Practical-1

Aim:- Implementation and Time analysis of linear and binary search algorithm.

1) **Linear Search:**

   Code:

```c
#include <stdio.h>

int main() {
    int n;
    printf("Enter length of numbers: ");
    scanf("%d", &n);

    int flag = 0;
    int arr[n];

    for(int i = 0; i < n; i++) {
        printf("Enter %d number: ", i + 1);
        scanf("%d", &arr[i]);
    }

    int num = 0;
    printf("Enter Number to Search: ");
    scanf("%d", &num);

    for(int i = 0; i < n; i++){
        if(arr[i] == num){
            printf("Element Found at Index: %d", i);
            flag = 1;
            break;
        }
    }
    if(flag == 0){
        printf("Element not Found");
    }
    return 0;
}
```

   Output:

```
Enter length of numbers: 5
Enter 1 number: 1
Enter 2 number: 2
Enter 3 number: 3
Enter 4 number: 4
Enter 5 number: 5
Enter Number to Search: 4
Element Found at Index: 3

=== Code Execution Successful ===
```

Time Complexity:

Best Case: $O(1)$
Avg Case: $O(n)$
Worst Case: $O(n)$

2) Binary Code:

Code:

```c
#include <stdio.h>

int main() {

    int n;
    printf("Enter length of numbers: ");
    scanf("%d", &n);

    int flag = 0;
    int arr[n];

    for(int i = 0; i < n; i++) {
        printf("Enter %d number: ", i + 1);
        scanf("%d", &arr[i]);
    }

    int num = 0;
    printf("Enter Number to Search: ");
    scanf("%d", &num);

    int low = 0, high = n;
    while(low <= high){
        int mid = low + (high - low)/2;
```

```
            if(arr[mid] == num){
                printf("Element Found at Index: %d", mid);
                flag = 1;
            }
            if(arr[mid] > num){
                high = mid - 1;
            }
            else{
                low = mid + 1;
            }
        }
        if(flag == 0){
            printf("Element not Found");
        }
        return 0;
    }
```

Output:

```
Enter length of numbers: 5
Enter 1 number: 2
Enter 2 number: 3
Enter 3 number: 6
Enter 4 number: 6
Enter 5 number: 7
Enter Number to Search: 6
Element Found at Index: 2

=== Code Execution Successful ===
```

Time Complexity:
Best Case: O(1)
Avg Case: O(log n)
Worst Case: O(log n)

Practical-2

Aim:- Implementation and Time analysis of sorting algorithms: Bubble sort, Selection sort and Quicksort.

1) Bubble Sort:

Code:

```c
#include <stdio.h>
int main() {
    int n;
    printf("Enter length of numbers: ");
    scanf("%d", &n);

    int arr[n];

    for(int i = 0; i < n; i++) {
        printf("Enter %d number: ", i + 1);
        scanf("%d", &arr[i]);
    }

    for(int i = 0; i < n - 1; i++) {
        int swapped = 0;
        for(int j = 0; j < n - 1 - i; j++) {
            if(arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = 1;
            }
        }
        if (!swapped) {
            break;
        }
    }

    printf("Sorted Array\n");
    for(int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Output:

```
Enter length of numbers: 5
Enter 1 number: 5
Enter 2 number: 4
Enter 3 number: 3
Enter 4 number: 2
Enter 5 number: 1
Sorted Array
1 2 3 4 5

=== Code Execution Successful ===
```

Time Complexity:
> **Best Case:** O(n)
> **Avg Case:** O(n²)
> **Worst Case:** O(n²)

2) Selection Sort:

Code:
```c
#include <stdio.h>
int main() {
    int n;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);
    int a[n];
    for (int i = 0; i < n; i++) {
        printf("\nEnter Element Number %d: ",i+1);
        scanf("%d", &a[i]);
    }
    printf("\nOriginal array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
    for (int i = 0; i < n - 1; i++) {
        int min = i;
        for (int j = i + 1; j < n; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        int temp = a[i];
```

