Greedy method: General method, applications- knapsack problem, spanning trees, Job sequencing with deadlines, Minimum cost spanning trees, Prim's Algorithm, Kruskal's Algorithms, An optimal randomized algorithm, Optimal Merge Patterns, Single source shortest path problem.

General Method:

- Greedy method is the most straight forward design technique. It is used to solve optimization problems, and can be applied to wide variety of problems.
- These problems have *n* inputs and require us to obtain a subset that satisfies some constraints.
- Any subset that satisfies these constraints is called a feasible solution. A feasible solution that either maximizes or minimizes a given objective function need to be find. A feasible solution that does this is called an optimal solution.
- An algorithm that works in stages can be devised by using greedy method. At each stage, a decision is made whether a particular input is optimal solution or not. This is done by considering the inputs in an order determined by some selection procedure.
- The selection procedure is based on some optimization measure. This measure may be objective function. This version of greedy technique is called subset paradigm.

The subset paradigm technique is described in the following algorithm abstractly.

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

- The function *select* selects and removes an input from *a[]* and the input value is assigned to *x*.
- The function *Feasible* determines whether *x* can be included into the solution or not.
- The function *union* combines *x* with the *solution* and updates.
- The above algorithm gives an abstract view of greedy technique. The functions select, feasible and union need to be properly implemented for solving a particular problem.

Ordering Paradigm: In this paradigm, each decision is made using an optimization criterion that can be computed using decisions already made.

Example 1: Change making:
Pay 67 rupees with fewest numbers of notes.
Construct the solution in stages.
At each stage a note is added, to increase the total amount.
To assure feasibility of the solution, the selected note should not cause the total amount added so far to exceed the desired amount.
The first note selected is 50 rupees. A 50 rupee note or 20 rupee note cannot be selected as a second note. A 10 rupee note is selected as a second note. A 5 rupee note and 2 rupee note are selected as the remaining change. (67=50+10+5+2)

Knapsack problem:

- We are given n objects and a knapsack. Object i has a weight w_i and the knapsack has a capacity m.
- If a fraction x_i, $0 \le x_i \le 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
- As the knapsack capacity is m, the total weight of all the chosen objects to be at most m.

Formally, the problem can be stated as

Maximize $\sum_{1 \le i \le n} p_i x_i$ ---- (1)

Subject to $\sum_{1 \le i \le n} w_i x_i \le m$ ---- (2)

and $0 \le x_i \le 1, \quad 1 \le i \le n$ ---- (3)

The profits and weights are positive numbers.

A <u>feasible solution</u> is any set $(x_1, \ldots\ldots, x_n)$, which satisfies (2) and (3).

An <u>optimal solution</u> is a feasible solution which maximizes $\sum_{1 \le i \le n} p_i x_i$.

Example: consider the following instances of the knapsack problem

$n=3, m=20, (p_1, p_2, p_3)=(25,24,15)$ and $(w_1, w_2, w_3)=(18,15,10)$

Some feasible solutions are

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1	$\left(\frac{1}{2}, \frac{1}{3}, \frac{1}{4}\right)$	9 + 5 + 2.5 = 16.5	12.5 + 8 + 3.75 = 24.25
2	$\left(1, \frac{2}{15}, 0\right)$	18 + 2 + 0 = 20	25 + 3.2 + 0 = 28.2
3	$\left(0, \frac{2}{3}, 1\right)$	0 + 10 + 10 = 20	0 + 16 +15= 31
4	$\left(0, 1, \frac{1}{2}\right)$	0 + 15 + 5 = 20	0 + 24 + 7.5 = 31.5

Out of these 4 feasible solutions, solution 4 yields the maximum profit. Therefore this solution is optimal for the given problem instance.

***In case the sum of all the weights is $\le m$, then $x_i=1, 1 \le i \le n$ is an optimal solution.

The knapsack problem calls for selecting a subset of objects and hence fits the subset paradigm. In addition, this problem also involves the selection of x_i for each object.

The greedy strategies to obtain feasible solutions are

<u>Strategy 1:</u>

- Fill the knapsack by including next the object with largest profit. If the object doesn't fit, then a fraction of it is included to fill the knapsack.

- Thus each time an object is included into the knapsack, we obtain the largest possible increase in profit value. If we follow this method for the above example, solution 2 is formed. But this solution is suboptimal.
  ∴ This greedy method did not yield an optimal solution.

Strategy 2:

- Fill the knapsack by including next the object with least weight.
- If we follow this method in the above example, solution 3 results. This too is suboptimal.

Strategy 3:

- This strategy strives to achieve a balance between the rate at which profit increases and the rate at which capacity is used.
- At each step, we include that object which has maximum profit per unit of capacity used. i.e., the objects are considered in order of the ratio $\frac{p_i}{w_i}$. Solution 4 results, if we follow this approach.

