Unit-III

UNIT III : Greedy method: General method, applications-Job sequencing with dead lines, 0/1 knapsack problem, Minimum cost spanning trees, Single source shortest path problem.
Applications: Allocation of funds/resources based on the priority in the computer systems.

Greedy Method

- Greedy algorithm obtains an optimal solution by making a sequence of decisions.
- Decisions are made one by one in some order.
- Each decision is made using a greedy-choice property or greedy criterion.
- A decision, once made, is (usually) not changed later.

- It gives an optimal solution when applied to problems with the **greedy-choice** property.
- A feasible solution is a solution that satisfies the constraints.
- An optimal solution is a feasible solution that optimizes the objective function.

Greedy method control abstraction/ general method

```
Algorithm  Greedy(a,n)
// a[1:n] contains the n inputs
{
    solution= //Initialize solution
    for i=1 to n do
    {
        x:=Select(a);
        if  Feasible(solution,x) then
                solution=Union(solution,x)
    }
return solution;
}
```

Example: Largest k-out-of-n Sum

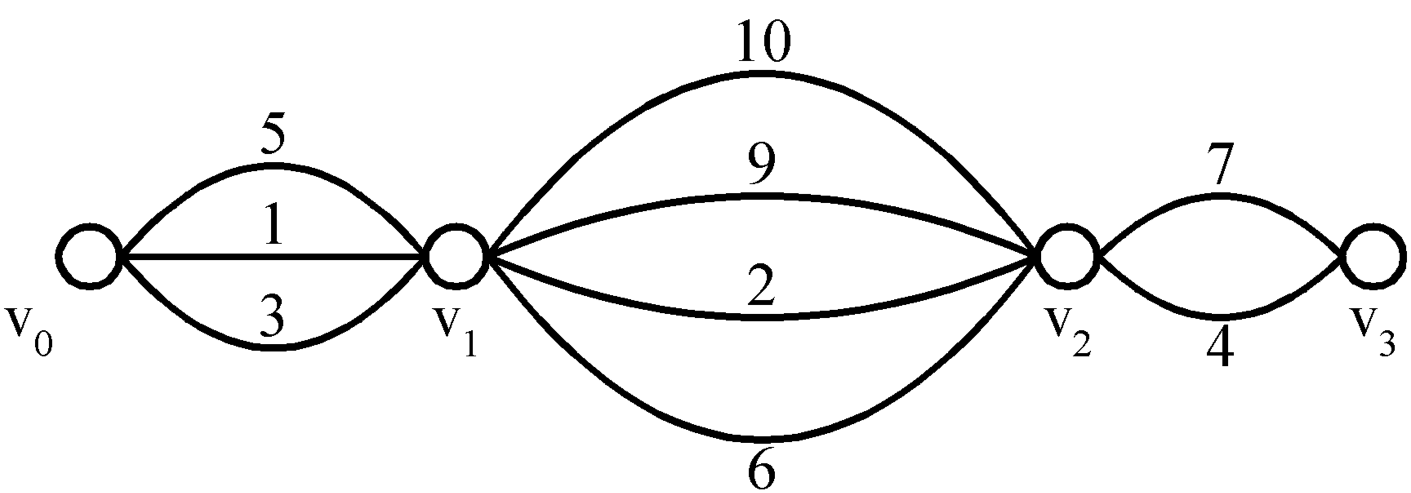
- Problem
  - Pick k numbers out of n numbers such that the sum of these k numbers is the largest.
- Exhaustive solution
  - There are C_k^n choices.
  - Choose the one with subset sum being the largest
- Greedy Solution

Is the greedy solution always optimal?

```
FOR i = 1 to k
    pick out the largest number and
    delete this number from the
input.
ENDFOR
```

Example:
Shortest Path on a Special Graph

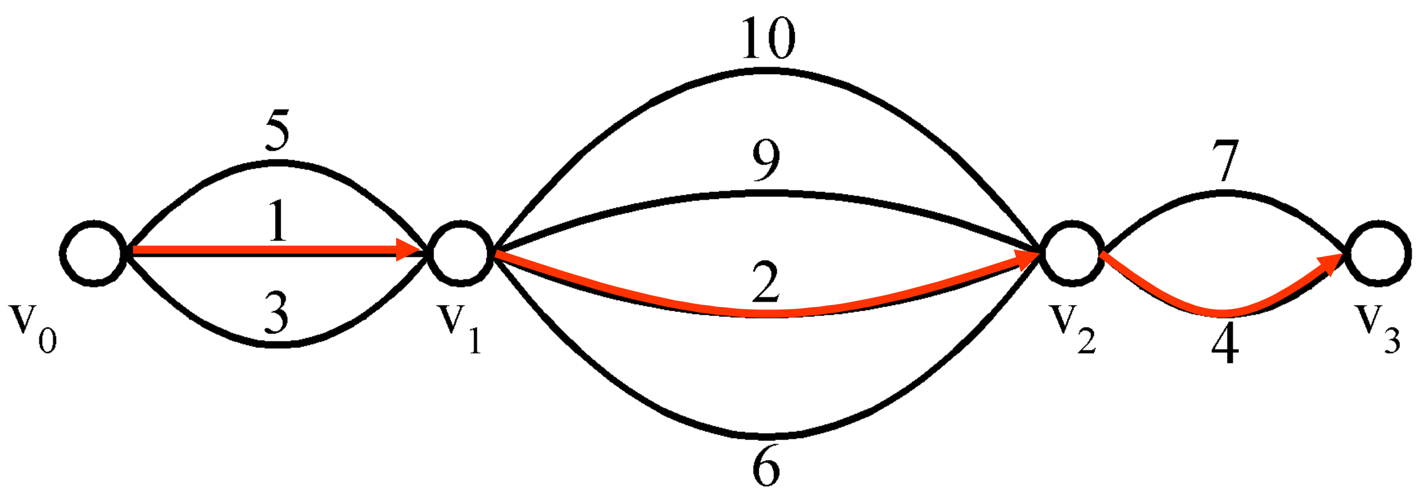
- Problem
  - Find a shortest path from v_0 to v_3
- Greedy Solution



