Algorithm:

MERGE (A, *p*, *q*, *r*)

```
1.  n1        q- p+1
•   n2        r-q
•   Create arrays L[1…n1+1] and R[1..n2+1]
•   for  i        1 to n1
•      do  L[i]        A[p+i -1]
•   For j        1 to n2
•      do R[j]        A[q+ j]
8.  L[n1+1]        ∞
9.  R[n2+1]        ∞
10. i        1
←11. j   1
•   for k        p to r
•      do if L[i]<= R[j]
•         then  A[k]        L[i ]
```

15.	*i*	*i*+1
16.	Else A[*k*]	R[*j*]
17.	*j*	*j*+1

We implement it so that it takes Θ(n) time, where $n = r - p + 1$, which is the number of elementsbeing merged.

ALGORITHM:

PARTITION (A, p, r)

```
1. $x \leftarrow A[p]$
2. $i \leftarrow p$-1
```

```
3. $j \leftarrow r$+1
•  while TRUE do
•      Repeat $j \leftarrow j$-1
•      until $A[j] \leq x$
•      Repeat $i \leftarrow i$+1
•      until $A[i] \geq x$
•      if $i < j$
•          then exchange $A[i] \leftrightarrow A[j]$
•          else return $j$
```

Partition selects the first key, $A[p]$ as a pivot key about which the array will partitioned:Keys≤ A[p] willbe moved towards the left .

Keys ≥ A[p] will be moved towards the right.

QuickSort (A,p,r)

- If $p < r$ then
- q Partition (A, p, r)
- Recursive call to Quick Sort (A, p, q)
- Recursive call to Quick Sort ($A, q + r, r$)

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 + $w[i] \leq W$
        then $x[i]$ = 1
            weight = weight + $w[i]$
            else
            $x[i]$ = ($w$ - weight) / $w[i]$
            weight = $W$
return $x$
```

Analysis: If the items are already sorted into decreasing order of v_i / w_i, then the while-loop takesa time in *O(n)*. Therefore, the total time including the sort is in *O(n log n)*.

KRUSKAL ALGORITHM:

Start with an empty set A, and select at every stage the shortest edge that has not been chosen or rejected, regardless of where this edge is situated in the graph.

```
KRUSKAL(G):
A = ∅
foreach v ∈ G.V:
  MAKE-SET(v)
foreach (u, v) ordered by weight(u, v), increasing:
   if FIND-SET(u) ≠ FIND-SET(v):
   A = A U {(u, v)}
      UNION(u, v)
return A
```

- Make_SET(v): Create a new set whose only member is pointed to by v. Note that forthis operation v must already be in a set.

PRIMS ALGORITHM:

- Initialize a tree with a single vertex, chosen arbitrarily from the graph.
- Grow the tree by one edge: Of the edges that connect the tree to vertices not yet in thetree, find the minimum-weight edge, and transfer it to the tree.
- Repeat step 2 (until all vertices are in the tree).

```
MST-PRIM($G, w, r$)
for each $u \in V[G]$
    do $key[u] \leftarrow \infty$
       $\pi[u] \leftarrow$ NIL
$key[r] \leftarrow 0$
$Q \leftarrow V[G]$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN($Q$)
       for each $v \in Adj[u]$
           do if $v \in Q$ and $w(u, v) < key[v]$
                then $\pi[v] \leftarrow u$
                     $key[v] \leftarrow w(u, v)$
```

Time Complexity: A simple implementation using an adjacency matrix graph representation and searching an array of weights to find the minimum weight edge to add requires $O(V^2)$ running time. Using a simple binary heap data structure and an adjacency list representation, Prim's algorithm can be shown to run in time $O(E \log V)$ where E is the number of edges and Vis the number of vertices.

ALGORITHM:

Algorithm All pair shortest path:

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each vertex $v$
   dist[$v$][$v$] ← 0
for each edge $(u,v)$
   dist[$u$][$v$] ← w($u$,$v$)  // the weight of the edge (u,v)
for $k$ from 1 to |V|
   for $i$ from 1 to |V|
     for $j$ from 1 to |V|
       if dist[$i$][$j$] > dist[$i$][$k$] + dist[$k$][$j$]
          dist[$i$][$j$] ← dist[$i$][$k$] + dist[$k$][$j$]
       end if
```

This is an $O(n^3)$ algorithm, where n is the number of vertices in the digraph.

