Date

# Practical No. 1.(A)

* Aim :- Write a program to perform linear search technique and also analyse its time complexity.

* Theory :

Linear Search :-

. The linear search technique also known as sequential search is a straight forward searching algorithm that checks every element in a collection sequentially until the desired element is found or all ~~a~~elements have been checked.

Algorithm :-

Step for algorithm

step① :- Start from the beginning of the Array (index = 0')

step② :- Iterate over each element of the array.

. compare the current element with the 'target'.

Date :

- if the current element is equal to the 'target', return the current index.

step③: if the 'target' is not found after iterating through all elements return '--'.

* Complexity :-

① Best case :

→ If the target element is found at the beginning of the array. the function will return after checked the first element, in this case the time complexity of O(1).

② Worst case :

→ If the target element is at the end of the array or is not present in the array at all, the function will iterate all n elements therefore it is the worst case time complexity of O(n).

③ Average case :-

→ On Average case if the target element is uniformly distributed throughout the array the function will need to check approximately n/2 elements. This the average case complexity is also also O(n).

* Result :- The program for linear search is executed successfully & time complexity is O(n).

* Output :-

Element found at index : 6.

Best case - 1, as it found at index 3
Average case :- 5, as it at middle
worst case :- 8, as it is last index.

* Result :-

Hence, program for linear search technique and its complexity analysis has been done successfully.

Date

# Practical No. 1. (B)

* Aim :- Write a program to perform binary search technique and also analyse its time complexity.

* Theory :-

# Binary Search :-

→ The Binary Search is an efficient algo. for finding an item from a sorted list of items. It work by repeatedly dividing the search interval in half.

* Algorithm :-

Step① :- Initialization
- set low to '0'
- Set high to 'n-1'

Step② :- While low is less then or equal to high.

- Calculate middle index
  mid = (low + high)/2

- Compare 'arr[mid]' with target
  if 'arr[mid] == Target
  return mid.

if 'arr [mid] = target
set low to mid + 1
if arr [mid] > target
set high to mid -1

Step3:- End of while loop
if target not found
return (-1)

* Time complexity :-

→ Time complexity of binary search is O(logn) because the search interval is halved with each iteration.

① Best case :-

→ It occurs when target element is found at middle of an first check.

∴ Time complexity = O(1)

② Worst case :-

→ It occurs when target element is not in array or located at first or end of the array.
it takes $\log_2 n$ comparison [O(log n)]

Date :

③ Average case :-

→ Consider scenario where the target element can be located at any position with equal probability.

- will also be proportional to $\log_2 n$ [O (log n)]

* Result :- The program for binary search is executed successfully and the time complexity is [O(log n)].

* output :-

Element 7 found at index 3

Best case - 4, as it at middle
Average case - 7, as it will found after few iteration
Worst case :- 9, as it is least element

* Result :-

Hence, program for binary search technique and its time complexity its analysis has been done successfully.

# Practical No. 2 (A)

* Aim :- Write a program to perform insertion sort techniques and also analyse its time complexity.

* Theory :-

Insertion sort is a simple sorting algorithm that works by iteratively inserting each elements of an unsorted list into its correct position in a sorted position of the list.

* Algorithm :-

- Start with second element of the array.
- Compare current element with ones before it and insert it into its correct position among the previosly sorted elements.
- Repeat this process for all elements until the entire array is sorted.

# Time complexity

- Best case [O(n)]
- happens when array is already sorted.

- algorithm only makes (n-1) comparisons.

* Average case (O[n^2])
  - this happens when element are in random order.

- Worst case [O(n^2)]
- this occurs when array is sorted in reverse order.
- Each insertion requires shifting all previously sorted elements.

* Output :-

Sorted Array in Ascending order :-
[1, 3, 4, 5, 9]

* Result :-

Hence, insertion sort technique is applied and executed successfully with analysing its time complexity.

Date

# Practical No. 2 (B)

* Aim :- Write a program to perform selection sort technique and also analyse its time complexity.

* Theory :-

- Selection sort is a straight forward comparision based sorting algorithm.
- It works by dividing the input list into two parts.
- Algorithm proceeds by finding smallest element from unsorted list, exchanging it with leftmost unsorted element and moving the sublist boundaries one element to right.

* Algorithm:-

Start with first element of the list.

find smallest element in sorted partion of the list.

swap this smallest element with first unsorted element.

Move boundary of sorted & unsorted sublists one elements to the right.

Repeat the process until the entire list is sorted.

