Design and Analysis of Algorithms Lab
CIC-357

Student Name: Sameer Sharma **Faculty Name**: Dr. Sudha Narang

Roll No: 02314802723

Group: 5C1

MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

MAIT

उद्यमेन हि सिध्यन्ति
कार्याणि न मनोरथैः

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Rubrics for Lab Assessment:

Rubrics		10 Marks			POs and PSOs Covered	
		0 Marks	1 Marks	2 Marks	PO	PSO
R1	Is able to identify and define the objective of the given problem?	No	Partially	Completely	PO1, PO2	PSO1, PSO2
R2	Is proposed design/procedure/algorithm solves the problem?	No	Partially	Completely	PO1,PO2, PO3	PSO1, PSO2
R3	Has the understanding of the tool/programming language to implement the proposed solution?	No	Partially	Completely	PO1,PO3, PO5	PSO1, PSO2
R4	Are the result(s) verified using sufficient test data to support the conclusions?	No	Partially	Completely	PO2,PO4, PO5	PSO2
R5	Individuality of submission?	No	Partially	Completely	PO8, PO12	PSO1, PSO3

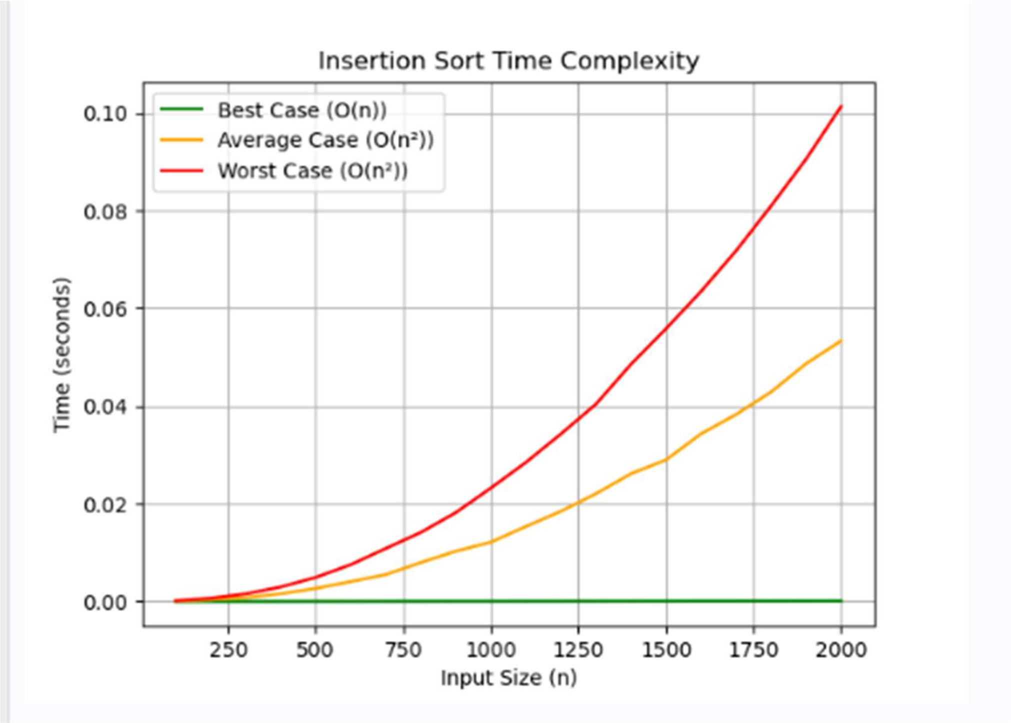
PROGRAM-1(A)

Aim-Insertion sort

THEORY:- Insertion Sort is a simple comparison-based sorting algorithm that builds the final sorted array one element at a time. It works much like how you might sort playing cards in your hands.