ALGORITHM:

Iterative LCS:

```
int lcs_length(char * A, char * B)
{
        allocate storage for array L;
        for (i = m; i >= 0; i--)
            for (j = n; j >= 0; j--)
            {
                    if (A[i] == '\0' || B[j] == '\0') L[i,j] = 0;
                    else if (A[i] == B[j]) L[i,j] = 1 + L[i+1, j+1];
                    else L[i,j] = max(L[i+1, j], L[i, j+1]);
            }
        return L[0,0];
}
```

The subsequence itself

What if you want the subsequence itself, and not just its length? This is important for some but not all of the applications we mentioned. Once we have filled in the array L described above, we can find the sequence by working forwards through the array.

```
sequence S = empty;
i = 0;
j = 0;
while (i < m && j < n)
```

```
{
        if (A[i]==B[j])
        {
           add A[i] to end of S;
           i++; j++;
        }
        else if (L[i+1,j] >= L[i,j+1]) i++;
        else j++;
}
```

So we can find longest common subsequences in time O(mn).

ALGORITHM:

Iterative LCS:

```
int lcs_length(char * A, char * B)
{
        allocate storage for array L;
        for (i = m; i >= 0; i--)
           for (j = n; j >= 0; j--)
           {
                if (A[i] == '\0' || B[j] == '\0') L[i,j] = 0;
                else if (A[i] == B[j]) L[i,j] = 1 + L[i+1, j+1];
                else L[i,j] = max(L[i+1, j], L[i, j+1]);
           }
        return L[0,0];
}
```

Advantages of this method include the fact that iteration is usually faster than recursion, we don't need to initialize the matrix to all -1's, and we save three if statements per iteration since we don't need to test whether L[i, j], L[i+1,j], and L[i,j+1] have already been computed (we know in advance that the answers will be no, yes, and yes). One disadvantage over memorizingis that this fills in the entire array even when it might be possible to solve the problem by lookingat only a fraction of the array's cells.

The subsequence itself

What if you want the subsequence itself, and not just its length? This is important for some but not all of the applications we mentioned. Once we have filled in the array L described above, we can find the sequence by working forwards through the array.

```
sequence S = empty;
i = 0;
j = 0;
while (i < m && j < n)
```

```
{
        if (A[i]==B[j])
        {
           add A[i] to end of S;
           i++; j++;
        }
        else if (L[i+1,j] >= L[i,j+1]) i++;
        else j++;
}
```

So we can find longest common subsequences in time O(mn).

NAIVE ALGORITHM:

Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

```
while there are untried conflagrations
{
  generate the next configuration
  if queens don't attack in this configuration then
  {
    print this configuration;
  }
}
```

BACKTRACKINGALGORITHM:

```
Algorithm N_QUEEN (k,n)
for i= 1 to n do
      if PLACE (k,i)then
      x[k]=i
      if k==n then
      print X[1…n]
      else
      N_QUEEN(K+1,n)
   end
end
```

```
Function PLACE (K,i)
//k is the number of queen being processed
//i is the number of columns
 for j=1 to k-1 do
 if x[j]==i then
 return false
     end
end
return true
```

COMPLEXITY ANALYSIS:

The recurrence of n-Queen problem is defined as $T(n)=n*T(n-1)+n^2$. Solution to recurrence would be O(n!)

NAIVEALGORITHM:

Generate all possible configurations of colors and print a configuration that satisfies the given constraints.

```
while there are untried conflagrations
{
  generate the next configuration
  if no adjacent vertices are colored with same color
  {
    print this configuration;
  }
}
```

There will be v^m configurations of colors.

BACKTRACKINGALGORITHM:

The idea is to assign colors one by one to different vertices, starting from the vertex 0. Before assigning a color, we check for safety by considering already assigned colors to the adjacent vertices. If we find a color assignment which is safe, we mark the color assignment as part of solution. If we do not a find color due to clashes then we backtrack and return false.

ALGORITHM:

GRAPH_COLORING (G, COLOR, i)

```
If CHECK_ VERTEX(i)== i then
          if i== N then
          print COLOR[1……n]
          else
          j=1
         while (j<=M)do
         COLOR (i+1) = j
         j= j+1
        end
        end
end
Function CHECK_VERTEX (i)
for j=1 to i-1 do
      if Adjacent (i,j) then
          if COLOR(i)= = COLOR(j) then
      return 0
      end
     end
end
return 1
```

ALGORITHM:

NAIVE _STRING_MATCHER (T,P)

```
n= length[T]
m= length[P]
  for s= 0 to n-m do
   if P[1….m]= T[(s+1)…..(s+m)]  then
    print "pattern occurs with shift s"
  end
end
```

ANALYSIS: In best case searchable text does not contain any of prefix of pattern .only one comparison requires moving pattern one position right. Worst case occurs when pattern is at last position and there are spurious hits all the ways.

ALGORITHM:

```
//T is the text of length n
// P is the pattern of length m
Rabin karp (string T[1…n], string P[1….m])
   hash T = hash (T[1…n])
   hash P = hash (P[1…m])
     for i=1 to n-m+1
```

```
          if hash T= hash P then
             if P[1…m]=T[ s+1……s+m] then
          else
    print match found at shift S
    match not found
```

ANALYSIS: In most cases it runs in linear time ie: O(n) and worst case is O (mn) it can happen only when prime no. used for hashing is very small.