TPCT's

College of Engineering, Osmanabad

Laboratory Manual

Design and Analysis of Algorithm

For

Third Year Students(CSE)

Manual Prepared by

Prof. Suyoga. Y. Bansode

Author COE, Osmanabad

TPCT's

College of Engineering

Solapur Road, Osmanabad

Department of Computer Science & Engineering

Vision of the Department:

To achieve and evolve as a center of academic excellence and research center in the field of Computer Science and Engineering. To develop computer engineers with necessary analytical ability and human values who can creatively design, implement a wide spectrum of computer systems for welfare of the society.

Mission of the Department:

The department strives to continuously engage in providing the students with in-depth understanding of fundamentals and practical training related to professional skills and their applications through effective Teaching-Learning Process and state of the art laboratories pertaining to CSE and interdisciplinary areas. Preparing students in developing research, design, entrepreneurial skills and employability capabilities.

College of Engineering

Technical Document

This technical document is a series of Laboratory manuals of Computer Science and Engineering Department and is a certified document of College of engineering, Osmanabad. The care has been taken to make the document error-free. But still if any error is found, kindly bring it to the notice of subject teacher and HOD.

Recommended by,

HOD

Approved by,

Principal

FOREWORD

It is my great pleasure to present this laboratory manual for third year engineering students for the subject of Design and Analysis of Algorithm to understand the paradigms and approaches used to analyze and design algorithms and to appreciate the impact of algorithm design in practice.

It also ensures that students understand how the worst-case time complexity of an algorithm is computed. This being a core subject, it becomes very essential to have clear theoretical and designing aspects.

This lab manual provides a platform to the students for understanding the basic concepts of designing and analyzing of algorithm. This practical background will help students to gain confidence in qualitative and quantitative approach to synthesize and analyze the algorithms.

H.O.D
CSE Dept

LABORATORY MANUAL CONTENTS

This manual is intended for the Third Year students of CSE branch in the subject of Design and Analysis of Algorithm. This manual typically contains practical/ Lab Sessions related to Design and Analysis of Algorithm covering various aspects related to the subject for enhanced understanding.

Students are advised to thoroughly go through this manual rather than only topics mentioned in the syllabus as practical aspects are the key to understanding and conceptual visualization of theoretical aspects covered in the books.

SUBJECT INDEX:

Dos and Don'ts in Laboratory :-

1. Turn off the machine once you are done using it.
2. Student should not attempt to repair, open, tamper or interfere with any of the computer, printing, cabling or other equipment in the laboratory.
3. Make entry in the Log Book as soon as you enter in the laboratory.
4. All the students are supposed to enter the terminal number in the log book.
5. Handle equipments with care.
6. Strictly observe the instructions given by the Teacher/ Lab Instructor.
7. Do not change the terminal on which you are working.
8. Do not install/remove any software on system without permission.
9. Do not open any irrelevant internet sites on lab computer.
10. Do not remove anything from computer laboratory without permission.
11. Do not misbehave in the computer laboratory.
12. Do not plug in external devices without scanning them for computer viruses.

Instruction for Laboratory Teachers:-

1. Submission related to whatever lab work has been completed should be done during the next lab session.
2. Students should be instructed to start the computers. After the experiment is over, the students must shut down the Computers and turn off the switches.
3. The promptness of submission should be encouraged by way of marking and evaluation patterns that will benefit the sincere students.

Experiment No.1
Heap sort

Aim:- Program to implement Heap sort

Objective: To write a C program to perform Heap sort using the divide and conquer technique

Theory: A max (min) heap is complete binary tree with the property that the value at each node is at least as large as (as small as) the values at its children (if they exist) Call this property the heap property.

Algorithm:
Step 1: Start the process.
Step 2: Declare the variables.
Step 3: Enter the list of elements to be sorted using the get function.
Step 4: Divide the array list into two halves the lower array list and upper array list using the merge sort function.
Step 5: Sort the two array list.
Step 6: Combine the two sorted arrays.
Step 7: Display the sorted elements using the get() function.
Step 8: Stop the process

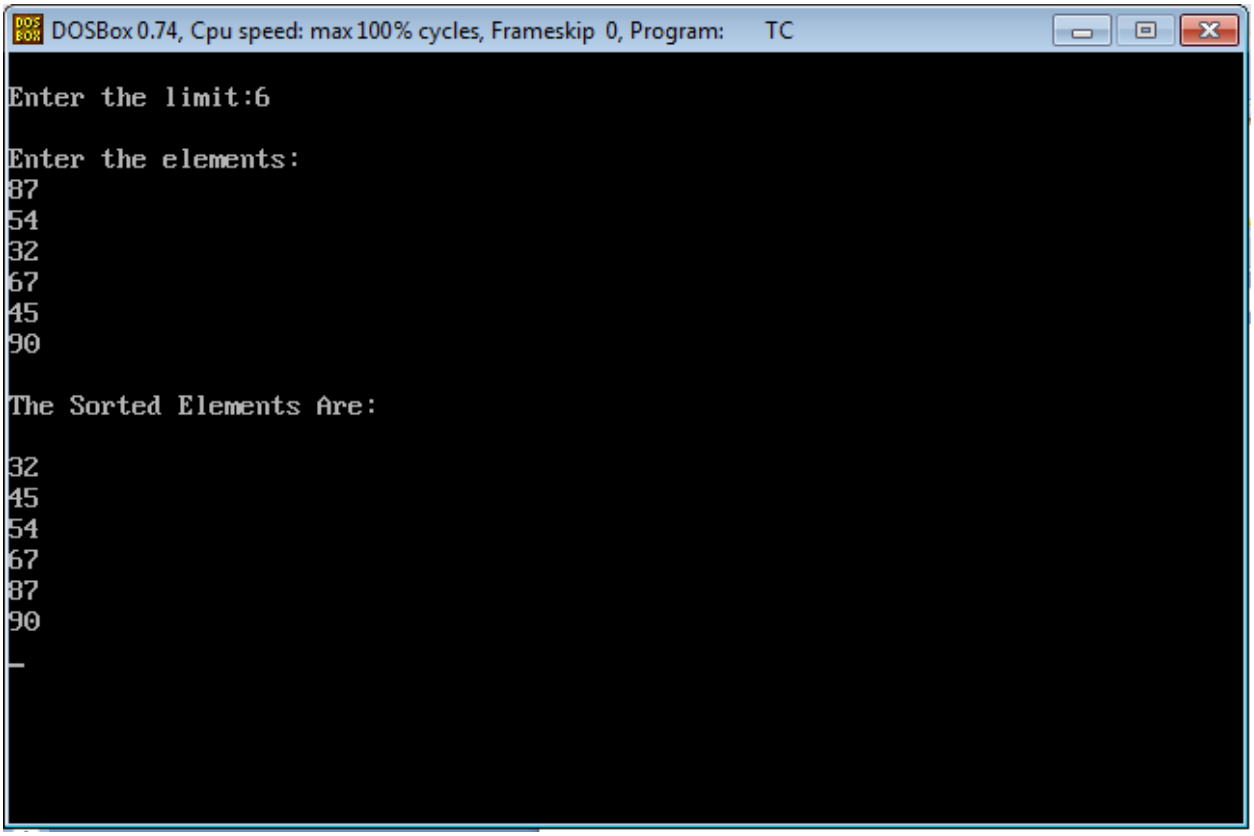
/* Program for Heapsort*/

```
#include<stdio.h>
void heapsort(int[],int);
void heapify(int[],int);
void adjust(int[],int);
void main() {
        int n,i,a[50];
        clrscr();
        printf("\nEnter the limit:");
        scanf("%d",&n);
        printf("\nEnter the elements:");
        for (i=0;i<n;i++)
          scanf("%d",&a[i]);
        heapsort(a,n);
        printf("\nThe Sorted Elements Are:\n");
        for (i=0;i<n;i++)
```

```
        printf("\n %d",a[i]);
    printf("\n");
      getch();
}
void heapsort(int a[],int n) {
    int i,t;
    heapify(a,n);
    for (i=n-1;i>0;i--) {
        t = a[0];
        a[0] = a[i];
        a[i] = t;
        adjust(a,i);
    }
}
void heapify(int a[],int n) {
    int k,i,j,item;
    for (k=1;k<n;k++) {
        item = a[k];
        i = k;
        j = (i-1)/2;
        while((i>0)&&(item>a[j])) {
            a[i] = a[j];
            i = j;
            j = (i-1)/2;
        }
        a[i] = item;
    }
}
void adjust(int a[],int n) {
    int i,j,item;
    j = 0;
    item = a[j];
    i = 2*j+1;
    while(i<=n-1) {
        if(i+1 <= n-1)
          if(a[i] <a[i+1])
           i++;
        if(item<a[i]) {
            a[j] = a[i];
            j = i;
```

```
                    i = 2*j+1;
                } else
                  break;
        }
        a[j] = item;
}
```

OUTPUT:



Time complexities: - Though the call of Heapify requires only *O(n)* operations, Adjust possibly requires *O(log n)* operations for each invocation. Thus the worst case time is *O(n log n).*

Conclusion: Hence we have studied and executed the program for Heap sort.

Experiment No.2
Binary Search

Aim:- Program to find the given element in Binary Search Tree.

Objective: To write a C program to perform binary search using the divide and conquer technique.

Theory: A binary search tree is a binary tree. It may be empty. If it is not empty, then it satisfies the following properties:
1. Every element has a key and no two elements have the same key.
2. The keys (if any) in the left sub tree are smaller than the key in the root.
3. The keys (if any) in the right sub tree are larger than the key in the root.
4. The left and right sub trees are also binary search trees.

