Program 1:

Design and implement C/C++ program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm.

Algorithm:

Algorithm : Kruskal(G)
// Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G=(V,E)
//Output: E_T,the set of edges composing a minimum spanning tree of G
{
Sort E in non decreasing order of the edge weights $w(e_{i1})<=\ldots\ldots>= w(e_{i\,|E|})$
$E_T \leftarrow \varnothing$; ecounter $\leftarrow$ 0 //Initialize the set of tree edges and its size
k $\leftarrow$ 0 //initialize the number of processed edges
while ecounter <|V|-1 do
{
k $\leftarrow$ k+1
if $E_T U \{e_{ik}\}$ is acyclic
$E_T \leftarrow E_T$ U $\{e_{ik}\}$; ecounter $\leftarrow$ ecounter+1
}
return E_T
}

Code:

```
#include<stdio.h>

int ne=1, min_cost=0;

void main()

{

int n,i,j,min,a,u,b,v,cost[20][20],parent[20];

printf("Enter the number of vertices: ");

scanf("%d", &n);

printf("\nEnter the cost matrix: \n");

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

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

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

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

parent[i]=0;
```

```
printf("\n The edges of spanning tree are\n");
while(ne<n)
{
min=999;
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
if(cost[i][j]<min)
{
min=cost[i][j];
a=u=i;
b=v=j;
}
}
}
while(parent[u])
u=parent[u];
while(parent[v])
v=parent[v];
if(u!=v)
{
printf("Edge %d\t(%d->%d)=%d\n",ne++,a,b,min);
min_cost=min_cost+min;
parent[v]=u;
}
cost[a][b]=cost[a][b]=999;
}
printf("\n Minimum cost=%d\n",min_cost);
}
```

Output:

```
Enter the number of vertices: 6

Enter the cost matrix:
23 34 56 78 34 12
11 33 78 899 89 34
222 44 66 87 98 444
11 33 44 76 54 22
14 56 78 89 90 54
12 45 67 89 65 46

 The edges of spanning tree are
Edge 1  (2->1)=11
Edge 2  (4->1)=11
Edge 3  (1->6)=12
Edge 4  (5->1)=14
Edge 5  (3->2)=44

 Minimum cost=92
```

Program 2:

Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.

Algorithm:

Algorithm : Prim(G)
// Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G=(V,E)
//Output: E_T, the set of edges composing a minimum spanning tree of G
{
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$

for i ← 0 to |V| - 1 do
find a minimum-weight edge e* = (v*, u*) among all the edges (v, u)
such that v is in V_T and u is in $V-V_T$
$V_T \leftarrow V_T \cup \{u^*\}$
$E_T \leftarrow E_T \cup \{e^*\}$

return E_T
}

Code:

```
#include<stdio.h>

int ne=1,min_cost=0;

void main()

{

int n,i,j,min,cost[20][20],a,u,b,v,source,visited[20];

printf("Enter the number of nodes: ");

scanf("%d",&n);

printf("Enter the cost matrix:\n");

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

{

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

{

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

```
}
}
for(i=1;i<=n;i++)
visited[i]=0;
printf("Enter the root node: ");
scanf("%d",&source);
visited[source]=1;
printf("\n Minimum cost spanning tree is\n");
while(ne<n)
{
min=999;
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
if(cost[i][j]<min)
if(visited[i]==0)
continue;
else
{
min=cost[i][j];
a=u=i;
b=v=j;
}
}
}
if(visited[u]==0||visited[v]==0)
{
printf("\nEdge %d\t(%d->%d)=%d\n",ne++,a,b,min);
min_cost=min_cost+min;
```

```
visited[b]=1;
}
cost[a][b]=cost[b][a]=999;
}
printf("\nMinimum cost=%d\n",min_cost);
}
```

Output:

```
Enter the number of nodes: 4
Enter the cost matrix:
23 567 1 4
34 3 67 999
2 4 65 34
34 67 98 12
Enter the root node: 1

