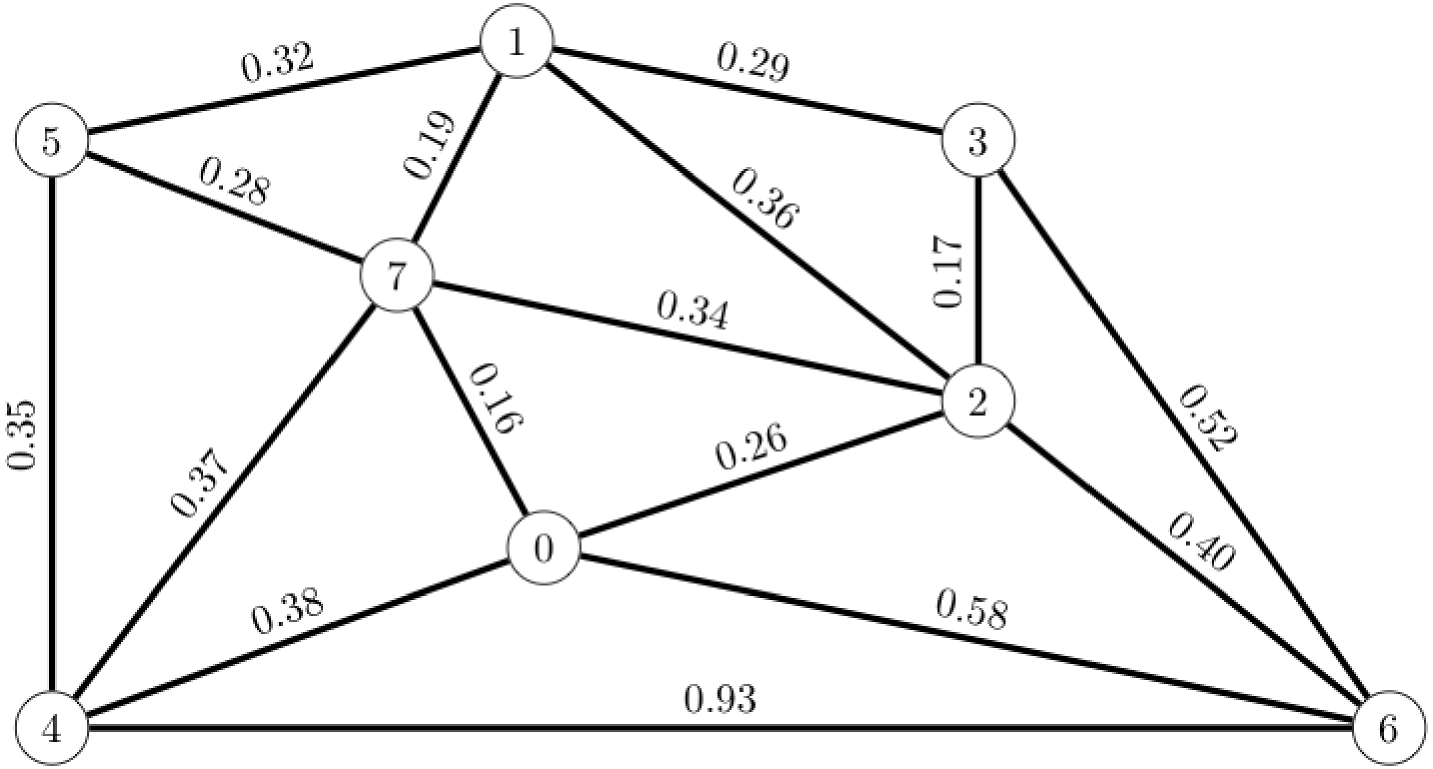
prob.4.

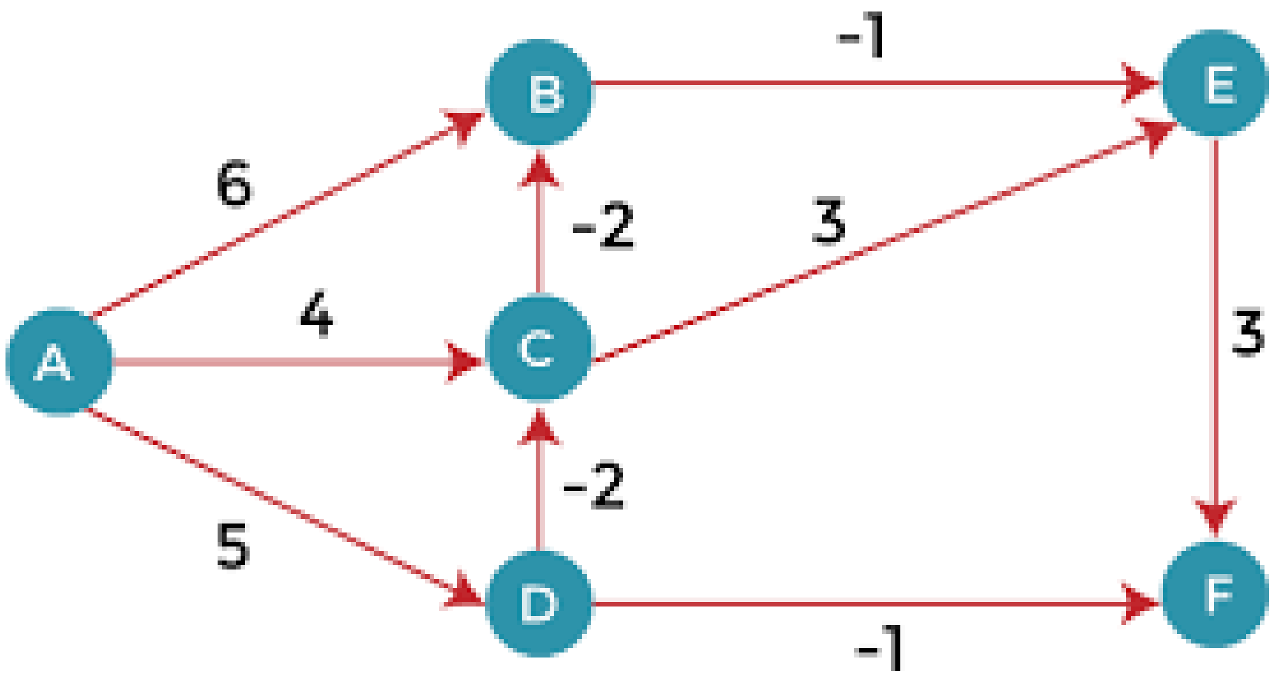
Bellman-Ford Algorithm (Single-source Shortest Path)


