



DESIGN AND ANALYSIS OF ALGORITHMS

LAB MANUAL

Prepared by

Prof. Ahmad Hassanat

Dr.Audi Albtoush

faculty of Information Technology

Mu'tah University

1. PROGRAM OUTCOMES:

B.TECH - PROGRAM OUTCOMES (POS)

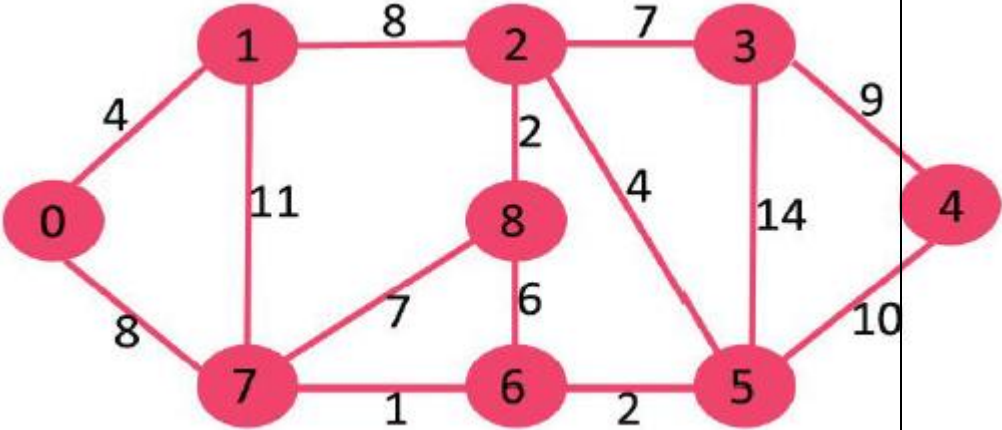
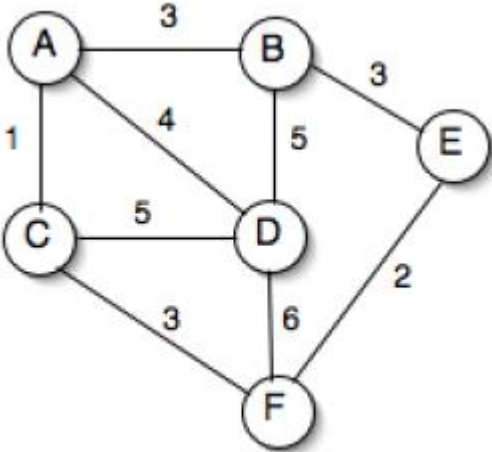
B.TECH - PROGRAM OUTCOMES (POS)	
PO-1	Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems(Engineering knowledge).
PO-2	Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences (Problem analysis).
PO-3	Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations (Design/development of solutions).
PO-4	Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions (Conduct investigations of complex problems).
PO-5	Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations (Modern tool usage).
PO-6	Apply reasoning informed by the contextual knowledge to assesssocietal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice (The engineer and society).
PO-7	Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development (Environment and sustainability).
PO-8	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice (Ethics).
PO-9	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings (Individual and team work).
PO-10	Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions (Communication).
PO-11	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments (Project management and finance).
PO-12	Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change (Life-long learning).

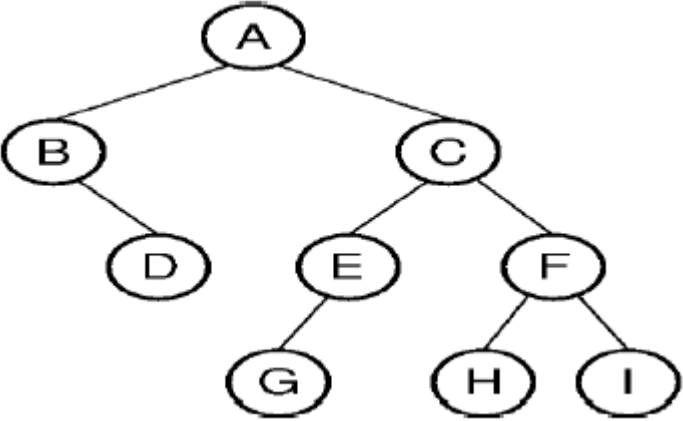
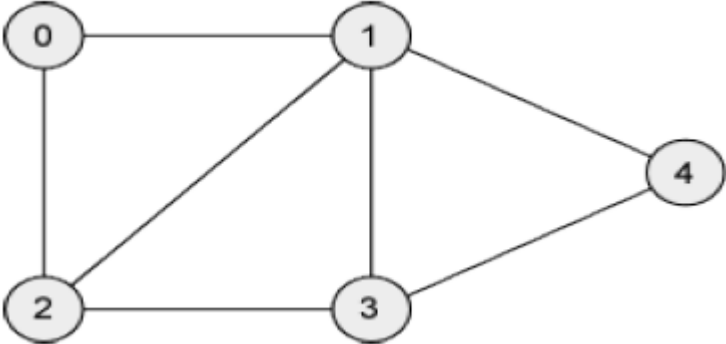
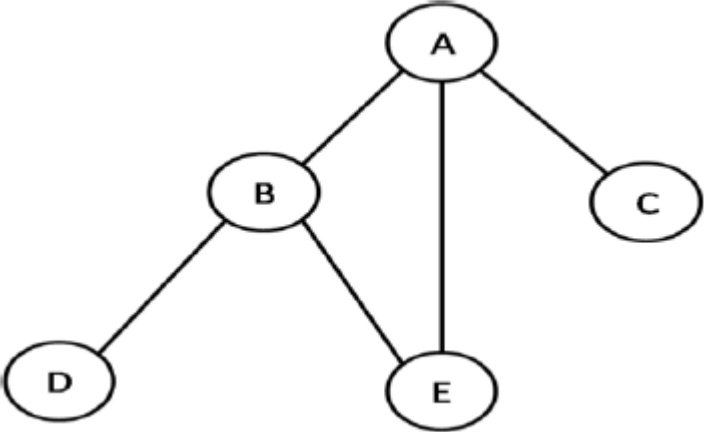
2. PROGRAM SPECIFIC OUTCOMES

PROGRAM SPECIFIC OUTCOMES (PSO's)	
PSO-1	Professional Skills: The ability to understand, analyze and develop computer programs in the areas related to algorithms, system software, multimedia, web design, big data analytics, and networking for efficient design of computer-based systems of varying complexity.
PSO-2	Problem-Solving Skills: The ability to apply standard practices and strategies in software project development using open-ended programming environments to deliver a quality product for business success.
PSO-3	Successful Career and Entrepreneurship: The ability to employ modern computer languages, environments, and platforms in creating innovative career paths to be an entrepreneur, and a zest for higher studies

3. ATTAINMENT OF PROGRAM OUTCOMES AND PROGRAM SPECIFIC OUTCOMES:

S.No	Experiment	Program Outcomes Attained	Program Specific Outcomes Attained
WEEK-1	Data Structures~ review		
WEEK-2	QUICK SORT Sort a given set of elements using the quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the 1 st to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.	PO-2,PO-3	PSO-1
WEEK-3	MERGE SORT Implement merge sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.	PO-2,PO-3	PSO-1
WEEK-4	WARSHALL'S ALGORITHM a. Obtain the Topological ordering of vertices in a given digraph <div data-bbox="224 1108 958 1522" data-label="Diagram"> <pre> graph TD 5((5)) --> 2((2)) 5((5)) --> 0((0)) 4((4)) --> 0((0)) 4((4)) --> 1((1)) 2((2)) --> 3((3)) 1((1)) --> 3((3)) </pre> </div> b. Compute the transitive closure of a given directed graph using Warshall's algorithm.	PO-2	
WEEK-5	KNAPSACK PROBLEM Implement 0/1 Knapsack problem using Dynamic Programming.	PO-3	PSO-1

WEEK-6	SHORTEST PATHS ALGORITHM From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm. 	PO-3	PSO-1
WEEK-7	MINIMUM COST SPANNING TREE Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm. 	PO-3	PSO-1
WEEK-8	TREE TRAVERSALS Perform various tree traversal algorithms for a given tree.	PO-2, PO-3	

			
WEEK-9	<p>GRAPH TRAVERSALS</p> <p>a. Print all the nodes reachable from a given starting node in a digraph using BFS method.</p>  <p>b. Check whether a given graph is connected or not using DFS method.</p> 	PO-2, PO-3	
WEEK-10	<p>SUM OF SUB SETS PROBLEM</p> <p>Find a subset of a given set $S = \{s_1, s_2, \dots, s_n\}$ of n positive integers whose sum is equal to a given positive integer d. For example, if $S = \{1, 2, 5, 6, 8\}$ and $d = 9$ there are two solutions $\{1, 2, 6\}$ and $\{1, 8\}$. A suitable message is to be displayed if the given problem instance doesn't have a solution.</p>	PO-3	

WEEK-11	TRAVELLING SALES PERSON PROBLEM Implement any scheme to find the optimal solution for the Traveling Sales Person problem and then solve the same problem instance using any approximation algorithm and determine the error in the approximation.	PO-3, PO-12	PSO-1
----------------	---	------------------------	--------------

