Dynamic Programming

Dynamic programming, like divide and conquer method, solves the problems by combining the solutions to the sub-problems.

[Divide and conquer algorithm partition the problem into independent sub-problems, solves the sub problems recursively and then combine their solution to solve the original problem.]

Dynamic programming is a technique, same as divide and conquers method, but in this, the sub-problems are not independent. This means sub-problems share the solutions of sub-problems.

Dynamic programming solves every sub-problem once, and then saves their answer in a table, so whenever the sub-problem is again encountered, the work of re-computation of the answer to the sub-problem is avoided.

Dynamic programming is applied to these problems, which exhibits following properties.

- **Optimal substructure** property says that, an optimal solution to the problem contains within it, optimal solution to the sub problems.
- A problem has **overlapping sub problem** property, when a recursive algorithm revisits the same problem repeatedly.

Dynamic Programming involves in following 4 steps.

- Characterize the structure of an optimal solution.
- Recursively define the value of optimal solution
- Compute the value of optimal solution in bottom up fashion.
- Construct the optimal solution from computed information.

Problems Can be solved using Dynamic programming:

- *Assembly line scheduling problem*
- *Matrix chain multiplication*
- *Longest common subsequences*
- *Optimal Binary Search tree,*
- *0-1 Knapsack problem*
- *Warshall's algorithm for transitive closure of a graph*
- *Floyd's algorithm for all-pair shortest path problem*
- *Longest increasing subsequence*

Matrix Chain Multiplication

Given a series of compatible matrices, the aim of MCM is to determine an optimal order for multiplying matrices that has the lower cost.

To multiply a chain of matrices (A1,A2. . . An), first we have to fully parenthesize the chain and then apply the standard algorithm to multiply pair of matrices.

The cost (number of scalar multiplication) of multiplying two matrices A_{pxq} and B_{qXr} is pxqxr.

For a chain of matrices < A1, A2 , A3, A4 >, The product A1 × A2 × A3 × A4 can be fully parenthesize in 5 different ways.

1. (A1* (A2 * (A3 * A4)))
2. (A1*((A2 * A3) * A4))
3. ((A1 * A2)*(A3 * A4))
4. ((A1*(A2* A3)) * A4)
5. (((A1*A2) * A3) *A4)

Each parenthesization has different cost in computing the matrix product.

Example: Given a chain of matrices (A1,A2,A3) of size 10 × 100, 100 × 15, 15 × 50, Find the cost of following parenthesization.

1. ((A1 * A2) * A3)
2. (A1*((A2 * A3))

Stpe-1: Determine the structure of optimal solution (Parenthesization)

To show the optimal solution to the MCM problem in term of optimal solution to sub-problem, first we need to parenthesize the sub expression A_i , A_{i+1},. . . A_j to yield multiplication of matrices Ai to Aj. Any parenthesization must split the product A_i to A_j between A_k and A_{k+1} for some $k = i$ to $j-1$.

For some value of k , first compute $(A_i \ldots A_k)$ and $(A_{k+1} \ldots A_j)$ and then multiply them together to produce $(A_i \ldots A_j)$.

The cost of this parenthesization = Cost of computing matrices $(A_i \ldots A_k)$

+ Cost of computing matrices $(A_{k+1} \ldots A_j)$

+ Cost of multiplying them together.

Stpe-2: Recursive Solution

In this step, we define the cost of an optimal solution to the problem recursively in terms of solution to sub-problems. Let M [i,j] be the minimum number of scalar multiplication needed to compute the matrix (Ai,…Aj). So the cost of cheapest way to compute A1. . .An will be in M[1,n].

If i = j , The chain has only one matrix, So no scalar multiplication is needed.

So M[i,j]=0

If i < j, we can use the solution for computing sub-product of step-1.

Let's assume optimal parenthesization splits the product $(A_i \ldots A_j)$ between Ak and Ak+1, where i≤k<j. Then M[i,j] is the minimum cost of computing sub-product $(A_i \ldots A_k)$ and $(A_{k+1} \ldots A_j)$ + cost of multiplying the two matrices.

(Let P contains indices of matrix chain. As Ai is Pi-1 × Pi, computing the matrix product (Ai. . .Ak) × (Ak+1 . . . Aj) require P-1 × Pk × Pj scalar multiplication.

$$M[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} (M[i,k] + M[k+1,j] + P_{i-1} \times P_k \times P_j \ , & \text{if } i<j \end{cases}$$

Stpe-3: Computing optimal cost.

Let $P = <P_0, P_1, . . . , P_n>$ is the input sequence containing the indices of matrices, where Matrix A_i has the dimension $P_{i-1} \times P_i$. Two tables M[1… n, 1…n] and S[1…n-1,2…n] are used, where M stores the cost and S stores the index of k that has the optimal cost.

Algorithm: `MatrixMultiplication(P)`

```
1. n = Length(P) - 1
2. for i =  1 to n
3.     M[i,i] = 0
4. for L = 2 to n
5.     for i = 1 to n +1 - L
6.          j= i + L- 1
7.          M[i,j] = ∞
8.          for k =  i to j-1
9.               q=M[i,k] +M[k+1,j] + P[i-1]×P[k]×P[j]
10.                  if q < M[i,j]
11.                      M[i,j] = q
12.                      S[i,j] = k
13. return M,S
```

Analysis: The basic operation is in line 9. It is present in a 3-level nested loop. The running time of algorithm is $O(n^3)$

Problem: Given a chain of matrices, represented by P =<30, 35, 15, 5, 10, 20, 25>, construct the optimal cost of multiplication. Construct the Cost table (M) and S table.

<u>Chain length =1</u>

M[1,1] = 0
M[2,2] = 0
M[3,3] = 0
M[4,4] = 0
M[5,5] = 0
M[6,6] = 0

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

Cost table M

	2	3	4	5	6
1	1	1	3	3	3
2		2	3	3	3
3			3	3	3
4				4	5
5					5

Root Table S

Chain length =2

M[1,2] = (k=1), M[1,1] + M[2,2] + 30 × 35 × 15 = 15750
M[2,3] = (k=2), M[2,2] + M[3,3] + 35 × 15 × 5 = 2625
M[3,4] = (k=3), M[3,3] + M[4,4] + 15 × 5 × 10 = 750
M[4,5] = (k=4), M[4,4] + M[5,5] + 5 × 10 × 20 = 1000
M[5,6] = (k=5), M[5,5] + M[6,6] + 10× 20 × 25 = 5000

Chain length =3

M[1,3] = Min((k=1), M[1,1] + M[2,3] + 30 *35*5 = 0 +2625+5250 =7875
(k=2), M[1,2] + M[3,3] + 30 * 15 *5 = 15750 + 0 + 2250 = 18000
M[2,4] = Compute for each k= 2,3 and use minimum
M[3,5] = Compute for each k= 3,4 and use minimum
M[4,6] = Compute for each k= 4,5 and use minimum

