UNIT III:
Greedy method: General method, applications - Job sequencing with deadlines, 0/1 knapsack problem, Minimum cost spanning trees, Single source shortest path problem.
Dynamic Programming: General method, applications-Matrix chain multiplication, Optimal binary search trees, 0/1 knapsack problem, All pairs shortest path problem, Travelling sales person problem, Reliability design.

Greedy Method:
The greedy method is perhaps (maybe or possible) the most straight forward design technique, used to determine a feasible solution that may or may not be optimal.

Feasible solution:- Most problems have n inputs and its solution contains a subset of inputs that satisfies a given constraint(condition). Any subset that satisfies the constraint is called feasible solution.

Optimal solution: To find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called optimal solution.

The greedy method suggests that an algorithm works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution.

Greedy algorithms neither postpone nor revise the decisions (ie., no back tracking).
Example: Kruskal's minimal spanning tree. Select an edge from a sorted list, check, decide, and never visit it again.
Application of Greedy Method:
- Job sequencing with deadline
- 0/1 knapsack problem
- Minimum cost spanning trees
- Single source shortest path problem.

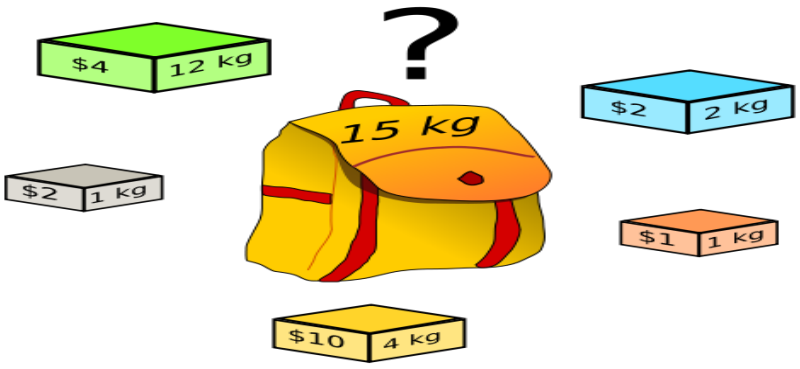
Algorithm for Greedy method

Algorithm Greedy(a,n)
//a[1:n] contains the n inputs.
{
Solution :=0;
For i=1 to n do
{
X:=select(a);
If Feasible(solution, x) then
Solution :=Union(solution,x);
}
Return solution;
}

Selection → Function, that selects an input from a[] and removes it. The selected input's value is assigned to x.
Feasible → Boolean-valued function that determines whether x can be included into the solution vector.
Union → function that combines x with solution and updates the objective function.

Knapsack problem

The **knapsack problem** or **rucksack (bag) problem** is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible



There are two versions of the problems

1. 0/1 knapsack problem
2. Fractional Knapsack problem
    a. Bounded Knapsack problem.
    b. Unbounded Knapsack problem.

Solutions to knapsack problems

- **Brute-force approach**:-Solve the problem with a straight farward algorithm
- **Greedy Algorithm**:- Keep taking most valuable items until maximum weight is reached or taking the largest value of eac item by calculating v_i=value$_i$/Size$_i$
- **Dynamic Programming**:- Solve each sub problem once and store their solutions in an array.

0/1 knapsack problem:

Let there be n items, z_1 to z_n where z_i has a value v_i and weight w_i. The maximum weight that we can carry in the bag is *W*. It is common to assume that all values and weights are nonnegative. To simplify the representation, we also assume that the items are listed in increasing order of weight.

Maximize $\sum_{i=1}^{n} v_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leqslant W, \qquad x_i \in \{0,1\}$

Maximize the sum of the values of the items in the knapsack so that the sum of the weights must be less than the knapsack's capacity.

