TIME COMPLEXITIES:

Algorithms	Time Complexity
Insertion,Selection	$O(n^2)$,$O(n)$
Binary Search	$O(\log_2 n)$,$O(1)$
Max Min	$O(n)$
Merge Sort	$O(n\log_2 n)$
Quick Sort	$O(n\log_2 n)$, $O(n^2)$
Knapsack	$O(n\log_2 n)$
Job Sequencing	$O(n^2)$
Minimum cost spanning tree- Kruskal	O(elogv)
Minimum cost spanning tree-prim	$O(v^2)$
Optimal Storage on Tapes	$O(n^2)$
Single source shortest Path	$O(v^2)$
Multistage graphs	O(NK)
All pair shortest path	$O(n^3)$
Single source shortest Path(Dynamic)	$O(n^3)$
Optimal Binary Search Tree	$O(n^2)$
0/1 Knapsack	O(n*m)
Travelling Salesperson problem	$O(n*2^n)$
Matrix Chain Multiplication	$O(n^3)$
N queen	O(n!)
Sum of subsets	O(p(n)*n!)
Graph coloring	$O(v^m)$
15 puzzle problem	$O(b^n d)$
Naïve string matching	O(nm)
String Matching with finite automata	O(n)
Least common subsequence	O(nm)

Introduction to Analysis of Algorithms

1. Insertion Sort

```
INSERTION_SORT (A,n)

   FOR j ← 2 TO length[A]
        DO  key ← A[j]
            {Put A[j] into the sorted sequence A[1 . . j − 1]}
            i ← j − 1
            WHILE i > 0 and A[i] > key
                    DO A[i +1] ← A[i]
                         i ← i − 1
            A[i + 1] ← key
```

2. Selection Sort

```
SELECTION_SORT (A,n)

            FOR i ← 1 TO n-1 DO
              min j ← i;
              min x ← A[i]
              FOR j ← i + 1 to n do
                 IF A[j] < min x then
                      min j ←
                      min x ← A[j]
              A[min j] ← A [i]
              A[i] ← min x
```

Divide and Conquer

1. Binary Search

```
int binary_search(int A[], int key, int imin, int imax)
{

 IF (imax < imin)
  RETURN KEY_NOT_FOUND;
 ELSE      {
           // calculate midpoint to cut set in half
           int imid = midpoint(imin, imax);
            // three-way comparison
           IF (A[imid] > key)
               // key is in lower subset
               RETURN binary_search(A, key, imin, imid-1);
           ELSE IF (A[imid] < key)
               // key is in upper subset
               RETURN binary_search(A, key, imid+1, imax);
           ELSE
               // key has been found
               RETURN imid;
         }
}
```

2. Max Min

```
 VOID  StraightMaxMin (Type a[], int n, Type& max, Type& min)
{   max = min = a[1];

FOR (int i=2; i<=n; i++){

IF (a[i]>max)  then max = a[i];

IF (a[i]<min) min = a[i];
}

}
VOID MaxMin(int i, int j, Type& max, Type& min)

    {
      IF (i == j) max = min = a[i]; // Small(P)
      ELSE IF (i == j-1) { // Another case of Small(P)
            IF (a[i] < a[j])
                  max = a[j]; min = a[i];
            ELSE { max = a[i]; min = a[j];
            }
        ELSE {     Type max1, min1;
 // If P is not small   divide P into subproblems.  Find where to split the set.
           int mid=(i+j)/2;
          // Solve the  subproblems.
           MaxMin(i, mid, max, min);
          MaxMin(mid+1, j, max1, min1);
        // Combine the solutions.
          IF (max < max1) max = max1;
          IF (min > min1) min = min1;

   }
 }
```

3. Quick Sort

```
void quicksort( Integer A[ ], Integer left, Integer right)
{ IF ( left < right ) then
        {       q = partition( A, left, right);
                quicksort( A, left, q–1);
                quicksort( A, q+1, right);

        }

}

Integer partition( integer AT[], Integer left, Integer right)
{
pivot = A[left]; lo = left+1; hi = right;
WHILE ( lo ≤ hi )
{       WHILE ( A[hi] > pivot )                         hi = hi – 1;
        WHILE ( lo ≤ hi and A[lo] <~pivot )     lo = lo + 1;
        IF ( lo ≤ hi ) then                             swap( A[lo], A[hi]);
}
swap( pivot, A[hi]);
 RETURN hi;

}
```