4. MAPPING COURSE OBJECTIVES LEADING TO THE ACHIEVEMENT OF PROGRAM OUTCOMES:

Course Objectives	Program Outcomes												Program Specific Outcomes		
	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO-1	PSO-2	PSO-3
I		√	√									√	√		
II		√	√									√	√		

5. SYLLABUS:

DESIGN AND ANALYSIS OF ALGORITHMS LABORATORY

OBJECTIVES:

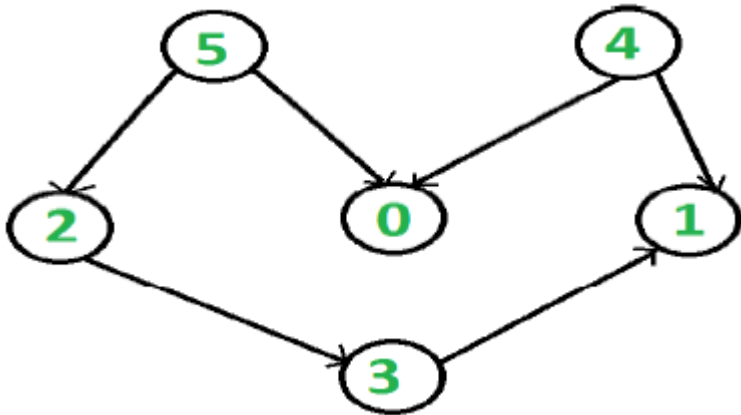
The course should enable the students to:

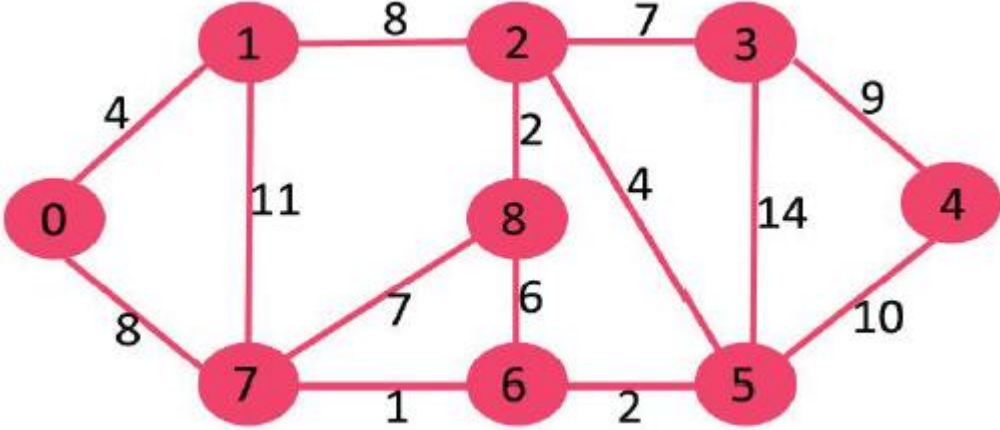
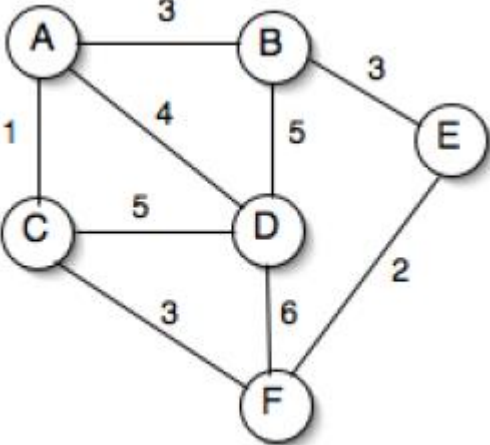
Learn how to analyze a problem and design the solution for the problem.

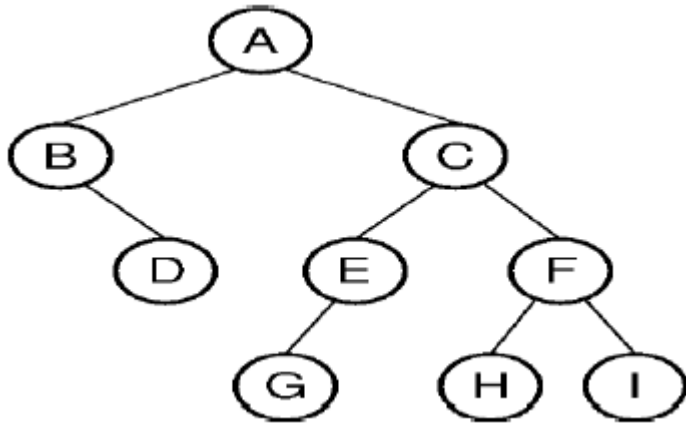
I. Design and implement efficient algorithms for a specified application.

II. Strengthen the ability to identify and apply the suitable algorithm for the given real world problem.

LIST OF EXPERIMENTS

WEEK-1	Data Structures~ review
WEEK-2	QUICK SORT
Sort a given set of elements using the quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the 1 st to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.	
WEEK-3	MERGE SORT
Implement merge sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.	
WEEK-4	WARSHALL'S ALGORITHM
<p>c. Obtain the Topological ordering of vertices in a given digraph.</p>  <pre>graph TD; 5((5)) --> 2((2)); 5((5)) --> 0((0)); 4((4)) --> 0((0)); 4((4)) --> 1((1)); 2((2)) --> 3((3)); 1((1)) --> 3((3));</pre> <p>d. Compute the transitive closure of a given directed graph using Warshall's algorithm.</p>	

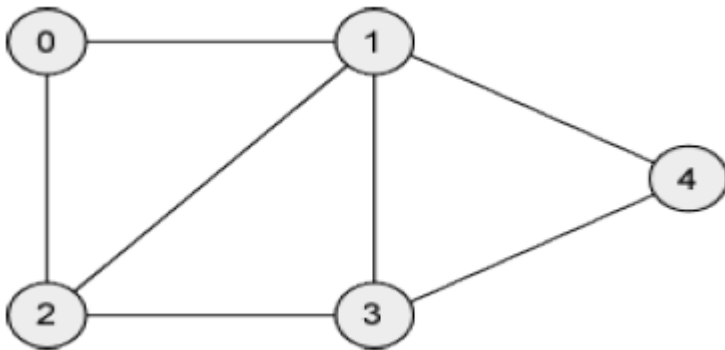
WEEK-5	KNAPSACK PROBLEM
Implement 0/1 Knapsack problem using Dynamic Programming.	
WEEK-6	SHORTEST PATHS ALGORITHM
<p>From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.</p> 	
WEEK-7	MINIMUM COST SPANNING TREE
<p>Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.</p> 	
WEEK-8	TREE TRAVESRSALS
Perform various tree traversal algorithms for a given tree.	



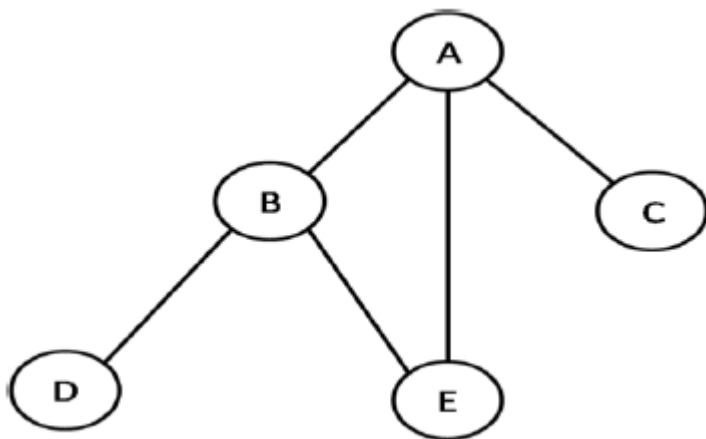
WEEK-9

GRAPH TRAVERSALS

a. Print all the nodes reachable from a given starting node in a digraph using BFS method.



b. Check whether a given graph is connected or not using DFS method.



WEEK-10

SUM OF SUB SETS PROBLEM

Find a subset of a given set $S = \{s_1, s_2, \dots, s_n\}$ of n positive integers whose sum is equal to a given positive integer d . For example, if $S = \{1, 2, 5, 6, 8\}$ and $d = 9$ there are two solutions $\{1, 2, 6\}$ and $\{1, 8\}$. A suitable message is to be displayed if the given problem instance doesn't have a solution.