 Minimum cost spanning tree is

Edge 1  (1->3)=1

Edge 2  (1->4)=4

Edge 3  (3->2)=4

Minimum cost=9
```

Program 3:

a. Design and implement C/C++ Program to solve All-Pairs Shortest Paths problem using Floyd's algorithm.
b. Design and implement C/C++ Program to find the transitive closure using Warshal's algorithm.

Program 3a

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
}
```

Code:

```
#include<stdio.h>
int min(int a, int b)
{
return(a<b?a:b);
}
void floyd(int D[][10],int n)
{
for(int k=1;k<=n;k++)
for(int i=1;i<=n;i++)
for(int j=1;j<=n;j++)
D[i][j]=min(D[i][j],D[i][k]+D[k][j]);
}
```

```
int main()
{
int n, cost[10][10];
printf("Enter the number of vertices: ");
scanf("%d",&n);
printf("Enter the cost matrix \n");
for(int i=1;i<=n;i++)
for(int j=1;j<=n;j++)
scanf("%d",&cost[i][j]);
floyd(cost,n);
printf("All pair shortest path \n");
for(int i=1;i<=n;i++)
{
for(int j=1;j<=n;j++)
printf("%d ",cost[i][j]);
printf("\n");
}
}
```

Output:

```
Enter the number of vertices: 4
Enter the cost matrix
33 66 2 888
23 6 89 999
999 7 45 222
23 999 56 23
All pair shortest path
32 9 2 224
23 6 25 247
30 7 32 222
23 32 25 23
```

Program 3b

Algorithm:

Algorithm Warshall(A[1..n,1..n])
//Implements Warshall's algorithm for computing the transitive closure
//Input: The Adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of digraph
{
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ to n do
{
for $i \leftarrow 1$ to n do
{
for $j \leftarrow 1$ to n do
{
$R^{(k)}[i,j] \leftarrow R^{(k-1)}[i,j]$ or $R^{(k-1)}[i,k]$ and $R^{(k-1)}[k,j]$
}
}
}
return $R^{(n)}$
}

Code:

```
#include<stdio.h>
void warshal(int A[][10], int n)
{
for(int k=1;k<=n;k++)
for(int i=1;i<=n;i++)
for(int j=1;j<=n;j++)
A[i][j]=A[i][j] || (A[i][k]&&A[k][j]);
}
void main()
{
int n,adj[10][10];
```

```
printf("Enter the number of vertices: ");
scanf("%d",&n);
printf("Enter the adjacency matrix: \n");
for(int i=1;i<=n;i++)
for(int j=1;j<=n;j++)
scanf("%d",&adj[i][j]);
warshal(adj,n);
printf("Transitive closure of the given graph is \n");
for(int i=1;i<=n;i++)
{
for(int j=1;j<=n;j++)
printf("%d",adj[i][j]);
printf("\n");
}
}
```

Output:

```
Enter the number of vertices: 4
Enter the adjacency matrix:
0 1 0 0
0 0 0 1
0 0 0 0
1 0 0 0
Transitive closure of the given graph is
1101
1101
0000
1101
```

Program 4:

Design and implement C/C++ program to find shortest path from a given vertex in a weighted connected graph to other vertices using Dijkstra's algorithm.

Algorithm:

```
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 dv of a shortest path from s to v and its penultimate vertex pv for
//every v in V.
{
        Initialise(Q)      // Initialise vertex priority queue to empty
        for every vertex v in V do
        {
                dv←∞; pv←null
                Insert(Q,v,dv)   //Initialise vertex priority queue in the priority queue
        }
        ds←0;  Decrease(Q,s ds)         //Update priority of s with ds
        Vt←Ø
        for i←0 to |v|-1 do
        {
                u* ← DeleteMin(Q)      //delete the minimum priority element
                Vt ←Vt U {u*}
                for every vertex u in V-Vt that is adjacent to u* do
                {
                        if du* + w(u*,u)<du
                        {
                                du←du* + w(u*, u):  pu←u*
                                Decrease(Q,u,du)
                        }
                }
        }
}
```

