1) Insertion sort

```
int main() {
	int numbers[100], key, i, j, n;

	printf("Enter the dimension of the array: ");
	scanf("%d", &n);

	// array input
	for(i = 0; i < n; i++) {
		printf("Enter numbers[%d]: ", i);
		scanf("%d", &numbers[i]);
	}

	// insertion sort
	for(j = 1; j < n; j++) {
		key = numbers[j];
		i = j - 1;

		while (i >= 0 && numbers[i] > key) {
			numbers[i + 1] = numbers[i];
			i = i - 1;
		}

		numbers[i + 1] = key;
	}

	// printing the sorted array
	for(i = 0; i < n; i++) {
		printf("%d ", numbers[i]);
	}

	return 0;
}
```

Algorithm:

for j = 2 to A.length

key ← A [j]

// Insert A[j] into the sorted sequence A[1 .. j-1]
i ← j – 1

while i > 0 and A[i] > key

    A[i+1] ← A[i]
    i ← i – 1

A[i+1] ← key

Time complexity: The worst-case and Average-case time complexity of the insertion sort algorithm is O(n^2)
The best-case time complexity of the insertion sort algorithm is O(n)

2) Selection Sort

```
#include <stdio.h>
int main()
{
int a[100], n, i, j, position, swap;
printf("Enter number of elements\n");
scanf("%d", &n);
printf("Enter %d Numbers",n);
for (i = 0; i < n; i++)
scanf("%d", &a[i]);
for(i = 0; i < n - 1; i++)
{
position=i;
for(j = i + 1; j < n; j++)
{
if(a[position] > a[j])
position=j;
}
if(position != i)
{
swap=a[i];
a[i]=a[position];
a[position]=swap;
}
}
printf("Sorted Array:\n");
for(i = 0; i < n; i++)
printf("%d\n", a[i]);
return 0;
}
```

Time complexity: In the case of selection sort, the worst-case, best-case, and average-case complexities are all O(n^2), where n is the size of the input array

Algorithm

```
void selection(int arr[], int n)
{
        int i, j, temp, min = 0 ;
        for ( i = 0 ; i <= n - 2 ; i++ )
        {
                min = i;
                for ( j = i+1 ; j <= n-1 ; j++ )
                {
                        if (arr[j] < arr[min])
                           min = j;
                }
                temp     = arr[i];
                arr[i]   = arr[min];
                arr[min] = temp;
        }
```

Selection Sort Algorithm

3) Merge Sort using divide and conquer

```
#include <stdio.h>
void merge(int arr[], int l, int m, int r)
{
 int i, j, k, n1 = m - l + 1, n2 = r - m, 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];
 for (i = 0, j = 0, k = l; i < n1 && j < n2; k++)
 arr[k] = L[i] <= R[j] ? L[i++] : R[j++];
 while (i < n1)
arr[k++] = L[i++];
 while (j < n2)
arr[k++] = R[j++];
}
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);
 }
}
int main() {
 int n, i;
 printf("Enter the number of elements in the array: ");
 scanf("%d", &n);
 int arr[n];
```

```
 printf("Enter the elements of the array:\n");
 for (i = 0; i < n; i++) scanf("%d", &arr[i]);
 mergeSort(arr, 0, n - 1);
 printf("Sorted array: ");
 for (i = 0; i < n; i++) printf("%d ", arr[i]);
 printf("\n");
 return 0;
}
```

Time complexity: The time complexity of the merge sort algorithm is O(n log n) for all cases, which means that the algorithm's performance is efficient even for large input sizes.

4) **Quick sort using divide and conquer**

```
#include <stdio.h>
void 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 - 1; 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 low, int high) {
 if (low < high) {
 int pi = partition(arr, low, high);
 quickSort(arr, low, pi - 1);
 quickSort(arr, pi + 1, high);
 }
}
int main() {
 int n;
 printf("Enter the number of elements: ");
 scanf("%d", &n);
```

```
int arr[n];
printf("Enter the elements:\n");
for (int i = 0; i < n; i++) {
scanf("%d", &arr[i]);
}
quickSort(arr, 0, n - 1);
printf("Sorted array: ");
for (int i = 0; i < n; i++) {
printf("%d ", arr[i]);
}
printf("\n");
return 0;
}
```

Time complexity: The time complexity of Quick Sort is O(n*log(n)) in the best and average cases, and O(n^2) in the worst case.

