	PART-B	
1.	"Sunshine" a job search portal is looking for engineering graduates, they need to sort the candidate's resume based on their ranking(Average Percentage). Ranking should be generated randomly. Design and develop a program in C to sort the resumes by using **heap sort** algorithm. Determine the time required to sort the elements. Repeat the experiment for different values of n and plot a graph of the time taken versus n.(n=no of elements).	CO3/1,4, 9
2.	Design and develop a program in C to implement horspool's algorithm for string matching.	CO3/1,4, 9
3.	Design and develop a program in C to Compute the transitive closure of a given directed graph using Warshall's algorithm. Repeat the experiment for different values of n and plot a graph of the time taken versus n (n=no of nodes).	CO4/1,4, 9
4.	Design and develop a program in C to find the all pair shortest path for a given graph using Floyd's algorithm. Determine the time required to find the solution. Repeat the experiment for different values of n and plot a graph of the time taken versus n (n=no of nodes).	CO4/1,4, 9
5.	Design and develop a program in C to find solution for Knapsack problem using dynamic programming. Determine the time required to find the solution. Repeat the experiment for different values of n and plot a graph of the time taken versus n (n=no of items).	CO4/1,4, 9
6.	Design and develop a program in C to find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm. Repeat the experiment for different values of n and plot a graph of the time taken versus n (n=no of nodes).	CO4/1,4, 9
7.	Design and develop a program in C to find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm. Repeat the experiment for different values of n and plot a graph of the time taken versus n (n=no of nodes).	CO4/1,4, 9
8.	Design and develop a program in C to find the single source shortest path for a given graph using Dijkstra's algorithm. Determine the time required to find the solution. Repeat the experiment for different values of n and plot a graph of the time taken versus n (n=no of nodes).	CO4/1,4, 9

9.	Design and develop a program in C to implement n/Queens problem by using backtracking method.	CO5/1,4, 9
10.	Design and develop a program in C to find the subset of a given set S={S1,S2,S3,…..Sn} of n positive integers whose sum is equal to a positive integer d and to display a suitable message if the given problem instance does not have the solution.	CO5/1,4, 9

1. "Sunshine" a job search portal is looking for engineering graduates, they need to sort the candidate's resume based on their ranking(Average Percentage). Ranking should be generated randomly. Design and develop a program in C to sort the resumes by using **heap sort** algorithm. Determine the time required to sort the elements. Repeat the experiment for different values of n and plot a graph of the time taken versus n.(n=no of elements).

Aim: Sort given set of elements using Heap sort and to find the time required to sort the elements.

Theory: The heap sort works as its name suggests - it begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays - one to hold the heap and the other to hold the sorted elements

ALGORITHM HeapSort(n)

Stage1: (heap constuction)Constuct a heap for a given array.

Stage2: (Maximum deletion)Apply the root deletion operation n-1 times to the remaining heap.

Heap_sort(n, a[])

Step 1 : *Create the heap using bottom_up approach*

heap(n,a)

Step 2 : *Repeatedly exchange and recreate the heap using bottom up approach*

heapsort(n,a) //exchange and recreate the heap from the root node

Algorithm heap(n,a)

```
        for k □ [n/2] to 1 do
                key □ i
    v□a[k]      //consider the item to be inserted
     heap□0     //assume heap to be false initially
while !heap and (2*k)<=n do
  j□2*k
        if(j<n)
        if(a[j]>a[j+1])
         j++
    if v<=a[j]
     heap=1
    else
     a[k]=a[j]
     k=j
end while
a[k]=v
  end for
```

Algorithm heapsort(n,a)

```
        last=n
        for w□1 to n do
                exchange a[1] and a[last]
                last—
                heap(last,a)
```

end for

Complex Analysis

How efficient is the algorithm? Assume $n=2^k-1$ so that heap's tree is full i.e. largest no of nodes occur on each level

Let $h=\log_2 n$

Each key on level i f the tree will travel to the leaf level h in the worst case of the heap construction algorithm.

Since moving to next level down requires two comparisons one is to find larger child and other to determine whether the exchange is required or not. The total number of key comparisons involving key on load i will be 2(h-i)…