0
1
2
3
4
5
6
7
0.32
0.29
0.19
0.28
0.36
0.17
0.34
0.16
0.35
0.37
0.26
0.52
0.40
0.38
0.58
0.93

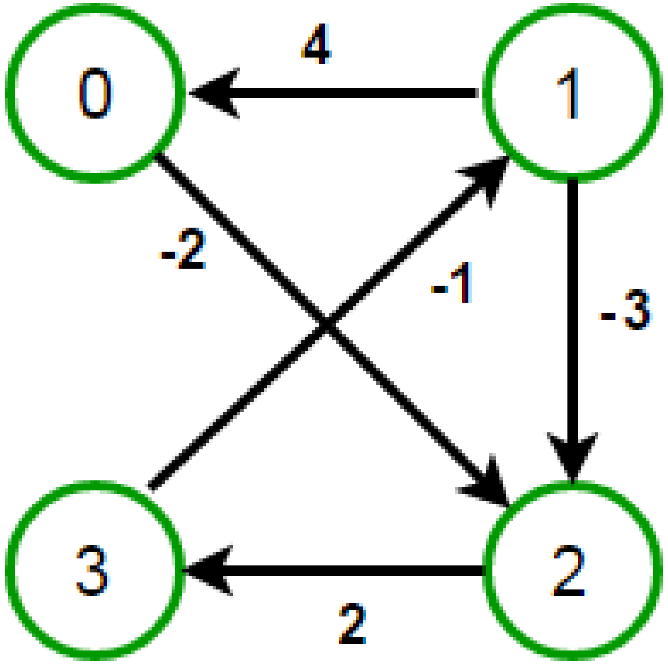
single source shortest path

- We have already solved this problem using Greedy approach (Dijkastra's algorithm) .
- Limitation is that Dijkastra algo. cannot handle graphs with negative edge weights.
- Bellman-Ford algorithm uses principle of optimality and edge relaxation procedure.
- (However, it cannot handle negative cycles in a graph).

Graph with Negative edges



Graph with Negative Cycle



Bellman-Ford Approach

- *Given a graph with |V| vertices, First step initializes distances from the source to all vertices as infinite*
- *and distance to the source itself as 0.*
- *Second step creates an array dist[ ] of size |V| with all values as infinite except dist[src] where src is source vertex.*
- *The next step calculates shortest distances.*
- *Iterate following |V|-1 times*

    Do following for each edge u-v

        if dist[u] + weight of edge u-v < dist[v]

            then update dist[v] = dist[u] + weight of edge u-v

- *If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle*

- *//To report if there is a negative weight cycle in the graph.*

  Again iterate for each edge u-v

  if dist[u] + weight of edge u-v < dist[v]

  then "Graph contains negative weight cycle"

-

- Initialization: Path lengths from all vertices to source are set to ∞.
- Relaxation: After initialization, every edge considered for relaxation.
- Reduce the upper bound of the edge of the shortest path to length of actual shortest path.
- DP approach:
- if $d(u) + cost(u,v) < d(v)$ then

  $$d(v) = d(u) + cost(u,v)$$

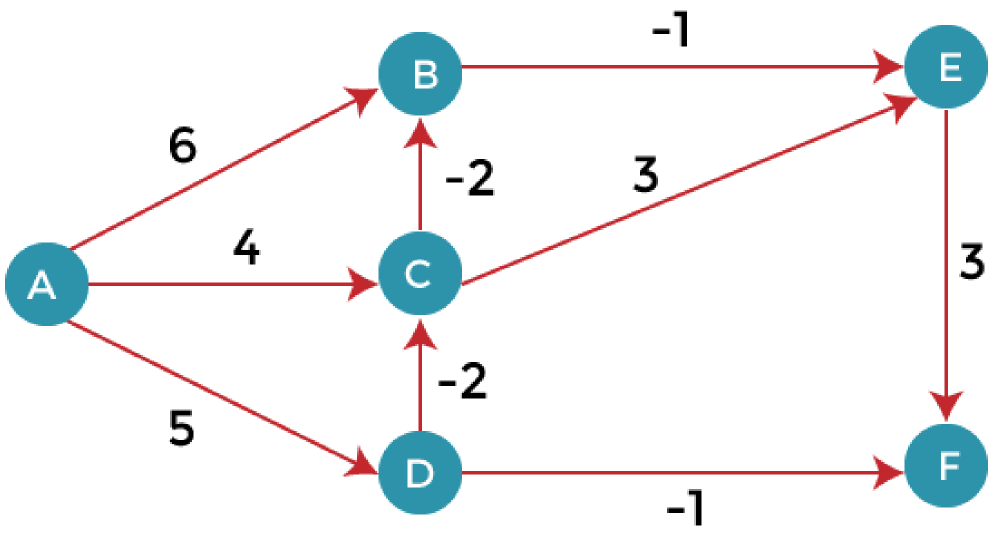
- After 1st iteration, shortest path from s to all immediate neighbors that are one hop away is updated. (vertices connected by one edge)
- After 2nd iteration, all vertices connected to s by two hops are updated.
- Process repeated n-1 times.

