# Divide and Conquer

**Divide and Conquer for finding Counterfeit coin (exactly one coin):**

```
Algorithm CounterfeitCoin(Bunch, numberofcoins)
{
if (numberofcoins = 2) {
weigh the two coins;
if one of them weighs less that is counterfeit coin;
else, no counterfeit coin in the Bunch
}
else {
Divide Bunch into two halves Bunch1 and Bunch2;
if Bunch1 weighs less than Bunch 2, call CounterfeitCoin(Bunch1, numberofcoins / 2);
else, call CounterfeitCoin(Bunch2, numberofcoins / 2);
}
}
```

**Finding Largest with Divide and Conquer:**

```
Algorithm DandCLargest (a,k, j)
{
        if (k = j) Return a[n];
        else
        {
                mid := (k + j) /2;
                x1= DandCLargest (a,k,mid);
                x2= DandCLargest (a,mid+1,j);
                 if (x1 > x2) Return x1;
                 else Return x2;
         }
}

First Call for Recursion:
DandCLargest (a,1,n);
```

Divide and Conquer algorithm design works on the principle of dividing the given problem into smaller sub problems which are similar to the original problem. The sub problems are ideally of the same size.

The Divide and Conquer strategy can be viewed as one which has three steps. The first step is called **Divide** which is nothing but dividing the given problems into smaller sub problems which are identical to the original problem and also these sub problems are of the same size. The second step is called **Conquer** where in we solve these sub problems recursively. The third step

is called **Combine** where in we combine the solutions of the sub problems to get the solution for the original problem.

## Control Abstraction for Divide and Conquer Algorithms – General Method

```
Algorithm DandC(P)   {
  if Small(P) then return S(P);
  else {
        Divide P into smaller instances P1,P2,……..,Pk, k≥1;
        Apply DandC to each of these sub-problems;
        Return  Combine(DandC(P1),DandC(P2)...,DandC(Pk)),
  }
}
```

## Binary Search

• Binary Search takes a **sorted array** as the input
• It works by comparing the target (search key) with the middle element of the array and terminates if it equals, else it divides the array into two sub arrays and continues the search in left (right) sub array if the target is less (greater) than the middle element of the array.
Reducing the problem size by half is part of the **Divide** step in Binary Search and searching in this reduced problem space is part of the **Conquer** step in Binary Search. There is no **Combine** step in Binary Search.

### With Divide and Conquer (recursion)

```
Algorithm BinSrch(a,i,l,x) {
                 // Given an array a[I:l] of elements in non decreasing order, 1≤ i ≤ l,
                 // determine whether x is //present and if so, return j such that x =a[j]; else return 0.
        if (l=i) then {  // if Small(P)
                 if (x=a(i)) then return i; else return 0;
        }
        else {
                 // Reduce P into a smaller subproblem.
                mid :=[(i + l)/2 ];
                if (x=a[mid]) then return mid;
                else if (x<a[mid] then
                          return BinSrch (a,i, mid -1,x);
                     else return  BinSrch (a, mid +1,l,x);
        }
}
```

### Iterative Binary Search (non-recursive)

```
Algorithm BinSearch (a,n,x) {
        low :=1;high :=n;
        while (low ≤ high) do {
             mid :=(low +high)/2;
```

```
            if ( x<a[mid]) then high :=mid – 1;
            else if (x>a[mid]) then low :=mid + 1;
                 else return mid;
        }
        return 0;
}
```

**Theorem Algorithm BinSearch(a,n,x) works correctly.**
**Proof:**
We assume that all statements work as expected and that comparisons such as x>a[mid] are appropriately carried out.
Initially low = 1, high = n, n ≥ 0, and a[1]≤ a[2]≤..≤ a[n].
If n = 0, the while loop is not entered and 0 is returned.
Otherwise we observe that each time through the loop the possible elements to be checked for equality with x are a [low], a[low + 1]…, ..., a[mid],..., a[high).
If x = a[mid], then the algorithm terminates successfully.
Otherwise the range is narrowed by either increasing low to mid + 1 or decreasing high to mid — 1. Clearly this narrowing of the range does not affect the outcome of the search.
If low becomes greater than high, then x is not present and hence the loop is exited.