$$C_{worst}^{(n)} = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i)$$

$$= \sum_{i=0}^{h-1} 2(h-i)\,2^i$$

$$= 2(n - \log_2(n+1))$$

PROGRAM

```
#include<stdio.h>
#include<time.h>
#define TRUE
#define FALSE

void heapbottomup(int h[],int n)
{
    int i,heap,v,j,k;
    for(i=n/2;i>0;i--)
    {
            k=i;
            v=h[k];
```

```
                heap=FALSE;
                while(!heap && (2*k)<=n)
                {
                        j=2*k;
                        if(j<n)
                        if(h[j]<h[j+1])
                        j=j+1;
                        if(v>=h[j])
                        heap=TRUE;
                        else
                        {
                                h[k]=h[j];
                                k=j;
                        }
        }
h[k]=v;
}

void heapsort(int h[],int n)
{
     int i,temp,last=n;
     if(n<=1)
     return;
    else
   {
                heapbottomup(h,n);
                temp=h[last];
                h[last]=h[1];
                h[1]=temp;
                last--;
                heapsort(h,n-1);
     }
}

main()
{
     int h[20],n,i;
     double clk;
     clock_t starttime,endtime;
     printf("\n Enter the number of resumes\n");
     scanf("%d",&n);
     for(i=1;i<=n;i++)
      {
            h[i]=rand()%100;
            printf("The candidates ranks are: \t%d",h[i]);
       }
       starttime=clock();
       heapsort(h,n);
```

```
        endtime=clock();
        clk=(double)(endtime-starttime)/CLOCKS_PER_SEC;
        printf("\n The ranks in sorted order: \n");
        for(i=1;i<=n;i++)
        printf("\t%d",h[i]);
        printf("\nThe run time is %f\n",clk);

    }
```

Expected output:

Enter the number of resumes: 5

The candidates ranks are:

62 34 50 28 16

The ranks in sorted order:

16 28 34 50 62

The time taken to is 0.989011

2. Design and develop a program in C to implement horspool's algorithm for string matching.

Aim: To find the given pattern in the text using horspool's algorithm

Theory :

The technique of input enhancement can be applied to the problem of string matching.The problem of string matching requires finding an occurrence of a given string of ***m*** characters called the ***pattern*** in a longer string of ***n*** characters called the ***text***. The brute-force algorithm for this problem simply matches corresponding pairs of characters in the pattern and the text left to right and, if a mismatch occurs, shifts the pattern one position to the right for the next trial. Since the maximum number of such trials is $n - m + 1$ and, in the worst case, ***m*** comparisons need to be made on each of them, the worst-case efficiency of

the brute-force algorithm is in the ***O(nm)*** class. On average, however, we should expect just a few comparisons before a pattern's shift, and for random natural-language texts, the average-case efficiency indeed turns out to be in ***O(n + m)***. The worst-case efficiency of Horspool's algorithm is in ***O(nm)*** . But for random texts, it is in O ***(n)***, and, although in the same efficiency class, Horspool's algorithm is obviously faster on average than the brute-force algorithm.

Algorithm design technique

ALGORITHM *ShiftTable(**P*** [0***..m*** − 1]***)***

//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern ***P*** [0***..m*** − 1] and an alphabet of possible characters
//Output: *Table*[0***..size*** − 1] indexed by the alphabet's characters and //
filled with shift sizes computed by formula (7.1)

for ***i*** ← 0 **to** ***size*** − 1 **do** *Table*[***i***] ← ***m***

for ***j*** ← 0 **to** ***m*** − 2 **do** *Table*[***P*** [***j***]] ← ***m*** − 1 − ***j***
return *Table*

ALGORITHM *HorspoolMatching(**P*** [0***..m*** − 1]***, T*** [0***..n*** − 1]***)***

//Implements Horspool's algorithm for string
matching //Input: Pattern ***P*** [0***..m*** − 1] and text ***T***
[0***..n*** − 1]

//Output: The index of the left end of the first matching substring

or −1 if there are no matches

*ShiftTable(**P*** [0***..m*** − 1]***)*** //generate *Table* of shifts

i ← ***m*** − 1 //position of the pattern's right end

while ***i*** ≤ ***n*** − 1 **do**

$k \leftarrow 0$ //number of matched characters

while $k \leq m - 1$ **and** $P[m - 1 - k] = T[i - k]$
do $k \leftarrow k + 1$

if $k = m$

return $i - m + 1$ **else**
$i \leftarrow i + Table[T[i]]$

return −1

Complex analysis

For any random test Horspool algorithm needs $\theta(size)$. So $\theta(size)$ is the average running time. Where size denotes the size of the alphabet.

Program:

```
#include<stdio.h>
#include<string.h>
#define MAX 500
int t[MAX];
void shifttable(char p[])
{
 int i,j,m;
 m=strlen(p);
 for(i=0;i<MAX;i++)
  t[i]=m;
 for(j=0;j<m-1;j++)
  t[p[j]]=m-1-j;
}
int horspool(char src[],char p[])
```

```
{
 int i,j,k,m,n;
 n=strlen(src);
 m=strlen(p);
 printf("\nLength of text=%d",n);
 printf("\n Length of pattern=%d",m);
 i=m-1;
 while(i<n)
 {
  k=0;
  while((k<m)&&(p[m-1-k]==src[i-k]))
   k++;
  if(k==m)
   return(i-m+1);
  else
   i+=t[src[i]];
 }
}
void main()
{
 char src[100],p[100];
 int pos;
 clrscr();
 printf("Enter the text in which pattern is to be searched:\n");
 gets(src);
 printf("Enter the pattern to be searched:\n");
```

```
gets(p);
shifttable(p);
pos=horspool(src,p);
if(pos>=0)
 printf("\n The desired pattern was found starting from position %d",pos+1);
else
 printf("\n The pattern was not found in the given text");
}
```

3. Design and develop a program in C to Compute the transitive closure of a given directed graph using Warshall's algorithm. Repeat the experiment for different values of n and plot a graph of the time taken versus n (n=no of nodes).