Following algorithm(GreedyKnapsack) obtains solutions corresponding to this strategy. The objects have to be sorted in decreasing order of $\frac{p_i}{w_i}$.

```
Algorithm GreedyKnapsack(m, n)
{
        for i:=1 to m 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];
}
```

If time to sort the objects is discarded, then this strategy requires ***O(n)*** time.

Exercise :

1. Find an optimal solution to the knapsack instance n=7, m=15, $(p_1,\ldots,p_7) = (10, 5, 15, 7, 6, 18, 3)$ and $(w_1,\ldots,w_7) = (2, 3, 5, 7, 1, 4, 1)$

Job sequencing with deadlines:

- The problem is the number of jobs, their profit and deadlines will be given and we have to find a sequence of job, which will be completed within its deadline, and it should yield a maximum profit.

Points To remember:

- Each job i is associated with a deadline $d_i \geq 0$ and a profit $p_i > 0$.
- The profit p_i is earned if and only if the job is completed by its deadline.
- To complete a job, one has 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 their deadline.
- The value of the feasible solution J is the sum of the profits of the jobs in J or $\sum_{i \in J} p_i$.

- An optimal solution is a feasible solution with maximum value.
- This problem fits the subset paradigm, since it involves identification of a subset.

Since one job can be processed in a single machine. The other job has to be in its waiting state until the job is completed and the machine becomes free.

So the waiting time and the processing time should be less than or equal to the dead line of the job.

Example : Let $n = 4$, $(p_1,p_2,p_3,p_4) = (100,10,15,27)$, $(d_1,d_2,d_3,d_4) = (2,1,2,1)$

Consider the jobs in the nonincreasing order of profits subject to the constraint that the resulting job sequence J is a feasible solution.

Sr.No.	Feasible Solution	Processing Sequence	Profit value
(i)	(1,2)	(2,1)	110
(ii)	(1,3)	(1,3) or (3,1)	115
(iii)	(1,4)	(4,1)	127 is the optimal one
(iv)	(2,3)	(2,3)	25
(v)	(3,4)	(4,3)	42
(vi)	(1)	(1)	100
(vii)	(2)	(2)	10
(viii)	(3)	(3)	15
(ix)	(4)	(4)	27

To formulate the greedy algorithm for an optimal solution, we must formulate an optimization measure to determine how the next job is chosen.

- Choose the objective function $\sum_{i \in J} p_i$ as the optimization measure.
- The next job to include is the one that increases $\sum_{i \in J} p_i$ the most, subject to the constraint that the resulting J is a feasible solution.
- This requires us to consider the jobs in the nonincreasing order of p_i's .
- Add one by one job to J and check whether it is feasible or not.
- To determine whether the given J is feasible solution, check only permutation in which jobs are ordered in nondecreasing order of deadlines.

Solution by using greedy approach

Job considered	Action	J	Assigned slots	Profit
1	Accept	{1}	(1,2)	100
4	Accept	{1,4}	(0,1)(1,2)	127
3	Reject	{1,4}	(0,1)(1,2)	127
2	Reject	{1,4}	(0,1)(1,2)	127

The optimal solution is J = {1, 4}

With a profit of 127.

Ex 2: Let n = 5, (p_1, . . ., p_5) = (20, 15, 10, 5, 1) and (d_1, . ., d_5) = (2, 2, 1, 3, 3)

Job considered	Action	J	Assigned slots	Profit
1	Accept	{1}	(1,2)	20
2	Accept	{1,2}	(0,1)(1,2)	35
3	Reject	{1,2}	(0,1)(1,2)	35
4	Accept	{1,2,4}	(0,1)(1,2)(2,3)	40
5	Reject	{1,2,4}	(0,1)(1,2)(2,3)	40

The optimal solution is J = {1, 2, 4}
With a profit of 40.

Ex 3: Let n = 7, (p_1, . . ., p_7) = (3, 5, 20, 18, 1, 6, 30) and (d_1, . ., d_7) = (1, 3, 4, 3, 2, 1, 2)

Job considered	Action	J	Assigned slots	Profit

The optimal solution is J =
With a profit of _____

A high level description of the greedy algorithm for Job Sequencing with deadlines problem is shown in the algorithm. This algorithm constructs an optimal set *J* of jobs that can be processed by their due times. The selected jobs can be processed in the order given by the following theorem.

> " Let J be a set of k jobs and σ = (i_1,i_2,….i_k) be a permutation of jobs in J such that $di_1 \le di_2 \le \ldots \le di_k$. Then J is a feasible solution iff the jobs in J can be processed in the order σ without violating any deadline."

```
Algorithm GreedyJob(d, J, n)
// J is a set of Jobs that can be completed by their deadlines.
{
        J := { 1 } ;
        for i =2 to n do
        {
                if ( all jobs in J U { i } can be completed by their deadlines ) then J := J U { i };
        }
}
```

To write complete algorithm,

- We can use an array *d[1:n]* to store the deadlines of the jobs in the order of their p-values.
- The set J itself can be represented by a one-dimensional array *J[1:k]* such that *J[r]*, $1 \le r \le k$ are the jobs in *J* and $d[J[1]] \le d[J[2]] \le \ldots\ldots \le d[J[k]]$.
- To test weather $J \cup \{i\}$ is feasible, we have just to insert *i* into *J* preserving the deadline ordering and then verify that $d[J[1]] \le r,\ 1 \le r \le k+1$.
- The insertion of *i* into *J* is simplified by the use of a fictitious *job 0* with *d[0]=0, J[0]=0*.