Chain length =4

M[1,4] = Compute the minimum of each k=1 ,2,3
M[2,5] = Compute the minimum of each k=2,3,4
M[3,6] = Compute the minimum of each k=3,4,5

Chain length =5

M[1,5] = Compute the minimum of each k=1 ,2,3,4
M[2,6] = Compute the minimum of each k=2,3,4,5

Chain length =6

M[1,6] = Compute for each k=1 ,2,3,4,5 and use minimum

Step-4: Constructing optimal solution

Optimal solution is constructed by from computed information stored in S table. The table contains each entry S[I,j], the value of k, which has optimal parenthesization of Ai,Ai+1,..Aj that spits the product between A and Ak+1. So we can write the final matrix multiplication in computing A[1..n] optimally is A[1..S[1,n]] X A[S[1,n]+1…n]

Algorithm: `PrintOptimalSolution(S,i,j)`

```
If i = j
  Print "A"i
Else, print "("
       PrintOptimalSolution(S,i,S[i,j])
       PrintOptimalSolution(S,S[i,j]+1,j)
       Print ")"
```

Longest Common Subsequences

A subsequence of a given sequence is just a sequence with 0 or more elements left out. In LCS problem, we are given two sequences, X =<x1,x2,…xm> and Y=<y1,y2,…yn> and we have to find the maximum length common subsequence of X and Y.

Example: LCS of X =<10012210> and Y=<1120112> is Z=<1120>

Application of LCS: File comparison, Biological application in the form of DNA testing.

Step-1 : Characterizing a longest common subsequence.

The LCS problem has an optimal substructure property. To use this, we must use pair of prefix of the two sequences.

Terminology:
X_i = <x1,x2,..xi> first i elements of X.
x_i = ith element in X.

Optimal substructure of LCS—

Let X =<x1,x2,…xm> and Y=<y1,y2,…yn> be the sequences and let Z=<z1,z2,…zk> be any LCS of X and Y, then

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then if $z_k \neq x_m$ Then Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then if $z_k \neq y_n$ Then Z is an LCS of X and Y_{n-1}.

Step-2: Recursive Solution.

From above theorem, We examine one or two sub-problem when finding n LCS of X and Y.

if $x_m = y_n$, find an LCS of Xm-1 and Yn-1 . then append xm=yn to this LCS

if $x_m \neq y_n$, we have to solve two sub problems, i,e

1. Finding LCS of X_{m-1} and Y.
2. Finding LCS of X and Y_{n-1}.

Which of these two LCS is longer is the LCS of X and Y.

Let C[i,j] is the length of an LCS of two subsequences Xi and Yj.

If i =0 or j = 0, Then either length or X or length of Y is 0. Therefore, LCS has also length 0.

So C[i,j] can be defined as—

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } i = 0 \\ C[i-1,j-1] + 1 & \text{if } i,j>0 \text{ and } xi = yj \\ \text{MAX}(C[i,j-1] ,C[i-1,j] & \text{if } i,j>0 \text{ and } xi \neq yj \end{cases}$$

Step-3: Computing the length of LCS.

`LCS-LENGTH(X,Y)`

X and Y are two sequences of length m and n. Table C[0…m,0…n]stores length of LCS of X and Y.Another table B[1…m,1…n] used to construct optimal solution.

```
m  = X.length
n = Y.length
for i = 0 to m
     c[i,0] = 0
for j = 1 to n
     c[0,j] = 0
for i = 1 to m
     for j = 1 to n
          if x[i] = y[j]
                    C[i,j]  = C[i−1,j-1] + 1
                    B[i,j]= '↖'
          else if C[i−1,j] > C[i,j-1]
                    C[i,j]  = C[i−1,j]
                    B[i,j]= '↑'
          else,     C[i,j]  = C[i,j-1]
                    B[i,j]= '←'
return C and B
```

Example : Compute LCS of X = <ABCBDAB> and Y = < BDCABA>

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0 ↑	0 ↑	0 ↑	1↖	1←	1↖
2	B	0	1↖	1←	1←	1 ↑	2↖	2←
3	C	0	1 ↑	1 ↑	2↖	2←	2↑	2↑
4	B	0	1↖	1 ↑	2 ↑	2 ↑	3↖	3←
5	D	0	1 ↑	2↖	2 ↑	2 ↑	3 ↑	3 ↑
6	A	0	1 ↑	2 ↑	2 ↑	3↖	3 ↑	4↖
7	B	0	1↖	2 ↑	2 ↑	3↑	4↖	4 ↑

Step-4: Constructing an LCS

Table B uses three symbol.

If '↖' is encountered in B[i,j],then xi = yj is an element of LCS.

If B[i,j] = ← , then LCS of Xi and Yj-1 is greater than that of Xi-1 and Yj. So move left to B[i,j-1].

If B[i,j] = ↑ , then LCS of Xi-1 and Yj is greater than that of Xi and Yj-1. So move up to B[i,j-1].

```
Print_LCS(B,X i, j)
1.If i = 0 or j = 0
2.     return
3.If B[i,j] =='↖'
4.        Print_LCS(B,X,i-1,j-1)
5.        print xi
6. Else if B[i,j] =='↑'
7.        Print_LCS(B,X,i-1,j)
8. Else   Print_LCS(B,X,i,j-1)
```

Optimal Binary Search Tree:

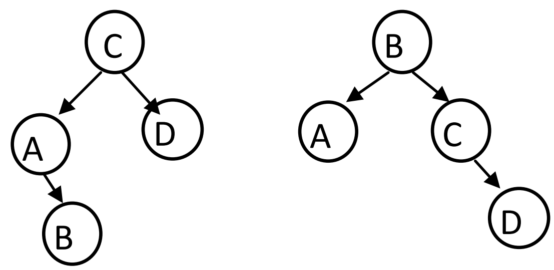
Given a sequence of n distinct keys $K=<k_1,k_2, ...K_n>$ in ascending order and $P=<p_1,p_2,\dots p_n>$ is the probability sequence such that p_i is the probability of searching of Key k_i.

The objective is to find a binary search tree for which the average number of comparison is smaller.

Example: Consider the table of Keys and Probability of searching the keys .

K	A	B	C	D
P	0.1	0.2	0.4	0.3

Suppose we construct the following binary search trees.