Greedy algorithm for knapsack

Algorithm GreedyKnapsack(m,n)
// p[i:n] and [1:n] contain the profits and weights respectively
// if the n-objects ordered such that p[i]/w[i]>=p[i+1]/w[i+1], m→ size of knapsack and x[1:n]→ the solution vector
{
For i:=1 to n do x[i]:=0.0
U:=m;
For i:=1 to n do
{
if(w[i]>U) then break;
x[i]:=1.0;
U:=U-w[i];
}
If(i<=n) then x[i]:=U/w[i];
}

Ex: - Consider 3 objects whose profits and weights are defined as
(P_1, P_2, P_3) = (25, 24, 15)
$W_1, W_2, W_3)$ = (18, 15, 10)
n=3→number of objects
m=20→Bag capacity

Consider a knapsack of capacity 20. Determine the optimum strategy for placing the objects in to the knapsack. The problem can be solved by the greedy approach where in the inputs are arranged according to selection process (greedy strategy) and solve the problem in stages. The various greedy strategies for the problem could be as follows.

(x_1, x_2, x_3)	$\sum x_i w_i$	$\sum x_i p_i$
(1, 2/15, 0)	$18x1+\frac{2}{15}x15 = 20$	$25x1+\frac{2}{15}x\ 24 = 28.2$
(0, 2/3, 1)	$\frac{2}{3}x15+10x1= 20$	$\frac{2}{3}x\ 24 +15x1 = 31$

(0, 1, ½)	$1x15+\frac{1}{2}x10 = 20$	$1x24+\frac{1}{2}x15 = 31.5$
(½, ⅓, ¼)	½ x 18+⅓ x15+ ¼ x10 = 16. 5	½ x 25+⅓ x24+ ¼ x15 = 12.5+8+3.75 = 24.25

Analysis: - If we do not consider the time considered for sorting the inputs then all of the three greedy strategies complexity will be O(n).

Job Sequence with Deadline:

There is set of n-jobs. For any job i, is a integer deadling di≥0 and profit Pi>0, the profit Pi is earned iff the job completed by its deadline.

To complete a job one had to process the job on a machine for one unit of time. Only one machine is available for processing jobs.

A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline.

The value of a feasible solution J is the sum of the profits of the jobs in J, i.e., $\sum_{i\in j} P_i$

An optimal solution is a feasible solution with maximum value.

The problem involves identification of a subset of jobs which can be completed by its deadline. Therefore the problem suites the subset methodology and can be solved by the greedy method.

Ex: - Obtain the optimal sequence for the following jobs.

$j_1 \quad j_2 \quad j_3 \quad j_4$

$(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$

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

n =4

Feasible solution	Processing sequence	Value
$j_1 j_2$ (1, 2)	(2,1)	100+10=110
(1,3)	(1,3) or (3,1)	100+15=115
(1,4)	(4,1)	100+27=127
(2,3)	(2,3)	10+15=25
(3,4)	(4,3)	15+27=42
(1)	(1)	100
(2)	(2)	10
(3)	(3)	15
(4)	(4)	27

In the example solution '3' is the optimal. In this solution only jobs 1&4 are processed and the value is 127. These jobs must be processed in the order j4 followed by j1. the process of job 4 begins at time 0 and ends at time 1. And the processing of job 1 begins at time 1 and ends at time2. Therefore both the jobs are completed within their deadlines. The optimization measure for determining the next job to be selected in to the solution is according to the profit. The next job to include is that which increases ∑pi the most, subject to the constraint that the resulting "j" is the feasible solution. Therefore the greedy strategy is to consider the jobs in decreasing order of profits.

The greedy algorithm is used to obtain an optimal solution.

We must formulate an optimization measure to determine how the next job is chosen.

```
algorithm js(d, j, n)
//d→ dead line, j→subset of jobs ,n→ total number of jobs
// d[i]≥1 1 ≤ i ≤ n are the dead lines,
// the jobs are ordered such that p[1]≥p[2]≥---≥p[n]
//j[i] is the ith job in the optimal solution 1 ≤ i ≤ k, k→ subset range
{
d[0]=j[0]=0;
j[1]=1;
k=1;
for i=2 to n do{
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
{
for q:=k to (r+1) setp-1 do j[q+1]= j[q];
j[r+1]=i;
k=k+1;
}
}
return k;
}
```