4. Merge Sort

```
MERGE-SORT (A, p, r)

   IF p < r                                  // Check for base case
     THEN q = FLOOR[(p + r)/2]                 // Divide step
          MERGE (A, p, q)                    // Conquer step.
          MERGE (A, q + 1, r)                // Conquer step.
          MERGE (A, p, q, r)                 // Conquer step.

MERGE (A, p, q, r )
{
   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 ]
   L[n1 + 1] ← ∞
   R[n2 + 1] ← ∞
  i ← 1
  j ← 1
  FOR k ← p TO r
      DO IF L[i ] ≤ R[ j]
           THEN A[k] ← L[i]
                i ← i + 1
           ELSE A[k] ← R[j]
                j ← j + 1

}
```

Greedy Method

1. Greedy Knapsack

Maximize $\sum p_i x_i$ where (i=0 to i=n)

Subject to $\sum w_i x_i <= M$

Where M=Maximum Weight and i=0 to i=n

Dynamic Programming

1. Multistage Graphs

Recursive Formula:

Forward computation

$$\text{cost}(i,j) = \begin{cases} C(j,T), & if\ i == k-1\ (second\ last\ stage\ in\ the\ graph) \\ \min\ \{c(j,l) + \text{cost}(i+1,l)\} & \\ where\ l \in Vi-1\ and\ <l,j> \in E\ ,for\ all\ such\ values\ of\ l & \end{cases}$$

Forward computation

$$\text{bcost}(i,j) = \begin{cases} C(S,j), & if\ i == 2\ (second\ stage\ in\ the\ graph) \\ \min\ \{\text{bcost}(l,j) + c(i-1,l)\} & \\ where\ l \in Vi+1\ and\ <j,l> \in E\ ,for\ all\ such\ values\ of\ l & \end{cases}$$

2. Single Source Shortest Path

$Dist^{k}[u]=\min\{ dist^{k-1}[u], \min l \{dist^{k}[l]+cost[i,u]\}\}$

for k=2,3,....n-1

3. All Pair Shortest Path

Recursive Formula

$A^k[i,j] = 0$ IF $(i==j)$

$A^k[i,j] = \min\{ A^{k-1}[i,k]+A^{k-1}[k,j]\}, A^{k-1}[i,j]\}$, otherwise.

4.0/1 Knapsack

Recursive formula:

$$B(i,w) = \begin{cases} 0, if \ \ i = 0 \ or \ j = 0 \\ B[i-1,w], if \ \ wi > w \\ \max\{B[i-1,w], bi + B[i-1,w-wi\} \ Otherwise \end{cases}$$

5. Travelling Salesperson Problem

Recursive Formula:

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{C1k + g(k, V - \{1, k\})\} \quad (1)$$

Generalizing (1), we obtain (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{ c_{ij} + g(j, S - \{j\})\} \quad (2)$$

6. Optimal Binary Search Tree

Recursive Formula:

Cost of tree= cost of left subtree + cost of right subtree + weight of root

$$c(i, j) = \min_{i < k \le j} \{c(i, k-1) + c(k, j)\} + w(i, j)$$

7. Matrix Chain Multiplication

$m[i,j]=\{ 0, \text{if } i=j,$

$\min(m[i,k]+m[k+1,j]+P_{i-1}.P_j.P_k), \text{otherwise}$

$\}$

Backtracking

1. N Queen's Problem

Backtracking Condition

If ((X[j]=i) OR(Abs(X[j] – i) == Abs(j-k))) // Two queens in same column or same diagonal
//then backtrack

2. Sum of Subsets

Explicit Condition: A vector X={x1,x2…Xn} for all n elements in the set where Xi=0 (element not added) or Xi=1 (element added in the solution tuple)

Implicit Condition: summation of the chosen numbers must be equal to given number M and one number can be used only once.

Backtracking Condition:

$$B_k(x_1, \ldots, x_k) = true \text{ iff } \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

$$\text{and } \sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$$

3. Graph Coloring

Explicit Condition: A vector X={x1,x2…Xn} for all n vertices for all possible combinations of colors

Implicit Condition: No two adjacent vertices or regions have the same color.

Backtracking Condition: If ((k,i) is an edge) and (Color[i]=Color[k]) then Backtrack!

String Matching Algorithms

1. Longest Common Subsequence

Recursive Formula:

$$C[i,j] = \begin{cases} 0 & if\ i = 0\ or\ j = 0 \\ C[i-1, j-1] + 1 & if\ i,j > 0\ and\ Xi == Yj \\ max\{C[i-1,j], C[i,j-1]\} & otherwise \end{cases}$$