1) Implement selection sort.
Algorithm:
```
void selectionSort(int arr[], int n){
int i, j, min_idx;
for (i = 0; i < n-1; i++){
min_idx = i;
for (j = i+1; j < n; j++)
if (arr[j] < arr[min_idx])
min_idx = j;
if(min_idx != i)
swap(&arr[min_idx], &arr[i]); }
```
Code:
```
#include <stdio.h>
void swap(int *xp, int *yp){
int temp = *xp;
*xp = *yp;
*yp = temp;}
void selectionSort(int arr[], int n){
int i, j, min_idx;
for (i = 0; i < n-1; i++){
min_idx = i;
for (j = i+1; j < n; j++)
if (arr[j] < arr[min_idx])
min_idx = j;
if(min_idx != i)
swap(&arr[min_idx], &arr[i]);}}
void printArray(int arr[], int size){
int i;
for (i=0; i < size; i++)
printf("%d ", arr[i]);
printf("\n");}
int main(){
int arr[] = {64, 25, 12, 22, 11};
int n = sizeof(arr)/sizeof(arr[0]);
selectionSort(arr, n);
printf("Sorted array: \n");
printArray(arr, n);
return 0;}
```
Time complexity: n^2
Space complexity: O(1)

2) Implement insertion sort.
Algorithm:
```
for j = 2 to A.length
key ← A [j]
i ← j – 1
while i > 0 and A[i] > key
A[i+1] ← A[i]
```

```
i ← i – 1
A[i+1] ← key
```
Code:
```
#include <math.h>
#include <stdio.h>
void insertionSort(int arr[], int n){
int i, key, j;
for (i = 1; i < n; i++){
key = arr[i];
j = i - 1;
while (j >= 0 && arr[j] > key){
arr[j + 1] = arr[j];
j = j - 1;}
arr[j + 1] = key;}}
void printArray(int arr[], int n){
int i;
for (i = 0; i < n; i++)
printf("%d ", arr[i]);
printf("\n");}
int main(){
int arr[] = {12, 11, 13, 5, 6}; int
n = sizeof(arr) / sizeof(arr[0]);
insertionSort(arr, n);
printf("Sorted array: \n");
printArray(arr, n);
return 0; }
```
Time complexity: best=n , avg=n^2 , worst=n^2
Space complexity: O(1)

3) Implement Binary Search.
Algorithm:
```
Algorithm BinSearch(a, n, x){
low = 1; high := n;
while (low<=high) do{
mid= [(low + high)/2];
if (x < a[mid]) then high = mid - 1;
else if (x > a[mid]) then low := mid + 1;
else return mid;}
return 0;}
```
Code:
```
#include<stdio.h>
int main(){
int i, j, n, arr[100], search, low, high, middle, temp;
printf("No. of elements in the array:\n");
scanf("%d", &n);
printf("Enter array elements:\n");
for(i=0; i<n; i++)
scanf("%d", &arr[i]);
printf("Enter element to be search:\n");
scanf("%d", &search);
for(i=0; i<(n-1); i++){
for(j=0; j<(n-i-1); j++){
if(arr[j]>arr[j+1]){
temp = arr[j];
arr[j] = arr[j+1];
arr[j+1] = temp;}}}
printf("Now the sorted array is:\n");
for(i=0; i<n; i++)
printf("%d ", arr[i]);
low = 0;
high = n-1;
middle = (low+high)/2;
while(low <= high){
if(arr[middle]<search)
low = middle+1;
else if(arr[middle]==search){
printf("\nThe number, %d found at position %d", search, middle);
break;}
else
high = middle-1;
middle = (low+high)/2;}
if(low>high)
printf("The number, %d is not found in given array\n", search);
return 0;}
```

Time complexity: best=O(1) , avg=O(log n) , worst=O(log n)
Space complexity: O(1)

