DAA MID-1

1) **Define an algorithm. What are the different characteristics of an algorithm?**

- An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.
- All algorithms must satisfy the following criteria:
- Input
  -An algorithm has zero or more inputs, taken from a specified set of objects.
- Output
  -An algorithm has one or more outputs, which have a specified relation to the inputs.
- Definiteness
  -Each step must be precisely defined; Each instruction is clear and unambiguous.
- Finiteness
  -The algorithm must always terminate after a finite number of steps.
- Effectiveness
  -All operations to be performed must be sufficiently basic that they can be done exactly and in finite length.

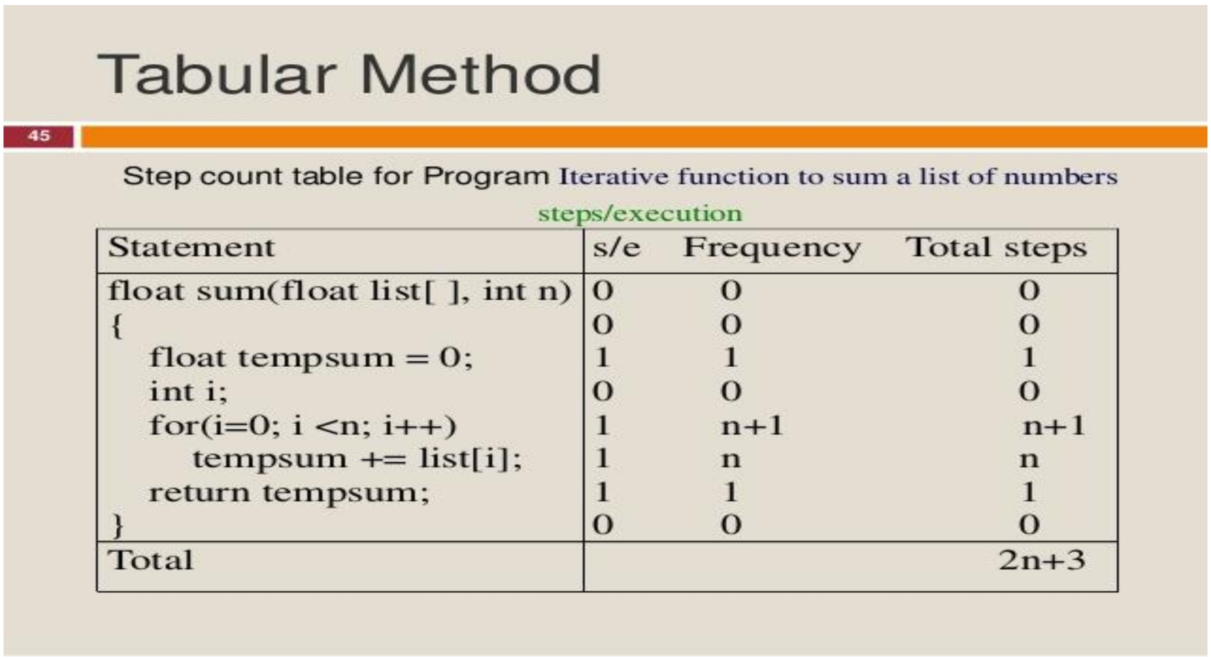
2. Define time complexity. Discuss the step count method for finding time complexity of an algorithm

- Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion.
- Time requirements can be defined as a numerical function T(n).
- Methods for finding time complexity of an algorithm :
- 1.step count
- 2.operation count
- 3.asymptotic analysis
- 3.recurrence relations
- 5.amortized analysis

Step count

- In this method, we count number of times one instruction is executing. From that we will try to find the complexity of the algorithm.
- Algorithm sum(a[], n)
- {
- sum = 0; // 1 step * 1
- for (i = 0; i < n; i++) // 1 step * (n+1)
- sum += a[i]; // 1 step * N
- return sum; // 1 step * 1
- }

- T(n)=1+n+1+n+1=2n+3=O(n)



Tabular Method

45

Step count table for Program Iterative function to sum a list of numbers

steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

3)Explain about Asymptotic notations used for algorithm analysis with examples

- Asymptotic analysis is the process of calculating the running time of an algorithm by considering the rate of growth of an algorithm.
- In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time).
- We calculate, how the time taken by an algorithm increases with the input size.
- Using asymptotic analysis, we can express the best case, average case and worst case time complexity of an algorithm.
- Big Oh Notation(O)
- The O(n) is the way to express the upper bound of an algorithm's running time.
- It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete.
-
- If f(n) and g(n) are two functions then
- f(n) = O(g(n)) if there exists c > 0 and n0 such that f(n) ≤ c.g(n) for all

n > n0.

Example:

- 1. 3n+2=O(n)

as 3n+2≤4n **for** all n≥2

- 2. 3n+3=O(n)

as $3n+3 \leq 4n$ **for** all $n \geq 3$

Omega Notation(Ω)

- The Ω(n) is the way to express the lower bound of an algorithm's running time.
- It measures the best case time complexity or best amount of time an algorithm can possibly take to complete.
- If f(n) and g(n) are two functions then
- f(n) = Ω(g(n)) if there exists c > 0 and n0 such that f(n) >=c.g(n) for all

n > n0.

Example:

f(n) = 3n + 2
g(n) = n

3n + 2 >= C n for all values of **C = 1** and **n >= 1**.

3n + 2 = Ω(n)

Theta Notation (θ)

- The θ(n) is the way to express both the lower bound and upper bound of an algorithm's running time.
- If f(n) and g(n) are two functions then f(n)= θ(g(n))

if there exists c1,c2 > 0 and n0 such that

c1* g(n) ≤f(n) ≤ c2*g(n) for all n > n0.