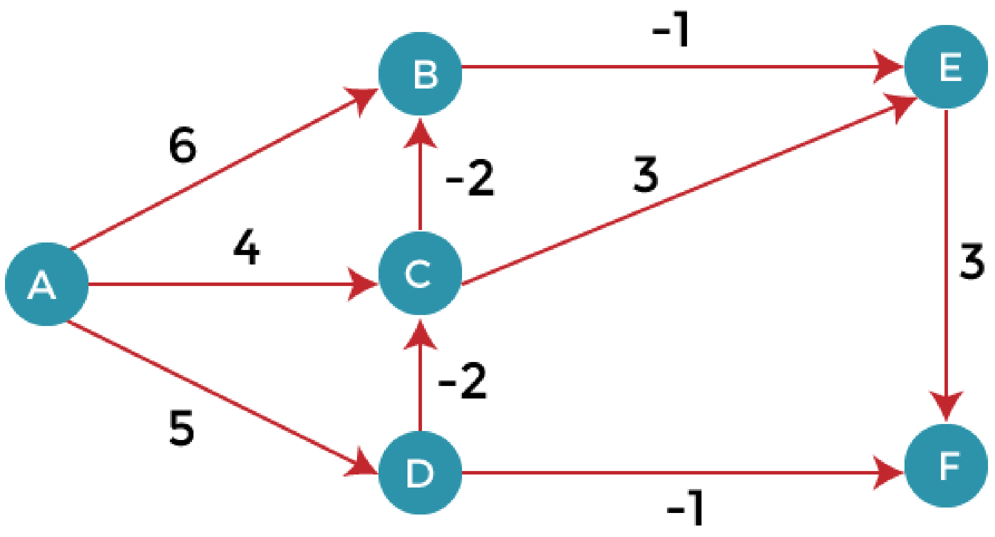
Bellman-Ford Algo.

- n = vertices in the graph
- repeat n – 1 times

```
        for each edge(u,v) do {
          if  d(u) + cost(u,v)  < d(v) then        //relax
              d(v) = d(u) + cost(u,v) ; }
        for each edge(u,v) do   {                  // check for negative cycle
          if  d(v) > d(u) + cost(u,v) then
              output  'Negative Cycle Present' }
```

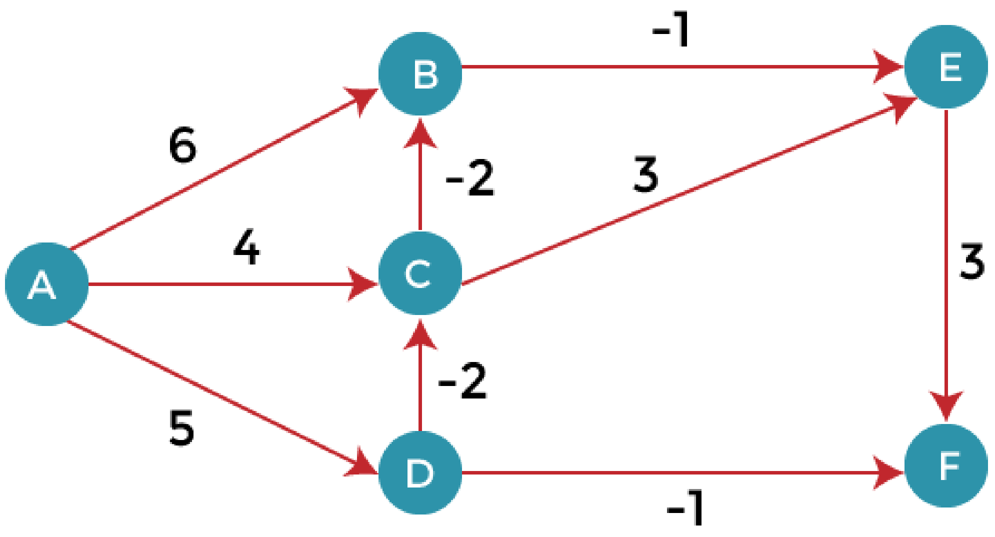
- Find shortest paths to all nodes starting from node A.



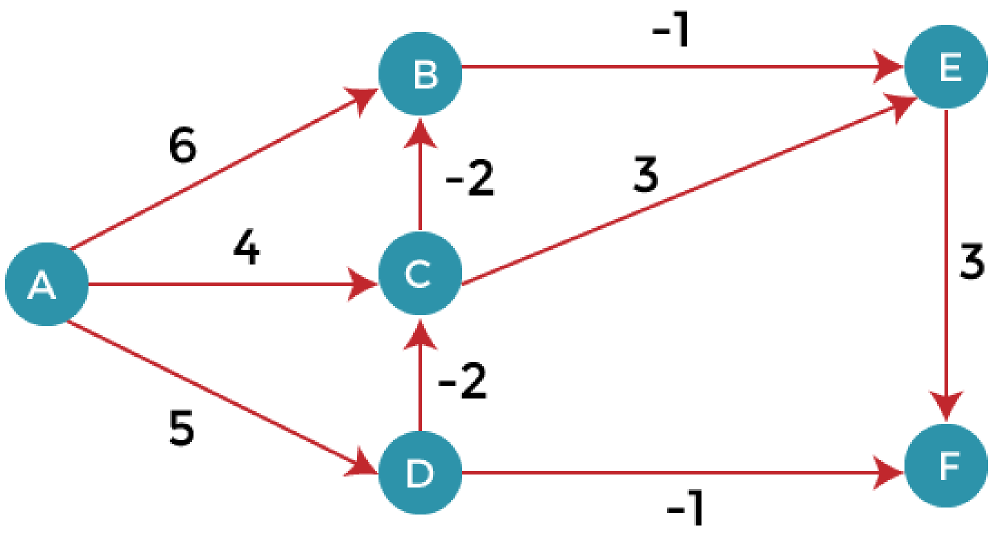


	B	C	D	E	F
	∞	∞	∞	∞	∞
AB	6	∞	∞	∞	∞
AC	6	4	∞	∞	∞
AD	6	4	5	∞	∞
CB	2	4	5	∞	∞
DC	1	3	5	∞	∞
BE	1	3	5	5	∞
CE	1	3	5	5	∞
DF	1	3	5	5	4
EF	1	3	5	5	4