Example:
Shortest Paths on a Special Graph

- Problem
  - Find a shortest path from v_0 to v_3
- Greedy Solution

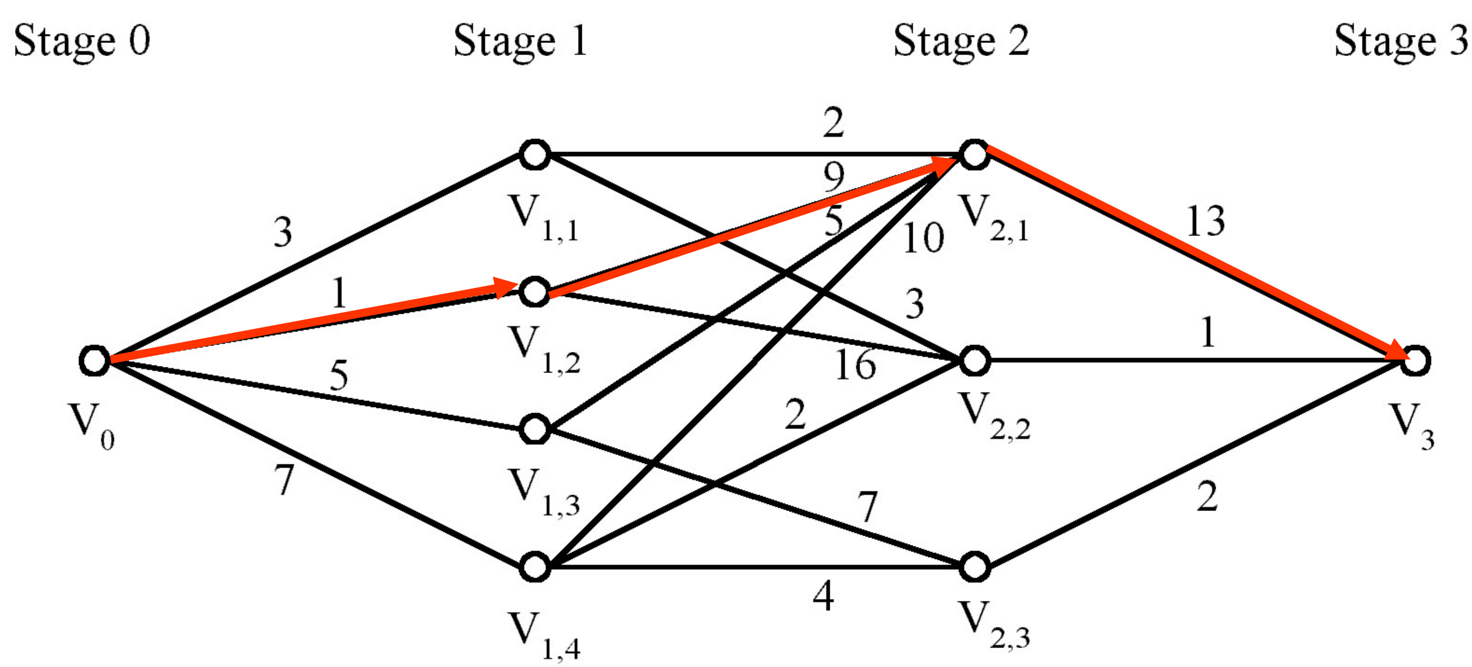
Is the solution optimal?



Example:
Shortest Paths on a Multi-stage Graph

Is the greedy solution optimal?

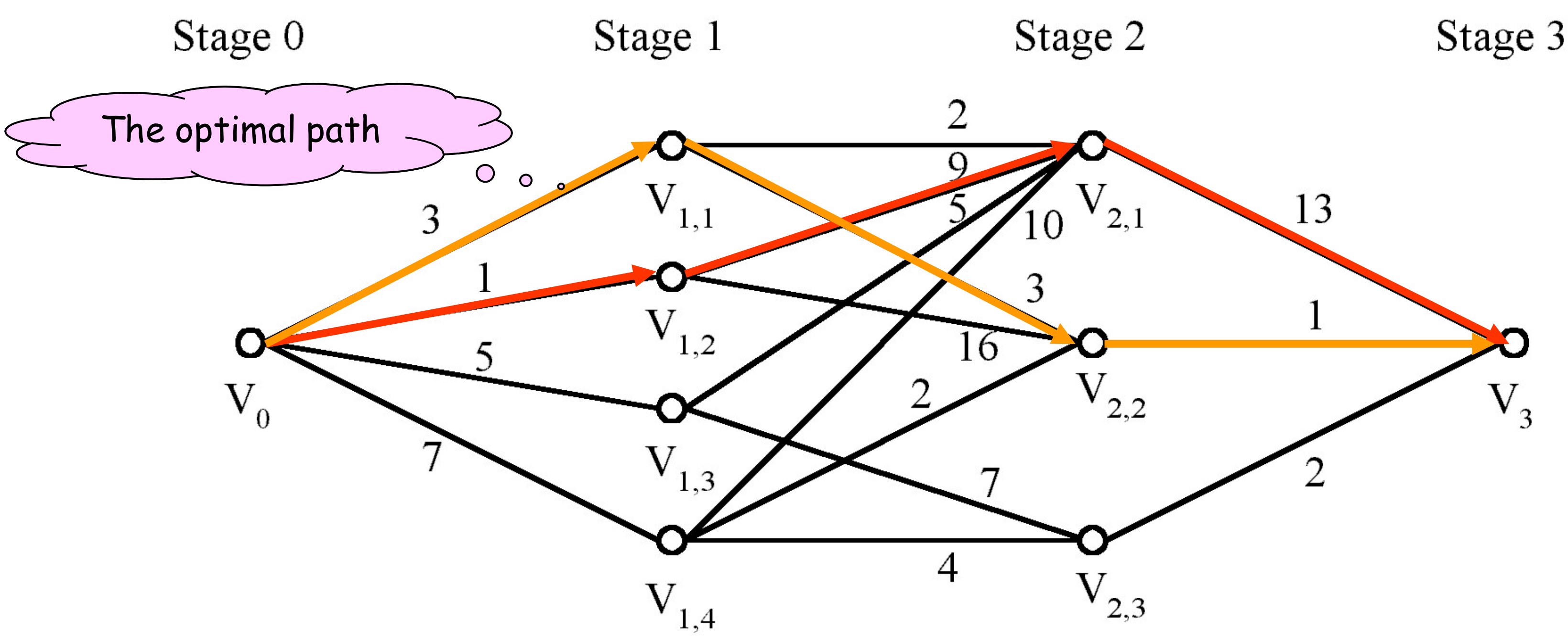
- Problem
  - Find a shortest path from v_0 to v_3