Example:

f(n) = 3n + 2
g(n) = n

C_1 n <= 3n + 2 <= C_2 n for all values of C_1 = 1, C_2 = 4 and n >= 2

3n + 2 = Θ(n)

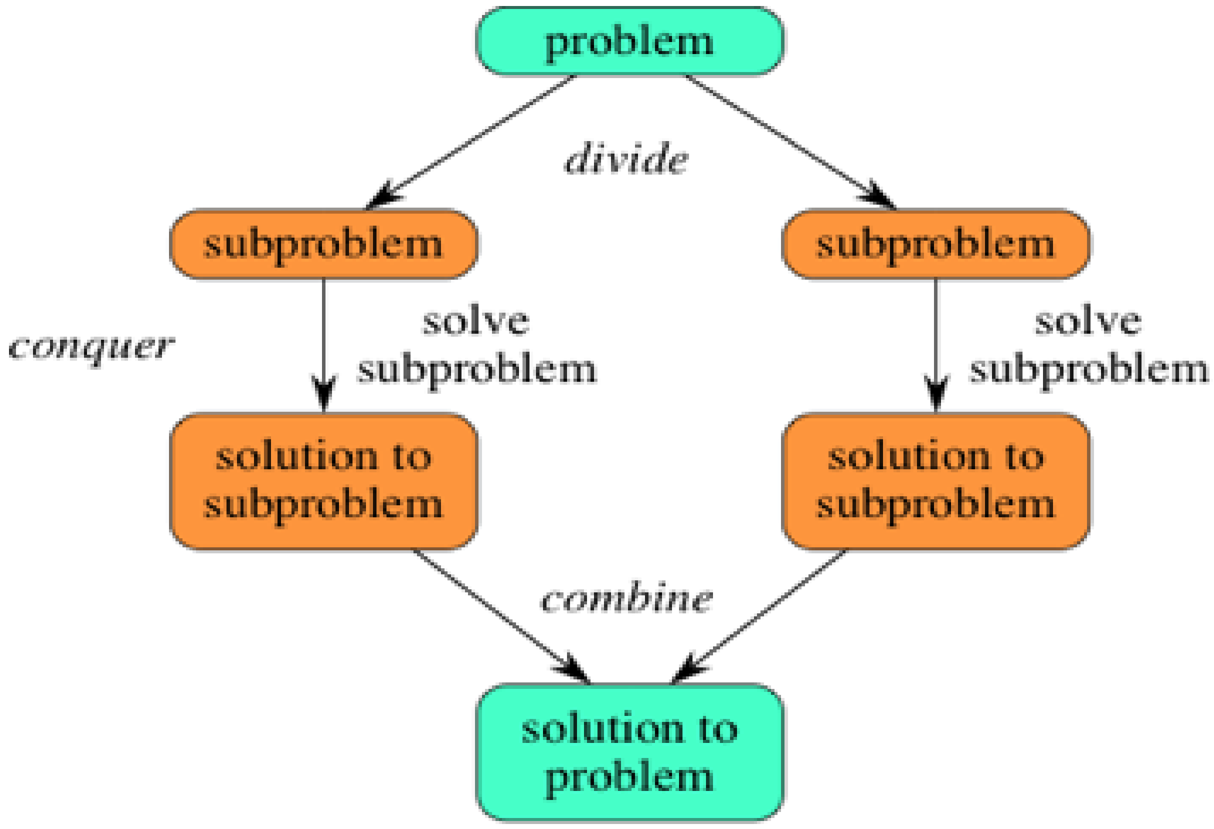
4. Discuss the divide and conquer general method.

Divide-and-Conquer

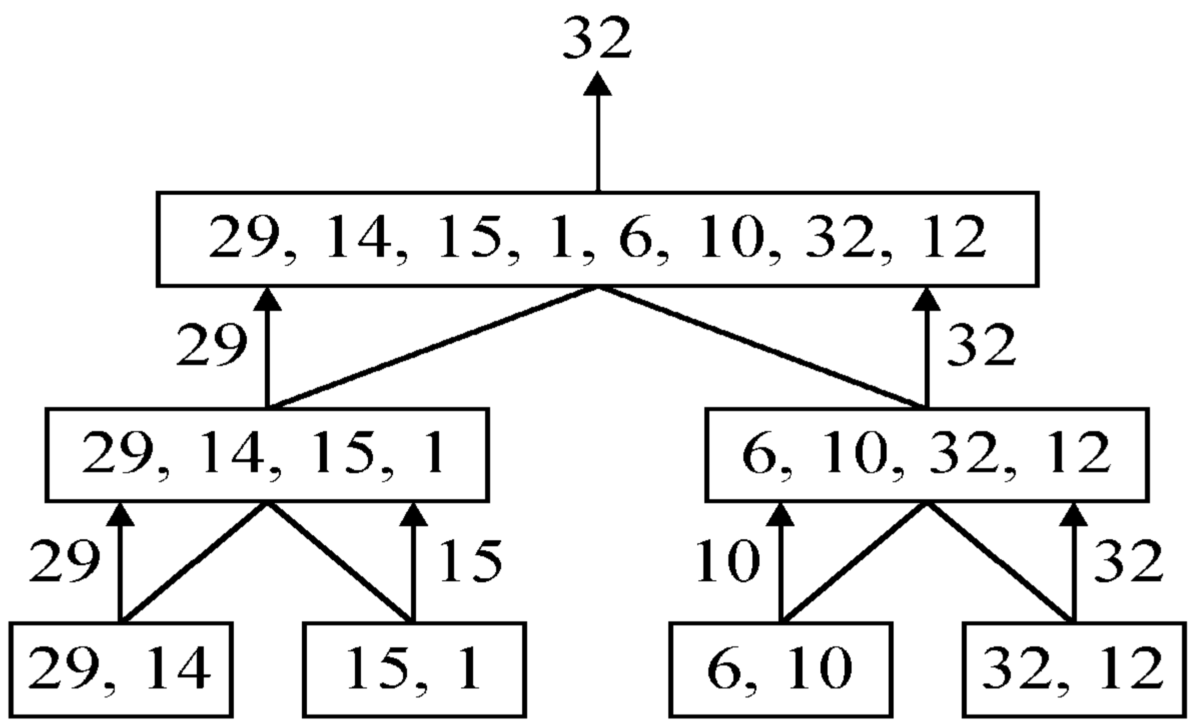
The most well known algorithm design strategy used for solving many problems.