WEEK-11	TRAVELLING SALES PERSON PROBLEM
Implement any scheme to find the optimal solution for the Traveling Sales Person problem and then solve the same problem instance using any approximation algorithm and determine the error in the approximation	
Reference Books: <ol style="list-style-type: none"> 1. Levitin A, “Introduction to the Design And Analysis of Algorithms”, Pearson Education, 2008. 2. Goodrich M.T., R Tomassia, “Algorithm Design foundations Analysis and Internet Examples”, John Wiley and Sons, 2006. 3. Base Sara, Allen Van Gelder, “ Computer Algorithms Introduction to Design and Analysis”, Pearson, 3rd Edition, 1999. 	
Web References: <ol style="list-style-type: none"> 1. http://www.personal.kent.edu/~rmuhamma/Algorithms/algorithm.html 2. http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=IntroToAlgorithms 3. http://www.facweb.iitkgp.ernet.in/~sourav/daa.html 4. https://github.com/owdaybtoush 	
SOFTWARE AND HARDWARE REQUIREMENTS FOR A BATCH OF 36 STUDENTS: HARDWARE: Desktop Computer Systems: 36 nos SOFTWARE: Application Software: C++ Programming Compiler	

6. INDEX:

Sl. No	Experiment	Page No
1	Data Structures~ review	--
2	QUICK SORT	13
3	MERGE SORT	18
4	WARSHALL'S ALGORITHM	25
5	KNAPSACK PROBLEM	28
6	SHORTEST PATHS ALGORITHM	31
7	MINIMUM COST SPANNING TREE	34
8	TREE TRAVERSALS	37
9	GRAPH TRAVERSALS	44
10	SUM OF SUB SETS PROBLEM	49
11	TRAVELLING SALES PERSON PROBLEM	52

WEEK-2

QUICK SORT

2.1 OBJECTIVE:

Sort a given set of elements using the Quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

2.2 RESOURCES:

Dev. C++

2.3 PROGRAM LOGIC:

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

There are many different versions of QuickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in Quicksort is partition. Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

2.4- Quick Sort Algorithm

Quick Sort Algorithm is a **Divide & Conquer** algorithm. It divides input array in two partitions, calls itself for the two partitions(recursively) and performs in-place sorting while doing so. A separate **partition()** function is used for performing this in-place sorting at every iteration. Quick sort is one of the most efficient sorting algorithms.

- **Time Complexity: $\theta(n\log(n))$**
- **Space Complexity: $O(\log(n))$**

Quick Sort Algorithm(Pseudo Code)

Working –

There are 2 Phases (3 major steps) in the Quick Sort Algorithm –

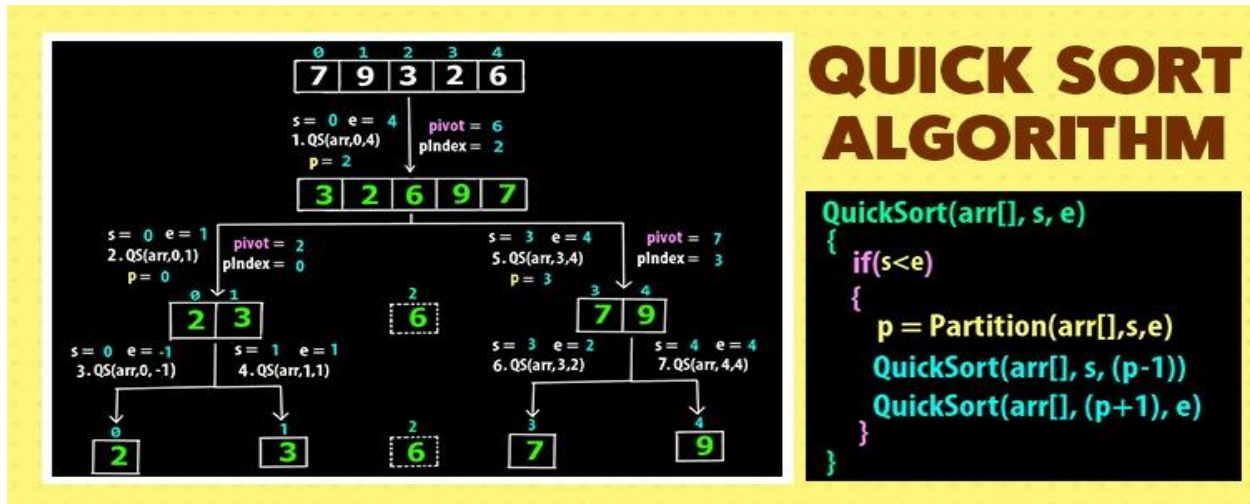
1. Division Phase – Divide the array(list) into 2 halves by finding the pivot point to perform the partition of the array(list).
 1. The in-place sorting happens in this partition process itself.
2. Recursion Phase –
 1. Call Quick Sort on the left partition (sub-list)
 2. Call Quick Sort on the right partition (sub-list)

```
QuickSort(arr[], s, e)
{
    if(s<e)
    {
        p = Partition(arr[],s,e)
        QuickSort(arr[], s, (p-1))
        QuickSort(arr[], (p+1), e)
    }
}
```

Quick Sort Partition Function(Pseudo Code)

```
Partition(arr[], s, e)
{
    pivot = arr[e]
    pIndex = s
    for (i=s to e-1)
    {
        if(arr[i]<=pivot)
        {
            swap(arr[i], arr[pIndex])
            pIndex++
        }
    }
    swap(arr[e], arr[pIndex])
    return pIndex
}
```

Example



2.5 PROCEDURE:

1. Create: Open Dev. C++, write a program after that save the program with .c extension.
2. Compile: Alt + F9
3. Execute: Ctrl + F10

2.6 C++ Program to Implement Quick Sort

```
#include <iostream>
using namespace std;
// quick sort sorting algorithm
int Partition(int arr[], int s, int e)
{
    int pivot = arr[e];
    int pIndex = s;

    for(int i = s; i < e; i++)
    {
        if(arr[i] < pivot)
        {
            int temp = arr[i];
            arr[i] = arr[pIndex];
            arr[pIndex] = temp;
            pIndex++;
        }
    }

    int temp = arr[e];
```

```

arr[e] = arr[pIndex];
arr[pIndex] = temp;

return pIndex;
}

void QuickSort(int arr[], int s, int e)
{
    if(s<e)
    {
        int p = Partition(arr,s, e);
        QuickSort(arr, s, (p-1)); // recursive QS call for left partition
        QuickSort(arr, (p+1), e); // recursive QS call for right partition
    }
}

int main()
{
    int size=0;
    cout<<"Enter Size of array: "<<endl;
    cin>>size;
    int myarray[size];

    cout<<"Enter "<<size<<" integers in any order: "<<endl;
    for(int i=0;i<size;i++)
    {
        cin>>myarray[i];
    }
    cout<<"Before Sorting"<<endl;
    for(int i=0;i<size;i++)
    {
        cout<<myarray[i]<<" ";
    }
    cout<<endl;

    QuickSort(myarray,0,(size-1)); // quick sort called

    cout<<" After Sorting"<<endl;
    for(int i=0;i<size;i++)
    {
        cout<<myarray[i]<<" ";
    }

    return 0;
}

```


2.7 INPUT/ OUTPUT

```
Enter Size of array:
5
Enter 5 integers in any order:
7
9
3
2
6
Before Sorting
7 9 3 2 6
After Sorting
2 3 6 7 9
```

2.8 LAB VIVA QUESTIONS:

1. What is the average case time complexity of quick sort.
2. Explain is divide and conquer.
3. Define in place sorting algorithm.
4. List different ways of selecting pivot element.

WEEK-3

MERGE SORT

3.1 OBJECTIVE:

Implement merge sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of n , the number of elements in the list to be sorted and plot a graph of the time taken versus n . The elements can be read from a file or can be generated using the random number generator.

3.2 RESOURCES:

Dev C++

3.3 PROGRAM LOGIC:

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.

The merge() function is used for merging two halves. The merge(a , low, mid, high) is key process that assumes that $a[\text{low}..\text{mid}]$ and $a[\text{mid}+1..\text{high}]$ are sorted and merges the two sorted sub-arrays into one.

3.4 PROCEDURE:

1. Create: Open Dev C++, write a program after that save the program with .c extension.
2. Compile: Alt + F9
3. Execute: Ctrl + F10

Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

One thing that you might wonder is what is the specialty of this algorithm. We already have a number of sorting algorithms then why do we need this algorithm? One of the main advantages of merge sort is that it has a time complexity of $O(n \log n)$, which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.

Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other

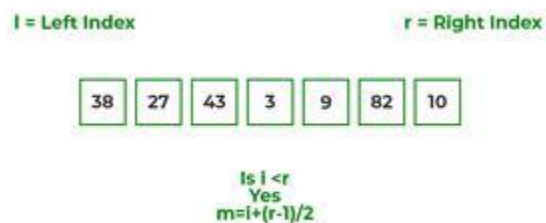
algorithms, such as quicksort, to improve the overall performance of a sorting routine.

Illustration:

To know the functioning of merge sort, let's consider an array

`arr[] = {38, 27, 43, 3, 9, 82, 10}`

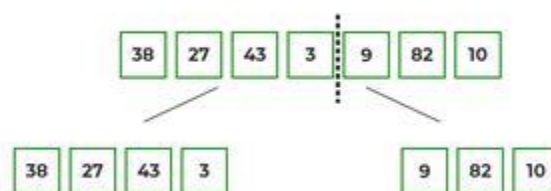
- At first, check if the left index of array is less than the right index, if yes then calculate its mid point



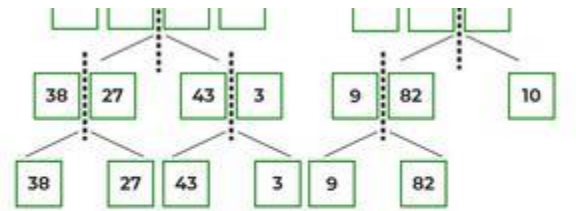
- Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.
- Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.