The algorithm that results from the above discussion is

Algorithm JS(*d, J, n*)

```
//The jobs are ordered such that p[1]>p[2]...>p[n]
//J[i] is the i^th job in the optimal solution also at termination d[J[ i]] ≤ d[ J[i+1]],1 < i < k
{
   d[0]:= J[0]:=0;
   J[1]:=1;
   k=1;
   for i :=2 to n do
   {     // consider jobs in nonincreasing order of P[i]; find the position for i and check feasibility 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
         {
            for q:=k to (r+1) step –1 do J [q+1]:=J[q];
            J[r+1]:=i; k:=k+1;
         }
   }
   return k;
}
```

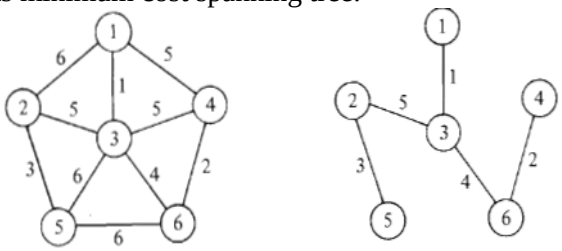
Complexity:

- Complexity of Job Sequencing can be measured in terms of two parameters. They are *n*, the number of jobs and S, the number of jobs included in the solution.
- The while loop iterates for at most *k* times. Each iteration takes $\Theta(1)$.
- If the if condition is true then the time taken to insert job *I* is $\Theta(k-r)$.
- Hence, the total time for each iteration of for loop is time $\Theta(k)$.
- Loop iterates for $(n-1)$ times.
- If *s* is the final value of *k*, then the time needed by algorithm JS is – $\Theta(sn)$
- Since sn, the worst case time, as a function of n alone is - $\Theta(n^2)$

Minimum-cost spanning trees:

- Let ***G=(V, E)*** be an undirected connected graph. A subgraph $t=(V,E')$ of **G** is a spanning tree of **G** iff ***t*** is a tree.
- In practical situations, the edges have weights assigned to them. The cost of spanning trees is the sum of the costs of the edges in that tree.
- The spanning tree of **G** represents all feasible choices. A minimum-cost spanning tree is a spanning tree with minimum cost.

- Figure shows a graph and its minimum cost spanning tree.



- Since the identification of minimum-cost spanning tree involves the selection of a subset of the edges, this problems fits the subset paradigm.
- A greedy method to obtain a minimum-cost spanning tree builds this tree edge by edge.
- The next edge to include is chosen according to some optimization criterion.
- The simplest such criterion is to choose an edge that result in a minimum increase in the sum of the costs of the edges so far included.

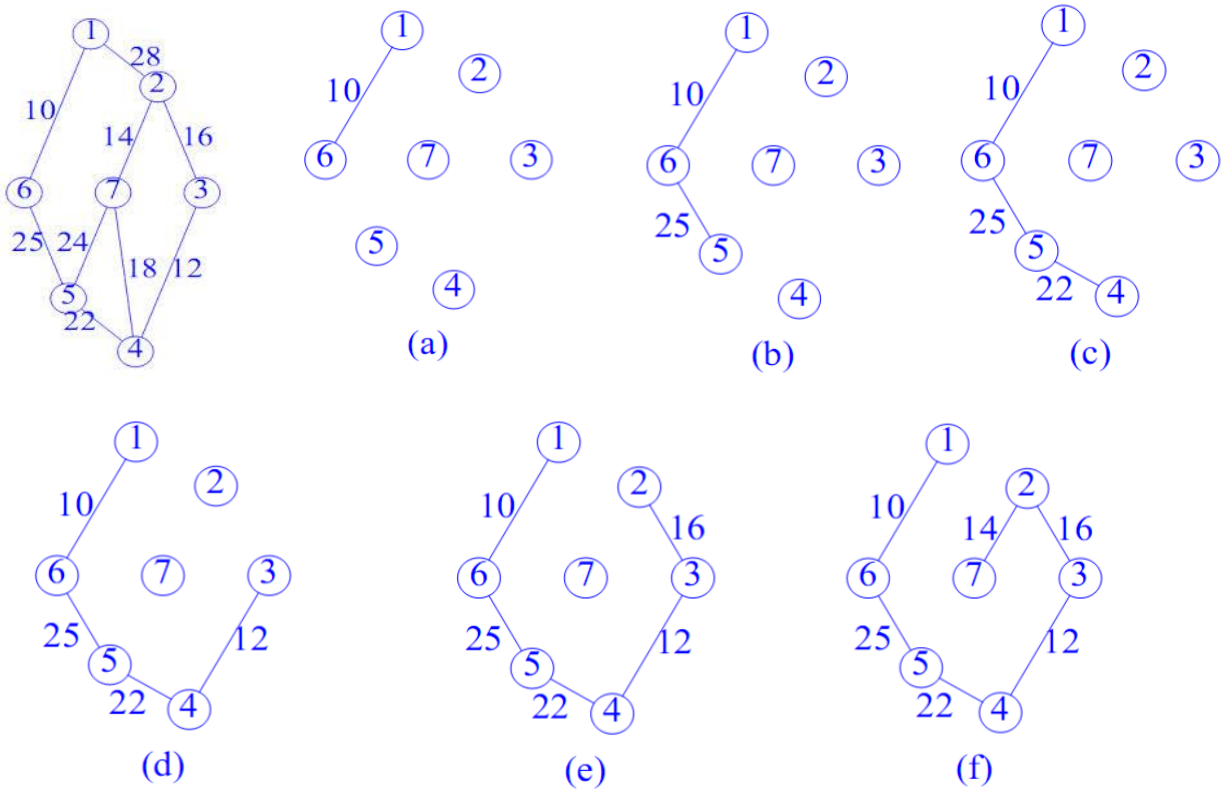
There are two possible ways to interpret this criterion. Their respective algorithms are