# Time complexity.

Best case [O(n^2)]
Average case [O(n^2)]
Worst case [O(n^2)]

Time complexity of selection sort is [O(n^2)] for all cases- because there are two nested loops : center loops run (n) times, and inner loop runs approximately (n-i) times.

* Outputs.

Sorted Array in Ascending Order:
[-9, -2, 0, 11, 45]

* Result :-

Hence, selection sort technique is applied and executed successfully with analysing its time complexity.

Date

# Practical NO. 3

* Aim :- Implementation various operation on Divide and conquer analyze its complexity.
  a) Quick sort  b) Merge sort
  c) Minimax.

* Theory :-

# Divide and conquer Algorithm :

→ Divide and conquer Algorithm is a problem solving technique used to solve problems by dividing the main problem into subproblems, solving them individually and then merging them to find solution to the original problem.

1. Divide : Split the original problem into smaller subproblems.
2. Conquer : Solve each subproblem recursively.
3. Combine : Merge the solution of the subproblems to solve original problem

## 1] Quick Sort :

→ Quick sort is a sorting algorithm based on the divide and conquer algorithm that picks an element as a pivot and partitions the given array around in correct position in sorted array.

Date

* Algorithm :-

```
Quicksort ( A, P, E ) {
  if ( P < r ) {
    q = Partition (A, P, r)
    quicksort ( A, P, q-1)
    quicksort ( A, q+1, r )
  }}
partition ( A, P, r) {
  x = A [r]
  i = P-1
  for ( j = P to r-1 ) {
    if (A [i] ≤ x ) {
      i = i + 1
      exchange a [i] with a [i]
  }}
  exchange a[i + 1] with a[r]
  return i+1
}
```

* Quicksort Complexity :

1] Time Complexity -

1) Best Case Complexity : $O(n * \log n)$
2) Average Case Complexity : $O(n * \log n)$
3) Worst Case complexity : $O(n^2)$

2] Space complexity :- $O(n * \log n)$

Date

## 2] Merge sort :-

→ Merge sort is a sorting algorithm that follows divide & conquer approach. It works recursively dividing the input array into smaller subarrays & sorting those subarrays then merging them back together to obtain them back together to obtain sorted array.

### * Algorithm :-

```
Merge - sort (arr, beg, end)

if beg < end
set mid = (beg + end) /2
merge - sort (arr, beg, mid)
merge - sort (arr, mid + 1, end)
merge (arr, beg, mid, end)
end of it.
void merge (int a, int beg, int mid, int end){
int i,j,k;
int n1 = mid - beg + 1;
int n2 = end - mid;
int leftArray[n1], RightArray[n2];
for (int i=0; i<n1; i++)
leftArray[i] = a[beg+i];
for (int j=0; j<=n2; j++)
RightArray[j] = a[mid+1+j];
i=0,
j=0;
K= beg;
```

```
while (i < n1 && j < n2) {
if (LeftArray[i] <= RightArray[j]) {
    a[k] = LeftArray[i];
    i++; }
  else {
    a[k] = RightArray[j];
     j++;
  } k++;
  }
  while (i < n1) {
   a[k] = LeftArray[i];
     i++;
     k++; }
  while (j < n2)
  { a[k] = RightArray[j];
    j++;
    k++;
   }
}
```

* Merge sort Complexity:

1] Time Complexity :

1) Best Case : O(n * Log n)
2) Average Case : O(n * log n)
3) Worst Case : O(n * Log n)

2] Space complexity : O(n)

3] MinMax Algorithm:

- MinMax is a kind of back tracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe.

* Algorithm:

```
func" minmax (node, depth, maximizing_player):
if is_terminal == 0 :
    return
utility (state) If maximize_player :
    max_eval = -infinity
    for action in
action (state) : eval = mimax, depth 1, false)
    max_eval = max (max_eval, eval)
    return max_eval
    else: min_eval = infinity
        for action in actions (state):
            eval = minmax (result, depth-1, True)
minmax_eval = min (min_eval, eval)
        return min_eval
```

Minmax Complexity:

1) Time Complexity :- $O(b^m)$
2) space complexity :- $O(bm)$

Result :- Hence a program executed successfully.

# Practical No. 4

* Aim: Implement various operations on Greedy strategy analyse its complexity.
  1. Prim's Algorithm
  2. Kruskal's Algorithm

* Theory:

* Greedy Algorithm:

→ A greedy algorithm is an approach to solving problems by choosing the best option available at each step without considering the global context. This method ensures a locally optimal solution, hoping it leads to a globally optimal solut^n. greedy algorithms are often simple & efficient but don't always guarantee the best overall outcome.

