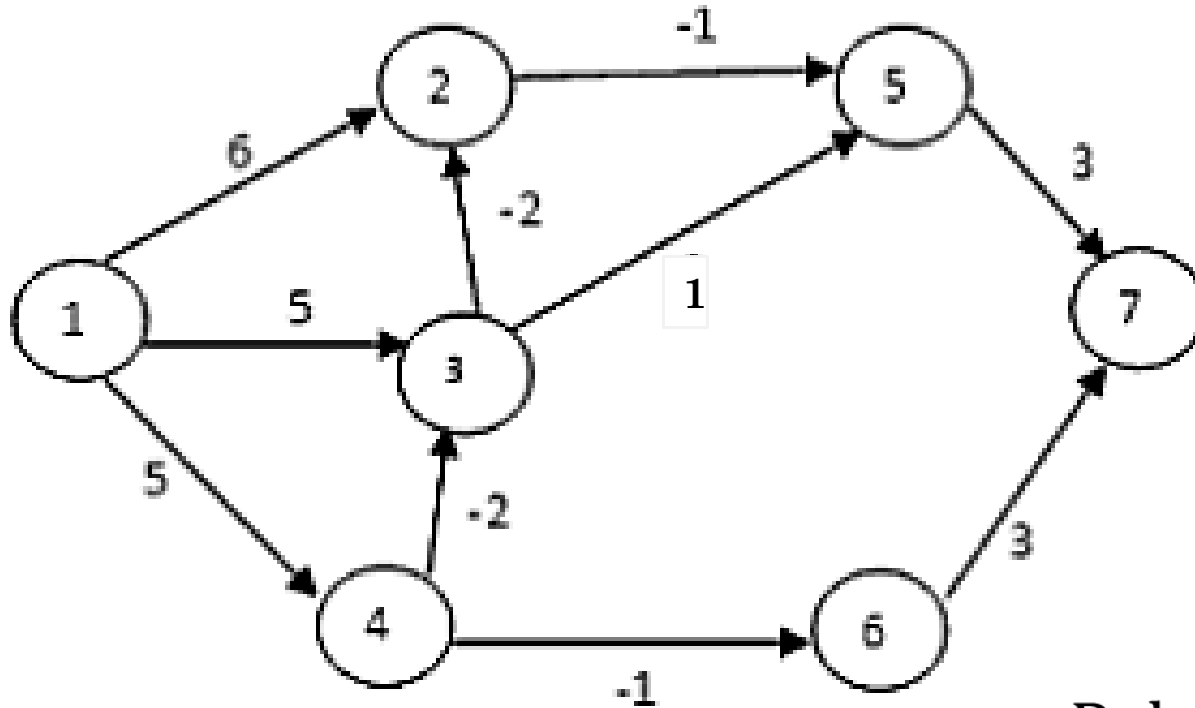
EdgeList={(1,2), (1,3), (1,4), (4,3), (3,2), (2,5), (3,5), (4,6), (5,7), (6,7) }

- Relaxation Formula:
  ```
  If (d[v]> d[u]+cost[u,v])
          then, d[v]= d[u]+cost[u,v]
  ```

Single Source Shortest Path

Bellman Ford Algorithm: Example

EdgeList={(1,2), (1,3), (1,4), (4,3), (3,2), (2,5), (3,5), (4,6), (5,7), (6,7) }

		1	2	3	4	5	6	7
Initialization	0	0	∞	∞	∞	∞	∞	∞
k=	1	0	6	5	5	∞	∞	∞
	2	0	3	3	5	5	4	∞
	3	0	1	3	5	2	4	5
	4	0	1	3	5	0	4	3
	5	0	1	3	5	0	4	3
	6	0	1	3	5	0	4	3

Path Length (k) = |V| -1 = 7-1 = 6

Single Source Shortest Path

Bellman Ford Algorithm:

Algorithm 1: Bellman-Ford Algorithm

Data: Given a directed graph G(V, E) , the starting vertex S, and the weight W of each edge

Result: Shortest path from S to all other vertices in G

$D[S] = 0$;

$R = V - S$;