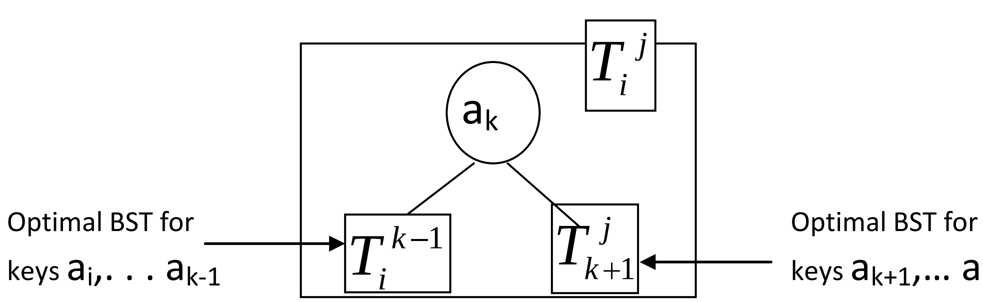
Average number of comparison in successful search in

First tree is = 2×0.1 + 3×0.2 + 1 × 0.4 + 2 ×0.3 = 1.8

Second tree is = 2×0.1 + 1×0.2 +2 × 0.4 +3 ×0.3 = 2.1

Step-1: Structure of Binary search Tree

An optimal binary search tree T_i^j consist of keys $a_i, . . . a_j$ optimally arranged. Let ak is the root of T_i^j, then its left sub tree T_i^{k-1} contains the keys $a_i, . . . a_{k-1}$ optimally arranged and its right sub tree T_{k+1}^j contains the keys $a_{k+1}, . . . a_j$ optimally arranged.



Step-2: Recursive Solution:

Let C[i,j] is the average number of comparison in a BST with keys a_i,.. a_j ,$1 \le i \le j \le n$.

If i = j , then the BST consist of one node, So average number of comparison C[i,j]=

$1 \times P_i = P_i$

If i > j, then there is no BST at all , So C[i,j] = 0

If i < j, Then C[i,j] = Pk ×1

+ average number of comparison of Binary sub tree T_i^{k-1}

+ average number of comparison of Binary sub tree T_{k+1}^{j}

So, C[i,j] =

$$\underset{i \le k \le j}{Min} \left\{ P_k + \sum_{s=i}^{k-1} ps \times (level\ of\ a_s\ in\ T_i^{k-1} + 1) + \sum_{s=k+1}^{j} ps \times (level\ of\ a_s\ in\ T_{k+1}^{j} + 1) \right.$$

$$= \underset{i \le k \le j}{Min} \left\{ P_k + \sum_{s=i}^{k-1} ps \times level\ of\ a_s\ in\ T_i^{k-1} + \sum_{s=i}^{k-1} ps \right.$$

$$+ \sum_{s=k+1}^{j} ps \times level\ of\ a_s\ in\ T_{k+1}^{j} + \sum_{s=k+1}^{j} ps$$

$$= \underset{i \le k \le j}{Min} \left\{ \sum_{s=i}^{k-1} ps \times level\ of\ a_s\ in\ T_i^{k-1} + \sum_{s=k+1}^{j} ps \times level\ of\ a_s\ in\ T_{k+1}^{j} \right\}$$

$$+ \sum_{s=i}^{k-1} ps + pk + \sum_{s=k+1}^{j} ps$$

$$= \underset{i \le k \le j}{Min} \left\{ \sum_{s=i}^{k-1} ps \times level\ of\ a_s\ in\ T_i^{k-1} + \sum_{s=k+1}^{j} ps \times level\ of\ a_s\ in\ T_{k+1}^{j} \right\}$$

$$+ \sum_{s=i}^{j} ps$$

$$= \underset{i \le k \le j}{Min} \{ C[i, k-1] + C[k+1, j] \} + \sum_{s=i}^{j} ps$$

To summarize,

$$C[i,j] = \begin{cases} 0\ , & \text{if } i>j \text{ and } i-1=j \\ Pi\ , & \text{if } i = j \\ \underset{i \le k \le j}{Min} \{ C[i, k-1] + C[k+1, j] \} + \sum_{s=i}^{j} ps & \end{cases}$$

Step-3: Computing average number of search in optimal Binary search Tree:

Construct the Cost table and root table with following keys.

Keys:	A	B	C	D
Freq	4	2	6	3

C[1,0] = 0
C[2,1] = 0
C[3,2] = 0
C[4,3] = 0
C[5,4] = 0

	0	1	2	3	4
1	0	4	8	20	26
2		0	2	10	16
3			0	6	12
4				0	3
5					0

Cost Table

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

Root Table

For the number of keys = 1

C[1,1] = p1 = 4

C[2,2] = p2 = 2

C[3,3] = p3 = 6

C[4,4] = p4 = 3

For the number of keys = 2 (d=1)

C[1,2] = Min {k=1 , C[1,0] + C[2,2] + 4 + 2 = 0 + 2 + 6 = **8**

k=2 , C[1,1] + C[3,2] + 4 + 2 = 4 + 0 + 6 = 10

C[2,3] = Min {k=2 , C[2,1] + C[3,3] + 2 + 6 = 0 + 6 + 8 = 14

k=3 , C[2,2] + C[4,3] + 2 + 6 = 2 + 0 + 8 = **10**

C[3,4] = Min {k=3 , C[3,2] + C[4,4] + 6 + 3 = 0 + 3 + 9 = **12**

k=4 , C[3,3] + C[5,4] + 6 + 3 = 6 + 0 + 9 = 15

For the number of keys = 3 (d=2)

C[1,3] = Min {k =1 , C[1,0] + C[2,3] + 4 + 2 + 6 = 0 + 10+ 12 = 22

k = 2 , C[1,1] + C[3,3] + 4 + 2 + 6 = 4 + 6 + 12 = 22

k = 3 , C[1,2] + C[4,3] + 4 + 2 + 6 = 8 + 0 + 12 = **20**

C[2,4] = Min {k =2 , C[2,1] + C[3,4] + 2 + 6 + 3 = 0 + 12+ 11 = 23

k =3 , C[2,2] + C[4,4] + 2 + 6 + 3 = 2 + 3 + 11 = **16**

k =4 , C[2,3] + C[5,4] + 2 + 6 + 3 = 10 + 0 + 11 = 21

For the number of keys = 4 (d=3)

C[1,4] = Min {k =1 , C[1,0] + C[2,4] + 4 + 2 + 6 + 3 = 0 + 16+ 15 = 31

k = 2 , C[1,1] + C[3,4] + 4 + 2 + 6 + 3 = 4 + 12 + 15 = 31

k = 3 , C[1,2] + C[4,4] + 4 + 2 + 6 + 3 = 8 + 3 + 15 = **26**

k = 4 , C[1,3] + C[5,4] + 4 + 2 + 6 + 3 = 20 + 0 + 15 = 35

Algorithm: OptimalBST(P,n)

// Computes the cost table and the root table from the set of n keys, whose probability of searching is given in P.

```
for  i = 1 to n
   C[i , i-1] =0
   C[i , i ] = P[i]
   R[i , i] = i
C[n+1 , n] = 0
for d = 1 to n-1
   for i = 1 to  n-d
        j = i + d
        C[i , j] = ∞
        S = SUM(P,i,j)
        for  k = i to j
             q = C[i,k-1] + C[k+1, j] + S
             if q < C[ i , j ]
                  C[ i , j ] = q
                  R[ i , j ] = k
return C,R
```

```
SUM(P,i,j)
S = 0
for k = i to j
      S = S + P[k]
return S
```

Analysis: The basic operation is to compute q and find the minimum q value. To compute this O(1) time is required. The basic operation is called within 3-level nested loop. So the total running time is

T(n) = $\sum_{d=1}^{n-1} \sum_{i=1}^{n-d} \sum_{k=i}^{j} O(1) = O(n^3)$

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

Root Table

Step-4: Constructing optimal solution

Constructing optimal BST.

optimal BST can be obtained from Root table. The root ~~tal~~ is present in R[1,n]. Let the root of OBST is K. Then the root of its left subtree is present in R[1,K-1] and root of its Right subtree is in R[K+1, n]. ~~other~~ Root of other subtrees can be obtained by applying above rules recursively.