Aim: To compute the transitive closure of a given direct graph using Warshall's algorithm.

Theory: Warshall's algorithm is used mainly for computing the transitive closure of a directed graph.

The transitive closure of a directed graph with n vertices can be defined as the n-by-n Boolean matrix $T=\{t_{ij}\}$ in which the element in the i^{th} row ($1<=i<=n$) and the j^{th}column ($1<=i<=n$) is 1 if there exists a non trivial directed path from the i^{th} vertex to the j^{th} vertex. Otherwise t_{ij} is 0.

Algorithm design technique

Step 1:*Make a copy of the adjacency matrix*

```
For i □ 0 to n-1 do
  For j □0 to n-1 do
      P[i,j] =A[I,j]
  end for
end for
```

Step 2*:* *Find the Transitive closure(path matrix)*

For k □ 0 to n-1 do

For i □0 to n-1 do

For j □ 0 to n-1 do

If (P[i,j] = 1 and (P[i,k] = 1 and P[k,j] = 1) then

P[i,j] =1

End if

End for

End for

End for

Step 3: *Finished*

Return

Complex Analysis

The space complexity of the Floyd-Warshall algorithm is $O(n^2)$.

Program:

```
#include<stdio.h>
#include<time.h>
int main()

{
   int a[10][10];
   int i,j,k,n;
  double clk;
  clock_t starttime,endtime;
  printf("Enter  number of vertices:");
  scanf("%d",&n);
  printf("Enter the adjacency matrix");
  for(i=0;i<n;i++)
       {
         for(j=0;j<n;j++)

          {

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

         }
```

```
        }
starttime=clock();
for(k=1;k<n;k++)
      {
         for(i=1;i<n;i++)
           {
               for(j=1;j<n;j++)
                 {
                    a[i][j]=a[i][j]||(a[i][k]&&a[k][j]);
                 }
           }
         }
   endtime=clock();
printf("The transitive closure of the digraph is:\n");
for(i=0;i<n;i++)
      {
        for(j=0;j<n;j++)
          {
            printf("%d",a[i][j]);
          }
        printf("\n");
      }
  clk=(double)(endtime-starttime)/CLOCKS_PER_SEC;
  printf("The runtime is %lf",clk);
```

}

Expected output:

Enter number of vertices: 4

Enter the adjacency matrix

0 1 0 0

0 0 0 1

0 0 0 0

1 0 1 0

The transitive closure of the digraph is :

1 1 1 1

1 1 1 1

0 0 0 0

1 1 1 1

4. Design and develop a program in C to find the all pair shortest path for a given graph using Floyd's algorithm. Determine the time required to find the solution. Repeat the experiment for different values of n and plot a graph of the time taken versus n (n=no of nodes).

Aim: To implement all pair shortest path problem using Floyd's algorithm.

Theory: All pair shortest path problem asks to find the distances (length of the shortest paths) from each vertex to all other vertices. It is convenient to record the lengths of the shortest paths in an n-by-n matrix d called distance matrix. the element d_{ij} in the i^{th} row and the j^{th} column of this matrix indicates the length of the shortest path from i^{th} vertex to j^{th} vertex. (1<=i,j<=n)

ALGORITHM Floyd's(G)

Step 1: *make a copy of the cost adjacency matrix*

```
for i =o to n-1 do
        for j = 0 to n-1 do
                d[i,j]=cost[i,j]
```

end for

end for

Step 2: *Find the shortest distances from all nodes to all other nodes*

for k = 0 to n-1 do

for i = 0 to n-1 do

for j = 0 to n-1 do

d[i,j]=min(d[i,j],d[i,k],d[k,j])

end for

end for

end for

Step 3 : *Finished*

Return

Complexity

The space complexity of the Floyd-Warshall algorithm is $O(n^2)$.