$C = cardinality(V)$;

for *each vertex* $k \in R$ **do**
   | $D[k] = \infty$;
end

for *each vertex* $i = 1$ *to* $(C - 1)$ **do**
    for *each edge* $(e1, e2) \in E$ **do**
     | $Relax(e1, e2)$;
    end
end

for *each edge* $(e1, e2) \in E$ **do**
    if $D[e2] > D[e1] + W[e1, e2]$ **then**
     | $Print("Graph \ contains \ negative \ weight \ cycle")$;
    end
end

Procedure *Relax* $(e1, e2)$

for *each edge* $(e1, e2)$ *in* E **do**
    if $D[e2] > D[e1] + W[e1, e2]$ **then**
     | $D[e2] = D[e1] + W[e1, e2]$;
    end
end

The Travelling Salesman Problem

- Trying all the possibilities – **Brute Force Approach**.

- There are **n** cities

- Salesman has to visit all cities, starting from a city.

- He is not allowed to visit any city more than once and come back to starting city.

- Cost of travelling cities is given.

- We have to find tour of salesman with lowest cost.

The Travelling Salesman Problem

- Let **G(V,E)** be the directed graph such that $|V|=n$.

- Vertices of a graph represent cities

- Weight associated with each edge gives cost of traversal(represented by cost matrix) between the cities connected by the edge.

- Let starting vertex is 1.

- Tour starts at vertex 1 and goes to $k \in V - \{1\}$

- From vertex k, there is tour to vertex 1, which goes through vertex $V - \{1, k\}$, only once.

- Tour from vertex k to 1 must be optimal.

- This problem satisfies optimal substructure property.

The Travelling Salesman Problem

- Recursive Formula:

$$SP(i, S) = \min_{k \in S}[d(i, k) + SP(k, S - \{k\})]$$

where,
  V= set of all vertices
  S = set of remaining or unvisited vertices →subset of V
  i = source vertex
  k = intermediate vertex
  and, SP(i, S) = shortest path from i to all other unvisited vertices.

The Travelling Salesman Problem

- Algorithm:

Algorithm 1: Dynamic Approach for TSP

Data: s: starting point; N: a subset of input cities; $dist()$: distance among the cities

Result: $Cost$: TSP result

$Visited[N] = 0$;

$Cost = 0$;

Procedure TSP(N, s)

  $Visited[s] = 1$;

  if $|N| = 2$ *and* $k \neq s$ **then**

    $Cost(N, k) = dist(s, k)$;

    Return Cost;

