Design & Analysis of Algorithms

Index

1.Bubble Sort

Write a C program to sort array of n elements in non-decreasing order using bubble sort technique.

Approach : Brute-force
Efficiency : $\Theta(n^2)$

Algorithm:

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

Program:

```
#include <stdio.h>
#include <stdlib.h>
void bubbleSort(int a[ ],int n)
{
   int i,j,temp;
   for(i=0;i<n-1;i++)     //Number of passes
      for(j=0; j<n-i-1; j++)    //For each pass one largest element bubbled to its position
         if(a[j]>a[j+1])
         {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
         }
}
int main( )
{
   int n,i;
   printf("Enter the number of elements :  ");
   scanf("%d",&n);
   int a[n];
   printf("Enter %d elements : \n",n);
   for(i=0;i<n;i++)
      scanf("%d",&a[i]);
   printf("\nElements before sorting :\n");
   for(i=0;i<n;i++)
      printf("\t%d",a[i]);
   bubbleSort(a,n);
   printf("\nElements after sorting :\n");
   for(i=0;i<n;i++)
      printf("\t%d",a[i]);
   return 0;
}
```

Output:

```
Enter the number of elements :  5
Enter 5 elements :
8 6 1 3 9

Elements before sorting :
        8       6       1       3       9
Elements after sorting :
        1       3       6       8       9
```

```
Enter the number of elements :  10
Enter 10 elements :
78 0 45 5 3 21 98 46 34 2

Elements before sorting :
        78      0       45      5       3       21      98      46      34      2
Elements after sorting :
        0       2       3       5       21      34      45      46      78      98
```

2.Selection Sort

Write a C program to sort array of n elements in non-decreasing order using selection sort technique.

Approach : Brute-force
Efficiency : $\Theta(n^2)$

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 nondecreasing order
for *i* ←0 **to** *n* − 2 **do**
    min←*i*
    for *j* ←*i* + 1 **to** *n* − 1 **do**
        if *A*[*j*]<*A*[*min*]
            min←*j*
    swap *A*[*i*] and *A*[*min*]

Program:

```
#include <stdio.h>
#include <stdlib.h>
void selectionSort(int a[ ],int n)
{
   int i,j,min,temp;
   for(i=0; i<n-1; i++)
   {
      min=i;
      for(j=i+1; j<n; j++)      //Find smallest element in the array
         if(a[j]<a[min])
            min=j;
      temp=a[min];      //Place the min element in ith position
      a[min]=a[i];
      a[i]=temp;
   }
}
int main( )
{
   int i,n;
   printf("Enter the value of n:");
   scanf("%d",&n);
   int a[n];
   printf("Enter %d elements : \n",n);
   for(i=0; i<n; i++)
      scanf("%d",&a[i]);
   printf("Before Sorting:\n");
   for(i=0; i<n; i++)
      printf("%d\t",a[i]);
   selectionSort(a,n);
   printf("\nAfter Sorting:\n");
```

```
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);
    return 0;
}
```

Output:

```
Enter the value of n:5
Enter 5 elements :
5 4 3 2 1
Before Sorting:
5       4       3       2       1
After Sorting:
1       2       3       4       5
```

```
Enter the value of n:10
Enter 10 elements :
85 12 9 44 11 2 46 12 3 7
Before Sorting:
85      12      9       44      11      2       46      12      3       7
After Sorting:
2       3       7       9       11      12      12      44      46      85
```

3.Brute force string matching

Write a C program to search the pattern in the given text using brute force string matching algorithm. Assume matching done by ignoring case-sensitivity of alphabets.

Algorithm:

BruteForceStringMatch$(T[0..n-1], P[0..m-1])$
//Implements brute-force string matching
//Input: An array $T[0..n-1]$ of n characters representing a text and
// an array $P[0..m-1]$ of m characters representing a pattern
//Output: The index of the first character in the text that starts a matching substring or -1 if the search is unsuccessful
for $i \leftarrow 0$ **to** $n-m$ **do**
    $j \leftarrow 0$
    while $j < m$ **and** $P[j] = T[i+j]$ **do**
        $j \leftarrow j+1$
    if $j = m$
        return i
return -1

Approach : Brute-force
Best-case Efficiency : Θ(m) – Pattern found at position-1
Worst case Efficiency : Θ(mn) – Pattern at last or not found

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
int StringMatch(char text[ ],char pattern[ ])
{
    int i,j,m,n;
    n=strlen(text);
    m=strlen(pattern);
    for(i=0; i<=n-m ; i++)    //Align pattern under every characters of text
    {
        j=0;
        while(j<m && tolower(text[i+j])==tolower(pattern[j]))   //Compare pattern with text at ith position
        {
```

```
            j++;
        }
        if(j==m)     //If all characters of pattern are matched with the text
        {
            return i+1;
        }
    }
    return -1;
}

int main( )
{
    char text[100],pattern[25];
    int pos;
    printf("Enter the text : \n");
    gets(text);
    printf("Enter the pattern :\n");
    gets(pattern);
    pos=StringMatch(text,pattern);
    if(pos!=-1)
        printf("The pattern %s is found at position %d\n",pattern,pos);
    else
        printf("Pattern Not Found!!");
    return 0;
}
```

Output:

```
Enter the text :
Design and Analysis of Algorithms
Enter the pattern :
analysis
The pattern analysis is found at position 12
```

```
Enter the text:
ABC
Enter the pattern:
def
Pattern not found!!
```

4.Binary search

Write a C program to search for a key element in the list of n element in non-decreasing order using binary search technique.

Algorithm:

BinarySearch(*A*[0..*n* − 1],key)
//Search for the key element in the array sorted in non-decreasing order.
//Input: An array *A*[0..*n* − 1] sorted in non-decreasing order, key element to be searched
//Output: Position of key elements in array A
low ← *0; high*←n-1
while *low* <=*high* **do**
    mid ← (low+high)/2
    if A[mid]=key
        return *mid*
    if key < A[mid]
        high ← mid -1
    else
        low ← mid + 1
return *-1*

Approach : Decrease and conquer
Best-case Efficiency : Ω(1) – Mid element is the key
Worst-case Efficiency : Θ(log n) – Key not found

Program:

```
#include <stdio.h>
#include <stdlib.h>
int binarySearch(int a[ ],int n,int key)
{
   int low=0,mid,high=n-1;
   while(low<=high)
   {
      mid=(low+high)/2;        //Find mid element
      if(a[mid]==key)
         return mid;
      if(key<a[mid])
         high=mid-1;          //Search in first half
      else
         low=mid+1;            //Search in second half
   }
   return -1;
}
int main( )
{
   int n,key,i;
   printf("Enter the number of elements : ");
   scanf("%d",&n);
   int a[n];
   printf("Enter %d elements in non-decreasing order : \n",n);    for(i=0; i<n; i++)
   scanf("%d",&a[i]);
   printf("Enter the key element \n");
   scanf("%d",&key);
   i=binarySearch(a,n,key);
   if(i==-1)
      printf("Key element is not found!");
   else
      printf("%d is present at position %d ",key,i+1);
   return 0;
}
```

Output:

```
Enter the number of elements : 5
Enter 5 elements in non-decreasing order :
1 3 5 7 9
Enter the key element
5
5 is present at position 3
```

```
Enter the number of elements : 15
Enter 15 elements in non-decreasing order :
9 30 45 121 127 129 217 316 319 411 421 430 450 512 515
Enter the key element
512
512 is present at position 14
```

5.Merge sort

Write a C program to sort array of n elements in non-decreasing order using merge sort technique.

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)

Approach : Divide and conquer
Best-case Efficiency : Θ(nlog(n)) – Subarrays are sorted
Worst-case Efficiency : Θ(nlog n) – Alternate elements subarray are sorted

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]

Program:

```
#include <stdio.h>
#include <stdlib.h>
void Merge(int b[ ],int c[ ],int a[ ],int p,int q)
{
  int i=0,j=0,k=0;
  while(i<p && j<q)    //whether any subarray b or c is exhausted
  {
    if(b[i]<=c[j])
    {
      a[k]=b[i];
      i++;
    }
    else
    {
      a[k]=c[j];
      j++;
    }
    k++;
  }
  if(i==p)
  {
    while(j<q && k<(p+q))
    {
      a[k]=c[j];
      j++;
```

