1)Bubble sort

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

int main()
{
   int arr[]={55,32,12,64,30};
   int n=sizeof(arr)/sizeof(arr[0]);
   bubblesort(arr,n);
   printf("Sorted Array: ");
   for(int i=0;i<n;i++)
      printf("%d\t",arr[i]);
}
```

2)Quick Sort

```
#include<stdio.h>
int swap(int* a, int* b)
{
   int temp=*a;
   *a=*b;
   *b=temp;
}
```

```
int partition(int arr[], int low, int high)
{
   int pivot=arr[high];
   int i=(low-1);
   for(int j=low;j<high;j++){
      if(arr[j]<pivot){
         i++;
         swap(&arr[i],&arr[j]);
      }
   }
   swap(&arr[i+1],&arr[high]);
   return (i+1);
}

void quicksort(int arr[], int l, int h)
{
   if(l<h){
      int pI=partition(arr,l,h);
      quicksort(arr,l,pI-1);
      quicksort(arr,pI+1,h);
   }
}

int main()
{
   int arr[]={55,32,12,64,30};
   int n=sizeof(arr)/sizeof(arr[0]);
   quicksort(arr,0,n-1);
   printf("Sorted Array: ");
   for(int i=0;i<n;i++)
      printf("%d\t",arr[i]);
}
```

3)Merge Sort

```
#include<stdio.h>

int merge(int arr[], int l, int mid, int h)
{
   int i=l, j=mid+1, k=l,b[l+h];
   while(i<=mid && j<=h){
      if(arr[i]<arr[j]){
         b[k]=arr[i];
         i++;
      }
      else{
         b[k]=arr[j];
         j++;
      }
      k++;
   }
   if(i>mid){
      while(j<=h){
         b[k]=arr[j];
         j++;
         k++;
      }
   }
   if(j>h){
      while(i<=mid){
         b[k]=arr[i];
         i++;
         k++;
      }
   }
   for(k=l;k<=h;k++)
      arr[k]=b[k];
```

```
}

void mergesort(int arr[],int l, int h)
{
   if(l<h){
      int mid=(l+h)/2;
      mergesort(arr,l,mid);
      mergesort(arr,mid+1,h);
      merge(arr,l,mid,h);
   }
}

int main()
{
   int arr[]={55,32,12,64,30};
   int n=sizeof(arr)/sizeof(arr[0]);
   mergesort(arr,0,n-1);
   printf("Sorted Array: ");
   for(int i=0;i<n;i++)
      printf("%d\t",arr[i]);
}
```

4)heap sort

```
#include<stdio.h>
void heapify(int arr[], int N, int i)
{
   int largest=i, left=(2*i)+1, right=(2*i)+2;
   if(left<N && arr[left]>arr[largest])
      largest=left;
   if(right<N && arr[right]>arr[largest])
      largest=right;
   if(largest!=i){
      int temp=arr[i];
      arr[i]=arr[largest];
```

```
        arr[largest]=temp;
        heapify(arr,N,largest);
    }
}

void heapSort(int arr[], int N)
{
    for(int i=N/2 -1;i>=0;i--)
    {
        heapify(arr,N,i);
    }
    for(int i=N-1;i>=0;i--){
        int temp=arr[0];
        arr[0]=arr[i];
        arr[i]=temp;
        heapify(arr,i,0);
    }
}

int main()
{
    int arr[]={55,32,12,64,30};
    int n=sizeof(arr)/sizeof(arr[0]);
    heapSort(arr,n);
    printf("Sorted Array: ");
    for(int i=0;i<n;i++)
        printf("%d\t",arr[i]);
}
```

5)FloydWarshall

```
#include<stdio.h>
#define INF 999
#define V 5
```

```
void printSol(int dist[][V])
{
   printf("The following matrix shows the shortest distance between any pair of vertices \n");
   for(int i=0;i<V;i++){
      for(int j=0;j<V;j++){
         if(dist[i][j]==INF)
            printf("%7s","INF");
         else
            printf("%7d",dist[i][j]);
      }
      printf("\n");
   }
}

void floydWarshall(int dist[][V])
{
   for(int k=0;k<V;k++){
      for(int i=0;i<V;i++){
         for(int j=0;j<V;j++){
            if(dist[i][k]+dist[j][k]<dist[i][j])
               dist[i][j]=dist[i][k]+dist[j][k];
         }
      }
   }
   printSol(dist);
}

int main()
{
   int graph[V][V]={{0,10,INF,30,INF},
   {10,0,5,INF,20},
   {INF,5,0,15,10},
   {30,INF,15,0,INF},
```