Code:

```
#include<stdio.h>

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

int minm(int m, int n)

{

return((m<n)?m:n);
```

```
}
void dijkstra(int source)
{
int s[10]={0};
int min, w=0;
for(int i=0;i<n;i++)
dist[i]=cost[source][i];
dist[source]=0;
s[source]=1;
for(int i=0;i<n-1;i++)
{
min=999;
for(int j=0;j<n;j++)
{
if((s[j]==0)&&(min>dist[j]))
{
min=dist[j];
w=j;
}
}
s[w]=1;
for(int v=0;v<n;v++)
{
if(s[v]==0&&cost[w][v]!=999)
{
dist[v]=minm(dist[v],dist[w]+cost[w][v]);
}
}
}
}
int main()
```

```
{
int source;
printf("Enter the number of vertices: ");
scanf("%d",&n);
printf("Enter the cost matrix \n");
for(int i=0;i<n;i++)
for(int j=0;j<n;j++)
scanf("%d",&cost[i][j]);
printf("Enter the source vertex: ");
scanf("%d",&source);
dijkstra(source);
printf("The shortest distance is: \n");
for(int i=0;i<n;i++)
printf("Cost from %d to %d is %d\n",source,i,dist[i]);
}
```

Output:

```
Enter the number of vertices: 4
Enter the cost matrix
2 4 6 8
7 9 7 23
54 6 8 3
21 4 6 9
Enter the source vertex: 2
The shortest distance is:
Cost from 2 to 0 is 13
Cost from 2 to 1 is 6
Cost from 2 to 2 is 0
Cost from 2 to 3 is 3
```

Program 5:

Design and implement C/C++ program to obtain the Topological ordering of vertices in a given digraph.

Algorithm:

```
Algorithm  topological_sort(a,n,T)
        //purpose :To obtain the sequence of jobs to be executed resut
          In topolocical order
        // Input:a-adjacency matrix of the given graph
        //n-the number of vertices in the graph
        //output:
        // T-indicates the jobs that are to be executed in the order

              For j<-0 to n-1 do
                     Sum-0
                     For  i<- 0to n-1 do
                        Sum<-sum+a[i][j]
                End for
                  Top ← -1
                   For i<- 0 to n-1 do
                          If(indegree [i]=0)
                                Top <-top+1
                                S[top]<- i
                   End if
                   End for
           While top!= 1
                u<-s[top]
              top<-top-1
           Add u to solution vector T
               For each vertex v adjacent to u
                      Decrement indegree [v] by one
                           If(indegree [v]=0)
                             Top<-top+1
                             S[top]<-v
        End if
        End for
        End while
        Write T
        return
```

Code:

```
#include<stdio.h>

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

```
void cal_colsum()
{
for(int j=0;j<n;j++)
{
colsum[j]=0;
for(int i=0;i<n;i++)
colsum[j]+=cost[i][j];
}
}
void source_removal()
{
int i,j,k,select[10]={0};
printf("Topological ordering is: ");
for(i=0;i<n;i++)
{
cal_colsum();
for(j=0;j<n;j++)
{
if(select[j]==0&&colsum[j]==0)
break;
}
printf("%d",j);
select[j]=1;
for(k=0;k<n;k++)
cost[j][k]=0;
}
}
void main()
{
printf("Enter the number of vertices: ");
```

```
scanf("%d",&n);
printf("Enter the cost matrix \n");
for(int i=0;i<n;i++)
for(int j=0;j<n;j++)
scanf("%d",&cost[i][j]);
source_removal();
}
```

Output:

```
Enter the number of vertices: 5
Enter the cost matrix
0 1 1 0 0
0 0 0 1 1
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0
Topological ordering is: 01234
```

Program 6:

Design and implement C/C++ program to solve 0/1 Knapsack Problem using Dynamic Programming method.

Algorithm:

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
// Sk: Set of items numbered 1 to k.
//Define B[k,w] = best selection from Sk with weight exactly equal to w

```
{
    for w ← 0 to n-1 do
            B[w] ← 0
    for k ← 1 to n do
    {
        for w ← W downto wk do
        {
            if B[w-wk]+bk > B[w] then
                B[w] ← B[w-wk]+bk
        }
    }
}
```

Code:

```
#include<stdio.h>

int n,m,p[10],w[10];

int max(int a, int b)