```
        k++;
      }
    }
    else
    {
      while(i<p && k<(p+q))
      {
        a[k]=b[i];
        k++;
        i++;
      }
    }
}

void mergeSort(int n,int a[ ])
{
    if(n>1)
    {
      int i,j,len;
      len=n/2;
      int b[len],c[n-len];    //divide array into two subarrays of equal parts
      for(i=0,j=0; i<len && j<len; i++,j++)
      {
        b[j]=a[i];
      }
      for(i=len,j=0; i<n && j<n-len; i++,j++)
      {
        c[j]=a[i];
      }
      mergeSort(len, b);
      mergeSort(n-len, c);
      Merge(b,c,a,len,n-len);
    }
}

int main( )
{
    int i,n;
    printf("Enter the number of elements : ");
    scanf("%d",&n);
    int a[n];
    printf("Enter %d elements :\n",n);
    for(i=0; i<n;i++)
      scanf("%d",&a[i]);
    printf("Array before sorting:\n");
    for(i=0;i<n;i++)
      printf("\t%d",a[i]);
    mergeSort(n,a);
    printf("\nArray after sorting:\n");
    for(i=0;i<n;i++)
      printf("\t%d",a[i]);
    return 0;
}
```

Output:

```
Enter the number of elements : 5
Enter 5 elements :
8 78
44
2 1
Array before sorting:
        8       78      44      2       1
Array after sorting:
        1       2       8       44      78
```

```
Enter the number of elements : 10
Enter 10 elements :
1 2 5 6 3 2 1 3 6 5
Array before sorting:
        1       2       5       6       3       2       1       3       6       5
Array after sorting:
        1       1       2       2       3       3       5       5       6       6
```

6.Quick sort

Write a C program to sort array of n elements in non-decreasing order using quick sort technique.

Algorithm:

Quicksort$(A[l..r])$
//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, 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$(A[s+1..r])$

HoarePartition$(A[l..r])$
//Partitions a subarray by Hoare's algorithm, using the first element as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right indices l and r $(l<r)$
//Output: Partition of $A[l..r]$, with the split position returned as this function's value
$p \leftarrow A[l]$
$i \leftarrow l$;
$j \leftarrow r + 1$
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

Approach : Divide and conquer
Best-case Efficiency : $\Theta(n\log(n))$ – Pivot gets middle position
Worst-case Efficiency : $O(n^2)$ – Already sorted array

Program:

```
#include <stdio.h>
#include <stdlib.h>
int HoarePartition(int a[],int l,int r)
{
   int p,i,j,temp;
   p=a[l];
   i=l;
   j=r+1;
```

```
    do    //till i and j crosses each other
    {
      do
      {
        i++;
      }while(a[i]<p && i<=r);
      do
      {
        j--;
      }while(a[j]>p);

      if(i<=r)
      {
        temp=a[i];
        a[i]=a[j];
        a[j]=temp;
      }
    }while(i<j);
    if(i<=r)      //Undo unnecessary swap
    {
      temp=a[i];
      a[i]=a[j];
      a[j]=temp;
    }
    temp=a[l];   //Swap pivot and jth element to put pivot in right position
    a[l]=a[j];
    a[j]=temp;
    return j;
}

void QuickSort(int a[],int l,int r)
{
    if(l<r)
    {
      int s=HoarePartition(a,l,r);  //Partition the array into sub-array based on pivot element
      QuickSort(a,l,s-1);    //Sort first sub-array
      QuickSort(a,s+1,r);    //Sort second sub-array
    }
}
int main( )
{
    int n,i;
    printf("Enter the number of elements : ");
    scanf("%d",&n);
    int a[n];
    printf("Enter %d elements:\n",n);
    for(i=0;i<n;i++)
      scanf("%d",&a[i]);
    printf("\nArray before sorting:\n");
    for(i=0; i<n;i++)
      printf("\t%d",a[i]);
    QuickSort(a,0,n-1);   //Quicksort(a[l,..,r])
    printf("\nArray after sorting:\n");
    for(i=0;i<n;i++)
```

```
        printf("\t%d",a[i]);
    return 0;
}
```

Output:

```
Enter the number of elements : 5
Enter 5 elements:
2 7 9 2 1

Array before sorting:
        2       7       9       2       1
Array after sorting:
        1       2       2       7       9
```

```
Enter the number of elements : 10
Enter 10 elements:
7 19 17 15 11 25 29 22 18 5

Array before sorting:
        7       19      17      15      11      25      29      22      18      5
Array after sorting:
        5       7       11      15      17      18      19      22      25      29
```

7.Insertion sort

Write a C program to sort array of n elements in non-decreasing order using insertion sort technique.

Algorithm:

InsertionSort(*A*[0..*n* − 1])
//Sorts a given array by insertion sort
//Input: An array *A*[0..*n* − 1] of *n* orderable elements
//Output: Array *A*[0..*n* − 1] sorted in nondecreasing order
for *i* ←1 **to** *n* − 1 **do**
    v ←*A*[*i*]
    j ←*i* − 1
    while *j* ≥ 0 **and** *A*[*j*]> *v* **do**
        A[*j* + 1]←*A*[*j*]
        j ←*j* − 1
    A[*j* + 1]←*v*

Approach : Decrease and conquer
Best-case Efficiency : Θ(n) – Sorted array
Worst-case Efficiency : Θ(n^2) – strictly Decresing array

Program:

```
#include <stdio.h>
#include <stdlib.h>
void InsertionSort(int a[ ],int n)
{
    int i,j,v;
    for(i=1; i<n; i++)     //Adding one element at a time to its original position
    {
        v=a[i];
        j=i-1;
        while(j>=0 a[j]>v)
        {
            a[j+1]=a[j];
            j--;
        }
        a[j+1]=v;
    }
}
```

```
int main( )
{
   int i,n;
   printf("Enter the value of n :");
   scanf("%d",&n);
   int a[n];
   printf("Enter %d element:\n",n);
   for(i=0; i<n; i++)
      scanf("%d",&a[i]);
   printf("Array before sorting:\n");
   for(i=0; i<n; i++)
      printf("\t%d",a[i]);
   InsertionSort(a,n);
   printf("\nArray after sorting:\n");
   for(i=0; i<n; i++)
      printf("\t%d",a[i]);
   return 0;
}
```

Output:

```
Enter the value of n :5
Enter 5 element:
11 33 77 44 55
Array before sorting:
        11      33      77      44      55
Array after sorting:
        11      33      44      55      77
```