## Performance of Binary Search

**Theorem:-If n is in the range $[2^{k-1}, 2^k)$, then BinSearch makes at most k element comparisons for a successful search and either $k$ —1 or $k$ comparisons for an unsuccessful search. (In other words the time for a successful search is $O(\log n)$ and for an unsuccessful search is $(\log n)$.**

**Proof:** Consider the binary decision tree describing the action of BinSearch on $n$ elements. All successful searches end at a circular node whereas all unsuccessful searches end at a square node.
If $2^{k-1} \le n < 2^k$, then all circular nodes are at levels 1, 2,... , $k$ whereas all square nodes are at levels $k$ and $k + 1$ (note that the root is at level 1). The number of comparisons needed to terminate at a circular node on level $i$ is $i$ whereas the number of element comparisons needed to terminate at a square node at level $i$ is only $i$ — 1. The theorem follows.

## Merge Sort

Merge sort is yet another sorting algorithm which works on the Divide and Conquer design principle.
• Merge sort works by dividing the given array into two sub arrays of equal size
• The sub arrays are sorted independently using recursion
• The sorted sub arrays are then merged to get the solution for the original array.

The **breaking** of the given input array into two sub arrays of equal size is part of the **Divide** step. The recursive calls to sort the sub arrays are part of the **Conquer** step. The merging of the sub arrays to get the solution for the original array is part of the **Combine** step.

The basic operation in Merge sort is comparison and swapping. Merge Sort Algorithm calls it self recursively. Merge Sort divides the array into sub arrays based on the *position* of the elements whereas Quick Sort divides the array into sub arrays based on the value of the elements. Merge Sort requires an auxiliary array to do the *merging* (**Combine step**). The *merging* of two sub arrays, which are already sorted, into an auxiliary array can be done in **O(n)** where **n** is the total number of elements in both the sub arrays. This is possible because both the sub arrays are sorted.

```
Algorithm MergeSort(low, high) {
          // Small(P) is true if there is only one element to sort .
          // In this case  the list is already sorted.
        if (low < high) then {
          //If there are more than one element
            //Divide P into subproblems.
          mid := [(low + high)/2] ;
          // Solve the subproblems.
          MergeSort(low, mid);
          MergeSort (mid + 1, high);
          //  Combine the solutions.
          Merge(low, mid, high);
        }
}
```

**Merging in Merge Sort**

```
Algorithm Merge (low, mid, high) {
                // a[low:high] is a global array containing two sorted  subsets in
                //  a [low:mid] and in a [mid + 1 :high]. The goal is to merge these
               //  two sets  into a single set residing in a [low:high]. B[] is an
              //    auxiliary global array.
    h:=low; i:=low ;  j : =mid +1;
    while ((h≤ mid) and ( j≤high)) do {
        if (a[h]≤a[j] then {
           b[i] := a[h];h :=h+1;
        }
        else {
                    b[i] := a[j];j :=j+1;
        }
         i:= i+1;
    }
     if ( h>mid) then
```

```
            for k:=j to high do
                { b[i] : =a[k]; i:=i+1;  }
        else  for k :=h to mid do
                { b[i] := a[k];  i:= i+1; }
        for k: =low to high do a[k] :=b[k];
}
```

**Complexity of Merge Sort is O(n log n) and binary search is O(log n).**

**This can be proved by repeated substitution in the recurrence relations.**

Suppose (for simplicity) that $n = 2^k$ for some entire k. as $n=2^k$ $k = \log_2 n$

**Merge Sort:**

Let T(n) the time used to sort n elements. As we can perform separation and merging in linear time, it takes cn time to perform these two steps, for some constant c. So, recurrence relation is : $T(n) = 2T(n/2) + cn$.

In the same way: $T(n/2) = 2T(n/4) + cn/2$, so $T(n) = 4T(n/4) + 2cn$.

Going in this way ...

$T(n) = 2^m T(n/2^m) + mcn$, and

$T(n) = 2^k T(n/2^k) + kcn = nT(1) + cn\log_2 n = O(n \log n)$.

**Binary Search:**

Let T(n) the time used to search n elements. As we need to search only one of the halves, the Recurrence relation is : $T(n) = T(n/2) + c$

In the same way: $T(n/2) = T(n/4) + c$, so $T(n) = T(n/4) + 2c$.