Example:
Shortest Paths on a Multi-stage Graph

Is the greedy solution *optimal?*

- Problem
  - Find a shortest path from v_0 to v_3

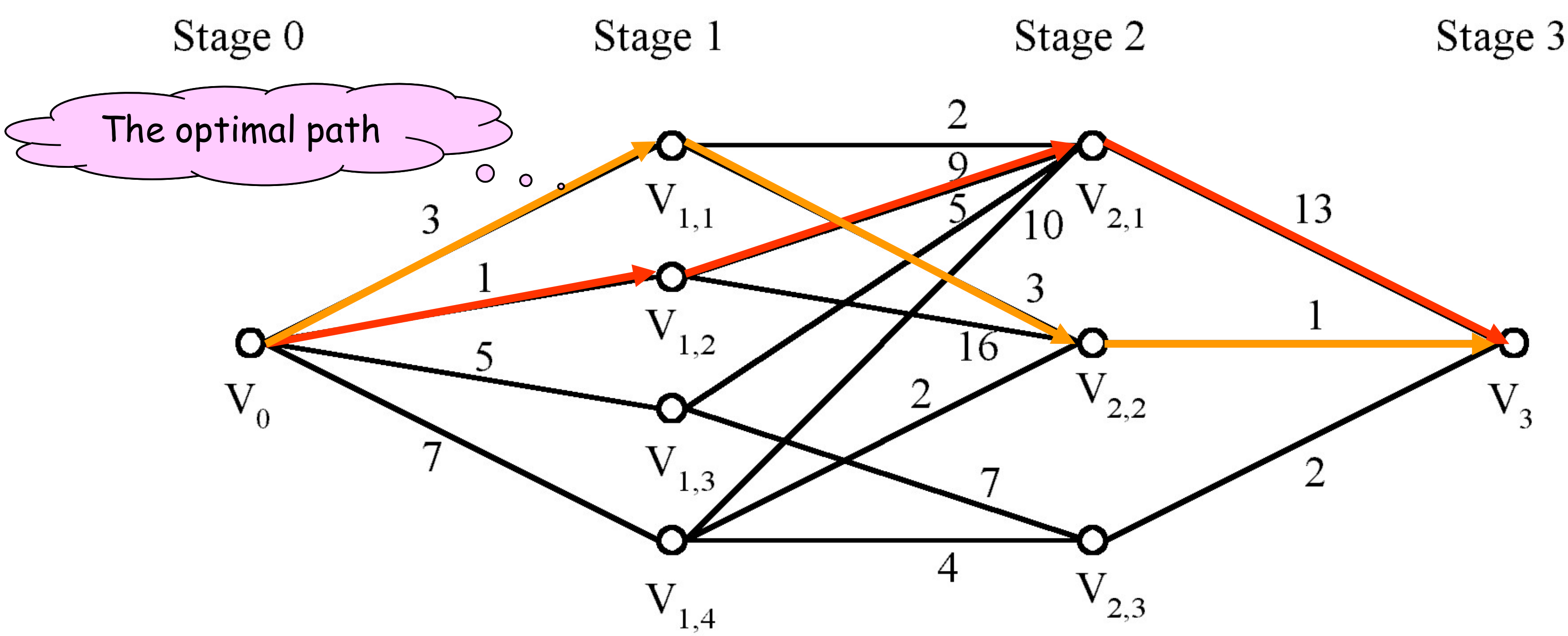


Example:
Shortest Paths on a Multi-stage Graph

Is the greedy solution ...? ✘

What algorithm can be used to find the optimum?

- Pr...
  - ...nd a shortest path from v_0 to v_3



The Fractional Knapsack Problem

- Given: A set S of n items, with each item i having
  - p_i - a positive profit
  - w_i - a positive weight
- Goal: Choose items, allowing fractional amounts(x_i), to maximize total profit but with weight at most m.

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subjected to} \sum_{1 \le i \le n} w_i x_i \le m$$

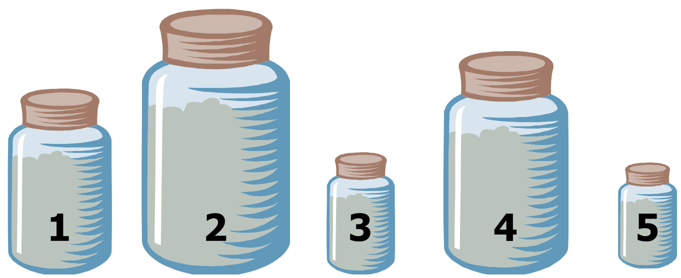
$$\text{and } 0 \le x_i \le 1, \qquad 1 \le i \le n$$

The Fractional Knapsack Problem

Greedy decision property:-

Select items in decreasing order of profit/weight.

Items:



w_i :	4 ml	8 ml	2 ml	6 ml	1 ml
p_i :	\$12	\$32	\$40	\$30	\$50
Value: (\$ per ml)	3	4	20	5	50

10 ml

"knapsack"

Solution:

- 1 ml of i_5
- 2 ml of i_3
- 6 ml of i_4
- 1 ml of i_2

- Solution vector

  $(x_1,x_2,x_3,x_4,x_5)= (0,1/8,1,1,1)$

- Profit $=12*0 + 32*1/8 + 40*1 + 30*1 + 50*1$

  $= 0+4+40+30+50$

  $=124.$

Greedy algorithm for the fractional Knapsack problem

```
Algorithm GreedyKnapsack(m,n)
//P[1:n]  and w[1:n] contain the profits and weights
// respectively of the n objects ordered such that
//p[i]/w[i]>=p[i+1]/w[i+1].
//m is the knapsack size and x[1:n] is the solution
// Vector.
{
        for i=1 to n do  x[i]=0;   // Initialize x.
        U=m;
        for i=1 to n do
        {
                if ( w[i]>U ) then break;
                if x[i]=1;  U=U-w[i];
        }
        if ( i <=n) then x[i]= U/w[i];
}
```

If you do not consider the time to sort the items, then the time taken by the above algorithm is O(n).

0/1 Knapsack Problem

- An item is either included or not included into the knapsack.

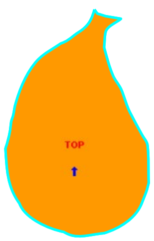
Formally the problem can be stated as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subjected to} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and } x_i = 0 \text{ or } 1, \quad 1 \le i \le n$$