Algorithm:

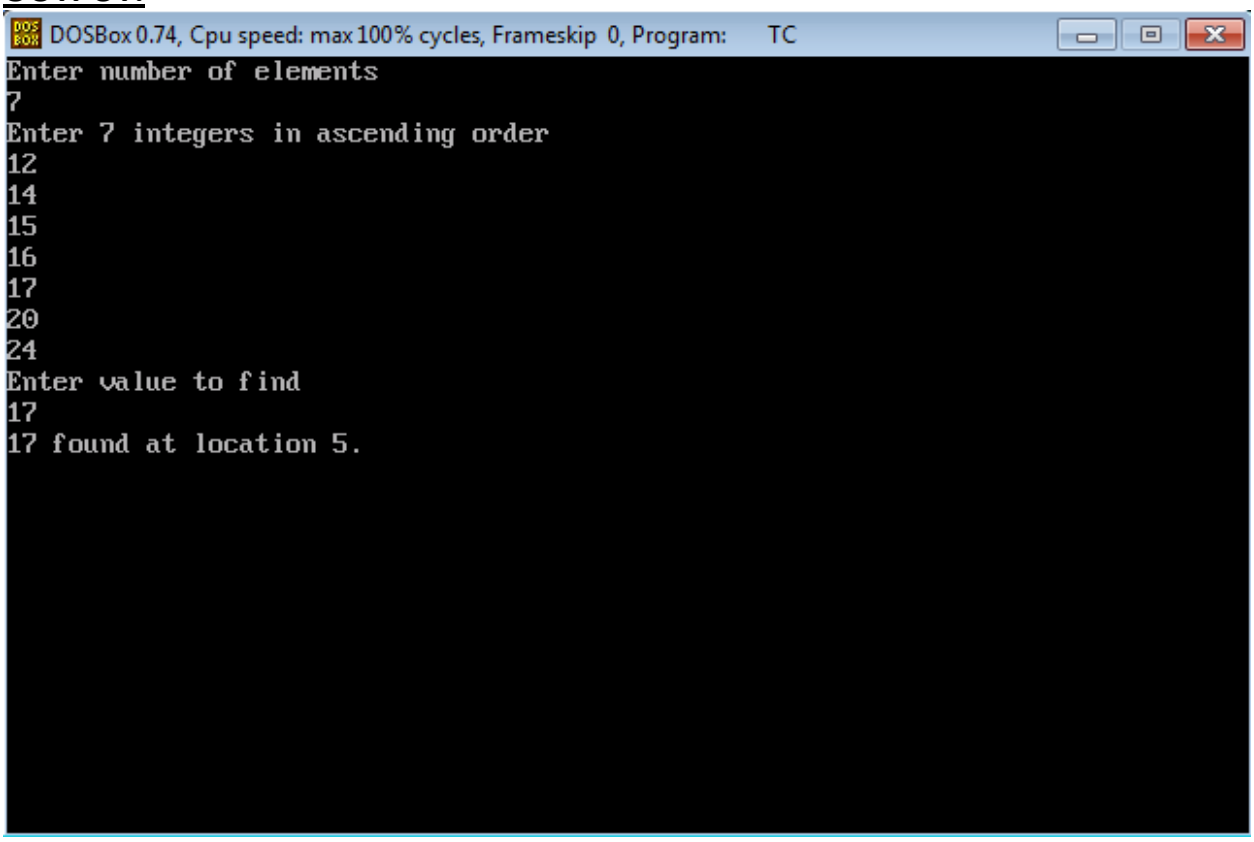
Step 1: Start the process.
Step 2: Declare the variables.
Step 3: Enter the list of elements to be searched using the get function.
Step 4: Divide the array list into two halves the lower array list and upper arraylist.
Step 5: It works by comparing a search key k with the arrays middle element a[m].
Step 6: If they match the algorithm stops; otherwise the same operation is repeated recursively for the first half of the array if k < a[m] and the second half if k > a[m].
Step 7: Display the searching results
Step 8: Stop the process.

/* program for binary search*/

```
#include <stdio.h>

Void main()
{
int c, first, last, middle, n, search, array[100];

printf("Enter number of elements\n");
scanf("%d",&n);

printf("Enter %d integers\n", n);
```

```
for(c =0; c < n;c++)
scanf("%d",&array[c]);

printf("Enter value to find\n");
scanf("%d",&search);

  first =0;
  last = n -1;
  middle =(first+last)/2;

while(first <= last){
if(array[middle]< search)
       first = middle +1;
else if(array[middle]== search){
printf("%d found at location %d.\n", search, middle+1);
break;
}
else
       last = middle -1;

    middle =(first + last)/2;
}
if(first > last)
printf("Not found! %d is not present in the list.\n", search);

getch();
}
```

OUTPUT:



Time complexities: -For Searching the element the worst case time complexity is *O(n)* and average case is *O(log n).*

Conclusion: Hence we have found the given element in the binary search tree.

Experiment No.3
Quick Sort

Aim:- Program to implement Quick sort.

Objective: To write a C program to perform Quick sort using the divide and conquer technique

Theory: Quick Sort divides the array according to the value of elements. It rearranges elements of a given array A[0..n-1] to achieve its partition, where the elements before position s are smaller than or equal to A[s] and all the elements after position s are greater than or equal to A[s].

Algorithm:

Step 1: Start the process.
Step 2: Declare the variables.
Step 3: Enter the list of elements to be sorted using the get()function.
Step 4: Divide the array list into two halves the lower array list and upper array list using the merge sort function.
Step 5: Sort the two array list.
Step 6: Combine the two sorted arrays.
Step 7: Display the sorted elements.
Step 8: Stop the process.

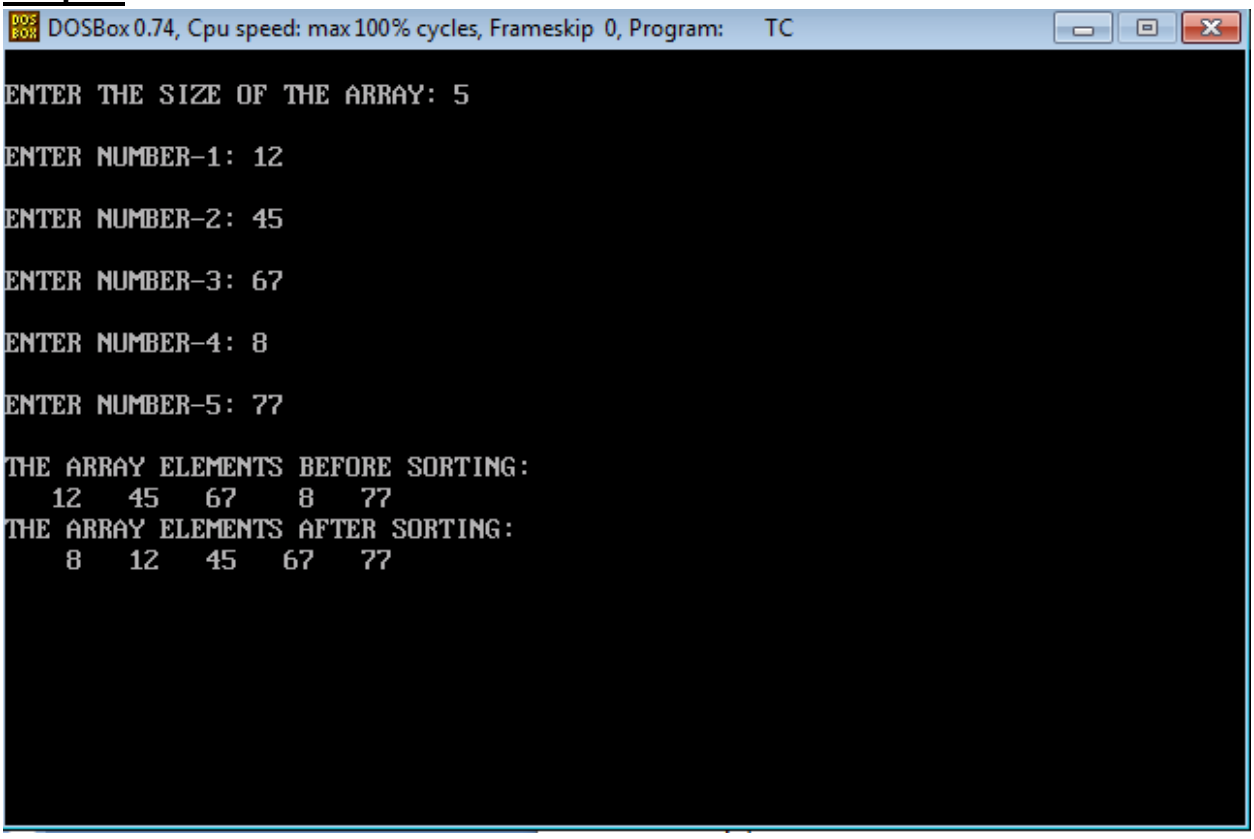
// C program to sort an array using Quick Sort Algorithm //

```
#include <stdio.h>
#include <conio.h>
void qsort();
int n;
void main()
{
        int a[100],i,l,r;
        clrscr();
        printf("\nENTER THE SIZE OF THE ARRAY: ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\nENTER NUMBER-%d: ",i+1);
```

```
		scanf("%d",&a[i]);
	}
	printf("\nTHE ARRAY ELEMENTS BEFORE SORTING: \n");
	for(i=0;i<n;i++)
	{
		printf("%5d",a[i]);
	}
	l=0;
	r=n-1;
	qsort(a,l,r);
	printf("\nTHE ARRAY ELEMENTS AFTER SORTING: \n");
	for(i=0;i<n;i++)
		printf("%5d",a[i]);
	getch();
}
void qsort(int b[],int left,int right)
{
	int i,j,p,tmp,finished,k;
	if(right>left)
	{
		i=left;
		j=right;
		p=b[left];
		finished=0;
		while (!finished)
		{
			do
			{
				++i;
			}
			while ((b[i]<=p) && (i<=right));
			while ((b[j]>=p) && (j>left))
			{
				--j;
			}
			if(j<i)
				finished=1;
			else
			{
				tmp=b[i];
```

```
                        b[i]=b[j];
                        b[j]=tmp;
                    }
                }
                tmp=b[left];
                b[left]=b[j];
                b[j]=tmp;
                qsort(b,left,j-1);
                qsort(b,i,right);
        }
        return;
}
```

Output:



Time complexities: -Quick sort has an average time of *O(n log n)* on n elements. Its worst case time is $O(n^2)$

Conclusion: Thus the C program to perform Quick sort using the divide and conquer technique has been completed successfully

Experiment No.4
Maximum and minimum

Aim:- Program to find out maximum and minimum using divide and conquer rule.

Objective: To write a C program to find out maximum and minimum using the divide and conquer technique

Theory: The problem is to find the maximum and minimum number using the divide and conquer method.