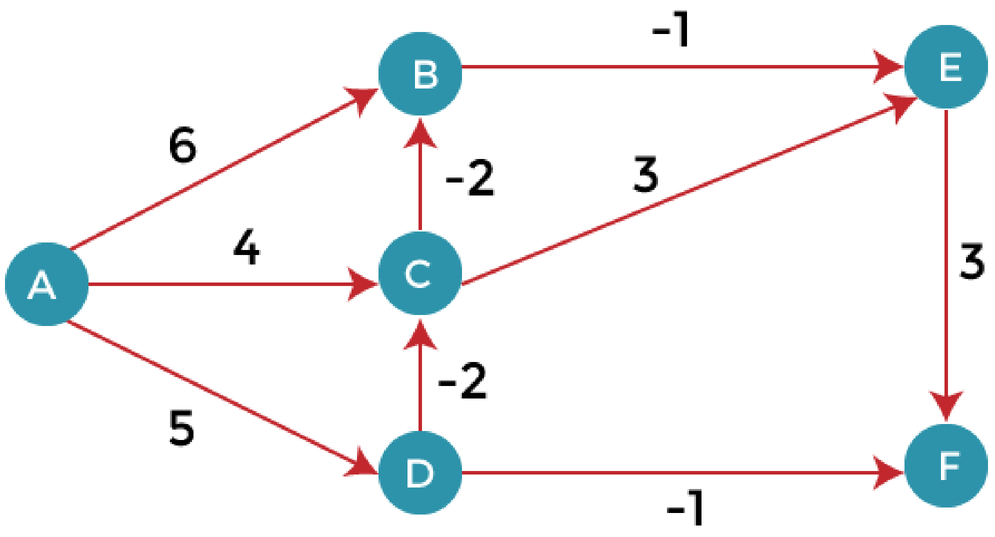
Different choice



	B	C	D	E	F
	∞	∞	∞	∞	∞
AB	6	∞	∞	∞	∞
AC	6	4	∞	∞	∞
AD	6	4	5	∞	∞
BE	6	4	5	5	∞
CE	6	4	5	5	∞
DC	6	3	5	5	∞
DF	6	3	5	5	4
EF	6	3	5	5	4
CB	1	3	5	5	4

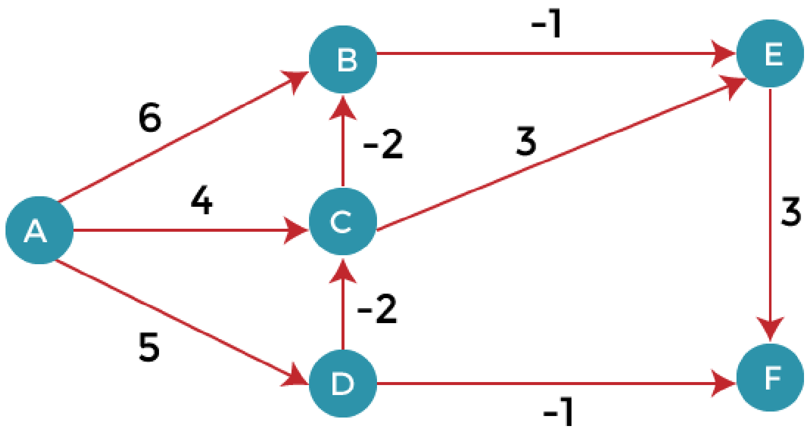


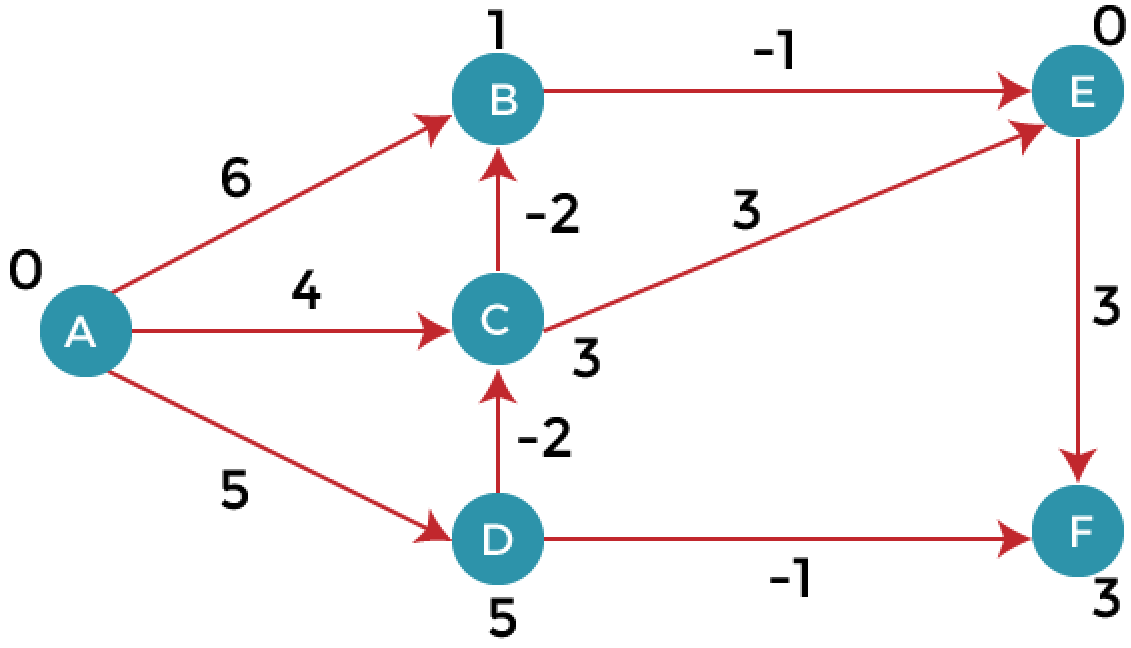
- First iteration is now over
- We carry out the second iteration going through all the edges once again