  else

    for $j \in N$ **do**

      for $i \in N$ *and* $visited[i] = 0$ **do**

        if $j \neq i$ *and* $j \neq s$ **then**

          $Cost(N, j) = \min (TSP(N - \{i\}, j) + dist(j, i))$

          $Visited[j] = 1$;
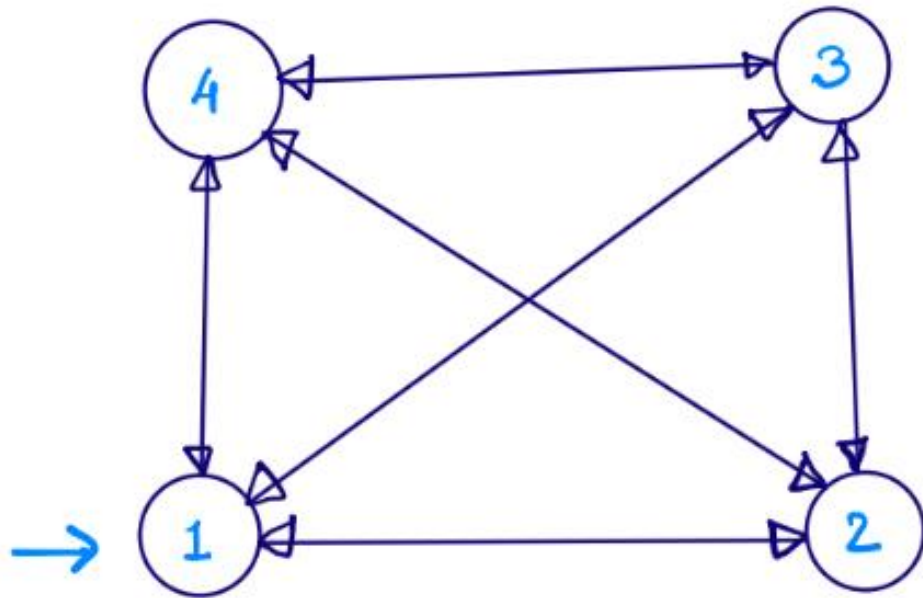
        end

      end

    end

  end

  Return $Cost$;
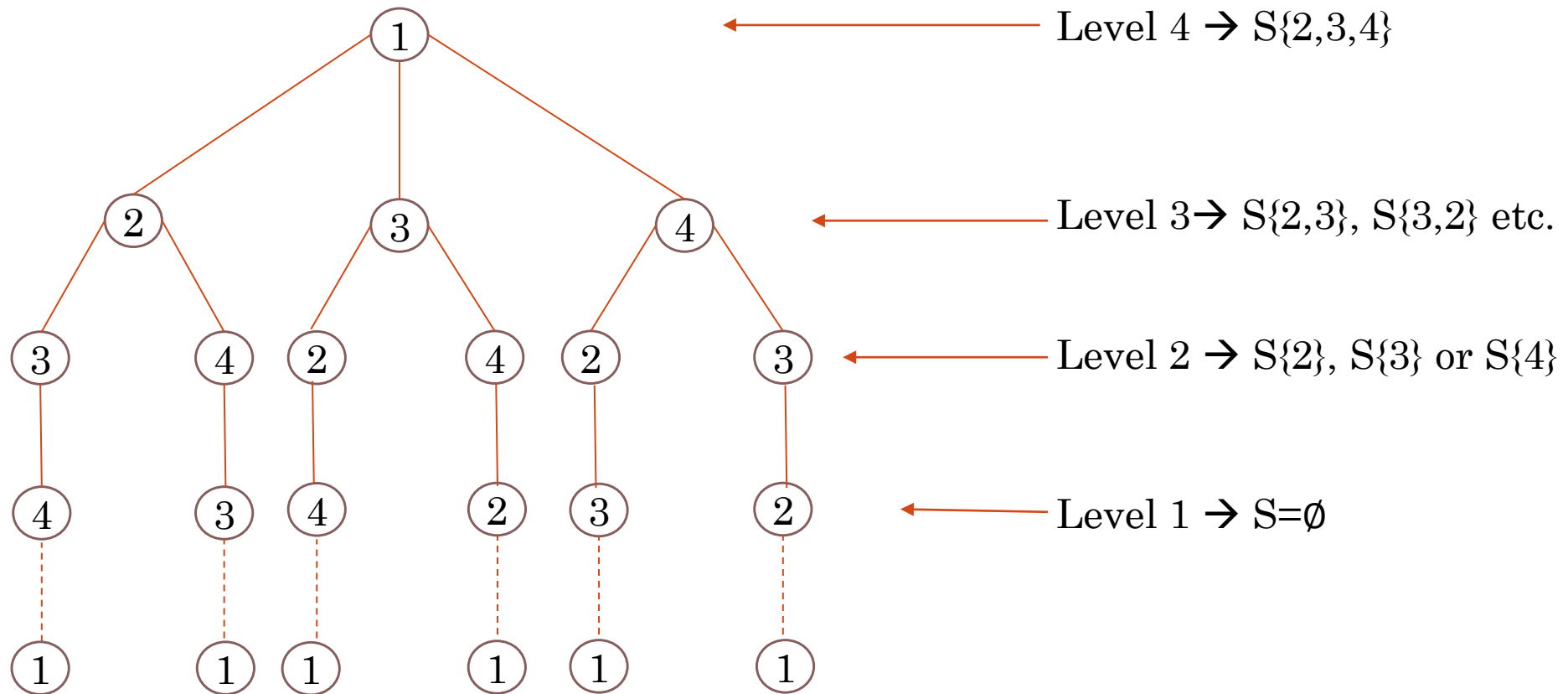
end

The Travelling Salesman Problem

- Example:

Directed Graph

	1	2	3	4
1	0	5	10	15
2	6	0	8	12
3	8	10	0	14
4	7	9	20	0

Cost Matrix

The Travelling Salesman Problem

Level 4 → S{2,3,4}

Level 3→ S{2,3}, S{3,2} etc.