{

return(a>b?a:b);

}

void knapsack_DP()

{
```

```
int V[10][10],i,j;
for(i=0;i<=n;i++)
for(j=0;j<=m;j++)
if(i==0||j==0)
V[i][j]=0;
else
if(j<w[i])
V[i][j]=V[i-1][j];
else
V[i][j]=max(V[i-1][j],p[i]+V[i-1][j-w[i]]);
for(i=0;i<=n;i++)
{
for(j=0;j<=m;j++)
printf("%d ",V[i][j]);
printf("\n");
}
printf("Items included are: ");
while(n>0)
{
if(V[n][m]!=V[n-1][m])
{
printf("%d ",n);
m=m-w[n];
}
```

```
n--;
}
}
int main()
{
int i;
printf("Enter the number of items: ");
scanf("%d",&n);
printf("Enter the weights of n items: ");
for(i=1;i<=n;i++)
scanf("%d",&w[i]);
printf("Enter the prices of n items: ");
for(i=1;i<=n;i++)
scanf("%d",&p[i]);
printf("Enter the capacity of Knapsack: ");
scanf("%d",&m);
knapsack_DP();
}
```

Output:

```
Enter the number of items: 4
Enter the weights of n items: 7 3 4 5
Enter the prices of n items: 42 12 40 25
Enter the capacity of Knapsack: 10
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 42 42 42 0
0 0 0 12 12 12 12 42 42 42 0
0 0 0 12 40 40 40 52 52 52 0
0 0 0 12 40 40 40 52 52 65 65
Items included are: 4 3
```

Program 7:

Design and implement C/C++ program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.

Code:

```
#include<stdio.h>

int n,m,p[10],w[10];

void greedy_knapsack()

{

float max, profit=0;

int k=0,i,j;

printf("item included is: ");

for(i=0;i<n;i++)

{

max=0;

for(j=0;j<n;j++)

{

if(((float)p[j])/w[j]>max)

{

k=j;

max=((float)p[j])/w[j];

}

}

if(w[k]<=m)
```

```
{
printf("%d ",k);
m=m-w[k];
profit=profit+p[k];
p[k]=0;
}
else
break;
}
printf("\nDiscrete Knapsack profit = %f\n",profit);
printf("Continuous Knapsack also includes item %d with portion: %f\n",k,((float)m)/w[k]);
profit=profit+((float)m)/w[k]*p[k];
printf("Continuous Knapsack profit = %f\n",profit);
}
int main()
{
int i;
printf("Enter the number of items: ");
scanf("%d", &n);
printf("Enter the weights of n items: ");
for(i=0;i<n;i++)
scanf("%d",&w[i]);
printf("Enter the prices of n items: ");
for(i=0;i<n;i++)
```

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

printf("Enter the capacity of Knapsack: ");

scanf("%d",&m);

greedy_knapsack();

}
```

Output:

```
Enter the number of items: 4
Enter the weights of n items: 2 1 3 2
Enter the prices of n items: 12 10 20 15
Enter the capacity of Knapsack: 5
item included is: 1 3
Discrete Knapsack profit = 25.000000
Continuous Knapsack also includes item 2 with portion: 0.666667
Continuous Knapsack profit = 38.333332
```

Program 8:

Design and implement C/C++ program to find a subset of a given set S={S1, S2, ……….., Sn} of n positive integers whose sum is equal to a given positive integer d.

Algorithm:

Algorithm SumOfSub(s, k, r)
//Find all subsets of w[1…n] that sum to m. The values of x[j], 1<= j < k, have already
//been determined. $s=\sum_{j=1}^{k-1} w[j]*x[j]$ and $r=\sum_{j=k}^{n} w[j]$. The w[j]'s are in ascending order.

```
{
    x[k] ← 1  //generate left child
    if (s+w[k] = m)
            write (x[1...n])  //subset found
    else if ( s + w[k]+w[k+1] <= m)
            SumOfSub( s + w[k], k+1, r-w[k])
    //Generate right child
    if( (s + r - w[k] >= m) and (s + w[k+1] <= m) )
    {
            x[k] ← 0
            SumOfSub( s, k+1, r-w[k] )
    }
}
```