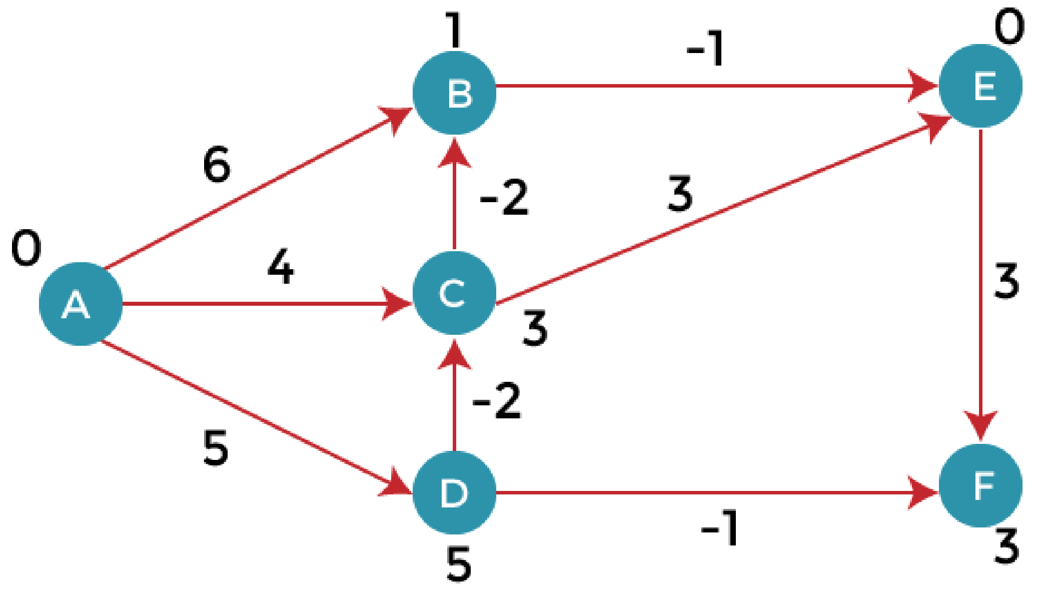


	B	C	D	E	F
---	1	3	5	5	4
AB	1	3	5	5	4
AC	1	3	5	5	4
AD	1	3	5	5	4
BE	1	3	5	0	4
CE					
DC					
DF					
EF	1	3	5	5	3
CB					

- path lengths on the graph



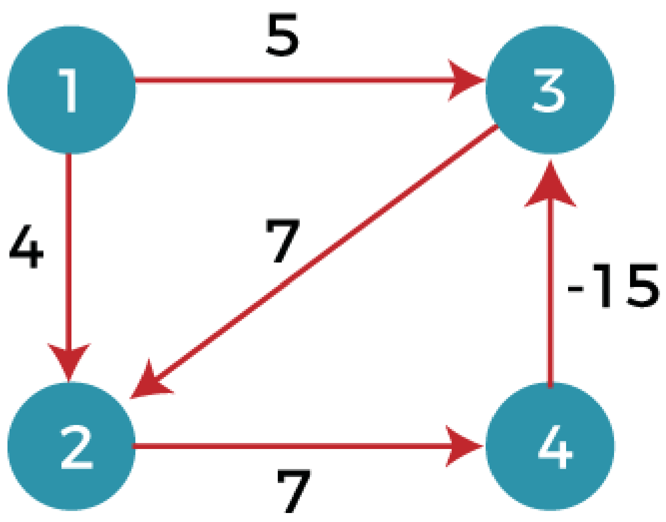




- Check that third iteration produces no change in path lengths.

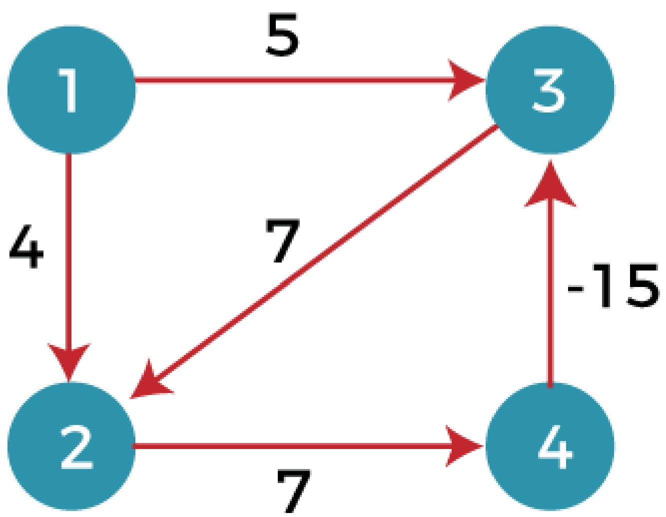
- So we need not go for 4th and 5th iterations.

- Now we take up a case to illustrate Negative Cycle in a graph

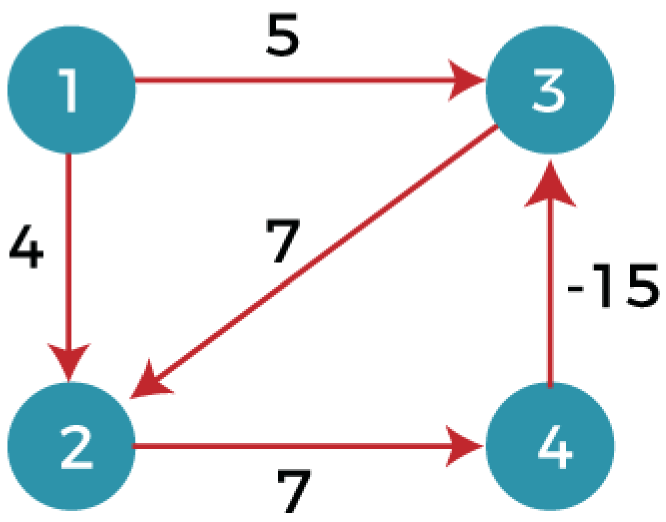


- FIRST ITERATION
-   2 3 4
- 1-3 ∞ 5 ∞
- 1-2 4 5 ∞
- 3-2 4 5 ∞
- 2-4 4 5 11
- 4-3 4 **- 4** 11

- All the edges have been considered.
- Now go for second iteration



- There are 4 edges, so we need to go through 3 iterations.
- SECOND ITERATION
- 2 3 4
- old 4 - 4 11
- 1-3 4 - 4 11
- 1-2 4 - 4 11
- 3-2 3 - 4 11
- 2-4 3 - 4 10
- 4-3 3 - 5 10