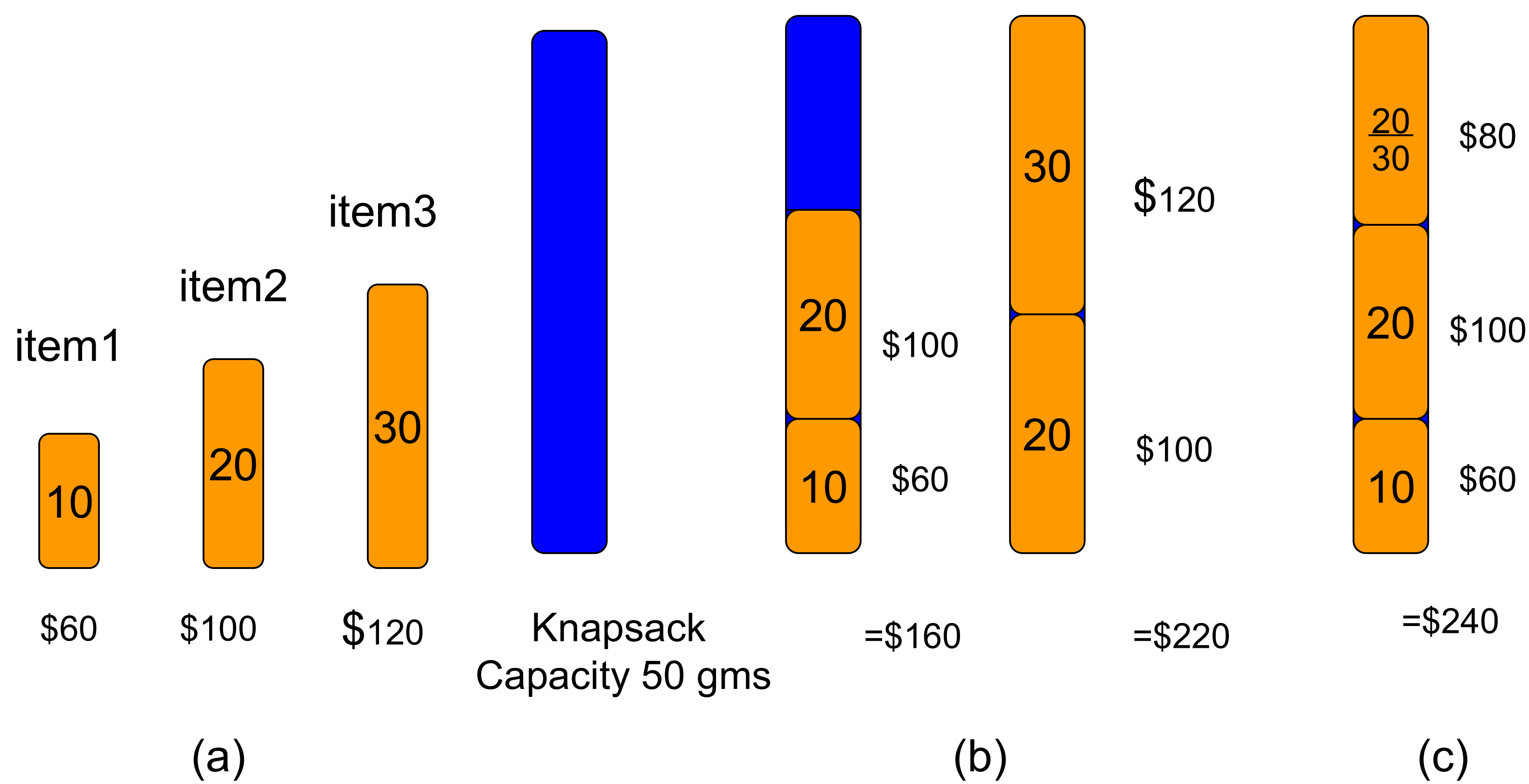
Which items should be chosen to maximize the amount of money while still keeping the overall weight under m kg ?



m

Is the fractional knapsack algorithm applicable?

- The greedy method works for fractional knapsack problem, but it does not for 0/1 knapsack problem.
- Ex:-



- There are 3 items, the knapsack can hold 50 gms.
- The value per gram of item 1 is 6, which is greater than the value per gram of either item2 or item3.

- The *greedy approach* (Decreasing order of profit's/weight's), does not give an optimal solution.
- As we can see from the above fig., the *optimal solution* takes item2 and item3.
- For the *fractional* problem, the *greedy approach* (Decreasing order of profit's/weight's) gives an *optimal solution* as shown in fig c.

Spanning Tree

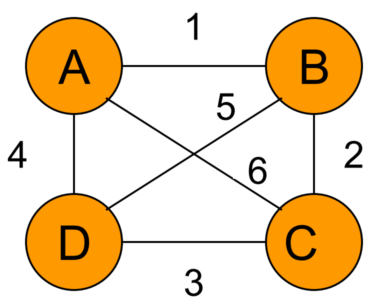
- A *tree* is a connected undirected graph that contains no cycles.

- A *spanning tree* of a graph G is a subgraph of G that is a tree and contains all the vertices of G.

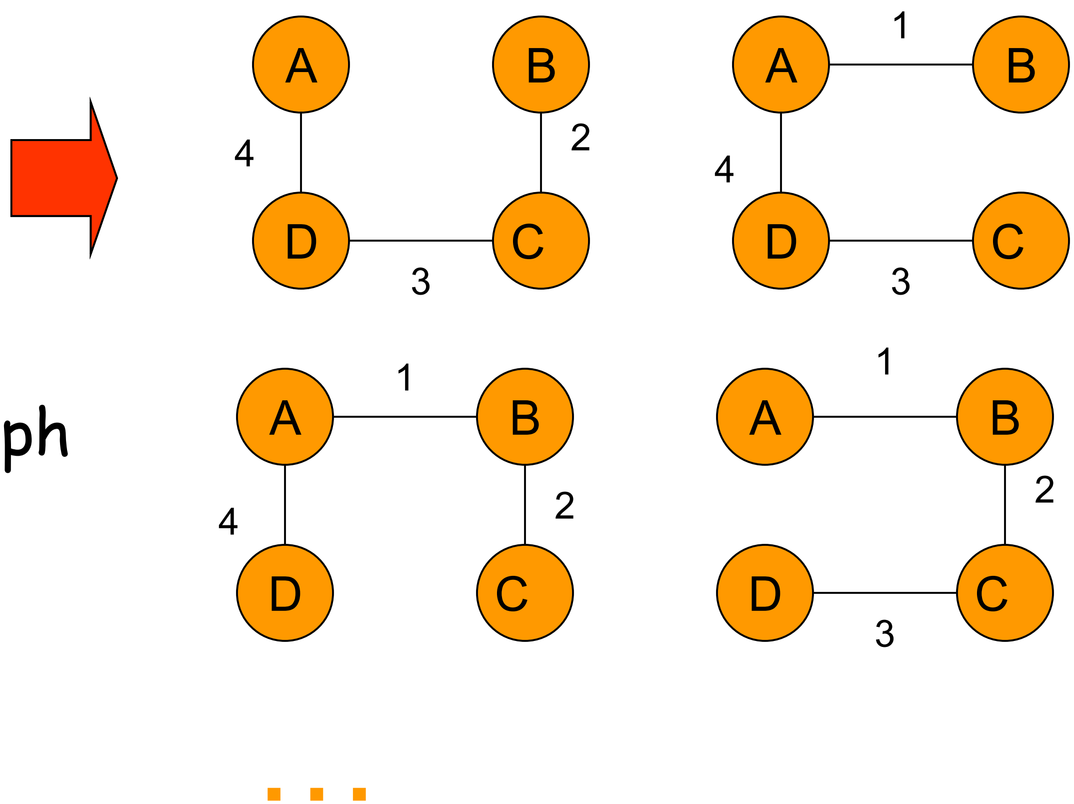
Properties of a Spanning Tree

- The spanning tree of a n-vertex undirected graph has exactly $n - 1$ edges.
- It connects all the vertices in the graph.
- A spanning tree has no cycles.

