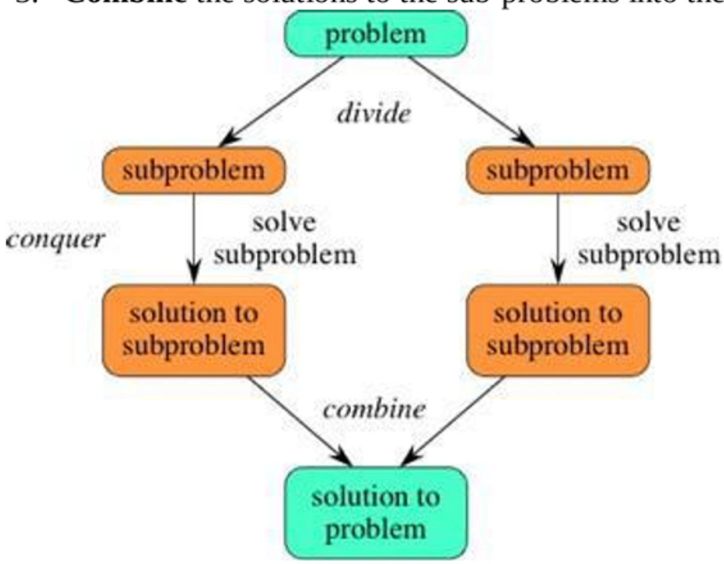
Module III

- **Divide & Conquer and Greedy Strategy**
  - **The Control Abstraction of Divide and Conquer**
  - **2-way Merge sort**
  - **Strassen's Algorithm for Matrix Multiplication-Analysis**

  - **The Control Abstraction of Greedy Strategy**
  - **Fractional Knapsack Problem**
  - **Minimum Cost Spanning Tree Computation- Kruskal's Algorithms – Analysis**
  - **Single Source Shortest Path Algorithm - Dijkstra's Algorithm-Analysis**

- **Divide and Conquer**
  - Divide and conquer algorithm is having three parts:
    1. **Divide** the problem into a number of sub-problems that are smaller instances of the same problem.
    2. **Conquer** the sub-problems by solving them recursively. If they are small enough, solve the sub-problems as base cases.
    3. **Combine** the solutions to the sub-problems into the solution for the original problem.



  - Control Abstraction: It is a procedure whose flow of control is clear but whose primary operations are specified by other procedure whose precise meanings are left undefined.
  - **Control Abstraction: Divide and Conquer**

```
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));
        }
}
```

    - If the given problem is small, return the result

- Otherwise, divide the problem into smaller instances $P_1, P_2, \ldots . P_k$
- Apply DAndC() to each of these sub-problems.
- Finally combine the results of all sub-problems.

- DAndC() can be described using the following recurrence relation:

$$T(n) = \begin{cases} g(n) & n \text{ is small} \\ T(n_1) + T(n_2) + \ldots\ldots + T(n_k) + f(n) & \text{Otherwise} \end{cases}$$