* Minimum Spanning Tree (MST):

A minimum spanning tree (MST) is a subset of edges in a weighted, connected graph that connects all vertices with the minimum possible total edge weight & no cycles. MST can be found using greedy algorithms like Algorithm, by selecting the smallest available edge that doesn't form a cycle

Date

1] Prim's Algorithm:

The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it consider all the edges that two sets & picks the minimum weight edge from these edges.

# Algorithm:

```
PRIM ( cost, near, n, mincost, T ) [
  // find the entry cost [k, l] which
  represent, minimum value in the matrix.

step 1:
T [1, 1] = k;
T [1, 2] = l;
T [1, 3] = cost [k, l];
min_cost = cost [k, l]
for i = 1 to n do {
if cost [k, l] < cost [l, i]
                near [i] = k
else
    near [i] = l
}
    near [k] = 0;
    near [l] = 0;
```

Date

step 2:

```
for i = 2 to n-1 do {
function ; find an index j in array
          near such that near [i] != 0
{
   cost [ j , near [j] should be minimum
   T [ i, e] , near [j],
   P [i , 2 ] = j
   T [i , 3 ]  = cost [ j , near [ j ] )
mincost = mincost + T [1 , 3 ];
near [ j ] = 0 ;
for    k = ( 1 for ) do {
  if (near [k ]! = 0 )
{
  if (near [k ] != 0 )
  }
and
cost (k , near [k, i ] > Cost [k, j ]
near [k ] = j .
  }
}}
```

2] Kruskal's Algorithm:

In kruskal's algorithm, sort all edges & nodes in the MST. It newly added edges doesn't form a cycle. It picks the minimum weighted edges at last. in each step in order to find the optimal.

* Algorithm :

In

```
Algorithm- Kruskal ( cost, n : T ) {
  T = NULL // no entry
  do {
     Select minimum cost edge
     e = { u, v }
     u = cost [ 1 , 1 ]
     v = cost [ 1 , 2 ]

  x = find (u)
  y = find (v)
  if ( x ≠ y )
  {  T = T ∪ e
     merge (x, y)
  }
  } while ( T contain not vertices )
  } // end of algorithm
```

* How to find MST using Krushal's algorithm

step 1: Sort all edges in non-decreasing order of their weight.

Step 2: Pick the smallest edge. Check if it forms a cycle with spanning tree formed so far. if the cycle, is not formed: include this edge. Else, discard it.

Step 3: Repeat Step# 2 until these are (V-1) edges in the spanning tree

Date

# complexity :

1] Prims Algorithm :

| Implemented method | Time complexity |
|---|---|
| Adjacency Matrix + Array | $O(v^2)$ |
| Adjacency list + Binary heap | $O((v+E)\log V)$ |
| Adjacency list + fibonacci heap | $O(E + v \log v)$ |

* space complexity : $O(V+E)$

2] Kruskal's Algorithm :

Time complexity :

① overall time complexity : $O(E \log E + E, a(v))$ since $\alpha(v)$ grows extremely slowly & is almost constant for all reasonable input.

② simplified time complexity : $O(E \log E)$

③ Alternative form : $O(E \log v)$

* space complexity : $O(V+E)$

* Result : Hence a program executed successfully.

Date

# Practical No: 5.

* Aim:- Implement various operation on Greedy strategy its complexity.
① Knapsack Problem ② Job sequencing problem

* Theory:-

* Greedy Algorithm:

A greedy Algorithm is an approach to solving problems by choosing the best option available content. This method ensures a locally optimal solution greedy algorithms are often simple and efficient but don't always guarantee the best overall outcome.

1] Knapsack problem:

Given N items, where each item has some weight & profit associated with it & also given a bag with capacity W. The task is to put the items into the bag such that the sum of profits associated with them the maximum possible.

* Algorithm:

Algorithm - knapsack (w[1..n], P[1..n], w, n, x[1..n], profit) I

step1: for i:=1 to n do, x[i] = 0, w = 0.

step2:- while (weight < w) do

else {
x [i] = $\frac{w - weight}{w[i]}$

weight = w
}
} // end while

step3: profit = 0
for i = 1 to n do
profit = profit (x [i] * P[i])
} // end of profit.

* Steps to solve problem:

- Case 1 (includes the $N^{th}$ item): value of the $N^{th}$ item plus maximum value obtained by remaining N-1 items & remaining weight.

- case 2 (excludes $N^{th}$ item): Maximum value obtained by N-1 items & W weight.

- If the weight of the $N^{th}$ item is greater than 'w'. then the $N^{th}$ item can't be included & case 2 is the only possibility.