Ex:-



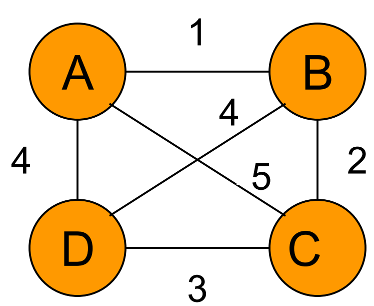
Undirected Graph



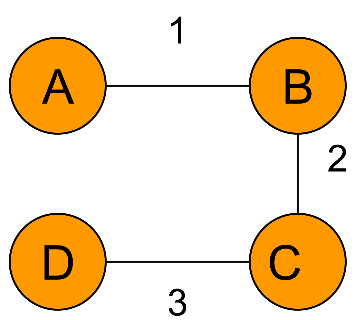
Some Spanning Trees

Minimum Cost Spanning Tree / Minimum Spanning Tree (MST)

- A *minimum spanning tree* is the one among all the spanning trees with the *smallest total cost.*



Undirected Graph



Minimum Spanning Tree

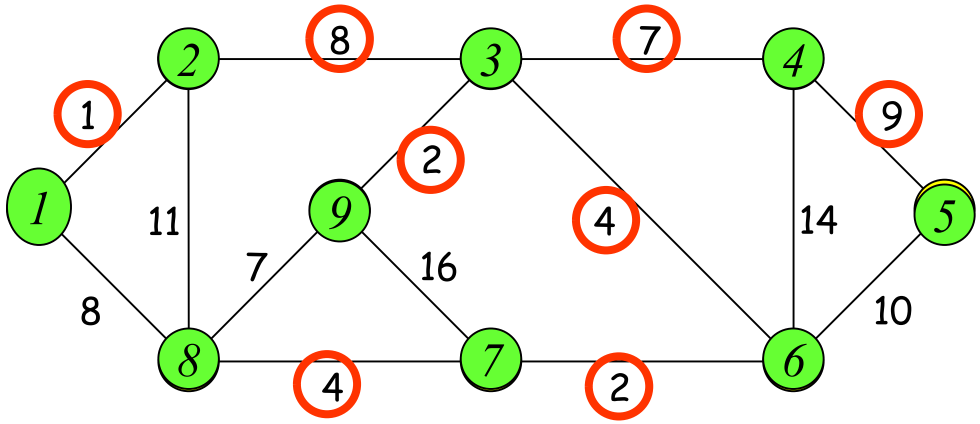
Applications of MSTs

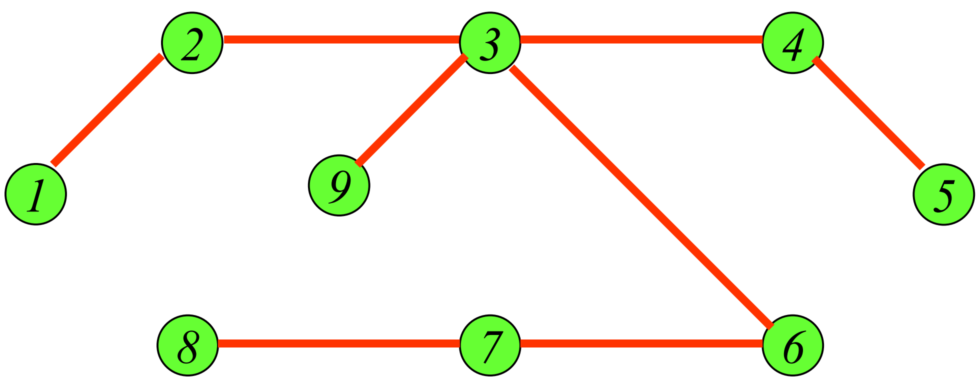
- Computer Networks
  - To find how to connect a set of computers using the minimum amount of wire.

MST-Prim's Algorithm

- Start with minimum cost edge.
- Select next minimum cost edge (i,j) such that *i* is a vertex already included in the tree, *j* is a vertex not yet included.
- Continue this process until the tree has n - 1 edges.

Prim's Algorithm



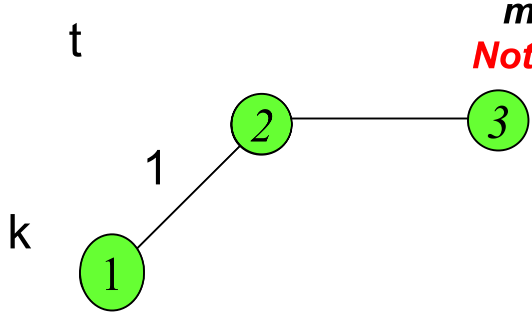


t [1:n-1,1:2]

	1	2
1	1	2
2	2	3
	3	9
.	3	6
.	6	7
.	7	8
	3	4
n-1	4	5

1. near[j] is a vertex in the tree such that cost [j,near[j]] is minimum among all choices for near[j].

Note: ***We define near[j] = 0 for all vertices j that are already in the tree.***



1	near[1]=0
2	near[2]=0
3	near[3]=0

	j	cost[j, near[j]]
3	near[3]=2	cost[3,2]=8
4	near[4]=1	cost[4,1]= ∞
5	near[5]=1	cost[5,1]= ∞
6	near[6]=1	cost[6,1]= ∞
7	near[7]=1	cost[7,1]= ∞
8	near[8]=1	cost[8,1]=8
9	near[9]=1	cost[9,1]= ∞

2. Select next min cost edge.

3. Next update near[j] for all vertices which are not yet included in the tree and then go to step 2.

4. Continue this procedure until tree contains n-1 edges.

Prim’s Algorithm

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G.
//cost[1:n,1:n] is the cost matrix such that cost[i,j] is either
// positive real number  or ∞ if no edge (i,j) exists.  cost[i,j]=0, if i=j.
// A minimum spanning tree is computed and stored
// as a set of edges in the array t[1:n-1,1:2]
{
    Let (k,l) be an edge of minimum cost in E

    mincost=cost[k,l];
    t[1,1]=k; t[1,2]=l;
    near[k]=near[l]=0;

        for i=1 to n do    // initialize  near
            if( near[i] ≠0 ) then
                if(  cost[i,k]< cost[i, l]  then near[i]=k;
                else  near[i]= l;
```

```
for i=2 to n-1 do
      {
            // Find n-2 additional edges for t.

