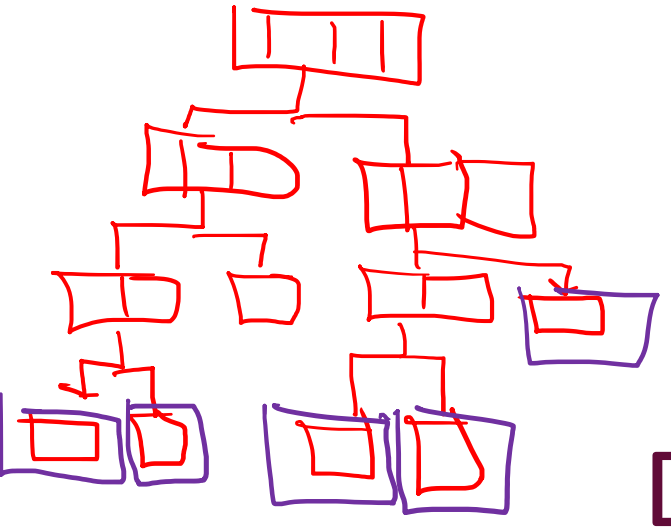
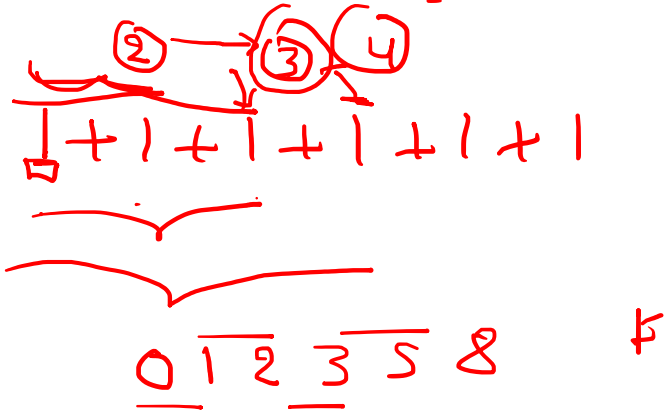
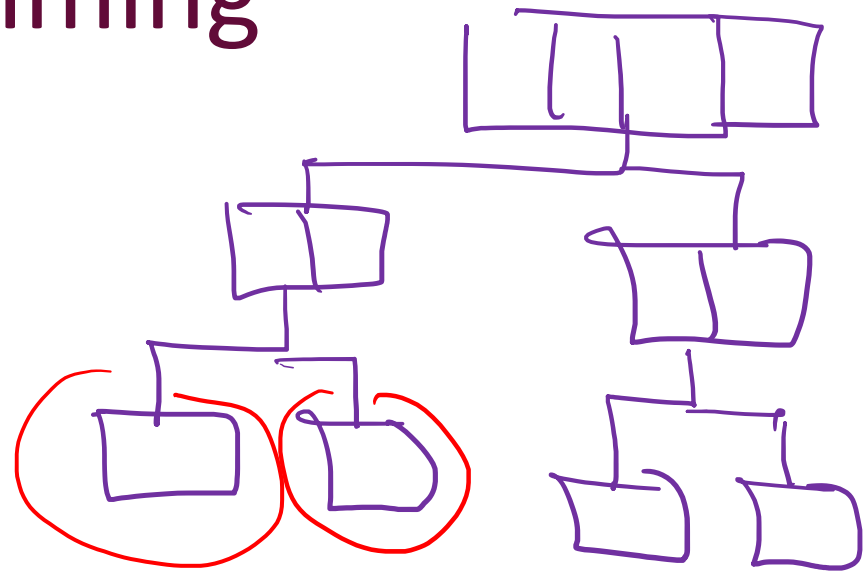
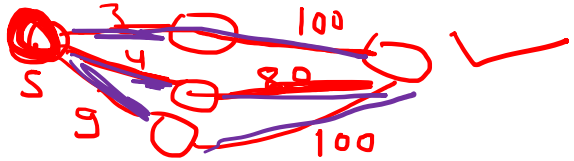


Divide
& Conq



Dynamic Programming





Dynamic Programming

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again.

The main use of dynamic programming is to solve optimization problems.

The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

How does the dynamic programming approach work?

The following are the steps that the dynamic programming follows:

- It breaks down the complex problem into simpler subproblems.
- It finds the optimal solution to these sub-problems.
- It stores the results of subproblems (memorization). The process of storing the results of subproblems is known as memorization.
- It reuses them so that same sub-problem is calculated more than once.
- Finally, calculate the result of the complex problem.

Approaches of dynamic programming

There are two approaches to dynamic programming:

- Top-down approach
- Bottom-up approach

Top-down approach

The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.

Advantages

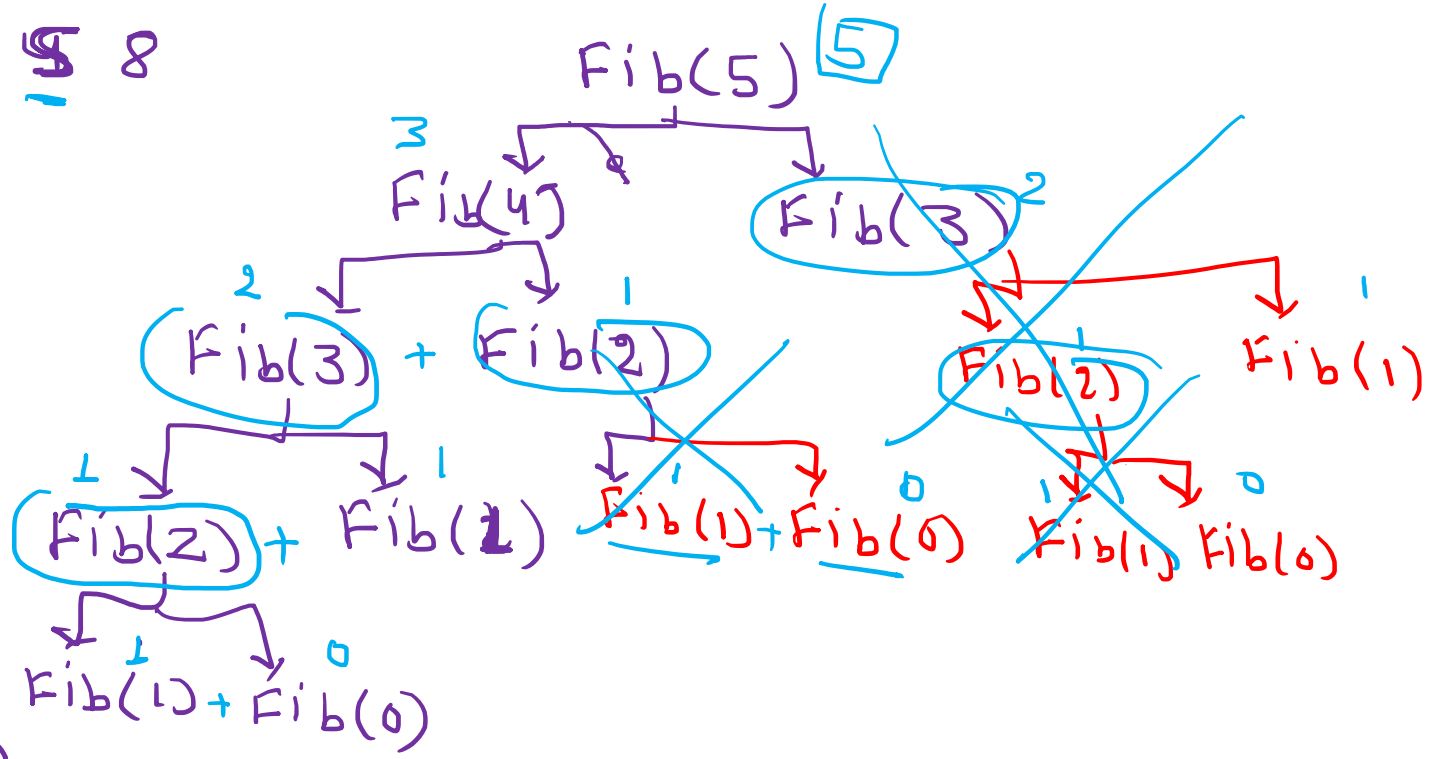
- It is very easy to understand and implement.
- It solves the subproblems only when it is required.
- It is easy to debug.

Disadvantages

- It uses the recursion technique that occupies more memory in the call stack.

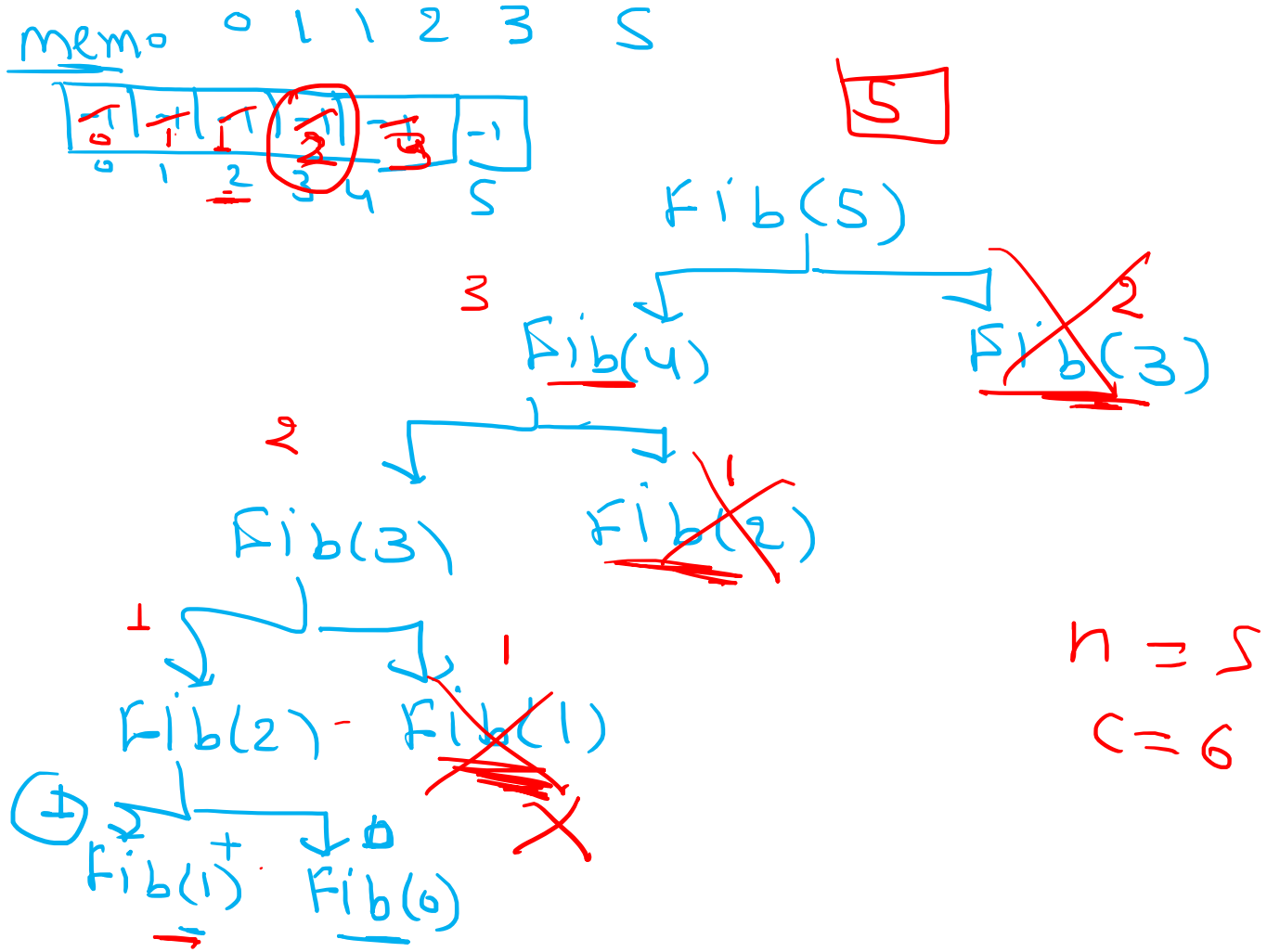
```
int fib(int n)
{
   if(n<0)
   error;
 if(n==0)
 return 0;
if(n==1)
return 1;
sum = fib(n-1) + fib(n-2);
}
```