```
Enter the value of n :10
Enter 10 element:
8 3 0 1 9 5 7 4 6 2
Array before sorting:
        8       3       0       1       9       5       7       4       6       2
Array after sorting:
        0       1       2       3       4       5       6       7       8       9
```

8.Depth First Search (DFS) graph traversal

Write a C program to traverse all vertices of graph (directed or undirected) using DFS graph traversal technique.

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 are first encountered by the
//DFS traversal mark each vertex in *V* with 0 as a mark of being "unvisited"
count ←0
for each vertex *v* in *V* **do**
    if *v* is marked with 0
        dfs(v)

Approach : Exhaustive search
Efficiency : $\Theta(|V|^2)$ – Adjacency matrix representation
$\Theta(|V|+|E|)$ – Adjacency list representation

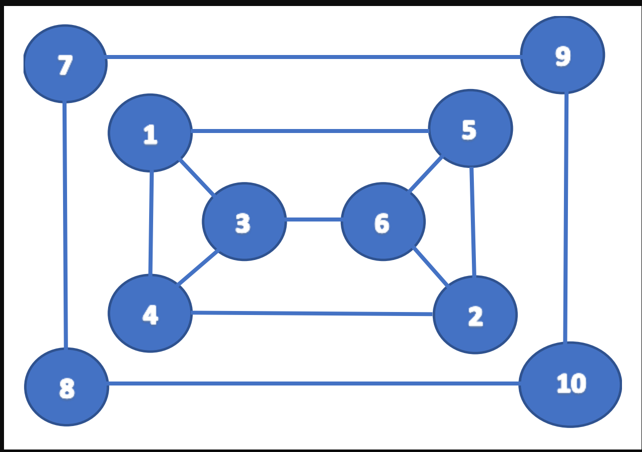
dfs(*v*)
//visits recursively all the unvisited vertices connected to vertex *v* by a path and numbers them in the order they are
//encountered via global variable *count*
count ←*count* + 1; mark *v* with *count*
for each vertex *w* in *V* adjacent to *v* **do**
    if *w* is marked with 0
        dfs(w)

Program:

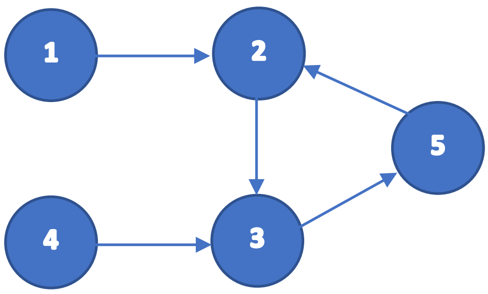
```
#include <stdio.h>
#include <stdlib.h>
int count=0;
int v,e;
int visited[20], mat[20][20];
void dfs(int w)
{
   int j;
   count++;
   visited[w]=count;      //Mark vertex w as visited
   printf("%d(%d)\t",w,visited[w]);
   for(j=1; j<=v; j++)
      if(mat[w][j]==1 && visited[j]==0)
         dfs(j);
}
void DFS( )     //To ensure all the vertices are visited
{
   int i;
   for(i=1; i<=v; i++)
   {
      if(visited[i]==0)
         dfs(i);
   }
}
int main( )
{
   int i;
   int v1,v2,ch;
   printf("Select the type of Graph:\n\t> 1.Directed Graph\n\t> 2.Undirected Graph\n");
   scanf("%d",&ch);
   if(ch!=1 && ch!=2)
   {
      printf("Invalid option !!");
      return 0;
   }
   printf("Enter the number of vertices : ");
   scanf("%d",&v);
   printf("Enter the number of edges : ");
   scanf("%d",&e);
   printf("Enter %d edges one by one :\n",e);
   for(i=1; i<=e; i++)
   {
      printf("Edge-%d : ",i);
      scanf("%d%d",&v1,&v2);
      if(ch==1)
         mat[v1][v2]=1;       //Directed graph
      else
         mat[v1][v2]=mat[v2][v1]=1;     //Undirected graph
   }
   printf("\nOrder of vertices processed:\n");
   DFS( );
   return 0;
}
```

Output:

```
Select the type of Graph:
        > 1.Directed Graph
        > 2.Undirected Graph
2
Enter the number of vertices : 10
Enter the number of edges : 12
Enter 12 edges one by one :
Edge-1 : 1 5
Edge-2 : 1 3
Edge-3 : 1 4
Edge-4 : 4 3
Edge-5 : 3 6
Edge-6 : 6 5
Edge-7 : 6 2
Edge-8 : 5 2
Edge-9 : 7 9
Edge-10 : 9 10
Edge-11 : 10 8
Edge-12 : 8 7

Order of vertices processed:
1(1)    3(2)    4(3)    6(4)    2(5)    5(6)    7(7)    8(8)    10(9)    9(10)
```




```
Select the type of Graph:
        > 1.Directed Graph
        > 2.Undirected Graph
1
Enter the number of vertices : 5
Enter the number of edges : 5
Enter 5 edges one by one :
Edge-1 : 1 2
Edge-2 : 2 3
Edge-3 : 4 3
Edge-4 : 3 5
Edge-5 : 5 2

Order of vertices processed:
1(1)    2(2)    3(3)    5(4)    4(5)
```




```
Select the type of Graph:
        > 1.Directed Graph
        > 2.Undirected Graph
3
Invalid option !!
```

9.Breadth First Search (BFS) graph traversal

Write a C program to traverse all vertices of graph (directed or undirected) using BFS graph traversal technique.

Approach : Exhaustive search
Efficiency : $\Theta(|V|^2)$ – Adjacency matrix representation
$\Theta(|V|+|E|)$ – Adjacency list representation

Algorithm:

BFS(*G*)
//Implements a breadth-first search traversal of a given graph
//Input: Graph *G* = (*V*, *E*)
//Output: Graph *G* whose vertices marked with consecutive integers in the order they are visited by the BFS traversal
//mark each vertex in *V* with 0 as a mark of being "unvisited"
count ←0
for each vertex *v* in *V* **do**
  if *v* is marked with 0
    bfs(*v*)

bfs(*v*)
//visits all the unvisited vertices connected to vertex *v* by a path and numbers them in the order they are visited
//via global variable *count*
count ←*count* + 1
mark *v* with *count* and initialize a queue with *v*
while the queue is not empty **do**
    for each vertex *w* in *V* adjacent to the front vertex **do**
        if *w* is marked with 0
            count ←*count* + 1
            mark *w* with *count*
            add *w* to the queue
    remove the front vertex from the queue

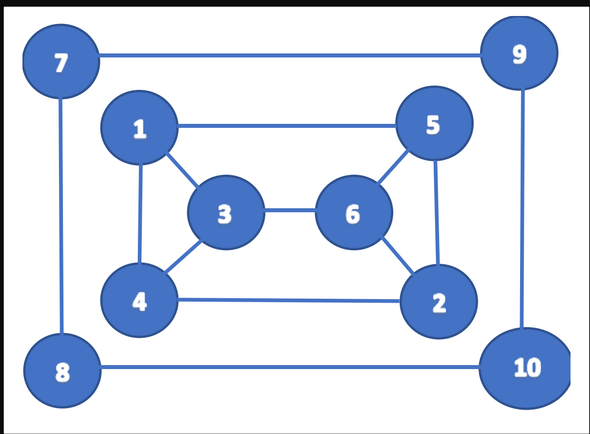
Program:

```
#include <stdio.h>
#include <stdlib.h>
int count=0;
int v,e;
int visited[20];
int mat[20][20];
int queue[20],front=0,rear=-1;

void bfs(int w)
{
  int j;
  count++;
  visited[w]=count;    //Mark vertex w as visited
  rear++;
  queue[rear]=w;   //Add w to the queue
  while(front<=rear)
  {
    printf("%d(%d)\t",queue[front],visited[queue[front]]);
    for(j=1; j<=v; j++)
    {
      if(visited[j]==0 && mat[queue[front]][j]==1)  //Add all adjacent vertices of queue[front] to queue
      {
        count++;
        visited[j]=count;
        rear++;
        queue[rear]=j;
      }
    }
    front++;  //Remove front vertex from queue after processing it
  }
}

void BFS()
{
  int i;
  for(i=1; i<=v; i++)   //Ensures all the vertices are visited
  {
    if(visited[i]==0)
      bfs(i);
  }
}
```

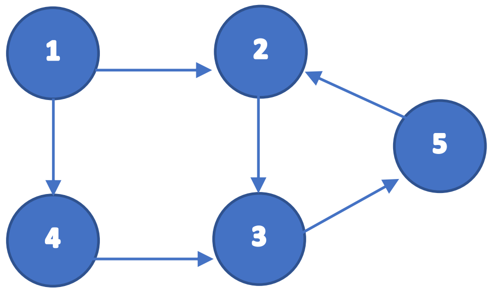
```
int main()
{
   int i;
   int v1,v2,ch;
   printf("Select the type of Graph:\n\t> 1.Directed Graph\n\t> 2.Undirected Graph\n");
   scanf("%d",&ch);
   if(ch!=1 && ch!=2)
   {
      printf("Invalid option !!");
      return 0;
   }
   printf("Enter the number of vertices : ");
   scanf("%d",&v);
   printf("Enter the number of edges : ");
   scanf("%d",&e);
   printf("Enter %d edges one by one : \n",e);
   for(i=1; i<=e; i++)
   {
      printf("Edge-%d : ",i);
      scanf("%d%d",&v1,&v2);
      if(ch==1)
         mat[v1][v2]=1;  //Directed graph
      else
         mat[v1][v2]=mat[v2][v1]=1;  //Undirected graph
   }
   printf("\nOrder of vertices processed:\n");
   BFS();
   return 0;
}
```

Output:

```
Select the type of Graph:
        > 1.Directed Graph
        > 2.Undirected Graph
2
Enter the number of vertices : 10
Enter the number of edges : 12
Enter 12 edges one by one :
Edge-1 : 1 3
Edge-2 : 1 4
Edge-3 : 1 5
Edge-4 : 4 3
Edge-5 : 3 6
Edge-6 : 6 5
Edge-7 : 6 2
Edge-8 : 5 2
Edge-9 : 7 9
Edge-10 : 9 10
Edge-11 : 10 8
Edge-12 : 8 7

Order of vertices processed:
1(1)    3(2)    4(3)    5(4)    6(5)    2(6)    7(7)    8(8)    9(9)    10(10)
```



```
Select the type of Graph:
        > 1.Directed Graph
        > 2.Undirected Graph
1
Enter the number of vertices : 5
Enter the number of edges : 6
Enter 6 edges one by one :
Edge-1 : 1 2
Edge-2 : 1 4
Edge-3 : 2 3
Edge-4 : 3 5
Edge-5 : 5 2
Edge-6 : 4 3

Order of vertices processed:
1(1)    2(2)    4(3)    3(4)    5(5)
```



10.Topological sorting

Write a C program to perform topological sorting on a directed graph.

Algorithm:

TopologicalSort*(G)*
//Perform topological sort on a directed graph
//Input: Directed Graph *G* = (*V*, *E*)
//Output: Graph *G* whose vertices are display in topological sorted order
p ←0
while not all vertices of V are visited **do**
    for each vertex *v* in V
        if *v* is marked with 0 and v has indegree 0
            count ←*count* + 1
            mark v with count
            p ← p+1
            sorted[p] ← v
            remove all outgoing edges from v

Approach : Decrease and conquer
Efficiency : $\Theta(|V|^2)$

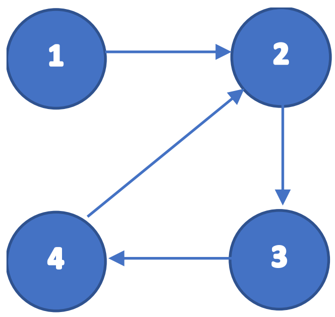
Program:

```
#include <stdio.h>
#include <stdlib.h>
int v,e,visited[20],mat[20][20],p=0;
int sorted[20],count=0,c=0,limit=0;
void TopologicalSort()
{
   int i,j,n;
   n=v;
   while(n!=0)  //Whether all vertices are removed
   {
      for(i=1; i<=v; i++)
      {
         if(visited[i]==0)
         {
         c++;    //To check whether G has cycle by counting the number of times this statement executes.
                 //The above statement can execute at most n(n+1)/2 times because for each time one vertex will be
                // removed, if the graph is not cyclic.  If the above statement executes more than n(n+1)/2 times then
                // the graph is cyclic and topological sort cant be performed.
         for(j=1; j<=v; j++)  //check for vertex having indegree 0
            if(mat[j][i]==1)
               break;
         if(j==v+1)       //vertex do not have indegree
```

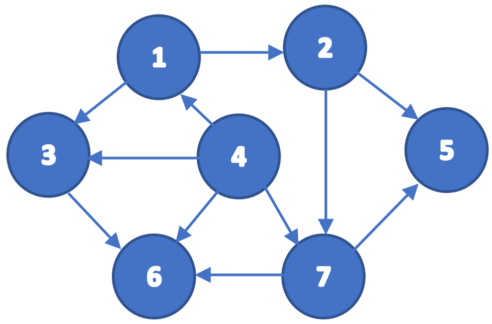
```
            {
               sorted[p++]=i;
               count++;
               visited[i]=count;
               n--;
               int k;
               for(k=1; k<=v; k++)       //Remove all outgoing edges of i
                  mat[i][k]=0;
               break;
            }
         }
         if(c > limit)  //There is cycle
            return;
      }
   }
}
int main()
{
   int i,v1,v2;
   printf("Enter the number of vertices : ");
   scanf("%d",&v);
   printf("Enter the number of edges : ");
   scanf("%d",&e);
   printf("Enter %d edges in the following format:\n",e);
   printf("Format : STARTING_VERTEX <space> TERMINAL_VERTEX\n");
   for(i=1; i<=e; i++)
   {
      printf("Edge-%d : ",i);
      scanf("%d%d",&v1,&v2);
      mat[v1][v2]=1;
   }
    limit=(v*(v+1))/2;
   TopologicalSort();
   if(c>limit)
      printf("Graph is cyclic. \nTopological sort cannot be performed.!!");
   else
   {
      printf("\nTopologically sorted order:\n");
      for(i=0; i<v; i++)
      printf("\t%d",sorted[i]);
   }
   return 0;
}
```

Output:

```
Enter the number of vertices : 4
Enter the number of edges : 4
Enter 4 edges in the following format:
Format : STARTING_VERTEX <space> TERMINAL_VERTEX
Edge-1 : 1 2
Edge-2 : 2 3
Edge-3 : 3 4
Edge-4 : 4 2
Graph is cyclic.
Topological sort cannot be performed.!!
```



```
Enter the number of vertices : 7
Enter the number of edges : 11
Enter 11 edges in the following format:
Format : STARTING_VERTEX <space> TERMINAL_VERTEX
Edge-1 : 1 3
Edge-2 : 3 6
Edge-3 : 7 6
Edge-4 : 7 5
Edge-5 : 2 5
Edge-6 : 1 2
Edge-7 : 2 7
Edge-8 : 4 1
Edge-9 : 4 3
Edge-10 : 4 6
Edge-11 : 4 7

Topologically sorted order:
        4       1       2       3       7       5       6
```



11.Heap Sort

Write a C program to sort array of n elements in non-decreasing order using heap sort technique.

Algorithm:

Heapsort*(H[1..n])*   *//Maximum key deletion*
//Sorting the heap using Maximum key deletion method
//Input : A heap H[1..n]
//Output : Sorted array of elements
Heapify(H[1..n])
for i←n down to 2 **do**
    swap (H[1],H[i])
    j ← j-1
    Heapify(H[1… j])

Approach : Transform and conquer
Worst & average case efficiency : Θ(nlog(n))

Heapify(*H*[1..*n*])   *//Heap Bottom up*
//Constructs a heap from elements of a given array by the bottom-up algorithm
//Input: An array *H*[1..*n*] of orderable items
//Output: A heap *H*[1..*n*]
for *i* ←_*n*/2_ **downto** 1 **do**
    k←*i*;
    v←*H*[*k*]
    heap←**false**
    while not *heap* **and** $2 * k \le n$ **do**
        $j \leftarrow 2 * k$
        if $j < n$ //there are two children
            if $H[j] < H[j+1]$
                $j \leftarrow j + 1$
        if $v \ge H[j]$
            heap←**true**
        else
            H[*k*]←*H*[*j*]; *k*←*j*
    H[*k*]←*v*

Program:

```
#include <stdio.h>
#include <stdlib.h>
void Heapify(int H[],int n)    //To construct the heap
{
  int k,v,i,j,heap=0;
  for(i=(n/2); i>=1; i--)
```

```
  {
    k=i;
    v=H[k];
    heap=0;
    while(!heap && (2*k)<=n)
    {
      j=2*k;   //Get the left child of a root node
      if(j<n)  //To check whether root had right child also
         if(H[j]<H[j+1])
           j++;
      if(v>=H[j])
         heap=1;
      else
      {
        H[k]=H[j];
        k=j;
      }
    }
    H[k]=v;
  }
}
void HeapSort(int H[],int n)
{
  int i;
  Heapify(H,n);
  for(i=n; i>=2; i--)
  {
    H[1]=H[1]+H[i]; //
    H[i]=H[1]-H[i]; // SWAP without using temp
    H[1]=H[1]-H[i]; // It is same even if swapping done using temp
    Heapify(H,i-1);
  }
}
int main()
{
  int n,i;
  printf("Enter the number of elements : ");
  scanf("%d",&n);
  int a[n+1];
  printf("Enter %d elements :\n",n);
  for(i=1; i<=n; i++)
    scanf("%d",&a[i]);
  printf("Array before sorting:\n");
  for(i=1; i<=n; i++)
    printf("\t%d",a[i]);
  HeapSort(a,n);
  printf("\nArray after sorting:\n");
  for(i=1; i<=n; i++)
    printf("\t%d",a[i]);
  return 0;
}
```

Output:

```
Enter the number of elements : 5
Enter 5 elements :
59 21 45 87 99
Array before sorting:
        59      21      45      87      99
Array after sorting:
        21      45      59      87      99
```

```
Enter the number of elements : 10
Enter 10 elements :
91 27 29 35 86 71 64 54 35 45
Array before sorting:
        91      27      29      35      86      71      64      54      35      45
Array after sorting:
        27      29      35      35      45      54      64      71      86      91
```

12.Horspool's string matching

Write a C program to search the pattern in the given text using Horspool's string matching algorithm. Assume matching done by ignoring case-sensitivity of alphabets.

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←0 //number of matched characters
  while *k* ≤ *m* − 1 **and** *P*[*m* − 1− *k*]= *T* [*i* − *k*] **do**
    k←*k* + 1
  if *k* = *m*
    return *i* − *m* + 1
  else
    i ←*i* + *Table*[*T* [*i*]]
return −1

Approach : Space and Time trade off
Worst case efficiency : O(mn) – But faster than Brute force string match

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
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*

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
int table[128];
int m,n;
void ShiftTable(char P[m])
{
   int i,j;
   for(i=0; i<128; i++)   //Initialize shift value m to all ascii values
```

```
        table[i]=m;
    for(j=0; j<=m-2; j++)  //Assign the shift to each character of pattern
        table[(int)tolower(P[j])]=table[(int)toupper(P[j])]=m-1-j;
}
int Horspool(char T[],char P[])
{
    int i,k;
    ShiftTable(P);
    i=m-1;
    while(i<=n-1)
    {
        k=0;
        while(k<=m-1 && (tolower(P[m-1-k])==tolower(T[i-k])))
            k++;
        if(k==m)    //Pattern match found
            return i-m+1;
        else
            i=i+table[(int)T[i]];   //shift amount
    }
    return -1;
}
int main()
{
    int pos;
    char text[100],pattern[25];
    printf("Enter the text:\n");
    gets(text);
    printf("Enter the pattern:\n");
    gets(pattern);
    n=strlen(text);
    m=strlen(pattern);
    pos=Horspool(text,pattern);
    if(pos==-1)
        printf("Pattern not found!!");
    else
        printf("Pattern %s found at position %d.",pattern,pos+1);
    return 0;
}
```

Output:

```
Enter the text:
Algorithm is the basic need, in order to find out program based solution to any problem.
Enter the pattern:
prob
Pattern prob found at position 81.
```

```
Enter the text:
algorithms
Enter the pattern:
msg
Pattern not found!!
```

13.Computing binomial co-efficient

Write a C program to compute binomial coefficient C(n,k) using dynamic programming approach.

Approach : Dynamic programming
Efficiency : Θ(nk)

Algorithm:

Binomial *(n, k)*
//Computes the binomial coefficient C(n,k)
//Input: Two integer values n and k of C(n,k)
//Output: Value C(n,k)

```
for i ←0  to n-1 do
    for j←0 to min{i,k} do
        if j = 0 or i==j
            C[i,j] ←1
        else
            C[i,j] ← C[ i-1, j-1] + C[i-1,j]
return C[n,k]
```

Program:

```c
#include <stdio.h>
#include <stdlib.h>
int min(int i,int k)
{
   if(i<k)
      return i;
   else
      return k;
}
int Binomial(int n,int k, int C[n+1][k+1])
{
   int i,j;
   for(i=0; i<=n; i++)
      for(j=0; j<=min(i,k); j++)
         if(j==0 || i==j)   //C(n,n)==1 or C(n,0) =1
            C[i][j]=1;
         else
            C[i][j]=C[i-1][j-1] + C[i-1][j];
   return C[n][k];
}
int main()
{
   int n,k,coeff;
   printf("Enter n and k in C(n,k) : ");
   scanf("%d%d",&n,&k);
   if(k>n)
   {
      printf("Invalid input!");  //C (n,k) for n<k is not defined
      exit(0);
   }
   int table[n+1][k+1];
   coeff=Binomial(n,k,table);
   printf("Binomial coefficient, C(%d,%d)=%d",n,k,coeff);
   return 0;
}
```

Output:

```
Enter n and k in C(n,k) : 8 3
Binomial coefficient, C(8,3)=56
```

```
Enter n and k in C(n,k) : 5 0
Binomial coefficient, C(5,0)=1
```

```
Enter n and k in C(n,k) : 3 7
Invalid input!
```

14.Warshall's Algorithm

Write a C program to compute transitive closure of a graph using Warshall's algorithm.

Algorithm:

Warshall(*A*[1..*n*, 1..*n*])

//ImplementsWarshall's algorithm for computing the transitive closure
//Input: The adjacency matrix *A* of a digraph with *n* vertices
//Output: The transitive closure of the 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 $R^{(n)}$

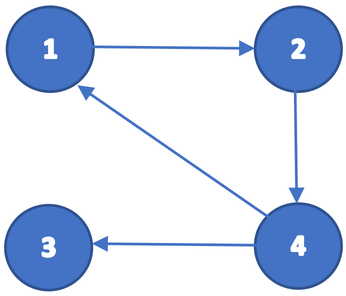
Approach : Dynamic programming
Efficiency : $\Theta(n^3)$

Program:

```
#include <stdio.h>
#include <stdlib.h>
int A[25][25];
void Warshalls(int n)
{
   int i,j,k;
   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]);
}
int main()
{
   int v,e,i,j,v1,v2;
   printf("Enter the number of vertices : ");
   scanf("%d",&v);
   printf("Enter the number of edges : ");
   scanf("%d",&e);
   printf("\nEnter %d edges :\n",e);
   for(i=1; i<=e; i++)
   {
      printf("Edge-%d : ",i);
      scanf("%d%d",&v1,&v2);
      A[v1][v2]=1;
   }
   printf("\nAdjacency matrix :\n");
   for(i=1; i<=v; i++)
   {
      for(j=1; j<=v; j++)
         printf(" %d",A[i][j]);
      printf("\n");
   }
   Warshalls(v);
   printf("\nTransitive closure : \n");
   for(i=1; i<=v; i++)
   {
      for(j=1; j<=v; j++)
         printf(" %d",A[i][j]);
      printf("\n");
   }
   return 0;
}
```

Output:

```
Enter the number of vertices : 4
Enter the number of edges : 4

Enter 4 edges :
Edge-1 : 1 2
Edge-2 : 2 4
Edge-3 : 4 3
Edge-4 : 4 1

Adjacency matrix :
 0 1 0 0
 0 0 0 1
 0 0 0 0
 1 0 1 0

Transitive closure :
 1 1 1 1
 1 1 1 1
 0 0 0 0
 1 1 1 1
```



15.Floyd's Algorithm

Write a C program to compute all pair shortest path of a positive weighted graph using Floyd's 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 with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ←*W* //is not necessary if *W* can be overwritten
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*

Approach : Dynamic programming
Efficiency : $\Theta(n^3)$

Program:

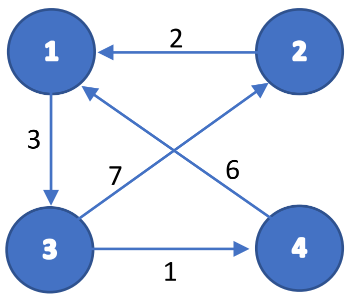
```
#include <stdio.h>
#include <stdlib.h>
int W[25][25];
int min(int a,int b)
{
   if(a<b)
      return a;
   else
      return b;
}
void Floyds(int n)
{
   int i,j,k;
   for(k=1; k<=n; k++)
      for(i=1; i<=n; i++)
         for(j=1; j<=n; j++)
            W[i][j]=min(W[i][j], W[i][k]+W[k][j]);
}
```

```
int main()
{
   int v,e,i,j,v1,v2,w;
   printf("Enter the number of vertices : ");
   scanf("%d",&v);
   printf("Enter the number of edges : ");
   scanf("%d",&e);
   for(i=1; i<=v; i++)
      for(j=1; j<=v; j++)
         if(i==j)
            W[i][j]=0;   //Vertex to itself is zero
         else
            W[i][j]=999;  //No direct edge
   printf("\nEnter %d edges :\n",e);
   for(i=1; i<=e; i++)
   {
      printf("Edge-%d : ",i);
      scanf("%d%d",&v1,&v2);
      printf("Enter the distance %d-->%d : ",v1,v2);
      scanf("%d",&w);
      W[v1][v2]=w;
   }
   printf("\nWeight matrix :\n");
   for(i=1; i<=v; i++)
   {
      for(j=1; j<=v; j++)
         printf(" %d",W[i][j]);
      printf("\n");
   }
   Floyds(v);
   printf("\nShortest distance matrix : \n");
   for(i=1; i<=v; i++)
   {
      for(j=1; j<=v; j++)
         printf(" %d",W[i][j]);
      printf("\n");
   }
   return 0;
}
```