5) Binary Search using divide and conquer

```
#include <stdio.h>

int binarySearch(int array[], int x, int low, int high)
{
  // Repeat until the pointers low and high meet each other
  while (low <= high) {
    int mid = low + (high - low) / 2;

    if (array[mid] == x)
      return mid;

    if (array[mid] < x)
      low = mid + 1;

    else
      high = mid - 1;
  }

  return -1;
}

int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 4;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
  return 0;
}
```

Algorithm

Algorithm BinSearch(a, n, x)
// Given an array $a[1:n]$ of elements in nondecreasing
// order, $n \geq 0$, determine whether x is present, and
// if so, return j such that $x = a[j]$; else return 0.
{
    $low := 1$; $high := n$;
    while $(low \leq high)$ **do**
    {
        $mid := \lfloor (low + high)/2 \rfloor$;
        if $(x < a[mid])$ **then** $high := mid - 1$;
        else if $(x > a[mid])$ **then** $low := mid + 1$;
            else return mid;
    }
    return 0;
}

6) Knapsack using greedy approach

```
#include<stdio.h>
void main()
{
int n,w[50],p[50],i,j;
float xr,x[50],total_profit=0,total_weight=0,ratio[50],u,m,temp;
printf("\nEnter number of items : ");
scanf("%d",&n);
printf("Enter capacity of knapsack : ");
scanf("%f",&m);
u=m;
for(i=0;i<n;i++)
{
x[i]=0;
}
printf("\nEnter the weights of each object\n");
for(i=0;i<n;i++)
{
printf("\tWeights of object %d = ", i+1);
scanf("%d",&w[i]);
}
printf("\nEnter the profit of each object \n");
for(i=0;i<n;i++)
{
printf("\tProfit of object %d = ", i+1);
scanf("%d",&p[i]);
}
for(i=0;i<n;i++)
```

```
{
ratio[i]=((float)p[i] / (float)w[i]);
}
int obj[n];
for(i=0;i<n;i++)
{
obj[i]=i+1;
}
for(i=0;i<n;i++)
{
for(j=0;j<n-1;j++)
{
if(ratio[j]<ratio[i])
{
temp=ratio[i];
ratio[i]=ratio[j];
ratio[j]=temp;
temp=w[i];
w[i]=w[j];
w[j]=temp;
temp=p[i];
p[i]=p[j];
p[j]=temp;
temp=obj[i];
obj[i]=obj[j];
obj[j]=temp;
}
}
}
printf("\n Table after sorting according to profit-weight ratio : \n");
printf("\nObject:\t\t");
for(i=0;i<n;i++)
{
printf("%d\t\t",obj[i]);
}
printf("\nProfit:\t\t");
for(i=0;i<n;i++)
{
printf("%d\t\t",p[i]);
}
printf("\nWeights:\t");
for(i=0;i<n;i++)
{
```

```
printf("%d\t\t",w[i]);
}
printf("\nRatio:\t\t");
for(i=0;i<n;i++)
{
printf("%f\t",ratio[i]);
}
printf("\n");
for(i=0;i<n;i++)
{
if(w[i]<=u)
{
x[i]=1;
u=u-w[i];
}
else if(w[i]>u)
{
break;
}
}
if(i<=n)
{
xr=(float)u/(float)w[i];
x[i]=xr;
}
printf("\n X=[");
for(i=0;i<n;i++)
{
printf("%.3f, ",x[i]);
}
printf("]");
for(i=0;i<n;i++)
{
total_profit+=x[i]*p[i];
total_weight+=x[i]*w[i];
}
printf("\n\nTotal weight=%.2f",total_weight);
printf("\nTotal profit=%.2f\n",total_profit);
}
```

Algorithm

Algorithm

- Consider all the items with their weights and profits mentioned respectively.
- Calculate P_i/W_i of all the items and sort the items in descending order based on their P_i/W_i values.
- Without exceeding the limit, add the items into the knapsack.
- If the knapsack can still store some weight, but the weights of other items exceed the limit, the fractional part of the next time can be added.
- Hence, giving it the name fractional knapsack problem.

Time complexity: all-case time complexity of O(n log n).

7) **Bellman Ford:**

```
#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 = 0; i < V; i++)
StoreDistance[i] = INT_MAX;
StoreDistance[source] = 0;
for (i = 1; i <= V-1; i++)
{
for (j = 0; j < E; 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 contains negative edge cycle\n");
}
FinalSolution(StoreDistance, V);
return;
}
int main()
{
int V,E,S;
printf("Enter number of vertices in graph\n");
scanf("%d",&V);
printf("Enter number of edges in graph\n");
scanf("%d",&E);
printf("Enter your source vertex number\n");
scanf("%d",&S);
struct Graph* graph = createGraph(V, E);
int i;
for(i=0;i<E;i++){
printf("\nEnter 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;
}
```

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[]
```

8) **Travelling Salesman:**

```
#include<stdio.h>
int a[10][10],visited[10],n,cost=0;
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 {
printf("\nEnter Elements of Row # : %d\n",i+1);
for( j=0;j scanf("%d",&a[i][j]);
visited[i]=0;
}
```