PROGRAM

```
#include<stdio.h>
#include<time.h>
double clk;
clock_t starttime,endtime;
int min(int a,int b)
{
    if(a<b)
```

```
        return a;
        else
        return b;
}
void floyd(int n,int W[10][10],int D[10][10])
{
        int i,j,k;
        for(i=0;i<n;i++)
        for(j=0;j<n;j++)
        D[i][j]=W[i][j];
        for(k=0;k<n;k++)
        {
                for(i=0;i<n;i++)
                {
                        for(j=0;j<n;j++)
                        {
                                D[i][j]=min(D[i][j],D[i][k]+D[k][j]);
                        }
                }
        }
}
void main()
{
        int i,j,n,D[10][10],W[10][10];
        printf("Enter no.of vertices: \n");
        scanf("%d",&n);
```

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

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

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

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

        starttime=clock();
        floyd(n,W,D);

        endtime=clock();
        clk=(double)(endtime-starttime)/CLOCKS_PER_SEC;
         printf("All pair shortest path matrix is\n");
        for(i=0;i<n;i++)

        {

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

                {

                        printf("%d\t",D[i][j]);

                }

        }

      printf("\nThe run time is %f\n",clk);
}
```

Expected output:

Enter no of vertices : 4

Enter the cost matrix

0 999 3 999

2 0 999 999

999 7 0 1

6 999 999 0

All pairs shortest path matrix is:

0 10 3 4

2 0 5 6

7 7 0 1

6 16 9 0

The time taken is 0.989011

5. Design and develop a program in C to find solution for Knapsack problem using dynamic programming. Determine the time required to find the solution. Repeat the experiment for different values of n and plot a graph of the time taken versus n (n=no of items).

Aim: To implement 0/1 knapsack problem using dynamic programming.

Theory: The knapsack problem is a problem in combinatorial optimization. It derives its name from the following maximization problem of the best choice of essentials that can fit into one bag to be carried on a trip. Given a set of items, each with a weight and a profit, determine the number items to include in a collection so that the total weight does not exceed the knapsack capasity and the total profit is as large as possible.

Let w_i be the weight of the i^{th} item, p_i be the profit accrued when the i^{th} item is carried in the knapsack, and C be the capacity of the knapsack. Let x_i be a variable the value of which is either zero or one. The variable x_i has the value one when the i^{th} item is carried in the knapsack.

Given $\{w_1, w_2, \ldots, w_n\}$ and $\{p_1, p_2, \ldots, p_n\}$, our *objective* is to maximize

$$\sum_{i=1}^{n} p_i x_i$$

subject to the constraint

$$\sum_{i=1}^{n} w_i x_i \leq C.$$

ALGORITHM Knapsack(Items, Weight, Value)

Step 1: *Obtain the matrix V which has the optimal solution for all sub problems*

```
for i = 0 to n do
    for j = 0 to m do
        if(i = 0 or j = 0)
            V[i,j] = 0
        elseif(W[i]>j)
```

V[i,j] = V[i-1,j]

else

V[i,j] = max(V[i-1,j],V[i-1,j-W[i]]+p[i])

end if

end for

end for

Step 2: *Finished*

return

PROGRAM:

```
#include<stdio.h>
#include<time.h>
int max(intx,int y)
{
  return((x>y)?x:y);
}
int knap(intn,int w[10],int value[10],intm,int v[10][10])
{
  inti,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],value[i]+v[i-1][j-w[i]]);
   }
   printf("\n The table for solving knapsack problem using dynamic programming
is:\n");
   for(i=0;i<=n;i++)
    {
     for(j=0;j<=m;j++)
       {
        printf("%d\t",v[i][j]);
```

```
        }
      printf("\n");
    }
}
main()
{
 doubleclk;
 clock_tstarttime,endtime;
 int v[10][10],n,i,j,w[10],value[10],m,result;
 printf("Enter the number of items:");
 scanf("%d",&n);
 printf("Enter the weights of %d items:/n",n);
 for(i=1;i<=n;i++)
  {
   scanf("%d",&w[i]);
  }
  printf("Enter the value of %d items:",n);
  for(i=1;i<=n;i++)
  {
   scanf("%d",&value[i]);
  }
  printf("Enter the capacity of the knapsack:");
  scanf("%d",&m);
  for(i=0;i<=n;i++)
  {
     for(j=0;j<=m;j++)
     {
        v[i][j]=0;
     }
  }
 starttime=clock();
 result=knap(n,w,value,m,v);
 endtime=clock();
 clk=(double)(endtime-starttime)/CLOCKS_PER_SEC;
 printf("Optimal solution for the knapsack problem is %d\n",v[n][m]);
 printf("%f\n",clk);
}
```

Expected output:

Enter the number of items: 4

Enter the weights of 4 items :

2

1

3

2

Enter the value of 4 items:

12

10

20

15

Enter the capacity of the knapsack:5

The table for solving knapsack problem using dynamic programming is:

0	0	0	0	0	0
0	0	12	12	12	12
0	10	12	22	22	22
0	10	12	22	30	32
0	10	15	25	30	37

Optimal solution for the knapsack problem is 37

The time taken to sort is 0.989011

6. Design and develop a program in C to find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm. Repeat the experiment for different values of n and plot a graph of the time taken versus n (n=no of nodes).

Aim: To find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.

Theory: A spanning tree of a connected graph is its connected acyclic sub graph that contains all the vertices of a graph. A minimum spanning tree of a weighted connected graph is its spanning tree of the smallest weight, where weight of the tree is defined as the sum of the weights on all its edges. Prims algorithm constructs a minimum spanning tree through a sequence of expanding sub trees. The initial sub tree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On the each iteration we expand the current tree in the greedy manner by

simply attaching to it the nearest vertex not in that tree. The algorithm stops after all the graphs vertices have been included in the tree being constructed.

Algorithm design technique

Step 1 :*Initialization*

min = 9999

visited[1] = 1

Step 2 :*starting from the 1st vertex find the edge which has the least cost and continue till n vertices*