- There are three stages in Divide and conquer strategy:

1. DIVIDE: Divide the problem into two or more sub problems.
2. CONQUER: Solve the sub problems recursively or non-recursively.
3. COMBINE: Combine the results of the sub problems to derive a solution of the given problem.



Ex:find the maximum of set S of n numbers



5)Write and explain recursive binary search algorithm.

<u>Input</u>: A sorted sequence of n elements stored in an array.

Output: The position of x (to be searched).
Step 1: If only one element remains in the array, solve it directly.
Step 2: Compare x with the middle element of the array.
Step 2.1: If x = middle element, then output it and stop.
Step 2.2: If x < middle element, then recursively solve the problem with x and the left half array.
Step 2.3: If x > middle element, then recursively solve the problem with x and the right half array

Recursive Binary Search Algorithm

Algorithm BinSrch(A, i, j, x)

```
{ // A→ Array, i→ least index, j → highest index, x → element for search.
    if (i = j) then
        { if (x == A[i]) then return i;
              else return 0;
        }
    else
        {
              mid ← (i+j)/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, j, x)
        }
    }
```

6)Discuss the time complexity of binary search for best and worst case.

$T(n) = 1 + T(n/2)$
$T(1) = 1$

- $T(n/2) = 1 + T(n/4)$,
  and $T(n/4) = 1 + T(n/8)$, etc.
- So $T(n) = 1 + T(n/2) = 1 + 1 + T(n/4) = 1 + 1 + 1 + T(n/8)$
  and the pattern is $T(n) = k + T(n/2^k)$
  now when $k = \log n$, then $n/(2^{\log n})=n/n = 1$,

- **T(n) =** log n + 1 = **O(log n)**
- Best case time complexity T(n)=O(1)
- Worst case time complexity T(n)=O(log n)
- Average case time complexity T(n)=O(log n)

7. Write the algorithm for finding maximum and minimum problem.

- Given an array of size n, you need to find the maximum and minimum element present in the array.
- In this approach, Divide the array into two equal parts and recursively find the maximum and minimum of those parts.
- After this, compare the maximum and minimum of those parts to get the maximum and minimum of the whole array.

```
• Algorithm MaxMin(a,i,j,max,min)
•  {
•  if ( i==j ) { max = min=a[i];} //small p
• else if ( i +1== j)//small p
• {
•              if ( a[i] < a[j] )
•              {
•              max = a[j] ;
•       min = a[i] ;}
•       else {
•      max = a[i] ;
•       min = a[j] ;}
•       }
• }
• else   //if p is not small,divide p into sub problems
•  {
•  int mid = i+j/2;
•  MaxMin(a,i,mid,max,min);
•  MaxMin(a,mid+1,j,max1,min1)
• if ( max < max1 )
•  max = max1;
•  if ( min> min1 )
•  min = min1;
• }
• }
```

8. Write divide and conquer merge sort algorithm and derive time complexity of this algorithm.

Algorithm:

```
Algorithm MergeSort (low, high)
            {
            if (low < high) then//if there are more than one element
            {
                  mid :=(low + high)/2      //Divide the list into two parts
            MergeSort (low, mid)  // Apply mergesort to the left list
                  MergeSort (mid+1, high)// Apply mergesort to the Right list
                  Merge (low, mid, high)// Merge two lists into one list
             }
            }
Algorithm  Merge (low, mid, high)
       {
            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   //add elements left in the second list
                        for k:=j to high  do
                        {   b[i]:= a[k];  i:=i+1
                        }
                  else
                  for k:=h to mid do //add elements left in the first list
                        { b[i]:= a[k];
                          i:=i+1
                        }
                  for k := low to high do // copying back the sorted list to a[]
                  a[k] := b[k]
     }
```

Time complexity of Mergesort can be described by the recurrence relation :

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

$T(n) = 2(2T(n/4) + n/2) + n$

$= 4\,T(n/4) + 2n$

$= 4(2T(n/8) + n/4) + 2\,n$

$= 8T(n/8) + 3n$

………………

$= 22^k\, T(n/ 2^k2) + k\,n$

$= 22^k\, T(1) + k\,n$

$= n2+ n \log n$

$T(n) = n + n (\log n) = O(n \log n)$

9. Write divide and conquer quick sort algorithm and derive time complexity of this algorithm

Algorithm:

```
Algorithm Quicksort (a, low, high)
{
if (low < high) then //if there are more than one element
{
 j := partition(a, low, high);
// j is the position of the partitioning element
 Quicksort (a,low, j – 1);
 Quicksort(a,j + 1 , high);
//there is no need for combining solutions
}
}
int partition(int a[], low , high)
{
v=a[low];i=low; j=high+1;a[j]=∞;
do
{        do{
                    i++;
                    }while(a[i]<v);
                    do{
                    j--;
                    }while(v<a[j]);
                    if(i<j) {swap(a[i[,a[j]);}
          }while(i<j);
a[low]=a[j];a[j]=v;
return(j);}
```

- The best case of quicksort occurs when the pivot element divides the array into two exactly equal parts, in every step. The recurrence equation is as follows:

  $T(n) = 2T(n/2) + n$

  $T(1)=1$

  using the substitution method the equation can be written as follows:

  $T(n)=2^kT(n/2^k) + kn$

  so $T(n)=O(n\log n)$

- The worst case of quicksort occurs when the first element is always the pivot element .