Going in this way ...

$T(n) = T(n/2^m) + mc$, and

$T(n) = T(n/2^k) + kc = T(1) + kc = kc + 1 = O(\log n)$.

**QuickSort:**

Quick sort is one of the most powerful sorting algorithms. It works on the Divide and Conquer design principle. Quick sort works by finding an element, called the **pivot,** in the given input array and **partitions** the array into three sub arrays such that the left sub array contains all elements which are less than or equal to the pivot. The middle sub array contains the pivot. The

right sub array contains all elements which are greater than the pivot. Now, the two sub arrays, namely the left sub array and the right sub array are sorted recursively.
The **partitioning** of the given input array is part of the **Divide** step. The recursive calls to sort the sub arrays are part of the **Conquer** step. Since the sorted sub arrays are already in the right place there is no **Combine** step for the Quick sort.

```
1.   Algorithm QuickSort (p,q)
2.   // Sorts the elements a[p],…, a[q] which reside in the global
3.   // array a[1 ;n] into ascending order ; a[n+1] is considered to
4.   // be defined and must be ≥ all the elements in a[1 :n]
5.   {
6.       if (p < q) then // if there are more than one element
7.       {
8.           // divide P into two sub problems.
9.               j : = Partition (a,p,q +1);
10.                  // j is the position of the partitioning element.
11.          // Solve the subproblems.
12.             QuickSort (p,j -1),
13.             QuickSort   ( j +1,q)
14.          // there is no need for combining solutions.
15.      }
16.  }
```

**Algorithm for Sorting by partitioning.**

```
1.   Algorithm Partition (a,m,p)
2.   //Within a[m],a[m+1],….a[p-1] the elements are
3.   //rearranged in such a manner that if initially t = a[m],
4.   // then after completion a[q] = t for some q between m
5.   // and p-1, a[k]≤ t for m≤ k<q, and a [k] ≥ t
6.   // for q< k<p. q is returned. Set a[p] = ∞.
7.   {
8.        v :=a[m], I :=m; j : = p;
9.        repeat
10.       {
11.           repeat
12.                   i :=i+1;
13.           until (a[i]≥v);

14.           repeat
15.             j : = j – 1;
16.           until (a[j] ≤ v);

17.           if (I < i) then interchange (a,I,j);
```

```
18.          } until (I ≥ j);

19.          a[m] : = a[j]; a[j] : = v; return j ;
20.    }
```

```
1.      Algorithm Interchange ( a , i, j)
2.      //Exchange a[i] with a [j]
3.      {
4.           p : = a[i];
5.           a [i] : = a[j];  a[j] : p;
6.      }
```

---

**Algorithm** Partition the array a[m : p-1] about a [m]

| (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) | (10) | i | (p) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 65 | 70 | 75 | 80 | 85 | 60 | 55 | 50 | 45 | + ∞ | 2 | 9 |
| 65 | 45 | 75 | 80 | 85 | 60 | 55 | 50 | 70 | + ∞ | 3 | 8 |
| 65 | 45 | 50 | 80 | 85 | 60 | 55 | 75 | 70 | + ∞ | 4 | 7 |
| 65 | 45 | 50 | 55 | 85 | 60 | 80 | 75 | 70 | + ∞ | 5 | 6 |
| 65 | 45 | 50 | 55 | 60 | 85 | 80 | 75 | 70 | + ∞ | 6 | 5 |
| 60 | 45 | 50 | 55 | 65 | 85 | 80 | 75 | 70 | + ∞ | | |

# Greedy Algorithms

## Greedy algorithms – overview

Greedy design technique is primarily used in Optimization problems.
Optimization problems are problems where in we would like to find the best of all possible solutions. In other words, we need to find the solution which has the optimal (maximum or minimum) value satisfying the given constraints.

The Greedy approach helps in constructing a solution for a problem through a sequence of steps where each step is considered to be a partial solution. This partial solution is extended progressively to get the complete solution.

In the greedy approach each step chosen has to satisfy the constraints given in the problem. Each step is chosen such that it is the best alternative among all feasible choices that are available. The choice of a step once made cannot be changed in subsequent steps.