0 1 1 2 3 5 8



the time complexity increases exponentially, and it becomes 2^n.

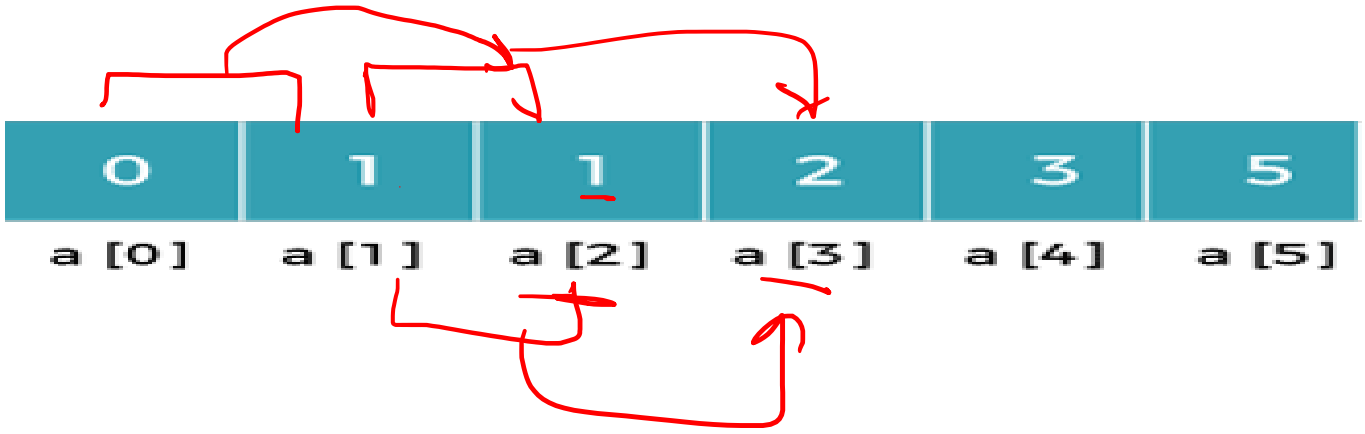
```
static int count = 0;
int fib(int n)
{
if(memo[n]!= NULL)
return memo[n];
count++;
  if(n<0)
  error;
 if(n==0)
 return 0;
 if(n==1)
return 1;
sum = fib(n-1) + fib(n-2);
memo[n] = sum;
}
```



If we use the dynamic programming approach, then the time complexity would be O(n).

Bottom-Up approach

The bottom-up approach is also one of the techniques which can be used to implement the dynamic programming. It uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion.




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

Longest Common Subsequence

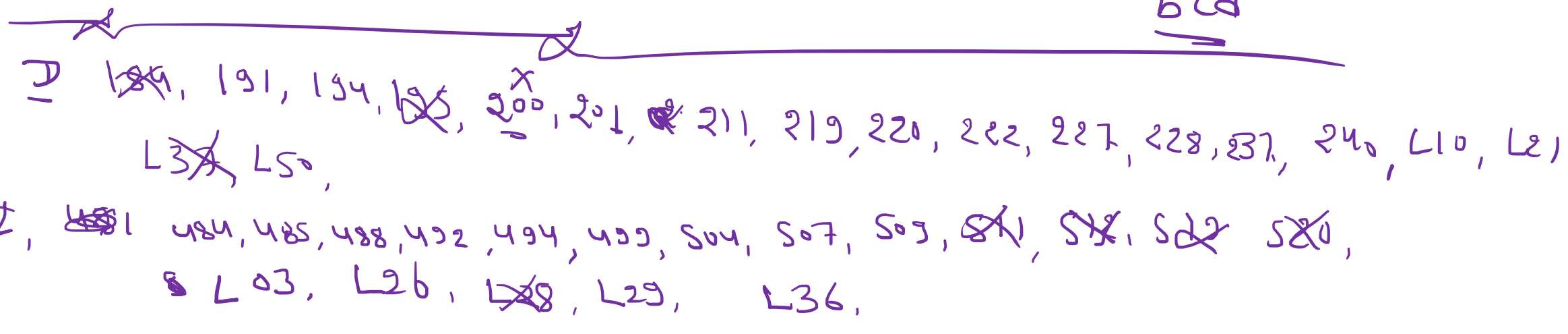
longest means that the subsequence should be the biggest one. The common means that some of the characters are common between the two strings. A subsequence is nothing but a series of elements that occur in the same order but are not necessarily contiguous

W_1 = abcd :- { a, b, c, d, ab, ac, ad, bc, cd, bd, abc, abd, bcd }

W_2 = bcd = { b, c, d, bd, cd, bcd }

LCS = 3

bcd



$$LCS[i][j] = \begin{cases} 0 & \text{if } X_i = 0 \text{ or } Y_j = 0 \\ LCS[i-1][j-1] + 1 & \text{if } X_i = Y_j \\ \max(LCS[i-1, j], LCS[I, j-1]) & \text{if } X_i \neq Y_j \end{cases}$$

1. $LCS(0,j) = 0$ for $\forall j$
2. $LCS(i,0) = 0$ for $\forall i$
3. $LCS(i,j) = 1 + LCS(i-1, j-1)$, $X_i = Y_j$
4. $LCS(i,j) = \max[LCS(i-1,j), LCS(i,j-1)]$

$X = a\ b\ a\ a\ b\ a$ ($n = 6$)

$Y = b\ a\ b\ b\ a\ b$ ($m = 6$)

$LCS(2,2) = 0$

abab

abab

baba

i	X	0	1 b	2 a	3 b	4 b	5 a	6 b
0		0	0	0	0	0	0	0
1	a	0	0	1	1	1	1	1
2	b	0	1	1	2	2	2	2
3	a	0	1	2	2	2	3	3
4	a	0	1	2	2	2	3	3
5	b	0	1	2	3	3	3	4
6	a	0	1	2	3	3	4	4

Step 1

$L(2,3) =$

$L(2,4) =$

$L(2,5) =$

$L(2,6) =$

$L(2,7) =$

Step 2

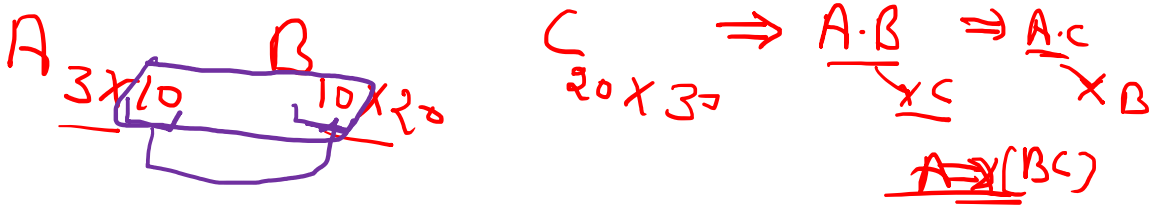
$L(3,2)$

$L(3,3)$

$\vdots$

$L(3,7)$

Matrix Chain Multiplication Problem



- Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that to find the most efficient way to multiply a given sequence of matrices.

In what order, n matrices A_1, A_2, A_3, …. A_n should be multiplied so that it would take a minimum number of computations to derive the result.

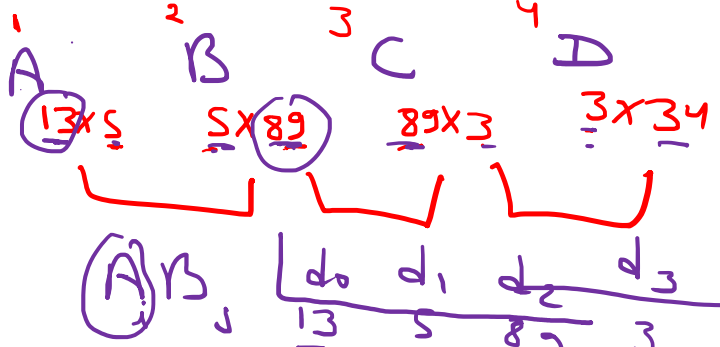
The optimal substructure is defined as,

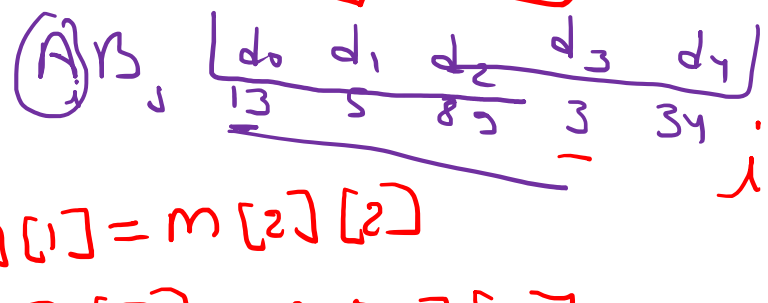
$$m[i, j] = \begin{cases} 0 & , \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + d_{i-1} \times d_k \times d_j\} & , \text{if } i < j \end{cases}$$