1. Prim's algorithm
2. Kruskal's algorithm

Prim's Algorithm:

- The set of edges selected by this algorithm should form a tree.
- Start from an arbitrary vertex and store it in A.
- Thus, if A is the set of edges selected so far, then A forms a tree.
- The next edge (u, v) to be included in A is a minimum-cost edge not in A with the property that $AU\{(u,v)\}$ is also a tree.

Following figures shows the working of prim's method on a graph.



Pseudocode algorithm to find a minimum cost spanning tree:

```
Algorithm Prim (E, cost, n, t)
//E is the set of edges in G. cost[1:n, 1:n] is the cost adjacency matrix.
//A MST is computed and stored as a set of edges in the array t[1:n-1, 1:2]. The final cost is returned.
{
    Let (k, l) be an edge with mincost in E;
    mincost := cost[k, l];
    t[1,1]:=k, t[1,2]:=l;
    for i:=1 to n do
        if cost[i,l] < cost[i,k] then near[i]:=l;
        else near[i]:=k;
    near[k]:=near[l]:=0;
    for i:=2 to n-1 do
    {
        let j be an index such that near[j]≠0 and cost[i,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
            if (near[k]≠0 and cost[k,near[k]]>cost[k,j]) then near[k]:=j;
    }
    return mincost;
}
```

- This algorithm will start with a tree that includes only a minimum cost edge of *G*. Then edges are added to this tree one by one.
- The next edge (*i*, *j*) to be added is such that
  - *i* is a vertex already included in the tree.
  - *j* is a vertex not yet included.
  - The cost of (*i*, *j*) is minimum among all edges (*k*, *l*) such that vertex *k* is in the tree and vertex *l* is not in the tree.
  - To determine this edge (*i*, *j*) efficiently, we associate with each vertex j not yet included in the tree a value *near[j]*.
  - For all vertices *j* that are already in the tree, set *near[j]=0*.
  - The next edge to include is defined by the vertex *j* such that *near[j]≠0* and *cost[j, near[j]]* is minimum.

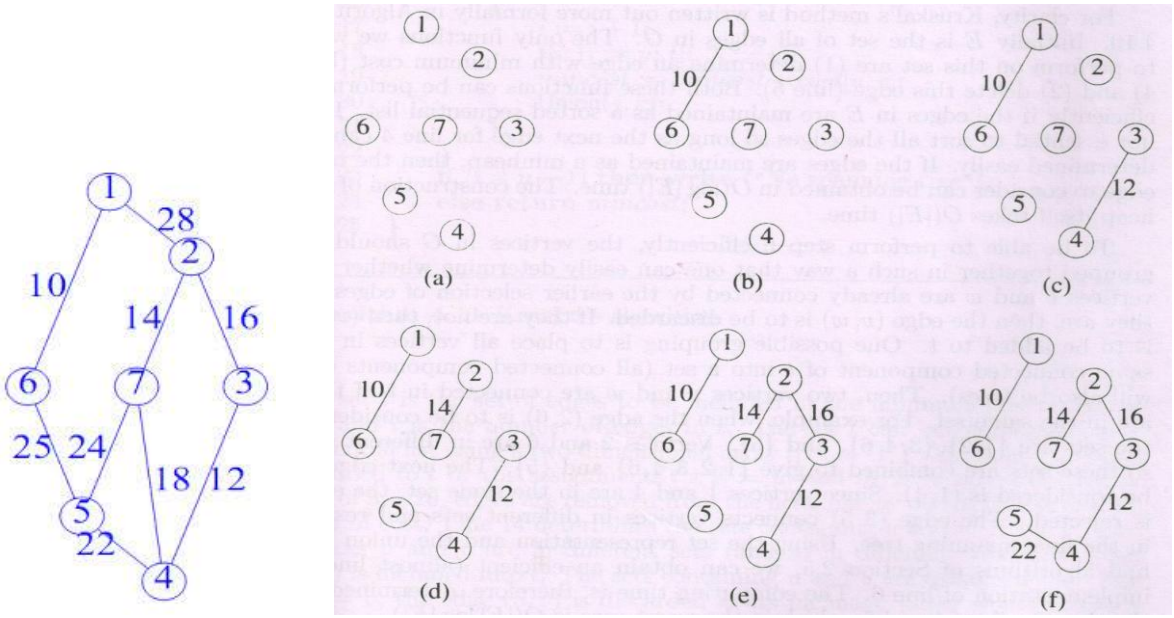
Running Time:

- Selecting edge with minimum cost : O(|E|)
- Initialization of *near[]* takes : $\Theta(n)$
- For loop runs for all the vertices i.e., *n* times and it includes updation of *near[]* which runs for *n* times. Cost is : $O(n^2)$
- Hence, prim runs in $O(n^2)$ time.

Kruskal's algorithm:

- In this algorithm, the edges of the graph are considered in nondecreasing order of cost.
- The set of *t* edges selected for the spanning tree be such that it is possible to complete *t* into a tree. Thus *t* may not be a tree at all stages in the algorithm.
- The set of edges selected are generally a forest since the set of edges *t* can be completed into a tree iff there are no cycles in *t*.
- This interpretation also results in a Minimum Spanning Tree.

Example: following figure shows stages in kruskal's algorithm. It begins with no edges selected.



Consider the graph shown above.

- We begin with no edges selected. Figure (a) shows the graph with no edges selected.
- First edge considered is (1,6). It is included in the spanning tree and it yields the graph in figure (b)
- Next, the edge (3,4) is considered and included in the tree and it shown in figure (c).
- Next, the edge (2,7) is considered and included in the tree and it shown in figure (d).
- Next, the edge (2,3) is considered and included in the tree and it shown in figure (e).
- Of the edges not yet considered, (7, 4) has the least cost. It is considered next. Its inclusion in the tree results in a cycle, so this edge is discarded.
- Next, the edge (5,4) is considered and included in the tree and it shown in figure (f).
- As edge (5,7) forms a cycle, addition of edge (5,6) completes the MST.

Kruskal's algorithm can be formally written as

```
t := Ø;
while( ( t has less than n-1 edges ) and ( E ≠ Ø ) ) do
{
   Choose an edge (v, w) from E of lowest cost;
   Delete (v, w) from E;
   If ((v, w) does not create a cycle in t) then
         Add (v, w) to t;
   else discard (v, w);
}
```

While implementing this, the steps

1. Determining minimum edge and deleting the edge can be performed efficiently by constructing min-heap with edge costs and taking root of it. This takes O(|E|) time for heap and O(log|E|) for selecting next edge and reheap.
2. The selected edge will form a cycle in *t* or not can be identified with find operation. And if it is not forming a cycle then adding this to the tree can be performed using union operation, by representing nodes as sets. Which are the operations of linear complexity So the total algorithm will take an O(|E| log(|E|). Where |E| is the edges of G.

Pseudo code for kruskal's algorithm:

```
Algorithm kruskal(E,cost,n,t)
//E→set of edges in G has 'n' vertices.
//cost[u,v]→cost of edge (u,v). t→set of edges in minimum cost spanning tree
// the minimum cost is returned.
{
        Construct a heap out of the edge costs;
        for i:=1 to n do parent[i]:= -1;
        i:=0; mincost:=0.0;
        while((i<n-1)and (heap not empty)) do
        {
                delete a minimum cost edge (u,v) from the heap;
                j:=find(u); k:=find(v);
                if ( j ≠ k) then
                {
                        i:=i+1;
                        t[i,1]:=u; t[i,2]:=v;
                        mincost=mincost+cost[u,v];
                        union(j,k);
                }
        }
        if ( i ≠ n-1) then write("No spanning tree"); else return minimum cost;
}
```

Analysis :
The time complexity of minimum cost spanning tree algorithm in worst case is O(|E|log|E|), where E is the edge set of G.

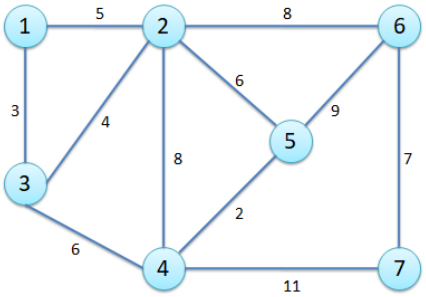
Optimal Randomized Algorithm :

Any algorithm for finding the minimum-cost spanning tree of a given graph G(V, E) will take $\Omega(|V| + |E|)$ time in the worst case.
Reason : since it has to examine each node and each edge at least once before determining the correct answer.