## Change making example:

Suppose, we want to make change for an amount 'A' using fewest no of currency notes. Assume the available denominations are Rs 1, 2, 5, 10, 20, 50, 100, 500, 1000.
To make a change for A=Rs 28, with the minimum number of notes, one would first choose a note of denomination Rs 20, 5, 2 and 1.

| Denomination table | | | | | |
|---|---|---|---|---|---|
| **for Rs 28** | | **for Rs 783** | | **for Rs 3799** | |
| 1000 X **0** | 0 | 1000 X | | 1000 X | |
| 500 X **0** | 0 | 500 X | | 500 X | |
| 100 X**0** | 0 | 100 X | | 100 X | |
| 50 X **0** | 0 | 50 X | | 50 X | |
| 20 X **1** | 20 | 20 X | | 20 X | |
| 10 X **0** | 0 | 10 X | | 10 X | |
| 5 X **1** | 5 | 5 X | | 5 X | |
| 2 X **1** | 2 | 2 X | | 2 X | |
| 1 X **1** | 1 | 1 X | | 1 X | |
| Total | **28** | Total | | Total | |

```
Algorithm change making(denom_value[ ], TargetAmount)
{
        // denom={1000, 500, 100, 50, 20, 10, 5, 2, 1}

        selected amount (SA) :=0;
        i :=1;
        while (SA < TargetAmount){
                if (SA + denom_value [i] <= TargetAmount ) {       // Select & Feasible

                        denom_select[i] ++;   // Union
```

```
                    SA := SA + denom_value[i];
                }
                else {
                    i++;
                }
        }
        Print denom _select
}
```

**Greedy Algorithm-General method**

In Greedy method the problems have **'n'** inputs called as candidate set, from which a subset is selected to form a solution for the given problem. Any subset that satisfies the given constraints is called a feasible solution. We need to find a feasible solution that maximizes or minimizes an objective function and such solution is called an optimal solution.

**In the above ex** currency notes denomination set { 1000 ….1000 ,500….500, 100….100, 50…50, 20…20,10…10,5…5,2..2,1…1}is candidate set.

**In the above ex** constraint is our solution make the exact target amount of cash. Hence, any feasible solution i.e. sum of selected notes should be equal to target amount.

**In the above ex** objective function is our solution should consist of the fewest number of currency notes. Hence, any optimal solution which is one of the feasible solutions that optimizes the objective function. There can be more than one optimal solution.

**Control Abstraction for Greedy General Method**

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

**Greedy method consists of 3 functions (steps).**

**1) Select:** it selects an input from array a[ ] (candidate set) and puts in the variable x.

**2) Feasible:** it is a Boolean function which checks whether the selected input meets the constraints or not.

**3) Union**: if the selected input i.e. 'x' makes the solution feasible, then x is included in the solution and objective function get updated.

**Characteristics of Greedy:**

1) These algorithms are simple and straightforward and easy to implement.
2) They take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future.
3) They work in stages and never reconsider any decision.

## Greedy Algorithms Applications

**Knapsack problem:**

**A thief robbing a store finds n items, the items each worth $v_i$ rupees and weights $w_i$ grams, where $v_i$ and $w_i$ are positive numbers. He wants to take as valuable load as possible but he can carry at most w grams in his knapsack(bag). Which item should he take?**

They are two types of knapsack problem.

**1) 0-1 knapsack problem:** Here the items may not be broken into smaller pieces, so thief may decide either to take an item or to leave to it(binary choice). It cannot be efficiently solved by greedy algorithm

**2) Fractional (General) Knapsack problem:** Here thief can take the fraction of items, meaning that the items can be broken into smaller pieces so that thief may be able to carry a fraction $x_i$ of item i. This can be solved easily by greedy.

If a fraction $x_i$ , $0 \le x_i \le 1$, of objects 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. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m.

Formally the problem can be stated as

$$\text{Maximize } \sum P_i x_i \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots..(1)$$

$$\text{Subject to } \sum W_i x_i \le m \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots..(2)$$

$$\text{And } 0 \le x_i \le 1,\ 1 \le i \le n \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots..(3)$$

The profit and weights are positive numbers. A feasible solution is any set $(x_1, x_2, \ldots\ldots\ldots\ldots\ldots x_n)$ satisfying (2) and(3). An optimal solution is feasible solution for which (1) is maximized.

