

# **Lab Manual**

## **Advanced Data Structures and Algorithms Laboratory**

**SYCSE**

**A.Y. 2024-25**



**KITs College of Engineering (Autonomous), Kolhapur**  
**Department of Computer Science and Engineering**

**KIT's College of Engineering (Autonomous), Kolhapur**  
**Department of Computer Science & Engineering**

**Course Name: - Advanced Data Structures and Algorithms Laboratory**  
**Course Code: - UCSPC0331**

**Experiment List**

<b>Experiment No.</b>	<b>Experiment Name</b>
<b>1</b>	<b>Write a program to implement the following operations on Binary Search Tree: a) Insert b) Delete c) Search d) Display</b>
<b>2</b>	<b>Write a program to create the graph and implement traversal methods.</b>
<b>3</b>	<b>Write a program to implement Merge sort for the given list of integer values.</b>
<b>4</b>	<b>Write a program to implement Quicksort for the given list of integer values.</b>
<b>5</b>	<b>Write a program to find the solution for the knapsack problem using the greedy method.</b>
<b>6</b>	<b>Write a program to find minimum cost spanning tree using Prim's algorithm.</b>
<b>7</b>	<b>Write a program to find minimum cost spanning tree using Kruskal's algorithm.</b>
<b>8</b>	<b>Write a program to find a single source shortest path for a given graph.</b>
<b>9</b>	<b>Write a program to find the solution for a 0-1 knapsack problem using dynamic programming.</b>

## **Experiment**

### **No.1**

## **Binary Search Tree**

**Aim:-** Program to implement the operations on Binary Search Tree.

**Objective:** To write a C program to implement the operations on Binary Search Tree.

### **Algorithm:**

1. Start
2. Call insert to insert an element into binary search tree.
3. Call delete to delete an element from binary search tree.
4. Call findmax to find the element with maximum value in binary search tree
5. Call findmin to find the element with minimum value in binary search tree
6. Call find to check the presence of the element in the binary search tree
7. Call display to display the elements of the binary search tree
8. Call makeempty to delete the entire tree.
9. Stop

### **Insert function:**

1. Get the element to be inserted.
2. Find the position in the tree where the element to be inserted by checking the elements in the tree by traversing from the root.
3. If the element to be inserted is less than the element in the current node in the tree then traverse left subtree
4. If the element to be inserted is greater than the element in the current node in the tree then traverse right subtree
5. Insert the element if there is no further move

### **Delete function:**

1. Get the element to be deleted.
2. Find the node in the tree that contain the element.
3. Delete the node an rearrange the left and right siblings if any present for the deleted node

### **Findmax function:**

1. Traverse the tree from the root.
2. Find the rightmost leaf node of the entire tree and return it
3. If the tree is empty return null.

**Findmin function:**

1. Traverse the tree from the root.
2. Find the leftmost leaf node of the entire tree and return it
3. If the tree is empty return null.

**Find function:**

1. Traverse the tree from the root.
2. Check whether the element to searched matches with element of the current node. If match occurs return it.
3. Otherwise if the element is less than that of the element of the current node then search the leaf subtree
4. Else search right subtree.

**Makeempty function:**

Make the root of the tree to point to null.

Program:

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

```
struct searchtree
{
int element;
struct searchtree *left,*right;
}*root;
```

```
typedef struct searchtree *node;
typedef int ElementType;
node insert(ElementType, node);
node delete(ElementType, node);
void makeempty();
node findmin(node);
node findmax(node);
node find(ElementType, node);
void display(node, int);
```

```
void main()
{
```

```
int ch;
ElementType a;
node temp;
makeempty();
while(1)
{
printf("\n1. Insert\n2. Delete\n3. Find\n4. Find min\n5. Find max\n6.Display\n7.
Exit\nEnter Your Choice : ");
scanf("%d",&ch);
```

```
switch(ch)
{
case 1:
printf("Enter an element : ");
scanf("%d", &a);
root = insert(a, root);
break;
```

```
case 2:
printf("\nEnter the element to delete : ");
scanf("%d",&a);
root = delete(a, root);
break;
```

```
case 3:
printf("\nEnter the element to search : ");
scanf("%d",&a);
temp = find(a, root);
if (temp != NULL)
printf("Element found");
else
printf("Element not found");
break;
```

```
case 4:
temp = findmin(root);
if(temp==NULL)
printf("\nEmpty tree");
else
printf("\nMinimum element : %d", temp->element);
break;
```

```
case 5:
```

```
temp = findmax(root);
if(temp==NULL)
printf("\nEmpty tree");
else
printf("\nMaximum element : %d", temp->element);
break;
```

```
case 6:
if(root==NULL)
printf("\nEmpty tree");
else
display(root, 1);
break;
```

```
case 7:
exit(0);
default:
printf("Invalid Choice");
}
}
}
```

```
node insert(ElementType x,node t)
{
if(t==NULL)
{
t = (node)malloc(sizeof(node));
t->element = x;
t->left = t->right = NULL;
}
else
{
if(x < t->element)
t->left = insert(x, t->left);

else
if(x > t->element)t->right = insert(x, t->right);
}
return t;
}
```

```

node delete(ElementType x,node t)
{
node temp;
if(t == NULL)
printf("\nElement not found");
else {
if(x < t->element)
t->left = delete(x, t->left);
else if(x > t->element)
t->right = delete(x, t->right);
else {
if(t->left && t->right)
{
temp = findmin(t->right);
t->element = temp->element;
t->right = delete(t->element,t->right);
}
else if(t->left == NULL)
{
temp = t;
t=t->right;
free (temp); }
else {
temp = t;
t=t->left;
free (temp);
}
}}
return t;
}

void makeempty() {
root = NULL;
}

node findmin(node temp) {
if(temp == NULL || temp->left == NULL)
return temp;
return findmin(temp->left);
}

node findmax(node temp) {
if(temp==NULL || temp->right==NULL)
return temp;

