1) NAIVE STRING :

Algo:

```
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
```

TIME COMPLEXITY: 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.

2) QUICK SORT BY D&C :

Algo:

```
quickSort(arr[], low, high) {
 if (low < high) {
 /* pi is partitioning index, arr[pi] is now at right place */
 pi = partition(arr, low, high);
 quickSort(arr, low, pi – 1); // Before pi
 quickSort(arr, pi + 1, high); // After pi
 }
}
partition (arr[], low, high)
{
 // pivot (Element to be placed at right position)
 pivot = arr[high];
 i = (low – 1) // Index of smaller element and indicates the
 // right position of pivot found so far
 for (j = low; j <= high- 1; j++){
 // If current element is smaller than the pivot
 if (arr[j] < pivot){
 i++; // increment index of smaller element
 swap arr[i] and arr[j]
 }
 }
 swap arr[i + 1] and arr[high])
 return (i + 1)
```

Time Complexity

Best O(n*log n)

Worst O(n2)

Average O(n*log n)

Space Complexity O(log n)

3) SELECTION SORT

Algo:

```
void selectionSort(int arr[], int n)
{
 int i, j, min_idx;
 // One by one move boundary of unsorted subarray
 for (i = 0; i < n-1; i++)
 {
 // Find the minimum element in unsorted array
 min_idx = i;
 for (j = i+1; j < n; j++)
 if (arr[j] < arr[min_idx])
 min_idx = j;
 // Swap the found minimum element with the first element
 if(min_idx != i)
 swap(&arr[min_idx], &arr[i]); }
```

Time Complex:
Best Case n2
Average Case n2
Worst Case n2

4) KNAPSACK BY GREEDY
Algo:

```
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
```

Time Compklexity: O(2n)

5) BELLMAN FORD :
Algo:

```
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[]
```

TIME COMPLEXITY:
O(V * E), where V is the number of vertices in the graph and E is the number of edges in the grap

6) MERGE SORT :
Algo:

```
void merge(int arr[], int l, int m, int r)
{
int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;
```

Thus, time complexity of merge sort algorithm is T(n) = Θ(nlogn).

7) BINARY SEARCH :
Algo:

```
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;
}
```

Time Complexity:
After each comparison the input size decreases by half.
• Best case complexity: O(1)
• Average case complexity: O(log n)
• Worst case complexity: O(log n)

8) INSERTION SORT :
Insertion sort 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
```

Best Case n
Average Case n2
Worst Case n2

9) LCS :
Algo:

```
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])
```

Time Complexity: O(m*n)

10) N QUEENS :
Algorithm for N-Queens Problem using Backtracking
Step 1 - Place the queen row-wise, starting from the left-most cell.

Step 2 - If all queens are placed then return true and print the solution matrix.

Step 3 - Else try all columns in the current row.

Condition 1 - Check if the queen can be placed safely in this column then mark the current cell [Row, Colu mn] in the solution matrix as 1 and try to check the rest of the problem recursively by placing the queen he re leads to a solution or not.

Condition 2 - If placing the queen [Row, Column] can lead to the solution return true and print the solution for each queen's position.

Condition 3 - If placing the queen cannot lead to the solution then unmark this [row, column] in the solutio n matrix as 0, BACKTRACK, and go back to condition 1 to try other rows.

Step 4 - If all the rows have been tried and nothing worked, return false to trigger backtracking.

Time Complexity: O(N!)

11) TRAVELLING SALESMAN PROBLEM :
ALGO :

```
TSP(N,S)
VISITED[S]=1;
if(n=2)and kis not equal to s then
    cost(n,k)=cost(s,k);
    return cost;
else
   for jEn do ( E is belongs to )
      for iEn and visited[i]=0   ( E is belongs to )
          if j is not equal to i and j is not equal to s then
             cost(n,j)=min(tsp(n-{i},j)+cost(j,i))
             visited[j]=1
         end
     end
     end
     return cost
end
```

TIME COMPLEXITY : O(N RAISED TO 2 . 2 RAISED TO N)