**Eg**; consider the following instance of the knapsack problem.

n=3, m=20 , $(P_1, P_2, P_3) = (25, 24, 15)$ & $(w_1, w_2, w_3) = (18,15,10)$

Note that knapsack problem calls for select a subset of the objects hence fits the subset paradigm.

1) We can try to fill the knapsack by including the object with largest profit(greedy approach to the profit) .If an object under consideration does not fit, then a fraction of it is included to fit the knapsack. Object 1 has the largest profit value.$P_1$=25. So it is placed into the knapsack first. Then $x_1$=1 and a profit of 25 is earned. Only 2 units of knapsack capacity are left. Objects 2 has the next largest profit $P_2$=24. But $W_2$ =15 & it does not fit into the knapsack. Using $x_2$ =2/15 fills the knapsack exactly with the part of the object 2.

The method used to obtain this solution is termed a greedy method at each step, we chose to introduce that object which would increase the objective function value the most.

| (x1 , x2 , x3) | ∑ wi xi | ∑ pi xi |
|---|---|---|
| (1 , 2/15 , 0) | 20 | 28.2 |

This is **not an optimal solution.**

2)We apply greedy approach by choosing value per unit weight is as high as possible

| Item(n) | Value(p1,p2,p3) | Weight(w1,w2,w3) | Val/weight |
|---|---|---|---|
| 1 | 25 | 18 | 1.388 |
| 2 | 24 | 15 | 1.6 |
| 3 | 15 | 10 | 1.5 |

Here $\mathbf{p_2/w_2 > p_3/w_3 > p_1/w_1}$. Now the items are arranged into non increasing order of $p_i/w_i$. Second item is the most valuable item. We chose item 2 first. Item 3 is second most valuable item. But we cannot choose the entire item3 as the weight of item 3 exceeds the capacity of knapsack. We can take ½ of the third item. Therefore the solution is $x_1 = 0$, $x_2 = 1$, $x_3 = ½$ and maximum profit is $\sum p_i x_i = 0*25 + 1*24 + ½ * 15 = 31.5$

| $(x_1, x_2, x_3)$ | $\sum w_i x_i$ | $\sum p_i x_i$ |
|---|---|---|
| (1, 2/15, 0) | 20 | 28.2 |
| (0, 2/3, 1) | 20 | 31 |
| (0, 1, 1/2) | 20 | 31.5 |

If the items are already arranged in non increasing order of pi/wi , then the function greedy knapsack obtains solution corresponding to this strategy.

**Algorithm** Greedy Knapsack(a,n)
// Objects are sorted in the non-increasing order of p[i]/w[i]
**{**
**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; U:=U - w[i];
**}**

```
  if (i <= n) then x[i] := U / w[i];
}
```

**Analysis of Greedy Knapsack**

- If the items are already sorted into decreasing order of vi/wi, then time complexity is **O(n)**
- Therefore Time complexity including sort is **O(n log n)**

## Minimum Spanning Tree

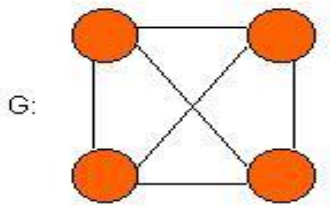
A tree is defined to be an undirected, acyclic and connected graph (or more simply, a graph in which there is only one path connecting each pair of vertices).

Assume there is an undirected, connected graph G. A spanning tree is a sub-graph of G, is a tree, and contains all the vertices of G. A minimum spanning tree is a spanning tree, but has weights or lengths associated with the edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum

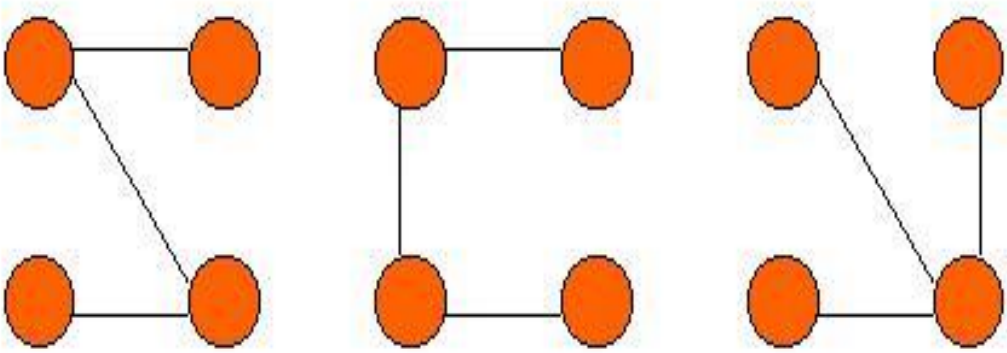
### Application of MST

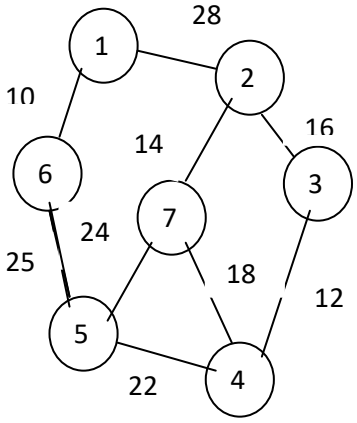
1) practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. MST can be used to determine the least costly paths with no cycles in this network, thereby connecting everyone at a minimum cost.

2) Another useful application of MST would be finding airline routes MST can be applied to optimize airline routes by finding the least costly paths with no cycles

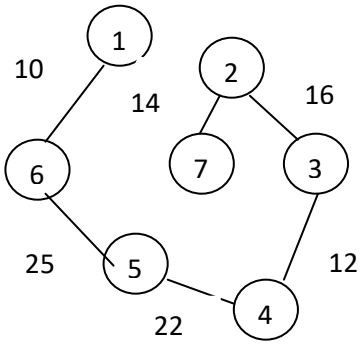


**Three (of the many possible) spanning trees from graph G**






(a)




(b)

**Prim's Algorithm (DJP algorithm,the Jarník algorithm, or the Prim-Jarník algorithm).**

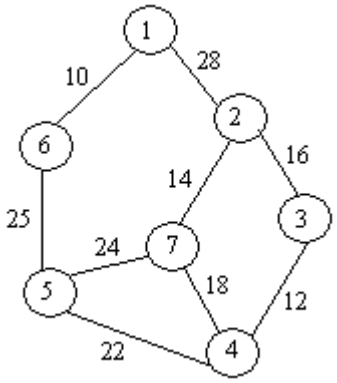
Prim's algorithm finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Steps

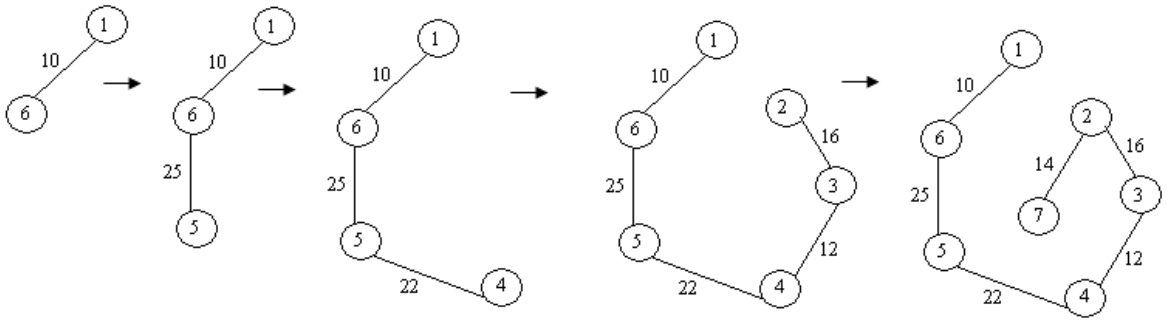
- Builds the tree edge by edge
- Next edge to be selected is one that result in a minimum increase in the sum of costs of the edges so far included
- Always verify that resultant is a tree

**Ex:**

Consider the connected graph given below




Minimum spanning tree using Prim's algorithm can be formed as follows.




**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 number or ∞ if no edges( i, j) exists,
//A minimum spanning tree is computed and stored as set of
//edges in the array t[1 :n -1,1:2], (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 of minimum cost in E;
        mincost :=cost[k,l];
        t[1,1] :=k;t[1,2] :=l;
        for i :=1 to n do  //Initialize near.
                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
        { // 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 the above algorithm is **$O(n^2)$.**

## Kruskal's Algorithm

Kruskal's algorithm is another algorithm that finds a minimum spanning tree for a connected weighted graph. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component).