- Now, again find that the left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.

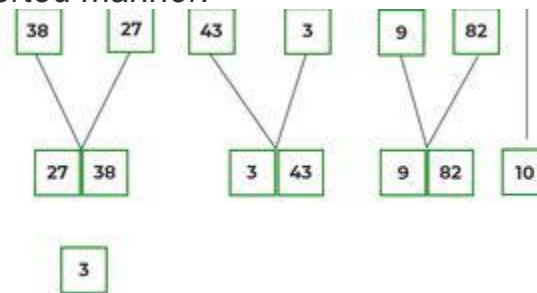


- Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.

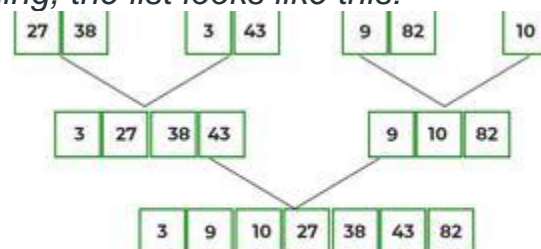


After dividing the array into smallest units merging starts, based on comparison of elements.

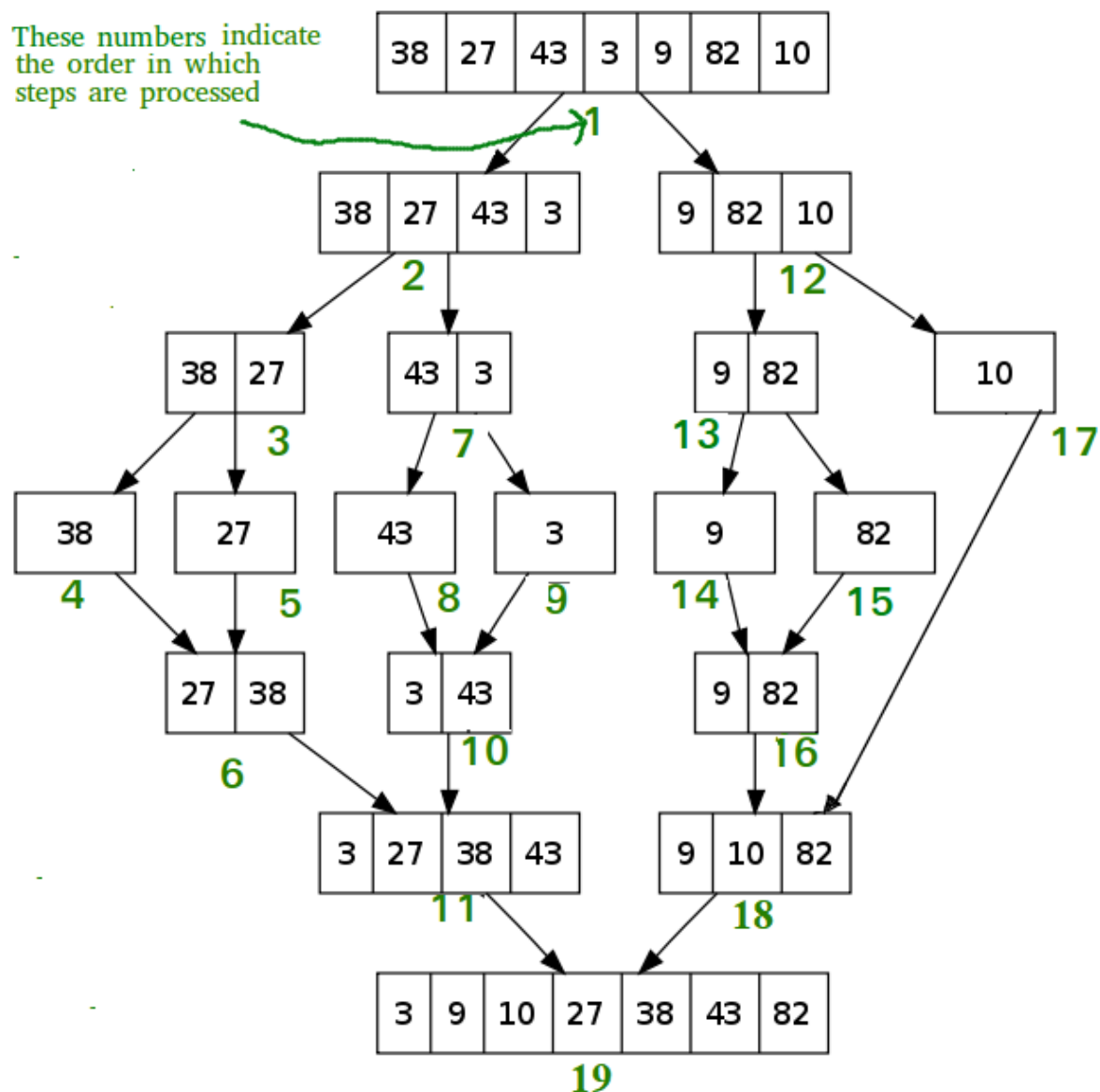
- After dividing the array into smallest units, start merging the elements again based on comparison of size of elements
- Firstly, compare the element for each list and then combine them into another list in a sorted manner.



- After the final merging, the list looks like this:



These numbers indicate the order in which steps are processed



Algorithm:

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

if left > right

return

mid= (left+right)/2

mergesort(array, left, mid)

mergesort(array, mid+1, right)

merge(array, left, mid, right)

step 4: Stop

3.5 SOURCE CODE:

```
C++ program for Merge Sort
#include <iostream>
using namespace std;

// Merges two subarrays of array[].
// First subarray is arr[begin..mid]
// Second subarray is arr[mid+1..end]
void merge(int array[], int const left, int const mid,
           int const right)
{
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;

    // Create temp arrays
    auto *leftArray = new int[subArrayOne],
        *rightArray = new int[subArrayTwo];

    // Copy data to temp arrays leftArray[] and rightArray[]
    for (auto i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    auto indexOfSubArrayOne
        = 0, // Initial index of first sub-array
        indexOfSubArrayTwo
        = 0; // Initial index of second sub-array
    int indexOfMergedArray
        = left; // Initial index of merged array

    // Merge the temp arrays back into array[left..right]
    while (indexOfSubArrayOne < subArrayOne
        && indexOfSubArrayTwo < subArrayTwo) {
        if (leftArray[indexOfSubArrayOne]
            <= rightArray[indexOfSubArrayTwo]) {
            array[indexOfMergedArray]
                = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
        else {
            array[indexOfMergedArray]
                = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // left[], if there are any
    while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }
    // Copy the remaining elements of
```

```

        // right[], if there are any
        while (indexOfSubArrayTwo < subArrayTwo) {
            array[indexOfMergedArray]
                = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
            indexOfMergedArray++;
        }
        delete[] leftArray;
        delete[] rightArray;
    }

    // begin is for left index and end is
    // right index of the sub-array
    // of arr to be sorted */
    void mergeSort(int array[], int const begin, int const end)
    {
        if (begin >= end)
            return; // Returns recursively

        auto mid = begin + (end - begin) / 2;
        mergeSort(array, begin, mid);
        mergeSort(array, mid + 1, end);
        merge(array, begin, mid, end);
    }

    // UTILITY FUNCTIONS
    // Function to print an array
    void printArray(int A[], int size)
    {
        for (auto i = 0; i < size; i++)
            cout << A[i] << " ";
    }

    // Driver code
    int main()
    {
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        auto arr_size = sizeof(arr) / sizeof(arr[0]);

        cout << "Given array is \n";
        printArray(arr, arr_size);

        mergeSort(arr, 0, arr_size - 1);

        cout << "\nSorted array is \n";
        printArray(arr, arr_size);
        return 0;
    }

```

Output

```
Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13
```

Time Complexity: $O(N \log(N))$, Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

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

3.6 LAB VIVA QUESTIONS:

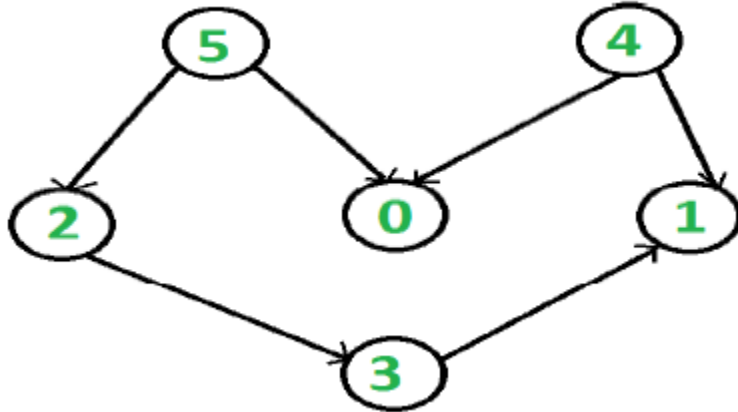
1. What is the running time of merge sort?
2. What technique is used to sort elements in merge sort?
3. Is merge sort in place sorting algorithm?
4. Define stable sort algorithm.

WEEK-4

WARSHALL'S ALGORITHM

4.1 OBJECTIVE:

1. Obtain the Topological ordering of vertices in a given digraph.



Compute the transitive closure of a given directed graph using Warshall's algorithm.