Code:

```
#include<stdio.h>
int x[10],w[10],count,d;
void sum_of_subsets(int s, int k, int rem)
{
x[k]=1;
if(s+w[k]==d)
{
printf("subset=%d\n",++count);
for(int i=0;i<=k;i++)
if(x[i]==1)
printf("%d ",w[i]);
```

```
printf("\n");
}
else
if(s+w[k]+w[k+1]<=d)
sum_of_subsets(s+w[k],k+1,rem-w[k]);
if((s+rem-w[k]>=d)&&(s+w[k+1])<=d)
{
x[k]=0;
sum_of_subsets(s,k+1,rem-w[k]);
}
}
int main()
{
int sum=0,n;
printf("enter number of elements:");
scanf("%d",&n);
printf("enter the elements in increasing order:");
for(int i=0;i<n;i++)
{
scanf("%d",&w[i]);
sum=sum+w[i];
}
printf("enter the sum:");
scanf("%d",&d);
if((sum<d) || (w[0]>d))
printf("No subset possible\n");
else
sum_of_subsets(0,0,sum);
}
```

Output:

```
sru-ubuntu@srujani-Ubuntu-VirtualBox:~$ gcc p8.c
sru-ubuntu@srujani-Ubuntu-VirtualBox:~$ ./a.out
enter number of elements:5
enter the elements in increasing order:1 2 3 4 5
enter the sum:10
subset=1
1 2 3 4
subset=2
1 4 5
subset=3
2 3 5
sru-ubuntu@srujani-Ubuntu-VirtualBox:~$
```

Program 9:

Design and implement C/C++ program to sort a given set of n integer elements using selection sort method and compute its time complexity. Run the program for varied values of n>5000 and record the time taken to sort. 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.

Algorithm:

ALGORITHM *SelectionSort*($A[0..n-1]$)

//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
    $min \leftarrow i$
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[j] < A[min]$ $min \leftarrow j$
    swap $A[i]$ and $A[min]$

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
void selectionSort(int arr[], int n)
{
int i,j,min_idx;
for(i=0;i<n-1;i++)
{
min_idx=i;
for(j=i+1;j<n;j++)
{
if(arr[j]<arr[min_idx])
{
min_idx=j;
}
}
```

```
int temp=arr[min_idx];
arr[min_idx]=arr[i];
arr[i]=temp;
}
}
int main()
{
int n,i;
clock_t start, end;
double cpu_time_used;
int sizes[]={5000,10000,15000,20000,25000};
for(i=0;i<sizeof(sizes)/sizeof(sizes[0]);i++)
{
n=sizes[i];
int arr[n];
srand(time(NULL));
for(int j=0;j<n;j++)
{
arr[j]=rand();
}
start=clock();
selectionSort(arr, n);
end=clock();
cpu_time_used=((double)(end-start)) / CLOCKS_PER_SEC;
printf("\n Time taken to sort array of size %d: %f seconds\n", n, cpu_time_used);
}
return 0;
}
```

Output:

```
Time taken to sort array of size 5000: 0.046000 seconds

Time taken to sort array of size 10000: 0.141000 seconds

Time taken to sort array of size 15000: 0.328000 seconds

Time taken to sort array of size 20000: 0.547000 seconds

Time taken to sort array of size 25000: 0.890000 seconds
```

Program 10:

Design and implement C/C++ program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n>5000 and record the time taken to sort. 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.

Algorithm:

ALGORITHM *Quicksort*($A[l..r]$)
//Sorts a subarray by quicksort
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right indices
// l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
$s \leftarrow Partition(A[l..r])$ //s is a split position
$Quicksort(A[l..s-1])$
$Quicksort(A[s+1..r])$

ALGORITHM *Partition*($A[l..r]$)
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
// indices l and r ($l < r$)
//Output: A partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l;\quad j \leftarrow r+1$
repeat
repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
swap($A[i]$, $A[j]$)
until $i \geq j$
swap($A[i]$, $A[j]$) //undo last swap when $i \geq j$
swap($A[l]$, $A[j]$)
return j

Code:

```
#include<stdio.h>

#include<stdlib.h>

#include<sys/time.h>

#include<time.h>

void fnGenRandInput(int[], int);

void fnDispArray(int[], int);

int fnPartition(int[], int, int);

void fnQuickSort(int [], int, int);
```

```
void fnSwap(int*, int*);
void fnSwap(int *a, int *b)
{
int t=*a;
*a=*b;
*b=t;
}
int main(int argc, char **argv)
{
FILE *fp;
struct timeval tv;
double dStart, dEnd;
int iaArr[500000],iNum,i,iChoice;
for(;;)
{
printf("\n1.Plot the Graph\n2.QuickSort\n3.Exit");
printf("\nEnter your choice\n");
scanf("%d",&iChoice);
switch(iChoice)
{
case 1:
fp=fopen("QuickPlot.dat","w");
for(i=100;i<100000;i+=100)
{
fnGenRandInput(iaArr,i);
gettimeofday(&tv,NULL);
dStart=tv.tv_sec+(tv.tv_usec/1000000.0);
fnQuickSort(iaArr,0,i-1);
gettimeofday(&tv,NULL);
dEnd=tv.tv_sec+(tv.tv_usec/1000000.0);
```

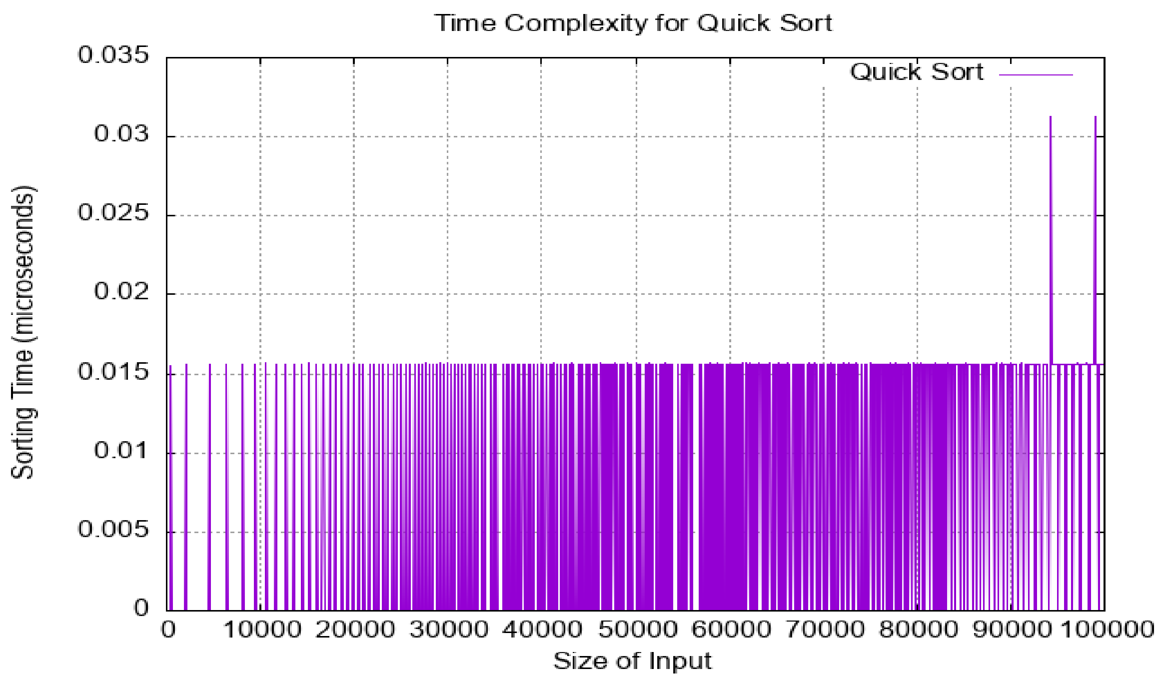
```
fprintf(fp,"%d\t%lf\n",i,dEnd-dStart);
}
fclose(fp);
printf("\nData File generated and stored in file<QuickPlot.dat>.\n Use a plotting utility\n");
break;
case 2:
printf("\nEnter the number of elements to sort\n");
scanf("%d",&iNum);
printf("\nUnsorted Array\n");
fnGenRandInput(iaArr,iNum);
fnDispArray(iaArr,iNum);
fnQuickSort(iaArr,0,iNum-1);
printf("\nSorted Array\n");
fnDispArray(iaArr,iNum);
break;
case 3:
exit(0);
}
}
return 0;
}
int fnPartition(int a[], int l, int r)
{
int i,j;
int p;
p=a[l];
i=l;
j=r+1;
do
{
```