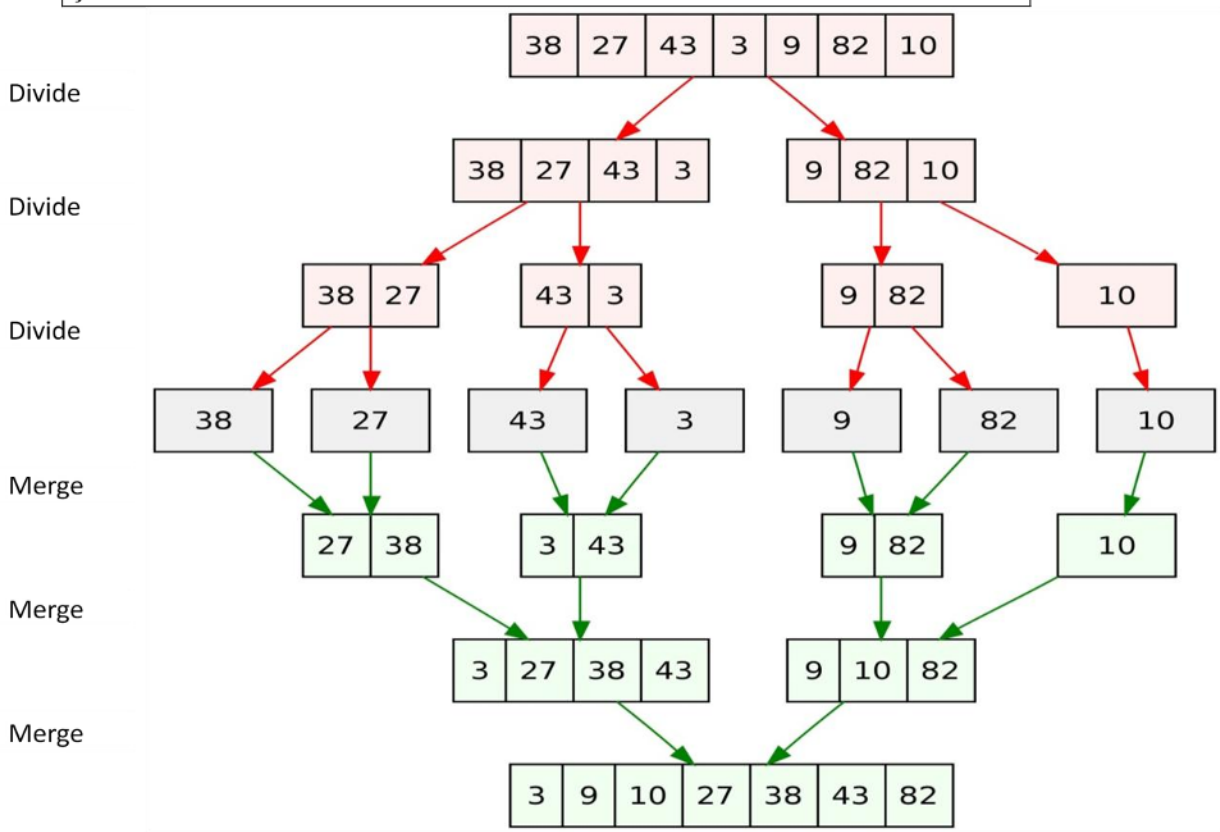
  - T(n): Time for divide and conquer on any input of size n
  - f(n): Complexity of dividing the problem and combining the results.
- Complexity of many divide and conquer algorithms are given by the following recurrence relation

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

- **2 Way Merge Sort**
  - Given a sequence of n elements a[l],…...a[n]. Split this array into two sets a[l],,.a.[n/2] and a[(n/2)+1],…a[n]. Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of n element.

```
Algorithm MergeSort(low, high)
{
        mid = (low + high )/2;
        MergeSort(low, mid);
        MergeSort(mid+1, high);
        Merge(low, mid, high);
}
Algorithm Merge(low, mid, high)
{
        i= low; x= low;  y= mid + 1;
        while((x ≤  mid) and (y ≤ high)) do
        {
                if ( a[x] ≤ a[y] ) then
                {
                        b[i] = a[x];
                        x = x+1;
                }
                else
                {
                        b[i] = a[y];
                        y = y+1;
                }
                i=i+1;
        }
        if( x ≤ mid) then
        {
```

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



- **Complexity**

  $$T(n) = \begin{cases} a & \text{if } n=1 \\ 2\,T(n/2) + cn & \text{Otherwise} \end{cases}$$

  a is the time to sort an array of size 1
  cn is the time to merge two sub-arrays
  2 T(n/2) is the complexity of two recursion calls

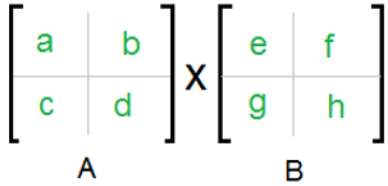
  $$\begin{aligned} T(n) &= 2\,T(n/2) + c\,n \\ &= 2(2\,T(n/4)+c(n/2)) + c\,n \end{aligned}$$

$= 2^2T(n/2^2) + 2\,c\,n$
$= 2^3T(n/2^3) + 3\,c\,n$
. . . .
$= 2^kT(n/2^k) + k\,c\,n$ [Assume that $2^k = n \rightarrow k = \log n$]
$= n\,T(1) + c\,n\log n$
$= a\,n + c\,n\log n$
$=$ **O(n log n)**