Algorithm:

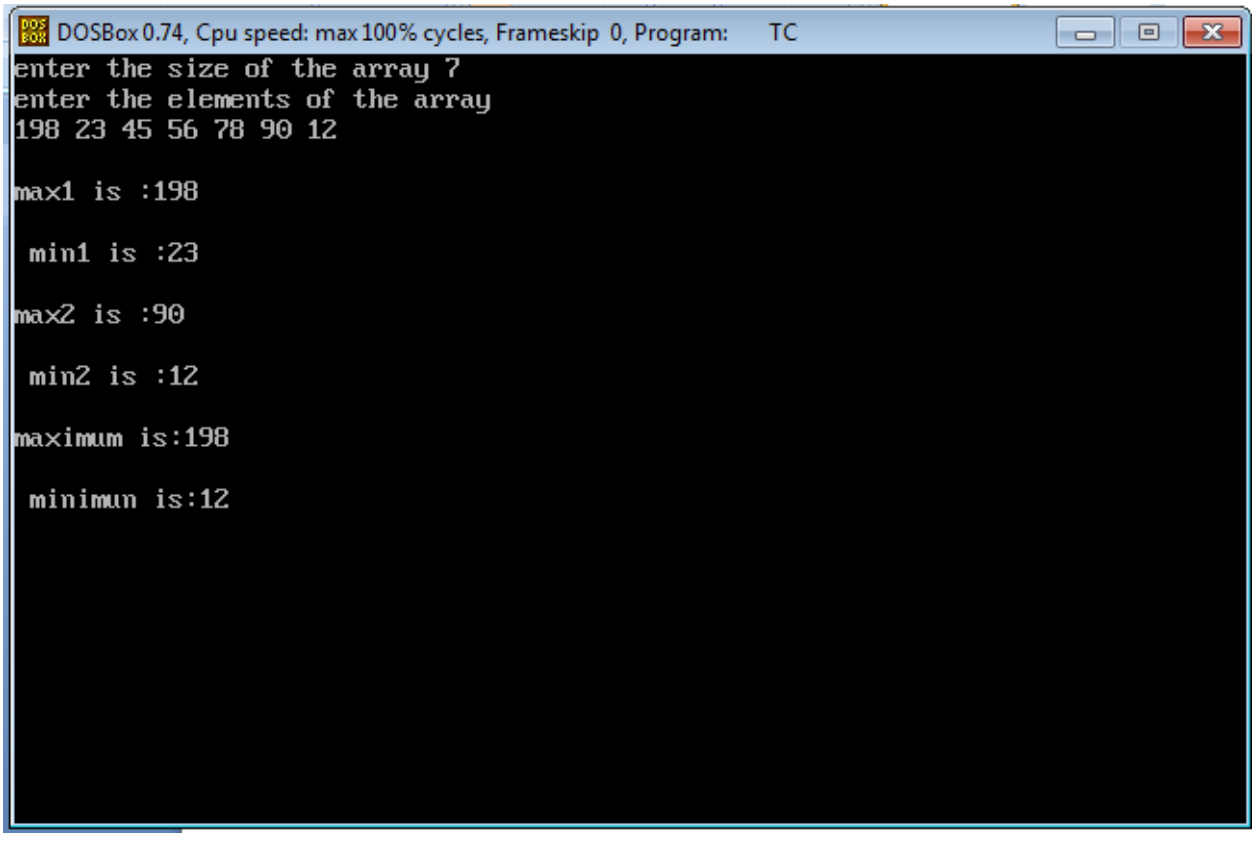
```
1   Algorithm MaxMin(i, j, max, min)
2   // a[1 : n] is a global array. Parameters i and j are integers,
3   // 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
4   // largest and smallest values in a[i : j], respectively.
5   {
6       if (i = j) then max := min := a[i]; // Small(P)
7       else if (i = j − 1) then  // Another case of Small(P)
8           {
9               if (a[i] < a[j]) then
10              {
11                  max := a[j]; min := a[i];
12              }
13              else
14              {
15                  max := a[i]; min := a[j];
16              }
17          }
18          else
19          {   // If P is not small, divide P into subproblems.
20              // Find where to split the set.
21                  mid := ⌊(i + j)/2⌋;
22              // Solve the subproblems.
23                  MaxMin(i, mid, max, min);
24                  MaxMin(mid + 1, j, max1, min1);
25              // Combine the solutions.
26                  if (max < max1) then max := max1;
27                  if (min > min1) then min := min1;
28          }
29  }
```

```
//Program to find out max and min using divide and conquer rule

#include<conio.h>
#include<stdio.h>
int main()
{
int i,j,a[100],max=0,min=1000,mid,n,max1,max2,min1,min2;
printf("enter the size of the array");
scanf("%d",&n);
printf("enter the elements of the array");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
j=n-1;
int p=0;
if(p==j)
{
max=min=a[p];//wen der is only one element in array
printf("max is:%d and min is :%d",max,min);
}
else if(p==j-1)
{
if(a[p]<a[j])
{
max=a[j];
min=a[p];
}
else
{
max=a[p];
min=a[j];
}
printf("max is:%d and min is :%d",max,min);
}
else
{
mid=(int)((p+j)/2);
for(i=0;i<mid;i++)
{
```

```
if(a[i]>max)
{
max=a[i];
}
if(a[i]<min)
{
min=a[i];
}
}
max1=max;
min1=min;
printf("\nmax1 is :%d\n",max1);
printf("\n min1 is :%d\n",min1);
max=0;
min=1000;
for(i=mid;i<n;i++)
{
if(a[i]>max)
{
max=a[i];
}
if(a[i]<min)
{
min=a[i];
}
}
max2=max;
min2=min;
printf("\nmax2 is :%d\n",max2);
printf("\n min2 is :%d\n",min2);

if(max1<max2)
{
max=max2;
}
else
max=max1;
if(min1<min2)
{
min=min1;
```

```
}
else
{
min=min2;
}
printf("\nmaximum is:%d\n",max);
printf("\n minimun is:%d\n",min);
}
getch();
return 0;
}
```

OUTPUT:



Conclusion: Thus the Program to find out maximum and minimum using divide and conquer rule has been completed successfully.

Experiment No.5
Merge Sort

Aim:- Program to implement Merge Sort.

Objective: To write a C program to perform merge sort using the divide and conquer technique.

Theory: .As another example of Divide and conquer a sorting algorithm that in the worst case its complexity is O(n log n). This algorithm is called Merge Sort. Merge sort describes this process using recursion and a function Merge which merges two sorted sets.

Algorithm:

Merge sort is a perfect example of a successful application of the divide-and-conquer technique.

1. Split array A[1..*n*] in two and make copies of each half in arrays B[1.. *n*/2] and C[1.. *n*/2]
2. Sort arrays B and C
3. Merge sorted arrays B and C into array A as follows:
   a) Repeat the following until no elements remain in one of the arrays:
      i. compare the first elements in the remaining unprocessed portions of the arrays
      ii. copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
   b) Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

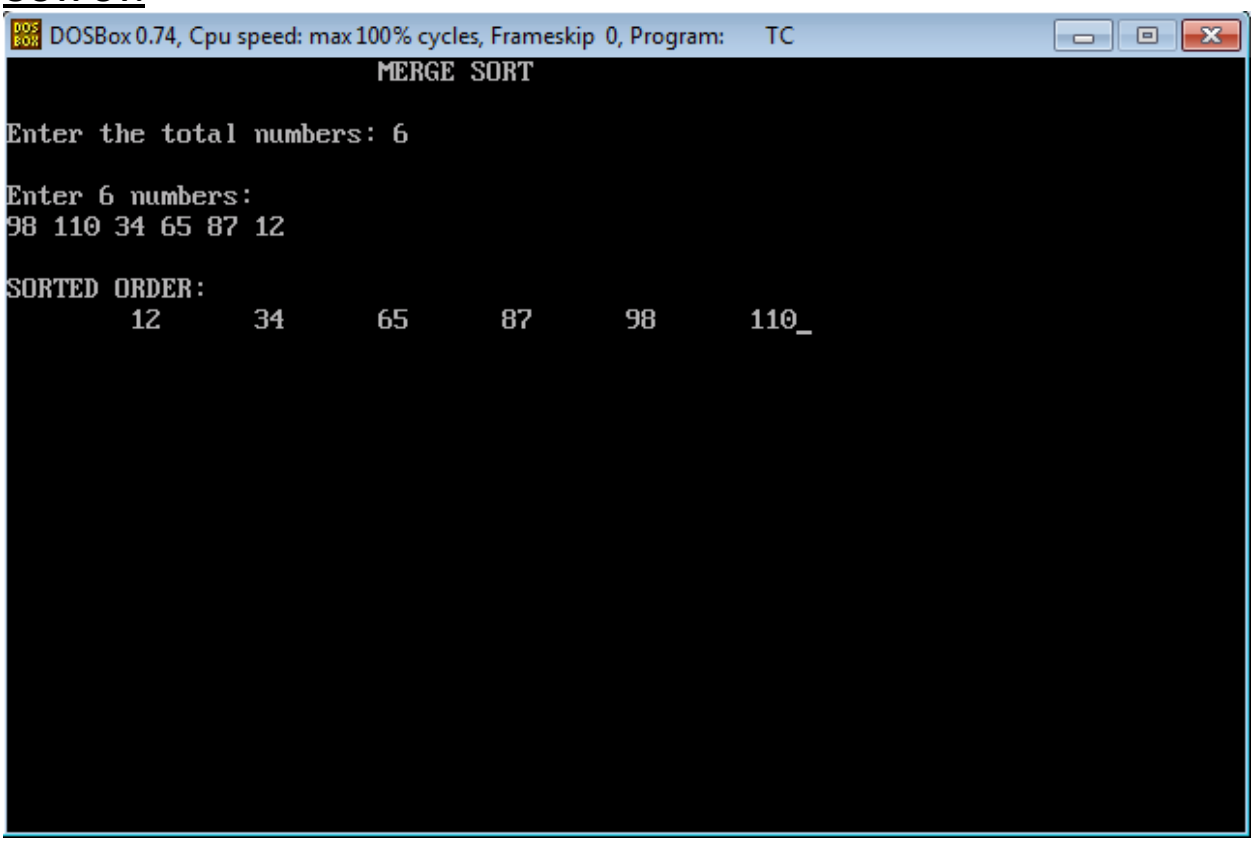
//Program to implement merge sort using Divide and conquer

```
#include<stdio.h>
#include<conio.h>
int a[50];
void merge(int,int,int);
void merge_sort(int low,int high)
{
 int mid;
 if(low<high)
 {
  mid=(low+high)/2;
  merge_sort(low,mid);
```

```
  merge_sort(mid+1,high);
  merge(low,mid,high);
 }
}
void merge(int low,int mid,int high)
{
 int h,i,j,b[50],k;
 h=low;
 i=low;
 j=mid+1;
 while((h<=mid)&&(j<=high))
 {
  if(a[h]<=a[j])
  {
   b[i]=a[h];
   h++;
  }
  else
  {
   b[i]=a[j];
   j++;
  }
  i++;
 }
 if(h>mid)
 {
  for(k=j;k<=high;k++)
  {
   b[i]=a[k];
   i++;
  }
 }
 else
 {
  for(k=h;k<=mid;k++)
  {
   b[i]=a[k];
   i++;
  }
 }
```

```
 for(k=low;k<=high;k++) a[k]=b[k];
}
int main()
{
 int num,i;
 printf("\t\t\tMERGE SORT\n");
 printf("\nEnter the total numbers: ");
 scanf("%d",&num);
 printf("\nEnter %d numbers: \n",num);
 for(i=1;i<=num;i++)
 {
  scanf("%d",&a[i]);
}
 merge_sort(1,num);
 printf("\nSORTED ORDER: \n");
 for(i=1;i<=num;i++) printf("\t%d",a[i]);
 getch();
}
```