Alg:-

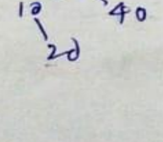
PRintOBST(Key, R, i, j, parent, type)

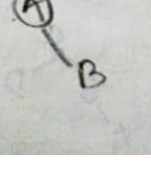
// R is the Root Table, i and j are the indices of keys to be used.
// Parent is the parent of root of a subtree. Type indicates the current
// key is left or Right child of its parent.
// Initially i=1, j=n, parent = null and type = null.

1. IF i = j, PRINT Key[i], "is" type "child of", Key[Parent]
2. IF i<j,
3. K = R[i,j]
4. IF type = null
5. PRINT Key[K], "is the Root"
6. ELSE, PRINT, Key[K], "is", type, "child of", Key[Parent]
Ø. // End If
7. PrintOBST (Key, R, i, K-1, K, "Left")
8. Print OBST (Key, R, K+1, j, K, "Right")
// End IF
9. Return.

o/p: 30 is the Root
10 is left child of 30
20 is ~~the~~ Right child of 10
40 is Right child of 30

C is Root
A is left child of C
B is the Right child of A
D is the Right child of C





Knapsack Problem: (0-1 knapsack| Discrete knapsack)

There are n objects and a knapsack of size M. Each object Xi is defined with weight Wi and Value Vi. The objective of knapsack problem is to obtain a filled knapsack that maximize the total profit.

In 0-1 knapsack problem, it is allowed either to take an item as a whole or not to take at all. i,e Xi is either 0 or 1.

Step-1: Structure of optimal Knapsack.

In dynamic programming, we need to show the optimal solution to an instance of knapsack in terms of optimal solution to its smaller sub instances.

Let an instance of first I items $1 \le I \le n$ with weight w1, w2, ….wi and values v1,v2,… vi and a capacity J , $1 \le J \le M$.

For this, there are two categories of sub instances/sub problems exist.

1. The subproblem that cannot include ith item. (If Wi > J)
2. The sub problem that can includes the ith item (if Wi ≤ J)

Step-2: Recursively define the value of optimal knapsack.

Let C[i,j] be the value of optimal knapsack that is the value of first i item fit into knapsack **capacity J.**

1. So for the first sub-problem that cannot include ith item, the value of the optimal solution is in C[i-1,j].
2. For the second sub problem, that can include ith item, we have two choice.
   a. Find optimal sub-instance without including ith item. i,e C[i-1, j]
   b. Find optimal sub-instance of first i-1 item in capacity (J-Wi) and adding Vi. I,e vi + C [I – 1 , J – wi]

   Which of these two solutions (a,b) is maximum is the solution of optimal value of first ith item.

So

$$C[i, j] = \begin{cases} 0 \quad , & \text{If } i = 0 \text{ or } J=0 \\ C[i-1, j] \quad , & \text{If } Wi > J \\ \text{MAX}(C[i-1, j], vi + C[i-1, J-Wi]) , & \text{If } Wi \leq J \end{cases}$$

Observation-→

	0	..	J-Wi	..	J	..	M
0							
:							
i-1			C[i-1, J-Wi]		C[i-1,j]		
i					C[I,j]		
:							
N							

Step-3: Compute the optimal knapsack.

Find the optimal filled knapsack for following instance

N = 4 , M = 5, Wi =(2,1,3,2) and V = (12,10,20,15)

Item	1	2	3	4
W	2	1	3	2
V	12	10	20	15

Solution:

C[11] = C[0,1]= 0 , as 2>1 ,

C[12] = Max(C[0,2],12+C[0,0]=12 , as 2<=2

C[13] = Max(C[0,3],12+C[0,1]=12 , as 2<=3

C[14] = Max(C[0,4],12+C[0,2]=12 , as 2<=4

C[15] = Max(C[0,5],12+C[0,3]=12 , as 2<=5

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0					
3	0					
4	0					

C[2,1] = Max(C[1,1] , 10+C[1,0]) = 10 , 1<=1

C[2,2] = Max(C[1,2] , 10+C[1,1]) = 12 , 1<=2

C[2,3] = Max(C[1,3] , 10+C[1,2]) = 22 , 1<=3

C[2,4] = Max(C[1,4] , 10+C[1,3]) = 22 , 1<=4

C[2,5] = Max(C[1,5] , 10+C[1,4]) = 22 , 1<=5

C[3,1] = C[2,1] = 10 , as 3>1

C[3,2] = C[2,2] = 12 , as 3>2

C[3,3] = Max(C[2,3] , 20+C[2,0]) = 22   , 3<=3

C[3,4] = Max(C[2,4] , 20+C[2,1]) = 30   , 3<=4

C[3,5] = Max(C[2,5] , 20+C[2,2]) = 32   , 3<=5

C[4,1] = C[3,1] = 10   ,   as 2>1

C[4,2] = Max(C[3,2], 15 + C[3,0]) = 15 , as 2<=2

C[4,3] = Max(C[3,3] , 15+ C[3,1]) = 25 , as 2<=3

C[4,4] = Max(C[3,4] , 15+ C[3,2]) = 30 , as 2<=4

C[4,5] = Max(C[3,5] , 15+ C[3,3]) = 37 , as 2<=5

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

```
Knapsack0_1(W,V,N,M)

  1. for i =0 to N
  2.      C[i,0 ] = 0
  3. for  j = 1 to M
  4.      C[0, j] = 0
  5. for i = 1 to N
  6.      for  j = 1 to M
  7.           if W[i] > j
  8.                C[i,j] = C[i-1, j]
  9.           else, C[i,j] = MAX ( C[i-1, j] , V[i] + C[i-1, j-W[i] ] )
  10.      return C
```

Analysis: Most of the time is invested in the loop 5 to 9 to fill the C[I,j]. This statement runs for j=1 to M and each j loop runs n times. So the algorithm runs in $\Theta(M \times N)$ times.

Step-4: Constructing optimal solution.

The optimal value in C[N,M] I,e C[4,5] = 37. So the optimal subset of items can be found by tracing back the entries in the table.

As C[4,5] ≠ C[3,5] , <u>**item 4 is added**</u> to the optimal knapsack.