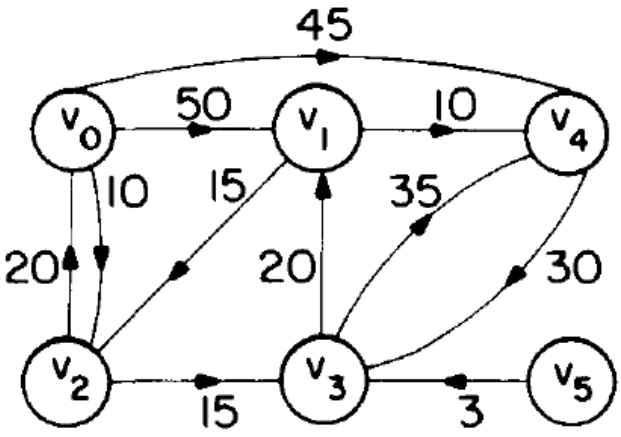
Note: The size of sub set j must be less than equal to maximum deadline in given list.

Single Source Shortest Paths:

- Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway.
- The edges have assigned weights which may be either the distance between the 2 cities connected by the edge or the average time to drive along that section of highway.
- For example if A motorist wishing to drive from city A to B then we must answer the following questions
  - Is there a path from A to B
  - If there is more than one path from A to B which is the shortest path
- The length of a path is defined to be the sum of the weights of the edges on that path.

Given a directed graph G(V,E) with weight edge w(u,v). e have to find a shortest path from source vertex S$\in$v to every other vertex v1$\in$ V−S.

- To find SSSP for directed graphs G(V,E) there are two different algorithms.
  - Bellman-Ford Algorithm
  - Dijkstra's algorithm
- Bellman-Ford Algorithm:- allow –ve weight edges in input graph. This algorithm either finds a shortest path form source vertex **S∈V** to other vertex **v∈V** or detect a –ve weight cycles in G, hence no solution. If there is no negative weight cycles are reachable form source vertex **S∈V** to every other vertex **v∈V**
- Dijkstra's algorithm:- allows only +ve weight edges in the input graph and finds a shortest path from source vertex **S∈V** to every other vertex **v∈V**.



	Path	Length
1)	$v_0\ v_2$	10
2)	$v_0\ v_2\ v_3$	25
3)	$v_0\ v_2\ v_3\ v_1$	45
4)	$v_0\ v_4$	45

Graph and shortest paths from v_0 to all destinations

- Consider the above directed graph, if node 1 is the source vertex, then shortest path from 1 to 2 is 1,4,5,2. The length is 10+15+20=45.
- To formulate a greedy based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also of an optimization measure.
- This is possible by building the shortest paths one by one.
- As an optimization measure we can use the sum of the lengths of all paths so far generated.
- If we have already constructed 'i' shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path.
- The greedy way to generate the shortest paths from Vo to the remaining vertices is to generate these paths in non-decreasing order of path length.
- For this 1st, a shortest path of the nearest vertex is generated. Then a shortest path to the 2nd nearest vertex is generated and so on.

Algorithm for finding Shortest Path

```
Algorithm ShortestPath(v, cost, dist, n)
//dist[j], 1≤j≤n, is set to the length of the shortest path from vertex v to vertex j in graph g with n-vertices.
// dist[v] is zero
{
for i=1 to n do{
s[i]=false;
dist[i]=cost[v,i];
}
s[v]=true;
dist[v]:=0.0; // put v in s
for num=2 to n do{
// determine n-1 paths from v
choose u form 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
if(dist[w]>(dist[u]+cost[u, w])) then
dist[w]=dist[u]+cost[u, w];
}
}
```

Minimum Cost Spanning Tree:

SPANNING TREE: - A Sub graph 'n' of o graph 'G' is called as a spanning tree if

(i) It includes all the vertices of 'G'

(ii) It is a tree

Minimum cost spanning tree: For a given graph 'G' there can be more than one spanning tree. If weights are assigned to the edges of 'G' then the spanning tree which has the minimum cost of edges is called as minimal spanning tree.