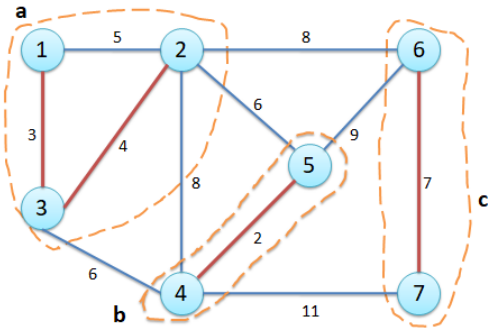
An optimal randomized algorithm which takes O(|V| + |E|) can be devised as follows

1. Randomly sample *m* edges from G
2. Let G^1 be the induced sub graph; i.e., G^1 has V as its node set and the sampled edges in the edge set. The subgraph G^1 need not be connected. Recursively find Minimum Cost Spanning Tree for each component of G^1. Let F be the resultant minimum cost spanning forest of G^1.
3. Using F eliminate certain edges (heavy edges) of G that cannot possibly in a Minimum Cost Spanning Tree. Let G^{11} be the graph that results from G after elimination of F-heavy edges.
4. Recursively find Minimum Cost Spanning Tree for G^{11}.

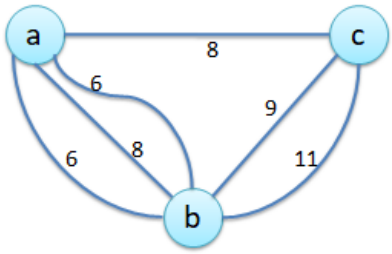
Example :
Given input graph G is

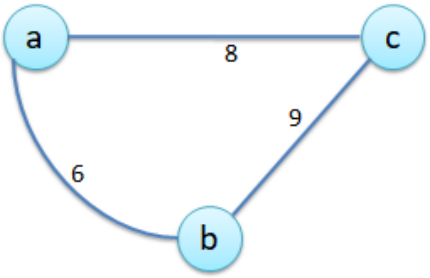


Randomly sampled 4 edges (1,3) , (2,3), (4,5) and (6,7)
Selected edges are forming three trees (forest). They should be minimum spanning trees.

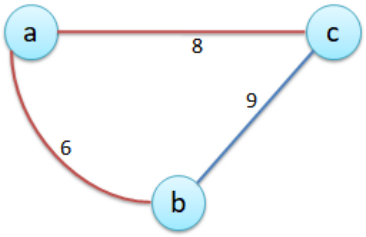


Using F translate the given Graph G into G1 and remove the heavy edges.

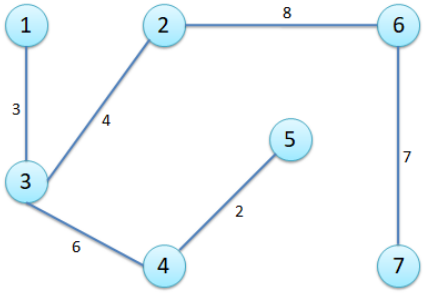




Apply the same process recursively on obtained graph



The resultant minimum spanning tree is

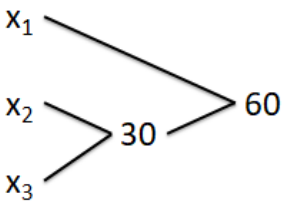


Optimal Merge Patterns

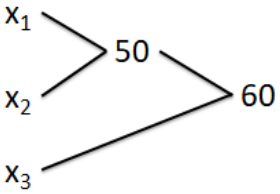
Problem : Determining an optimal way (one requiring the fewest comparisons) to pair-wise merge n sorted files.

- Merging two sorted files having *n* and *m* records to obtain one sorted file takes *O(n+m)* time.
- When more than two sorted files are to be merged together, the merge can be accomplished by repeatedly merging sorted files in pairs.
- Given n sorted files, there are many ways in which to pairwise merge them into a single sorted file.
- Different pairings require different computing time.

Eg: Consider three files x1, x2, x3 with record lengths 30, 20, 10.



This merge pattern is taking 90 comparisons



This merge pattern is taking 110 comparisons.

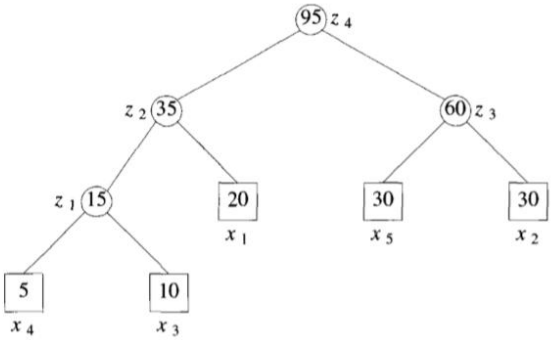
Greedy method to obtain an optimal merge pattern :

Selection criterion: since merging *n*-record and *m*-record files requires *n+m* record moves, the obvious choice is at each step merge the two smallest size files together.