CODE:-

```
#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>


using namespace std;

struct OperationCount {

   int comparisons = 0;

   int shifts = 0;

};


OperationCount insertionSort(vector<int>& arr) {

   OperationCount ops;

   int n = arr.size();

   for (int i = 1; i < n; ++i) {

      int key = arr[i];

      int j = i - 1;


      while (j >= 0 && arr[j] > key) {
            ops.comparisons++;
            arr[j + 1] = arr[j];
            ops.shifts++;
            j--;
        }
        // If while loop didn't run or ended due to arr[j] <= key, count that comparison
        if (j >= 0) ops.comparisons++;

        arr[j + 1] = key;
        ops.shifts++;
    }
    return ops;
}

void printArray(const vector<int>& arr) {
    for (int val : arr)
        cout << val << " ";
    cout << "\n";
}
int main() {
    srand((unsigned)time(0));
    vector<int> bestCase(10);
    for (int i = 0; i < 10; ++i) bestCase[i] = i + 1;
    vector<int> averageCase(10);
    for (int i = 0; i < 10; ++i) averageCase[i] = rand() % 100 + 1;
    vector<int> worstCase(10);
    for (int i = 0; i < 10; ++i) worstCase[i] = 10 - i;
    cout << "Best Case (Already Sorted):\nBefore: ";
    printArray(bestCase);
    OperationCount bestOps = insertionSort(bestCase);
    cout << "After:  ";
    printArray(bestCase);
    cout << "Comparisons: " << bestOps.comparisons << ", Shifts: " << bestOps.shifts << "\n\n";
    cout << "Average Case (Random Array):\nBefore: ";
    printArray(averageCase);
    OperationCount avgOps = insertionSort(averageCase);
    cout << "After:  ";
    printArray(averageCase);
    cout << "Comparisons: " << avgOps.comparisons << ", Shifts: " << avgOps.shifts << "\n\n";
    cout << "Worst Case (Reverse Sorted):\nBefore: ";
    printArray(worstCase);
    OperationCount worstOps = insertionSort(worstCase);
    cout << "After:  ";
    printArray(worstCase);
    cout << "Comparisons: " << worstOps.comparisons << ", Shifts: " << worstOps.shifts << "\n";

    return 0;
}
```


Insertion Sort Time Complexity
Best Case (O(n))
Average Case (O(n²))
Worst Case (O(n²))
Time (seconds)
0.00
0.02
0.04
0.06
0.08
0.10
Input Size (n)
250
500
750
1000
1250
1500
1750
2000

Program-1(b)

AIM:- Selection Sort

THEORY:- Selection Sort is a simple, comparison-based sorting algorithm that divides the input list into two parts:

- The sorted part at the beginning (initially empty).
- The unsorted part that occupies the rest of the list.

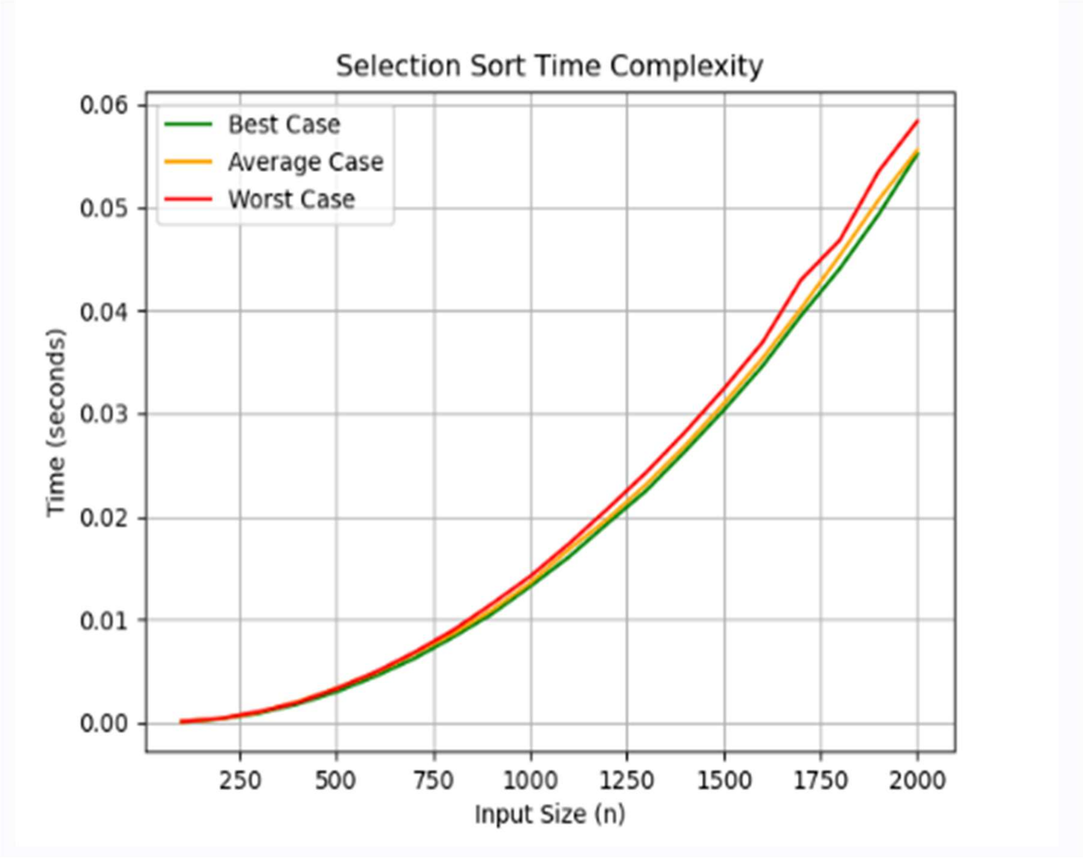
It repeatedly selects the smallest (or largest) element from the unsorted part and swaps it with the leftmost unsorted element, expanding the sorted part by one each time.