Level 2 → S{2}, S{3} or S{4}

Level 1 → S=∅

The Travelling Salesman Problem

Solution: Recursive Formula:

$$SP(i, S) = \min_{k \in S}[d(i, k) + SP(k, S - \{k\})]$$

Level 4:

Here, i = 1; S = {2,3,4} and k∈ {2,3,4}

$$SP(1,\{2,3,4\}) = \min[d(1,2) + SP(2, \{3,4\}),$$
$$d(1,3) + SP(3, \{2,4\}),$$
$$d(1,4) + SP(4, \{2,3\})]$$

Terms SP(2, {3,4}), SP(3, {2,4}), SP(4, {2,3}) are unknow for now

Level 3:

for i = 2 S = {3,4} and k∈ {3,4}

$$SP(2, \{3,4\}) = \min[d(2,3) + SP(3, \{4\}),$$
$$d(2,4) + SP(4, \{3\})]$$

for i = 3 S = {2,4} and k∈ {2,4}

$$SP(3, \{2,4\}) = \min[d(3,2) + SP(2, \{4\}),$$
$$d(3,4) + SP(4, \{2\})]$$

for i = 4 S = {2,4} and k∈ {2,3}

$$SP(4, \{2,3\}) = \min[d(4,2) + SP(2, \{3\}),$$
$$d(4,3) + SP(3, \{2\})]$$

Terms SP(3, {4}), SP(4, {3}), SP(2, {4}), SP(4, {2}), SP(2,{3}) and SP(3,{2}) are unknow for now

The Travelling Salesman Problem

Solution: Recursive Formula:

$$SP(i, S) = \min_{k \in S}[d(i, k) + SP(k, S - \{k\})]$$

Level 2:

$SP(2,\{3\}) = \min[d(2,3) + SP(3, \emptyset)]$

$SP(2,\{4\}) = \min[d(2,4) + SP(4, \emptyset)]$

$SP(3,\{2\}) = \min[d(3,2) + SP(2, \emptyset)]$

$SP(3,\{4\}) = \min[d(3,4) + SP(4, \emptyset)]$

$SP(4,\{2\}) = \min[d(4,2) + SP(2, \emptyset)]$

$SP(4,\{3\}) = \min[d(4,3) + SP(3, \emptyset)]$

Terms $SP(2, \emptyset), SP(3, \emptyset), SP(4, \emptyset),$ are unknow for now

Level 1:

$SP(2, \emptyset) = d(2,1) = 6$

$SP(3, \emptyset) = d(3,1) = 8$

$SP(4, \emptyset) = d(4,1) = 7$

Substitute above values in previous levels

The Travelling Salesman Problem

Level 1:

$$SP(2, \emptyset) = d(2,1) = 6$$
$$SP(3, \emptyset) = d(3,1) = 8$$
$$SP(4, \emptyset) = d(4,1) = 7$$

Substitute above values in previous levels

Level 2:

$$SP(2,\{3\}) = min[d(2,3) + SP(3, \emptyset)] = 8+8 = 16$$
$$SP(2,\{4\}) = min[d(2,4) + SP(4, \emptyset)] = 12+7 = 19$$
$$SP(3,\{2\}) = min[d(3,2) + SP(2, \emptyset)] = 10+6 = 16$$
$$SP(3,\{4\}) = min[d(3,4) + SP(4, \emptyset)] = 14+7 = 21$$
$$SP(4,\{2\}) = min[d(4,2) + SP(2, \emptyset)] = 9+6 = 15$$
$$SP(4,\{3\}) = min[d(4,3) + SP(3, \emptyset)] = 20+8 = 28$$