```

```
return findmax(temp ->right);  
}
```

```
node find(ElementType x, node t) {  
    if(t==NULL)  
        return NULL;  
    if(x<t->element)  
        return find(x,t->left);  
    if(x>t->element)  
        return find(x,t->right);  
    return t; }
```

```
void display(node t,int level) {  
    int i;  
    if(t) {  
        display(t->right, level+1);  
        printf("\n");  
        for(i=0;i<level;i++)  
            printf(" ");  
        printf("%d", t->element);  
        display(t->left, level+1);  
    }  
}
```

Sample Output:

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit

Enter your Choice : 1

Enter an element : 10

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit



Enter your Choice : 1

Enter an element : 20

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit

Enter your Choice : 1

Enter an element : 5

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit

Enter your Choice : 4

The smallest Number is 5

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit

Enter your Choice : 3

Enter an element : 100

Element not Found

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit

Enter your Choice : 2

Enter an element : 20

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit

Enter your Choice : 6

20

10

1. Insert
2. Delete
3. Find
4. Find Min
5. Find Max
6. Display
7. Exit

Enter your Choice : 7

## Graph Traversal Methods

**Aim:-** Program to create the graph and implement traversal methods.

**Objective:** To write a C program to create the graph and implement traversal methods.

### Breadth First Search:

BFS explores graph moving across to all the neighbors of last visited vertex traversals i.e., it proceeds in a concentric manner by visiting all the vertices

```
Algorithm : BFS(G)
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = (V, E)
//Output: Graph G with its vertices marked with consecutive integers in
the order they
//have been visited by the BFS traversal
{
    mark each vertex with 0 as a mark of
    being "unvisited" count  $\leftarrow$  0
    for each vertex v in V do
    {
        if v is marked
            with 0
            bfs(v)
    }
}
Algorithm : bfs(v)
//visits all the unvisited vertices connected to vertex v and assigns them
the numbers
//in order they are visited via global variable count
{
    count  $\leftarrow$  count + 1
    mark v with count and initialize queue with v
    while queue is not empty do
    {
        a := front of queue
        for each vertex w adjacent to a do
        {
            if w is marked with 0
            {
                count  $\leftarrow$ 
                count + 1
                mark
                w with count
                add w to the end of the queue
            }
        }
        remove a from the front of the queue
    }
}
```

that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it and so on, until all the vertices in the same connected component as the starting vertex are visited. Instead of a stack, BFS uses queue.

Complexity:

BFS has the same efficiency as DFS: it is  $\Theta(V^2)$  for Adjacency matrix representation and  $\Theta(V+E)$  for Adjacency linked list representation.

### Program:

```
/*Print all the nodes reachable from a given starting node in a digraph using BFS
```

```
method.*/#include<stdio.h>
```

```
void BFS(int [20][20],int,int [20],int);
```

```
void main()
```

```
{
```

```
    int n,a[20][20],i,j,visited[20],source;
    clrscr();
```

```
    printf("Enter the number of
    vertices:");scanf("%d",&n);
```

```
    printf("\nEnter the adjacency
    matrix:\n");for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&a[i][j]);
```

```
    for(i=1;i<=n;i++)
        visited[i]=0;
```

```
    printf("\nEnter the source
    node:");scanf("%d",&source);
    visited[source]=1;
```

```
    BFS(a,source,visited,n);
```

```
    for(i=1;i<=n;i++)
```

```
    {
```

```
        if(visited[i]!=0)
            printf("\n Node %d is reachable",i);
```

```
        else
            printf("\n Node %d is not reachable",i);
```

```
    }
```

```
    getch();
```

```
}
```

```
void BFS(int a[20][20],int source,int visited[20],int n)
```