Date :

1] Job scheduling (sequencing) Algorithm:

It is an given array of jobs where every job has a deadline & associated profit if the job is finished before deadline.
→ find a time slot, is empty & deadline is gre
→ If no such exists, then ignore the job

* Complexity :-

1] Knapsack Problem :

→ Time Complexity : O (n log n)
→ Space complexity : O (1).

2] Job sequencing Algorithm :

→ Time Complexity : O (N log N)
→ Auxilliary space : O (N).

* Result :- Hence a program executed successfully.

Date

# Practical NO.6

* Aim:- Implement various operation on Dynamic programming & analyze its complexity. ① All pair shortest path.

* Theory:

* Dynamic programming:

is a powerful technique for solving optimization problems by breaking them down into simpler subproblems. The out-pair: shortest path (ASP's).

* floyd-Warshall Algorithm (All pair S.P).

Given a weighted grap $G = (V, E)$ with vertices V & edges E, where of an edges is denoted by $w(u, v)$.

* Algorithm steps:

1. Initialization: create a distance matrix dist where dist[i][j] is initialized to:
   - 0 if $i = j$.
   - $w[i, j]$.
   - (initialise).

2. Iterative update: for each intermediate vertex, update distance matrix dist.

Date

dist[i][j] = min (dist[i][j], dist[i][k] + dist[k][j])

This formula checks whether a shortest path exists from vertex i to vertex j via vertex k & updates dist[i][j] accordingly.

* Algorithm:

```
def floyd_warshall(graph):
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]

    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            elif graph[i][j] != 0:
                dist[i][j] = graph[i][j]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist
```

Date :

* complexity Analysis :-

- Time complexity : $O(V^3)$
  (Due to three nested loops iterating over vertices V)

- Space Complexity : $O(V^2)$
  (To store the dist matrix for V vertices)

* Result :- Hence a program executed successfully.

Date

# Practical NO. 7.

* Aim:- Implement multistage graph algorithm also analyse its complexity.
a) forward Algorithm b) Backward Algorithm.

* Theory:-

The multistage graph problem is a type of optimization problem in graph theory where a directed graph is divided into multiple stages, and the goal is to find the shortest (or longest) path from a source node in the first stage to a distination node in the last stage. A multistage graph is structured such that nodes are grouped into stages, and edge exist only between nodes of consecutive stages.

1] forward Algorithm (Dynamic programming):
- The forward approach solves the multistage graph problem by moving from the source node towards the destination. It incrementally computes the shortest path to all nodes in each stage, stage by stage.

- set the distance from the source code to itself as 0. and to all others nodes as infinity.
- for each starting from the source node to itself as 0 compute the minimum

Date

distance for each node in the next stage based on the current stages nodes.

* Algorithm :

```
def forward_approach (graph, source, destination):
    dist = [infinity] * number_of_nodes
    dist [source] = 0

    for stage in stages:
        for u in stage:
            for (v, cost) in neighbours [u]:
                dist [v] = min (dist[v], dist
                                [u] + cost)

    return dist [destination].
```

2] Backward Algorithm :

- The approach works in reverse, starting from the destination node & working backward to the source node.

- set the distance to the destinat^n node as 0, and to all other nodes as infinity.

- starting from the last stage (destinat^n) calculate the minimum distance for each node in the previous stage.

Date :

* Algorithm:

```
def backward_Algorithm(graph, source, destination):
    dist = [infinity] + number_of_nodes
    dist[destination] = 0

    for stage in reversed(stages):
        for u in stage:
            for (v, cost) in neighbors[u]:
                dist[u] = min(dist[u], dist[v]
                              + cost)

    return dist[source]
```

# Time complexity:

1) forward Algo : O(V+E), where
   V is the no. of vertices
   E is the no. of edges

2) backward Algo : O(V+E)

* Result :- Program executed successfully.

Date :

# Practical NO. 8.

* Aim :- Implement various operation in travelling salesman problem & analyze its complexity.

* Theory:

- The travelling salesman Problem (TSP) is a classic optimization problem where a salesman must visit a list of cities exactly once, starting & ending at a specific city, while minimizing the total travel cost or distance.

* Algorithm :

1. Input :
   - n : number of cities.
   - dist[][] : Distance matrix (dist[i][j] is the cost to travel from city i to j.