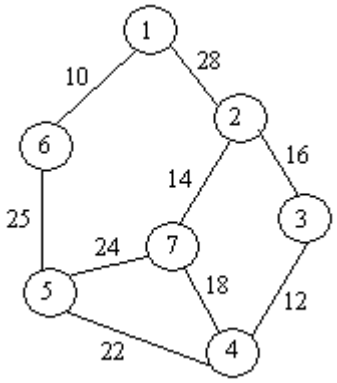
Kruskal's Algorithm builds the MST in forest. Initially, each vertex is in its own tree in forest. Then, algorithm considers each edge in turn, order by increasing weight. If an edge (*u*, *v*) connects two different trees, then (*u*, *v*) is added to the set of edges of the MST, and two trees connected by an edge (*u*, *v*) are merged into a single tree on the other hand, if an edge (*u*, *v*) connects two vertices in the same tree, then edge (*u*, *v*) is discarded. The resultant may not be a tree in all stages. But can be completed into a tree at the end.

```
t = EMPTY;
while ((t has fewer than n-1 edges) && (E != EMPTY))
{
    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
            add (v, w) to t;
    else
            discard (v, w);
}
```

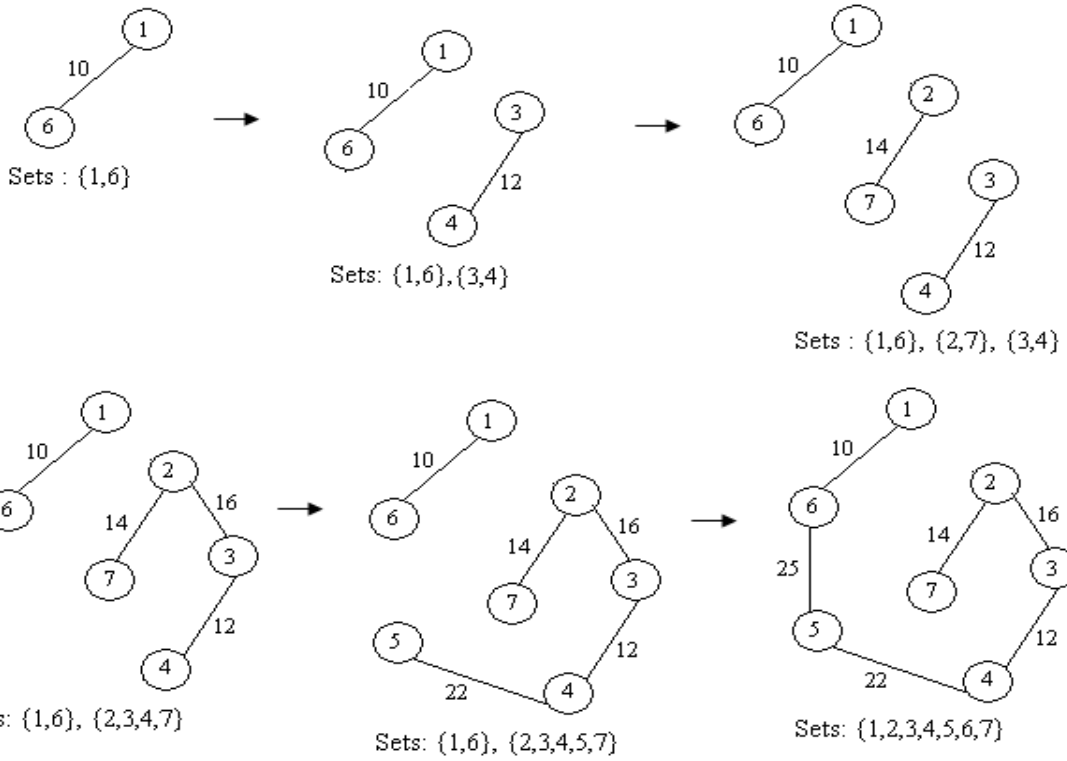
To check whether there exist a cycle, place all vertices in the same connected component of *t* into a set. Then two vertices *v* and *w* are connected in *t* then they are in the same set.