```
        a[i] = a[min];
        a[min] = temp;
    }
    printf("\nSorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
    return 0;
}
```

Output:

```
Enter the number of elements in the array: 5

Enter Element Number 1: 1

Enter Element Number 2: 2

Enter Element Number 3: 3

Enter Element Number 4: 4

Enter Element Number 5: 5

Original array: 1 2 3 4 5

Sorted array: 1 2 3 4 5


=== Code Execution Successful ===
```

Time Complexity:
> **Best Case:** $O(n^2)$
> **Avg Case:** $O(n^2)$
> **Worst Case:** $O(n^2)$

3) Quick Sort:

Code:
```
#include <stdio.h>
void Swap(int *a, int *b) {
    int temp = *a;
```

```c
        *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; 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);
    }
}

void PrintArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int n;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);
    int a[n];
    for (int i = 0; i < n; i++) {
        printf("Enter Element Number %d: ",i+1);
        scanf("%d", &a[i]);
    }
    printf("\nOriginal array: ");
    PrintArray(a, n);
    QuickSort(a, 0, n - 1);
```

```
        printf("\nSorted array: ");
        PrintArray(a, n);
        return 0;
    }
```

Output:

```
Enter the number of elements in the array: 5
Enter Element Number 1: 1
Enter Element Number 2: 5
Enter Element Number 3: 3
Enter Element Number 4: 4
Enter Element Number 5: 2

Original array: 1 5 3 4 2

Sorted array: 1 2 3 4 5

=== Code Execution Successful ===
```

Time Complexity:
> **Best Case:** O(n log n)
> **Avg Case:** O(n log n)
> **Worst Case:** O(n²)

Practical-3

<u>Aim</u>: **Implementation and Time analysis of factorial program using iterative and recursive method.**

1) Factorial program with iterative method :-

Code:
```c
#include <stdio.h>
```

```c
int computeFactorial(int num) {
    int fact = 1;
    if (num < 0) {
        printf("Factorial is not defined for negative numbers.\n");
        return -1;
    } else if (num == 0) {
        return 1;
    } else {
        for (int i = 1; i <= num; i++) {
            fact *= i;
        }
        return fact;
    }
}

int main() {
    int inputValue;
    printf("Enter an integer: ");
    scanf("%d", &inputValue);

    int factorialResult = computeFactorial(inputValue);
    if (factorialResult != -1) {
        printf("Factorial of %d is: %d\n", inputValue, factorialResult);
    }

    return 0;
}
```

Output:

```
Enter an integer: 7
Factorial of 7 is: 5040



=== Code Execution Successful ===
```

Time Complexity:

Best Case: $O(1)$

Avg Case: $O(n)$

Worst Case: $O(n^2)$

2) Factorial program with iterative method :-

Code:
```c
#include <stdio.h>

int computeFactorial(int value) {
    if (value == 0) {
        return 1;
    } else {
        return value * computeFactorial(value - 1);
    }
}
int main() {
    int userInput;
    printf("Enter an integer: ");
    scanf("%d", &userInput);

    if (userInput < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        printf("Factorial of %d = %d\n", userInput, computeFactorial(userInput));
    }

    return 0;
}
```

Output:
```
Enter an integer: 7
Factorial of 7 is: 5040


=== Code Execution Successful ===
```

Time Complexity:
Best Case: $O(1)$
Avg Case: $O(n)$
Worst Case: $O(n)$

Practical-4

Aim: Implementation Prim's algorithm.

Code:

```c
#include <stdio.h>
#define V 3
#define INF 999999
int main() {
    int cost[V][V] = {  {10, 15, 7}, {1, 20, 7}, {3, 13, 5}  };
    int selected[V] = {0};
    int edge_count = 0;
    int mincost = 0;
    selected[0] = 1;
    printf("Prim's Algorithm - Edges in MST:\n");
```