4.2 RESOURCES:

Dev C++

4.3 PROGRAM LOGIC:

Topological ordering

In topological sorting, a temporary stack is used with the name “s”. The node number is not printed immediately; first iteratively call topological sorting for all its adjacent vertices, then push adjacent vertex to stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Transitive closure

Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j . The reach-ability matrix is called transitive closure of a graph.

Warshall's algorithm

Warshall's algorithm is used to determine the transitive closure of a directed graph or all paths in a directed graph by using the adjacency matrix. For this, it generates a sequence of n matrices. Where, n is used to describe the number of vertices. A sequence of vertices is used to define a path in a simple graph.

4.4 PROCEDURE:

1. Create: Open Dev C++, write a program after that save the program with .c extension.
2. Compile: Alt + F9
3. Execute: Ctrl + F10

4.5 SOURCE CODE:

```
#include <iostream>
int n,a[10][10],p[10][10];
void path()
{
    int i,j,k;
    for(i=0;i<n;i++)
    for(j=0;j<n;j++)
    p[i][j]=a[i][j];
    for(k=0;k<n;k++)
    for(i=0;i<n;i++)
    for(j=0;j<n;j++)
    if(p[i][k]==1&& p[k][j]==1) p[i][j]=1;
}
int main()
{
    int i,j;
    //clrscr();
    printf("Enter the number of nodes:");
    scanf("%d",&n);
    printf("\nEnter the adjacency matrix:\n");
    for(i=0;i<n;i++)
    for(j=0;j<n;j++)
    scanf("%d",&a[i][j]);
    path();
    printf("\nThe path matrix is showm below\n");
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        printf("%d ",p[i][j]);
        printf("\n");
    }
    //getch();
}
```

Output

```
Enter the number of nodes:4
Enter the adjacency matrix:
0 1 0 0
1 0 1 1
1 0 1 0
1 1 1 1
The path matrix is shown below
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
```

4.6 LAB VIVA QUESTIONS:

1. Define transitive closure.
2. Define topological sequence.
3. What is the time complexity of Warshall's algorithm?

WEEK-5 KNAPSACK PROBLEM

5.1 OBJECTIVE:

Implement 0/1 Knapsack problem using Dynamic Programming.

5.2 RESOURCES:

Dev C++

5.3 PROGRAM LOGIC:

Given some items, pack the knapsack to get the maximum total profit. Each item has some Weight and some profit. Total weight that we can carry is no more than some fixed number W .

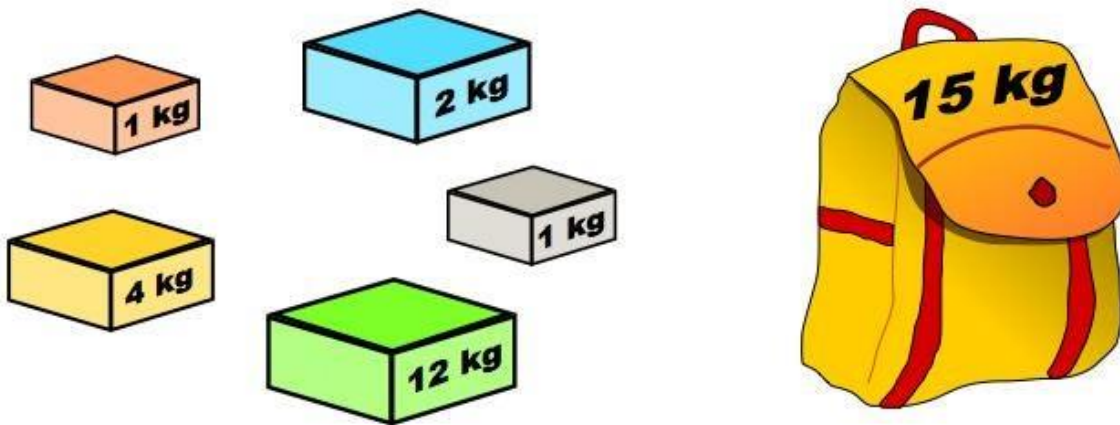
0/1 Knapsack Problem using recursion:

To solve the problem follow the below idea:

A simple solution is to consider all subsets of items and calculate the total weight and profit of all subsets. Consider the only subsets whose total weight is smaller than W . From all such subsets, pick the subset with maximum profit.

Optimal Substructure: *To consider all subsets of items, there can be two cases for every item.*

- **Case 1:** *The item is included in the optimal subset.*
- **Case 2:** *The item is not included in the optimal set.*



5.4 PROCEDURE:

1. Create: Open Dev C++, write a program after that save the program with .c extension.
2. Compile: Alt + F9
3. Execute: Ctrl + F10

5.5 SOURCE CODE:

```
#include<iostream>
int w[10],p[10],v[10][10],n,i,j,cap,x[10]={0};
int max(int i,int j){
return ((i>j)?i:j);
}
int knap(int i,int j){
int value;
if(v[i][j]<0){
if(j<w[i])
value=knap(i-1,j);
else
value=max(knap(i-1,j),p[i]+knap(i-1,j-w[i]));
v[i][j]=value;
}
return(v[i][j]);
}
int main(){
int profit,count=0;
printf("\nEnter the number of objects ");
scanf("%d",&n);
printf("Enter the profit and weights of the elements \n ");
for(i=1;i<=n;i++){
printf("\nEnter profit and weight For object no %d :",i);
scanf("%d%d",&p[i],&w[i]);
}
printf("\nEnter the capacity ");
scanf("%d",&cap);
for(i=0;i<=n;i++)
for(j=0;j<=cap;j++)
if((i==0)||j==0)
v[i][j]=0;
else
v[i][j]=-1;
profit=knap(n,cap);
i=n;
j=cap;
while(j!=0&&i!=0){
if(v[i][j]!=v[i-1][j]){
x[i]=1;
```

```

j=j-w[i];
i--;
}
else
i--;
}
printf("object included are \n ");
printf("Sl.no\tweight\tprofit\n");
for(i=1;i<=n;i++)
if(x[i])
printf("%d\t%d\t%d\n",++count,w[i],p[i]);
printf("Total profit = %d\n",profit);
}
4.6 INPUT/ OUTPUT

```

```

Enter profit and weight For object no 2 :2 3
Enter profit and weight For object no 3 :5 3

Enter the capacity 6
object included are
  Sl.no  weight  profit
1       3       2
2       3       5
Total profit = 7

```

5.5 LAB VIVA QUESTIONS:

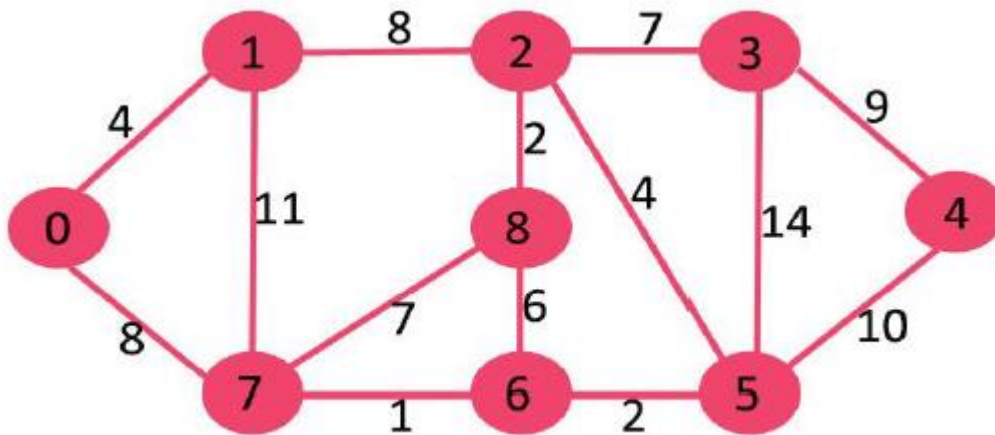
1. Define knapsack problem.
2. Define principle of optimality.
3. What is the optimal solution for knapsack problem?
4. What is the time complexity of knapsack problem?

WEEK-6

SHORTEST PATHS ALGORITHM

6.1 OBJECTIVE:

From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.



6.2 RESOURCES:

Dev C++

6.3 PROGRAM LOGIC:

1) Create a set S that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty. **2)** Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first. **3)** While S doesn't include all vertices

a) Pick a vertex u which is not there in S and has minimum distance value. **b)** Include u to S .

c) Update distance value of all adjacent vertices of u .

To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v , if sum of distance value of u (from source) and weight of edge $u-v$, is less than the distance value of v , then update the distance value of v .

shortest path problem

is the problem of finding a **path** between two **vertices** (or nodes) in a **graph** such that the sum of the **weights** of its constituent edges is minimized.

The problem of finding the shortest path between two intersections on a road map may be modeled as a special case of the shortest path problem

in graphs, where the vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of the segment.