**Example:**

Consider the connected graph given below:




Minimum spanning tree using Kruskal's algorithm can be formed as given below.



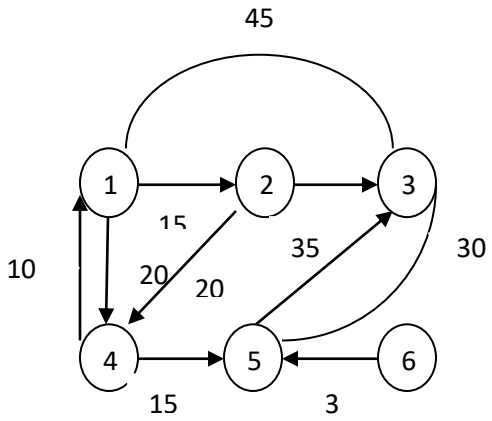

## Kruskal's Algorithm

```
Float kruskal (int E[ ][ ], float cost[ ][ ], int n, int t[ ][2])
{
    int parent[w];
    consider heap out of edge cost;
    for (i=1; i<=n; i++)
            parent[i] = -1;          //Each vertex in different set
    i=0;
    mincost = 0;
    while((i<n-1) && (heap not empty))
```

```
    {
        Delete a minimum cost edge (u,v) from the heap and re heapify;
        j = Find(u);    k = Find(v);            // Find the set
        if (j != k)
        {
            i++;
            t[i][1] = u;
            t[i][2] = v;
            mincost += cost[u][v];
            Union(j, k);
        }
        if (i != n-1)
            printf("No spanning tree \n");
        else
            return(mincost);
    }
}
```

Time complexity of the above algorithm is **O(n log n)**

## Greedy Algorithm for Single-source shortest paths to all other vertices

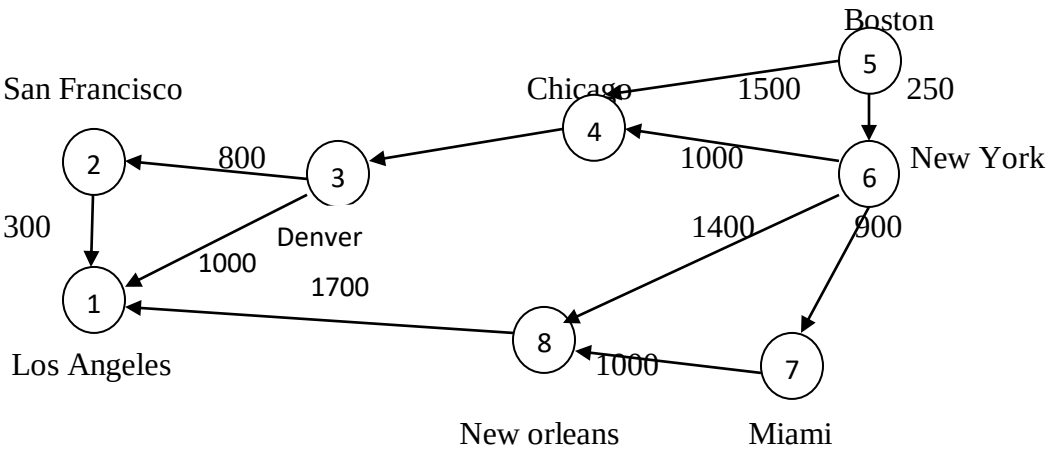



| Path | Length |
|---|---|
| 1) 1.4 | 10 |
| 2) 1.4.5 | 25 |
| 3) 1.4.5.2 | 45 |
| 4) 1.3 | 45 |

(b) Shortest path from 1

**Greedy algorithm to generate shortest paths**

```
Algorithm Shortest paths (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[1:n ,1;n]
{
     for i:=1 to n do
     {    // Initialize S.
          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 from among those vertices not
          in S such that dist[u] is minimum;
           S[n] :=true;//put u in S.
            for (each w adjacent to u with S[w] = false ) do
                  // Update distances.
                   if(dist[w] > dist[u] + cost[u,w])) then
                      dist[w] :=dist[u] + cost[u, w];
       }
}
```




(a) Digraph

| | 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 |

(a) Length – adjacency matrix