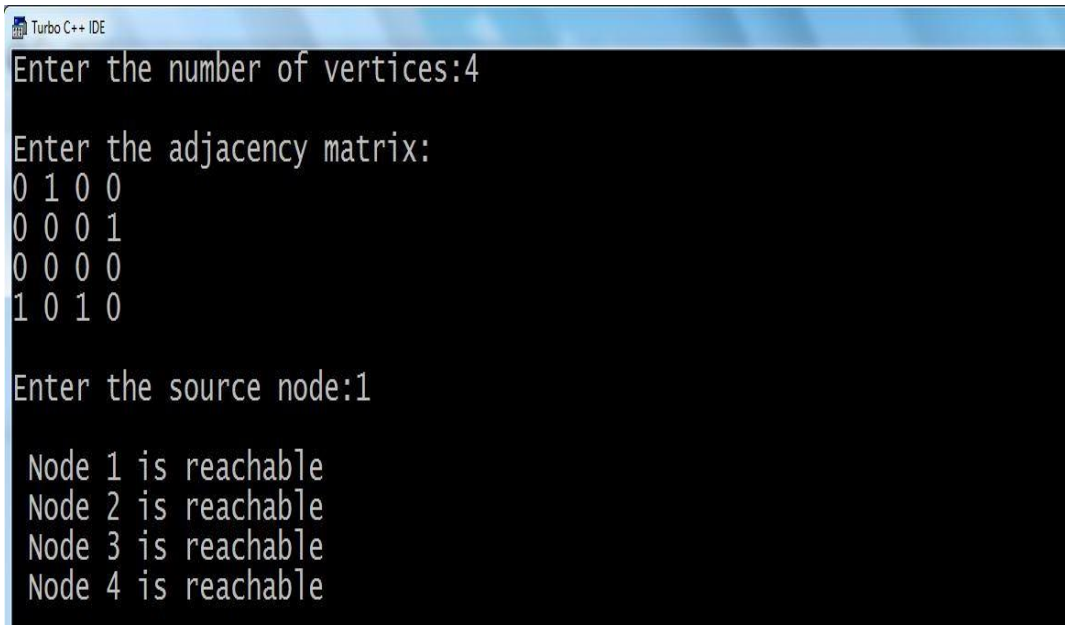
```
{
```

```
    int queue[20],f,r,u,v;
```

```
    f=0;
```

```
r=-1;
queue[++r]=source;
while(f<=r)
{
    u=queue[f++];
    for(v=1;v<=n;v++)
    {
        if(a[u][v]==1 && visited[v]==0)
        {    queue[++r]=v;
            visited[v]=1;
        }
    } //for v
} // while
}
```

## OUTPUT:



```
Turbo C++ IDE
Enter the number of vertices:4
Enter the adjacency matrix:
0 1 0 0
0 0 0 1
0 0 0 0
1 0 1 0
Enter the source node:1
Node 1 is reachable
Node 2 is reachable
Node 3 is reachable
Node 4 is reachable
```

### Depth First Search:

Depth-first search starts visiting vertices of a graph at an arbitrary vertex by marking it as having been visited. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. This process continues until a vertex with no adjacent unvisited vertices is encountered. At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there. The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end.

```

Algorithm : DFS(G)
//Implements a depth-first search traversal of a given graph
//Input : Graph G = (V,E)
//Output : Graph G with its vertices marked with consecutive
integers in the order they
//have been first encountered by the DFS traversal
{
    mark each vertex in V with 0 as a mark of being
    "unvisited".count  $\leftarrow$  0
    for each vertex v in V do
        if v is marked
            with 0
            dfs(v)
}

Algorithm : dfs(v)
//visits recursively all the unvisited vertices connected to vertex v by
a path
//and numbers them in the order they are encountered via global
variable count
{
    count  $\leftarrow$  count+1
    mark v with count
    for each vertex w in V
        adjacent to v do if w is
        marked with 0
        dfs(w)
}

```

**Complexity:** For the adjacency matrix representation, the traversal time efficiency is in  $\Theta(|V|^2)$  and for the adjacency linked list representation, it is in  $\Theta(|V|+|E|)$ , where  $|V|$  and  $|E|$  are the number of graph's vertices and edges respectively.

## Program:

```
/* Check whether a given graph is connected or not using DFS
```

```
method.*/#include<stdio.h>
```

```
void DFS(int [20][20],int,int [20],int);
```

```
void main()
```

```
{
    int n,a[20][20],i,j,visited[20],source;
    clrscr();

    printf("Enter the number of vertices:
    ");scanf("%d",&n);

    printf("\nEnter the adjacency
    matrix:\n");for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&a[i][j]);

    for(i=1;i<=n;i++)
        visited[i]=0;

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

    DFS(a,source,visited,n);

    for(i=1;i<=n;i++)
    {
        if(visited[i]==0)
        {
            printf("\nGraph is not
            connected");getch();
            exit(0);
        }
    }
    printf("\nGraph is
    connectd\n");getch();
}
```

```
void DFS(int a[20][20],int u,int visited[20],int n)
```

```
{
    int v;

    visited[u]=1;
    for(v=1;v<=n;v++)
    {
        if(a[u][v]==1 && visited[v]==0)
            DFS(a,v,visited,n);
    }
}
```



## OUTPUT:

```
Turbo C++ IDE
Enter the number of vertices: 4
Enter the adjacency matrix:
0 1 1 1
1 0 0 1
1 0 0 1
1 1 1 0
Enter the source node: 1
Graph is connectd
_
```

```
Turbo C++ IDE
Enter the number of vertices: 5
Enter the adjacency matrix:
0 1 1 0 0
1 0 0 1 0
1 0 0 1 0
0 1 1 0 0
0 0 0 0 0
Enter the source node: 1
Graph is not connected
```