```
do{i++;}
while(a[i]<p);
do{j--;}
while(a[j]>p);
fnSwap(&a[i],&a[j]);
}
while(i<j);
fnSwap(&a[i],&a[j]);
fnSwap(&a[l],&a[j]);
return j;
}
void fnQuickSort(int a[], int l, int r)
{
int s;
if(l<r)
{
s=fnPartition(a,l,r);
fnQuickSort(a,l,s-1);
fnQuickSort(a,s+1,r);
}
}
void fnGenRandInput(int X[], int n)
{
srand(time(NULL));
for(int i=0;i<n;i++)
{
X[i]=rand()%10000;
}
}
void fnDispArray(int X[], int n)
```

```
{
for(int i=0;i<n;i++)
printf("%5d\n",X[i]);
}
```

Output:

```
Enter the number of elements to sort
4

Unsorted Array
   581
  3498
  4832
  4542

Sorted Array
   581
  3498
  4542
  4832

1.Plot the Graph
2.QuickSort
3.Exit
Enter your choice
1

Data File generated and stored in file < QuickPlot.dat >.
 Use a plotting utility

1.Plot the Graph
2.QuickSort
3.Exit
```



Program 11:

Design and implement C/C++ program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n>5000 and record the time taken to sort. 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.

Algorithm:

```
ALGORITHM Mergesort(A[0..n − 1])
    //Sorts array A[0..n − 1] by recursive mergesort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    if n > 1
        copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
        copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
        Mergesort(B[0..⌊n/2⌋ − 1])
        Mergesort(C[0..⌈n/2⌉ − 1])
        Merge(B, C, A)
```

```
ALGORITHM Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
    //Merges two sorted arrays into one sorted array
    //Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
    //Output: Sorted array A[0..p + q − 1] of the elements of B and C
    i ← 0; j ← 0; k ← 0
    while i < p and j < q do
        if B[i] ≤ C[j]
            A[k] ← B[i]; i ← i + 1
        else A[k] ← C[j]; j ← j + 1
        k ← k + 1
    if i = p
        copy C[j..q − 1] to A[k..p + q − 1]
    else copy B[i..p − 1] to A[k..p + q − 1]
```

Code:

```
#include<stdio.h>

#include<stdlib.h>
```

```
#include<time.h>// Function prototypes
void mergeSort(int arr[], int low, int high);
void merge(int arr[], int low, int mid, int high);
double timeMergeSort(int arr[], int n);
// Main function
int main() {
srand(time(NULL));
int step = 500;
printf("n\tTime (ms)\n");
for(int n = 500; n <= 10000; n += step) {
double totalTime = 0.0;
for (int i = 0; i < 5; i++) { // Repeat 5 times and take average time
// Generate random numbers to fill the array
int *arr = (int *)malloc(n * sizeof(int));
for (int j = 0; j < n; j++) {
arr[j] = rand() % 1000;
}
totalTime += timeMergeSort(arr, n);
free(arr);
}
double averageTime = totalTime / 5.0;
printf("%d\t%.2f\n", n, averageTime);
}
return 0;
```

```
}
// Merge sort algorithm
void mergeSort(int arr[], int low, int high) {
   if (low < high) {
         int mid = (low + high) / 2;
   mergeSort(arr, low, mid);
   mergeSort(arr, mid + 1, high);
   merge(arr, low, mid, high);
   }
}
void merge(int arr[], int low, int mid, int high) {
   int n1 = mid - low + 1;
   int n2 = high - mid;
   int *L = (int *)malloc(n1 * sizeof(int));
   int *R = (int *)malloc(n2 * sizeof(int));
   for(int i = 0; i < n1; i++)
      L[i] = arr[low + i];
   for(int j = 0; j < n2; j++)
      R[j] = arr[mid + 1 + j];
   int i=0;
   int j=0;
   int k=low;
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
```