Best Case, Average Case and Worst Case Complexity of Merge Sort **= O(n log n)**

- **Divide and Conquer Matrix Multiplication**
  - Native matrix multiplication complexity = $O(n^3)$
  - **Divide and Conquer Matrix Multiplication Algorithm**
    1. We have to compute the product of 2 nxn matrices A and B.
    2. Assume that n is the power of 2. That is $n=2^k$
       If n is not a power of 2, then enough rows and columns of 0's can be added to both A and B so that the resulting dimensions are the power of two.
    3. Then partition A and B into 4 square matrices, each of size n/2 x n/2
    4. AB can be computed using the formula
       $$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
       $$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
       $$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
       $$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$
    5. If n=2, these formulas are computed using a multiplication operation for the elements of A and B
    6. If n>2, the elements of C can be computed using matrix multiplications and addition operations applied to the matrices of size n/2 x n/2
    7. This algorithm will continue applying itself to smaller sized sub-matrices until n becomes suitably small(n=2) so that the product is computed directly.
  - **Complexity**
    - For multiplying two matrices of size n x n, we make 8 recursive calls above, each on a matrix with size n/2 x n/2.
    - Addition of two matrices takes $O(n^2)$ time.
    - Time complexity $= 8\,T(n/2) + O(n^2)$
      $=$ $\mathbf{O(n^3)}$ [By Master's Theorem]

- **Strassen's Matrix Multiplication**
  - **Algorithm**
    1. A and B are the matrices with dimension nxn
    2. If n is not a power of 2, then enough rows and columns of 0's can be added to both A and B so that the resulting dimensions are the power of two.
    3. Partition A and B in to 4 square matrices of size n/2 x n/2



a, b, c and d are sub-matrices of A, of size n/2 x n/2
e, f, g and h are sub-matrices of B, of size n/2 x n/2

4. Compute 7 n/2 x n/2 matrices
   $P_1 = a (f – h)$
   $P_2 = h (a + b)$
   $P_3 = e (c + d)$
   $P_4 = d (g – e)$
   $P_5 = (a + d) (e + h)$
   $P_6 = (b – d)(g + h)$
   $P_7 = (a – c) (e + f)$

   It requires 7 matrix multiplications and 10 matrix additions and subtractions
5. Then compute C

   $$C = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$$

   $C_1 = P_4 + P_5 + P_6 – P_2$
   $C_2 = P_1 + P_2$
   $C_3 = P_3 + P_4$
   $C_4 = P_1 – P_3 + P_5 – P_7$

- **Complexity**
  - For multiplying two matrices of size n x n, we make 7 matrix multiplications and 10 matrix additions and subtractions
  - Addition/Subtraction of two matrices takes $O(n^2)$ time.
  - Time complexity $= 7\,T(n/2) + O(n^2)$
    $= O(n^{\log 7})$ = $\mathbf{O(n^{2.81})}$ [By Master's Theorem]

  - $$\mathbf{T(n) = \begin{cases} b & \text{if } n<2 \\ 7\,T(n/2) + c\,n^2 & \text{Otherwise} \end{cases}}$$

$T(n) = 7\,T(n/2) + c\,n^2$
$= 7^2T(n/2^2) + 7\,c\,n^2/4 + c\,n^2$
$= 7^3T(n/2^3) + 7^2\,c\,n^2/4^2 + 7\,c\,n^2/4 + c\,n^2$
……………………………………………..
$= 7^kT(n/2^k) + (7^{k-1}/4^{k-1})\,c\,n^2 + \ldots\ldots + (7/4)\,c\,n^2 + c\,n^2$
$= 7^kT(n/2^k) + [1+(7/4) + \ldots\ldots + (7^{k-1}/4^{k-1})\,]\,c\,n^2$
$\leq 7^kT(n/2^k) + [1+(7/4) + \ldots\ldots\ldots\ldots]\,c\,n^2$
$= 7^kT(n/2^k) + [1/(1-(7/4))]\,c\,n^2$
$= 7^{\log n}T(1) -[4/3]\,c\,n^2$ [Assume that $n/2^k = 1 \rightarrow k = \log n$]
$= n^{\log 7}O(1) -[4/3]\,c\,n^2$
$\mathbf{= O(n^{\log 7}) = O(n^{2.81})}$

- **Example**
  1. Multiply the following two matrices using Strassen's Matrix Multiplication Algorithm

$$A = \begin{bmatrix} 6 & 8 \\ 9 & 7 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 5 \\ 3 & 6 \end{bmatrix}$$

- **Greedy Strategy**
  - **Control Abstraction**

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

- Select() selects an input from the array a[] and remove it. The selected input value is assigned to x.
- Feasible() is a Boolean valued function that determines whether x can be included into the solution subset.
- Union() combines x with the solution and updates the objective function.