6.4 PROCEDURE:

1. Create: Open Dev C++, write a program after that save the program with .c extension.
2. Compile: Alt + F9
3. Execute: Ctrl + F10

6.5 SOURCE CODE:

```
#include<iostream>
#define infinity 999
void dij(int n, int v,int cost[20][20], int dist[]){
    int i,u,count,w,flag[20],min;
    for(i=1;i<=n;i++)
        flag[i]=0, dist[i]=cost[v][i];
    count=2;
    while(count<=n){
        min=99;
        for(w=1;w<=n;w++)
            if(dist[w]<min && !flag[w]) {
                min=dist[w];
                u=w;
            }
        flag[u]=1;
        count++;
        for(w=1;w<=n;w++)
            if((dist[u]+cost[u][w]<dist[w]) && !flag[w])
                dist[w]=dist[u]+cost[u][w];
    }
}

int main(){
    int n,v,i,j,cost[20][20],dist[20];
    printf("enter the number of nodes:");
    scanf("%d",&n);
    printf("\n enter the cost matrix:\n");
```



```

for(i=1;i<=n;i++)
for(j=1;j<=n;j++){
scanf("%d",&cost[i][j]);
if(cost[i][j] == 0)
cost[i][j]=infinity;
}
printf("\n enter the source matrix:");
scanf("%d",&v);
dij(n,v,cost,dist);
printf("\n shortest path : \n");
for(i=1;i<=n;i++)
if(i!=v)
printf("%d->%d,cost=%d\n",v,i,dist[i]);
}

```

6.6 INPUT/ OUTPUT

```

enter the number of nodes:9

enter the cost matrix:
0 4 0 0 0 0 0 8 0
4 0 8 0 0 0 0 11 0
0 8 0 7 0 4 0 0 2
0 0 7 0 9 14 0 0 0
0 0 0 9 0 10 0 0 0
0 0 4 14 10 0 2 0 0
0 0 0 0 0 2 0 1 6
8 11 0 0 0 0 1 0 7
0 0 2 0 0 0 6 7 0

enter the source matrix:1

shortest path :
1->2,cost=4
1->3,cost=12
1->4,cost=19
1->5,cost=21
1->6,cost=11
1->7,cost=9
1->8,cost=8
1->9,cost=14

-----
Process exited after 148.6 seconds with return value 9
Press any key to continue . . . _

```

6.7 LAB VIVA QUESTIONS:

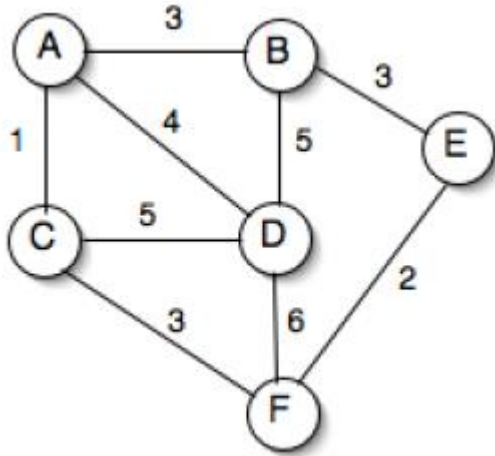
1. What is the time complexity of Dijkstra's algorithm?
2. Define cost matrix.
3. Define directed graph.
4. Define connected graph.

WEEK-7

MINIMUM COST SPANNING TREE

7.1 OBJECTIVE:

Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.



7.2 RESOURCES:

Dev C++

7.3 PROGRAM LOGIC:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

7.4 PROCEDURE:

1. Create: Open Dev C++, write a program after that save the program with .c extension.
2. Compile: Alt + F9
3. Execute: Ctrl + F10

7.5 SOURCE CODE:

```
#include<iostream>

int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
int main() {
printf("\n Implementation of Kruskal's algorithm\n\n");
printf("\nEnter the no. of vertices\n");
scanf("%d",&n);
printf("\nEnter the cost adjacency matrix\n");
for(i=1;i<=n;i++){
for(j=1;j<=n;j++) {
scanf("%d",&cost[i][j]);
```

```

if(cost[i][j]==0)
cost[i][j]=999;
}
}
printf("\nThe edges of Minimum Cost Spanning Tree are\n\n");
while(ne<n){
for(i=1,min=999;i<=n;i++) {
for(j=1;j<=n;j++){
if(cost[i][j]<min){
min=cost[i][j];
a=u=i;
b=v=j;
}
}
}
u=find(u);
v=find(v);
if(uni(u,v)){
printf("\n%d edge (%d,%d) =%d\n",ne++,a,b,min);
mincost +=min;
}
cost[a][b]=cost[b][a]=999;
}
printf("\n\tMinimum cost = %d\n",mincost);
}
int find(int i){
while(parent[i])
i=parent[i];
return i;
}
int uni(int i,int j){
if(i!=j) {
parent[j]=i;
return 1;
}
return 0;
}
}

```

7.6 INPUT/ OUTPUT

```
Implementation of Kruskal's algorithm

Enter the no. of vertices
6

Enter the cost adjacency matrix
999 3 1 4 999 999
3 999 999 5 3 999
1 999 999 5 999 3
4 5 5 999 999 6
999 3 999 999 999 2
999 999 3 6 2 999

The edges of Minimum Cost Spanning Tree are

1 edge <1,3> =1
2 edge <5,6> =2
3 edge <1,2> =3
4 edge <2,5> =3
5 edge <1,4> =4

Minimum cost = 13
```

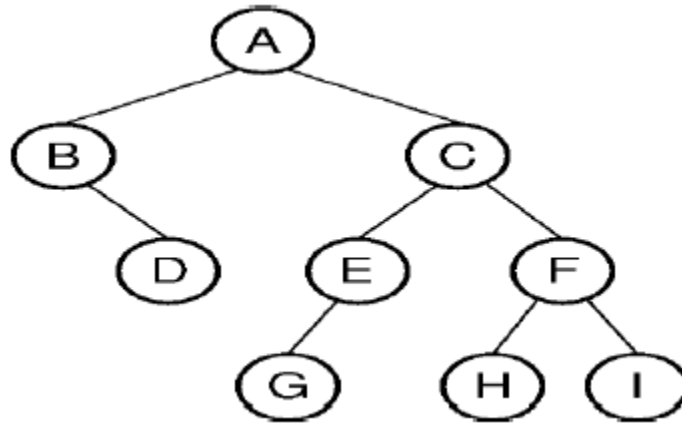
7.7 LAB VIVA QUESTIONS:

1. What is the time complexity of Kruskal's algorithm.
2. Define spanning tree.
3. Define minimum cost spanning tree

WEEK-8 TREE TRAVERSALS

8.1 OBJECTIVE:

Perform various tree traversal algorithms for a given tree.



8.2 RESOURCES:

Dev C++

8.3 PROGRAM LOGIC:

Traversal is a process to visit all the nodes of a tree and may print their values too.

Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. The following are the generally used methods for traversing trees:

8.4 Example:

InOrder(root) visits nodes in the following order:

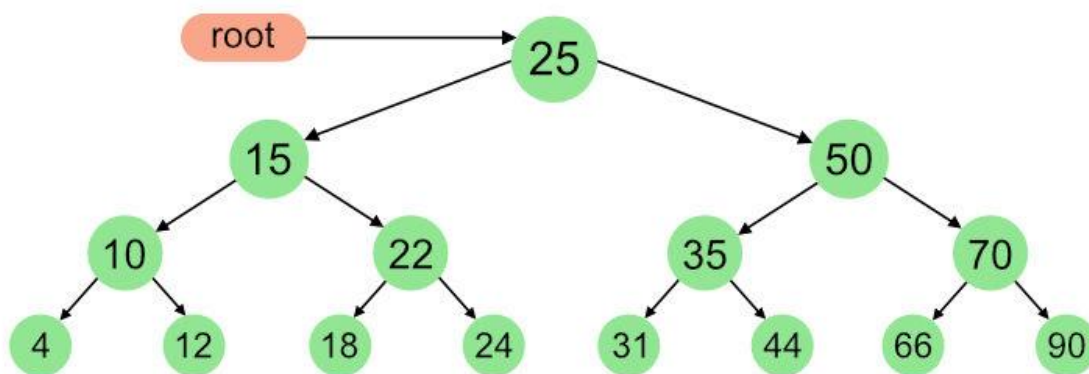
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Inorder Traversal (Practice):

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call *Inorder(left->subtree)*
2. Visit the root.
3. Traverse the right subtree, i.e., call *Inorder(right->subtree)*

Uses of Inorder Traversal:

In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.

Example: In order traversal for the above-given figure is 4 2 5 1 3.

8.5.1 SOURCE CODE:

```
// C++ program for different tree traversals
#include <iostream>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node {
    int data;
    struct Node *left, *right;
};

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    cout << node->data << " ";

    /* now recur on right child */
    printInorder(node->right);
}

/* Driver code*/
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    // Function call
    cout << "\nInorder traversal of binary tree is \n";
    printInorder(root);
}
```

```
    return 0;
}
```

Output

```
Inorder traversal of binary tree is
4 2 5 1 3
```

Time Complexity: $O(N)$

Preorder Traversal ([Practice](#)):

Algorithm *Preorder*(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call *Preorder*(left->subtree)
3. Traverse the right subtree, i.e., call *Preorder*(right->subtree)

Uses of Preorder:

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expressions on an expression tree.

Example: Preorder traversal for the above-given figure is 1 2 4 5 3.