Eg: no.of files = 5
(x1, x2, x3, x4, x5) sizes (20, 30, 10, 5, 30)
Merge x3 and x4 to get z1 ➔ |z1| = 15
Merge z1 and x1 to get z2 ➔ |z2| = 35
Merge x2 and x5 to get z3 ➔ |z3| = 60
Merge z2 and z3 to get z4 ➔ |z4| = 95

The total number of record moves = 205

It can be represented as a binary merge tree.



- The leaf nodes are drawn as squares and represent the given five files. These nodes are called external nodes.
- Remaining nodes are drawn as circles and are called internal nodes.
- Each internal node has exactly two children.
- The number in each node is the length (no.of records) of the file represented by that node.

- The external node x4 is at a distance of 3 from the root node z4. i.e., the records of file x4 are moved three times, once to get z1, once to get z2 and finally to get z4.
- Total number of record moves for the binary merge tree is $\sum_{i=1}^{n} d_i\, q_i$
  - d*i* is the distance from the root to the external node for file xi
  - q_i is the length of x_i
- This sum is the weighted external path length of the tree.

Algorithm to generate a 2-way merge tree :

```
node = record {
        node    *lchild, *rchild;
        integer weight;
}
Algorithm Tree(n)
{
        //list is a global list of n single node binary trees
        for i := 1 to n-1 do
        {
                pt := new node;
                pt->lchild := Least(list);
                pt->rchild := Least(list);
                pt->weight := pt->lchild->weight + pt->rchild->weight;
                insert(list, pt);
        }
        return (Least(list))
}
```

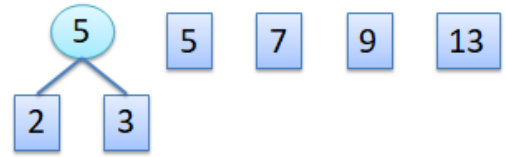
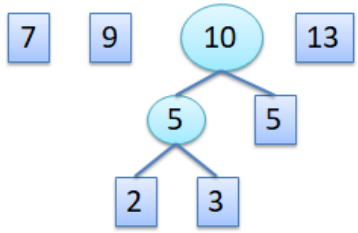
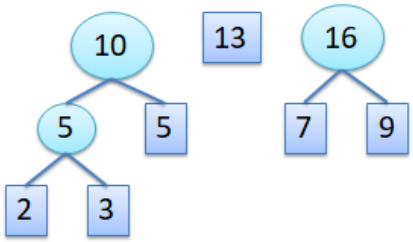
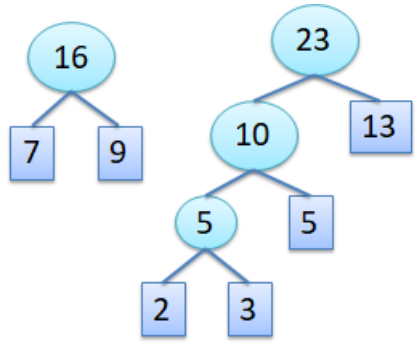
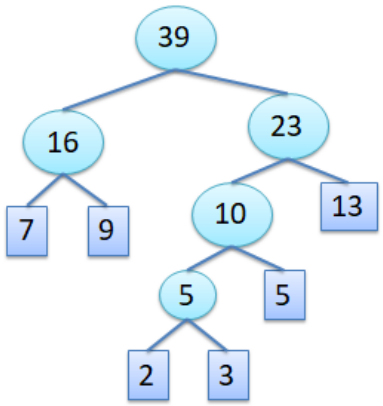
- Input to this algorithm is list of n trees. Each node in a tree has 3 fields lchild, rchild, weight.
- Initially, each tree in list has exactly one node.
- Least(list) function finds a tree in list whose root has least weight.
- Insert() function is used to insert a node into the list.

Analysis :

- Main for loop is executed *(n – 1)* times
- If list is kept in nondecreasing order according to weight value in the roots, then **Least(list)** requires O(1) time
- **Insert(list, t)** can be done in O(n) time.
- Hence, the total time taken is $O(n^2)$

Eg: Trace the algorithm for 6 files with lengths 2, 3, 5, 7, 9, 13

Iteration	List
Initially	2 3 5 7 9 13

1	
2	
3	
4	
5	

Optimal merge pattern

1. Merge files whose lengths are 2 and 3
2. Merge files whose lengths are 5 and 5
3. Merge files whose lengths are 7 and 9
4. Merge files whose lengths are 10 and 13
5. Merge files whose lengths are 16 and 23

Exercise :

Find an optimal binary merge pattern for ten files whose lengths are
28, 32, 12, 5, 84, 53, 91, 35, 3, 11