- **Fractional Knapsack Problem**
  - We are given with *n* objects and a knapsack(or bag) of capacity *m*. The object *i* has weight W_i and profit P_i. If a fraction X_i is placed into the knapsack, then a profit P_iX_i is obtained. The objective is to obtain an optimal solution of the knapsack that maximizes the total profit earned.
  - The total weight of all the chosen objects should not be more than *m*.
  - Fractional knapsack problem can be stated as

Maximize $\sum_{i=1}^{n} PiXi$ ——— (1)

Subject to $\sum_{i=1}^{n} WiXi \le m$ ——— (2)

$0 \le Xi \le 1$ and $1 \le i \le n$ ——— (3)

The profits and weights are positive numbers.
  - A feasible solution is one that satisfies equation 2 and 3.
  - An optimal solution is a feasible solution that satisfies equation 1.
  - In greedy strategy we are arranging the objects in the descending order of profit/weight.
  - **Algorithm**

```
Algorithm GreedyKnapsack(m, n)
//p[1:n] is the profits and w[1:n] is the weights of n objects such that p[i]/w[i] ≥ p[i+1]/w[i+1].
{
        for i= 1 to n do
                x[i] = 0.0;          // x[1:n] is the solution vector
        U = m;                       // m is the knapsack capacity
```

```
        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];
}
```

- **Time Complexity**
  - The for loop will execute maximum n times. So the time complexity = **O(n)**

- **Example**
  1. Find the optimal solution for the following fractional Knapsack problem. n=7, m=15, P={10, 5, 15, 7, 6, 18, 3} and W={2, 3, 5, 7, 1, 4, 1}
     - Arrange the objects in the descending order of profit/weight

       i ={ 1, 2, 3, 4, 5, 6, 7}
       P ={ 10, 5, 15, 7, 6, 18, 3}
       W ={ 2, 3, 5, 7, 1, 4, 1}
       Pi/Wi ={ 5, 1.66, 3, 1, 6, 4.5, 3}

       Now the i, P and W arrays are

       i ={5, 1, 6, 3, 7, 2, 4}
       P ={6, 10, 18, 15, 3, 5, 7}
       W ={1, 2, 4, 5, 1, 3, 7}

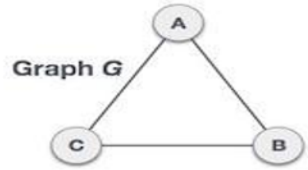
       Initially U=m=15

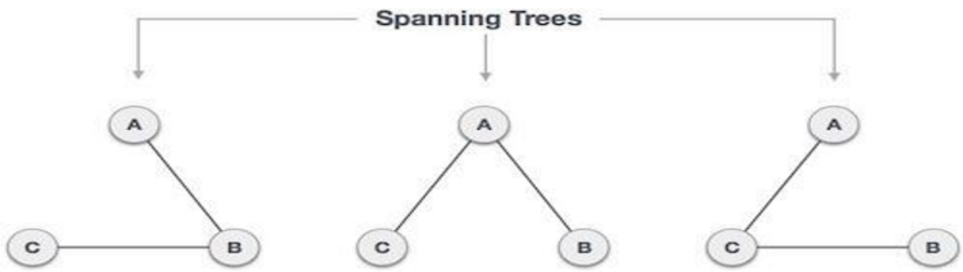
Item	**Pi**	**Wi**	**Xi**	**U = U-Wi**
5	6	1	1	14
1	10	2	1	12
6	18	4	1	8
3	15	5	1	3
7	3	1	1	2
2	5	3	2/3	0
4	7	7	0	0

     - Total weight of the chosen objects are
       $\sum_{i=1}^{n} WiXi$ = 2x1 + 3x2/3 + 5x1 + 7x0 +1x1 + 4x1 + 1x1 = 15
     - Profit earned is $\sum_{i=1}^{n} PiXi$ = 10x1 + 5x2/3 + 15x1 + 7x0 + 6x1 + 18x1 + 3x1 = **55.33**
     - Solution vector **X={1, 2/3, 1, 0, 1, 1, 1}**