                Let j be an index such that near[j]≠0 and
                cost[j,near[j]] is minimum;

                t[i,1]=j; t[i,2]=near[j];
                mincost=mincost+cost[j,near[j]];
                near[j]=0;

                for k=1 to n do  // update near[]
                if( ( near[k] ≠0 ) and (cost[k,near[k]>cost[k,j]))  then
                          near[k]=j;
      }
return mincost;
}
```

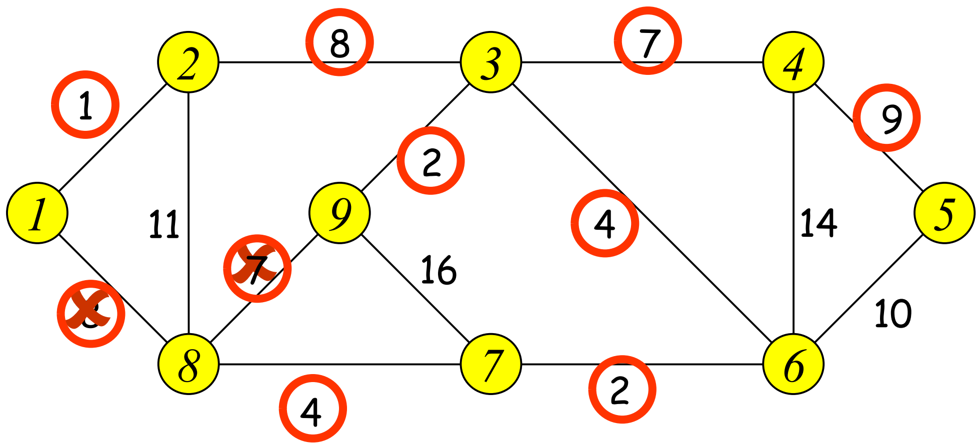
Time complexity of Prims algorithm

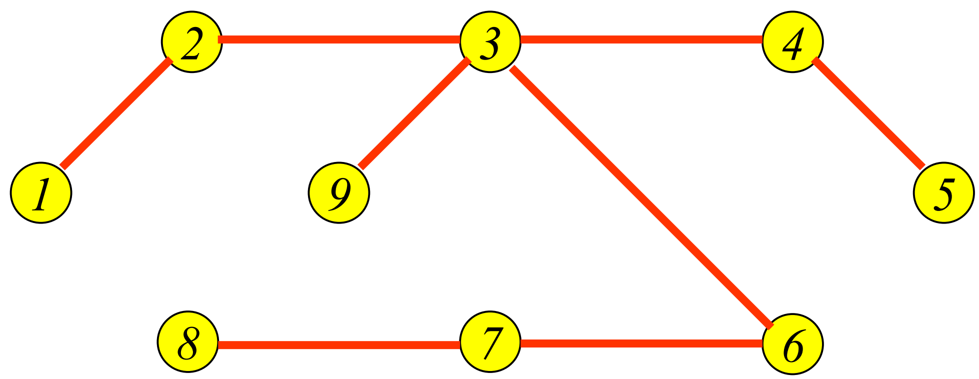
- Line 8 takes o(E).
- The for loop of line 12 takes o(n).
- 17 and 18 and the for loop of line 22 require o(n) time.
- Each iteration of the for loop of line 14 takes o(n) time.
- Therefore, the total time for the for loop of line 14 is $o(n^2)$.
- Hence, time complexity of Prim is $o(n^2)$.

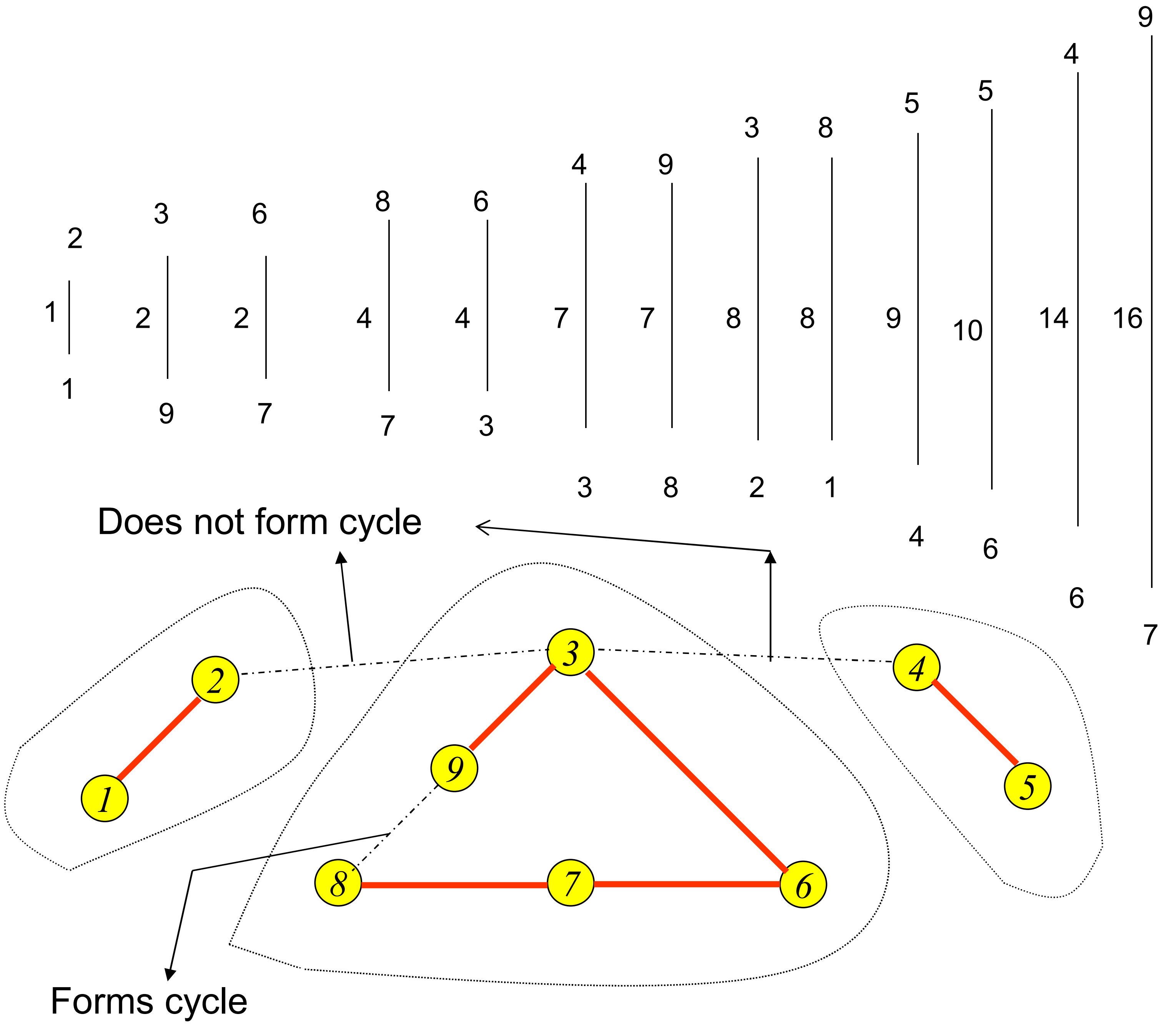
Kruskal's Method

- Start with a forest that has no edges.
- Add the next minimum cost edge to the forest if it will not cause a cycle.
- Continue this process until the tree has n - 1 edges.

Kruskal's Algorithm






Does not form cycle
Forms cycle

Disjoint Sets

- Two sets A and B are said to be **disjoint** if there are no common elements i.e., $A \cap B = \varnothing$.
- *Example:*

  1) $S_1=\{1,7,8,9\}$, $S_2=\{2,5,10\}$, and $S_3=\{3,4,6\}$.

     are three disjoint sets.

- We identify a set by choosing a ***representative element*** of the set. It doesn't matter which element we choose, but once chosen, it can't be changed.

- Disjoint set operations:
  - FIND-SET(x): Returns the representative of the set containing x.
  - UNION(i,j): Combines the two sets i and j into one new set. A new representative is selected.

    (Here i and j are the representatives of the sets)

Disjoint Sets Example

- Make-Set(1)
- Make-Set(2)
- Make-Set(3)
- Make-Set(4)
- Union(1,2)
- Union(3,4)
- Find-Set(2)   returns 1
- Find-Set(4)   returns 3
- Union(2,4)

MST-Kruskal's Algorithm

Algorithm kruskal(E,cost,n,t)

// E is the set of edges in G.

//cost[1:n,1:n] is the cost matrix such that cost[i,j] is either

// positive real number or ∞ if no edge (i,j) exists. cost[i,j]=0, if i=j.

// A minimum spanning tree is computed and stored

// as a set of edges in the array t[1:n-1,1:2].