```
While (count < n-1)                          // initialize  beforecount to 0
    min = 999
for i □ 1 to n-1 do
 for j □ 0 to n-1 do
     if(visited[i]  and !visited[j] and min > c[i][j])
           min = c[i,j]
           a = I                             //assign the source vertex to a
           b = j                             //assign the destination vertex to b
     end if
        print a□ b = c[a][b]                 //print the source and the destination
                                               vertex along with the cost
        cost +=c[a][b]          //add the cost of the edge [a , b] to   the total cost
        visited[b] = 1                       // make vertex b as visited
        count++
end while
```

Complex Analysis

In Prim's algorithm minimum spanning tree grows from an arbitrary given seed or node. Here main loop executes for (n-1) number of times. At each iteration it takes

time in $\Theta(n)$. So net complexity is $\Theta(n * n)$. Prims uses heap data structure for implementations.

Program:

```
#include<stdio.h>
#include<time.h>
int min(int a,int b)
{
    return((a<b)?a:b);
}
void prims(int n,int cost[10][10])
{
    int i,j,k,u,v,min,source;
   int sum,t[20][20],p[10],d[10],visited[10];
   min=9999;
   source=0;
   for(i=0;i<n;i++)
     for(j=0;j<n;j++)
       if(cost[i][j]!=0&&cost[i][j]<=min)
        {
min=cost[i][j];
source=i;
        }
    for(i=0;i<n;i++)
    {
        d[i]=cost[source][i];
        visited[i]=0;
        p[i]=source;
    }
visited[source]=1;
sum=0;
k=0;
for(i=1;i<n;i++)
{
    min=9999;
    u=-1;
   for(j=0;j<n;j++)
    {
       if(visited[j]==0)
       {
           if(d[j]<=min)
               {
                 min=d[j];
                 u=j;
                }
          }
    }
t[k][0]=u;
t[k][1]=p[u];
k++;
sum=sum+cost[u][p[u]];
visited[u]=1;
   for(v=0;v<n;v++)
```

```
    {
            if(visited[v]==0&&cost[u][v]<d[v])
            {
                  d[v]=cost[u][v];
                   p[v]=u;
            }
    }
}
if(sum>=9999)
    printf("\n Spanning tree does not exist");
else
   {
        printf("Spanning tree exists and minimum spanning tree is\n");
        for(i=0;i<n-1;i++)
          {
               printf("%d  %d\n",t[i][0],t[i][1]);
          }
         printf("\n The cost of spanning tree is %d ",sum);
    }
}
int main()
{
int cost[10][10],n,i,j;
double clk;
clock_t starttime,endtime;
printf("Enter the number of vertices :");
scanf("%d",&n);
printf("Enter the cost adjacency matrix :");
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&cost[i][j]);
starttime=clock();
prims(n,cost);
endtime=clock();
clk=(double)(endtime-starttime)/CLOCKS_PER_SEC;
printf("The time taken is %f\n",clk);
}
```

Expected output:

Enter the number of vertices: 6

Enter the cost adjacency matrix :

0	3	999	999	6	5
3	0	1	999	999	4
999	1	0	6	999	4
999	999	6	0	8	5

6 999 999 8 0 2

5 4 4 5 2 0

Spanning tree exists and minimum spanning tree is

0 1

1 2

1 5

4 5

3 5

The cost of spanning tree is 15

The time taken is 0.989011

7. Design and develop a program in C to find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm. Repeat the experiment for different values of n and plot a graph of the time taken versus n (n=no of nodes).

Aim: To find the minimum cost spanning tree of a given undirected graph using Kruskal's algorithm.

Theory: **Kruskal's algorithm** is an algorithm in graph theory that finds a mirnimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

Algorithm design technique

Step 1: *initialization*

Count =0

Total = 0

Flag = 0

Step 2: *find the min and check the cycle for each vertex*