Output:

```
Enter the number of vertices : 4
Enter the number of edges : 5

Enter 5 edges :
Edge-1 : 2 1
Enter the distance 2-->1 : 2
Edge-2 : 1 3
Enter the distance 1-->3 : 3
Edge-3 : 3 2
Enter the distance 3-->2 : 7
Edge-4 : 3 4
Enter the distance 3-->4 : 1
Edge-5 : 4 1
Enter the distance 4-->1 : 6
```

```
Weight matrix :
 0 999 3 999
 2 0 999 999
 999 7 0 1
 6 999 999 0

Shortest distance matrix :
 0 10 3 4
 2 0 5 6
 7 7 0 1
 6 16 9 0
```



16.Knapsack problem

Write a C program to compute optimal solution for knapsack problem using Knapsack memory function.

Algorithm:

MFKnapsack*(i, j)*
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integers *i&j* indicating number of first items considered & the knapsack capacity respectively
//Output: The value of an optimal feasible subset of the first *i* items
//Note: Uses as global variables input arrays *Weights*[1..*n*], *Values*[1..*n*], and table *F*[0..*n*, 0..*W*] whose entries are
//initialized with −1’s except for row 0 and column 0 initialized with 0’s
if *F*[*i, j*]< 0
if *j* <*Weights*[*i*]
value←*MFKnapsack(i* − 1*, j)*
else
value←max*(MFKnapsack(i* − 1*, j), Values*[*i*]+ *MFKnapsack(i* − 1*, j* −*Weights*[*i*]*))*
F[*i, j*]←*value*
return *F*[*i, j*]

Approach : Dynamic programming
Efficiency : Θ(nW)

Program:

```
#include <stdio.h>
#include <stdlib.h>
int weight[25],value[25],V[25][25];
int max(int a,int b)
{
   if(a>b)
      return a;
   else
      return b;
}
int MFK(int i,int j)
{
   if(i>=0 && j>=0)
   {
      int val;
      if(V[i][j]<0)
      {
         if(j<weight[i])   //If current item doesn't fit in current capacity of knapsack
            val=MFK(i-1,j);
         else
            val=max(MFK(i-1,j),value[i]+MFK(i-1,j-weight[i]));
         V[i][j]=val;
      }
   }
   return V[i][j];
}
int main()
{
   int n,w,i,j,x,y,soln;
   printf("Enter the number of items : ");
   scanf("%d",&n);
   printf("Enter the threshold weight of knapsack : ");
   scanf("%d",&w);
   for(i=0; i<=n; i++)
      for(j=0; j<=w; j++)
         if(i==0 || j==0)
```

```
            V[i][j]=0;   //Wight is zero or item is not selected
          else
            V[i][j]=-1;
    printf("Enter weight and value of %d items:\n",n);
    printf("Format: WEIGHT <space> VALUE\n");
    for(i=1; i<=n; i++)
    {
      printf("Item-%d : ",i);
      scanf("%d%d",&x,&y);
      weight[i]=x;
      value[i]=y;
    }
    soln=MFK(n,w);
    printf("The optimal solution is %d.",soln);
    return 0;
  }
```

Output:

```
Enter the number of items : 5
Enter the threshold weight of knapsack : 6
Enter weight and value of 5 items:
Format: WEIGHT <space> VALUE
Item-1 : 3 25
Item-2 : 2 20
Item-3 : 1 15
Item-4 : 4 40
Item-5 : 5 50
The optimal solution is 65.
```

17.Prim's Algorithm

Write a C program to construct minimum spanning tree(MST) of a graph using Prim's algorithm.

Algorithm:

Approach : Greedy approach
Efficiency : $\Theta(|V|^2)$

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
$E_T \leftarrow \emptyset$
for $i \leftarrow 1$ **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

Program:

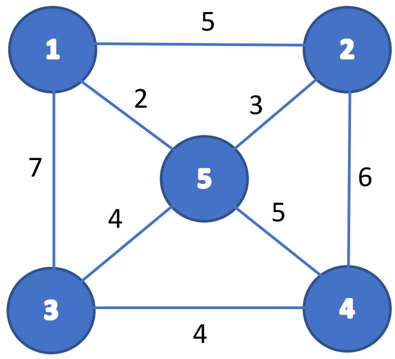
```
#include <stdio.h>
#include <stdlib.h>
int v,e,adjMat[20][20],VT[20],V_VT[20],edges[25][3],inc=0;

void initialize()   //Initializes Vt and V-Vt
{
  int i;
  for(i=1; i<=v; i++)
  {
    VT[i]=0;   //Vt is null initially
```

```
        V_VT[i]=1;  //V-Vt is V
    }
    VT[1]=1; // Initialize VT with vertex-1
    V_VT[1]=0;
}
void prims()
{   int i,j,k,v1,v2,min;
    initialize();
    for(i=1; i<v; i++) //<v because one vertex is already selected
    {
        min=9999;
        for(j=1; j<=v; j++) //To find minimum weight edge
            for(k=1; k<=v; k++)
                if(VT[j]!=0 && V_VT[k]!=0 && adjMat[j][k]<min)
                {
                    min=adjMat[j][k];
                    v1=j;
                    v2=k;
                }
        edges[inc][0]=v1;  //Store end vertices of minimum weight edge in a 2-D array
        edges[inc][1]=v2;
        edges[inc][2]=min;  //Store minimum weight along with the end vertices
        inc++;
        VT[v2]=1;
        V_VT[v2]=0;
    }
}
int main()
{
    int i,j,v1,v2,w,total=0;
    printf("Enter the number of vertices : ");
    scanf("%d",&v);
    printf("Enter the number of edges : ");
    scanf("%d",&e);
    printf("Enter %d edges :\n",e);
    for(i=1; i<=v; i++)
        for(j=1; j<=v; j++)
            if(i==j)
                adjMat[i][j]=0; //from vertex to itself
            else
                adjMat[i][j]=9999; //No direct edge
    for(i=1; i<=e; i++)
    {
        printf("Edge-%d : ",i);
        scanf("%d%d",&v1,&v2);
        printf("Enter the weight of edge %d %d : ",v1,v2);
        scanf("%d",&w);
        adjMat[v1][v2]=adjMat[v2][v1]=w;
    }
    prims();
    printf("\nEdges in minimum spanning tree :\n");
    for(i=0; i<inc; i++)
    {
        v1=edges[i][0];
```

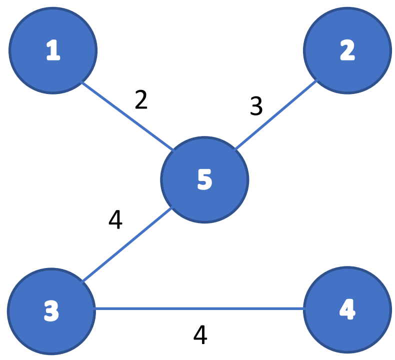
```
        v2=edges[i][1];
        total+=edges[i][2];
        printf("%d --> %d  (Weight:%d)\n",v1,v2,adjMat[v1][v2]);
    }
    printf("\nSum of edge weight in minimum spanning tree : %d",total);
    return 0;
}
```

Output:

```
Enter the number of vertices : 5
Enter the number of edges : 8
Enter 8 edges :
Edge-1 : 1 2
Enter the weight of edge 1 2 : 5
Edge-2 : 2 4
Enter the weight of edge 2 4 : 6
Edge-3 : 3 4
Enter the weight of edge 3 4 : 4
Edge-4 : 1 3
Enter the weight of edge 1 3 : 7
Edge-5 : 1 5
Enter the weight of edge 1 5 : 2
Edge-6 : 2 5
Enter the weight of edge 2 5 : 3
Edge-7 : 3 5
Enter the weight of edge 3 5 : 4
Edge-8 : 4 5
Enter the weight of edge 4 5 : 5

Edges in minimum spanning tree :
1 --> 5  (Weight:2)
5 --> 2  (Weight:3)
5 --> 3  (Weight:4)
3 --> 4  (Weight:4)

Sum of edge weight in minimum spanning tree : 13
```