Level 3:

$$SP(2, \{3,4\}) = min[d(2,3) + SP(3, \{4\}),$$
$$d(2,4) + SP(4, \{3\})]$$
$$= min[(8+21),(12+28)] = 29$$
$$SP(3, \{2,4\}) = min[d(3,2) + SP(2, \{4\}),$$
$$d(3,4) + SP(4, \{2\})]$$
$$= min[(10+19),(14+15)] = 29$$
$$SP(4, \{2,3\}) = min[d(4,2) + SP(2, \{3\}),$$
$$d(4,3) + SP(3, \{2\})]$$
$$= min[(9+16), (20+16)] = 25$$

Level 4:

$$SP(1,\{2,3,4\}) = min[d(1,2) + SP(2, \{3,4\}),$$
$$d(1,3) + SP(3, \{2,4\}),$$
$$d(1,4) + SP(4, \{2,3\})]$$

$$= min[(5+29), (10+29), (15+25)] = 34$$

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$$

The Travelling Salesman Problem

Time Complexity:

$$SP(i, S) = \min_{k \in S}[d(i, k) + SP(k, S - \{k\})] \dots \dots \dots ①$$

- Let M be number of **SP(i,S)**'s that have to be computed (i.e. no. of subproblems) before we can compute **SP(1, V-{1})**.

- For each value of |S| there are n-1 choices for I .

- No. of distinct set of S of size k not including 1 and i is $\binom{n-2}{k}$.

- Hence , $M = \sum_{k=0}^{n-2}(n-1)\binom{n}{k} = (n-1)2^{n-2}$

The Travelling Salesman Problem

Time Complexity:

- The computation of SP(i, S) with |S|=k requires (k-1) comparisons when solving equation ① .

- Time T required to find an optimal tour

$$T = \sum_{k=1}^{n-2}(k-1)(n-1)\binom{n-2}{k}$$

$$= (n-1)\sum_{k=1}^{n-2}(k-1)\binom{n-2}{k}$$

But,

$$\sum_{k=1}^{n-2}(k-1)\binom{n-2}{k} = \sum_{k=1}^{n-2}\left(k\binom{n-2}{k}\right) - \sum_{k=1}^{n-2}\binom{n-2}{k}$$

$$= \sum_{k=1}^{n-2}(n-2)\boxed{\binom{n-3}{k-1}} - \sum_{k=1}^{n-2}\binom{n-2}{k}$$

$$\because \text{ for integer } k, \qquad k\binom{r}{k} = r.\binom{r-1}{k-1}$$

The Travelling Salesman Problem

Time Complexity:

$$= (n-2) \sum_{k=1}^{n-2} \binom{n-3}{k-1} - (2^{n-2}-1)$$

$$= (n-2)(2^{n-3}) - (2^{n-2}-1)$$

$$since \sum_{k=0}^{k=n} \binom{n}{k} = 2^n$$

Hence, $\text{T} = (n-1)[(n-2)(2^{n-3}) - (2^{n-2}-1)]$

$$= (n-1)[(n-2)(2^n)/8 - (2^n/4 - 1)]$$

$$= (n-1)(n-2)(2^n)/8 - (n-1)(2^n/4 + (n-1)]$$

$$= O(n^2 2^n)$$

Matrix Chain Multiplication

- It is an optimization problem that can be solved using dynamic programming.

- Given a problem, we want to find most efficient way to multiply the given matrices together.

- Problem is not to perform multiplications, but to decide in which order to perform the multiplications.

- Suppose we want to multiply four matrices $A \times B \times C \times D$ of dimensions 50×20, 20×1, 1×10, 10×100 respectively.

- This will involve iteratively multiplying two matrices at a time.

- Matrix multiplication is not commutative i.e. $A \times B \neq B \times A$, but it is associative i.e. $(A \times B) \times C = A \times (B \times C)$.

- So, $A \times B \times C \times D$ can be done in different ways depending upon how parenthesis is done.

Matrix Chain Multiplication

- *A is having dimensions (50x20)*

- *B is having dimension (20x1)*

- *C is having dimension (1x10)*

- *D is having dimension (10x100)*

$$A \times B \times C \times D = A \times (B \times C) \times D = \big(A \times (B \times C)\big) \times D = ((A \times (B \times C)) \times D)$$

Generalised Recursive Formula:

$$C\left[i, j\right] = \min_{i \le k < j} \left\{ C\left[i, k\right] + C\left[k + 1, j\right] + d_{i-1} * d_k * d_j \right\}$$

Matrix Chain Multiplication

Example:

Find optimal multiplication of the chain of 4 matrices

A1, A2, A3, A4 with dimension 3x2, 2x4, 4x2 and 2x5

respectively.