Now remaining capacity is $J\text{-}W_4$ = 5-2 =3. So check C[3,3] I,e C[i-1, $j\text{-}W_4$]

As C[3,3] = C[2,3] so <u>**item 3 is not added**</u>.

As C[2,3] ≠ C[1,3] , So **item 2 is added**.

Now remaining capacity id J – W_2. = 3-1 =2 , So Check C[i-1, j-W_2] = C[1,2]

As C[1,2] ≠ C[0,2], so **item 1 is added**., check C[i-1,j-W_1] = C[0,0].

As I =0 and j=0 stop.

Now the solution is

Items	1 (1)	2 (1)	3 (0)	4 (1)	$\sum_{i=1}^{4} WiXi$	$\sum_{i=1}^{4} ViXi$
W	2	1	0	2	5	37
V	12	10	0	15		

Algorithm: PrintOptimalSubset(C,W,i,j)

//C is the cost table and W is the weight vector. i is the index of number of items in the sub-instance and J is the capacity of the knapsack. Initially i =N and J =M.

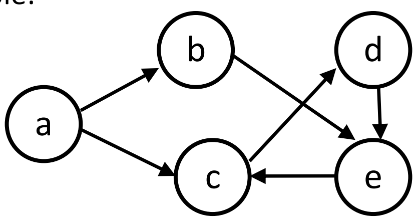
```
if  i =0 OR  j = 0
    return
if C[i,j] = C[i-1,j]
    PrintOptimalSubset(C,W,i-1,j)
else,
    PrintOptimalSubset(C,W, i-1, j-W[i])
    print "item", i
```

Analysis: the algorithm starts checking foe C[N,M] and at each step, N s decreased by 1 column wise and M is decreased by a weight until it reaches C[0,0]. So a total of Maximum of N+M time is required to reach (0,0). So the running time of this algorithm is O(N+M).

Warshall's Algorithm for Transitive Closure of a digraph

Transitive closure of a directed graph with n vertices is defined as nXn Boolean matrix T = t_{ij} = 1, if there exist a directed path from Vi to Vj .T_{ij}= 0 , if there is no such path exist between Vi to Vj.

Example:



$$T = \begin{array}{cc} & \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \\ e \end{array} & \begin{vmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{vmatrix} \end{array}$$

Warshall's Algrithm(named after S. Warshall) constructs transitive closure of the directed graph by dynamic programming.

Step-1: Structure of the problem

The solution T can be obtained through a series of nXn Boolean matrix $R^{(0)}, R^{(1)}, .. . R^{(n)}$,

where $R_{ij}^{(k)}$ is 1 , if there exist a directed path from Vi to Vj with intermediate vertex numbered, not higher than k. (<=k)

(Note : $R_{ij}^{(k)}$ *is the element in i-th row and j-th column of* $R^{(k)}$ *matrix)*

So $R^{(0)}$ has no intermediate vertex and is same as adjacent matrix.

and $R^{(n)}$ reflects path with all n vertices as intermediate vertices.

$R^{(k)}$ is computed from $R^{(k-1)}$

If $R_{ij}^{(k)}$ =1 means there exist a path from Vi to Vj with each intermediate vertex numbered not greater than k.

So the path is

Vi**,** **< A list of intermediate vertices numbered not greater than K>** , Vj

Step-2: Recursively define value of the optimal solution

There are two situations in the path.

1. List of intermediate vertices does not contain kth vertex.

Then the path from Vi to Vj has intermediate vertices not greater than K-1.

So $R_{ij}^{(k-1)} = 1$

2. List of intermediate vertices contains k^{th} vertex.

Then the path can be re-written as

Vi,**<intermediate vertices numbered not >K-1>,** Vk,**<intermediate vertices numbered not > K-1>**, Vj

This means there exist a path from Vi to Vk with intermediate vertices numbered not > k-1 ,

so $R_{ik}^{(k-1)} = 1$

AND there exist a path from Vk to Vj with intermediate vertices numbered not > k-1 ,

so $R_{kj}^{(k-1)} = 1$

From above two situations, it is observed that**,**

If $R_{ij}^{(k)} = 1$, Then either $R_{ij}^{(k-1)} = 1$ or both $R_{ik}^{(k-1)} = 1$ and $R_{kj}^{(k-1)} = 1$

So $R_{ij}^{(k)} = R_{ij}^{(k-1)}\ OR\ (R_{ik}^{(k-1)}\ AND\ R_{kj}^{(k-1)})$

Observation:

1. If $R_{ij} = 1$ in $R^{(k-1)}$, it remains same in $R^{(k)}$

2. If $R_{ij} = 0$ in $R^{(k-1)}$, it is changed to 1 , if $R_{ik} = 1$ in $R^{(k-1)}$ and $R_{kj} = 1$ in $R^{(k-1)}$

Step-3: Computing the value of transitive closure

Warshall(A, n)

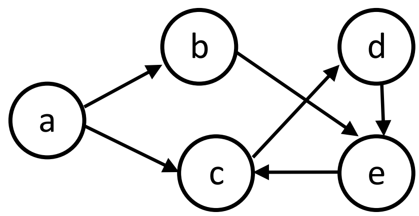
// A is the adjacent matrix of size nXn

1. $R^{(0)}$ = A
2. for k = 1 to n
3.   for i = 1 to n
4.   for j = 1 to n
5. $R^{(k)}$[i,j]=$R^{(k-1)}$[i,j]==1 OR ($R^{(k-1)}$[i,k] AND $R^{(k-1)}$[k,j]
6. return $R^{(n)}$

Analysis: The basic operation is in line number 5, which takes O(1) time. Total time =

$$\sum_{k=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{n} O(1) = O(n^3)$$

Example: Find transitive closure of the graph.



$$R^{(0)} = \begin{array}{c|ccccc|} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 1 & 0 \\ d & 0 & 0 & 0 & 0 & 1 \\ e & 0 & 0 & 1 & 0 & 0 \\ \hline \end{array} \qquad R^{(1)} = \begin{array}{c|ccccc|} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 1 & 0 \\ d & 0 & 0 & 0 & 0 & 1 \\ e & 0 & 0 & 1 & 0 & 0 \\ \hline \end{array}$$

$$R^{(2)} = \begin{array}{c|ccccc|} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 1 & 0 \\ d & 0 & 0 & 0 & 0 & 1 \\ e & 0 & 0 & 1 & 0 & 0 \\ \hline \end{array} \qquad R^{(3)} = \begin{array}{c|ccccc|} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 1 & 1 \\ b & 0 & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 1 & 0 \\ d & 0 & 0 & 0 & 0 & 1 \\ e & 0 & 0 & 1 & 1 & 0 \\ \hline \end{array}$$