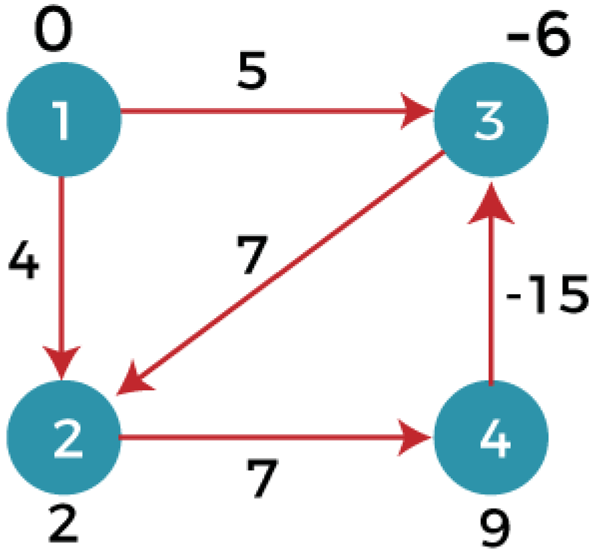
- THIRD ITERATION

	2	3	4
	3	- 5	10
1-3	3	- 5	10
1-2	3	- 5	10
3-2	2	- 5	10
2-4	2	- 5	9
4-3	2	- 6	9

- THIRD ITERATION

	2	3	4
	3	- 5	10
1-3	3	- 5	10
1-2	3	- 5	10
3-2	2	- 5	10
2-4	2	- 5	9
4-3	2	- 6	9

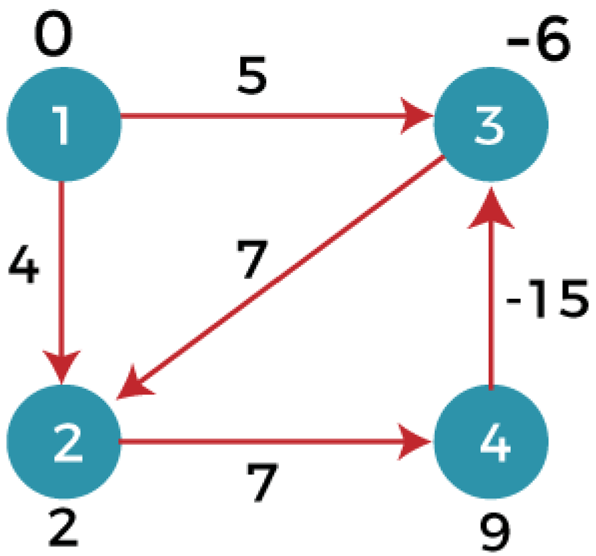
- New distances shown on graph



- There are 4 vertices in the graph
- So there should be no change after 3rd iteration.
- If there is a change, that indicates presence of a NEGATIVE CYCLE

- *FOURTH ITERATION*

	2	3	4
	2	- 6	9
1-3	2	- 6	9
1-2	2	- 6	9
3-2	1	- 6	9



- Since there is a change, it is evident that there is a negative cycle in the graph.

Complexity of Bellman-Ford

- If the graph has n vertices and m edges,
- the complexity is ***O(mn).***

<u>prob. 5</u>

Matrix chain multiplication

- Scientific work many times involves multiplication of chain of matrices
- A B D F T R D M N
- So does order of multiplication affects total number of computations any way?
- Consider chain multiplication of A B C.
- (A B) C or A (B C) would produce the same result.
- But total number of multiplications <u>need not be same.</u>

- The cost of multiplying 2 matrices A(i x j) and B(j x k) is i x j x k
- Suppose A is 2x3, B is 3x4, C is 4x5
- Let us do [BC] 3x4 with 4x5. the result is 3x5 matrix
- Now multiply A with [BC]. 2x3 with 3x5 . It results in 2x5 matrix
- In terms of number of multiplications
- A [B C] = 3 x 4 x 5 + 2 x 3 x 5 = 60 + 30 = 90 ,

- but, [A B] C = 2 x 3 x 4 + 2 x 4 x 5 = 24 + 40 = 64

- so the order does matter.
- Brute force may not produce optimal order of multiplication, when matrix sizes are large, and the matrix chain is long.
- Different ways of grouping 4 matrices.

- ((AB)C) (D)
- ((A(BC))D)
- (AB)(CD)
- A((BC)D)
- A(B(CD))

DP Approach

- A chain A B C D E needs to be split after k matrices
- (A B C) D E
- *Divide and conquer strategy* cannot be used, as value of k is not known beforehand.
- DP approach tries all possible values of k and stores them on a table as has been shown earlier for other DP applications.

- The DP algorithm computes the minimum number of multiplications to multiply a sequence of n matrices.
- For this first of all we need to create an array named *size* that contains sizes of matrices to be multiplied
- Thus if A is 4x5, B is 5x8, C is 8x6, D is 6x3, *size* will be the nx1 array
- [4 5 8 6 3]

- A 2 dimensional array named s[,] is used to store partial solutions.
- s[i , j] stores the minimum number of multiplications needed to multiply matrices i through j.

- s[1,1] = 0 , as it refers to just the first matrix
- In fact all s[i, i]= 0 , as it refers to simply the ith matrix.

- s[1,2] stores multiplications needed for first matrix and second matrix

- for i = 1 to n

```
        s[ i, i ] = 0
for w = 1 to n – 1     // where w = j – i
        for i =  1 to n – w  {
                s[ i, j ] = infinity
                for k = i to j – 1 {
                        Q = s [ i, k ] + s[ k+1, j] + size[i-1] * size[k] * size[j]
                        if ( Q  <  s[i,j ] )
                                s[ i, j ] = Q      // replace by the smaller value
                }
        }
}
```

Complexity of DP matrix chain

- For a chain of n matrices,
- Each "*for*" loop runs in time O(n).
- There are 3 nested *for* loops
- the DP algorithm runs in $O(n^3)$ time

Example 1.

- Consider the matrix chain
- A(4x3) B(3x5) C(5x2)

- brute force way to figure out best grouping out of 2 possibilities
- [A(4x3) B(3x5)] [C(5x2)] 4x3x5+4x5x2 = 100
- [A(4x3)] [B(3x5) C(5x2)] 4x3x2+3x5x2 = 54

- Now we shall show how DP algorithm can be used to do it automatically

- Let us trace the Dyn Prog algorithm for
- A(4x3) B(3,5) C(5x2)

- First form the array named *size* , based on dimensions of the matrices
- *size* = [4 3 5 2]

- s[i, j]
- set s[i , i] = 0

	1	2	3
1	0		
2		0	
3			0

- s[i, j]
- size = [4 3 5 2]

	1	2	3
1	0	∞	
2		0	
3			0

- set w = 1, multiplying AB
- s[1,2] = infinity
- i =1, k =1, j = 2
- *Q = s [i, k] + s[k+1, j]*
  *+ size[i-1] * size[k] * size[j]*

- Q = s [1,1] + s[2,2]
  + size[0] * size[1] * size[2]