OUTPUT:



Time Complexities:-

All cases have same efficiency: Θ(*n* log *n*),
Space requirement: Θ(*n*) (NOT in-place)

Conclusion:- Thus the C program to perform merge sort using the divide and conquer technique has executed successfully.

Experiment No.6
Knapsack Problem

Aim:- Program to implement Knapsack problem

Objective: To write a C program to solve knapsack problem using Greedy method

Theory:

Let us try to apply the greedy method to solve the knapsack problem. We are given n objects and a knapsack or bag. Object i has a weight w_i and the knapsack has a capacity m. If a fraction x_i, $0 \leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m. Formally, the problem can be stated as

$$\text{maximize} \sum_{1 \leq i \leq n} p_i x_i \qquad 1)$$

$$\text{subject to} \sum_{1 \leq i \leq n} w_i x_i \leq m \qquad 2)$$

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

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying 2) and 3) above. An optimal solution is a feasible solution for which 1) is maximized.

Algorithm:

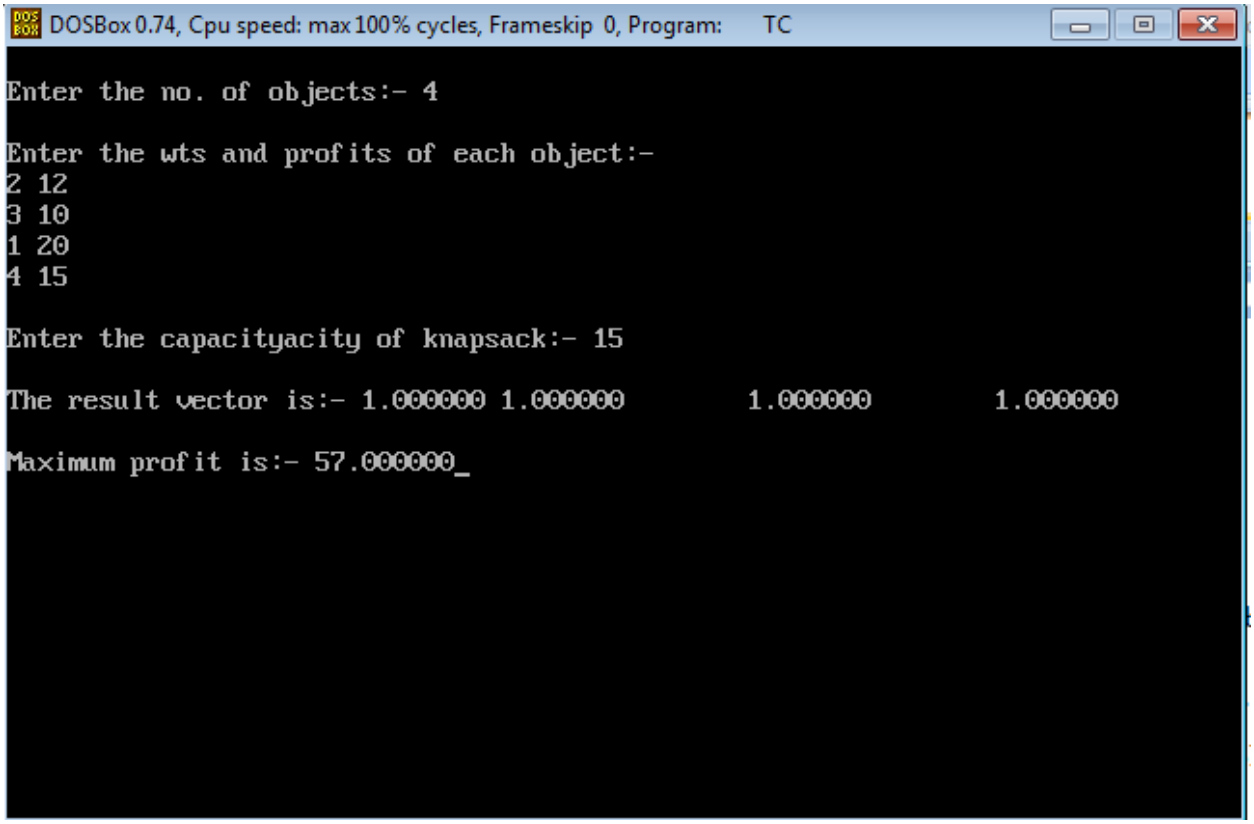
Step1: Start the program.
Step2: Declare the variable.
Step3: Using the get function read the number of items, capacity of the bag, Weight of the item and value of the items.
Step4: Find the small weight with high value using the find function.
Step5: Find the optimal solution using the function findop ().
Step6: Display the optimal solution for the items.
Step7: Stop the process

// Program to implement Knapsack problem using Greedy method

```
# include<stdio.h>

void knapsack(int n, float weight[], float profit[], float capacity) {
  float x[20], tp = 0;
  int i, j, u;
  u = capacity;

  for (i = 0; i < n; i++)
    x[i] = 0.0;

  for (i = 0; i < n; i++) {
    if (weight[i] > u)
      break;
    else {
      x[i] = 1.0;
      tp = tp + profit[i];
      u = u - weight[i];
    }
  }

  if (i < n)
    x[i] = u / weight[i];

  tp = tp + (x[i] * profit[i]);

  printf("\nThe result vector is:- ");
  for (i = 0; i < n; i++)
    printf("%f\t", x[i]);

  printf("\nMaximum profit is:- %f", tp);

}

int main() {
  float weight[20], profit[20], capacity;
  int num, i, j;
  float ratio[20], temp;
```

```
  printf("\nEnter the no. of objects:- ");
  scanf("%d", &num);

  printf("\nEnter the wts and profits of each object:- ");
  for (i = 0; i < num; i++) {
    scanf("%f %f", &weight[i], &profit[i]);
  }
  printf("\nEnter the capacityacity of knapsack:- ");
  scanf("%f", &capacity);

  for (i = 0; i < num; i++) {
    ratio[i] = profit[i] / weight[i];
  }

  for (i = 0; i < num; i++) {
    for (j = i + 1; j < num; j++) {
      if (ratio[i] < ratio[j]) {
        temp = ratio[j];
        ratio[j] = ratio[i];
        ratio[i] = temp;

        temp = weight[j];
        weight[j] = weight[i];
        weight[i] = temp;

        temp = profit[j];
        profit[j] = profit[i];
        profit[i] = temp;
      }
    }
  }

  knapsack(num, weight, profit, capacity);
  getch();
  return(0);
}
```

OUTPUT:



Conclusion:

Thus the C program for solving Knapsack problem has been executed successfully.

Experiment No.7
Prim's algorithm

Aim:- Program to implement prim's algorithm using greedy method

Objective: Write a C program to find the minimum spanning tree to implement prim's algorithm using greedy method

Algorithm:

```
1   Algorithm Prim(E, cost, n, t)
2   // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3   // adjacency matrix of an n vertex graph such that cost[i, j] is
4   // either a positive real number or ∞ if no edge (i, j) exists.
5   // A minimum spanning tree is computed and stored as a set of
6   // edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
7   // the minimum-cost spanning tree. The final cost is returned.
8   {
9       Let (k, l) be an edge of minimum cost in E;
10      mincost := cost[k, l];
11      t[1, 1] := k; t[1, 2] := l;
12      for i := 1 to n do  // Initialize near.
13          if (cost[i, l] < cost[i, k]) then near[i] := l;
14          else near[i] := k;
15      near[k] := near[l] := 0;
16      for i := 2 to n − 1 do
17      { // Find n − 2 additional edges for t.
18          Let j be an index such that near[j] ≠ 0 and
19          cost[j, near[j]] is minimum;
20          t[i, 1] := j; t[i, 2] := near[j];
21          mincost := mincost + cost[j, near[j]];
22          near[j] := 0;
23          for k := 1 to n do // Update near[ ].
24              if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
25                  then near[k] := j;
26      }
27      return mincost;
28  }
```