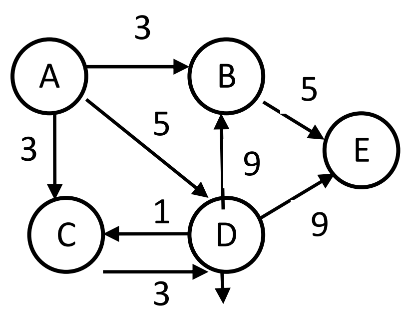
$$R^{(4)} = \begin{array}{c|ccccc|} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 1 & 1 \\ b & 0 & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 1 & 1 \\ d & 0 & 0 & 0 & 0 & 1 \\ e & 0 & 0 & 1 & 1 & 1 \\ \hline \end{array} \qquad R^{(5)} = \begin{array}{c|ccccc|} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 1 & 1 \\ b & 0 & 0 & 1 & 1 & 1 \\ C & 0 & 0 & 1 & 1 & 1 \\ d & 0 & 0 & 1 & 1 & 1 \\ e & 0 & 0 & 1 & 1 & 1 \\ \hline \end{array}$$

Floyd's Algorithm for All Pair Shortest Path Problem

All pair shortest path problem is to find the shortest path (distance) from each vertex to every other vertices of graph

The algorithm computes a series of Distance matrix $D^{(0),}$ $D^{(1)}$,. . . $D^{(n)}$

Example:



D =

	A	B	C	D	E
A	0	3	3	5	0
B	0	0	0	0	5
C	0	0	0	2	0
D	0	9	1	0	9
E	0	0	0	0	0

W =

	a	b	c	d	e
a	0	3	3	5	0
b	0	0	0	0	5
c	0	0	0	2	0
d	0	9	1	0	9
e	0	0	0	0	0

[Weighted Matrix

D =

	A	B	C	D	E
a	0	3	3	5	∞
b	∞	0	∞	∞	5
c	∞	∞	0	2	∞
d	∞	9	1	0	9
e	∞	∞	∞	∞	0

[Distance Matrix

Step-1: Structure of the problem

The problem has optimal substructure property as $D^{(k)}$ is computed from $^{D(k-1)}$.

$D^{(k)}$ is the distance matrix taking <1 …k> as intermediate vertices.

Let $D_{ij}^{(k)}$ is an element of $D^{(k)}$ (k=0 ,..n) ,then , it is the length of the shortest path from Vi to Vj with intermediate vertex numbered, not higher than k (ie, ≤k).

So D(0) has no intermediate vertex and is same as Weighted matrix,(0 changed to ∞)

and D(n) contains length of shortest path with all n vertices as intermediate vertices.

Let $D_{ij}^{(k)}$ is the cost of a shortest path from Vi to Vj with each intermediate vertex numbered not higher than k, then the path from Vi to Vj is

Vi, < A list of intermediate vertices numbered ≤ K> , Vj

Step-2: recursive solution

There are two situations in the path.

1. List of intermediate vertices does not contain kth vertex.

2. List of intermediate vertices contains kth vertex.

<u>For situation 1,</u>

The shortest path from Vi to Vj has intermediate vertices numbered 1 , …, K-1.

So $D_{ij}^{(k)} = D_{ij}^{(k-1)}$

<u>For situation 2,</u>

The path can be re-written as

Vi,<intermediate vertices numbered 1.. .K-1>, Vk,<intermediate vertices numbered 1,…, K-1>, Vj

This means there exist a shortest path from Vi to Vk with intermediate vertices numbered 1,…, K-1, whose length is $D_{ik}^{(k-1)}$

AND a shortest path from Vk to Vj with intermediate vertices numbered 1,…, K-1, whose length is $D_{kj}^{(k-1)}$

From above two situations, length of shortest path from Vi to Vj with each intermediate verticex <= k is ,

$$D_{ij}^{(k-1)} = Min(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)})$$

Step-3: Computing All pair shortest path

Algorithm:

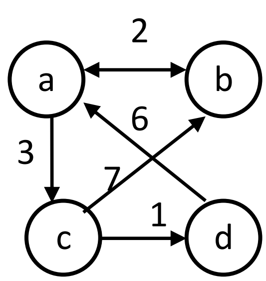
Floyd(W, n)

```
// A is the adjacent matrix of size nXn
1.  D^(0) = W
2.  for  k = 1 to n
3.   for  i = 1 to n
4.        for j = 1 to n
5.            D^(k)[i,j]=Min(D^(k-1)[i,j],D^(k-1)[i,k]+D^(k-1)[k,j])
6.  return R^(n)
```

Analysis: The basic operation is in line number 5, which takes O(1) time.

Total time = $\sum_{k=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{n} O(1) = O(n^3)$

Example: Find transitive closure of the graph.



$$W = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ D \end{array} & \begin{bmatrix} 0 & 0 & 3 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 7 & 0 & 1 \\ 6 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$D^{(0)} = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ \infty & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$$P^{(0)} = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} \text{Nil} & \text{Nil} & 1 & \text{Nil} \\ \text{b} & \text{Nil} & 0 & \text{Nil} \\ \text{Nil} & \text{c} & \text{Nil} & \text{c} \\ \text{D} & \text{Nil} & \text{Nil} & \text{Nil} \end{bmatrix} \end{array}$$

$$D^{(1)} = \begin{array}{c|cccc|} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & \infty & 0 & 5 & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \\ \hline \end{array} \qquad P^{(1)} = \begin{array}{c|cccc|} & a & b & c & d \\ \hline a & \text{Nil} & \text{Nil} & 1 & \text{Nil} \\ b & b & \text{Nil} & a & \text{Nil} \\ c & \text{Nil} & c & \text{Nil} & c \\ d & D & \text{Nil} & a & \text{Nil} \\ \hline \end{array}$$

$$D^{(2)} = \begin{array}{c|cccc|} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & \infty & 0 & 5 & \infty \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \\ \hline \end{array} \qquad P^{(2)} = \begin{array}{c|cccc|} & a & b & c & d \\ \hline a & \text{Nil} & \text{Nil} & 1 & \text{Nil} \\ b & b & \text{Nil} & a & \text{Nil} \\ c & b & c & \text{Nil} & c \\ d & D & \text{Nil} & a & \text{Nil} \\ \hline \end{array}$$

$$D^{(3)} = \begin{array}{c|cccc|} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & \infty & 0 & 5 & 6 \\ c & 9 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \\ \hline \end{array} \qquad P^{(3)} = \begin{array}{c|cccc|} & a & b & c & d \\ \hline a & \text{Nil} & C & 1 & C \\ b & b & \text{Nil} & a & C \\ c & b & c & \text{Nil} & C \\ d & D & c & a & \text{Nil} \\ \hline \end{array}$$

$$D^{(4)} = \begin{array}{c|cccc|} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & \infty & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \\ \hline \end{array} \qquad P^{(4)} = \begin{array}{c|cccc|} & a & b & c & d \\ \hline a & \text{Nil} & C & 1 & C \\ b & b & \text{Nil} & a & C \\ c & d & c & \text{Nil} & C \\ d & D & c & a & \text{Nil} \\ \hline \end{array}$$

Maximum flow problem

A flow network is a network of conduits and junctions, where each conduit has a maximum capacity to transmit the materials. The maximum flow problem is to calculate the maximum low rate of materials from a given source to a given sink.