  The recurrence equation is as follows:

  $T(n) = T(n - 1) + n$

  using the substitution method the equation can be written as follows:

  $T(n)= T(n - k) + (n+(n-1)+(n-2) ..... + (n-k+1))$

  so $T(n)=O(n^2)$

- The average case time complexity of quick sort is :

  $T(n)=O(n \log n)$

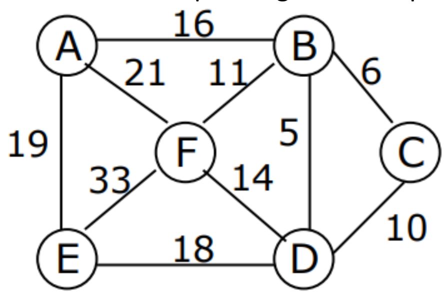
11. What is minimum cost spanning tree? Explain Prim's algorithm with suitable example

Minimum cost spanning trees

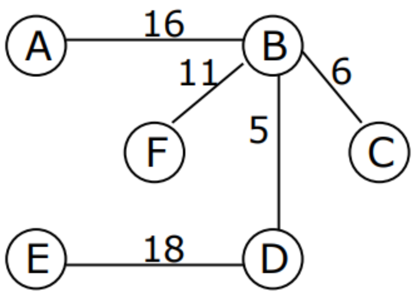
Suppose you have a connected undirected graph with a weight (or cost) associated with each edge

• The cost of a spanning tree would be the sum of the costs of its edges.

• A minimum-cost spanning tree is a spanning tree that has the lowest cost.



A connected, undirected graph



A minimum-cost spanning tree

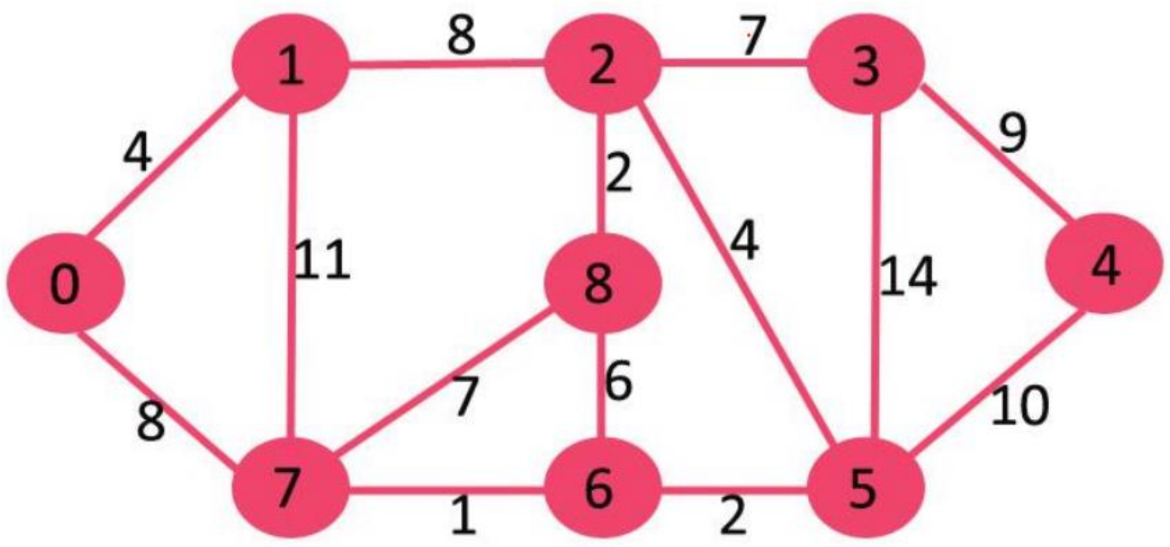
Prim’s algorithm

Steps for finding MST using Prim’s algorithm:

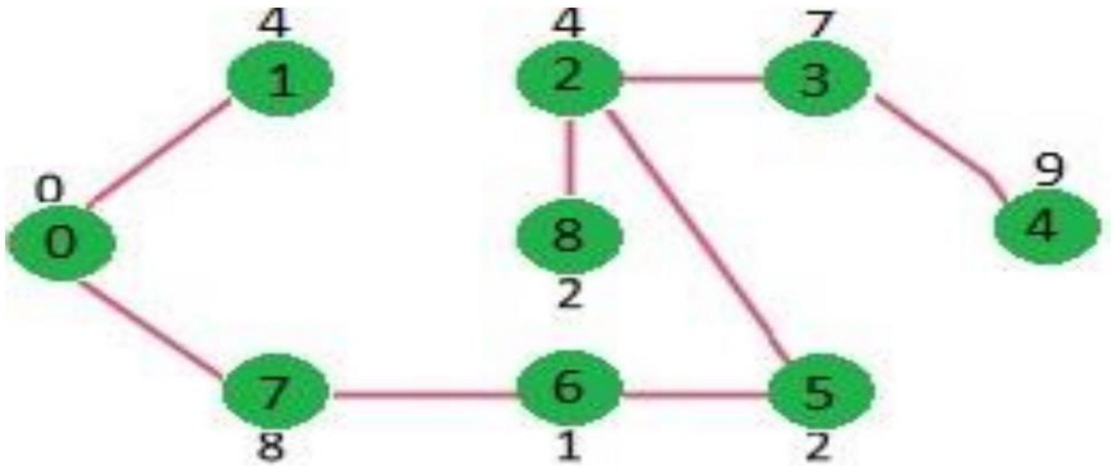
1. Choose a vertex V as starting vertex of the spanning tree.
2. Select an edge with minimum weight that connects V.
3. If this edge does not form a cycle ,then add it to the tree else discard it.
4. Repeat steps 2-3 until all vertices are added to the tree.