// C Program to implement prim's algorithm using greedy method

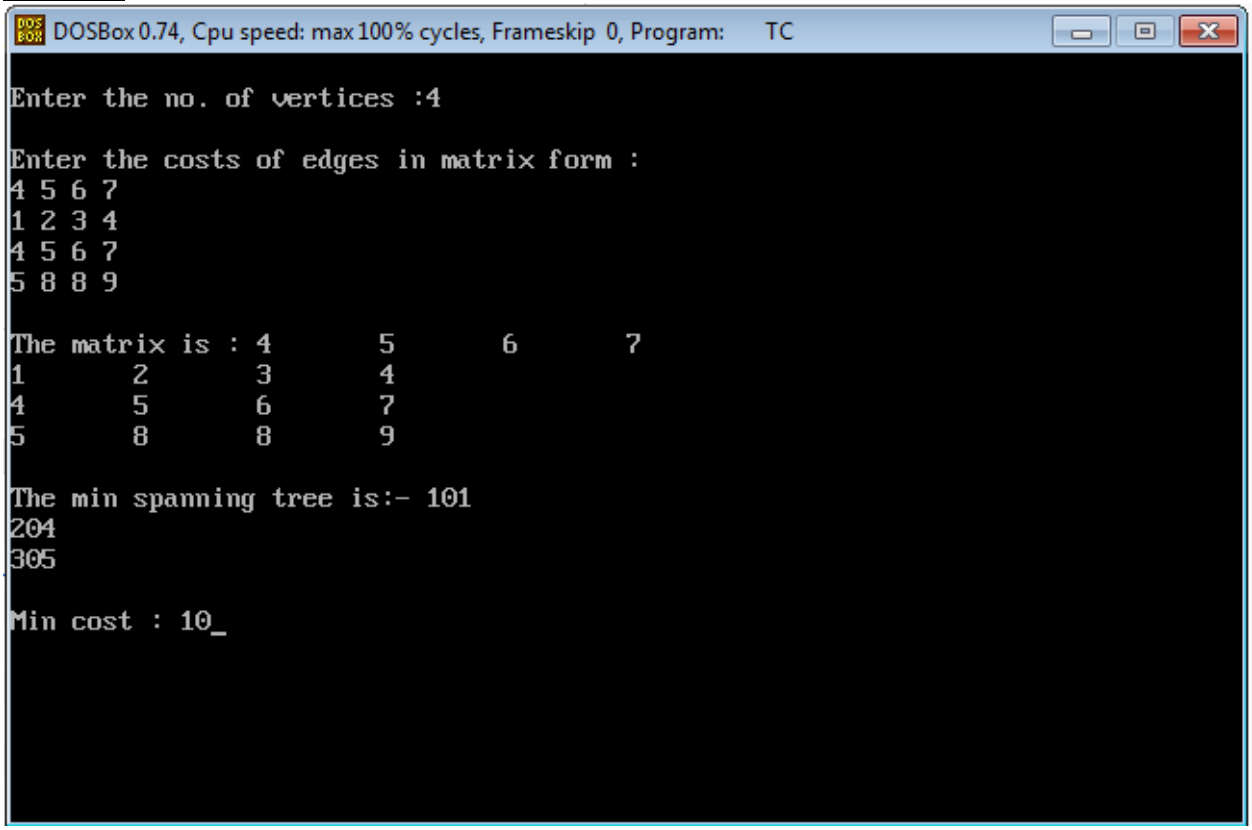
```
#include<stdio.h>
#include<conio.h>

int n, cost[10][10];
```

```
void prim() {
  int i, j, startVertex, endVertex;
  int k, nr[10], temp, minimumCost = 0, tree[10][3];

  /* For first smallest edge */
  temp = cost[0][0];
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      if (temp > cost[i][j]) {
        temp = cost[i][j];
        startVertex = i;
        endVertex = j;
      }
    }
  }
  /* Now we have fist smallest edge in graph */
  tree[0][0] = startVertex;
  tree[0][1] = endVertex;
  tree[0][2] = temp;
  minimumCost = temp;

  /* Now we have to find min dis of each vertex from either
   startVertex or endVertex by initialising nr[] array
   */

  for (i = 0; i < n; i++) {
    if (cost[i][startVertex] < cost[i][endVertex])
      nr[i] = startVertex;
    else
      nr[i] = endVertex;
  }

  /* To indicate visited vertex initialise nr[] for them to 100 */
  nr[startVertex] = 100;
  nr[endVertex] = 100;

  /* Now find out remaining n-2 edges */
  temp = 99;
  for (i = 1; i < n - 1; i++) {
    for (j = 0; j < n; j++) {
```

```
      if (nr[j] != 100 && cost[j][nr[j]] < temp) {
        temp = cost[j][nr[j]];
        k = j;
      }
    }
    /* Now i have got next vertex */
    tree[i][0] = k;
    tree[i][1] = nr[k];
    tree[i][2] = cost[k][nr[k]];
    minimumCost = minimumCost + cost[k][nr[k]];
    nr[k] = 100;

    /* Now find if k is nearest to any vertex
     than its previous near value */

    for (j = 0; j < n; j++) {
      if (nr[j] != 100 && cost[j][nr[j]] > cost[j][k])
        nr[j] = k;
    }
    temp = 99;
  }
  /* Now i have the answer, just going to print it */
  printf("\nThe min spanning tree is:- ");
  for (i = 0; i < n - 1; i++) {
    for (j = 0; j < 3; j++)
      printf("%d", tree[i][j]);
    printf("\n");
  }

  printf("\nMin cost : %d", minimumCost);
}

void main() {
  int i, j;
  clrscr();

  printf("\nEnter the no. of vertices :");
  scanf("%d", &n);

  printf("\nEnter the costs of edges in matrix form :");
```

```
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
      scanf("%d", &cost[i][j]);
    }
  printf("\nThe matrix is : ");
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      printf("%d\t", cost[i][j]);
    }
    printf("\n");
  }
  prim();
  getch();
}
```

Output:



Time Complexities:- The time required by algorithm Prim is $O(n^2)$, where n is the number of vertices in the graph G.

Conclusion:- Thus the C program to find the minimum spanning tree using prim's algorithm has executed successfully.

Experiment No.8
Kruskal's algorithm

Aim:- Program to implement Kruskal's algorithm using greedy method

Objective: Write a C program to find the minimum spanning tree to implement Kruskal's algorithm using greedy method

Algorithm:

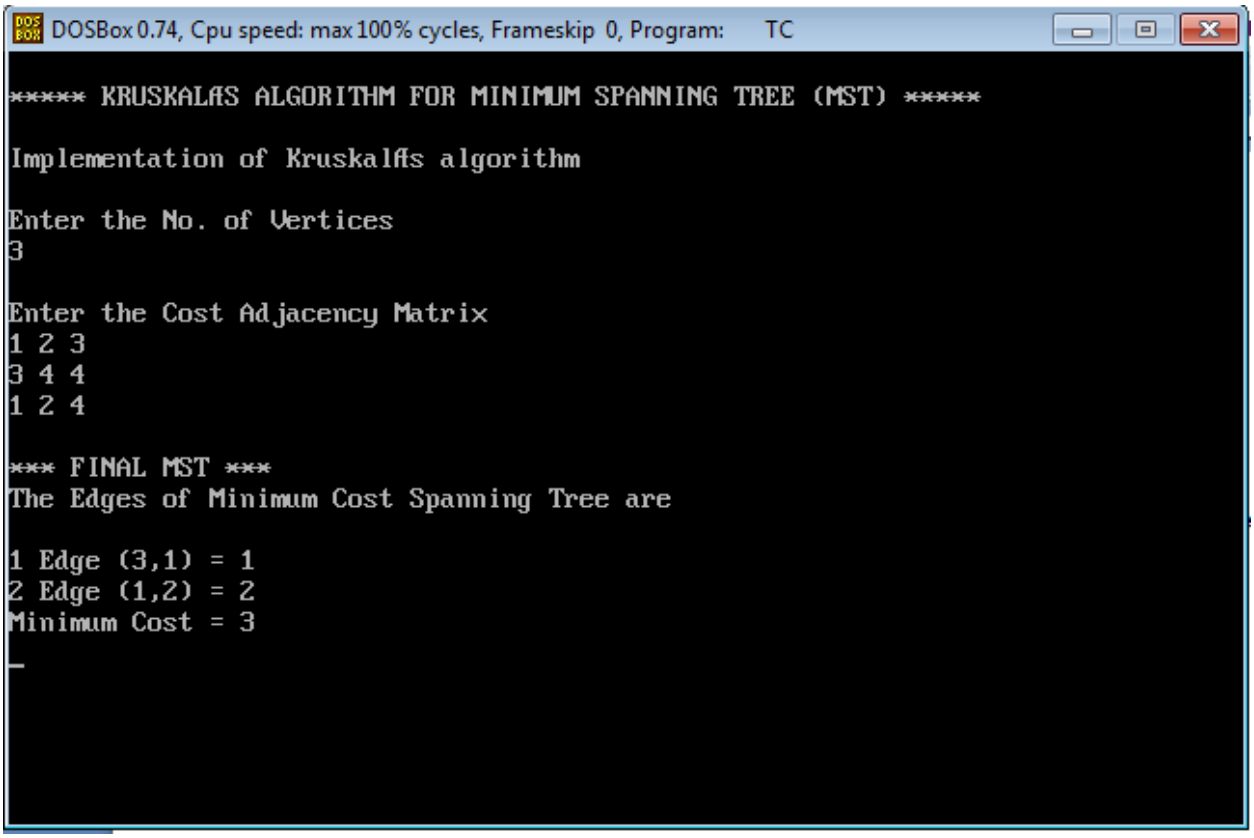
```
1   Algorithm Kruskal(E, cost, n, t)
2   // E is the set of edges in G. G has n vertices. cost[u, v] is the
3   // cost of edge (u, v). t is the set of edges in the minimum-cost
4   // spanning tree. The final cost is returned.
5   {
6       Construct a heap out of the edge costs using Heapify;
7       for i := 1 to n do parent[i] := -1;
8       // Each vertex is in a different set.
9       i := 0; mincost := 0.0;
10      while ((i < n - 1)  and (heap not empty)) do
11      {
12          Delete a minimum cost edge (u, v) from the heap
13          and reheapify using Adjust;
14          j := Find(u); k := Find(v);
15          if (j ≠ k) then
16          {
17              i := i + 1;
18              t[i, 1] := u; t[i, 2] := v;
19              mincost := mincost + cost[u, v];
20              Union(j, k);
21          }
22      }
23      if (i ≠ n - 1) then write ("No spanning tree");
24      else return mincost;
25  }
```