```
While count!= n-1 and min != 999
    Find_min();
    Flag □check_cycle(x , y);
    If flag = 1
    Print x , y and cost[x , y]        // print cost of the min edge
        Increment
        Total += coat{x , y]              // print the total coat
        End if
        Print  □ total cost
    End while
```

AlgorithmFind_min()

```
Min □ 999
For i □ 1 to n do
    For j □ 1 to ndo
    If cost [I,j] < min
        Min □ cost [I , j]
        X = I                          //assign the source vertex to x
        Y = j                          //assign the destination vertex to y
        End if
        End for
    End for
```

Algorithm Check_cycle(x , y)

```
If parent[x] = parent[y] and parent[x] != 0
    Return 0
Else
If parent[x]= 0 and parent[y] =0
```

parent[x] = parent[y] = x

else

if parent[x] = 0

parent[x] = parent[y]

else

if parent[y] = 0

parent[y] = parent[x]

else

if parent[x] ! = parent[y]

parent[y] = parent[x]

return -1

Complexity analysis:

The evaluation of the execution time of the Kruskal's algorithm is as follows :

- $\Theta(aloga)$ to sort the edges, which is equivalent to the complexity $\Theta(alogn)$ becomes $(n-1) \leq a \leq (n-1)/2$.
- $\Theta(n)$ to initiate the n disjoint sets.
- $\Theta(2a\ \alpha\ (2a, n))$ for all the find and merge operations where α is the slow growing function.
- At worst $O(a)$ for the remaining operations

Program:

```
#include<stdio.h>

#include<time.h>

struct edge

{

     int u,v,cost;

};

typedef struct edge edge;

int find(int v,int parent[])

{
```

```
    while(parent[v]!=v)
     {
       v=parent[v];
     }
   return v;
}
void union_ij(int i,int j,int parent[])
{
     if(i<j)
       parent[j]=i;
    else
       parent[i]=j;
}
void kruskal(int n,edge e[],int m)
{
     int count,k,i,sum,u,v,j,t[10][10],p,parent[10];
     edge temp;
     count=0;
     k=0;
    sum=0;
   for(i=0;i<m;i++)
    {
       for(j=0;j<m-1;j++)
        {
            if(e[j].cost>e[j+1].cost)
              {
temp.u=e[j].u;
temp.v=e[j].v;
```

```
temp.cost=e[j].cost;
e[j].u=e[j+1].u;
e[j].v=e[j+1].v;
e[j].cost=e[j+1].cost;
e[j+1].u=temp.u;
e[j+1].cost=temp.cost;
                }
          }
    }
   for(i=0;i<n;i++)
   parent[i]=i;
   p=0;
   while(count!=n-1)
    {
u=e[p].u;
v=e[p].v;
i=find(u,parent);
j=find(v,parent);
if(i!=j)
{
t[k][0]=u;
t[k][1]=v;
k++;
count++;
sum+=e[p].cost;
union_ij(i,j,parent);
}
           p++;
```

```
    }
 if(count==n-1)
    {
printf("Spanning tree exists\n");
printf("The spanning tree is as follows:\n");
for(i=0;i<n-1;i++)
              {
                  printf("%d  %d\t",t[i][0],t[i][1]);
              }
           printf("\nThe cost of the spanning tree is %d\n",sum);
     }
else
      printf("\n spanning tree does not exist");
}
int main()
{
int n,m,a,b,i,cost;
double clk;
clock_t starttime,endtime;
edge e[20];
printf("Enter the number of vertices:");
scanf("%d",&n);
printf("Enter the number of edges:\n");
scanf("%d",&m);
printf("Enter the edge list( u  v  cost)\n");
for(i=0;i<m;i++)
{
scanf("%d%d%d",&a,&b,&cost);
```