• Here, we consider the spanning tree to consists of both nodes and edges.

Example



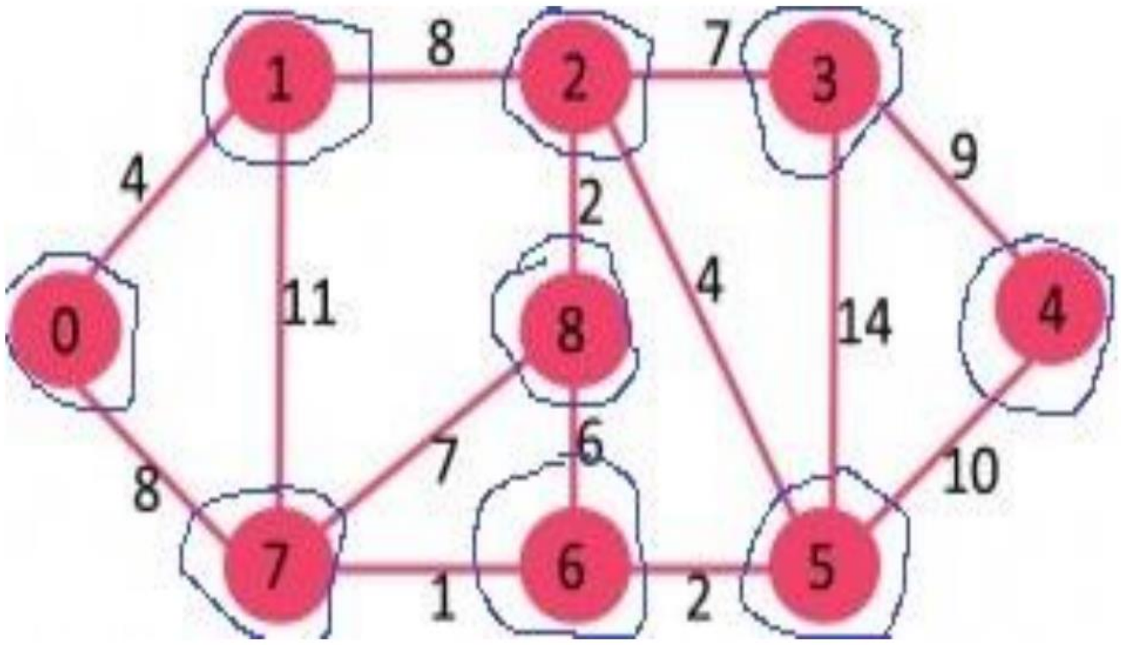
MINIMUM COST SPANNING TREE



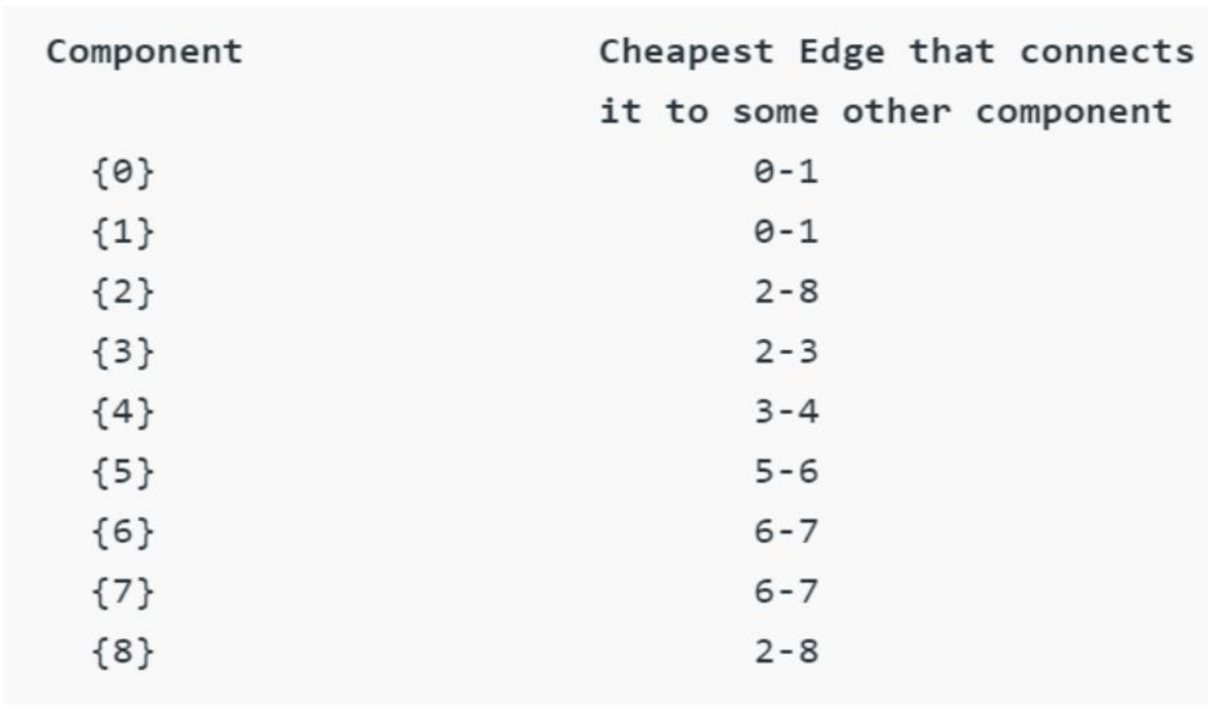
12. Explain Kruskal’s minimum cost spanning tree algorithm with suitable example.