2. Initialization :
   - create dp[2^n][n] initialized to inf.
   - set dp[1][0] = 0 (start from city 0).

3. State Transition :
   - for each subset of cities.
   - for each city i in the subset :
   - for each city j in the subset, update :

Date :

4. Final calculation:

- Compute the minimum cost of visiting all cities & returning to city 0:

min_cost = min (dp [(1 << n) - 1] [i] + dist [i][0] for i in range (1, n)).

5. Output:

Return min_cost as the shortest path.

* Complexity:

- Time Complexity : $O(n^2 * 2^n)$

- Space complexity : $O(n * 2^n)$

* Result :- Hence a program executed successfully.

Date :

# Practical No. 9.

* Aim :- Implement various operation on Matrix chain Multiplication and Analyze its complexity.

* Theory :

- Matrix chain Multiplication (MCM) is an optimization problem in data design and algorithms where the goal is to find the most efficient way to multiply a sequence of matrics. The efficiency is measured in terms of the number of scalar multiplication required. Matrix Multiplication is associative, which means that the order of multiplicat$^n$ can be e rearranged without changing the result, but the cost of multiplying the matrics can very greatly depending on the parethesization.

- The Matrix chain Multiplication problem can be solved efficiently using dynamic programing.

* Algorithm :

```
def matrixChainOrder(P):
    n = len(P) - 1    # number of matrices
    m = [[0 for _ in range(n)] for _ in range(n)]
```

Date :

# m[i][i] is zero since multiplying one mateise costs 0.

```
for L in range (2, n+1):
    for i in range (n-L+1):
        j = i+L-1
        m[i][j] = float('inf')
        for k in range (i, j):
            q = m[i][k] + m[k+1][j] + p[i]
            * p[k+1] * p[j+1]
            if q < m[i][j]:
                m[i][j] = q
return m[0][n-1]
```

```
def print Optimal parens (s, i, j):
    if * i == j:
        print (f "A {i+1}", end = " ")
    else:
        print (" (", end = " ")
        print Optimal Parens (s, i, s[i][j])
        print Optimal Parens (s, s[i][j] +1, j)
        print (" ) ", end = " ")
```

* Complexity :

O : Time complexity : $O(n^3)$

O space complexity : $O(n^2)$

* Result :- Hence a program executed successfully.

Date :

# Practical No. 10.

* Aim :- Implement various operation on Backtracking and analyze its complexity.
a) N-queen problem
b) Graph colouring.

* Theory :

a] N-queen Problem :

- The N-queens problem is a classic combinatorial problem where the goal is to place N queens on an N x N chess board such that no two queens attacks each others. In short, queen can attack another queen if they are in the same row, column, or diagonal.

* Backtracking Approach :
- Backtracking is a systematic way of trying out different possibilities & discarding those that doesn't satisfy the constraints (in this case, no two queens attacking each others).

* Steps :

1. Start with an empty board.

2. Place queens row by row, & starting

Date :

from the ~~th~~ first raw.

3. for each raw, try placing a queen a queen in one column at a time. After placing the queen, move to the next raw.

4. check if placing the queen is valid.

5. If placing the queen results in a conflict, back track by removing the queen & trying the next columns.

6. If a valid position is found for all queens, print the solution.

7. Repeat until all possible solutions are found.

* Complexity :

- Time complexity : $O(N!)$ because for each queen, we try placing it in all columns & check for valid placements recursively.

- space Complexity : $O(N^2)$ due to the board being stored in a 2D list.

Date :

b] Grap' colouring :

- Graph colouring is the process of assigning colors to the vertices of a graph such that no. two adjacent vertices share the same colour. The minimum number of colors needed to achieve this is called the graphics chromatic number.

* Approach :

- The backtracking method explors all possible color assigments for the vertices, backtracking when a violation of the coloring condition occurs.

* Steps :

1. Initialization :
   - create an adjacency list as matrix.
   - create an array color[] to store colours.

2. Check validity :
   - Define isSafe (vertex, color, c) to check if color c can be assigned to vertex without conflicts with adjacent vertices.

3. Backtracking function:
   - Define graph_colouring (vertex):
     - If all vertices are coloured (vertex == number_of_vertical, return True,

Date :

- For each color from 1 to m:
  - If is_safe is True for vertex and color:
    - Assign the colour & call graph_colouring (vertex + 1)
    - If the recursive call return True, return True.
    - Backtrack by removing the color assignment if needed.

4. Main function:
   - Call graph_colouring (0) to start from the first vertex.
   - Check if the colouring was successful & print the result.

* Complexity:

- Time complexity: The worst case is $O(m^n)$, where n is the number of vertices & m is the number of colours.

- Space Complexity: $O(n)$ for the color array.

* Result:- Hence a query executed successfully