### **Experiment No.3**

#### **Merge Sort**

**Aim:-** Program to implement Merge sort for the given list of integer values.

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

**Theory:** An 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);
```

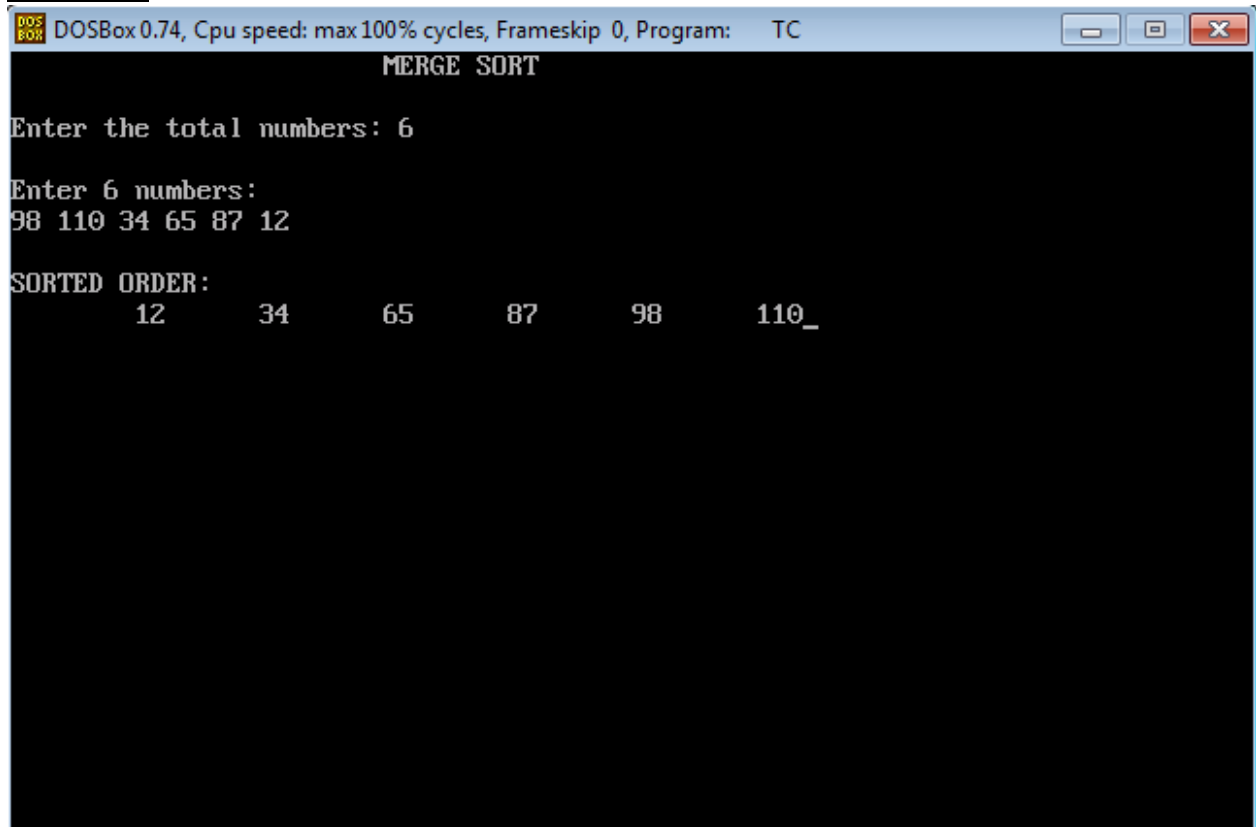
```

    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:**



The screenshot shows a DOSBox window titled "DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC". The window contains a black terminal area with white text. The text reads: "MERGE SORT", "Enter the total numbers: 6", "Enter 6 numbers:", "98 110 34 65 87 12", "SORTED ORDER:", and a list of numbers "12 34 65 87 98 110\_" on the next line.

```

MERGE SORT

Enter the total numbers: 6

Enter 6 numbers:
98 110 34 65 87 12

SORTED ORDER:
12 34 65 87 98 110_

```

### **Time Complexities:-**

All cases have same efficiency:  $\Theta(n \log n)$ ,

Space requirement:  $\Theta(n)$  (NOT in-place)