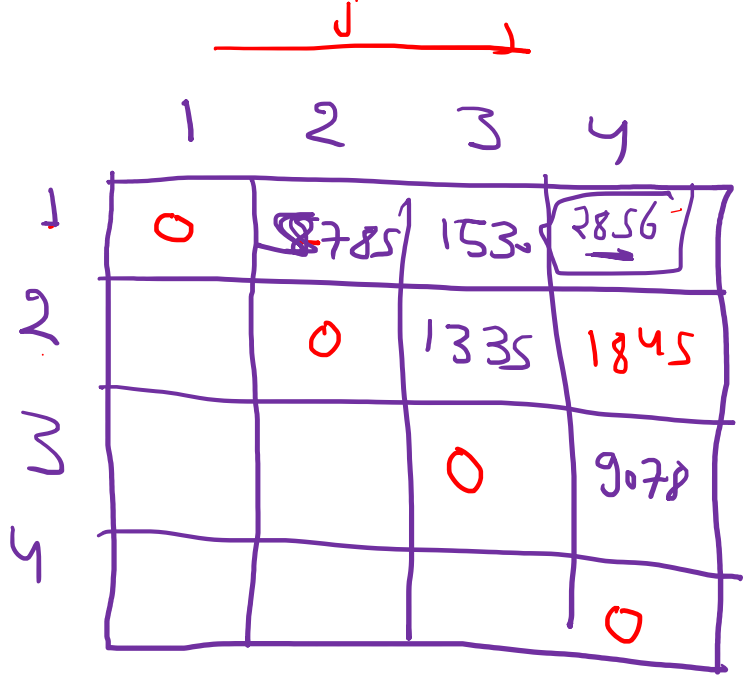
Where d = $\{d_0, d_1, d_2, \ldots, d_n\}$ is the vector of matrix dimensions.

m[i, j] = Least number of multiplications required to multiply matrix sequence $A_i \ldots A_j$.

Ex:







Step 1

$m[1][1] = m[2][2]$

$m[3][3] = m[4][4] = 0$

Step 2

$m[1][2] = \underset{13\times5}{A} \cdot \underset{5\times89}{B} = 13 \times 5 \times 89 = 785$ ($d_0\ d_1\ d_2$)

$m[2][3] = \underset{5\times89}{B} \cdot \underset{89\times3}{C} = 5 \times 89 \times 3 = 1335$ ($d_1\ d_2\ d_3$)

$m[3][4] = \underset{89\times3}{C} \cdot \underset{3\times34}{D} = 89 \times 3 \times 34 = 9078$ ($d_2\ d_3\ d_4$)

Step 3

Step3   2 part

ABC ⇒ (A·B)·C

$m[1][2]$ 13×89   89×3

$i=1,\ k=2,\ j=3$

$m[1][3] = \{m[1,2] + m[3,3] + d_0 d_2 d_3$

8785 + 0 + 13 * 89 × 3

= ~~12256~~ 9256

A·(B·C) $i=1\ k=1\ j=3$

13×5   5×3

$= m[1,1] + m[2,3] + d_0 d_1 d_3$

= 0 + 1335 + 13×5×3

= 1530

min

BCD

2 3 4 $i=2\ k=3$

(B·C) × D $j=4$

5×3   3×34

$m[2][4] = m[2][3] + m[4][4] + d_1 d_3 d_4$

= 1335 + 0 + 5×3×34

= 1845

min

2 3 4 $i=2\ k=3\ j=4$

B·(C·D)

$m[2][2] + m[3][4] + d_1 d_3 d_4$

= 0 + 9078 + 5×3×34

= 9588

Step 4

A B C D

1 2 3 4

$(AB)\cdot(CD)$

$i=1$, $k=2$, $j=4$

$= m[1][2] + m[3,4] + d_0 d_2 d_4$

$= 1587$

5785

$i=1$, $k=1$, $j=4$

$A \cdot (BCD)$

$m[1][1] + m[2][4] + d_0 d_1 d_4$

4055

$i=1$, $k=3$, $j=4$

$(ABC)\cdot D$

$= m[1][3] + m[4][4] + d_0 d_3 d_4$

2856

$m[1][4] = 2856$

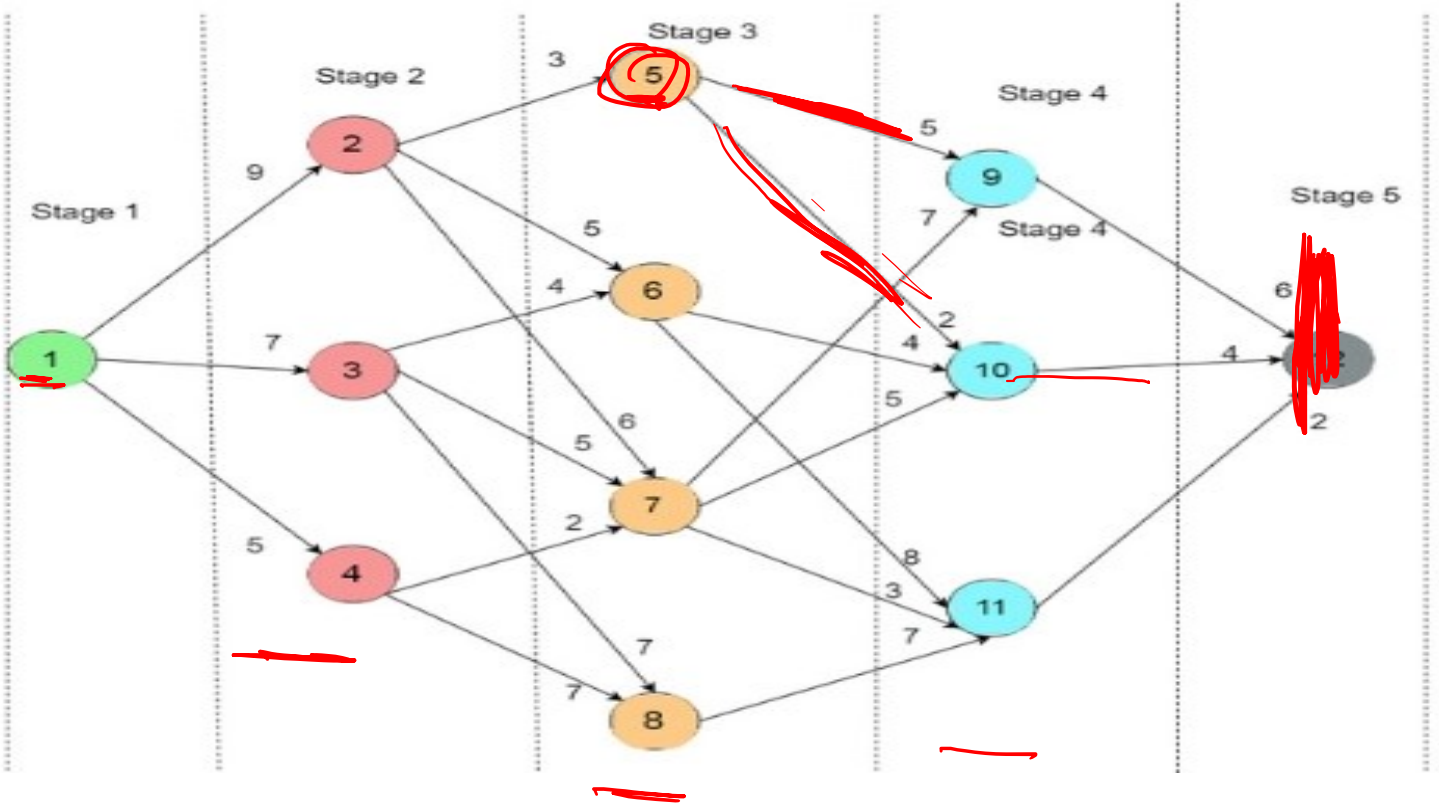
D : 185, 198, 203, 205, 206, 209, 216, ~~217~~, 222, 223, 224, 234, L10,

I : ~~487~~, ~~496~~, 491, 504, ~~814~~, ~~820~~, ~~585~~, L29, ~~436~~, L40,

Multistage Graph

A **Multistage graph** is a directed, weighted graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage only.

The main aim of this graph is to find the *minimum cost path between starting and ending vertex*.



Approaches to Multistage Graph

- Forward Approach
- Backward Approach

Forward Approach (here start from destination vertex)

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

i denotes stage

j denotes vertex id

cost(i,j) – Cost of node j in stage i to destination

c(j,l) – cost of edge between node j,l)

Step 1

Forward (desti

stage = {v_1, v_2, v_3, v_4, v_5}

c(i,j)

Cost(5,12)=0;

Step 2

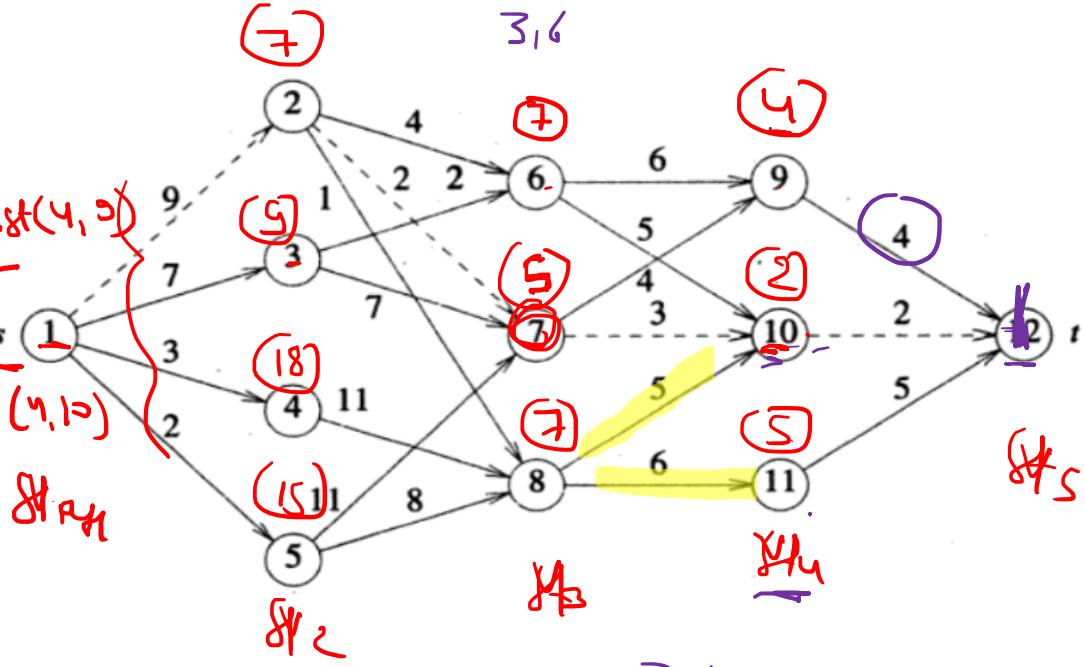
- Cost(4,9)=4+ Cost(5,12)=4;
- Cost(4,10)=2+ Cost(5,12)=2;
- Cost(4,11)=5+ Cost(5,12)=5;