- **Examples**
  1. Find the optimal solution for the following fractional Knapsack problem. Given number of items(n)=4, capacity of sack(m) = 60, W={40,10,20,24} and P={280,100,120,120}
  2. Find an optimal solution to the fractional knapsack problem for an instance with number of items 7, Capacity of the sack W=15, profit associated with the items (p1,p2,…,p7)= (10,5,15,7,6,18,3) and weight associated with each item (w1,w2,…,w7)= (2,3,5,7,1,4,1).

- **Spanning Trees**
  - A spanning tree is a subset of undirected connected Graph G=(V,E), which has all the vertices covered with minimum possible number of edges.





- **Properties of Spanning Tree**
  - A connected graph G can have more than one spanning tree.
  - All possible spanning trees of graph G, have the same number of edges and vertices.
  - The spanning tree does not have any cycle (loops)
  - Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**
  - Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**
  - Spanning tree has **n-1** edges, where **n** is the number of nodes.
- Maximum number of Spanning Trees of a graph with n nodes
  - Complete Graph: $\mathbf{n^{n-2}}$
  - Other Graphs
    1. Create Adjacency Matrix for the given graph.
    2. Replace all the diagonal elements with the degree of nodes.
    3. Replace all non-diagonal 1's with -1.
    4. Total number of spanning tree for that graph = Co-factor for any element in that matrix.
- **Minimum Spanning Tree (MST)**
  - In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.
  - In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.
  - Minimum Spanning-Tree Algorithms
    - Prim's Algorithm
    - Kruskal's Algorithm
- **Application of Spanning Tree**
  - Civil Network Planning
  - Computer Network Routing Protocol
  - Cluster Analysis
  - Handwriting Recognition
  - Image Segmentation

- **Examples**
  1. Write the total number of spanning trees possible for a complete graph with 6 vertices.

2. Consider a complete undirected graph with vertex set {0, 1, 2, 3, 4}. Entry Wij in the matrix W below is the weight of the edge {i, j}. What is the minimum possible weight of a spanning tree T in this graph such that vertex 0 is a leaf node in the tree T?

$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$

3. Let (u,v) be a minimum-weight edge in a graph G. Show that (u,v) belongs to some minimum spanning tree of G.
   - Suppose that T is a Minimum Spanning Tree, which does not include the smallest edge, E.
   - Add E to T. Now a circle C is formed.
   - This graph will remains connected if an edge is removed from the circle C.
   - So remove an edge E'(except E) from C which also belongs to T
   - This operation would result a new spanning tree whose weight is <= weight of T.
   - We have a contradiction. Hence, proved.
4. Let G be a weighted undirected graph with distinct positive edge weights. If every edge weight is increased by same value, will the minimum cost spanning tree change. Justify your answer
   - **The Minimum Spanning Tree doesn't change**. In Kruskal's algorithm, we will sort the edges first. IF we increase all weights, then order of edges won't change. So, MST does not change.

- **Minimal Cost Spanning Tree Computation**
  - **Kruskal's Algorithm.**
    - Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree.
    - Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree
    - In this algorithm, the edges of the graph are considered in the increasing order of cost.
    - If the selected edge will form a cycle, then discard it.
    - This selection process continues until there are n-1 edges.

```
Algorithm Kruskals(E, cost, n, t)
{
        Construct a heap out of edge costs using Heapify();
        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 and reheapify using Adjust();
                j = Find(u);
                k = Find(v);
                if j ≠ k then
```

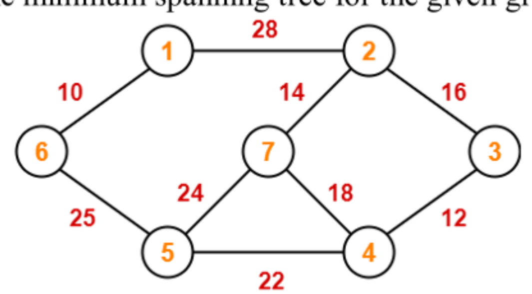
```
                {
                        i = i+1;
                        t[i, 1] = u;        t[i, 2] = v;
                        mincost = mincost + cost[u, v];
                        Union(j, k);
                }
        }
        if i ≠ n-1 then
                Write ("No Spanning Tree");
        else
                return mincost;
}
```