```
e[i].u=a;
e[i].v=b;
e[i].cost=cost;
}
starttime=clock();
kruskal(n,e,m);
endtime=clock();
clk=(double)(endtime-starttime)/CLOCKS_PER_SEC;
printf("The time taken is %f\n",clk);
return 0;
  }
```

Expected output:

Enter the number of vertices : 6

Enter the number of edges: 10

Enter the edge list(u v cost)

0 1 3

1 2 1

2 3 6

3 4 8

4 0 6

0 6 5

1 6 4

2 6 4

3 6 5

4 6 2

Spanning tree exists

The spanning tree is as follows:

0 1 1 2 15 35 4 5

The cost of the spanning tree is

16

The time taken is 0.655643

8. Design and develop a program in C to find the single source shortest path for a given graph using Dijkstra's algorithm. Determine the time required to find the solution. Repeat the experiment for different values of n and plot a graph of the time taken versus n (n=no of nodes).

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

Theory: **Dijkstra's algorithm**, is a graph search algorithm that solves the single-source shortest path problem for a graph with non negative edge path costs, outputting a shortest path tree. This algorithm is often used in routing.

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols.

Algorithm design technique

Step 1: *Initialization*

For i□ 1 to n do

S[i] =0

D[i] = a[source , i]

End for

Step 2: *add source to S*

S[source] = 1

D[source]=1

count=2;

Step 3:*find the shortest distance and shortest path*

while(count<=n)

//find u and d[u] such that d[u]u is minimum and u belongs to V- S

min = 999

for j 0 to n-1 do

if s[j] = 0 and d[j] <= min

min d[j]

u = j

end if

end for

S[u]=1;

Count++:

End while// find the new vertex u and the destination which gives the shortest path and destination

for v 0 to n do

if (d[u] + a[u , v] < d[v])

d[v] =d[u] + a[u , v]

end if

end for

end for

step 4:*finished*

return

Complex analysis:

Time Complexity of Dijkstra's Algorithm is $O(v * v)$ but with min-priority queue it drops down to $O(V + E\ logV)$.

However, if we have to find the shortest path between all pairs of vertices, both of the above methods would be expensive in terms of time. Discussed below is another algorithm designed for this case.

PROGRAM:

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

#define MAX 10

int choose(int dist[],int s[],int n)
{
  int j=1,min=100,w;
  for(w=1;w<=n;w++)
   if((dist[w]<min) && (s[w]==0))
     {
       min=dist[w];
       j=w;
     }
   return j;
}

void spath(int v,int cost[][MAX],int dist[],int n)
 {
    int w,u,i,num,s[MAX];
    for(i=1;i<=n;i++)
     {
        s[i]=0;
          dist[i]=cost[v][i];
        }
     s[v]=0;
     dist[i]=999;
     for(num=2;num<=n;num++)
      {
       u=choose(dist,s,n);
       s[u]=1;
      for(w=1;w<=n;w++)
      if((dist[u]+cost[u][w])<dist[w] && !s[w])
          dist[w]=dist[u]+cost[u][w];
       }
 }
```

```
void main()
{
  int i,j,cost[MAX][MAX],dist[MAX],n,v;
  double clk;
  clock_t starttime,endtime;

  printf("\nEnter number of vertices:");
  scanf("%d",&n);
  printf("\nEnter the cost of adjacency matrix\n");
 for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
      scanf("%d",&cost[i][j]);
 printf("\nEnter the source vertex");
 scanf("%d",&v);

 starttime=clock();
 spath(v,cost,dist,n);
 endtime=clock();
 printf("\nShortest  distance\n");
 for(i=1;i<=n;i++)
     printf("\n%d to %d = %d",v,i,dist[i]);
     clk=(double)(endtime-starttime)/CLOCKS_PER_SEC;
     printf("The time taken is %f\n",clk);

}
```

Expected output:

Enter the number of vertices : 5

Enter the cost of adjacency matrix

999	4	2	999	8
999	999	999	4	5
999	999	999	1	999
999	999	999	999	3
999	999	999	999	999

Enter the source vertex : 1

Shortest distance

1 to 1 = 0

1 to 2 = 4

1 to 3 = 2

1 to 4 = 3

1 to 5 = 6

The time taken is 0.456643

9. Design and develop a program in C to implement n/Queens problem by using backtracking method.

Aim: Place the n-queens on the chess board using the backtracking methodology

Theory: Backtracking is a more intelligent variation.The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. If a partially constructed solution can be de-veloped further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legiti-mate option for the next component, no alternatives for *any* remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

The ***n-queens problem***. The problem is to place ***n*** queens on an ***n*** × ***n*** chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. For ***n*** = 1**,** the problem has a trivial solution, and it is easy to see that there is no solution for ***n*** = 2 and ***n*** = 3.

Algorithm design technique

Step 1: Start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1.

Step 2:Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4).

Step 3 : Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem.

Complexity Analysis:

Backtracking is a recursive method which starts a queen at an edge and, ideally, saves the possible attack positions. Then another queen is placed at a safe position…repeat. However, if it is found that N number of queens cannot be placed on that board, it will backtrack and try another safe position. This is over 100 times as fast as brute force and has a time complexity of O(2^n). Some others are even faster. The systematic and greedy search(1000 x faster than the previous) and systematic random permutations(1000 x faster again) both have a O(n^k) or polynomial time complexity. Some algorithms are *sub-linear* O(n^(1.5)), and an Optimized smart random permutations algorithm(currently in development) is estimated to be linear O(n).
PROGRAM:

```
//backtracking

#include<stdio.h>

#include<math.h>

#define FALSE 0

#define TRUE 1


int x[20];

int place(int k,int i)

{

        int j;

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

        {

                if((x[j]==i)||(abs(x[j]-i)==abs(j-k)))
```

```
                return FALSE;
        }
return TRUE;
}

void nqueens(int k,int n)
{
        int i,a;
        for(i=1;i<=n;i++)
        {
                if(place(k,i))
                {
                        x[k]=i;
                        if(k==n)
                        {
                                for(a=1;a<=n;a++)
                                        printf("%d\t",x[a]);
                                printf("\n");
                        }
                        else
                        nqueens(k+1,n);
                }
        }
}
void main()
{
```

```
    int n;
    printf("\nEnter the number of queens:");
    scanf("%d",&n);
    printf("\n The solution to N Queens problem is: \n");
    nqueens(1,n);
}
```