8.5.2 SOURCE CODE:

```
// C++ program for different tree traversals
#include <iostream>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node {
    int data;
    struct Node *left, *right;
};

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;
```



```

/* first print data of node */
cout << node->data << " ";

/* then recur on left subtree */
printPreorder(node->left);

/* now recur on right subtree */
printPreorder(node->right);
}

/* Driver code*/
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    // Function call
    cout << "\nPreorder traversal of binary tree is \n";
    printPreorder(root);

    return 0;
}

```

Output

```

Preorder traversal of binary tree is
1 2 4 5 3

```

Time Complexity: $O(N)$

Postorder Traversal ([Practice](#)):

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left->subtree)
2. Traverse the right subtree, i.e., call Postorder(right->subtree)
3. Visit the root

Uses of Postorder:

Postorder traversal is used to delete the tree. Please see [the question for the deletion of a tree](#) for details. Postorder traversal is also useful to get the postfix expression of an expression tree

Example: Postorder traversal for the above-given figure is 4 5 2 3 1

8.5.3 SOURCE CODE:

```
// C++ program for different tree traversals
#include <iostream>
using namespace std;

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct Node {
    int data;
    struct Node *left, *right;
};

// Utility function to create a new tree node
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

/* Given a binary tree, print its nodes according to the
"bottom-up" postorder traversal. */
void printPostorder(struct Node* node)
{
    if (node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);

    // now deal with the node
    cout << node->data << " ";
}

/* Driver code*/
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
```

```
root->left->left = newNode(4);  
root->left->right = newNode(5);
```

```
    // Function call  
    cout << "\nPostorder traversal of binary tree is \n";  
    printPostorder(root);
```

```
    return 0;
```

```
}
```

Output

```
Postorder traversal of binary tree is  
4 5 2 3 1
```

Time Complexity: $O(N)$

8.7 LAB VIVA QUESTIONS:

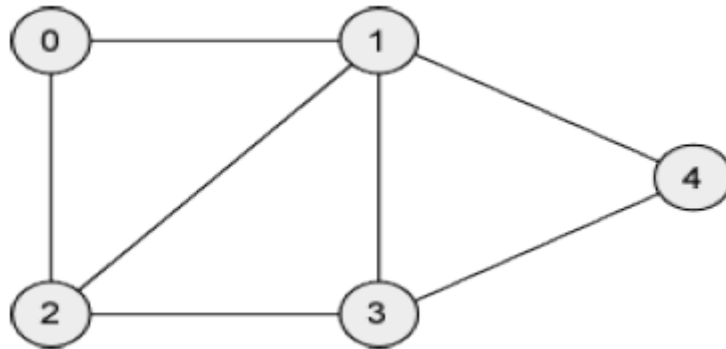
1. Define binary tree.
2. List different tree traversals.
3. Explain inorder travels with example.
4. Explain preorder travels with example.
5. Explain postorder travels with example.

WEEK-9

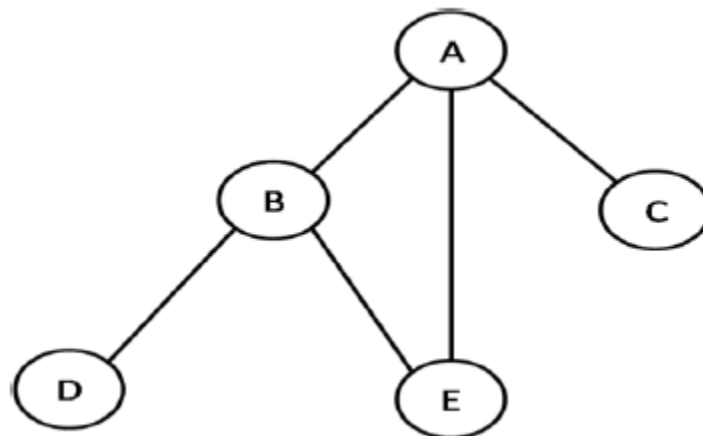
GRAPH TRAVERSALS

9.1 OBJECTIVE:

1. Print all the nodes reachable from a given starting node in a digraph using BFS method.



2. Check whether a given graph is connected or not using DFS method.



9.2 RESOURCES:

Dev C++

9.3 PROGRAM LOGIC:

Breadth first traversal

Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search.

1. Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
2. If no adjacent vertex is found, remove the first vertex from the queue.
3. Repeat Rule 1 and Rule 2 until the queue is empty.

Depth first traversal

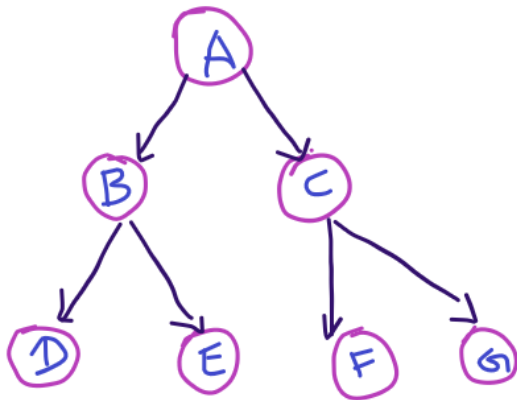
Depth First Search (DFS) algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search.

1. Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
2. If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
3. Repeat Rule 1 and Rule 2 until the stack is empty.

Time Complexity: $O(V+E)$

Where V is the number of vertices and E is the number of edges in the graph.

Example



BFS - ABCDEFG

DFS - ABDECFG

9.4 PROCEDURE:

1. Create: Open Dev C++, write a program after that save the program with .c extension.
2. Compile: Alt + F9
3. Execute: Ctrl + F10

9.5.1 SOURCE CODE:

```
//Breadth first traversal
#include<iostream>

int a[20][20],q[20],visited[20],n,i,j,f=-1,r=0;
void bfs(int v){
    q[++r]=v;
    visited[v]=1;
    while(f<=r) {
        for(i=1;i<=n;i++){
            if(a[v][i] && !visited[i]){
                visited[i]=1;
                q[++r]=i;
            }
        }
        f++;
        v=q[f];
    }
}
int main(){
    int v;
    printf("\n Enter the number of vertices:");
    scanf("%d",&n);
    for(i=1;i<=n;i++){
        q[i]=0;
        visited[i]=0;
    }
    printf("\n Enter graph data in matrix form:\n");
    for(i=1;i<=n;i++){
        for(j=1;j<=n;j++){
            scanf("%d",&a[i][j]);
        }
    }
    printf("\n Enter the starting vertex:");
    scanf("%d",&v);
    bfs(v);

    printf("\n The node which are reachable are:\n");
    for(i=1;i<=n;i++){
        if(visited[i])
            printf("%d\t",q[i]);
        else
            printf("\n Bfs is not possible");
    }
}
```

}

Output

```
Enter the number of vertices:5

Enter graph data in matrix form:
0 1 1 0 0
1 0 1 1 1
1 1 0 1 0
0 1 1 0 1
0 1 0 1 0

Enter the starting vertex:1

The node which are reachable are:
1      2      3      4      5

...Program finished with exit code 0
Press ENTER to exit console.
```

9.5.2 SOURCE CODE:

//Checking whether a given graph is connected or not using DFS method
#include<iostream>

```
int a[20][20],reach[20],n;
void dfs(int v){
int i; reach[v]=1;
for(i=1;i<=n;i++){
if(a[v][i] && !reach[i]) {
printf("\n %d->%d",v,i);
dfs(i);
}
};
int main(){
int i,j,count=0;
printf("\n Enter number of vertices:");
scanf("%d",&n);
for(i=1;i<=n;i++){
reach[i]=0;
for(j=1;j<=n;j++){
a[i][j]=0;
}
printf("\n Enter the adjacency matrix:\n");
for(i=1;i<=n;i++){
for(j=1;j<=n;j++){
scanf("%d",&a[i][j]);
}
}
dfs(1);
printf("\n");
```

```

for(i=1;i<=n;i++){
if(reach[i])
count++;
}
if(count==n)
printf("\n Graph is connected");
else
printf("\n Graph is not connected");
}

```

Output

```

Enter number of vertices:5

Enter the adjacency matrix:
0 1 1 0 0
1 0 1 1 1
1 1 0 1 0
0 1 1 0 1
0 1 0 1 0

1->2
2->3
3->4
4->5

Graph is connected

...Program finished with exit code 0
Press ENTER to exit console.

```

9.6 LAB VIVA QUESTIONS:

1. Define graph, connected graph.
2. List the different graph traversals.
3. Explain DFS traversal.
4. Explain BFS traversal.
5. What are the time complexities of BFS and DFS algorithms?

WEEK-10

SUM OF SUB SETS PROBLEM

10.1 OBJECTIVE:

Find a subset of a given set $S = \{s_1, s_2, \dots, s_n\}$ of n positive integers whose sum is equal to a given positive integer d . For example, if $S = \{1, 2, 5, 6, 8\}$ and $d = 9$ there are two solutions $\{1, 2, 6\}$ and $\{1, 8\}$. A suitable message is to be displayed if the given problem instance doesn't have a solution.