Representation:

- A flow network is a connected directed graph G= (V,E) in which each edge (u,v) ϵE has a capacity C(u,v) ≥ 0
- If (u,v) ∉ E, then C(u,v) = 0
- Every vertex lies on some path from the source to sink

Flow: Let G be a flow network with capacity function C, a flow in G is a function satisfy following constraints.

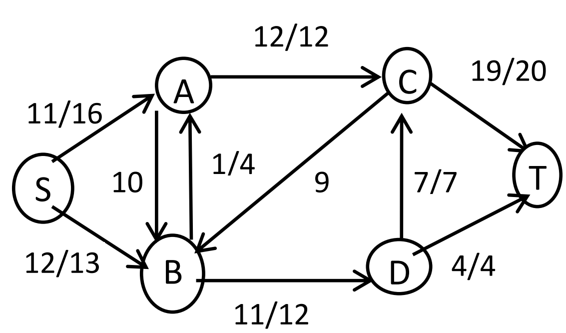
1. Capacity constraints: For all u,v ϵ V , f(u,v) ≤ C(u,v)
2. Skew Symmetry : For all u,v ϵ V , f(u,v) = - f(v,u)
3. Flow conservation : For all u ϵ V-{s,t} , $\sum_{v \in V} f(u,v) = 0$

The quantity f(u,v) is called flow from u to v , which may be +ve, -ve or zero. The value of flow

$$|f| = \sum_{v \in V} f(s,v)$$

Maximum flow problem is defined as :----

Given flow network G with source 's' and sink 't', we wish to find a flow of maximum value.



In this network, the total flow = f(S,A) + f(S,B) = 11 + 12 = 23.

Ford-Fulkerson method

Ford-Fulkerson method is based on following ideas.

- Residual network
- Augmenting path
- Cut

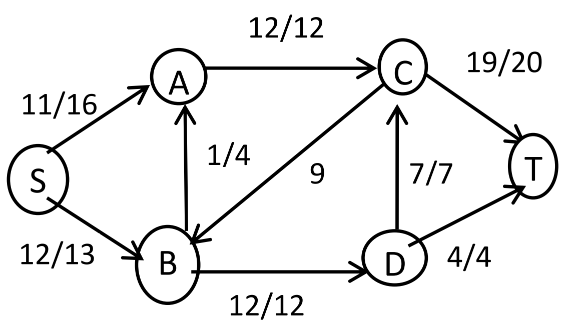
Residual Network(G_f)

Given a flow network G and a flow, the residual network consists of the edges that can admit more flows.

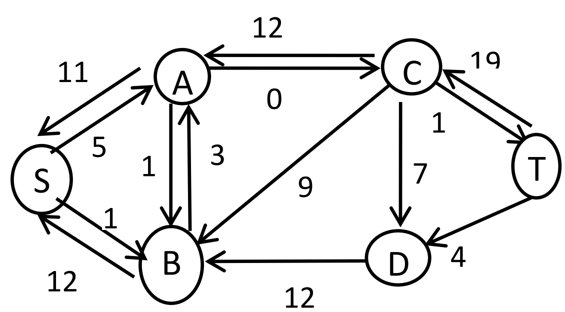
Residual capacity Cf (u,v) of an edge (u,v) is the additional flows that can be pushed from u to v.

$C_f = C(u,v) - f(u,v)$

- The edge of G that are in G_f can admit more flows.
- The G_f may contain edges that are not originally in G.



Flow network (G)



Residual network (Gf)

Augmenting path (P)

Augmenting path is a simple path from s to t in residual network. Each edge in augmenting path admits additional flows from u to v without violating the capacity constraints on that edge.

Ford Fulkerson method works as follows.

1. Initialize flow f = 0.
2. WHILE there is an augmenting path P.
3. Augment flow f along path P.
4. Reutrn f.

Algorithm: Ford_Fulkerson(G,s,t)

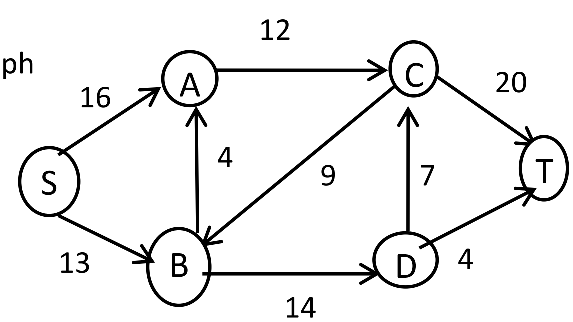
```
//

1. for each (u,v)∈ E[G]
2.      f[u,v] = 0
3.      f[v,u] = 0
4. While there exist a path(P) from s to t in residual network
5.      Cf(P) = Min{Cf(u,v) : (u,v)  in P }
6.      for each (u,v) ∈ p
7.              f[u,v] =  f[u,v] + Cf(P)
8.              f[v,u] = -f[u,v]
8. return f
```

Analysis: For efficient result, augmenting path must be chosen by running BFS algorithm, which runs O (V,E) times.

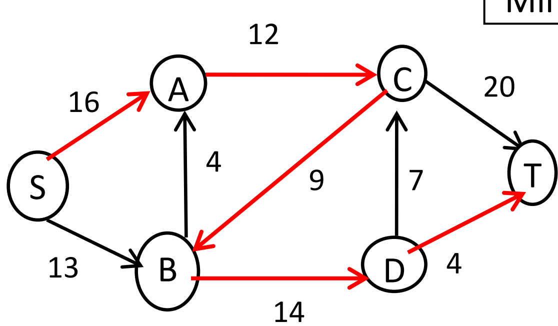
Line -1 to 3 runs in O (V) times. While loop in line 4 to 8 is executed at most |f*| times, where f* is the maximum flow found by the algorithm. So total running time is O(E|f*|) times.

Example: Find maximum flow in following graph

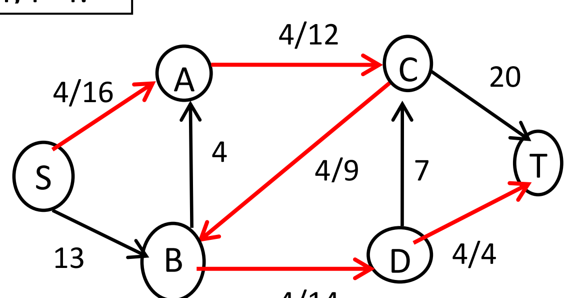


Step-1: Choose path <S,A,C,B,D,T>

Min = 4, f=4.