Step 3

- 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

i=3, j=6

cost(i,j) = min{ cost(6,9)+cost(4,9), cost(6,10)+cost(4,10) }



3,6

3,7

3,8

Shortest path

path

V	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	2/3	7	6	8	8	10	10	10	12	12	12	12

Step 4

Cost(2,2)=min{4+cost(3,6),2+cost(3,7),1+cost(3,8)}=min{11,7,8}=7

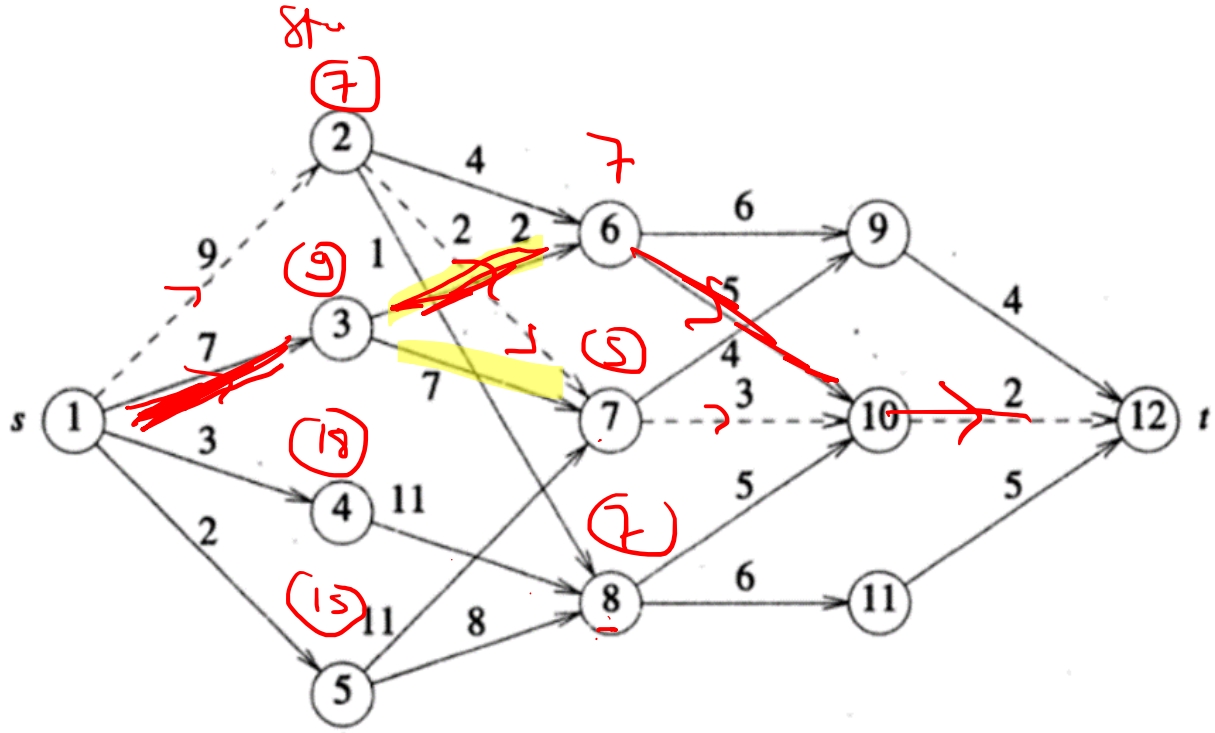
- Cost(2,3)=min{2+cost(3,6),7+cost(3,7)}=min{9,12}=9
- Cost(2,4)=11+cost(3,8)=18
- Cost(2,5)=min{11+cost(3,7), 8+cost(3,8)}=min{16,15}=15

steps

- 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



Backward Approach (here start from source vertex)

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

Step 1

cost(1,1)=0;

Step 2

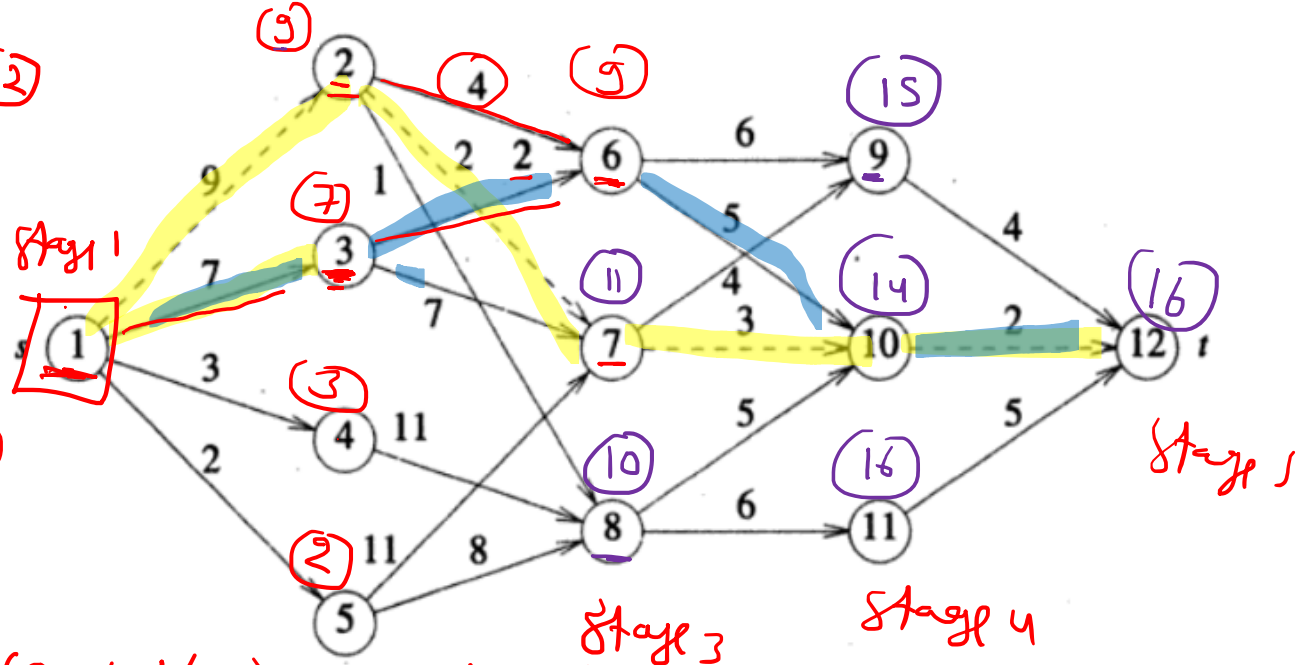
- cost(2,2)=9+cost(1,1)=9;
- cost(2,3)=7+cost(1,1)=7;
- cost(2,4)=3+cost(1,1)=3;
- cost(2,5)=2+cost(1,1)=2;

Step 3

- cost(3,6)=min{4+cost(2,2), 2+cost(2,3)}

  =min{13,9}=9;

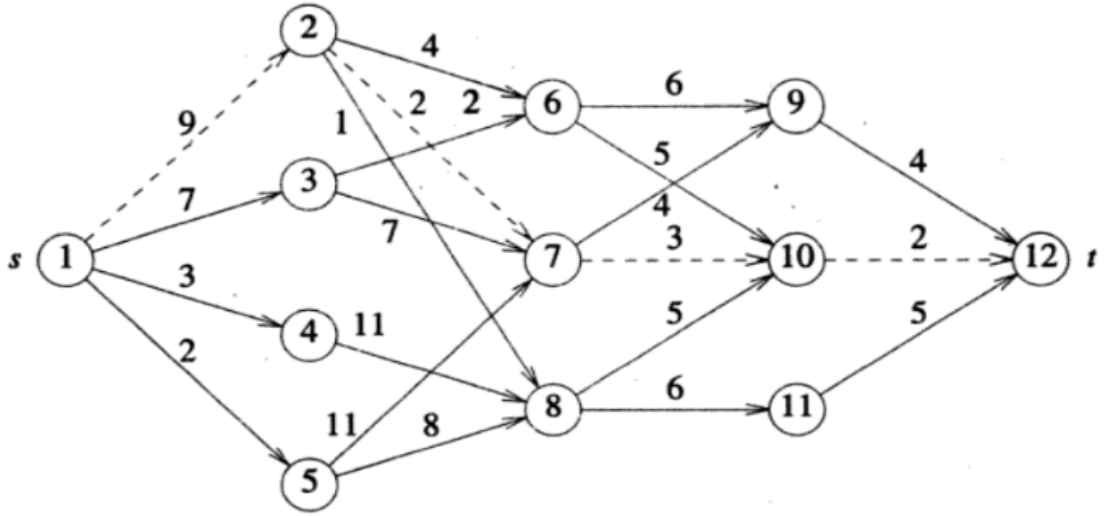
- cost(3,7)=11
- cost(3,8)=10
- cost(4,9)=15
- cost(4,10)=14
- cost(4,11)=16
- cost(5,12)=16