```
        arr[k] = L[i];i++;} else {
            arr[k] = R[j];
    j++;
    }
    k++;
    }
    while(i < n1) {
        arr[k] = L[i];
    i++;
    k++;
    }
    while(j < n2) {
    arr[k] = R[j];
    j++;
    k++;
    }
    free(L);
    free(R);
}
// Timing function for Merge Sort
double timeMergeSort(int arr[], int n)
{
    clock_t start = clock();
    mergeSort(arr, 0, n - 1);
```

```
    clock_t end = clock();

    return ((double)(end - start)) * 1000.0 / CLOCKS_PER_SEC;

}
```

Output:

```
n        Time (ms)
500      0.00
1000     0.00
1500     3.00
2000     0.00
2500     0.00
3000     0.00
3500     3.20
4000     0.00
4500     3.00
5000     0.00
5500     3.20
6000     0.00
6500     3.20
7000     3.00
7500     3.20
8000     3.20
8500     0.00
9000     0.00
9500     6.20
10000    3.20
```

Program 12:

Design and implement C/C++ program for N Queen's problem using Backtracking.

Algorithm:

```
Algorithm NQueens (k, n)
//Using backtracking, this procedure prints all possible placements of n queens
//on an n x n chessboard so that they are non-attacking
{
        for i ← 1 to n do
        {
                if(Place(k,i) )
                {
                        x[k] ← i
                        if (k=n)
                                write ( x[1...n])
                        else
                                Nqueens (k+1, n)
                }
        }
}

Algorithm Place( k, i)
//Returns true if a queen can be placed in kth row and ith column. Otherwise it
//returns false. x[] is a global array whose first (k-1) values have been set. Abs(r)
//returns the absolute value of r.
{
        for j ← 1 to k-1 do
        {
                if (  (x[j]=i or Abs(x[j]-i) = Abs(j-k) )
                {
                        return false
                }
        }
}
```

Code:

```
#include<stdio.h>

#include<math.h>

#include<stdlib.h>

int place(int x[], int k)

{

for(int i=1;i<k;i++)
```

```
{
if((x[i] == x[k]) || (abs(x[i]-x[k]) == abs(i-k)))
return 0;
}
return 1;
}
int nqueens(int n)
{
int x[10],k,count=0;
k=1;
x[k]=0;
while(k!=0)
{
x[k]++;
while((x[k]<=n) && (!place(x,k)))
x[k]++;
if(x[k]<=n)
{
if(k==n)
{
printf("\nSolution %d\n", ++count);
for(int i=1;i<=n;i++)
{
for(int j=1;j<=n;j++)
```

```
printf("%c", j==x[i]?'Q':'X');

printf("\n");

}

}

else

{

++k;

x[k]=0;

}

}

else

k--;

}

return count;

}

void main()

{

int n;

printf("Enter the size of chessboard: ");

scanf("%d",&n);

printf("\n The number of possibilities are %d\n", nqueens(n));

}
```

Output:

```
sru-ubuntu@srujani-Ubuntu-VirtualBox:~$ gcc p12.c
sru-ubuntu@srujani-Ubuntu-VirtualBox:~$ ./a.out
Enter the size of chessboard: 4

Solution 1
XQXX
XXXQ
QXXX
XXQX

Solution 2
XXQX
QXXX
XXXQ
XQXX

 The number of possibilities are 2
sru-ubuntu@srujani-Ubuntu-VirtualBox:~$ ./a.out
Enter the size of chessboard: 3

 The number of possibilities are 0
sru-ubuntu@srujani-Ubuntu-VirtualBox:~$ ./a.out
Enter the size of chessboard: 1

Solution 1
Q

 The number of possibilities are 1
sru-ubuntu@srujani-Ubuntu-VirtualBox:~$
```