//Program to implement Kruskal's Algorithm using Greedy method

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
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);
void main()
{
clrscr();
printf("\n***** KRUSKAL'S ALGORITHM FOR MINIMUM SPANNING TREE (MST)
*****\n");
printf("\nImplementation of Kruskal's algorithm\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("\n*** FINAL MST ***");
printf("\nThe Edges of Minimum Cost Spanning Tree are\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",ne++,a,b,min);
mincost +=min;
}
cost[a][b]=cost[b][a]=999;
}
printf("\nMinimum Cost = %d\n",mincost);
getch();
}
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;
}
```

OUTPUT:



Time Complexities:- The computing time is O(|E| log |E|) where E is the edge set of graph G.

Conclusion:- Thus the C program to find the minimum spanning tree using Kruskal's algorithm has executed successfully.

Experiment No.9
Graph Traversal: Breadth First Search

Aim:- Program to implement graph traversal using Breadth First Search

Objective: Write a C program to implement graph traversal using Breadth First Search

Theory:

BFS Breadth First Search is an algorithm used to search the Tree or Graph. BFS search starts from root node then traversal into next level of graph or tree and continues, if item found it stops otherwise it continues. The disadvantage of BFS is it requires more memory compare to Depth First Search (DFS).

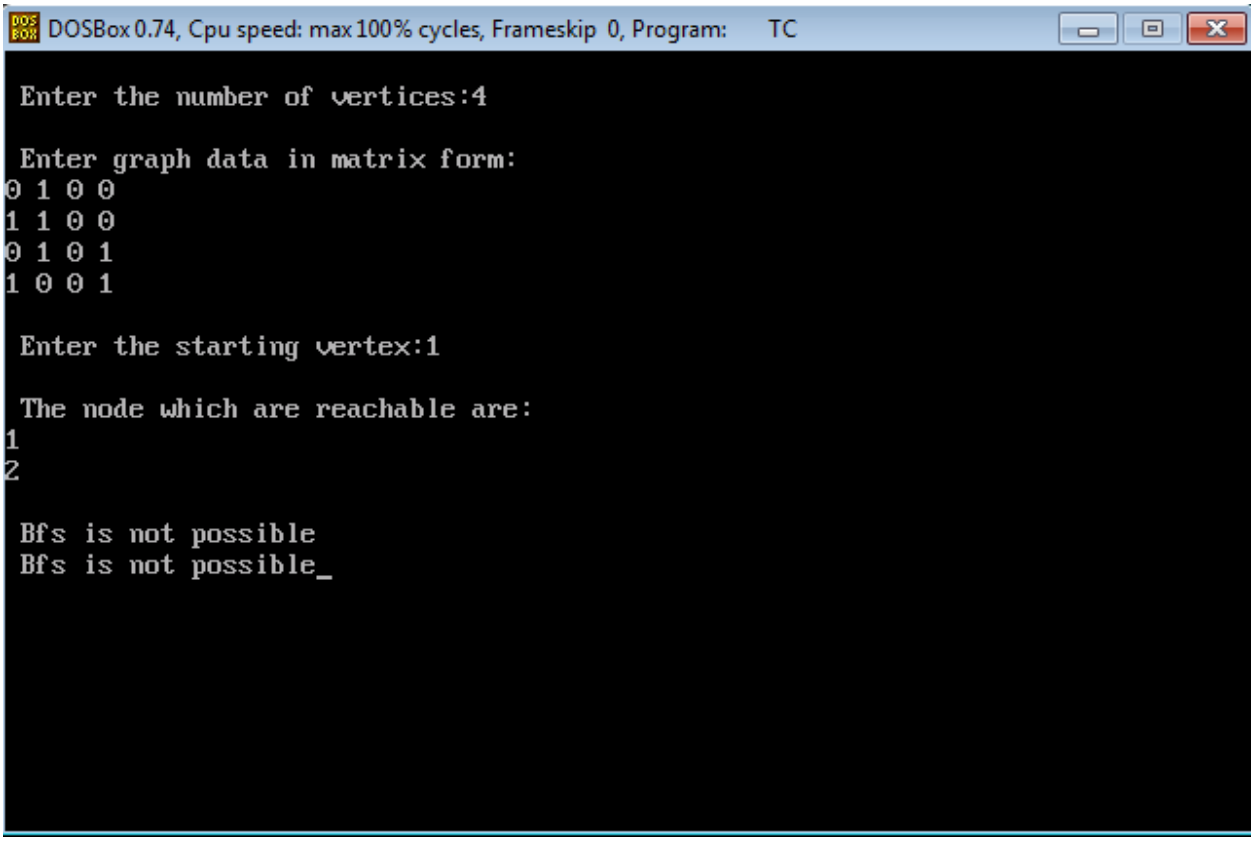
Algorithm:

```
1   Algorithm BFS(v)
2   // A breadth first search of G is carried out beginning
3   // at vertex v. For any node i, visited[i] = 1 if i has
4   // already been visited. The graph G and array visited[ ]
5   // are global; visited[ ] is initialized to zero.
6   {
7       u := v; // q is a queue of unexplored vertices.
8       visited[v] := 1;
9       repeat
10      {
11          for all vertices w adjacent from u do
12          {
13              if (visited[w] = 0) then
14              {
15                  Add w to q; // w is unexplored.
16                  visited[w] := 1;
17              }
18          }
19          if q is empty then return; // No unexplored vertex.
20          Delete u from q; // Get first unexplored vertex.
21      } until(false);
22  }
```

//Program for graph traversal using BFS//

```
#include<stdio.h>
#include<conio.h>
int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;
void bfs(int v) {
        for (i=1;i<=n;i++)
          if(a[v][i] && !visited[i])
           q[++r]=i;
        if(f<=r) {
                visited[q[f]]=1;
                bfs(q[f++]);
        }
}
void main() {
        int v;
        clrscr();
        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",i); else
           printf("\n Bfs is not possible");
        getch();
}
```

OUTPUT:



Time Complexities:- Let T(n,e) and S(n,e) be the maximum time and maximum additional space taken by algorithm BFS on any graph G with n vertices and e edges. T(n, e) = Θ(n + e) and S(n, e) = Θ(n) if G is represented by its adjacency lists. If G is represented by its adjacency matrix, then T(n, e) =Θ(n^2) and S(n ,e) = Θ(n).

Conclusion:- Thus the C program for graph traversal using breadth first search has executed successfully

Experiment No.10
Graph Traversal: Depth First Search

Aim:- Program to implement graph traversal using Depth First Search

Objective: - Write a C program to implement graph traversal using Depth First Search

Theory:- DFS Depth First Search is an algorithm used to search the Tree or Graph. DFS search starts from root node then traversal into left child node and continues, if item found it stops otherwise it continues. The advantage of DFS is it requires less memory compare to Breadth First Search (BFS).

Algorithm:-

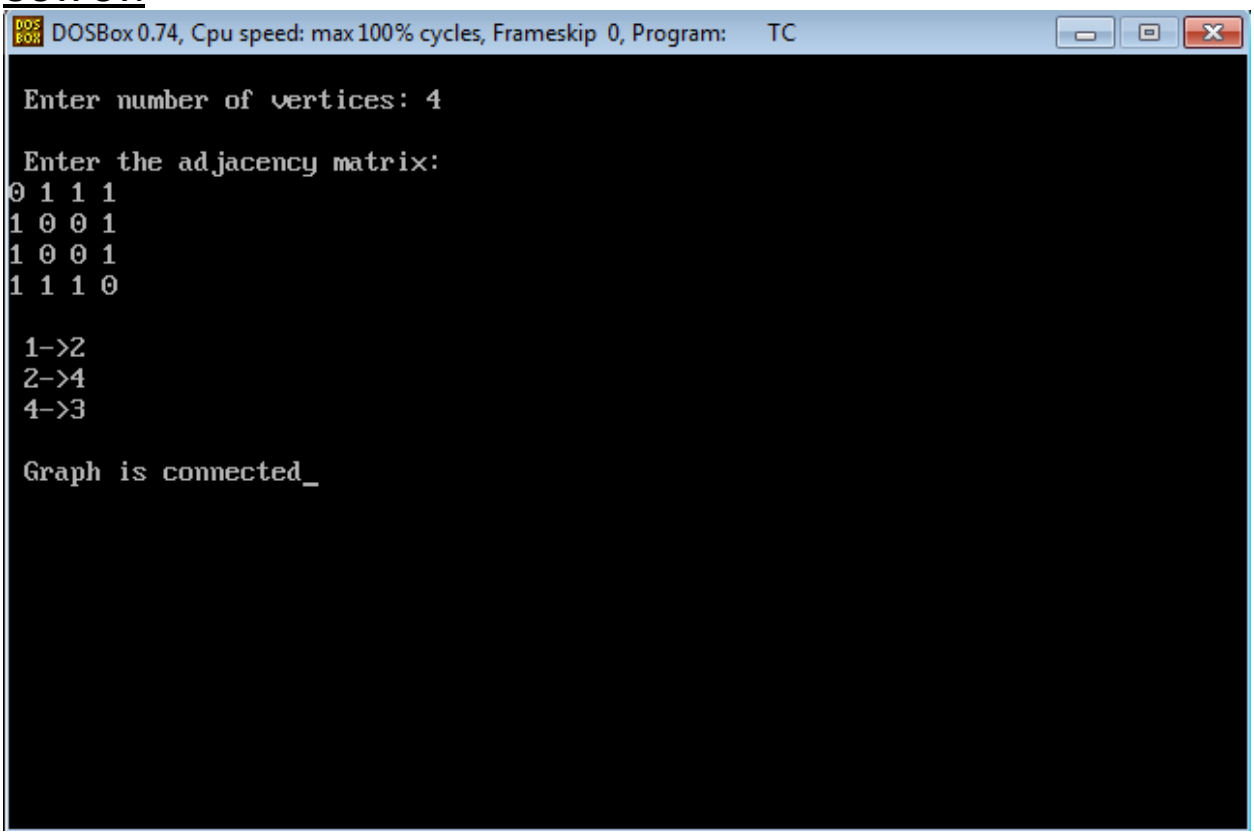
```
1   Algorithm DFS(v)
2   // Given an undirected (directed) graph G = (V, E) with
3   // n vertices and an array visited[ ] initially set
4   // to zero, this algorithm visits all vertices
5   // reachable from v. G and visited[ ] are global.
6   {
7       visited[v] := 1;
8       for each vertex w adjacent from v do
9       {
10          if (visited[w] = 0) then DFS(w);
11      }
12  }
```

// Program for graph traversal using Depth First Search//