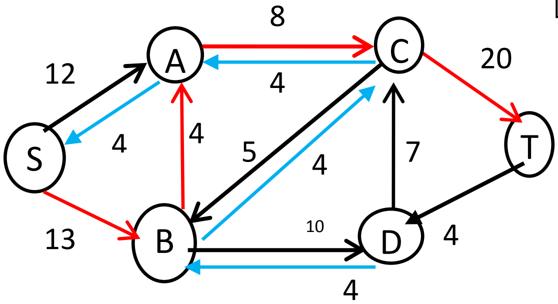


[Residual n/w]

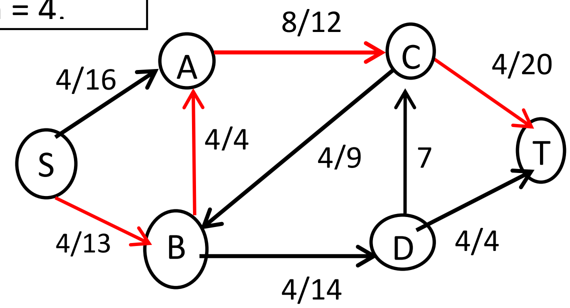


[Flow n/w]

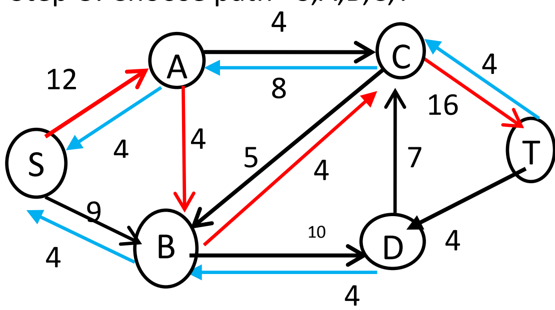
Step-2: Choose path <S,B,A,C,T>



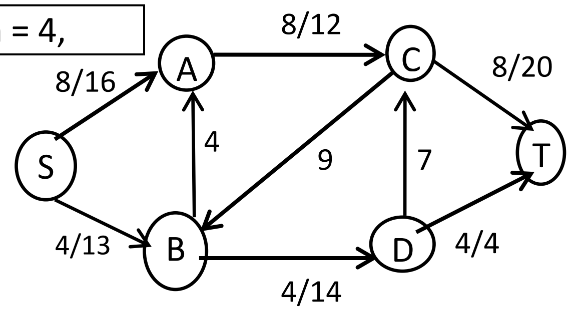
Min = 4.



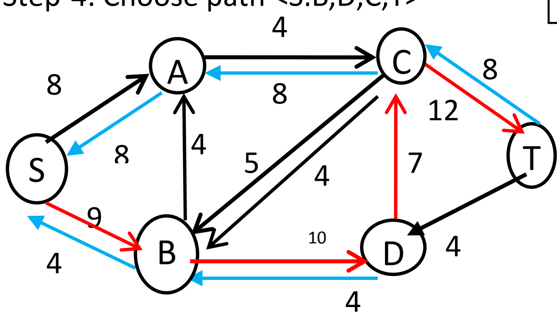
Step-3: Choose path <S,A,B,C,T>



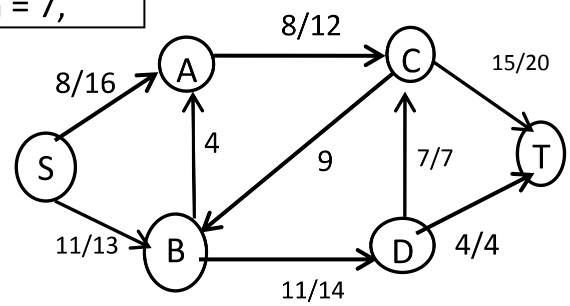
Min = 4,



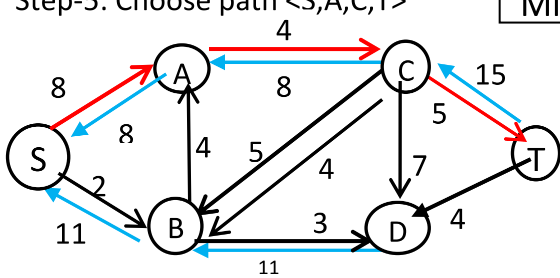
Step-4: Choose path <S.B,D,C,T>



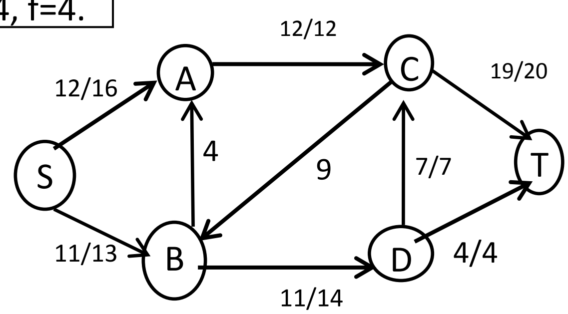
Min = 7,

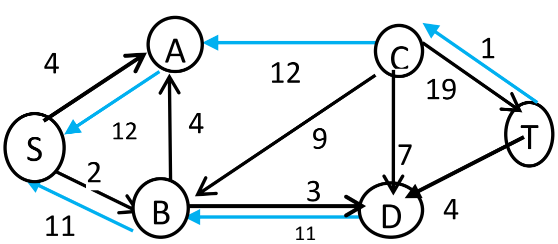


Step-5: Choose path <S,A,C,T>



Min = 4, f=4.



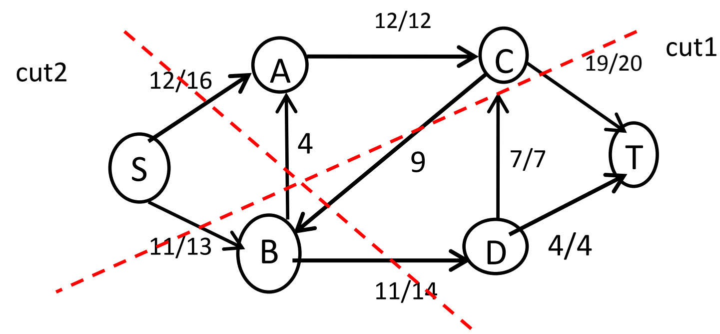


[Final Flow n/w]

No more flow can be admitted in any path from S to T

Cut :

A cut (S,T) of a flow network is a partition of V into S and T = V-S, such that s ϵ S and t ϵ T. If f is a flow, net flow across the cut(S,T) is defined to be f(S,T) , The net flow across any cut is same.



Cut-1 consist of S = {S,A,C} and T ={B,D,T}, the flow value of the cut1=

= f(S,B) + F(A,B) + f(C,B) + f(C,D) + f(C,T)

= 11 + 0 + 0 + -7 + 19 = 23

Cut-2 consist of S = {S,B} and T ={A,C,D,T}, the flow value of the cut1=

= f(S,A) + F(B,A) + f(B,C) + f(B,D)

= 12 + 0 + 0 + 11 = 23