```
{
        for i:=1 to n do
         Make-Set(i); // each vertex is in a different set.
         Sort the edges of E into increasing order by cost.
```

```
mincost:=0; i:=1;
for each edge (u,v) € E, taken in increasing order by cost do
{    j:= Find-Set(u); k:= Find-Set(v)
     if( j≠k ) then
     {
            t[i,1]:=u; t[i,2]:=v;
            mincost:=mincost+cost[u,v];
            Union(j,k);
             i:=i+1;
     }
  return mincost;
}
```

Time complexity of kruskal's algorithm

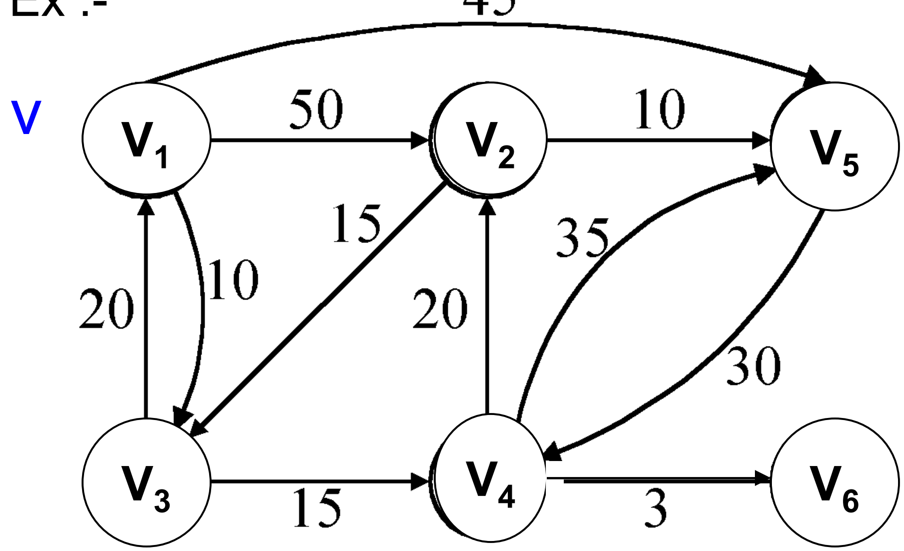
- With an efficient Find-set and union algorithms, the running time of kruskal's algorithm will be dominated by the time needed for sorting the edge costs of a given graph.
- Hence, with an efficient sorting algorithm(merge sort), the complexity of kruskal's algorithm is

  o(ElogE).

The Single-Source Shortest path Problem (SSSP)

- Given a positively weighted directed graph G with a source vertex v, find the shortest paths from v to all other vertices in the graph.

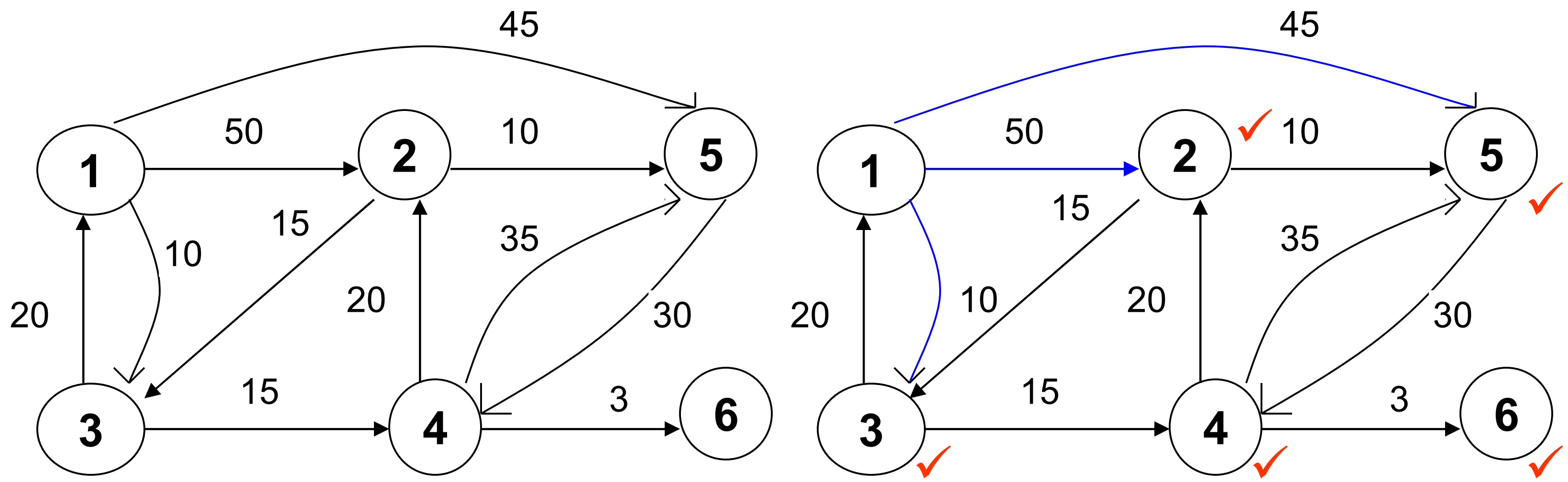
Ex :-



(a)

	Path	Length
1)	V_1V_3	10
2)	$V_1V_3V_4$	25
3)	$V_1V_3V_4V_2$	45
4)	V_1V_5	45
5)	$V_1V_3V_4V_6$	**28**

(b)



Iteration	S	Dist[2]	Dist[3]	Dist[4]	Dist[5]	Dist[6]
Initial	{1}	50	10	∞	45	∞
1	{ 1,3 }	50	10	25	45	∞
2	{ 1,3,4 }	45	10	25	45	28
3	{ 1,3,4,6 }	45	10	25	45	28
4	{ 1,3,4,5,6 }	45	10	25	45	28

SSSP-Dijkstra's algorithm

- Dijkstra's algorithm assumes that cost$(e) \geq 0$ for each e in the graph.
- Maintains a set S of vertices whose SP from v (source) has been determined.
- a) Select the next minimum distance node u, which is not in S.
- (b) ***for*** each node w adjacent to u ***do***

  if(dist[w]>dist[u]+cost[u,w])) then

  dist[w]:=dist[u]+cost[u,w];

- Repeat step (a) and (b) until S=n (number of vertices).