```c
    while (edge_count < V - 1) {
        int min = INF, x = -1, y = -1;
        for (int i = 0; i < V; i++) {
            if (selected[i]) {
                for (int j = 0; j < V; j++) {
                    if (!selected[j] && cost[i][j]) {
                        if (cost[i][j] < min) {
                            min = cost[i][j];
                            x = i;
                            y = j;   }
                    }
                }
            }
        }
        if (x != -1 && y != -1) {
            printf("%d - %d : %d\n", x, y, cost[x][y]);
            mincost += cost[x][y];
            selected[y] = 1;
            edge_count++;
        }
    }
    printf("Minimum cost = %d\n", mincost);
    return 0;
}
```

Output:

```
Prim's Algorithm - Edges in MST:
0 - 2 : 7
2 - 1 : 13
Minimum cost = 20


=== Code Execution Successful ===
```

Time Complexity:

Best Case: $O(V^2)$

Avg Case: $O(V^2)$

Worst Case: $O(V^2)$

Practical-5

Aim: Implementation Kruskal's algorithm.

Code:
```c
#include <stdio.h>
#include <stdlib.h>

const int inf = 999999;
int k, a, b, u, v, n, ne = 1;
int mincost = 0;
int cost[3][3] = {{10, 10, 2},{1, 0,11},{18, 16, 10}};
int  p[9] = {0};
int applyfind(int i)
{
   while(p[i] != 0)
     i=p[i];
   return i;
}
int applyunion(int i,int j)
```

```c
{
    if(i!=j) {
        p[j]=i;
        return 1;
    }
    return 0;
}
int main()
{
    n = 3;
    int i, j;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (cost[i][j] == 0) {
                cost[i][j] = inf;
            }
        }
    }
    printf("kruskal's Algorithm - Edges in MST:\n");
    while(ne < n) {
        int min_val = inf;
        for(i=0; i<n; i++) {
            for(j=0; j <n; j++) {
                if(cost[i][j] < min_val) {
                    min_val = cost[i][j];
                    a = u = i;
                    b = v = j;
                }
            }
        }
        u = applyfind(u);
        v = applyfind(v);
        if(applyunion(u, v) != 0) {
            printf("%d -> %d\n", a, b);
            mincost +=min_val;
        }
        cost[a][b] = cost[b][a] = 999;
        ne++;
    }
    printf("Minimum cost = %d",mincost);
    return 0;
}
```

Output:

```
kruskal's Algorithm - Edges in MST:
1 -> 0
0 -> 2
Minimum cost = 3

=== Code Execution Successful ===
```

Time Complexity:
> **Best Case:** $O(V^2)$
> **Avg Case:** $O(V^3)$
> **Worst Case:** $O(V^3)$

Practical-6

Aim: Implementation of a knapsack problem using dynamic programming.

Code:
```c
// knapsack by dynamic programming
#include <stdio.h>
#define MAX_ITEMS 100
#define MAX_CAPACITY 100
int max(int a, int b) {
    return (a > b) ? a : b;
}
int main() {
    int n, W;
    int weight[MAX_ITEMS], value[MAX_ITEMS];
    int i, w;
    n = 3;
    W = 500;
    weight[0] = 100; value[0] = 160;
    weight[1] = 205; value[1] = 180;
    weight[2] = 130; value[2] = 190;
```

```
   int dp[MAX_ITEMS+1][MAX_CAPACITY+1];
   for (i = 0; i <= n; i++) {
      for (w = 0; w <= W; w++) {
         if (i == 0 || w == 0)
            dp[i][w] = 0;
         else if (weight[i-1] <= w)
            dp[i][w] = max(value[i-1] + dp[i-1][w-weight[i-1]], dp[i-1][w]);
         else
            dp[i][w] = dp[i-1][w];
      }
   }
   printf("0/1 Knapsack Problem using Dynamic Programming\n");
   printf("Maximum value in knapsack = %d\n", dp[n][W]);
   return 0;
}
```

Output:

```
0/1 Knapsack Problem using Dynamic Programming
Maximum value in knapsack = 380