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

## **Experiment No.4**

### **Quick Sort**

**Aim:-** Program to implement Quicksort for the given list of integer values.

**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];

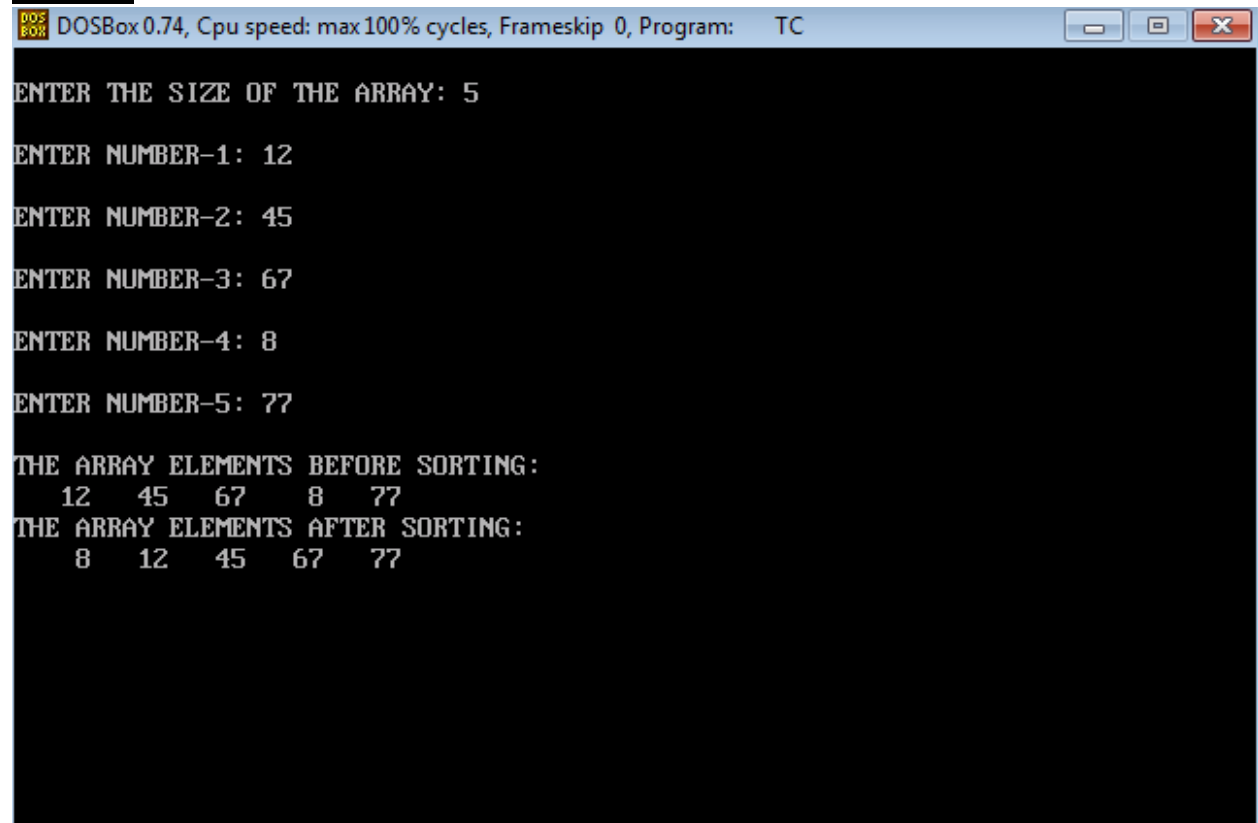
```

```

        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:**



```

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC
ENTER THE SIZE OF THE ARRAY: 5
ENTER NUMBER-1: 12
ENTER NUMBER-2: 45
ENTER NUMBER-3: 67
ENTER NUMBER-4: 8
ENTER NUMBER-5: 77
THE ARRAY ELEMENTS BEFORE SORTING:
12 45 67 8 77
THE ARRAY ELEMENTS AFTER SORTING:
8 12 45 67 77

```

**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.5**

### **Knapsack Problem**

**Aim:-** Program to find the solution for the knapsack problem using the greedy method.

**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

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

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

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

The profits and weights are positive numbers.

A feasible solution (or filling) is any set  $(x_1, \dots, x_n)$  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 capacity 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;

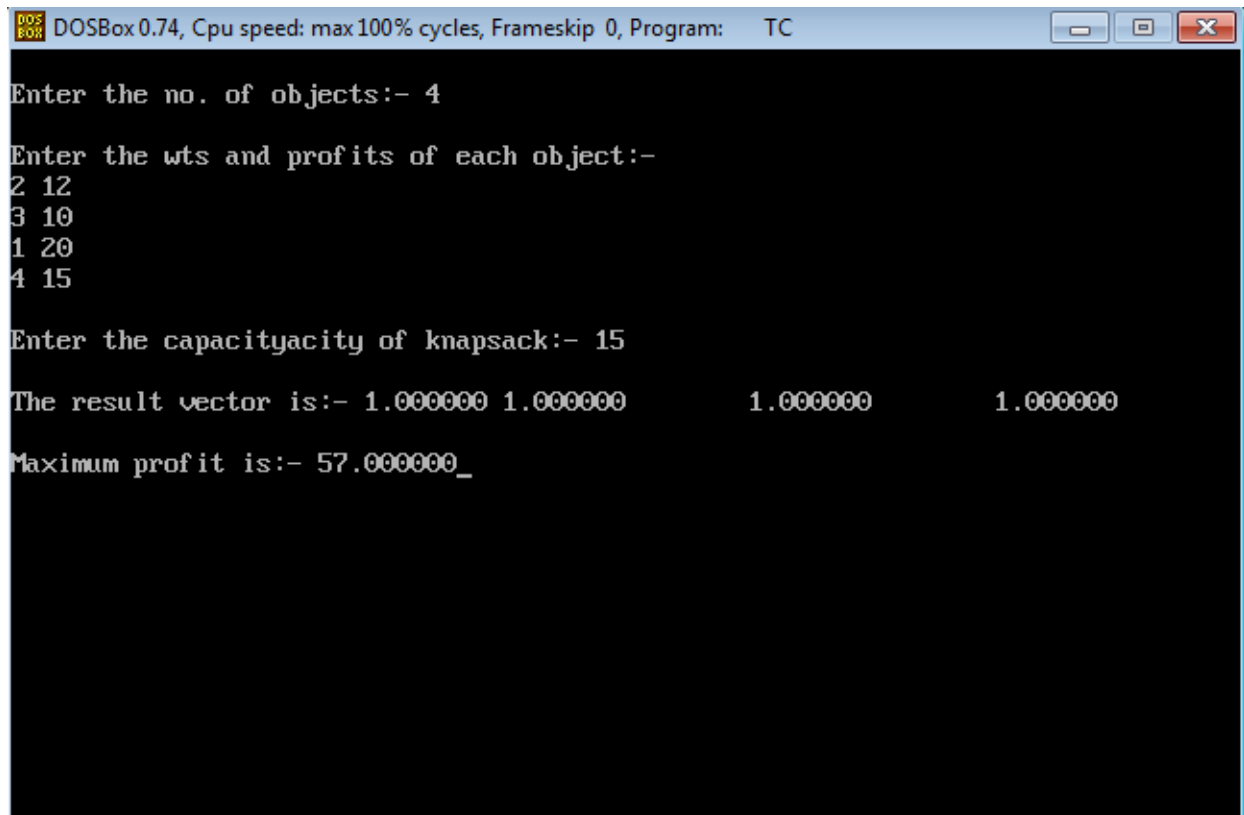
            temp = weight[j];
            weight[j] = weight[i];
            weight[i] = temp;