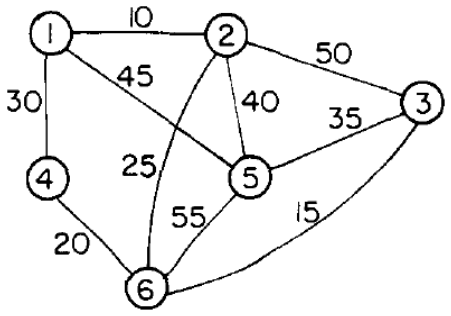
The greedy method suggests that a minimum cost spanning tree can be obtained by contacting the tree edge by edge. The next edge to be included in the tree is the edge that results in a minimum increase in the some of the costs of the edges included so far.

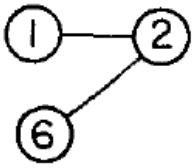
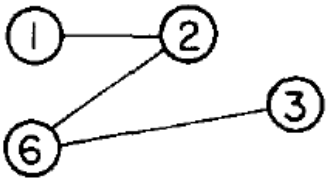
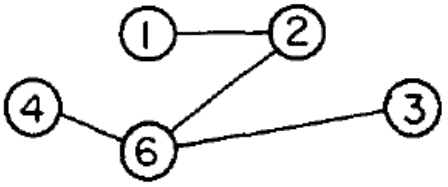
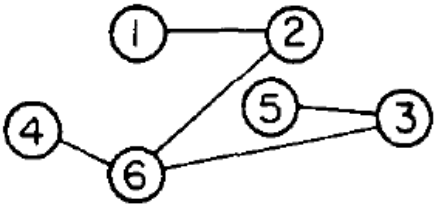
There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms

→Prim's Algorithm

→Kruskal's Algorithm

Prim's Algorithm: Start with any *one node* in the spanning tree, and repeatedly add the cheapest edge, and the node it leads to, for which the node is not already in the spanning tree.



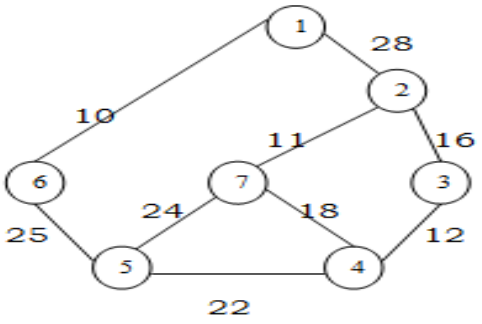
Edge	Cost	Spanning tree
(1,2)	10	
(2,6)	25	
(3,6)	15	
(6,4)	20	
(1,4)	reject	
(3,5)	35	

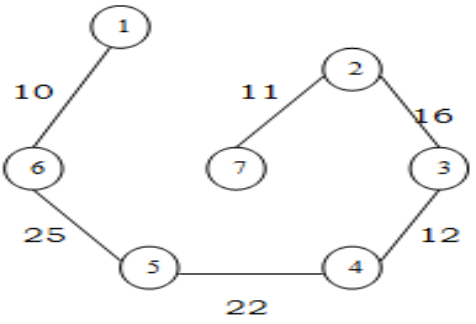
Stages in Prim's Algorithm

PRIM'S ALGORITHM: -

i) Select an edge with minimum cost and include in to the spanning tree.
ii) Among all the edges which are adjacent with the selected edge, select the one with minimum cost.
iii) Repeat step 2 until 'n' vertices and (n-1) edges are been included. And the sub graph obtained does not contain any cycles.

Notes: - At every state a decision is made about an edge of minimum cost to be included into the spanning tree. From the edges which are adjacent to the last edge included in the spanning tree i.e. at every stage the sub-graph obtained is a tree.