=== Code Execution Successful ===
```

Time Complexity:

Best Case: $O(n \cdot W)$

Avg Case: $O(n \cdot W)$

Worst Case: $O(n \cdot W)$

Practical-7

Aim: **Implementation of matrix chain multiplication using dynamic programming.**

Code:
```
// matrix multiplication by dynamic programming
#include <stdio.h>
#include <limits.h>
int matrixChainOrder(int p[], int n) {
    int m[n][n];

    for (int i = 1; i < n; i++)
        m[i][i] = 0;
    for (int L = 2; L < n; L++) {
        for (int i = 1; i <= n - L; i++) {
            int j = i + L - 1;
            m[i][j] = INT_MAX;
            for (int k = i; k < j; k++) {
                int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }
```

```
        return m[1][n - 1];
}
int main() {
    int arr[] = {5, 4, 6, 2, 7};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Minimum number of multiplications is %d\n", matrixChainOrder(arr, size));
    return 0;
}
```

Output:

```
Minimum number of multiplications is 865



=== Code Execution Successful ===
```

Time Complexity:

        Best Case: $O(n^3)$

        Avg Case: $O(n^3)$

        Worst Case: $O(n^3)$

Practical-8

Aim: Implementation of making a change problem using dynamic programming.

Code:// making change problem by dynamic programming

```c
#include <stdio.h>
#include <limits.h>

int minCoins(int coins[], int m, int V) {
    int dp[V + 1];
    dp[0] = 0;
```

```c
    for (int i = 1; i <= V; i++)
        dp[i] = INT_MAX;
    for (int i = 1; i <= V; i++) {
        for (int j = 0; j < m; j++) {
            if (coins[j] <= i && dp[i - coins[j]] != INT_MAX && dp[i - coins[j]] + 1 < dp[i])
                dp[i] = dp[i - coins[j]] + 1;
        }
    }
    return dp[V] == INT_MAX ? -1 : dp[V];
}
int main() {
    int coins[] = {5, 6, 7};
    int m = sizeof(coins) / sizeof(coins[0]);
    int V = 15;
    int result = minCoins(coins, m, V);
    printf("Minimum Coins Required is: %d\n", result);
    return 0;
}
```

Output:

```
Minimum Coins Required is: 3


=== Code Execution Successful ===
```

Time Complexity:

   Best Case: $O(V \cdot m)$

   Avg Case: $O(V \cdot m)$

   Worst Case: $O(V \cdot m)$

Practical-9

<u>Aim</u>: Implementation of Graph and Searching (DFS and BFS).

Code:

```c
// Graph & Searching DFS BFS
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX_VERTICES 10

typedef struct Graph {
    int adj[MAX_VERTICES][MAX_VERTICES];
```

```c
    int numVertices;
} Graph;

void initGraph(Graph* graph, int numVertices) {
    graph->numVertices = numVertices;
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            graph->adj[i][j] = 0;
        }
    }
}

void addEdge(Graph* graph, int startVertex, int endVertex) {
    graph->adj[startVertex][endVertex] = 1;
    graph->adj[endVertex][startVertex] = 1;
}

void DFSHelper(Graph* graph, int vertex, bool* visited) {
    visited[vertex] = true;
    printf("%d ", vertex);

    for (int i = 0; i < graph->numVertices; i++) {
        if (graph->adj[vertex][i] == 1 && !visited[i]) {
            DFSHelper(graph, i, visited);
        }
    }
}

void DFS(Graph* graph, int startVertex) {
    bool visited[MAX_VERTICES] = {false};
    printf("DFS starting from vertex %d:\n", startVertex);
    DFSHelper(graph, startVertex, visited);
    printf("\n");
}

void BFS(Graph* graph, int startVertex) {
    bool visited[MAX_VERTICES] = {false};
    int queue[MAX_VERTICES], front = -1, rear = -1;

    visited[startVertex] = true;
    queue[++rear] = startVertex;

    printf("BFS starting from vertex %d:\n", startVertex);
```

```c
    while (front != rear) {
        int vertex = queue[++front];
        printf("%d ", vertex);