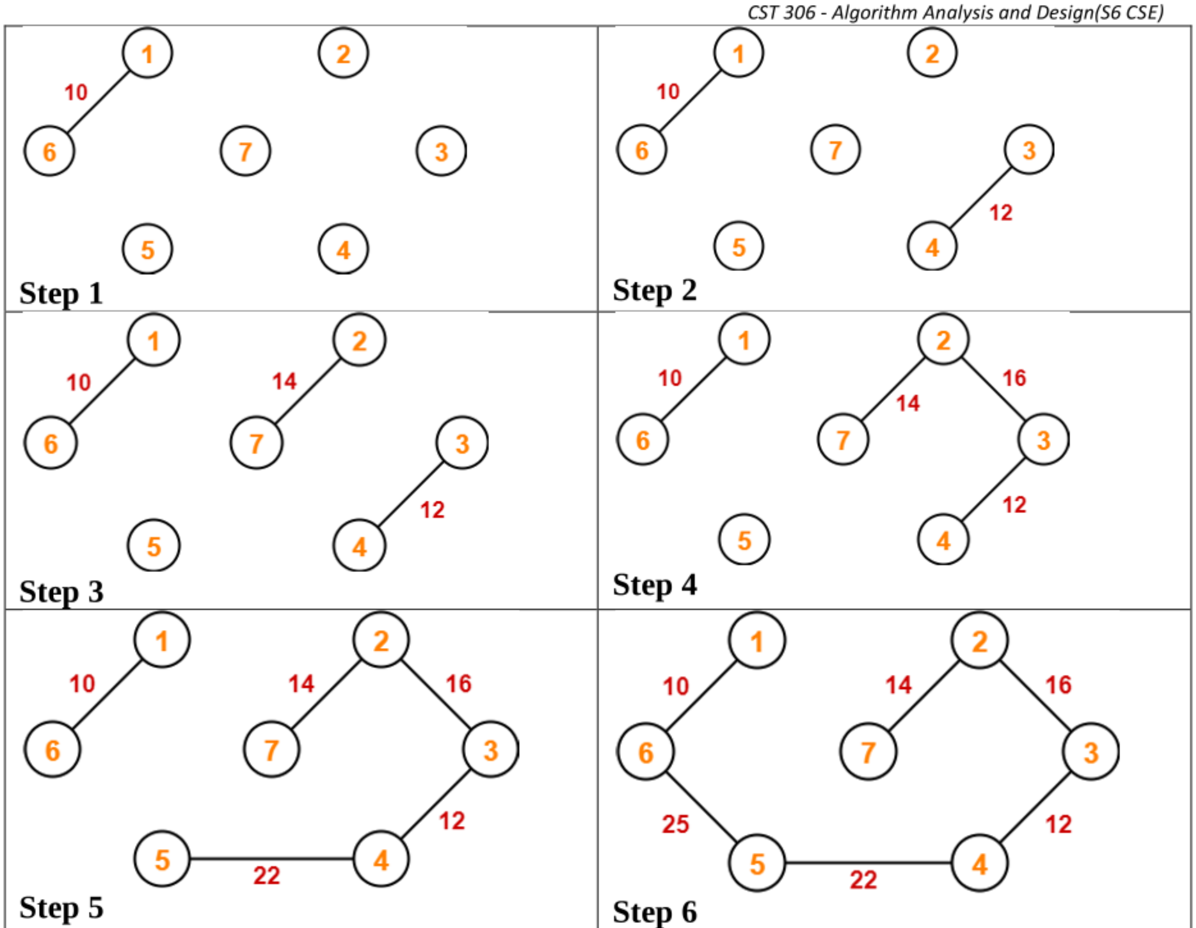
E is the set of edges and n is the number of vertices in G.
cost[u,v] is the cost of edge (u, v).
t is the set of edges in the minimum cost spanning tree.
The final cost is returned

- Heapify() is used to construct a minheap based on the edge cost of G.
- Adjust() is used to reconstruct a minheap if there is a deletion occurs.
- Initially all vertices are belongs to different sets. Find() returns the set number of that particular vertex. j and k are the set number of vertex u and v respectively.
- If j=k means vertex u and v are belongs to the same set. Inclusion of (u, v) should definitly form a cycle. So discard it.
- If j≠k means vertex u and v are belongs to different set. Inclusion of (u, v) will not form a cycle. So add it to the minimum spanning tree edge list.
- Finally there are n-1 edges, then retrun it. Otherwise there is no spanning tree.