```
Algorithm ShortestPaths(v,cost,dist,n)
//dist[j], 1≤ j≤ n, is  the length of the shortest path
//from vertex v to vertex j in a digraph G with n vertices.
// dist[v] is set to zero. G is represented by its cost adjacency
// matrix cost[1:n, 1;n].
{
    for i:=  1 to n do
    {       // Initialize S.
        s[i]:=false;  dist[i]:=cost[v,i];
    }
    s[v]:=true; // put v in S.
```

```
for num:=2 to n do
   {
         Determine  n-1 paths from v.
         Choose u from among those vertices not in S such that
         dist[u] is minimum;

         s[u]:=true;   // Put u in S.
         for ( each w adjacent to u with s[w]= false) do
           // Uupdate distance
         if( dist[w]>dist[u]+cost[u,w]) ) then
               dist[w]:=dist[u]+cost[u,w];
   }
}
```

Time complexity of Dijkstra's Algorithm

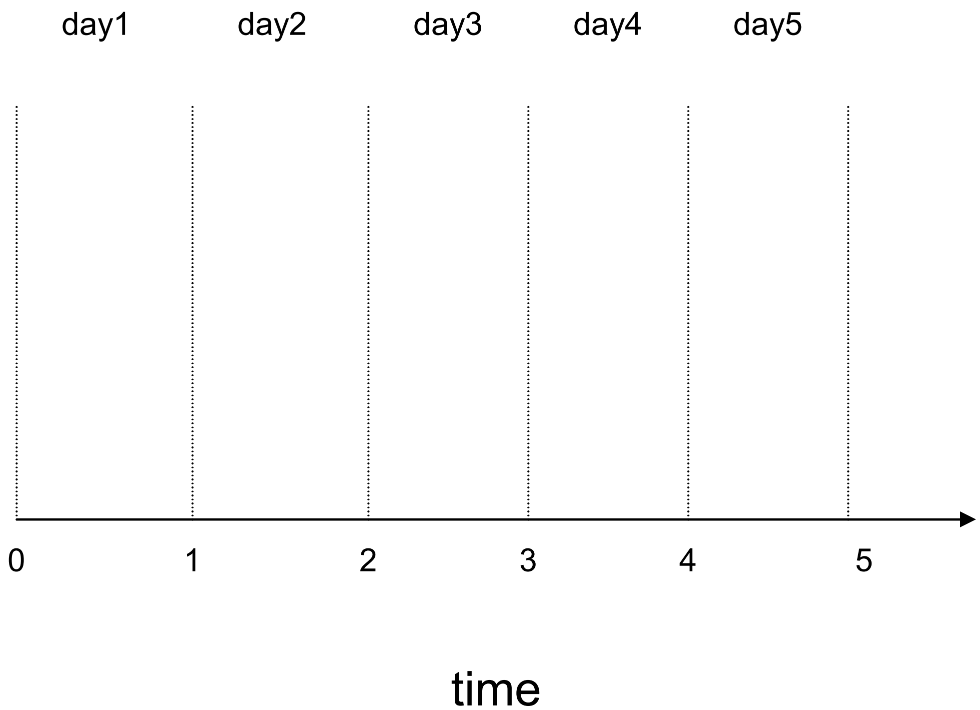
- The for loop of line 7 takes o(n).
- The for loop of line 12 takes o(n).
  – Each execution of this loop requires o(n) time at lines 15 and 18.
  – So the total time for this loop is o(n^2).
- Therefore, total time taken by this algorithm is o(n^2).

Job sequencing with deadlines

- We are given a set of *n* jobs.
- *Deadline di>=0* and a *profit pi>0* are associated with each *job i.*
- For any job profit is earned if and only if the job is *completed by its deadline*.
- To complete a job, a job has to be processed by a machine for *one unit* of time.
- Only *one machine* is available for processing jobs.
- A *feasible solution* of this problem is a *subset of jobs* such that each job in this subset can be completed by its deadline
- The *optimal solution* is a feasible solution which will *maximize* the total profit.
- *The objective is to find an order of processing of jobs which will maximize the total profit.*

Ex:-n=4, $(p_1,p_2,p_3,p_4) = (100,10,15,27)$

$(d_1,d_2,d_3,d_4) = (2,1,2,1)$



Ex:-n=4, (p1,p2,p3,p4)=(100,10,15,27)

(d1,d2,d3,d4)=(2,1,2,1)

The maximum deadline is 2 units, hence the feasible solution set must have <=2 jobs.

	feasible solution	processing sequence	value
1.	(1,2)	2,1	110
2.	(1,3)	1,3 or 3,1	115
3.	(1,4)	4,1	127
4.	(2,3)	2,3	25
5.	(3,4)	4,3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Solution 3 is optimal.

Greedy Algorithm for job sequencing with deadlines

1. Sort p_i into decreasing order. After sorting $p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_i$.
2. Add the next job i to the solution set if i can be completed by its deadline.
3. Stop if all jobs are examined. Otherwise, go to step 2.

Ex:- 1) $n=5$, ($p1,\ldots\ldots.,p5$) = (20,15,10,5,1) and

($d1,\ldots\ldots.d5$) = (2,2,1,3,,3)

The optimal solution is {1,2,4} with a profit of 40.

Ex:- 2) $n=7$, ($p1,\ldots\ldots.,p7$) = (3,5,20,18,1,6,30) and

($d1,\ldots\ldots..d7$) = (1,3,4,3,2,1,2)

Find out an optimal solution.

Algorithm JS(d,j,n)

//d[i]≥1, 1 ≤ i ≤ n are the deadlines.

//The jobs are ordered such that p[1] ≥p[2] …… ≥p[n]

// j[i] is the i^{th} job in the optimal solution, 1≤ i ≤ k

```
{
        d[0]=j[0]=0;    // Initialize
        j[1]=1; // Include job 1
        k=1;
```

```
for i=2 to n do
{       //Consider jobs in Descending order of p[i].
        // Find position for i and  check feasibility of
        // insertion.
        r=k;
        while(  (  d[ j[r] ] > d[ i ] ) and ( d[ j[ r ] ] > r )  ) do
                r = r-1;

        if(   d[ j[r] ] ≤ d[i]  and  d[ i ]  > r )) then
        {
                //  Insert i into  j[ ].
                for q=k to (r+1) step -1  do j[q+1] = j[q];

                j[r+1] :=i;
                k:=k+1;
        }
}
return k;

}
```

Time taken by this algorithm is $o(n^2)$