        for (int i = 0; i < graph->numVertices; i++) {
            if (graph->adj[vertex][i] == 1 && !visited[i]) {
                visited[i] = true;
                queue[++rear] = i;
            }
        }
    }
    printf("\n");
}

int main() {
    Graph graph;
    int numVertices, numEdges, startVertex, start, end;

    printf("Enter the number of vertices: ");
    scanf("%d", &numVertices);
    initGraph(&graph, numVertices);

    printf("Enter the number of edges: ");
    scanf("%d", &numEdges);
    for (int i = 0; i < numEdges; i++) {
        printf("Enter edge (start and end vertex): ");
        scanf("%d %d", &start, &end);
        addEdge(&graph, start, end);
    }

    printf("Enter the start vertex for DFS and BFS: ");
    scanf("%d", &startVertex);

    DFS(&graph, startVertex);
    BFS(&graph, startVertex);

    return 0;
}
```

Output:

```
Enter the number of vertices: 4
Enter the number of edges: 3
Enter edge (start and end vertex): 0 1
Enter edge (start and end vertex): 1 2
Enter edge (start and end vertex): 2 3
Enter the start vertex for DFS and BFS: 1
DFS starting from vertex 1:
1 0 2 3
BFS starting from vertex 1:
1 0 2 3


=== Code Execution Successful ===
```

Time Complexity:
> **Best Case:** $O(V^2)$
> **Avg Case:** $O(V^2)$
> **Worst Case:** $O(V^2)$

Practical-10

Aim: Implement LCS problem.

Code:
```c
// LCS problem
#include <stdio.h>
#include <string.h>

#define MAX 100

int LCS(char* X, char* Y, int m, int n) {
    int dp[m+1][n+1];

    for (int i = 0; i <= m; i++) {
```

```
      for (int j = 0; j <= n; j++) {
        if (i == 0 || j == 0) {
          dp[i][j] = 0;
        } else if (X[i-1] == Y[j-1]) {
          dp[i][j] = dp[i-1][j-1] + 1;
        } else {
          dp[i][j] = (dp[i-1][j] > dp[i][j-1]) ? dp[i-1][j] : dp[i][j-1];
        }
      }
    }

    return dp[m][n];
}

void printLCS(char* X, char* Y, int m, int n) {
    int dp[m+1][n+1];

    for (int i = 0; i <= m; i++) {
      for (int j = 0; j <= n; j++) {
        if (i == 0 || j == 0) {
          dp[i][j] = 0;
        } else if (X[i-1] == Y[j-1]) {
          dp[i][j] = dp[i-1][j-1] + 1;
        } else {
          dp[i][j] = (dp[i-1][j] > dp[i][j-1]) ? dp[i-1][j] : dp[i][j-1];
        }
      }
    }

    int index = dp[m][n];
    char lcs[index + 1];
    lcs[index] = '\0';

    int i = m, j = n;
    while (i > 0 && j > 0) {
      if (X[i-1] == Y[j-1]) {
        lcs[index-1] = X[i-1];
        i--;
        j--;
        index--;
      } else if (dp[i-1][j] > dp[i][j-1]) {
        i--;
      } else {
        j--;
```

```c
        }
    }

    printf("LCS: %s\n", lcs);
}

int main() {
    char X[MAX], Y[MAX];

    printf("Enter first string: ");
    scanf("%s", X);

    printf("Enter second string: ");
    scanf("%s", Y);

    int m = strlen(X);
    int n = strlen(Y);

    int length = LCS(X, Y, m, n);
    printf("Length of LCS: %d\n", length);

    printLCS(X, Y, m, n);

    return 0;
}
```

Output:

```
Enter first string: AFDSASDFFDASDSAFFSDA
Enter second string: ADFS
Length of LCS: 4
LCS: ADFS


=== Code Execution Successful ===
```

Time Complexity:

       Best Case: $O(m{\cdot}n)$

       Avg Case: $O(m{\cdot}n)$

       Worst Case: $O(m{\cdot}n)$