Q = 0 + 0 + 4*3*5 = 60

since Q < s[1,2] ,

so set s[1,2] = 60

- s[i, j]
- size = [4 3 5 2]

- 1   2   3

1	0	60	
2		0	
3			0

- set w = 1. multiplying AB
- s[1,2] = infinity
- i =1, k =1, j = 2
- *Q = s [i, k] + s[k+1, j]*

  *+ size[i-1] * size[k] * size[j]*

- Q = s [1,1] + s[2,2]

  + size[0] * size[1] * size[2]

  = 0+0+4*3*5 = 60

Q < s[1,2] , <u>so set s[1,2] = 60</u>

- s[i, j]
- size = [4 3 5 2]

- 1   2   3

1 0   60

2   0   ∞

3   0

- set w = 1, multiplying BC
- s[2,3] = infinity
- i =2, k =2, j = 3
- *Q = s [i, k] + s[k+1, j]*

  *+ size[i-1] * size[k] * size[j]*

- Q = s [2,2] + s[3,3]

  + size[1] * size[2] * size[3]

  = 0+0+3*5*2 = 30

Q < s[2,3] , so set s[2,3] = 30

- s[i, j]
- size = [4 3 5 2]

- 1 2 3

	1	2	3
1	0	60	
2		0	30
3			0

- set w = 1. multiplying BC
- s[2,3] = infinity
- i =2, k =2, j = 3
- *Q = s [i, k] + s[k+1, j]*

  *+ size[i-1] * size[k] * size[j]*

- Q = s [2,2] + s[3,3]

  + size[1] * size[2] * size[3]

  = 0+0+3*5*2 = 30

Q < s[2,3] , <u>so set s[2,3] = 30</u>

- s[i, j]
- size = [4 3 5 2]

	1	2	3
1	0	60	∞
2		0	30
3			0

- set w = 2, multiplying A.[BC]
- set s[1,3] = infinity
- *Q = s [i, k] + s[k+1, j]*

  *+ size[i-1] * size[k] * size[j]*

There are 2 ways of computing s[1,3] , with k=1, and k=2

i = 1, k = 1,   j = 3

- Q=s[1,1] + s[2,3]

  +size[0] *size[1] * size[3]

  = 0+30+4*3*2 = 54

Q < s[1,3] , so set s[1,3] = 54

- s[i, j]
- size = [4 3 5 2]

- 1   2   3

1 0   60   54

2   0   30

3   0

- set w = 2, multiplying A.[BC]
- set s[1,3] = infinity
- *Q = s [i, k] + s[k+1, j]*

  *+ size[i-1] * size[k] * size[j]*

There are 2 ways of computing s[1,3] , with k=1, and k=2

i = 1, k = 1,   j = 3

- Q=s[1,1] + s[2,3]

  +size[0] *size[1] * size[3]

  = 0+30+4*3*2 = 54

Q < s[1,3] , <u>so set s[1,3] = 54</u>

- s[i, j]
- size = [4 3 5 2]

	1	2	3
1	0	60	~~54~~
2		0	30
3			0

- second way of computing s[1,3]
- multiplying [AB].C
- *Q = s [i, k] + s[k+1, j]*

  *+ size[i-1] * size[k] * size[j]*

  i = 1, k = 2, j = 3
- Q=s[1,2] + s[3,3]

  +size[0] *size[2] * size[3]

  = 60+ 0 +4*5*2 = 100

since Q not < s[1,3] ,

so <u>keep old value of s[1,3] = 54</u>

- s[i, j]
- size = [4 3 5 2]

	1	2	3
1	0	60	54
2		0	30
3			0

- Table is now complete
- k=1 gives best value of 54

- Verify
- (A) (BC)
- BC = 3x5x2 = 30
- A.BC = 4x3x2 = 24

Example 2.

- Let us trace the DP algorithm for chain of 4 matrices
- A(5x3) B(3,1) C(1x4) D(4x6)
- size = [5 3 1 4 6]

- s[i, j]

	1	2	3	4
1	0			
2		0		
3			0	
4				0

- w = 0
- set s[i , i] = 0
- Now we need to compute s[1,2], s[2,3], s[3,4]
- Next we shall compute s[1,3], s[2,4]
- Finally, we shall compute s[1,4]

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	∞		
2		0		
3			0	
4				0

- set w = 1, compute s[1,2]
- multiplying [AB]
- (AB) s[1,2] = ∞
- Q = s [i, k] + s[k+1, j]

  + size[i-1] * size[k] * size[j]

- Q = s [1,1] + s[2,2]

  + size[0] * size[1] * size[2]

  = 0+0+5*3*1 = 15

Q < s[1,2] , so set s[1,2] = 15

- s[i, j]
- size = [5 3 1 4 6]

- 1 2 3 4

	1	2	3	4
1	0	15		
2		0		
3			0	
4				0

- set w = 1
- [AB]
- s[1,2] = infinity
- Q = s [i, k] + s[k+1, j]

  + size[i-1] * size[k] * size[j]

- Q = s [1,1] + s[2,2]

  + size[0] * size[1] * size[2]

  = 0+0+5*3*1 = 15

Q < s[1,2] , so set s[1,2] = 15

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15		
2		0		
3			0	
4				0

- Now compute s[2,3] and s[3,4] yourself

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15		
2		0	12	
3			0	24
4				0

- set w = 1
- s[1,2] = infinity
- Q = s [i, k] + s[k+1, j]
  + size[i-1] * size[k] * size[j]

- Q = s [1,1] + s[2,2]
  + size[0] * size[1] * size[2]
  = 0+0+5*3*1 = 15

Q < s[1,2] , so set s[1,2] = 15

- [BC] s[2,3] = 3*1*4 = 12
- [CD] s[3,4] = 1*4*6 = 24

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15		
2		0	12	
3			0	24
4				0

- Now we know computations needed for [A B] , [B C], and [C D]

- Next set gap w = 2,
- compute first s[1,3] [ABC]
- and later s[2,4] [BCD]
- 2 ways of computing s[1,3]
- k =1 [A] [BC]

- and k = 2 [A B] [C]

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15		
2		0	12	
3			0	24
4				0

- set w = 2,
- compute s[1,3]   [ABC]
- 2 ways of computing s[1,3]
- k =1 and k = 2
- Q = s [i, k] + s[k+1, j]