The SUBSET-SUM problem involves determining whether or not a subset from a list of integers can sum to a target value. For example, consider the list of `nums = [1, 2, 3, 4]`. If the `target = 7`, there are two subsets that achieve this sum: $\{3, 4\}$ and $\{1, 2, 4\}$. If `target = 11`, there are no solutions.

10.2 RESOURCES:

Dev C++

10.3 PROGRAM LOGIC:

Given a set of non-negative integers, and a value *sum*, determine if there is a subset of the given set with sum equal to given *sum*.

10.4 PROCEDURE:

1. Create: Open Dev C++, write a program after that save the program with .c extension.
2. Compile: Alt + F9
3. Execute: Ctrl + F10

10.5 SOURCE CODE:

```
#include<iostream>
#define TRUE 1
#define FALSE 0
int inc[50],w[50],sum,n;
void sumset(int ,int ,int);
int promising(int i,int wt,int total) {
return (((wt+total)>=sum)&&((wt==sum)||((wt+w[i+1]<=sum))));
}
int main() {
int i,j,n,temp,total=0;
printf("\n Enter how many numbers: ");
scanf("%d",&n);
printf("\n Enter %d numbers : ",n);
for (i=0;i<n;i++) {
scanf("%d",&w[i]);
total+=w[i];
}
```

```

printf("\n Input the sum value to create sub set: ");
scanf("%d",&sum);
for (i=0;i<=n;i++)
for (j=0;j<n-1;j++)

if(w[j]>w[j+1]) {
temp=w[j];
w[j]=w[j+1];
w[j+1]=temp;
}
printf("\n The given %d numbers in ascending order: ",n);
for (i=0;i<n;i++)
printf("%3d",w[i]);
if((total<sum))
printf("\n Subset construction is not possible");
else{
for (i=0;i<n;i++)
inc[i]=0;
printf("\n The solution using backtracking is:\n");
sumset(-1,0,total);
}
}

void sumset(int i,int wt,int total){
int j;
if(promising(i,wt,total)) {
if(wt==sum){
printf("\n{");
for (j=0;j<=i;j++)
if(inc[j])
printf("%3d",w[j]);
printf(" }\n");
} else {
inc[i+1]=TRUE;
sumset(i+1,wt+w[i+1],total-w[i+1]);
inc[i+1]=FALSE;
sumset(i+1,wt,total-w[i+1]);
}
}
}
}

```

Output

```
Enter how many numbers: 5
Enter 5 numbers : 1 2 6 5 8
Input the sum value to create sub set: 9
The given 5 numbers in ascending order: 1 2 5 6 8
The solution using backtracking is:
{ 1 2 6 }
{ 1 8 }
```

10.6 LAB VIVA QUESTIONS:

1. Define is Back-Tracking.
2. Explain Sum of subset problem.
3. What is time complexity of sum of subset problem?

WEEK-11

TRAVELLING SALES PERSON PROBLEM

11.1 OBJECTIVE:

Implement any scheme to find the optimal solution for the Traveling Sales Person problem and then solve the same problem instance using any approximation algorithm and determine the error in the approximation

The traveling salesman problem (TSP) is an algorithmic problem tasked with finding the shortest route between a set of points and locations that must be visited. In the problem statement, the points are the cities a salesperson might visit.

11.2 RESOURCES:

Dev C++

11.3 PROGRAM LOGIC:

1. Check for the disconnection between the current city and the next city
2. Check whether the travelling sales person has visited all the cities
3. Find the next city to be visited
4. Find the solution and terminate

11.4 PROCEDURE:

1. Create: Open Dev C++, write a program after that save the program with .c extension.
2. Compile: Alt + F9
3. Execute: Ctrl + F10

11.5 SOURCE CODE:

```
#include<stdio.h>
int s,c[100][100],ver;
float optimum=999,sum;
/* function to swap array elements */
void swap(int v[], int i, int j) {
    int t;
    t = v[i];
    v[i] = v[j];
    v[j] = t;
}
/* recursive function to generate permutations */
void brute_force(int v[], int n, int i) {
    // this function generates the permutations of the array from element i to element n-1
    int j,sum1,k;
    //if we are at the end of the array, we have one permutation
    if (i == n) {
```

```

if(v[0]==s) {
for (j=0; j<n; j++)
printf ("%d ", v[j]);
sum1=0;

for( k=0;k<n-1;k++) {
sum1=sum1+c[v[k]][v[k+1]];
}
sum1=sum1+c[v[n-1]][s];
printf("sum = %d\n",sum1);
if (sum1<optimum)
optimum=sum1;
}
}
else
// recursively explore the permutations starting at index i going through index n-1*/
for (j=i; j<n; j++) { /* try the array with i and j switched */
swap (v, i, j);
brute_force (v, n, i+1);
/* swap them back the way they were */
swap (v, i, j);
}
}
int nearest_neighbour(int ver) {
int min,p,i,j,vis[20],from;
for(i=1;i<=ver;i++)
vis[i]=0;
vis[s]=1;
from=s;
sum=0;
for(j=1;j<ver;j++) {
min=999;
for(i=1;i<=ver;i++)
if(vis[i] !=1 && c[from][i]<min && c[from][i] !=0 ) {
min= c[from][i];
p=i;
}
vis[p]=1;
from=p;
sum=sum+min;
}
sum=sum+c[from][s];
}
int main () {
int ver,v[100],i,j;
printf("Enter n : ");
scanf("%d",&ver);
for (i=0; i<ver; i++)
v[i] = i+1;
printf("Enter cost matrix\n");
for(i=1;i<=ver;i++)
for(j=1;j<=ver;j++)
scanf("%d",&c[i][j]);

```

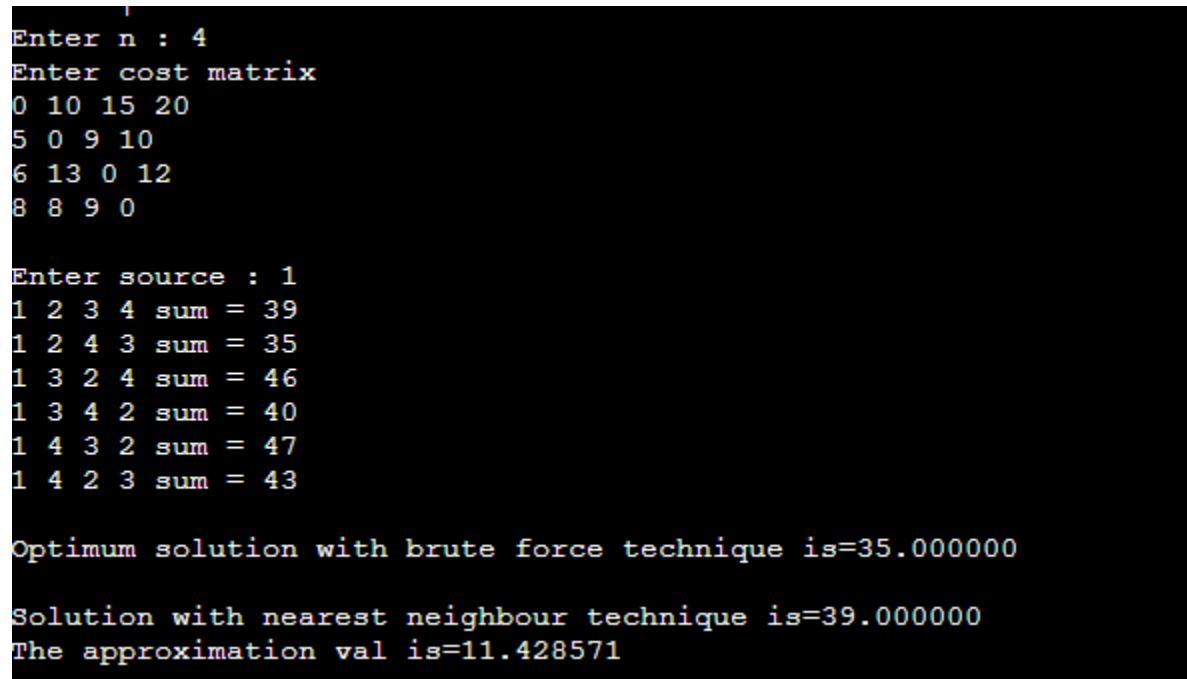
```

printf("\nEnter source : ");
scanf("%d",&s);

brute_force (v, ver, 0);
printf("\nOptimum solution with brute force technique is=%f\n",optimum);
nearest_neighbour(ver);
printf("\nSolution with nearest neighbour technique is=%f\n",sum);
printf("The approximation val is=%f",((sum/optimum)-1)*100);
printf(" % ");
}

```

Outbut



```

Enter n : 4
Enter cost matrix
0 10 15 20
5 0 9 10
6 13 0 12
8 8 9 0

Enter source : 1
1 2 3 4 sum = 39
1 2 4 3 sum = 35
1 3 2 4 sum = 46
1 3 4 2 sum = 40
1 4 3 2 sum = 47
1 4 2 3 sum = 43

Optimum solution with brute force technique is=35.000000

Solution with nearest neighbour technique is=39.000000
The approximation val is=11.428571

```

11.6 LAB VIVA QUESTIONS:

1. Define Optimal Solution.
2. Explain Travelling Sales Person Problem.
3. What is the time complexity of Travelling Sales Person Problem?