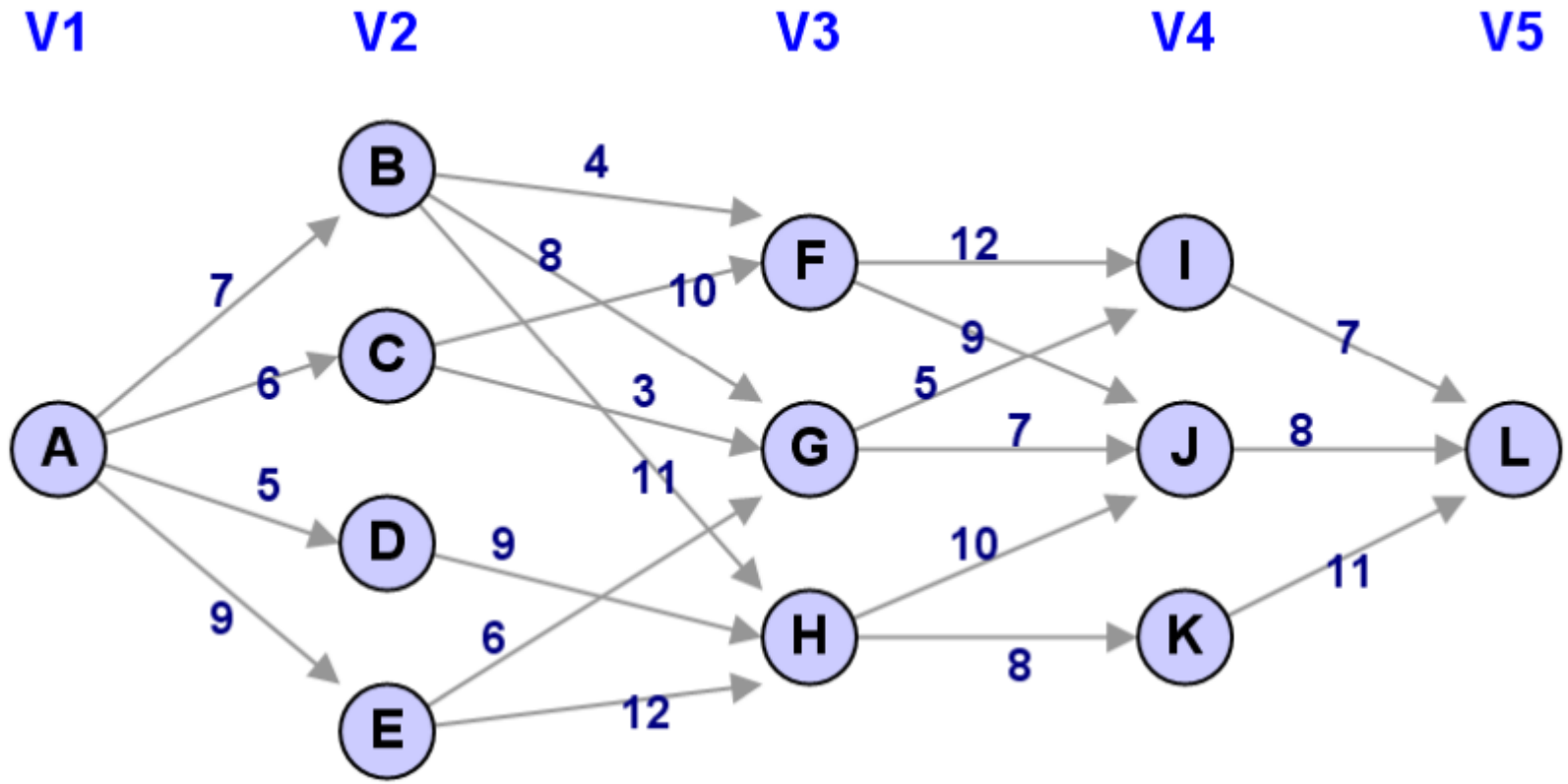


V	1	2	3	4	5	6	7	8	9	10	11	12
Cost	0	9	7	3	2	9	11	10	15	14	16	16
d	1	1	1	1	1	3	2	2|5	6|7	6|7	8	10

D 194, 221, 222, 228, 230,
L13 L21, L50.

T , S04, , S24, L03,




V1
V2
V3
V4
V5
A
B
C
D
E
F
G
H
I
J
K
L
7
6
5
9
4
8
10
3
11
9
6
12
12
9
5
7
10
8
7
8
11

Traveling-salesman Problem using dynamic programming

Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

There is a non-negative cost c (i, j) to travel from the city i to city j.

The goal is to find a tour of minimum cost.

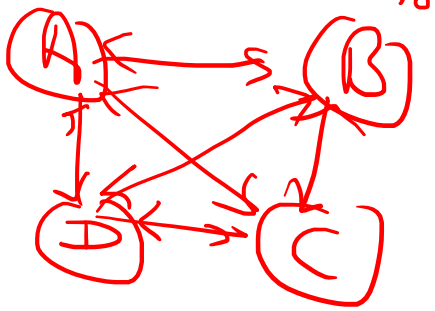
$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}$$

i start vertex
S : Set of cities exactly once's visit

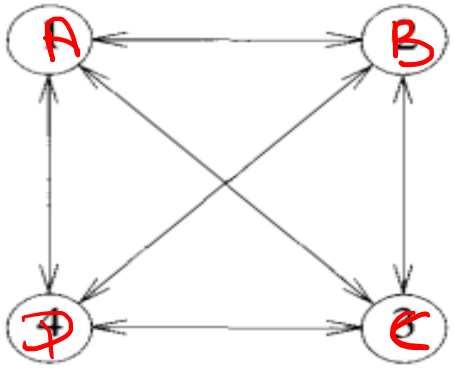
A → B → C → D → A

exactly once's

Shortest path



Example 5.26 Consider the directed graph of Figure 5.21(a). The edge lengths are given by matrix c of Figure 5.21(b).



(a)

	A	B	C	D
A	0	10	15	20
B	5	0	9	10
C	6	13	0	12
D	8	8	9	0

(b)

$g(i, S)$

$g(A, \{B, C, D\})$ = 35

$g(B, \{C, D\})$ = 25 (10)

$g(C, \{B, D\})$ = 25 (15)

$g(D, \{B, C\})$ = 23 (20)

$g(C, \{D\})$ = 20 (12)

$g(D, \{C\})$ = 15 (9)

$g(B, \{D\})$ = 18 (10)

$g(D, \{B\})$ = 13 (8)

$g(B, \{C\})$ = 15 (8)

$g(C, \{B\})$ = 18 (9)

$g(D, \phi)$ = 8 (A)

$g(C, \phi)$ = 6 (A)

$g(B, \phi)$ = 8 (A)

$g(D, \phi)$ = 8 (A)

$g(B, \phi)$ = 5 (A)

$g(C, \phi)$ = 6 (A)

$g(B, \phi)$ = 5 (A)

$A \to B \to D \to C \to A$ or $A \to B \to D \to C$

$$g(i, S) = \min_{j \in S} \{ c_{ij} + g(j, S - j) \}$$

$$g(X, \phi) = c_{ni}$$

Step 1

$$g(D, \phi) = c_{DA} = 3$$

$$g(C, \phi) = c_{CA} = 6$$

$$g(B, \phi) = c_{BA} = 5$$

Step 2

$$g(C, \{D\}) = \min_{j \in S} (\text{cost}_{CD} + g(D, \phi))$$

$$12 + 8 = 20$$

$$g(D, \{C\})$$

$$g(B, \{D\})$$

$$g(B, \{C\})$$

$$g(i, S) = \min \{ \text{cost}_{iB} + g(B, \phi) \}$$

Step

Step 3 $g(i, S) = \min_{j} \{ c(i, j) + g(j, S - \{j\}) \}$

Complexity 

(II) 184, 186, ~~191~~, ~~198~~, 194, ~~218~~, 222, 226, 227, 229,

~~232~~ ~~235~~, ~~238~~,

(I) 496, 504, ~~508~~ 519, 522, 524, ~~527~~ ~~528~~, 529,

0/1 Knapsack Problem | Dynamic Programming

Greedy method

① Fractional knapsack problem

Knapsack Problem-

- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.

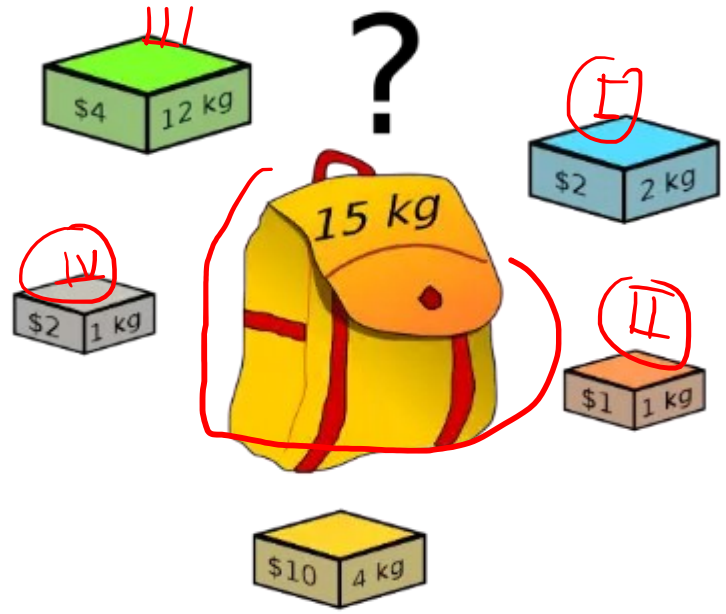
The problem states-

Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.

In 0/1 Knapsack Problem,

- We can not take the fraction of any item.
- We have to either take an item completely or leave it completely.
- It is solved using dynamic programming approach.



Knapsack Problem

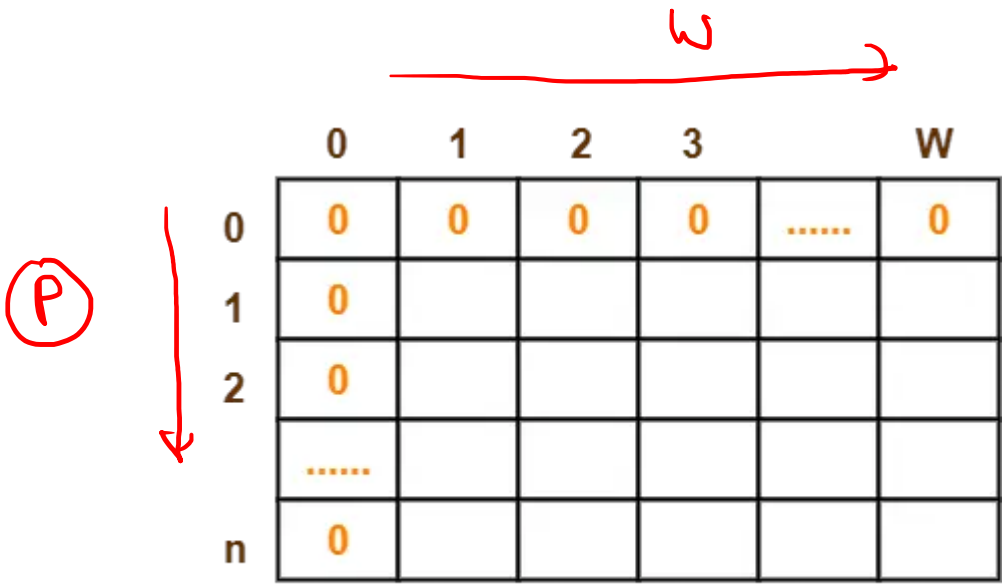
Consider-

- Knapsack weight capacity = w
- Number of items each having some weight and value = n

0/1 knapsack problem is solved using dynamic programming in the following steps-

Step-01:

- Draw a table say 'T' with (n+1) number of rows and (w+1) number of columns.
- Fill all the boxes of 0th row and 0th column with zeroes as shown-



T-Table

Step-02:

Start filling the table row wise top to bottom from left to right.
Use the following formula-

$$T(i, W) = \max \{ T(i-1, W), value_i + T(i-1, W - W(i)) \}$$

Here, T(i ,W) = maximum value of the selected items if we can take items 1 to i and have weight restrictions of W.