4) Implement merge sort.
Algorithm:
Code:

```
#include <stdio.h>
#include <stdlib.h>
void merge(int arr[], int l, int m, int r){
int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;
int L[n1], R[n2];
for (i = 0; i < n1; i++)
L[i] = arr[l + i];
for (j = 0; j < n2; j++)
R[j] = arr[m + 1 + j];
i = 0;
j = 0;
k = l;
while (i < n1 && j < n2) {
if (L[i] <= R[j]) {
```

```
arr[k] = L[i];
i++;}
else {
arr[k] = R[j];
j++;}
k++;}
while (i < n1) {
arr[k] = L[i];
i++;
k++;}
while (j < n2) {
arr[k] = R[j];
j++;
k++;}}
void mergeSort(int arr[], int l, int r){
if (l < r) {
int m = l + (r - l) / 2;
mergeSort(arr, l, m);
mergeSort(arr, m + 1, r);
merge(arr, l, m, r);}}
void printArray(int A[], int size){
int i;
for (i = 0; i < size; i++)
printf("%d ", A[i]);
printf("\n");}
int main(){
int arr[] = { 12, 11, 13, 5, 6, 7 };
int arr_size = sizeof(arr) / sizeof(arr[0]);
printf("Given array is \n");
printArray(arr, arr_size);
mergeSort(arr, 0, arr_size - 1);
printf("\nSorted array is \n");
printArray(arr, arr_size);
return 0;}
```
Time complexity: = Θ(nlogn)
Space complexity: O(n)

5) Implement quick sort.
Algorithm:
```
quickSort(arr[], low, high) {
if (low < high) {
pi = partition(arr, low, high);
quickSort(arr, low, pi – 1); // Before pi
quickSort(arr, pi + 1, high); // After pi}}
partition (arr[], low, high){
pivot = arr[high];
i = (low – 1)
for (j = low; j <= high- 1; j++){
```

```
if (arr[j] < pivot){
i++;
swap arr[i] and arr[j]}}
swap arr[i + 1] and arr[high])
return (i + 1)
```
Code:
```
#include<stdio.h>
void swap(int *a, int *b) {
int t = *a;
*a = *b;
*b = t;}
int partition(int array[], int low, int high) {
int pivot = array[high];
int i = (low - 1);
for (int j = low; j < high; j++) {
if (array[j] <= pivot) {
i++;
swap(&array[i], &array[j]);}}
swap(&array[i + 1], &array[high]);
return (i + 1);}
void quickSort(int array[], int low, int high) {
if (low < high) {
int pi = partition(array, low, high);
quickSort(array, low, pi - 1);
quickSort(array, pi + 1, high);}}
void printArray(int array[], int size) {
for (int i = 0; i < size; ++i) {
printf("%d ", array[i]);}
printf("\n");}
int main(){
int arr[30],ele,i,j;
printf("Enter no of elements:");
scanf("%d",&ele);
printf("Enter elements:");
for(i=0;i<ele;i++){
scanf("%d",&arr[i]);}
printf("given array is:");
printArray(arr,ele);
quickSort(arr,0,ele-1);
printf("Sorted array is:");
printArray(arr,ele);
return 0;}
```
Time complexity: best=O(nlogn) , avg=O(n^2) , worst=O(nlogn)
Space complexity: O(log n)

6) Implement Knapsack Problem.
Algorithm:
```
Greedy-fractional-knapsack (w, v, W)
```

```
FOR i =1 to n
do x[i] =0
weight = 0
while weight < W
do i = best remaining item
IF weight + w[i] ≤ W
then x[i] = 1
weight = weight + w[i]
else
x[i] = (w - weight) / w[i]
weight = W
return x
```