- **Complexity**
  - The edges are maintained as a minheap, then the next edge to consider can be obtained in O(log |E|) time.
  - Construction of heap itself takes O(|E|) time.
  - Overall complexity of Kruskal's algorithm is **O(|E| log|E|).**

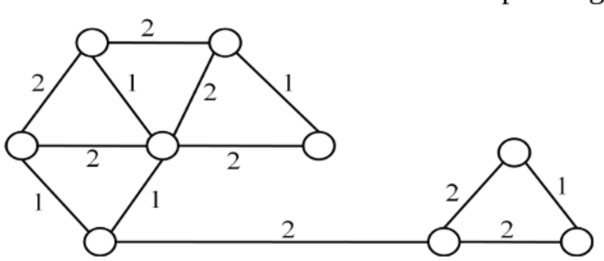
- **Example**
  - Construct the minimum spanning tree for the given graph using Kruskal's Algorithm



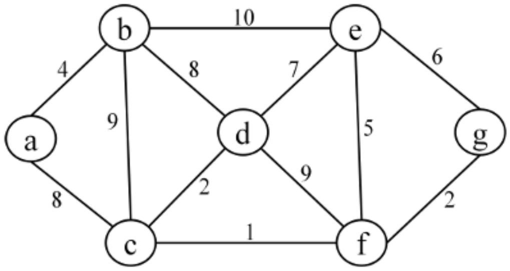


This is the minimum cost spanning tree.
mincost= 10 + 25 + 22 + 12 + 16 + 14 = 99

- **Examples**
  1. Find the number of distinct minimum spanning trees for the weighted graph below



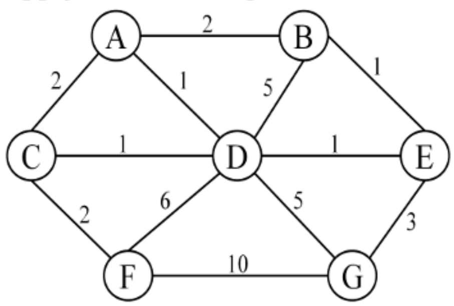
  2. Consider a weighted complete graph G on the vertex set $\{v_1, v_2, \ldots, v_n\}$ such that the weight of the edge (v_i, v_j) is $2|i-j|$. Find the weight of a minimum spanning tree of G.
  3. An undirected graph G=(V, E) contains n (n > 2) nodes named $v_1, v_2, \ldots, v_n$. Two vertices v_i, v_j are connected if and only if $0 < |i – j| <= 2$. Each edge (v_i, v_j) is assigned a weight i + j. What will be the cost of the minimum spanning tree (as a function of n) of such a graph with n nodes?
  4. Apply Kruskal’s algorithm on the graph given below.



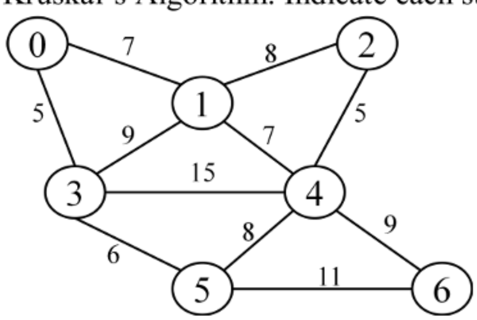
5. Consider a complete undirected graph with vertex set {0, 1, 2, 3, 4}. Entry wij in the matrix W below is the weight of the edge {i, j}. What is the Cost of the Minimum Spanning Tree T using Kruskal's Algorithm in this graph such that vertex 0 is a leaf node in the tree T?