Steps for finding MST using Kruskal’s algorithm:

1. Sort the edges in ascending order of their cost.
2. Add the cheapest edge that does not create a cycle
3. Repeat step 2 until all vertices are visited in the tree.

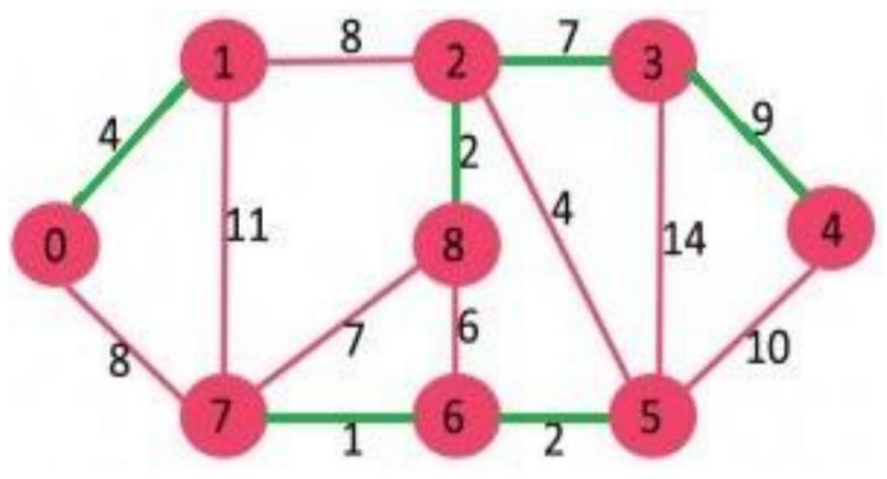


• For every component, find the cheapest edge that connects it to some other component.

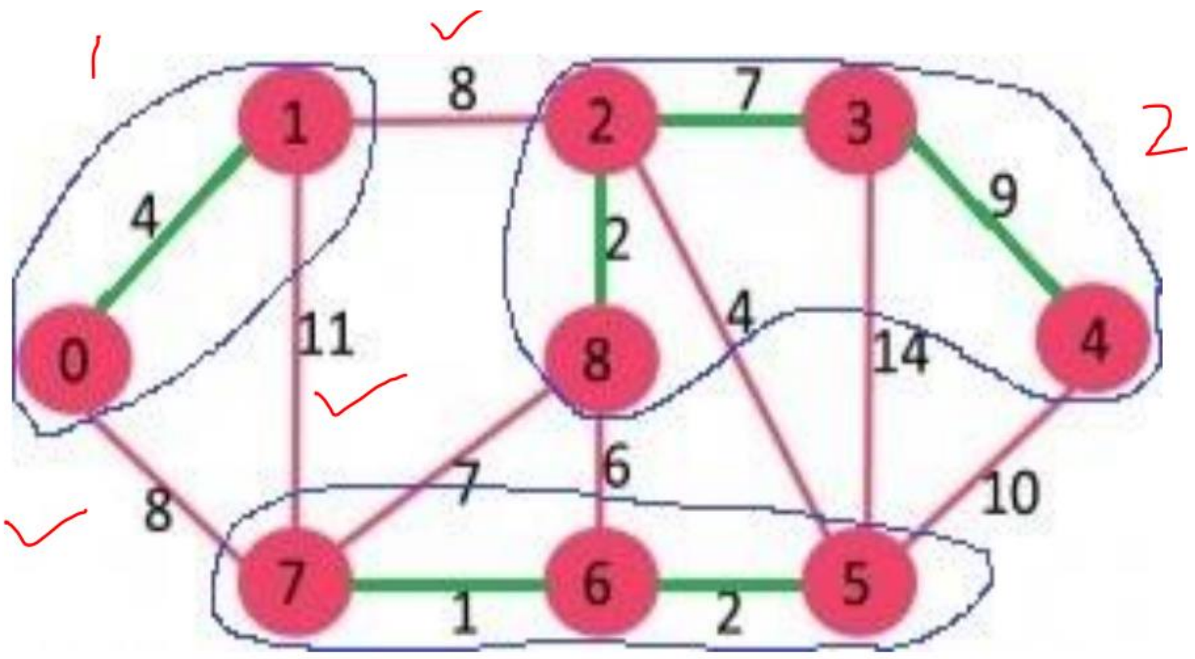


Component	Cheapest Edge that connects it to some other component
{0}	0-1
{1}	0-1
{2}	2-8
{3}	2-3
{4}	3-4
{5}	5-6
{6}	6-7
{7}	6-7
{8}	2-8

- Now MST becomes {0-1, 2-8, 2-3, 3-4, 5-6, 6-7}.



• After above step, components are {{0,1}, {2,3,4,8}, {5,6,7}}. The components are encircled with blue color.