```
    {INF,20,10,INF,0}};
    floydWarshall(graph);
    return 0;
}
```

6)TSP

```
#include<stdio.h>
int tsp_g[5][5]={{0, 10, 15, 20, 8},
   {10, 0, 35, 25, 12},
   {15, 35, 0, 18, 30},
   {20, 25, 18, 0, 22},
   {8, 12, 30, 22, 0}};
int visited[30],n,cost=0;
#define INF 999

void tsp(int c)
{
    int adj_vertex=INF,min=INF;
    visited[c]=1;
    printf("%d",c+1);
    for(int k=0;k<n;k++)
    {
        if((tsp_g[c][k]!=0)&&(visited[k]==0)){
            if(tsp_g[c][k]<min){
                min=tsp_g[c][k];
                adj_vertex=k;
            }
        }
    }
    if(min!=INF)
        cost=cost+min;
    if(adj_vertex==INF){
        adj_vertex=0;
        printf("%d",adj_vertex+1);
```

```
      cost=cost+tsp_g[c][adj_vertex];
      return;
   }
   tsp(adj_vertex);
}

int main()
{
  int i;
  n = 5;
  for(i = 0; i < n; i++) {
    visited[i] = 0;
  }
  printf("Shortest Path: ");
  tsp(0);
  printf("\nMinimum Cost: ");
  printf("%d\n", cost);
  return 0;
}
```

7)Dijkstra

```
#include<stdio.h>
#include<limits.h>
#include<stdbool.h>
#define V 9
 int minDist(int dist[],bool sptSet[])
 {
    int min=INT_MAX,min_index;
    for(int v=0;v<V;v++){
      if((sptSet[v]==false)&& (dist[v]<=min)){
          min=dist[v];
          min_index=v;}
    }
    return min_index;
```

```
}

void printSol(int dist[])
{
   printf("Vertex\t\tDistance from Source\n");
   for(int i=0;i<V;i++){
     printf("%d\t\t\t\t%d\n",i,dist[i]);
   }
}

void dijkstra(int graph[V][V],int src)
{
   int dist[V];
   bool sptSet[V];
   for(int i=0;i<V;i++){
     dist[i]=INT_MAX;
     sptSet[i]=false;
   }
   dist[src]=0;
   for(int count=0;count<V-1;count++){
     int u=minDist(dist,sptSet);
     sptSet[u]=true;
     for(int v=0;v<V;v++){
        if(!sptSet[v]&&graph[u][v]
          &&dist[u]!=INT_MAX
          &&dist[u]+graph[u][v]<dist[v])
           dist[v]=dist[u]+graph[u][v];
     }
   }
   printSol(dist);
}

int main()
```

```
{
   /* Let us create the example graph discussed above */
   int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                       { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                       { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                       { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                       { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                       { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                       { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                       { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                       { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

   // Function call
   dijkstra(graph, 0);

   return 0;
}
```

8)Bellman Ford

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

// Define the number of vertices in the graph
#define V 5
#define E 7

// Function to print the solution
void printSolution(int dist[], int n)
{
   printf("Vertex\t\t Distance from Source\n");
   for (int i = 0; i < n; i++)
      printf("%d \t\t %d\n", i, dist[i]);
}
```

```
// The main function that finds shortest distances from src to all other vertices using Bellman-Ford algorithm
void BellmanFord(int graph[E][3], int src)
{
   // Initialize distance of all vertices as infinite.
   int dist[V];
   for (int i = 0; i < V; i++)
      dist[i] = INT_MAX;
   dist[src] = 0;

   // Relax all edges |V| - 1 times.
   for (int i = 1; i <= V - 1; i++) {
      for (int j = 0; j < E; j++) {
         int u = graph[j][0];
         int v = graph[j][1];
         int weight = graph[j][2];
         if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            dist[v] = dist[u] + weight;
      }
   }

   // Check for negative-weight cycles.
   for (int i = 0; i < E; i++) {
      int u = graph[i][0];
      int v = graph[i][1];
      int weight = graph[i][2];
      if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
         printf("Graph contains negative weight cycle\n");
         return;
      }
   }

   printSolution(dist, V);
```

```
}

int main()
{
   // Create a graph represented as an edge list
   // graph[E][3] where E is the number of edges and each edge is represented by three values (u, v, w)
   int graph[E][3] = { { 0, 1, 6 },   // Jadavpur to Garia
                       { 0, 2, 5 },   // Jadavpur to Ballygunge
                       { 2, 3, 4 },   // Ballygunge to Rashbihari
                       { 2, 4, 3 },   // Ballygunge to Kalighat
                       { 3, 4, 3 },   // Rashbihari to Kalighat
                       { 2, 1, 2 },   // Ballygunge to Garia (discounted off-peak)
                       { 1, 3, 1 } }; // Garia to Rashbihari (discounted off-peak)

   // Source vertex (Jadavpur)
   int src = 0;

   // Run Bellman-Ford algorithm
   BellmanFord(graph, src);

   return 0;
}
```