$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$

6. Apply Kruskal's algorithm on the following graph. Let A be the source vertex



7. Compute the Minimum Spanning Tree and its cost for the following graph using Kruskal's Algorithm. Indicate each step clearly



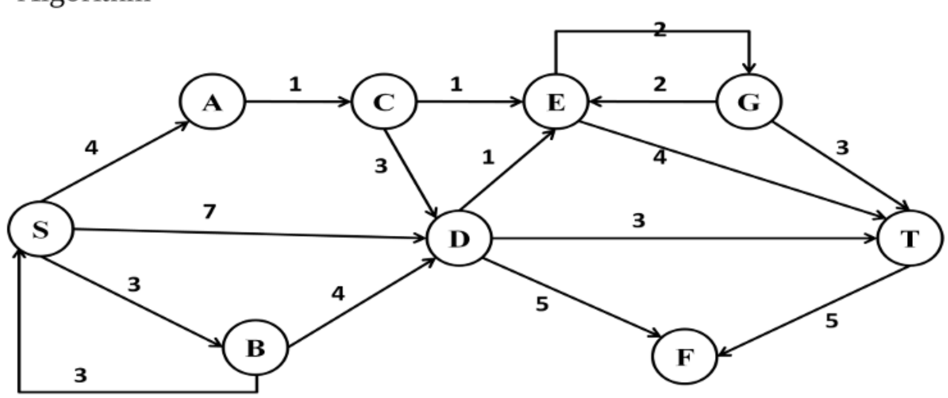
- **Single Source Shortest Path Algorithms**
  - The shortest path problem is the problem of finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.
  - Different shortest path problems are:
    - **Single Source Shortest Path Problem**:
      - Given a connected weighted graph G=(V,E), find the shortest path from a given source vertex s to every other vertices (V-{s}) in the graph.
      - The weight of any path(**w(p)**) is the sum of the weights of its constituted edges.
      - The weight of the shortest path from u yo v = **min{w(p): p is a path from u to v}**

- **Single Destination Shortest Path Problem**: To find shortest paths from all vertices in the directed graph to a single destination vertex *v*
- **All Pairs Shortest Path Problem**: To find shortest paths between every pair of vertices in the graph

- Single Source Shortest Path Algorithms are:
  - Dijkstra's Algorithm
  - Bellman Ford Algorithm

- **Dijkstra's Algorithm**
  - Given a graph and a source vertex *S* in graph, find shortest paths from *S* to all vertices in the given graph.
  - **Algorithm Dijkstra(G,W, S)**
    1. For each vertex v in G
       1.1 distance[v] = infinity
       1.2 previous[v] = Null
    2. distance[S] = 0
    3. Q = set of vertices of graph G
    4. While *Q is not empty*
       4.1 u = vertex in Q with minimum distance
       4.2 remove u from Q
       4.3 for *each neighbor v of u which is still in Q*
          4.3.1 alt = distance[u] + W(u,v)
          4.3.2 if alt < distance[v]
             4.3.2.1 distance[v] = alt
             4.3.2.2 previous[v] = u
    5. Return distance[], previous[]
  - **Complexity**
    - The complexity mainly depends on the implementation of Q
    - The simplest version of Dijkstra's algorithm stores the vertex set *Q* as an ordinary linked list or array, and extract-minimum is simply a linear search through all vertices in *Q*. In this case, the running time is $O(E + V^2)$ = $\mathbf{O(V^2)}$
    - Graph represented using adjacency list can be reduced to **O(E log V)** with the help of binary heap.

- **Examples**
  1. Is it possible to find all pairs of shortest paths using Dijkstra's algorithm? Justify
  2. Find the shortest path from s to all other vertices in the following graph using Dijkstra's Algorithm



3. Let G be a weighted undirected graph with distinct positive edge weights. If every edge weight is increased by same value, will the shortest path between any pair of vertices change. Justify your answer
   - **The shortest path may change**.
     - There may be different paths from s to t.
     - Let shortest path(Path-1) be of cost 15 and has 5 edges.
     - Let there be another path(Path-2) of cost 25 and has 2 edges.
     - All edge costs are increased by 10.
     - Path-1 cost is increased by 5*10 and becomes 15 + 50=65.
     - Path-2 cost is increased by 2*10 and becomes 25 + 20=45
     - Now Path-1 cost > Path-2 cost
     - So the shortest path may change.
4. In a weighted graph, assume that the shortest path from a source ‘s’ to a destination ‘t’ is correctly calculated using a shortest path algorithm. Is the following statement true? If we increase weight of every edge by 1, the shortest path always remains same. Justify your answer with proper example.