13. Discuss the single source shortest path algorithm with an example.

• Greedy algorithm has the following steps:

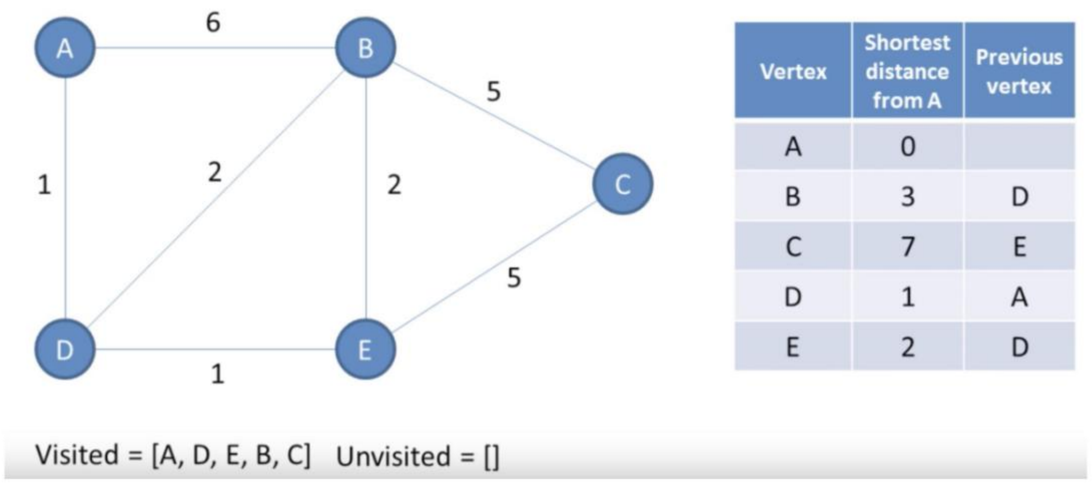
1.select a source vertex and assign distance values to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex.

2.visit a vertex u which is not visited and has minimum distance value.

3. Update distance value of all adjacent vertices of u.For every adjacent vertex v

If d(u)+(u,v)<d(v) them d(v)=d(u)+(u,v)

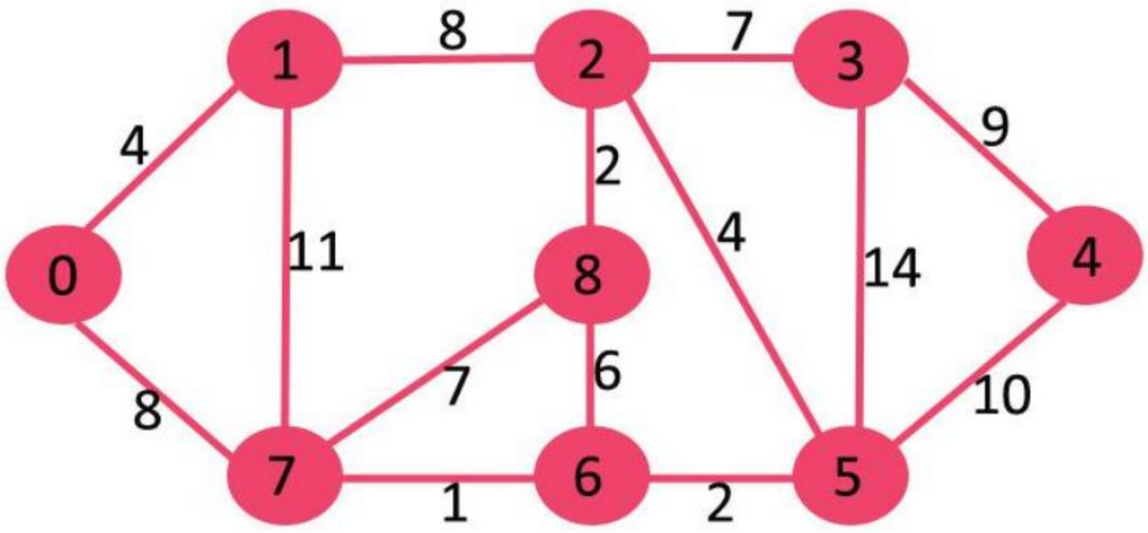
4.repeat step-2&3 until all vertices are visited.

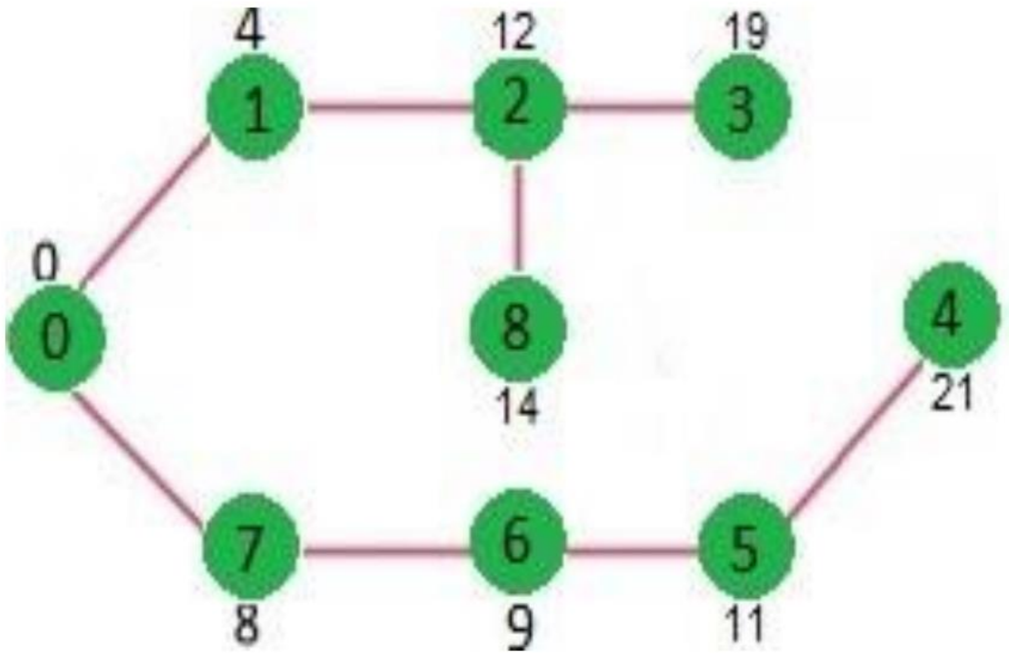


Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Visited = [A, D, E, B, C] Unvisited = []

EXAMPLE:





14. Differentiate between greedy method and dynamic programming.

BASIS FOR COMPARISON	GREEDY METHOD	DYNAMIC PROGRAMMING
Basic	Generates a single decision sequence.	Many decision sequence may be generated.