10. Design and develop a program in C to find the subset of a given set S={S1,S2,S3,…..Sn} of n positive integers whose sum is equal to a positive integer d and to display a suitable message if the given problem instance does not have the solution.

Aim: To find the subset of a given set of n positive integers whose sum is equal to a positive integer d

Theory: Find a subset of a given set $A = \{a_1, \ldots, a_n\}$ of n positive integers whose sum is equal to a given positive integer d. For example, for $A = \{1, 2, 5, 6, 8\}$ and $d = 9$, there are two solutions: {1, 2, 6} and {1, 8}. It is convenient to sort the set's elements in increasing order. So, we will assume that

$$a_1 < a_2 < \cdots < a_n.$$

The root of the tree represents the starting point, with no decisions about the given elements made as yet. Its left and right children represent, respectively, inclusion and exclusion of a_1 in a set being sought. Similarly, going to the left from a node of the first level corresponds to inclusion of a_2 while going to the right corresponds to its exclusion, and so on. Thus, a path from the root to a node on the ith level of the tree indicates which of the first i numbers have been included in the subsets represented by that node.

We record the value of s, the sum of these numbers, in the node. If s is equal to d, we have a solution to the problem. We can either report this result and stop or, if all the solutions need to be found, continue by backtracking to the node's parent. If s is not equal to d, we can terminate the node as nonpromising if either of the following two inequalities holds:

$$s + a_{i+1} > d \quad \text{(the sum } s \text{ is too large),}$$

$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum } s \text{ is too small).}$$

j
$=i+1$

Algorithm design technique

Step 1: The root of the tree represents the starting point, with no decisions about the given elements made as yet. Its left and right children represent, respectively, inclusion and exclusion of a_1 in a set being sought.

Step 2: going to the left from a node of the first level corresponds to inclusion of a_2 while going to the right corresponds to its exclusion, and so on.

Step 3: Thus, a path from the root to a node on the ith level of the tree indicates which of the first i numbers have been included in the subsets represented by that node.

step 4: We record the value of s, the sum of these numbers, in the node. If s is equal to d, we have a solution to the problem. We can either report this result and stop or, if all the solutions need to be found, continue by backtracking to the node's parent. If s is not equal to d, we can terminate the node as nonpromising if either of the following two inequalities holds:

$$s + a_{i+1} > d \text{ (the sum } s \text{ is too large)},$$

$$s + \sum_{j=i+1}^{n} a_j < d \text{ (the sum } s \text{ is too small)}.$$

Complexity Analysis:

The worst case "brute force" solution for the N-queens puzzle has an O(n^n) time complexity. This means it will look through every position on an NxN board, N times, for N queens. It is by far the slowest and most impractical method. If you refactor and prevent it from checking queens occupying the same row as each other, it will still be brute force, but the possible board states drop from 16,777,216 to a little over 40,000 and has a time complexity of O(n!).

PROGRAM:

```
#include<stdio.h>
  int count,w[10],d,x[10];
  void subset(int cs,int k,int r)
{
   int i;
   x[k]=1;
   if((cs+w[k])>d)
    {
      printf("\n Subset solution = %d\n",++count);
      for(i=0; i<=k; i++)
      {
        if(x[i]==1)
        printf("%d\n",w[i]);
      }
    }
  else if(cs+w[k]+w[k+1] <=d)
       subset(cs+w[k],k+1,r-w[k]);
   if((cs+r-w[k]>=d)&&(cs+w[k+1])<=d)
   {
     x[k]=0;
     subset(cs,k+1,r-w[k]);
   }
}
```

```
int main()
{
    int sum=0,i,n;
    printf("enter no of elements\n");
    scanf("%d",&n);
    printf("Enter the elements in ascending order\n");
    for(i=0; i<n; i++)
    scanf("%d",&w[i]);
    printf("Enter the required sum\n");
    scanf("%d",&d);
    for(i=0; i<n; i++)
    sum +=w[i];
 if(sum < d)
    {
        printf("no solution exits\n");
              }
    printf("The solution is\n");
    count =0;
    subset(0,0,sum);
    return 0;
    getch();
}
```