Code:

```
#include<stdio.h>
void knapsack(int n, float weight[], float profit[], float capacity) {
float x[20], tp = 0;
int i, j, u;
u = capacity;
for (i = 0; i< n; i++)
x[i] = 0.0;
for (i = 0; i< n; i++) {
if (weight[i] > u)
break;
else {
x[i] = 1.0;
tp = tp + profit[i];
u = u - weight[i];}}
if (i< n)
x[i] = u / weight[i];
tp = tp + (x[i] * profit[i]);
printf("\nThe result vector is:- ");
for (i = 0; i< n; i++)
printf("%f\t", x[i]);
printf("\nMaximum profit is:- %f", tp);}
int main() {
float weight[20], profit[20], capacity;
int num, i, j;
float ratio[20], temp;
printf("\nEnter the no. of objects:- ");
scanf("%d", &num);
printf("\nEnter the wts and profits of each object:- ");
for (i = 0; i<num; i++) {
scanf("%f %f", &weight[i], &profit[i]);}
printf("\nEnter the capacityacity of knapsack:- ");
scanf("%f", &capacity);
for (i = 0; i<num; i++) {
ratio[i] = profit[i] / weight[i];}
for (i = 0; i<num; i++) {
```

```
for (j = i + 1; j <num; j++) {
if (ratio[i] < ratio[j]) {
temp = ratio[j];
ratio[j] = ratio[i];
ratio[i] = temp;
temp = weight[j];
weight[j] = weight[i];
weight[i] = temp;
temp = profit[j];
profit[j] = profit[i];
profit[i] = temp;}}}
knapsack(num, weight, profit, capacity);
return(0);}
```

Time complexity: O(nW)
Space complexity: O(N*W)

7) Implement MST using Kruskal's Algorithm.
Algorithm:

```
MST-KRUSKAL(G, w)
A ← Ø
for each vertex v V[G]
do MAKE-SET(v)
sort the edges of E into nondecreasing order by weight w
for each edge (u, v) E, taken in nondecreasing order by weight
do if FIND-SET(u) ≠ FIND-SET(v)
then A ← A {(u, v)}
UNION(u, v)
return A
```

Code:

```
#include<stdio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
printf("\n\n\tImplementation of Kruskal's algorithm\n\n");
printf("\nEnter the no. of vertices\n");
scanf("%d",&n);
printf("\nEnter the cost adjacency matrix\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
 {
scanf("%d",&cost[i][j]);
if(cost[i][j]==0)
cost[i][j]=999;}}
printf("\nThe edges of Minimum Cost Spanning Tree are\n\n");
while(ne<n)
{
for(i=1,min=999;i<=n;i++)
 {
for(j=1;j<=n;j++)
 {
if(cost[i][j]<min)
 {
min=cost[i][j];
 a=u=i;
 b=v=j;}}}
 u=find(u);
 v=find(v);
if(uni(u,v))
 {
printf("\n%d edge (%d,%d) =%d\n",ne++,a,b,min);
mincost +=min;
 }
cost[a][b]=cost[b][a]=999;
}
printf("\n\tMinimum cost = %d\n",mincost);
}
int find(int i)
{
while(parent[i])
i=parent[i];
return i;}
int uni(int i,int j){
if(i!=j){
parent[j]=i;
return 1;}
return 0;}
```

Time complexity: O(E logE) or O(V logV)
Space complexity: O(|E| + |V|)

8) Implement MST using Prims Algorithm.
Algorithm:

```
MST-prim (G, w, r)
for each u E G.V
u.key = ∞
U.π = NIL
r.key = 0
Q = G.V
while Q Ø
u = EXTRACT-MIN(Q)
for each v G. Adj[u]
if v € Q and w(u, v) &lt; v. key
```

```
V.π = U
v.key = w(u, v)
```