- This step leads to completely filling the table.
- Then, value of the last box represents the maximum possible value that can be put into the knapsack.

Step-03:

To identify the items that must be put into the knapsack to obtain that maximum profit,

- Consider the last column of the table.
- Start scanning the entries from bottom to top.
- On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.
- After all the entries are scanned, the marked labels represent the items that must be put into the knapsack.

Time Complexity-

- Each entry of the table requires constant time $\theta(1)$ for its computation.
- It takes $\theta(nw)$ time to fill $(n+1)(w+1)$ table entries.
- It takes $\theta(n)$ time for tracing the solution since tracing process traces the n rows.
- Thus, overall $\theta(nw)$ time is taken to solve 0/1 knapsack problem using dynamic programming.

Problem-

For the given set of items and knapsack capacity = 5 kg, find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach.

- Knapsack capacity (w) = 5 kg
- Number of items (n) = 4

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

item	w	p
4	5	6

item	w	p
3	4	5

I_2 I_1

Step-01:

- Draw a table say 'T' with (n+1) = 4 + 1 = 5 number of rows and (w+1) = 5 + 1 = 6 number of columns.
- Fill all the boxes of 0th row and 0th column with 0.

we know

$$T(i, W) = \max \{ T(i-1, W), value_i + T(i-1, W - W(i)) \}$$

empty

Step 1

$T(1,1) = \max \{ T(0,1), 3 + T(0, 1-2) \}$

$= \max(0, 3 + T(0,-1))$

i w ↳ = 0

$T(1,2) = \max(T(0,2), 3 + T(0, 2-2))$

$= \max(0, 3+0)$

$= 3$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

max

T-Table

Step 2 $i=2,\ w_1=2\ \ w_2=3\ \ Value_2=4$

$T(2,1) = \max(T(1,1),\ 4+T(1,1-3)\}$

$= 0$

$T(2,2) = \max(T(1,2),\ 4+T(1,2-3)\}$

$= 3$

$T(2,3) = \max(T(1,3),\ 4+T(1,3-3)\}$

$= \max(3,\ 4+0\}$

$= 4$

$i \quad w$

$T(2,5) = \max(T(1,5),\ 4+T(1,5-3)\}$

$= \max(3,\ 4+T(1,2)\}$

$= \max(3,\ 4+3]$

$= 7$

I_1	I_2	I_3	I_4	I_5
1	1	0	0	0

)

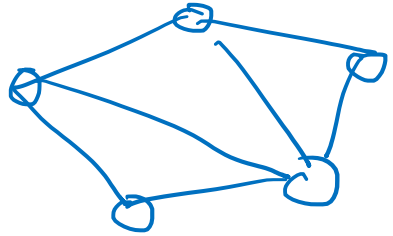
I

~~482~~, 488, 488, ~~489~~, 491, 494, 495, 504, 510, 514, 520

~~525~~ 526, L03, L12, L26, L29,

All Pair Shortest Path Floyd Warshall

All Pair Shortest Path Floyd Warshall Algorithm Example



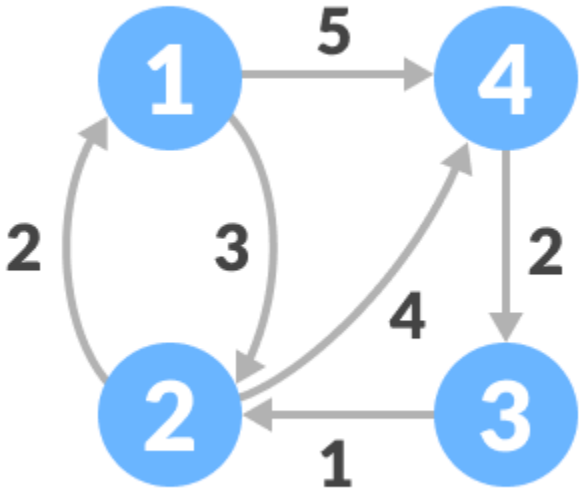
- Floyd Warshall Algorithm is a famous algorithm.
- It is used to solve All Pairs Shortest Path Problem.
- It computes the shortest path between every pair of vertices of the given graph.
- Floyd Warshall Algorithm is an example of dynamic programming approach.
- complexity of Floyd Warshall algorithm is $O(n^3)$.
- Floyd Warshall Algorithm is best suited for dense graphs.

- Let G = <V, E> be a directed graph, where V is a set of vertices and E is a set of edges with nonnegative length. Find the shortest path between each pair of nodes.
- L = Matrix, which gives the length of each edge

L[i, j] = 0, if i == j // Distance of node from itself is zero
L[i, j] = ∞ , if $i \neq j$ and $(i, j) \notin E$
L[i, j] = w (i, j), if $i \neq j$ and $(i, j) \in E$ // w(i, j) is the weight of the edge (i, j)

Problem-

Consider the following directed weighted graph-
Using Floyd Warshall Algorithm, find the shortest path distance between every pair of vertices.



Step-01:

•Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph

Step-02:

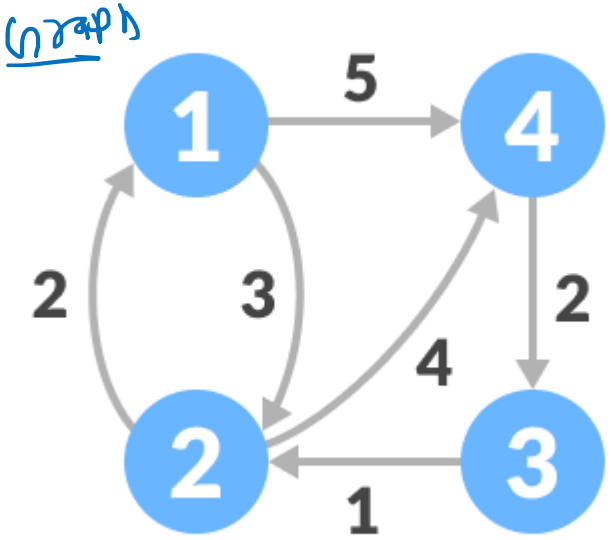
•Write the initial distance matrix.

•It represents the distance between every pair of vertices in the form of given weights.

•For diagonal elements (representing self-loops), distance value = 0.

•For vertices having a direct edge between them, distance value = weight of that edge.

•For vertices having no direct edge between them, distance value = ∞.



$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

$A^0 =$

	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	∞	1	0	∞
4	∞	∞	2	0

$D^1(2,3) = \{ D^0(2,1) + D^0(1,3),\ D^0(2,3) \}$

$2 + \infty = \infty,\quad \infty = \infty$

$D^1(2,4) = \min\{ D(2,1) + D(1,4),\ D(2,4) \}$

$= \{ 2 + 5,\ 4 \} = 4$

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

$= \infty + 3,\ 1 = 1$

$D^1(3,4) = \min(D(3,1) + D(1,4),\ D(3,4) \}$

$\infty + 5,\ \infty = \infty$

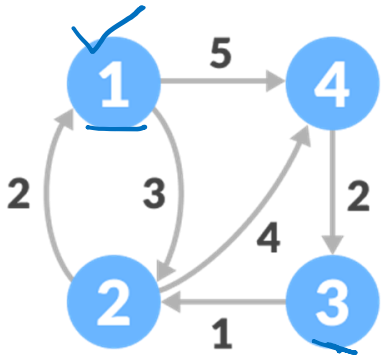
$D^1(4,2) = \{ D(4,1) + D(1,2),\ D(4,2) \}$

$A^1 =$

	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	∞	1	0	∞
4	∞	∞	2	0

→

	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	∞	1	0	∞
4	∞	∞	2	0



$A^1 =$

	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	∞	1	0	∞
4	∞	∞	2	0

$D^2(1,3) = \min\{D^1(1,2)+D^1(2,3), D^1(1,3)\}$
$3+\infty, \infty = \infty$

$D^2(1,4) = \min(D^1(1,2)+D^1(2,4), D^1(1,4)\}$
$3+4, 5 = 5$

$D^2(3,1) = \min(D^1(3,2)+D^1(2,1), D^1(3,1)\}$
$1+2, \infty = 3$

$D^2(3,4) = \min(D^1(3,2)+D^1(2,4), D^1(3,4)\}$
$1+4, \infty = 5$

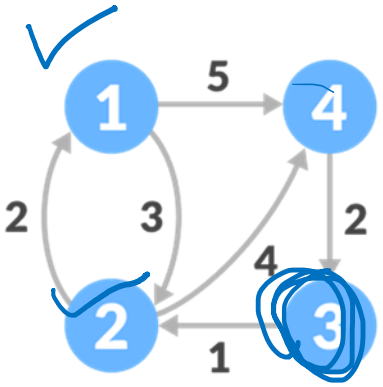
D^1

$A^2 =$

	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	3	1	0	5
4	∞	∞	2	0

→

	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	3	1	0	5
4	∞	∞	2	0



$A^2 =$

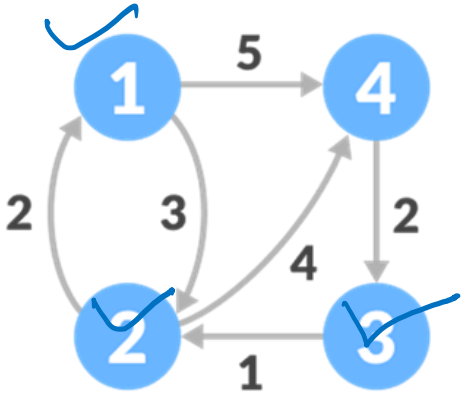
	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	3	1	0	5
4	∞	∞	2	0