```
printf("\n\nThe cost list is:\n\n");
for(i=0;i {
printf("\n\n");
for(j=0;j printf("\t%d",a[i][j]);
}
}
int least(int c)
{
int i,nc=999;
int min=999,kmin;
for(i=0;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 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);
}
void put()
{
printf("\n\nMinimum cost:");
printf("%d",cost);
}
```

```
void main()
{
get();
printf("\n\nThe Path is:\n\n");
mincost(0);
put();
}
```

9) LCS

```
#include <stdio.h>
void 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 - 1; 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 low, int high) {
 if (low < high) {
 int pi = partition(arr, low, high);
 quickSort(arr, low, pi - 1);
 quickSort(arr, pi + 1, high);
 }
}
int main() {
 int n;
 printf("Enter the number of elements: ");
 scanf("%d", &n);
 int arr[n];
 printf("Enter the elements:\n");
 for (int i = 0; i < n; i++) {
 scanf("%d", &arr[i]);
```

```
}
quickSort(arr, 0, n - 1);
printf("Sorted array: ");
for (int i = 0; i < n; i++) {
printf("%d ", arr[i]);
}
printf("\n");
return 0;
#include <stdio.h>
#include <string.h>
int max(int a, int b) {
return (a > b) ? a : b;
}
int lcs(char *str1, char *str2, int m, int n) {
int L[m + 1][n + 1];
int i, j;
for (i = 0; i <= m; i++) {
for (j = 0; j <= n; j++) {
if (i == 0 || j == 0)
L[i][j] = 0;
else if (str1[i - 1] == str2[j - 1])
L[i][j] = L[i - 1][j - 1] + 1;
else
L[i][j] = max(L[i - 1][j], L[i][j - 1]);
}
}
return L[m][n];
}
int main() {
char str1[100], str2[100];
printf("Enter the first string: ");
scanf("%s", str1);
printf("Enter the second string: ");
scanf("%s", str2);
int m = strlen(str1);
int n = strlen(str2);
printf("Length of LCS is %d\n", lcs(str1, str2, m, n));
return 0;
}
```

Time complexity:

the time complexity of the LCS algorithm in all cases is O(m*n), where m and n are the lengths of the input strings.’

10) 9-Queen Problem

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

int board[20],count;
 void queen(int row,int n);
int main()
{
 int n,i,j;


 printf(" - N Queens Problem Using Backtracking -");
 printf("\n\nEnter number of Queens:");
 scanf("%d",&n);
 queen(1,n);
 return 0;
}
```

```
//function for printing the solution
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
  }
 }
}

/*funtion to check conflicts
If no conflict for desired postion returns 1 otherwise returns 0*/
int place(int row,int column)
{
  printf("r n c are %d%d\n",row,column);
 int i;
 for(i=1;i<=row-1;++i)
 {
  //checking column and digonal conflicts
  if(board[i]==column){
  printf("i m returning zero %d\n",i);
```

```
    return 0;
                          }
   else
    if(abs(board[i]-column)==abs(i-row))
     return 0;
  }
   printf("m rtrning 1 here fr %d \n",i);
  return 1; //no conflicts
 }
```

```
//function to check for proper positioning of queen
void queen(int row,int n)
{
 int column;
 for(column=1;column<=n;++column)
 {
  printf("column in queen is %d\n",column);
  if(place(row,column))
  {
   printf("i m in if %d%d \n",row,column);

   board[row]=column; //no conflicts so place queen
   if(row==n) //dead end
    print(n); //printing the board configuration
   else //try queen with next position
    queen(row+1,n);
  }
 }
printf("end of for wit  %d\n",column);
}
```

Algorithm:

ALGORITHM:

```
N - Queens (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
    N - Queens (k + 1, n);
  }
}
```

11) Naïve String matching

```
#include <stdio.h>
#include <string.h>
int main() {
 char text[1000];
```

```
char pattern[100];
int i, j, text_length, pattern_length, match_count;

printf("Enter text: ");
fgets(text, 1000, stdin);
printf("Enter pattern to search: ");
fgets(pattern, 100, stdin);

text_length = strlen(text);
pattern_length = strlen(pattern);
match_count = 0;

for (i = 0; i <= text_length - pattern_length; i++) {
for (j = 0; j < pattern_length; j++) {
if (text[i + j] != pattern[j])
break;
}
if (j == pattern_length) {
printf("Pattern found at index %d\n", i);
match_count++;
}
}

if (match_count == 0)
printf("Pattern not found in the text.\n");
else
printf("Pattern found %d time(s) in the text.\n", match_count);

return 0;
}
```

The time complexity of the Naive Algorithm is O(mn), where m is the size of the pattern to be searched and n is the size of the container string.

Algorithm:

```
Naive (T[1...n], P[1...m])
{
        For(s=1 to n-m) do
        {
                If(P[1...m] = T[s...s+m]) then
                        Print("Pattern finding with shift", s);
        }
}
```