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

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

```

## **OUTPUT:**



The screenshot shows a DOSBox window titled "DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC". The window contains a black terminal with white text. The text shows the user inputting the number of objects (4), the weights and profits for each object (2 12, 3 10, 1 20, 4 15), and the knapsack capacity (15). The program then outputs the result vector (1.000000 1.000000 1.000000 1.000000) and the maximum profit (57.000000\_).

```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC
Enter the no. of objects:- 4
Enter the wts and profits of each object:-
2 12
3 10
1 20
4 15
Enter the capacity of knapsack:- 15
The result vector is:- 1.000000 1.000000 1.000000 1.000000
Maximum profit is:- 57.000000_
```

## **Conclusion:**

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

## Experiment No.6

### Prim's algorithm

**Aim:-** Program to find minimum cost spanning tree using Prim's algorithm.

**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  $\infty$  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]  $\neq$  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]  $\neq$  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 first 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:**

```

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

Enter the no. of vertices :4

Enter the costs of edges in matrix form :
4 5 6 7
1 2 3 4
4 5 6 7
5 8 8 9

The matrix is : 4      5      6      7
1      2      3      4
4      5      6      7
5      8      8      9

The min spanning tree is:- 101
204
305

Min cost : 10_

```

**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.7 Kruskal's algorithm

**Aim:-** Program to find minimum cost spanning tree using Prim's algorithm.

**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          $j := \text{Find}(u)$ ;  $k := \text{Find}(v)$ ;
15         if  $(j \neq k)$  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  $(i \neq n - 1)$  then write ("No spanning tree");
24     else return  $mincost$ ;
25 }
```

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

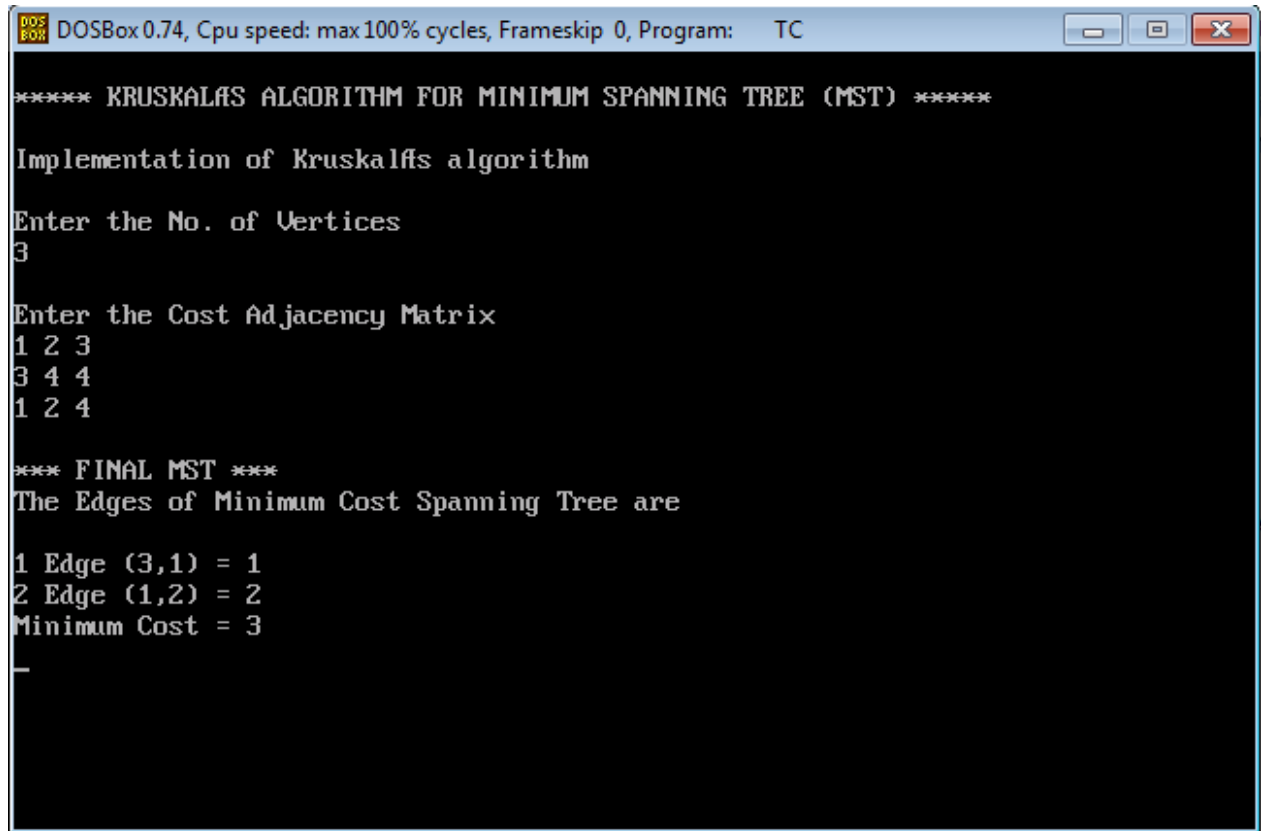
```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
clrscr();
printf("\n***** KRUSKAL'S ALGORITHM FOR MINIMUM SPANNING TREE (MST)
*****\n");
printf("\nImplementation of Kruskal's algorithm\n");
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:**