Graph



Minimum spanning tree

18.Kruskal's Algorithm

Write a C program to construct minimum spanning tree(MST) of undirected graph using Kruskal's 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 nondecreasing order of the edge
//weights $w(e_{i1}) \leq \ldots \leq w(e_{i|E|})$
$E_T \leftarrow \emptyset$
$ecounter \leftarrow 0$ //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 $E_T \cup \{e_{ik}\}$ is acyclic
$E_T \leftarrow E_T \cup \{e_{ik}\}$
$ecounter \leftarrow ecounter + 1$
return E_T

Approach : Greedy approach
Efficiency : $\Theta(|E| \log(|E|))$ – When efficient sorting algorithm is used
$\Theta(|E|^2)$ – Using insertion sort

Program:

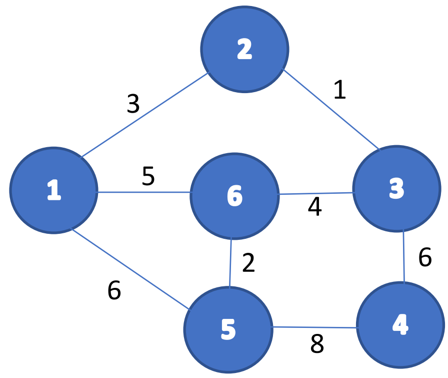
```
#include <stdio.h>
#include <stdlib.h>
int v,e,inserted[20],total=0;
struct node
{
   int v1,v2,w;
   struct node * link;
};
typedef struct node * NODE;

NODE addGraphEdge(NODE graph,int v1,int v2,int w)  //Store graph edges in linked list graph
{
   NODE temp=(NODE)malloc(sizeof(struct node));  //Create new node
   temp->v1=v1;
   temp->v2=v2;
   temp->w=w;
   if(graph == NULL)  //If linked list is empty
   {
      graph=temp;
      graph->link=NULL;
   }
   else
   {
      NODE cur= graph;
      NODE prev=NULL;
      while(cur!=NULL && temp->w > cur->w)  //Insertion sort
      {
         prev=cur;
         cur=cur->link;
      }
      if(prev==NULL)
      {
         graph=temp;
         temp->link=cur;
      }
      else
      {
         prev->link=temp;
         temp->link=cur;
      }
   }
   return graph;
}

NODE addTreeEdge(NODE tree,int v1,int v2,int w)   //Adds tree edges to minimum spanning tree
{
   NODE temp=(NODE)malloc(sizeof(struct node));
   temp->v1=v1;
   temp->v2=v2;
   temp->w=w;
   temp->link=NULL;
   if(tree == NULL)
      tree=temp;
```

```
    else
    {
        NODE cur=tree;
        while(cur->link!=NULL)
            cur=cur->link;
        cur->link=temp;
    }
    return tree;
}

NODE Kruskal(NODE graph,NODE tree)
{
    int Vcount=0,Ecount=0,c=0;
    NODE cur=graph;
    while(cur!=NULL)
    {
        if(inserted[cur->v1]==0)  //Count vertices in tree after adding an edge
        {
            c++;
            inserted[cur->v1]=1;   //mark added vertex as inserted
        }
        if(inserted[cur->v2]==0)  //Count second vertex  vertex
        {
            c++;
            inserted[cur->v2]=1;
        }
        Ecount++;  //One edge need to be added to tree
        if(Ecount>=(Vcount+c))  //Cheek for cycle.  If in any case of undirected graph, edgs>= vertices, there is cycle
        {
            Ecount--;     //Adding edge will result in cycle
        }
        else
        {
            Vcount=Vcount+c;
            tree=addTreeEdge(tree,cur->v1,cur->v2,cur->w);
            total+=cur->w;
        }
        c=0;
        cur=cur->link;
    }
    return tree;
}

int main()
{
    int i,v1,v2,w;
    NODE graph=NULL, tree=NULL;
    printf("Enter the number of vertices : ");
    scanf("%d",&v);
    printf("Enter the number of edges : ");
    scanf("%d",&e);
    printf("Enter %d edges :\n",e);
    for(i=1; i<=e; i++)
    {
```

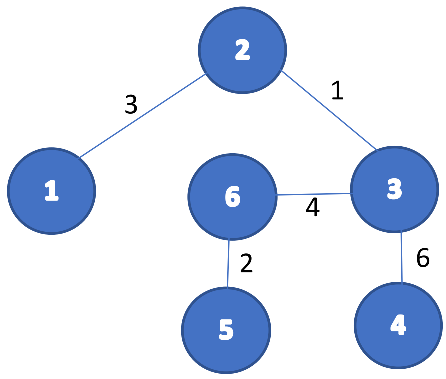
```
        printf("Edge-%d : ",i);
        scanf("%d%d",&v1,&v2);
        printf("Enter the weight of edge %d %d : ",v1,v2);
        scanf("%d",&w);
        graph=addGraphEdge(graph,v1,v2,w);
    }
    tree=Kruskal(graph,tree);
    printf("\nEdges in minimum spanning tree :\n");
    NODE cur=tree;
    while(cur!=NULL)
    {
        printf("%d --> %d (Weight:%d)\n",cur->v1,cur->v2,cur->w);
        cur=cur->link;
    }
    printf("\nSum of edge weight in minimum spanning tree : %d",total);
    return 0;
}
```

Output:

```
Enter the number of edges : 8
Enter 8 edges :
Edge-1 : 1 2
Enter the weight of edge 1 2 : 3
Edge-2 : 2 3
Enter the weight of edge 2 3 : 1
Edge-3 : 3 4
Enter the weight of edge 3 4 : 6
Edge-4 : 1 6
Enter the weight of edge 1 6 : 5
Edge-5 : 6 3
Enter the weight of edge 6 3 : 4
Edge-6 : 1 5
Enter the weight of edge 1 5 : 6
Edge-7 : 6 5
Enter the weight of edge 6 5 : 2
Edge-8 : 5 4
Enter the weight of edge 5 4 : 8

Edges in minimum spanning tree :
2 --> 3 (Weight:1)
6 --> 5 (Weight:2)
1 --> 2 (Weight:3)
6 --> 3 (Weight:4)
3 --> 4 (Weight:6)

Sum of edge weight in minimum spanning tree : 16
```



Graph



Minimum spanning tree

19.Dijkstra's Algorithm

Write a C program to find single source shortest path of a graph using Dijkstra's 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 d_v of a shortest path from *s* to *v* and its penultimate vertex p_v for every vertex *v* in *V*
Initialize(Q) //initialize priority queue to empty
for every vertex *v* in *V*

$d_v \leftarrow \infty$;
$p_v \leftarrow$ null
Insert(Q, v, d_v) //initialize vertex priority in the priority queue
$d_s \leftarrow 0$;
Decrease(Q, s, d_s) //update priority of s with d_s
$V_T \leftarrow \emptyset$
for $i \leftarrow 0$ to $|V| - 1$ do
$u^* \leftarrow$ *DeleteMin(Q)* //delete the minimum priority element
$V_T \leftarrow V_T \cup \{u^*\}$
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)

Approach : Greedy approach
Efficiency : $\Theta(|V|^2)$ – Adjacency matrix

Program:

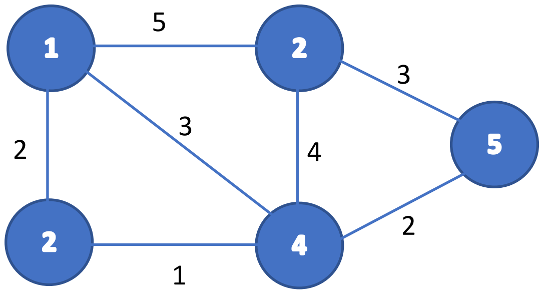
```
#include <stdio.h>
#include <stdlib.h>
int v,e,s,D[20],P[20],adjMat[25][25];
struct node
{
   int vertex;
   int distance;
   struct node * llink,*rlink;
};
typedef struct node * NODE;
NODE initialize(NODE queue)
{
   int i;
   NODE cur;
   for(i=1; i<=v; i++)
   {
      D[i]=9999;
      P[i]=0;
      NODE temp=(NODE)malloc(sizeof(struct node));  //Doubly linked list is used for easier deletions
      temp->vertex=i;
      temp->distance=9999;
      temp->rlink=NULL;
      if(queue==NULL)
      {
         queue=temp;
         queue->llink=NULL;
         cur=queue;
      }
      else
      {
         cur->rlink=temp;
         temp->llink=cur;
         cur=cur->rlink;
      }
   }
   return queue;
}
```

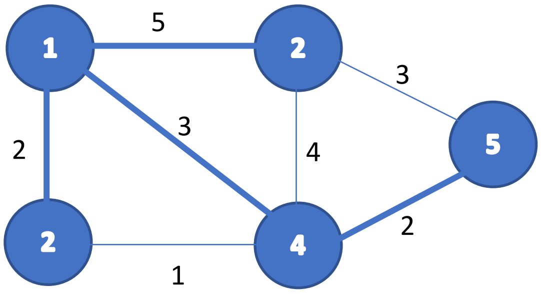
```
int deleteMin(NODE *queue)  //Delete minimum distance vertex from queue
{
   int del,m=9999;
   NODE cur=(*queue),prev,min=NULL;
   while(cur!=NULL)
   {
      if(cur->distance < m)
      {
         min=cur;   //Point min to node having minimum distance from source
         m=cur->distance;
      }
      cur=cur->rlink;
   }
   if(min==(*queue))
   {
      (*queue)=min->rlink;
      if(*queue != NULL)
              (min->rlink)->llink=NULL;
   }
   else
   {
      prev=min->llink;
      prev->rlink=min->rlink;
      if(min->rlink != NULL)
               (min->rlink)->llink=prev;
   }
   del=min->vertex;
   free(min);
   return del;
}
void Decrease(NODE queue,int v,int d)
{
   NODE cur;
   cur=queue;
   while(cur!=NULL)
   {
      if(cur->vertex == v)
      {
         cur->distance=d;
         return;
      }
      cur=cur->rlink;
   }
}

void Dijikstras(NODE queue)
{
   int i,j,u,VT[20],V_VT[20];
   queue=initialize(queue);
   for(i=1; i<=v; i++)
   {
      VT[i]=0;
      V_VT[i]=1;
   }
```

```
    VT[s]=1;
    V_VT[s]=0;
    D[s]=0;
    Decrease(queue,s,D[s]); //Decrease distance in queue
    for(i=1; i<=v; i++)
    {
       u=deleteMin(&queue);
       VT[u]=1;
       V_VT[u]=0;
       for(j=1; j<=v; j++)
          if(VT[u]==1 && V_VT[j]==1 && adjMat[u][j]<9999)
             if(D[u]+adjMat[u][j] < D[j])
             {
                D[j]=D[u]+adjMat[u][j];
                Decrease(queue,j,D[j]);
                P[j]=u;
             }
    }
}

int main()
{
    int i,j,v1,v2,w;
    NODE queue=NULL;
    printf("Enter the number of vertices : ");
    scanf("%d",&v);
    printf("Enter the number of edges : ");
    scanf("%d",&e);
    printf("Enter %d edges :\n",e);
    for(i=1; i<=v; i++)
       for(j=1; j<=v; j++)
          if(i==j)
             adjMat[i][j]=0;
          else
             adjMat[i][j]=9999;
    for(i=1; i<=e; i++)
    {
       printf("Edge-%d : ",i);
       scanf("%d%d",&v1,&v2);
       printf("Enter the weight of edge %d-->%d : ",v1,v2);
       scanf("%d",&w);
       adjMat[v1][v2]=adjMat[v2][v1]=w;
    }
    printf("Enter the source vertex : ");
    scanf("%d",&s);
    Dijikstras(queue);
    printf("\nSingle source (vertex-%d) shortest distances :\n",s);
    for(i=1;i<=v; i++)
    {
       if(D[i]==0) continue;           //If you want to display source to source distance(zero), remov this statement
       printf("Distance %d-->%d : %d\tPrevious vertex : %d\n",s,i,D[i],P[i]);
    }
    return 0;
}
```

Output:

```
Enter the number of vertices : 5
Enter the number of edges : 7
Enter 7 edges :
Edge-1 : 1 2
Enter the weight of edge 1-->2 : 5
Edge-2 : 1 3
Enter the weight of edge 1-->3 : 2
Edge-3 : 3 4
Enter the weight of edge 3-->4 : 1
Edge-4 : 1 4
Enter the weight of edge 1-->4 : 3
Edge-5 : 2 4
Enter the weight of edge 2-->4 : 4
Edge-6 : 2 5
Enter the weight of edge 2-->5 : 3
Edge-7 : 4 5
Enter the weight of edge 4-->5 : 2
Enter the source vertex : 1

Single source (vertex-1) shortest distances :
Distance 1-->2 : 5      Previous vertex : 1
Distance 1-->3 : 2      Previous vertex : 1
Distance 1-->4 : 3      Previous vertex : 1
Distance 1-->5 : 5      Previous vertex : 4
```





20.N-Queen Problem

*Write a C program to find solutions for placing n queens in a n*n chess board, such that no two queens attack each other.*

Algorithm:

***Nqueen**(n)*
//All solution for placing n queens in n*n chess board without attack of any two queens
//Input: Positive integer n indicates the number of queens
//Output: All solutions of placing n queens in n*n chess board without attack of any two queens
k ←∞; *count*←0 ;
col[k] ← 0
***while** k!=0*
    col[k] ←col[k] +1
    ***while** (col[k]<=n **and** cannotBePlaced(k, col[1..n])*
        col[k] ←col[k]+1
    if col[k]<n
        if k==n
            count ← count+1
            PRINT solution
        else
            k←k+1
            col[k] ←0
    else
        k←k-1
***return** count*

Approach : Backtracking
Efficiency : $\Theta(n^n)$

Program:

```
#include <stdio.h>
#include <stdlib.h>
int cannotBePlaced(int k,int col[ ])
{
   int i;
   for(i=1; i<=k-1; i++)       //Check whether 2 queens attack vertically, horizontally or diagonally
      if(col[k]==col[i] || (abs(i-k)==abs(col[i]-col[k])))
         return 1;       //Queen k cannot be placed in row k and column col[k]
```

```
    return 0;
  }
  int NQueen(int n)
  {
    int k=1;          //Indicated queen to be placed and row number
    int count=0;          //Number of possible solutions
    int i,j,col[n+1];
    col[k]=0;                  //Queen 1 is selected but yet to be placed in row 1 and column a[1]
    while(k!=0)          //Backtrack till any  one queen exists
    {
      col[k]=col[k]+1;            //Place queen k in 1st column
      while(col[k]<=n && cannotBePlaced(k,col))
        col[k]=col[k]+1;                  //Move queen k to next column
      if(col[k]<=n)                     //Queen successfully placed
      {
        if(k==n)          //All queens are placed
        {
          count++;
          printf("\nSolution-%d :\n",count);
          for(i=1; i<=n; i++)
          {
            for(j=1; j<=n; j++)
              if(col[i]==j)
                printf("  Q%d",i);
              else
                printf("   * ");
            printf("\n\n");
          }
        }
        else
        {
          k++;        //Select next queen
          col[k]=0;      //Queen k is yet to be placed
        }
      }
      else
        k--;        //Backtrack and select previous queen
    }
    return count;

  }
  int main()
  {
    int n,solutions;
    printf("Enter the number of queens : ");
    scanf("%d",&n);
    solutions=NQueen(n);
    if(solutions==0)
      printf("No solution!!");
    return 0;
  }
```

Output:

```
Enter the number of queens : 3
No solution!!
```

```
Enter the number of queens : 4

Solution-1 :
   *   Q1   *    *

   *    *    *   Q2

  Q3   *    *    *

   *    *   Q4   *


Solution-2 :
   *    *   Q1   *

  Q2   *    *    *

   *    *    *   Q3

   *   Q4   *    *
```

***