  + size[i-1] * size[k] * size[j]

<u>First way of computing s[1,3]</u>

i = 1, k = 1,   j = 3   [A] [BC]

- Q=s[1,1] + s[2,3] +size[0] *size[1] * size[3]

  = 0+12+5*3*4 = 72

Q < s[1,3] , so set s[1,3] = 72

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15	72	
2		0	12	
3			0	24
4				0

- set w = 2,
- compute first s[1,3] [ABC]
- and later s[2,4] [BCD]
- 2 ways of computing s[1,3]
- k =1 and k = 2
- Q = s [i, k] + s[k+1, j]

  + size[i-1] * size[k] * size[j]

<u>First way of computing s{1,3]</u>

i = 1, k = 1, j = 3

- Q=s[1,1] + s[2,3] +size[0] *size[1] * size[3]

  = 0+12+5*3*4 = 72

Q < s[1,3] , <u>so set s[1,3] = 72</u>

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15	~~72~~	
2		0	12	
3			0	24
4				0

- First value of s[1,3] = 72.
- Second way of computing s[1,3]
- i = 1, k = 2,   j = 3   3   [A B] [C]

Now by taking k =2, second value of Q is

Q = s[1,2] + s[3,3] +size[0] *size[2] * size[3]

= 15 +0 + 5*1*4 = 35

since Q < s[1,3] *(which was 72),* reset

s[1,3] = 35

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15	35	
2		0	12	
3			0	24
4				0

- First value of s[1,3] = 72.
- Second way of computing s[1,3]
- i = 1, k = 2,   j = 3   3   [A B] [C]

Now by taking k =2, second value of Q is

Q = s[1,2] + s[3,3] +size[0] *size[2] * size[3]

= 15 +0 + 5*1*4 = 35

since Q < s[1,3] *(which was 72),* reset

s[1,3] = 35

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15	35	
2		0	12	42
3			0	24
4				0

- Similarly compute s[2,4] [BCD]
- set s[1,3] = infinity
- *Q = s [i, k] + s[k+1, j]*

  *+ size[i-1] * size[k] * size[j]*

i = 2, k = 2, j = 4

- Q=s[1,1] + s[2,3] +size[0] *size[1] * size[3]

  = 0+12+5*3*4 = 72

Q < s[1,3] , so set s[1,3] = 72

s[2,4] = 42

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15	35	
2		0	12	42
3			0	24
4				0

- Similarly compute s[2,4] [BCD]
- set s[1,3] = infinity
- *Q = s [i, k] + s[k+1, j]*

  *+ size[i-1] * size[k] * size[j]*

i = 2, k = 2, j = 4

- Q=s[1,1] + s[2,3] +size[0] *size[1] * size[3]

  = 0+12+5*3*4 = 72

Q < s[1,3] , so set s[1,3] = 72

s[2,4] = 42

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15	35	132
2		0	12	42
3			0	24
4				0

- Finally, set gap w = 3,
- compute s[1,4]   [ABCD]
- Q = s [i, k] + s[k+1, j]

  + size[i-1] * size[k] * size[j]

  i = 1, k = 1,   j = 4   [A] [BCD]
- Q=s[1,1] + s[2,4] +size[0] *size[1] * size[4]

  = 42+ 5*3*6 = 132

Set s[1,4] to 132

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15	35	~~132~~
2		0	12	42
3			0	24
4				0

- Finally, set w = 3, compute s[1,4]
- Q = s [i, k] + s[k+1, j]

  + size[i-1] * size[k] * size[j]
- Q=s[1,1] + s[2,4] +size[0] *size[1] * size[4]

  = 42+ 5*3*6 = 132

Set s[1,4] to132

second value of Q, [A B] [CD]

i = 1, k = 2, j = 4

- Q=s[1,2] + s[3,4] +size[0] *size[2] * size[4]

  = 15+ 24 + 5*1*6 = 69

Set s[1,4] to 69 , as it is less than 132

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15	35	69
2		0	12	42
3			0	24
4				0

- Finally, set w = 3, compute s[1,4]
- Q = s [i, k] + s[k+1, j]

  + size[i-1] * size[k] * size[j]
- Q=s[1,1] + s[2,4] +size[0] *size[1] * size[4]

  = 42+ 5*3*6 = 132

Set s[1,4] to132

second value of Q, [A B]   [CD]

i = 1, k = 2,   j = 4

- Q=s[1,2] + s[3,4] +size[0] *size[2] * size[4]

  = 15+ 24 + 5*1*6 = 69

Set <u>s[1,4] to 69</u> , as it is less than 132

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15	35	~~69~~
2		0	12	42
3			0	24
4				0

Third way of computing Q,

i = 1, k = 3, j = 4 [ABC] [D]

- Q=s[1,3] + s[4,4] +size[0] *size[3] * size[4]

  = 35+ 0 + 5*4*6 = 155

Since Q is not less than 69, it remains at 69 .

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15	35	69
2		0	12	42
3			0	24
4				0

Third way of computing Q,

i = 1, k = 3, j = 4 [ABC] [D]

- Q=s[1,3] + s[4,4] +size[0] *size[3] * size[4]

  = 35+ 0 + 5*4*6 = 155

Since Q is not less than 69, it remains at 69 .

- s[i, j]
- size = [5 3 1 4 6]

	1	2	3	4
1	0	15	35	69
2		0	12	42
3			0	24
4				0

- Thus the best value for [ABCD] is 69.
- How to group the matrices?
- It was seen that k = 2 gives minimum value of 69
- So grouping is

- [A B] [C D]

- verify
- mults for AB = 5x3x1 =15
- mults for CD = 1x4x6 = 24
- mults for AB . CD = 5x1x6 = 30

```
• for i = 1 to n                        //n is number of matrices
        s[ i, i ] = 0
  for w = 1 to n – 1    {                                  // where w = j – i
        for i =  1 to n – w  {
              s[ i, j ] = infinity
              for k = i to j – 1 {
                    Q = s [ i, k ] + s[ k+1, j] + size[i-1] * size[k] * size[j]
                    if ( Q  <  s[i,j ] )
                          set s[ i, j ] = Q
              }
        }
  }
```

Complexity of matrix chain algo.

- Since there are 3 for loops in the algorithm,
- the complexity of the algorithm is $O(n^3)$.