$A^3 =$

	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	∞	1	0	5
4	5	3	2	0

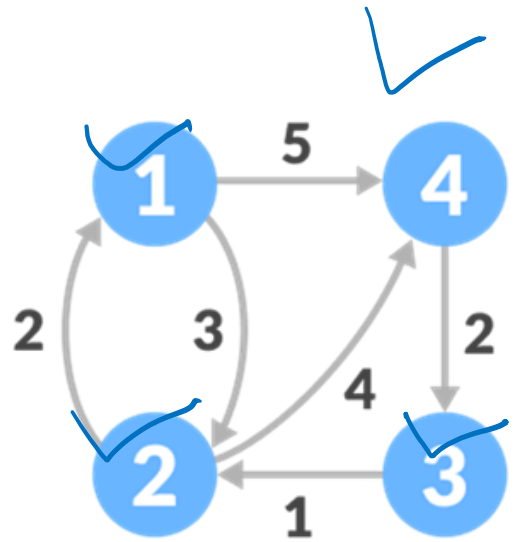
→

	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	3	1	0	5
4	5	3	2	0



$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

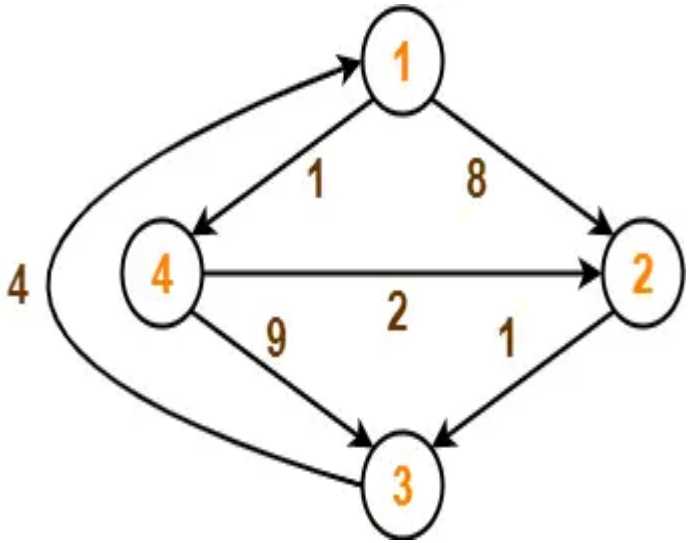
$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

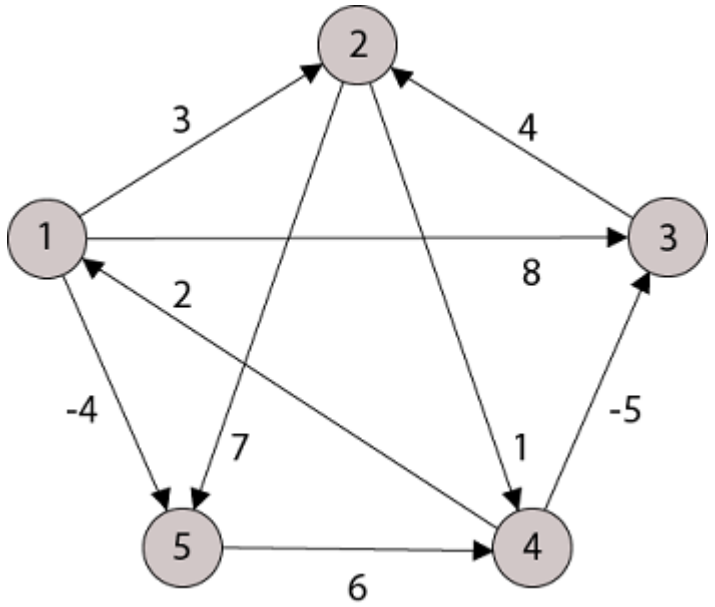


$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

Floyd-Warshall Algorithm

```
FLOYD - WARSHALL (W)
 n ← rows [W].
D0 ← W
for k ← 1 to n
   do for i ← 1 to n
      do for j ← 1 to n
       do dij(k) ← min (dij(k-1),dik(k-1)+dkj(k-1) )
return D(n)
```





답 ~~182~~ ~~185~~, 186, 187, 194, 195, ~~208~~, ~~216~~, 218, 222, ~~223~~, ~~232~~, 236, ~~238~~ ~~238~~, ~~239~~, ~~246~~, L10, L13,

Reliability Design Problem in Dynamic Programming

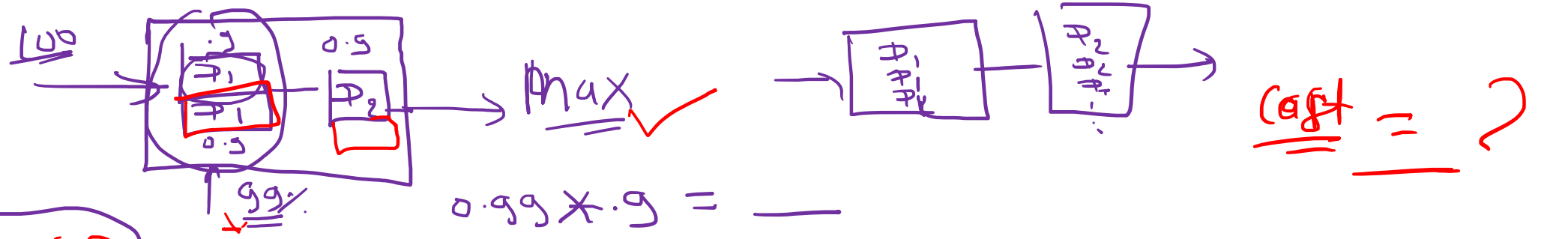
The reliability design problem is the designing of a system composed of several devices connected in series or parallel. Reliability means the probability to get the success of the device.

Let's say, we have to set up a system consisting of $D_1, D_2, D_3, \ldots\ldots\ldots$, and D_n devices, each device has some costs $C_1, C_2, C_3, \ldots\ldots\ldots, C_n$. Each device has a reliability of **0.9** then the entire system has reliability which is equal to the **product** of the reliabilities of all devices i.e., $\pi r_i = (0.9)^4$.

1. $R_{S1} = [1-(1-r_i)^m]$

   Where m is number of copy of devices

2. ***No. of devices at a stage =floor(C +C_i – $\sum_{i=1}^{n} C_i$ / C_i)***



$0.99 \times .9 = \_\_\_$

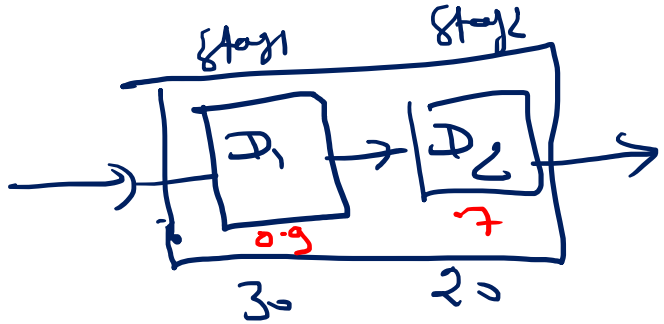
$D_1 = 90$

$C_1 = 30$

$D_2 = 70\%$

$C_2 = 20$

$C = 100$



$0.7 \times 0.9 = .63$

Step 1

$$\text{No. of device at Stage 1} = \left\lfloor \frac{C + C_i - \sum_{i=1}^{n} C_i}{C_i} \right\rfloor = \left\lfloor \frac{100 + 30 - 50}{30} \right\rfloor = \frac{80}{30} = 2$$

$$\text{No. of device of Stage 2} = \left\lfloor \frac{100 + 20 - 50}{20} \right\rfloor = \frac{70}{20} = 3$$

$S^0 = \{1.0, 0\}$

$D_1 = 90\%$
$Cost = 30$

$$r_{S_1} = 1-(1-r_i)^m$$
$$= 1-(1-0.9)^2$$
$$= 0.99$$

device 1

$S_1^1 = \{0.9, 30\}$

$S_1^2 = \{.99, 60\}$

$S^1 = \{(0.9, 30), (0.99, 60)\}$

$D_2 = 70\% = 0.7$
$Cost = 20$

device 2

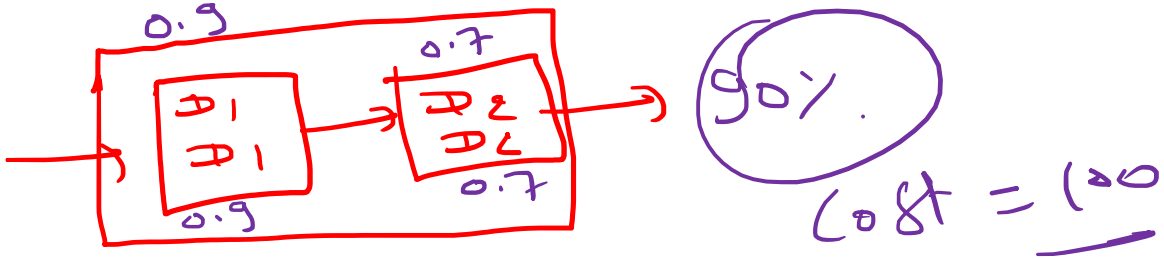
$S_2^1 = \{0.63, 50\}, \{0.693, 80\}\}$

$S_2^2 = \{.819, 70\}, \{.90, 100\}\}$

$S_2^3 = \{0.873, 90\}, \{-, 120\}\}$

$r_{S_2} = 1-(1-0.7)^2 = .91$

$r_{S_2} = 1-(1-0.7)^3 = 0.97$



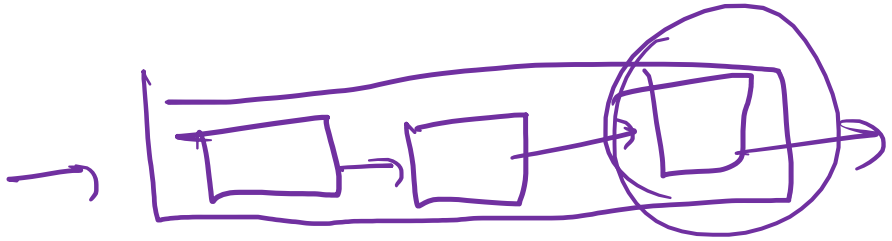
Cost = 100

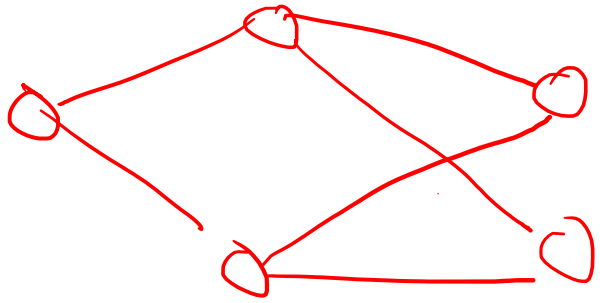
D 194, 215, 222, 225, 229, ~~233~~, D

I ~~484~~, ~~489~~, ~~491~~, 496, 499, 501, 504, ~~507~~, 511, 514
515, 519, L03, L07, L12, L15, L26, L28, L29,

Design a three-stage system with device types D1, D2, and D3. The costs are $30, $15, and $20 respectively. The cost of the system is to be no more than $105. The reliability of each device type is 0.9, 0.8, and 0.5 respectively.

P_i	C_i	r_i
P_1	30	0.9
P_2	15	0.8
P_3	20	0.5





Bellman Ford Algorithm

Bellman Ford Algorithm

1. Dynamic Programming is used in the Bellman-Ford algorithm.

2. Bellman ford algorithm is a single-source shortest path algorithm. This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph.