9)KMP

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

void calculateLPS(char* pat, int M, int* lps)
{
   int length=0;
   lps[0]=0;
   int i=1;
   while(i<M){
      if(pat[i]==pat[length]){
```

```
            length++;
            lps[i]=length;
            i++;
        }
        else{
            if(length!=0)
                length=lps[length-1];
            else{
                lps[i]=0;
                i++;
            }
        }
    }
}

void KMPsearch(char* pat, char* txt)
{
    int M=strlen(pat);
    int N=strlen(txt);
    int lps[M];
    calculateLPS(pat,M,lps);
    int i=0,j=0;
    while(i<N){
        if(pat[j]==txt[i]){
            i++;
            j++;
        }
        if(j==M){
            printf("Pattern found at index: %d",i-j);
            j=lps[j-1];
        }
        else if(pat[j]!=txt[i]){
            if(j!=0)
```

```
            j=lps[j-1];
        else
            i++;
      }
   }
}

int main()
{
   char txt[] = "The challenges include change of skin disease color according to skin natures, elevated and depressed surface of skin lesions";
   char pat[] = "skin disease color according to skin natures";
   KMPsearch(pat, txt);
   return 0;
}
```

10)Naïve

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

void naive(char* pat, char* txt)
{
   int i,j;
   int M= strlen(pat);
   int N= strlen(txt);
   for(i=0;i<N-M;i++){
      for(j=0;j<M;j++){
         if(txt[i+j]!=pat[j])
            break;
      }
         if(j==M){
            printf("The pattern found at index: %d\n",i);
      }
   }
```

```
}

int main()
{
   char pat[]="aaba";
   char txt[] = "abacaabac";
   naive(pat,txt);
   return 0;
}
```

11)Knapsack(0|1)

```
#include<stdio.h>

int max(int a, int b){return a>b?a:b;}
int knapsack(int W, int wt[], int val[], int n)
{
   if(n==0||W==0)
      return 0;
   if(wt[n-1]>W)
      return knapsack(W,wt,val,n-1);
   else
      return(max(val[n-1]+knapsack(W-wt[n-1],wt,val,n-1), knapsack(W, wt, val, n-1)));
}

int main()
{
  int profit[] = { 4,2,5,3};
   int weight[] = {3,2,4,3};
   int W = 5;
   int n = sizeof(profit) / sizeof(profit[0]);
   printf("%d", knapsack(W, weight, profit, n));
   return 0;
}
```

12)Matrx Chain Multiplication

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

int matrixChain(int p[], int i, int j)
{
   if(i==j)
      return 0;
   int k,count,min=INT_MAX;
   for(k=i;k<j;k++){
      count=matrixChain(p,i,k)+matrixChain(p,k+1,j)+(p[i-1]*p[k]*p[j]);
      if(count<min)
         min=count;
   }
   return min;
}

int main()
{
   int arr[]={10, 50, 20, 100};
   int n = sizeof(arr)/sizeof(arr[0]);
   printf("The minimum number of multiplication is: %d", matrixChain(arr,1,n-1));
   return 0;
}
```

13)Nqueen

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

#define N 8

int totalsol = 0;
bool solution=false;
```

```
void printSol(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i][j])
                printf("\tQ");
            else
                printf("\tX");
        }
        printf("\n");
    }
    printf("\n");
}

bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    // Check row on the left side
    for (i = 0; i < col; i++) {
        if (board[row][i])
            return false;
    }

    // Check upper diagonal on the left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j])
            return false;
    }

    // Check lower diagonal on the left side
    for (i = row, j = col; i < N && j >= 0; i++, j--) {
        if (board[i][j])
```

```
        return false;
    }

    return true;
}

bool SolveUntil(int board[N][N], int col)
{
    if (col >= N) {
        totalsol++;
        if(!solution){
            printSol(board);
            solution=true;
        }
        return false;
    }

    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;

            if (SolveUntil(board, col + 1))
                return true;

            board[i][col] = 0;  // Backtrack
        }
    }

    return false;
}

bool SolveNQ()
{
```

```
    int board[N][N] = {0};

    if (SolveUntil(board, 0) == false) {
        if (totalsol == 0)
            printf("Solution does not exist\n");
        return false;
    }

    return true;
}

int main()
{
    SolveNQ();
    printf("Total Number of possible solutions is: %d\n", totalsol);
    return 0;
}
```