Single Source Shortest Paths

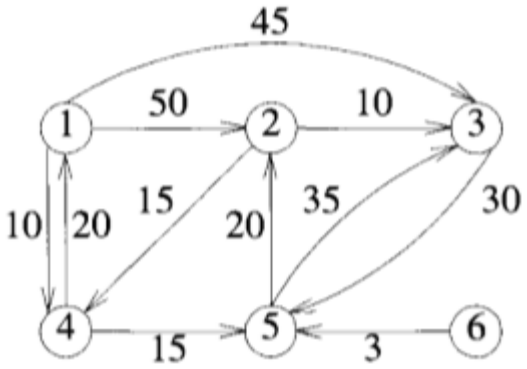
- **Problem** : Given a directed graph G = (V, E), a weighting function *cost* for the edges of G, and a source vertex v_0, The problem is to determine the shortest paths from v_0 to all the remaining vertices of G.
- It is assumed that all the weights are positive.
- The shortest path between v_0 and other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.
- A multistage solution must be conceived to formulate a greedy-based algorithm to generate shortest paths.
- Here the optimization measure is, each individual path must be of minimum length.
- The greedy way to generate the shortest paths from v_0 to the remaining vertices is to generate these paths in increasing order of the path length.
- First, a shortest path to the nearest vertex is generated, and then a shortest path to the second nearest vertex is generated, and so on.
- For eg : nearest vertex to (v0 = 1) is 4 cost[1, 4] = 10
  Second nearest vertex to (v0 = 1) is 5 distance = 25 path 1, 4, 5 is generated.
- Inorder to generate the shortest paths in this order, we need to determine
  1. The next vertex to which a shortest path must be generated &
  2. A shortest path to this vertex.
- Greedy algorithm to generate shortest paths is

```
Algorithm ShortestPaths(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 a digraph G with
// n  vertices. dist[v] is set to zero. G is represented by its cost adjacency matrix cost[l :n,1:n].

        for i := 1 to n do
        {
                S[i] := false;
                dist[i] := cost[v, i];
        }
        S[v] := true;
        for num := 1 to n-1 do
        {
                choose u from among those vertices not in S such that dist[u] is minimum;
                S[u] := false;
                for (each w adjacent to u with S[w]=false) do
                        //update the distances
                        if (dist[w] > dist[u] + cost[u,w]) then
                                dist[w] := dist[u] = cost[u, w];
        }
}
```

- Let S denote the set of vertices (including v_0) to which the shortest paths have already been generated.
- For w, not in S, let dist[w] be the length of the shortest path starting from v_0, going through only those vertices that are in S, and ending at w.

Example : Use algorithm ShortestPaths to obtain in non-decreasing order the lengths of the shortest paths from vertex 1to all remaining vertices in the digraph.



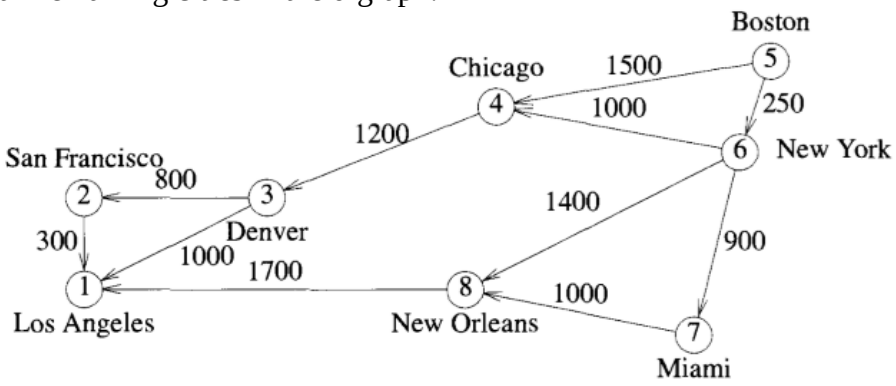
$V_0 = 1$

$$\text{Cost matrix} = \begin{bmatrix} 0 & 50 & 45 & 10 & \infty & \infty \\ \infty & 0 & 13 & 15 & \infty & \infty \\ \infty & \infty & 0 & \infty & 30 & \infty \\ 20 & \infty & \infty & 0 & 15 & \infty \\ \infty & 20 & 35 & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 \end{bmatrix}$$

Trace of the algorithm

Iteration	S	Vertex selected	Distance					
			[1]	[2]	[3]	[4]	[5]	[6]
Initially	--	--	0	50	45	10	∞	∞
1	{1}	4	0	50	45	10	25	∞
2	{1, 4}	5	0	45	45	10	25	∞
3	{1,4,5}	2	0	45	45	10	25	∞
4	{1,2,4,5}	3	0	45	45	10	25	∞
5	{1,2,3,4,5}	6	0	45	45	10	25	∞
6	{1,2,3,4,5,6}							

Example 2: Use algorithm ShortestPaths to obtain in non-decreasing order the lengths of the shortest paths from city Boston to all remaining cities in the digraph.



Length-adjacency matrix

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

Tracing

Iteration	S	Vertex selected	Distance							
			LA [1]	SF [2]	DEN [3]	CHI [4]	BOST [5]	NY [6]	MIA [7]	NO [8]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									