```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

***** KRUSKAL'S ALGORITHM FOR MINIMUM SPANNING TREE (MST) *****

Implementation of Kruskal's algorithm

Enter the No. of Vertices
3

Enter the Cost Adjacency Matrix
1 2 3
3 4 4
1 2 4

*** FINAL MST ***
The Edges of Minimum Cost Spanning Tree are

1 Edge (3,1) = 1
2 Edge (1,2) = 2
Minimum Cost = 3
_
```

**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.8

### Dijkstra's algorithm

**Aim:-** Program to find a single source shortest path for a given graph.

**Objective:** Write a C program to find a single source shortest path for a given graph.

#### Single Source Shortest Paths Problem:

For a given vertex called the source in a weighted connected graph, find the shortest paths to all its other vertices. Dijkstra's algorithm is the best known algorithm for the single source shortest paths problem. This algorithm is applicable to graphs with nonnegative weights only and finds the shortest paths to a graph's vertices in order of their distance from a given source. It finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on. It is applicable to both undirected and directed graphs

```
Algorithm : Dijkstra(G,s)
//Dijkstra's algorithm for single-source shortest paths
//Input :A weighted connected graph G=(V,E) with nonnegative weights and its vertex s
//Output : The length  $d_v$  of a shortest path from s to v and its penultimate vertex  $p_v$  for
//every v in V.
{
    Initialise(Q)    // Initialise vertex priority queue to
                    // empty for every vertex v in V do
    {
         $d_v \leftarrow \infty$ ;  $p_v \leftarrow \text{null}$ 
        Insert(Q,v, $d_v$ ) //Initialise vertex priority queue in the priority queue
    }
     $d_s \leftarrow 0$ ; Decrease(Q,s  $d_s$ )      //Update priority of s with
     $V_t \leftarrow \emptyset$ 
    for  $i \leftarrow 0$  to  $|V|-1$  do
    {
         $u^* \leftarrow \text{DeleteMin}(Q)$     //delete the minimum priority
         $V_t \leftarrow V_t \cup \{u^*\}$ 
        for every vertex u in  $V - V_t$  that is adjacent to  $u^*$  do
        {
            if  $d_{u^*} + w(u^*,u) < d_u$ 
            {
                 $d_u \leftarrow d_{u^*} + w(u^*, u)$ ;  $p_u \leftarrow u^*$ 
                Decrease(Q,u, $d_u$ )
            }
        }
    }
}
```

**Complexity:** The Time efficiency for graphs represented by their weight matrix and the priority queue implemented as an unordered array and for graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is  $O(|E| \log |V|)$ .

**Program:**

```
/*To find shortest paths to other vertices using Dijkstra's
```

```
algorithm.*/#include<stdio.h>
```

```
void dij(int,int [20][20],int [20],int [20],int);
```

```
void main()
```

```
{
```

```
    int
```

```
    i,j,n,visited[20],source,cost[20][20],d[20];clrscr();
```

```
    printf("Enter no. of vertices: ");
```

```
    scanf("%d",&n);
```

```
    printf("Enter the cost adjacency matrix\n");for(i=1;i<=n;i++)
```

```
    {
```

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

```
        {
```

```
            scanf("%d",&cost[i][j]);
```

```
        }
```

```
    }
```

```
    printf("\nEnter the source node: ");
```

```
    scanf("%d",&source);
```

```
    dij(source,cost,visited,d,n);
```

```
    for(i=1;i<=n;i++)
```

```
        if(i!=source)
```

```
            printf("\nShortest path from %d to %d is %d",source,i,d[i]);
```

```
    getch();
```

```
}
```

```
void dij(int source,int cost[20][20],int visited[20],int d[20],int n)
```

```

{
    int i,j,min,u,w;
    for(i=1;i<=n;i++)
    {
        visited[i]=0;
        d[i]=cost[source]
        [i];
    }
    visited[source]
    =1;
    d[source]=0;
    for(j=2;j<=n;j++)
    {
        min=999;
        for(i=1;i<=n;i++)
        {
            if(!visited[i])
            {
                if(d[i]<min)
                {
                    min=d[i]
                    ];u=i;
                }
            }

        } //for i
        visited[u]=
        1;
        for(w=1;w<=n;w++)
        {
            if(cost[u][w]!=999 && visited[w]==0)
            {
                if(d[w]>cost[u][w]+d[u])
                d[w]=cost[u][w]+d[u];
            }
        } //for
        w
    } // for j
}

```

## OUTPUT:

```
Turbo C++ IDE
Enter no. of vertices: 6
Enter the cost adjacency matrix
999 3 999 999 6 5
3 999 1 999 999 4
999 1 999 6 999 4
999 999 6 999 8 5
6 999 999 8 999 2
5 4 4 5 2 999

Enter the source node: 1

Shortest path from 1 to 2 is 3
Shortest path from 1 to 3 is 4
Shortest path from 1 to 4 is 10
Shortest path from 1 to 5 is 6
Shortest path from 1 to 6 is 5_
```



## Experiment No.9

### **0/1 Knapsack**

**Aim:-** Program to implement 0-1 knapsack problem using dynamic programming.

**Objective:** To write a C program to find the solution for a 0-1 knapsack problem using dynamic programming.

#### **0/1 Knapsack problem:**

Given: A set  $S$  of  $n$  items, with each item  $i$  having

- $b_i$  - a positive benefit
- $w_i$  - a positive weight

Goal: Choose items with maximum total benefit but with weight at most  $W$ . i.e.

- Objective: maximize  $\sum_{i=1}^n b_i$
- Constraint:  $\sum_{i=1}^n w_i \leq W$

**Algorithm:** 0/1Knapsack( $S, W$ )

//Input: set  $S$  of items with benefit  $b_i$  and weight  $w_i$ ; max. weight  $W$

//Output: benefit of best subset with weight at most  $W$

// $S_k$ : Set of items numbered 1 to  $k$ .

//Define  $B[k,w]$  = best selection from  $S_k$  with weight exactly equal to  $w$

```
{
    for  $w \leftarrow 0$  to  $n-1$  do
         $B[w] \leftarrow 0$ 
    for  $k \leftarrow 1$  to  $n$ 
    do
    {
        for  $w \leftarrow W$  downto  $w_k$  do
```

**Complexity:** The Time efficiency and Space efficiency of 0/1 Knapsack algorithm is  $\Theta(nW)$ .

**Program:**

```
#include<stdio.h>
#define MAX 50

int p[MAX],w[MAX],n;
int knapsack(int,int);
int max(int,int);

void main()
{
    int m,i,optsoln;
    clrscr();

    printf("Enter no. of objects:
    ");scanf("%d",&n);

    printf("\nEnter the
    weights:\n");for(i=1;i<=n;i++)
        scanf("%d",&w[i]);

    printf("\nEnter the
    profits:\n");for(i=1;i<=n;i++)
        scanf("%d",&p[i]);

    printf("\nEnter the knapsack
    capacity:");scanf("%d",&m);

    optsoln=knapsack(1,m);

    printf("\nThe optimal solution
    is:%d",optsoln);getch();
}

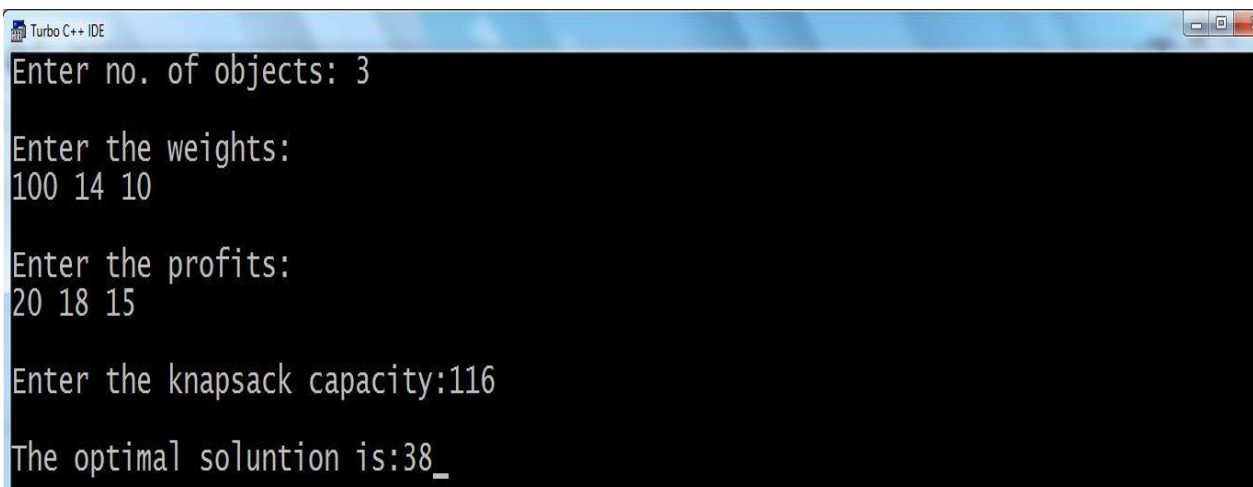
int knapsack(int i,int m)
{
    if(i==n)
        return (w[n]>m) ? 0 : p[n];

    if(w[i]>m)
        return knapsack(i+1,m);

    return max(knapsack(i+1,m),knapsack(i+1,m-w[i])+p[i]);
}
```

```
int max(int a, int b)
{
    if(a>b)
        return a;
    else
        return b;
}
```

### OUTPUT:

A screenshot of the Turbo C++ IDE window. The title bar reads "Turbo C++ IDE". The main text area shows the following input and output:

```
Enter no. of objects: 3
Enter the weights:
100 14 10
Enter the profits:
20 18 15
Enter the knapsack capacity:116
The optimal solution is:38_
```