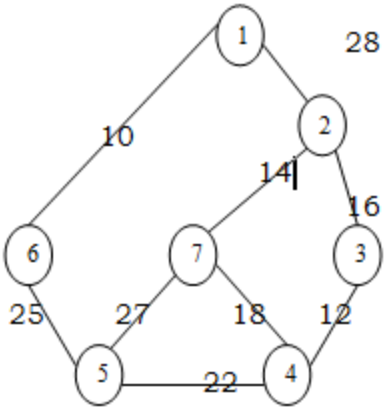


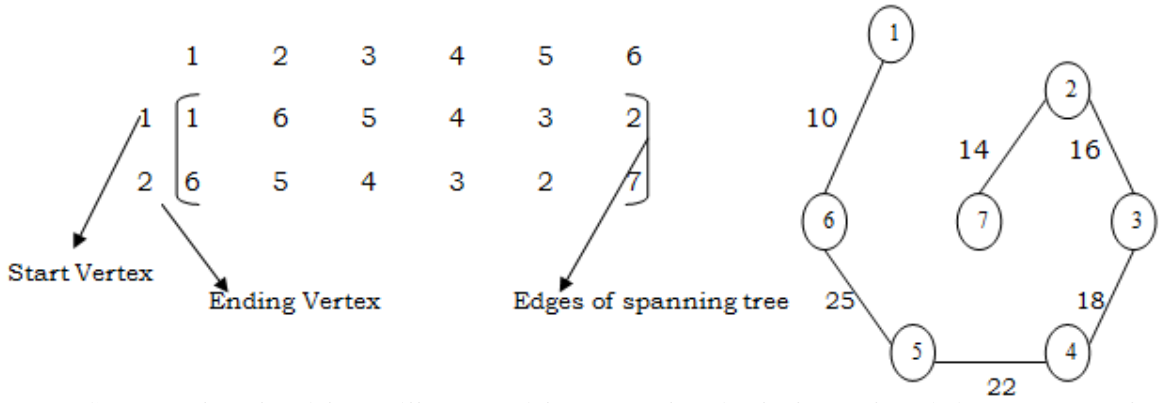
Prim's minimum spanning tree algorithm

```
Algorithm Prim (E, cost, n,t)
// E is the set of edges in G. Cost (1:n, 1:n) is the
// Cost adjacency matrix of an n vertex graph such that
// Cost (i,j) is either a positive real no. or ∞ if no edge (i,j) exists.
//A minimum spanning tree is computed and
//Stored in the array T(1:n-1, 2).
//(t (i, 1), + t(i,2)) is an edge in the minimum cost spanning tree. The final cost is returned
        {
        Let (k, l) be an edge with min cost in E
        Min cost: = Cost (x,l);
        T(1,1):= k; + (1,2):= l;
for i:= 1 to n do//initialize near
        if (cost (i,l)<cost (i,k) then n east (i):  l;
        else near (i): = k;
        near (k): = near (l): = 0;
        for i: = 2 to n-1 do
{//find n-2 additional edges for t
let j be an index such that near (i) ≠0 & cost (j, near (i)) is minimum;
t (i,1): = j + (i,2): = near (j);
min cost: = Min cost + 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 Z(k): = ji
}
return mincost;
}
```

The algorithm takes four arguments E: set of edges, cost is nxn adjacency matrix cost of (i,j)= +ve integer, if an edge exists between i&j otherwise infinity. 'n' is no/: of vertices. 't' is a (n-1):2matrix which consists of the edges of spanning tree.
E = { (1,2), (1,6), (2,3), (3,4), (4,5), (4,7), (5,6), (5,7), (2,7) }
n = {1,2,3,4,5,6,7)

Cost	1	2	3	4	5	6	7
1	α	28	α	α	α	10	α
2	28	α	16	α	α	α	14
3	α	10	α	12	α	α	α
4	α	α	12	α	22	α	18
5	α	α	α	22	α	25	24
6	10	α	α	α	25	α	α
7	α	14	α	18	24	α	α





i) The algorithm will start with a tree that includes only minimum cost edge of G. Then edges are added to this tree one by one.

ii) The next edge (i,j) to be added is such that i is a vertex which is already included in the treed and j is a vertex not yet included in the tree and cost of i,j is minimum among all edges adjacent to 'i'.

iii) With each vertex 'j' next yet included in the tree, we assign a value near 'j'. The value near 'j' represents a vertex in the tree such that cost (j, near (j)) is minimum among all choices for near (j)

iv) We define near (j):= 0 for all the vertices 'j' that are already in the tree.

v) The next edge to include is defined by the vertex 'j' such that (near (j)) ≠ 0 and cost of (j, near (j)) is minimum.

Analysis: -

The time required by the prince algorithm is directly proportional to the no/: of vertices. If a graph 'G' has 'n' vertices then the time required by prim's algorithm is **$O(n^2)$**

Kruskal's Algorithm: Start with *no* nodes or edges in the spanning tree, and repeatedly add the cheapest edge that does not create a cycle.