So, here $P[d_0, d_1, d_2, d_3, d_4] = P[3, 2, 4, 2, 5]$

Matrix Chain Multiplication

Algorithm: To find value of an optimal solution.

```
Matrix-Chain(p, n)
{   for (i = 1 to n) m[i, i] = 0;
    for (l = 2 to n)
    {
        for (i = 1 to n − l + 1)
        {
            j = i + l − 1;
            m[i, j] = ∞;
            for (k = i to j − 1)
            {
                q = m[i, k] + m[k + 1, j] + p[i − 1] * p[k] * p[j];
                if (q < m[i, j])
                {
                    m[i, j] = q;
                    s[i, j] = k;
                }
            }
        }
    }
    return m and s; (Optimum in m[1, n])
}
```

Complexity: The loops are nested three deep.

Each loop index takes on $\leq n$ values.

Hence the time complexity is $O(n^3)$. Space complexity $\Theta(n^2)$.

Matrix Chain Multiplication

Algorithm: Construct an optimal solution

```
Mult(A, s, i, j)
{
    if (i < j)
    {
        X = Mult(A, s, i, s[i, j]);
                X is now A_i ⋯ A_k, where k is s[i, j]
        Y = Mult(A, s, s[i, j] + 1, j);
                Y is now A_{k+1} ⋯ A_j
        return X * Y;   multiply matrices X and Y
    }
    else return A[i];
}
```
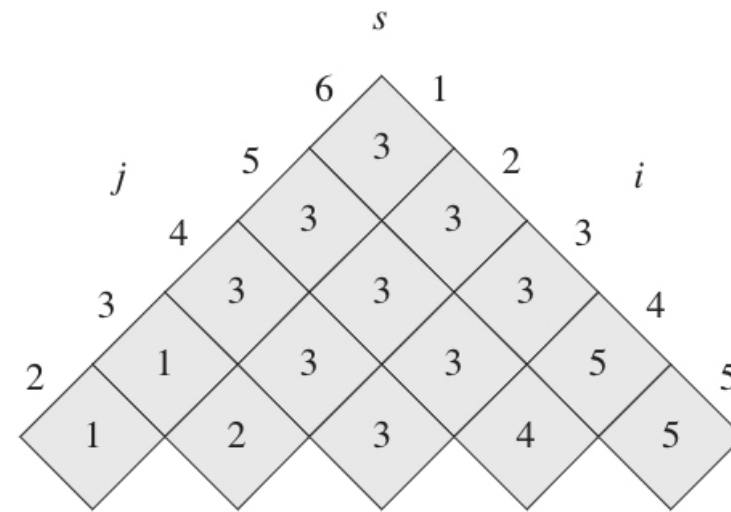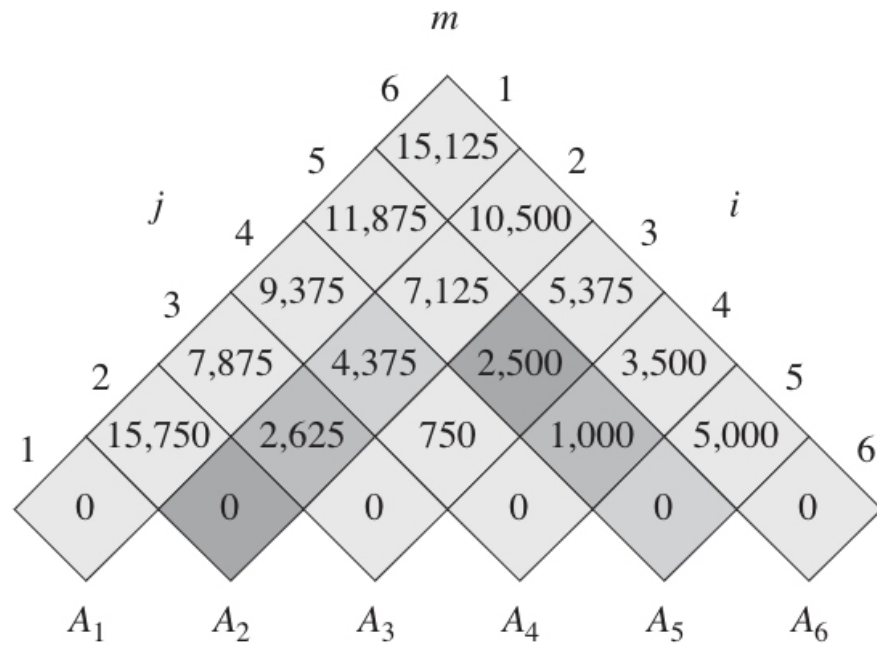
To compute $A_1 A_2 \cdots A_n$, call Mult$(A, s, 1, n)$.