BASIS FOR COMPARISON	GREEDY METHOD	DYNAMIC PROGRAMMING
Reliability	Less reliable	Highly reliable
Follows	Top-down approach.	Bottom-up approach.
Solutions	Contain a particular set of feasible set of solutions.	There is no special set of feasible set of solution.
Efficiency	More	Less
Overlapping subproblems	Cannot be handled	Chooses the optimal solution to the subproblem.
Example	Fractional knapsack, shortest path.	0/1 Knapsack

15. Explain Dynamic programming general method

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

There are two ways of applying dynamic programming:

- **Top-Down**
- **Bottom-Up**

Top-down approach

The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.

Advantages

- It is very easy to understand and implement.
- It solves the subproblems only when it is required.
- It is easy to debug.
- Example:

```
int fib(int n)
{
   if(n<0)
   error;
 if(n==0)
 return 0;
 if(n==1)
return 1;
sum = fib(n-1) + fib(n-2);
}
```

- The bottom-up is the approach used to avoid the recursion, thus saving the memory space. The bottom-up is an algorithm that starts from the beginning, whereas the recursive algorithm starts from the end and works backward. In the bottom-up approach, we start from the base case to find the answer for the end. As we know, the base cases in the Fibonacci series are 0 and 1. Since the bottom approach starts from the base cases, so we will start from 0 and 1.

```
int fib(int n)
{
   int A[];
   A[0] = 0, A[1] = 1;
   for( i=2; i<=n; i++)
   {
       A[i] = A[i-1] + A[i-2]
   }
   return A[n];
}
```

Refer:-https://www.javatpoint.com/dynamic-programming

10. State the Greedy Knapsack problem?

We are given n items and a knapsack or bag of capacity m.We need to put these items in a knapsack to get the maximum profit value in the knapsack.

•Here each item i has a weight wi and profit pi and the knapsack has a capacity m.

• If a fraction xi , 0 ≤ xi ≤ 1 of item i is placed into the knapsack then a profit of pi *xi is earned.

• The objective is to fill the knapsack that maximizes the total profit earned

• Formally the problem can be stated as follows:

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \qquad \text{——} \quad (1)$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \qquad \text{——} \quad (2)$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n \qquad \text{——} \quad (3)$$

• The profits and weights arepositive numbers.

• A feasible solution is any set (xi,..., xn) satisfying (2)and (3)above.

• An optimal solution is a feasible solutionfor which (1) is maximized.

• There are two types of knapsack problems:

• 1. Fractional Knapsack problems

• -In these problems we can break items for maximizing the total value of knapsack.

• 2. 0/1 Knapsack Problems

• -In these problem we are not allowed to break items. We either take the whole item or don’t take it.

• Methods for selecting items in knapsack problem:

• 1. Greedy selection based on profit

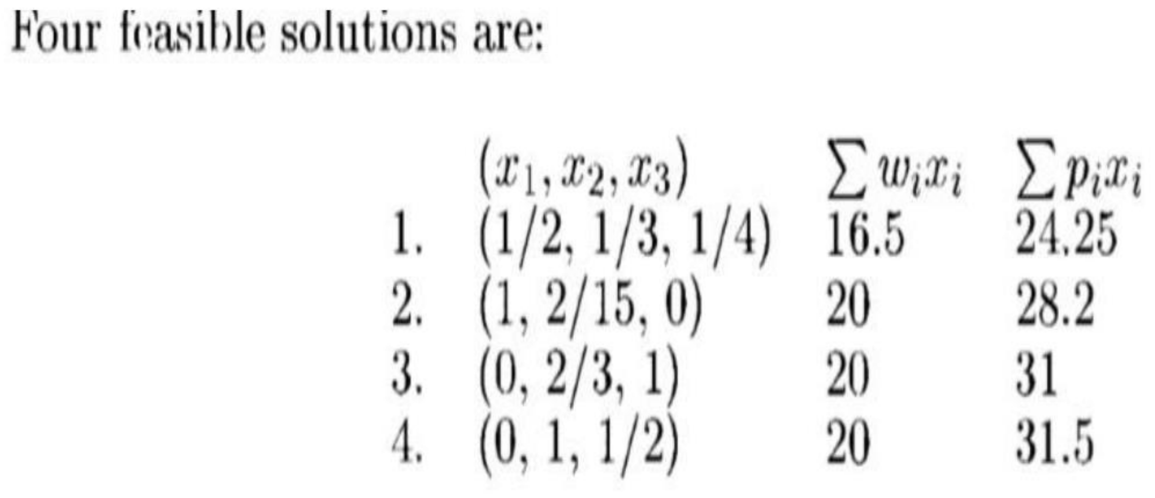
• 2. Greedy selection based on weight •

3.Greedy selection based on profit/weight ratio

Find an optimal solution to the knapsack instance n=3 m=20 (P1,P2,P3)=(25,24,15) and (W1,W2,W3)=(18,15,10)

Optimal solution is =(0,1,1/2)

maximumprofit=31.5



Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5