In Kruskals algorithm for determining the spanning tree we arrange the edges in the increasing order of cost.

i) All the edges are considered one by one in that order and deleted from the graph and are included in to the spanning tree.

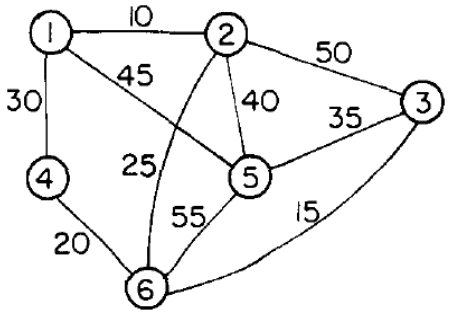
ii) At every stage an edge is included; the sub-graph at a stage need not be a tree. Infect it is a forest.

iii) At the end if we include 'n' vertices and n-1 edges without forming cycles then we get a single connected component without any cycles i.e. a tree with minimum cost.

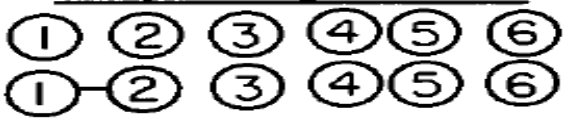
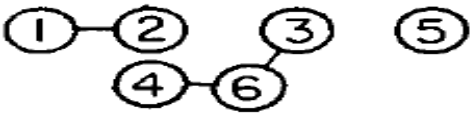
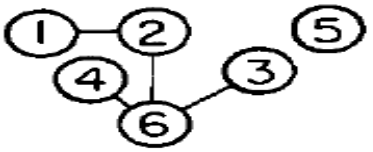
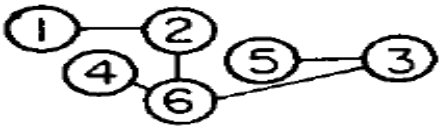
At every stage, as we include an edge in to the spanning tree, we get disconnected trees represented by various sets. While including an edge in to the spanning tree we need to check it does not form cycle. Inclusion of an edge (i,j) will form a cycle if i,j both are in same set. Otherwise the edge can be included into the spanning tree.

Kruskal minimum spanning tree algorithm

```
Algorithm Kruskal (E, cost, n,t)
//E is the set of edges in G. 'G' has 'n' vertices
//Cost {u,v} is the cost of edge (u,v) t is the set
//of edges in the minimum cost spanning tree
//The final cost is returned
{ construct a heap out of the edge costs using heapify;
        for i:= 1 to n do parent (i):= -1 // place in different sets
//each vertex is in different set              {1} {1} {3}
        i: = 0; min cost: = 0.0;
        While (i<n-1) and (heap not empty))do
{
Delete a minimum cost edge (u,v) from the heaps; and reheapify using adjust;
j:= find (u); k:=find (v);
if (j≠k) then
{  i: = 1+1;
   + (i,1)=u; + (i, 2)=v;
    mincost: = mincost+cost(u,v);
    Union (j,k);
    }
}
if (i≠n-1) then write ("No spanning tree");
   else return mincost;
}
```



Consider the above graph of , Using Kruskal's method the edges of this graph are considered for inclusion in the minimum cost spanning tree in the order (1, 2), (3, 6), (4, 6), (2, 6), (1, 4), (3, 5), (2, 5), (1, 5), (2, 3), and (5, 6). This corresponds to the cost sequence 10, 15, 20, 25, 30, 35, 40, 45, 50, 55. The first four edges are included in T. The next edge to be considered is (I, 4). This edge connects two vertices already connected in T and so it is rejected. Next, the edge (3, 5) is selected and that completes the spanning tree.

Edge	Cast	Spanning Forest
(1,2)	10	
(3,6)	15	
(4,6)	20	
(2,6)	25	
(1,4)	30	(reject)
(3,5)	35	

Stages in Kruskal's algorithm

Analysis: - If the no/: of edges in the graph is given by /E/ then the time for Kruskals algorithm is given by 0 (|E| log |E|).