```
#include<stdio.h>
#include<conio.h>
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);
        }
```

```
}
void main() {
        int i,j,count=0;
        clrscr();
        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");
        getch();
```

OUTPUT:



Time Complexities:- Let T(n, e) and S(n, e) be the maximum time and maximum - additional space taken by algorithm DFS for an n-vertex and e-edge graph, then S(n, e) = Θ(n) and T(n, e) = Θ(n + e) if adjacency lists are used and T(n, e) =Θ(n^2) if adjacency matrices are used.

Conclusion:- Thus the C program for graph traversal using depth first search has executed successfully

Experiment No.11
N Queen's problem

Aim:- Program to implement N queen's problem

Objective: - Write a C program to solve N queen's problem using Backtracking

Theory:- N Queen's problem is the puzzle. Placing chess queens on a chessboard, so that No two queens attack each other. Backtracking is used to solve the problem.
The *n*-queens problem consists of placing *n* queens on an *n* x *n* checker board in such a way that they do not threaten each other, according to the rules of the game of chess. Every queen on a checker square can reach the other squares that are located on the same horizontal, vertical, and diagonal line. So there can be at most one queen at each horizontal line, at most one queen at each vertical line, and at most one queen at each of the 4*n*-2 diagonal lines. Furthermore, since we want to place as many queens as possible, namely exactly *n* queens, there must be exactly one queen at each horizontal line and at each vertical line. The concept behind backtracking algorithm which is used to solve this problem is to successively place the queens in columns. When it is impossible to place a queen in a column (it is on the same diagonal, row, or column as another token), the algorithm backtracks and adjusts a preceding queen

Algorithm:-

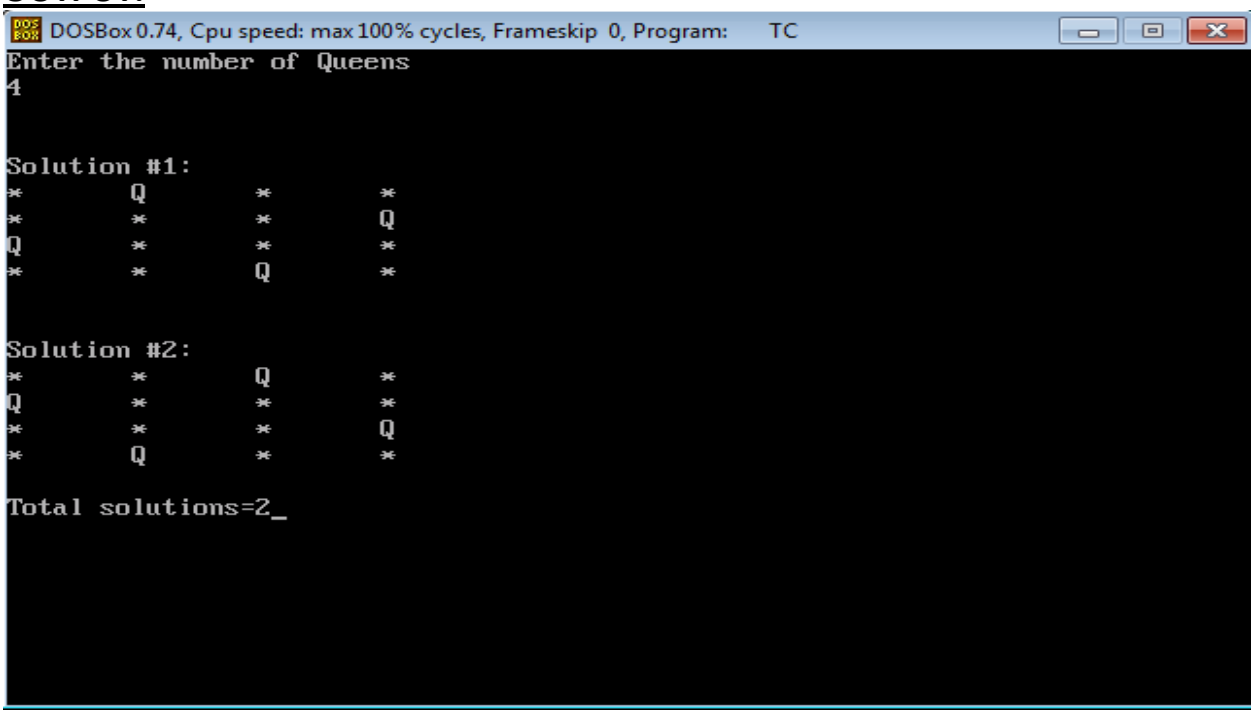
```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
15  }
```

// C Program for N queen's problem using Backtracking //

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int a[30],count=0;
int place(int pos) {
        int i;
        for (i=1;i<pos;i++) {
                if((a[i]==a[pos])||((abs(a[i]-a[pos])==abs(i-pos))))
                  return 0;
        }
        return 1;
}
void print_sol(int n) {
        int i,j;
        count++;
        printf("\n\nSolution #%d:\n",count);
        for (i=1;i<=n;i++) {
                for (j=1;j<=n;j++) {
                        if(a[i]==j)
                           printf("Q\t"); else
                           printf("*\t");
                }
                printf("\n");
        }
}
void queen(int n) {
        int k=1;
        a[k]=0;
        while(k!=0) {
                a[k]=a[k]+1;
                while((a[k]<=n)&&!place(k))
                  a[k]++;
                if(a[k]<=n) {
                        if(k==n)
                           print_sol(n); else {
                                k++;
                                a[k]=0;
                        }
```

```
                } else
                  k--;
        }
}
void main() {
        int i,n;
        clrscr();
        printf("Enter the number of Queens\n");
        scanf("%d",&n);
        queen(n);
        printf("\nTotal solutions=%d",count);
        getch();
}
```

OUTPUT:



Complexity:-

The power of the set of all possible solutions of the n queen's problem is n! and the bounding function takes a linear amount of time to calculate, therefore the running time of the n queens problem is O(n!).

Conclusion:- Thus the C program to solve N queen's problem using Backtracking has executed successfully

Experiment No.12
All Pairs shortest Path

Aim:- Program to implement all pairs shortest path

Objective: - Write a C program to find all pairs shortest path using Floyd's algorithm

Theory:- Floyd's algorithm is applicable to both directed and undirected graphs provided that they do not contain a cycle. It is convenient to record the lengths of shortest path in an n- by- n matrix D called the distance matrix. The element dij in the ith row and jth column of matrix indicates the shortest path from the ith vertex to jth vertex (1<=i, j<=n). The element in the ith row and jth column of the current matrix D(k-1) is replaced by the sum of elements in the same row i and kth column and in the same column j and the kth column if and only if the latter sum is smaller than its current value.

Algorithm:-

Algorithm Floyd(W[1..n,1..n])
//Implements Floyd's algorithm for the all-pairs shortest paths problem
//Input: The weight matrix W of a graph
//Output: The distance matrix of shortest paths length
{
D ← W
for k←1 to n do
{
for i ← 1 to n do
{
for j ← 1 to n do
{
D[i,j] ← min (D[i, j], D[i, k]+D[k, j])
}
}
}
return D
}