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define infinity 9999
#define MAX 20
int G[MAX][MAX],
spanning[MAX][MAX], n;
int prims();
int main()
{
int i, j, total_cost;
printf("\nEnter no. of vertices: ");
scanf("%d", &n);
printf("\nEnter the adjacency matrix:\n");
for (i = 0; i < n; i++)
for (j = 0; j < n; j++)
scanf("%d", &G[i][j]);
total_cost = prims();
printf("\nspanning tree matrix:\n");
for (i = 0; i < n; i++)
{
printf("\n");
for (j = 0; j < n; j++)
printf("%d ", spanning[i][j]);
}
printf("\n\nTotal cost of spanning tree: %d", total_cost);
return 0;
}
int prims()
{
int cost[MAX][MAX];
int u, v, min_distance, distance[MAX], from[MAX];
int visited[MAX], no_of_edges, i, min_cost, j;
// create cost[][] matrix,spanning[][]
for (i = 0; i < n; i++)
for (j = 0; j < n; j++)
{
if (G[i][j] == 0)
cost[i][j] = infinity;
else
cost[i][j] = G[i][j];
spanning[i][j] = 0;
}
distance[0] = 0;
visited[0] = 1;
for (i = 1; i < n; i++)
{
distance[i] = cost[0][i];
from[i] = 0;
visited[i] = 0;
}
min_cost = 0; // cost of spanning tree
no_of_edges = n - 1; // no. of edges to be added
while (no_of_edges > 0)
{
min_distance = infinity;
for (i = 1; i < n; i++)
if (visited[i] == 0 && distance[i] < min_distance)
{
v = i;
min_distance = distance[i];
}
u = from[v];
spanning[u][v] = distance[v];
spanning[v][u] = distance[v];
no_of_edges--;
visited[v] = 1;
for (i = 1; i < n; i++)
if (visited[i] == 0 && cost[i][v] < distance[i]){
distance[i] = cost[i][v];
from[i] = v;}
min_cost = min_cost + cost[u][v];}
return (min_cost);}
```

Time complexity: O ((V + E) l o g V)

Space complexity: O(E+V)

9) Implement Bellman Ford.

Algorithm:

```
function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
    previous[V] <- NULL
  distance[S] <- 0
  for each vertex V in G
    for each edge (U,V) in G
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
  for each edge (U,V) in G
    If distance[U] + edge_weight(U, V) < distance[V}
      Error: Negative Cycle Exists
  return distance[]
```

Code:

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

```
#include<stdlib.h>
#include<string.h>
#include<limits.h>
struct Edge
{
int source,destination,weight;
};
struct Graph
{
int V,E;
struct Edge*edge;
};
struct Graph*createGraph(int V, int E)
{
struct Graph*graph=(struct Graph*)malloc(sizeof(struct Graph));
graph->V=V;
graph->E=E;
graph->edge=(struct Edge*)malloc(graph->E*sizeof(struct Edge));
return graph;
}
void FinalSolution(int dist[],int n)
{
printf("\nVertex\tDistance from source vertex\n");
int i;
for(i=1;i<=n;++i)
printf("%d \t\t %d\n",i,dist[i]);
}
void BellmanFord(struct Graph*graph, int source)
{
int V=graph->V;
int E=graph->E;
int StoreDistance[V];
int i,j;
for(i=1;i<=V;i++)
StoreDistance[i]=INT_MAX;
StoreDistance[source]=0;
for(i=1;i<=V-1;i++)
{
for(j=0;j<E-1;j++)
{
int u=graph->edge[j].source;
int v=graph->edge[j].destination;
int weight=graph->edge[j].weight;
if(StoreDistance[u]+weight<StoreDistance[v])
StoreDistance[v]=StoreDistance[u]+weight;}}
for(i=0;i<E;i++)
{
int u=graph->edge[i].source;
int v=graph->edge[i].destination;
int weight=graph->edge[i].weight;
if(StoreDistance[u]+weight<StoreDistance[v])
printf("This graph contain negative edge cycle\n");
}
FinalSolution(StoreDistance, V);
return;
}
int main()
{
int V,E,S;
printf("enter no> of vertices in graph\n");
scanf("%d",&V);
printf("enter no> of edges in graph\n");
scanf("%d",&E);
printf("enter no> of source vertices in graph\n");
scanf("%d",&S);
struct Graph* graph= createGraph(V,E);
int i;
for(i=0;i<E;i++){
printf(" Enter the edge %d properties source, destination, weight respectively\n",i+1);
scanf("%d",&graph->edge[i].source);
scanf("%d",&graph->edge[i].destination);
scanf("%d",&graph->edge[i].weight);}
BellmanFord(graph,S);
return 0;}
```

Time complexity: O(VE)
Space complexity: O(V)

10) Implement Travelling Salesman Problem (TSP).
Algorithm:

```
Procedure TSP (N, s)
Visited[s] = 1;
if |N| = 2 and k!=s then
Cost(N, k)=dist(s, k);
Return Cost;
else
for j € N do
for i € N and visited[i] = 0 do
if j!=i and j!=s then
Cost (N, j) = min ( TSP(N − {i}, j) + dist(j, i))
Visited[j] = 1;
end
end
end
end
Return Cost;
```

end

Code:

```
#include<stdio.h>
int a[10][10],visited[10],n,cost=0;
void get()
{
int i,j;
printf("Enter No. of Cities: ");
scanf("%d",&n);
printf("\nEnter Cost Matrix\n"); for(i=0;i <n;i++) {
printf("\nEnter Elements of Row # : %d\n",i+1);
for( j=0;j <n;j++) scanf("%d",&a[i][j]);
visited[i]=0;
}
printf("\n\nThe cost list is:\n\n");
for(i=0;i <n;i++)
{
printf("\n\n"); for(j=0;j <n;j++)
printf("\t%d",a[i][j]);
}
}
void mincost(int city)
{
int i,ncity; visited[city]=1; printf("%d -->",city+1);
ncity=least(city); if(ncity==999) { ncity=0; printf("%d",ncity+1);
cost+=a[city][ncity]; return; } mincost(ncity);
}
int least(int c)
{
int i,nc=999; int min=999,kmin; for(i=0;i <n;i++) {
if((a[c][i]!=0)&&(visited[i]==0))
if(a[c][i] < min) { min=a[i][0]+a[c][i]; kmin=a[c][i]; nc=i; } }
if(min!=999) cost+=kmin; return nc; } void put() {
printf("\n\nMinimum cost:");
printf("%d",cost); } void main()
{
get();
printf("\n\nThe Path is:\n\n");
mincost(0);
put(); }
```

Time complexity: O(N!)
Space complexity: O(1)

11) Implement Longest Common Subsequence (LCS).
Algorithm:

```
X.label = X
Y.label = Y
LCS[0][] = 0
```

```
LCS[][0] = 0
Start from LCS[1][1]
Compare X[i] and Y[j]
If X[i] = Y[j]
LCS[i][j] = 1 + LCS[i-1, j-1]
Point an arrow to LCS[i][j]
Else
LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
Point an arrow to max(LCS[i-1][j], LCS[i][j-1])
```

Code:

```
#include<stdio.h>
#include<string.h>
void longest_common_subsequence_algorithm();
void print_sequence(int a, int b);
int a, b, c, d;
int temp[30][30];
char first_sequence[30], second_sequence[30], longest_sequence[30][30];
int main()
{
printf("\nEnter the First String:\t");
scanf("%s", first_sequence);
printf("\nEnter the Second String:\t");
scanf("%s", second_sequence);
printf("\nLongest Common Subsequence:\t");
longest_common_subsequence_algorithm();
print_sequence(c, d);
printf("\n");
 return 0;
}
void longest_common_subsequence_algorithm()
{
c = strlen(first_sequence);
d = strlen(second_sequence);
 for(a = 0; a <= c; a++)
 {
temp[a][0] = 0;
}
for(a = 0; a <= d; a++)
 {
temp[0][a] = 0;}
for(a = 1; a <= c; a++){
for(b = 1; b <= d; b++){
if(first_sequence[a - 1] == second_sequence[b - 1]){
temp[a][b] = temp[a - 1][b - 1] + 1;
longest_sequence[a][b] = 'c';}
 else if(temp[a - 1][b] >= temp[a][b - 1]){
temp[a][b] = temp[a - 1][b];
longest_sequence[a][b] = 'u';}
else{
temp[a][b] = temp[a][b - 1];
longest_sequence[a][b] = 'l';}}}}
void print_sequence(int a, int b)
{
if(a == 0 || b == 0){
return;}
 if(longest_sequence[a][b] == 'c'){
print_sequence(a - 1, b - 1);
 printf("%c", first_sequence[a - 1]);}
else if(longest_sequence[a][b] == 'u'){
print_sequence(a - 1, b);}
else{
print_sequence(a, b - 1);}}
```

Time complexity: O(m*n)
Space complexity: O(n * m)

12) Implement N Queens.
Algorithm:

```
Algorithm NQueens(k,n)
{
for(i=1 to n ) do
{
if Place(k,i) then
{
x[k]=I
If(k=n) then write (x[1:n]);
else Nqueens(k+1,n);
}}}
Place(k,i)
{
For j=1 to k-1 do
if((x[j]=i)
then return false;
Return true;
}
```

Code:

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
int board[20],count;
int main()
{
int n,i,j;
void queen(int row,int n);
printf(" - N Queens Problem Using Backtracking -");
printf("\n\nEnter number of Queens:");
scanf("%d",&n);
queen(1,n);
return 0;
}
void print(int n)
{
int i,j;
printf("\n\nSolution %d:\n\n",++count);
for(i=1;i<=n;++i)
printf("\t%d",i);
for(i=1;i<=n;++i)
{
printf("\n\n%d",i);
for(j=1;j<=n;++j) //for nxn board
 {
if(board[i]==j)
printf("\tQ"); //queen at i,j position
else
printf("\t-"); //empty slot
 }
}
}
int place(int row,int column)
{
int i;
for(i=1;i<=row-1;++i)
{
if(board[i]==column)
return 0;
else
if(abs(board[i]-column)==abs(i-row))
return 0;
}
return 1; //no conflicts
}
void queen(int row,int n)
{
int column;
for(column=1;column<=n;++column)
{
if(place(row,column))
 {
board[row]=column; //no conflicts so place queen
if(row==n) //dead end
print(n); //printing the board configuration
else
queen(row+1,n);}}}
```
Time complexity: `O(N!)`
Space complexity: O(N^2)

13) Implement Naive string matching methods.

Algorithm:

```
Start
pat_len := pattern Size
str_len := string size
for i := 0 to (str_len - pat_len), do
for j := 0 to pat_len, do
if text[i+j] ≠ pattern[j], then
break
if j == patLen, then
display the position i, as there pattern found
End
```

Code:

```
#include<stdio.h>
#include<string.h>
int match(char st[100], char pat[100]);
int main(int argc, char **argv) {
 char st[100], pat[100];
 int status;
 printf("*** Naive String Matching Algorithm ***\n");
 printf("Enter the String.\n");
 gets(st);
 printf("Enter the pattern to match.\n");
 gets(pat);
 status = match(st, pat);
 if (status == -1)
 printf("\nNo match found");
 else
 printf("Match has been found on %d position.", status);
 return 0;
}
int match(char st[100], char pat[100]) {
 int n, m, i, j, count = 0, temp = 0;
 n = strlen(st);
 m = strlen(pat);
 for (i = 0; i <= n - m; i++) {
 temp++;
 for (j = 0; j < m; j++) {
 if (st[i + j] == pat[j])
 count++;
 }
 if (count == m)
 return temp;
 count = 0;}
 return -1;}
```

Time complexity: O(mn)

Space complexity: O(1)