3. If the weighted graph contains the negative weight values, then the Dijkstra algorithm does not confirm whether it produces the correct answer or not. In contrast to Dijkstra algorithm, bellman ford algorithm guarantees the correct answer even if the weighted graph contains the negative weight values.

Rule of this algorithm

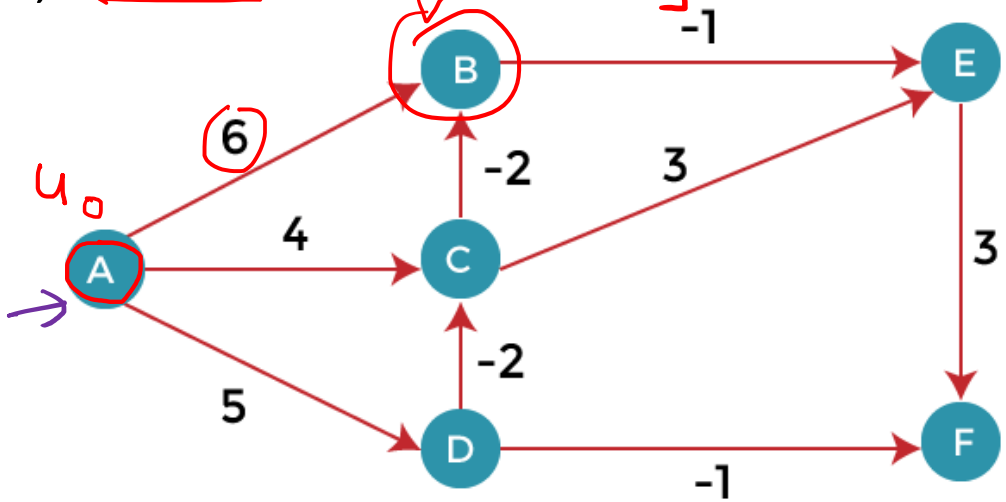
We will go on relaxing all the edges (n - 1) times where, n = number of vertices

Relaxing means:

If (d(u) + c(u , v) < d(v))

then

d(v) = d(u) + c(u , v)



Time Complexity

The time complexity of Bellman ford algorithm would be O(E|V| - 1).

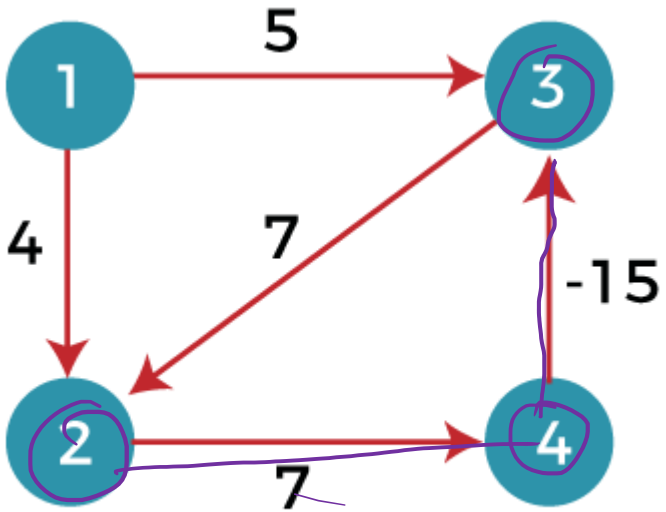
$\Rightarrow O(n^2)$

Complete graph

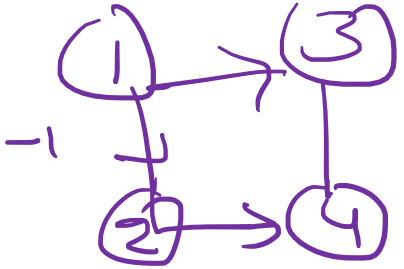
$\frac{n(n-1)}{2} \cdot n = O(n^3)$

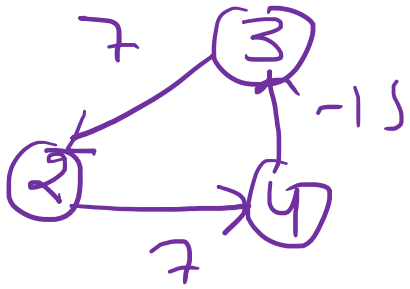
Drawbacks of Bellman ford algorithm

- The bellman ford algorithm does not produce a correct answer if the sum of the edges of a cycle is negative



$\Rightarrow 7 + 7 + (-15) = -1$





Step 1

To find the shortest path of the above graph, the first step is note down all the edges which are given below:

(A, B), (A, C), (A, D), (B, E), (C, E), (D, C), (D, F), (E, F), (C, B)

Let's consider the source vertex as 'A'; therefore, the distance value at vertex A is 0 and the distance value at all the other vertices as infinity shown as below:

n = 6

iteration = 5

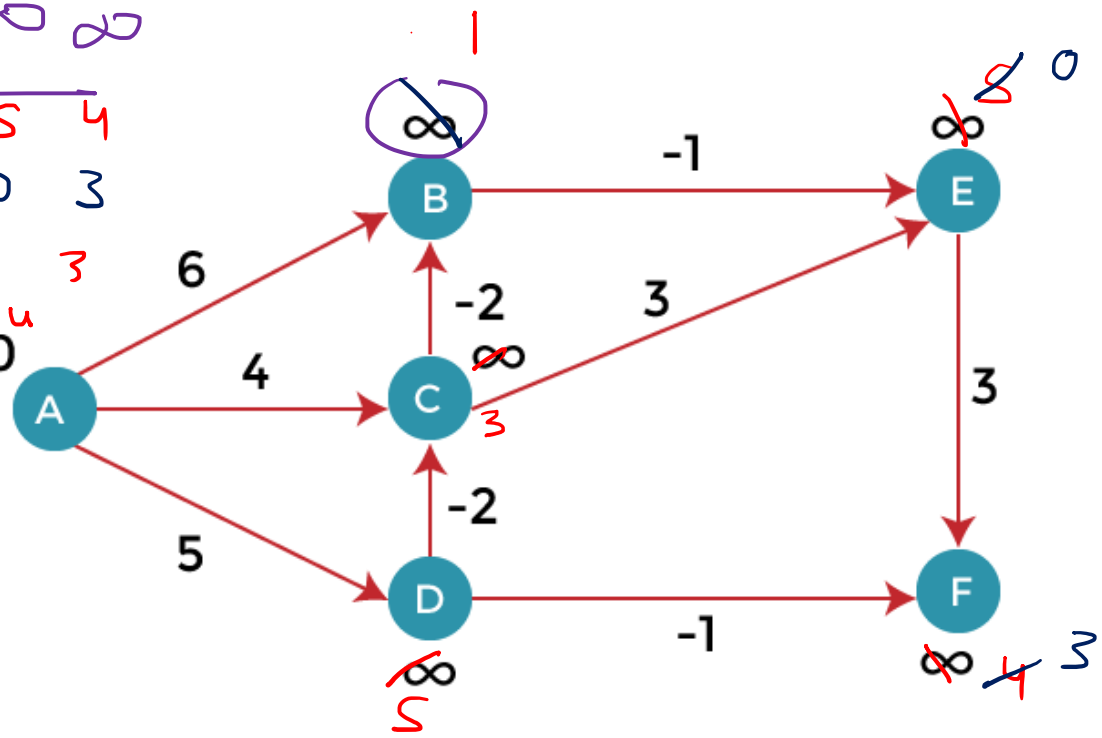
A	B	C	D	E	F
0	∞	∞	∞	∞	∞
0	1	3	5	5	4
0	1	3	5	0	3
0	1	3	5	0	3

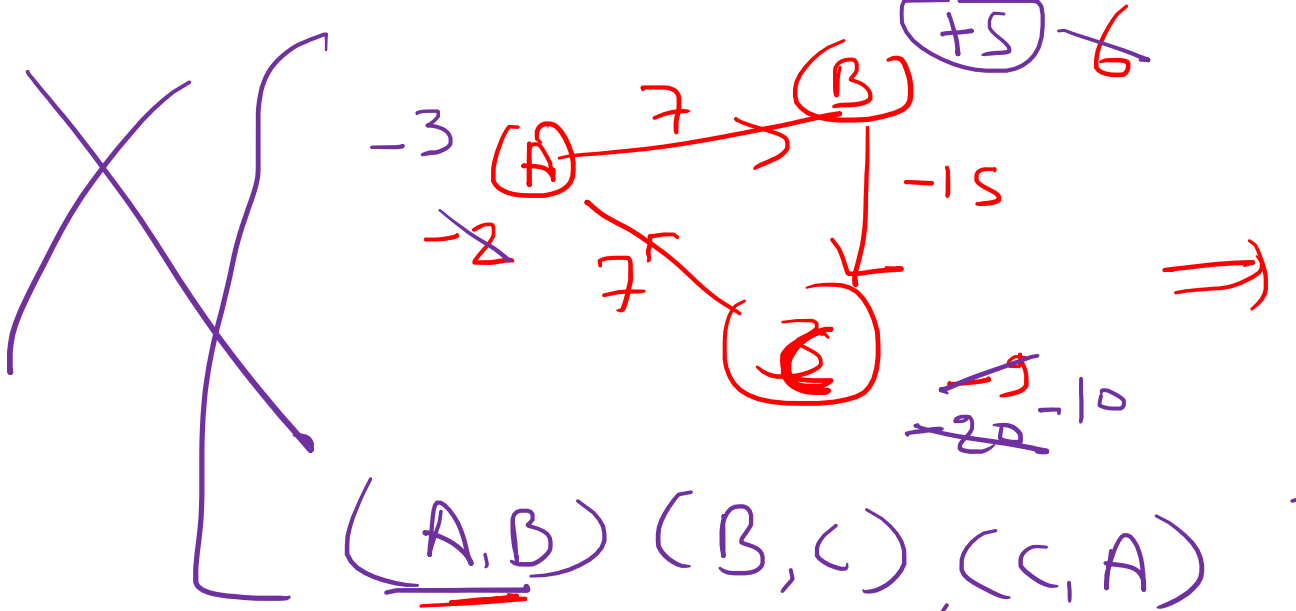
iteration : 1

Iteration 2

Iteration 3

A	B	C	D	E	F
	1	3	5	0	3





$n = 3$

iteration = 2

$\Rightarrow$

iteration 1

(A,B) (B,C), (C,A)

iteration 2

Negative weight cycle

iteration 3

(D) 187, 199, 201, 202, ~~205~~ 206, 207, 212, 213, 215, 216, 217

219, 223, 224 229, 230, ~~234~~, 236, 237, 238, ~~240~~, L13, ~~L37~~,

(I) 499, 504, 512, 519, ~~521~~, 525, L03, L29, T004.