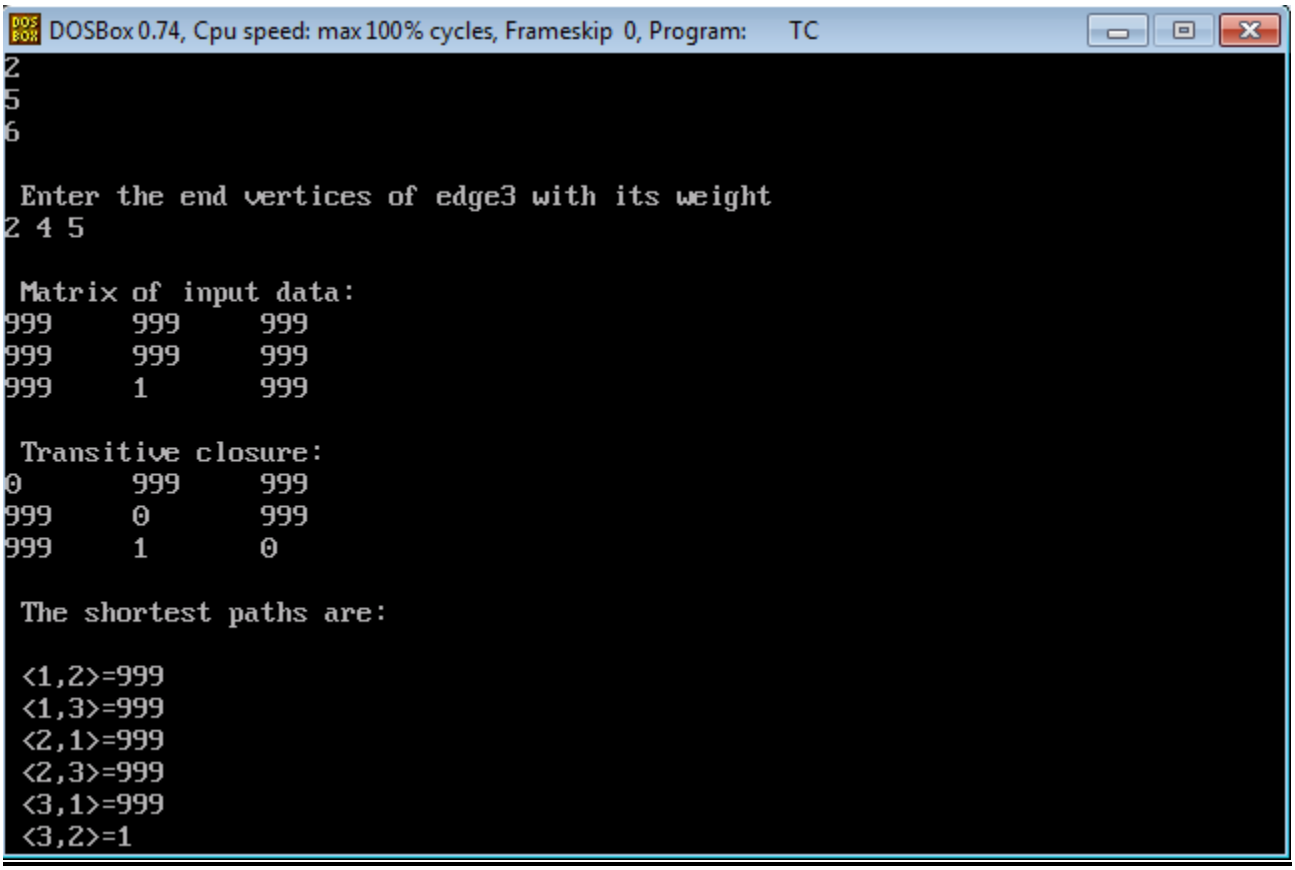
// All pairs Shortest path //

Program:

```
#include<stdio.h>
#include<conio.h>

void floyd(int[10][10],int);
int min(int,int);
void main()
{
int n,a[10][10],i,j;
printf("Enter the no.of nodes : ");
scanf("%d",&n);
printf("\nEnter the cost adjacency matrix\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
scanf("%d",&a[i][j]);
floyd(a,n);
getch();
}
void floyd(int a[10][10],int n)
{
int d[10][10],i,j,k;
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
d[i][j]=a[i][j];
}
for(k=1;k<=n;k++)
{
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
d[i][j]=min(d[i][j],d[i][k]+d[k][j]);
}
}
}
printf("\nThe distance matrix is\n");
for(i=1;i<=n;i++)
```

```
{
for(j=1;j<=n;j++)

{
printf("%d\t",d[i][j]);
}
printf("\n");
}
}
int min (int a,int b)
{
if(a<b)
return a;
else
return b;
}
```

Output:



Complexity:- The time efficiency of Floyd's algorithm is cubic i.e. Θ (n3)

Conclusion:- Thus the C program to find shortest path using Floyd's algorithm has executed successfully

3.Quiz on the subject:-

1.What is an algorithm?

An algorithm is a sequence of unambiguous instructions for solving a problem. For obtaining a required output for any legitimate input in a finite amount of time.

2.How is an algorithm's time efficiency measured?

Time efficiency indicates how fast the algorithm runs. An algorithm's time efficiency is measured as a function of its input size by counting the number of times its basic operation (running time) is executed. Basic operation is the most time consuming operation in the algorithm's innermost loop.

3. What is Big 'Oh' notation?

A function t(n) is said to be in O(g(n)), denoted t(n) O(g(n)) , if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integers n0 such that $t(n) \leq cg(n)$ for all $n \geq n0$

4.What is recursive call?

Recursive algorithm makes more than a single call to itself is known as recursive call. An algorithm that calls itself is direct recursive.An algorithm"A" is said to be indirect recursive if it calls another algorithm which in turn calls "A".

5. How is the efficiency of the algorithm defined?

The efficiency of an algorithm is defined with the components.
(i) Time efficiency -indicates how fast the algorithm runs
(ii) Space efficiency -indicates how much extra memory the algorithm needs

6.What are the characteristics of an algorithm?

Every algorithm should have the following five characteristics
(i) Input
(ii) Output
(iii) Definiteness
(iv) Effectiveness
(v) Termination
Therefore, an algorithm can be defined as a sequence of definite and effective instructions, which terminates with the production of correct output from the given input. In other words, viewed little more formally, an algorithm is a step by step formalization of a mapping function to map input set onto an output set.

7.What do you mean by time complexity and space complexity of an algorithm?

Time complexity indicates how fast the algorithm runs. Space complexity deals with extra memory it require. Time efficiency is analyzed by determining the number of repetitions of the basic operation as a function of input size. Basic operation: the operation that contributes most towards the running time of the algorithm The running time of an algorithm is the function defined by the number of steps (or amount of memory) required to solve input instances of size n.

8.Define Big Theta Notations

A function t(n) is said to be in Θ(g(n)) , denoted t(n) Є Θ (g(n)) , if t(n) is bounded both above and below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constants c1 and c2 and some nonnegative integer n0 such that c1 g(n) ≤t(n) ≤ c2g(n) for all n ≥ n0

9. What is performance measurement?

Performance measurement is concerned with obtaining the space and the time requirements of a particular algorithm.

10.What is space complexity and time complexity ?

The space complexity of an algorithm is the amount of memory it needs to run to completion. The time complexity of an algorithm is the amount of computer time it needs to run to completion

11. Give the two major phases of performance evaluation

Performance evaluation can be loosely divided into two major phases:
(i) a prior estimates (performance analysis)
(ii) a Posterior testing(performance measurement)

12. Define the divide an conquer method.

Given a function to compute on ‘n’ inputs the divide-and-comquer strategy suggests splitting the inputs in to’k’ distinct susbsets, 1<k <n, yielding ‘k’ subproblems. The subproblems must be solved, and then a method must be found to combine subsolutions into a solution of the whole. If the subproblems are still relatively large, then the divide-and conquer strategy can possibly be reapplied.

13. What is the binary search?

If ‘q’ is always chosen such that ‘aq’ is the middle element(that is, q=[(n+1)/2), then the resulting search algorithm is known as binary search.

14.Give computing time for Binary search?

In conclusion we are now able completely describe the computing time of binary search by giving formulas that describe the best, average and worst cases.

Successful searches: Θ(1) Θ (logn) Θ (Logn)

best average worst

Unsuccessful searches: Θ (logn)

best, average, worst

15. Define external path length?

The external path length E, is defines analogously as sum of the distance of all external nodes from the root.

16. Define internal path length.

The internal path length 'I' is the sum of the distances of all internal nodes from the root.

17. What is the maximum and minimum problem?

The problem is to find the maximum and minimum items in a set of 'n' elements. Though this problem may look so simple as to be contrived, it allows us to demonstrate divideand-conquer in simple setting.

18. What is the Quick sort?

Quicksort is divide and conquer strategy that works by partitioning it's input elements according to their value relative to some preselected element(pivot). it uses recursion and the method is also called partition –exchange sort.

19. Write the Analysis for the Quick sort.

O(nlogn) in average and best cases

$O(n^2)$ in worst case

20. What is Merge sort? and Is insertion sort better than the merge sort?

Merge sort is divide and conquer strategy that works by dividing an input array in to two halves,sorting them recursively and then merging the two sorted halves to get the original array sorted Insertion sort works exceedingly fast on arrays of less then 16 elements, though for large 'n' its computing time is O(n2).

21. Explain the greedy method.

Greedy method is the most important design technique, which makes a choice that looks best at that moment. A given 'n' inputs are required us to obtain a subset that

satisfies some constraints that is the feasible solution. A greedy method suggests that one can device an algorithm that works in stages considering one input at a time.

22. Define feasible and optimal solution.

Given n inputs and we are required to form a subset such that it satisfies some given constraints then such a subset is called feasible solution. A feasible solution either maximizes or minimizes the given objective function is called as optimal solution

23.What are the constraints of knapsack problem?

To maximize $\mathbf{\Sigma p_i x_i \ \ 1 \le i \le n}$

The constraint is **:** $\mathbf{\Sigma w_i x_i} \geq m$ and $0 \leq \mathbf{x_i} \leq 1$ $1 \leq i \leq n$ $\mathbf{1 \le i \le n}$

where m is the bag capacity, n is the number of objects and for each object i , w_i and p_i are the weight and profit of object respectively.

24.What is a minimum cost spanning tree?

A spanning tree of a connected graph is its connected acyclic subgraph that contains all vertices of a graph. A minimum spanning tree of a weighted connected graph is its spanning tree of the smallest weight where bweight of the tree is the sum of weights on all its edges.

A minimum spanning subtree of a weighted graph (G,w) is a spanning subtree of G of minimum weight w(T)= Σ w(e) e€ T

Minimum Spanning Subtree Problem: Given a weighted connected undirected graph (G,w), find a minimum spanning subtree

25. Specify the algorithms used for constructing Minimum cost spanning tree.

a) Prim's Algorithm

b) Kruskal's Algorithm

26. What is Knapsack problem?

A bag or sack is given capacity and n objects are given. Each object has weight wi and profit pi .Fraction of object is considered as x_i (i.e) $0<=x_i<=1$ .If fraction is 1 then entire object is put into sack. When we place this fraction into the sack we get $\mathbf{w_i x_i}$ and $\mathbf{p_i x_i}$

27. Write any two characteristics of Greedy Algorithm?

To solve a problem in an optimal way construct the solution from given set of candidates. As the algorithm proceeds, two other sets get accumulated among this one set contains the candidates that have been already considered and chosen while the other set contains the candidates that have been considered but rejected.

28. Write the difference between the Greedy method and Dynamic programming.

Greedy method: 1. Only one sequence of decision is Generated 2. It does not guarantee to give an optimal solution always.

Dynamic programming: 1. Many number of decisions are generated. 2. It definitely gives an optimal solution always.

29. State the requirement in optimal storage problem in tapes.

Finding a permutation for the n programs so that when they are stored on the tape in this order the MRT is minimized. This problem fits the ordering paradigm.

30. State Kruskal's Algorithm.

The algorithm looks at a MST for a weighted connected graph as an acyclic subgraph with |v|-1 edges for which the sum of edge weights is the smallest.

4. Conduction of Viva-Voce Examinations:

Teacher should conduct oral exams of the students with full preparation. Normally, the objective questions with guess are to be avoided. To make it meaningful, the questions should be such that depth of the students in the subject is tested. Oral examinations are to be conducted in cordial environment amongst the teachers taking the examination. Teachers taking such examinations should not have ill thoughts about each other and courtesies should be offered to each other in case of difference of opinion, which should be critically suppressed in front of the students.

5. Evaluation and marking system:

Basic honesty in the evaluation and marking system is absolutely essential and in the process impartial nature of the evaluator is required in the examination system to become. It is a primary responsibility of the teacher to see that right students who are really putting up lot of hard work with right kind of intelligence are correctly awarded.

The marking patterns should be justifiable to the students without any ambiguity and teacher should see that students are faced with just circumstances.

External examiner should evaluate student by checking practical performance and conducting viva.