Matrix Chain Multiplication

Example 2:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

Longest Common Subsequence(LCS)

- A Subsequence of a sequence is the same sequence with some elements (possibly none) left out.

- Given a sequence $X = (x_1, x_2, x_3..., x_m)$, the subsequence of X is $(x_{i_1}, x_{i_2}, x_{i_3} ..., x_{i_k})$ where $i_1 < i_2 < i_3 < ... < i_k$.

- For example, given the sequence X=(B C D B A C G H),

 (B D A C) and (C A G H) are some of the subsequence of X

Longest Common Subsequence(LCS)

- Given two sequences X and Y, a sequence Z is a common subsequence of X and Y

- For Example, X= (A D C B E J M) and Y = (B A C D E B M), then the sequence (D B M) having length 3 is common subsequence of both X and Y.

- However, (D B M) is not longest common subsequence of X and Y.

- Subsequence (A D B M) of length 4 common to both X and Y.

- Longest Common subsequence problem is stated below:

"Given two sequence $X=(x_1, x_2, x_3..., x_m)$ and $Y=(y_1, y_2, y_3..., y_n)$, find a maximum length common subsequence of X and Y."

Longest Common Subsequence(LCS)

Recursive Formula:

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

Longest Common Subsequence(LCS)

Example: 1

Find the Longest Common Subsequence of the sequences

X = (A D C B E J M) and

Y = (B A C D E B M)

		0	B 1	A 2	C 3	D 4	E 5	B 6	M 7
	0	0	0	0	0	0	0	0	0
A	1	0	0	1	1	1	1	1	1
D	2	0	0	1	1	2	2	2	2
C	3	0	0	1	2	2	2	2	2
B	4	0	1	1	2	2	2	3	3
E	5	0	1	1	2	2	3	3	3
J	6	0	1	1	2	2	3	3	3
M	7	0	1	1	2	2	3	3	4

Longest Common Subsequence(LCS)

Algorithm:

```
lcs(char X[],char Y[],int c[][n],int m,int n)
{/* the two sequences are input in arrays X and Y. The length of the
longest common sequence is obtained in c[m][n] */
1.    for(i=0;i<=m;i++)
2.       c[i][0]=0;
3.    for(j=0;j<=n;j++)
4.       c[0][j]=0;
5.    for(i=1;i<=m;i++)
6.       for(j=1;j<=n;j++)
7.          if(X[i] != Y[j])
8.             c[i][j]=max (c[i-1][j],c[i][j-1]);
9.          else
10.            c[i][j]=c[i-1][j-1]+1;
11.   return c[m][n];
}/* End of lcs(...) */
```

Longest Common Subsequence(LCS)

Algorithm:

```
lcs_print(char X[],int c[][n],int m,int n)
{
1.    if(c[m][n]==0)
2.       return;
3.    if(c[m][n]==c[m-1][n])
4.       lcs_print(X,c,m-1,n);
5.    else
6.    if(c[m][n]==c[m][n-1])
7.       lcs_print(X,c,m,n-1);
8.    else
9.    {
10.      lcs_print(X,c,m-1,n-1);
11.      printf("%c\t",X[m]);
12.   }/* else of if(c[m][n]==c[m][n-1]) */
}/* End of lcs_print(...) */
```