Code:-

```
#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

using namespace std;

struct OperationCount {

    int comparisons = 0;

    int swaps = 0;

};


OperationCount selectionSort(vector<int>& arr) {

    OperationCount ops;

    int n = arr.size();

    for (int i = 0; i < n - 1; ++i) {

        int minIndex = i;

        for (int j = i + 1; j < n; ++j) {

            ops.comparisons++;

            if (arr[j] < arr[minIndex])
```

```
                minIndex = j;
        }
        if (minIndex != i) {
            swap(arr[i], arr[minIndex]);
            ops.swaps++;
        }
    }
    return ops;
}

void printArray(const vector<int>& arr) {
    for (int val : arr)
        cout << val << " ";
    cout << "\n";
}

int main() {
    srand((unsigned)time(0));

    // Best case: already sorted array
    vector<int> bestCase(10);
    for (int i = 0; i < 10; ++i)
        bestCase[i] = i + 1;

    // Average case: random array
    vector<int> averageCase(10);
    for (int i = 0; i < 10; ++i)
        averageCase[i] = rand() % 100 + 1;
    // Worst case: reverse sorted array
    vector<int> worstCase(10);
    for (int i = 0; i < 10; ++i)
        worstCase[i] = 10 - i;
    cout << "Best Case (Already Sorted):\nBefore: ";
    printArray(bestCase);
    OperationCount bestOps = selectionSort(bestCase);
    cout << "After:  ";
    printArray(bestCase);
    cout << "Comparisons: " << bestOps.comparisons << ", Swaps: " << bestOps.swaps <<
"\n\n";
    cout << "Average Case (Random Array):\nBefore: ";
    printArray(averageCase);
    OperationCount avgOps = selectionSort(averageCase);
    cout << "After:  ";
    printArray(averageCase);
    cout << "Comparisons: " << avgOps.comparisons << ", Swaps: " << avgOps.swaps <<
"\n\n";

    cout << "Worst Case (Reverse Sorted):\nBefore: ";
    printArray(worstCase);
    OperationCount worstOps = selectionSort(worstCase);
    cout << "After:  ";
    printArray(worstCase);
    cout << "Comparisons: " << worstOps.comparisons << ", Swaps: " << worstOps.swaps <<
"\n";
    return 0;
}
```

OUTPUT:-


Selection Sort Time Complexity
Best Case
Average Case
Worst Case
0.06
0.05
0.04
0.03
0.02
0.01
0.00
Time (seconds)
250
500
750
1000
1250
1500
1750
2000
Input Size (n)

Program-1(c)

AIM:- Bubble Sort

THEORY:- Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

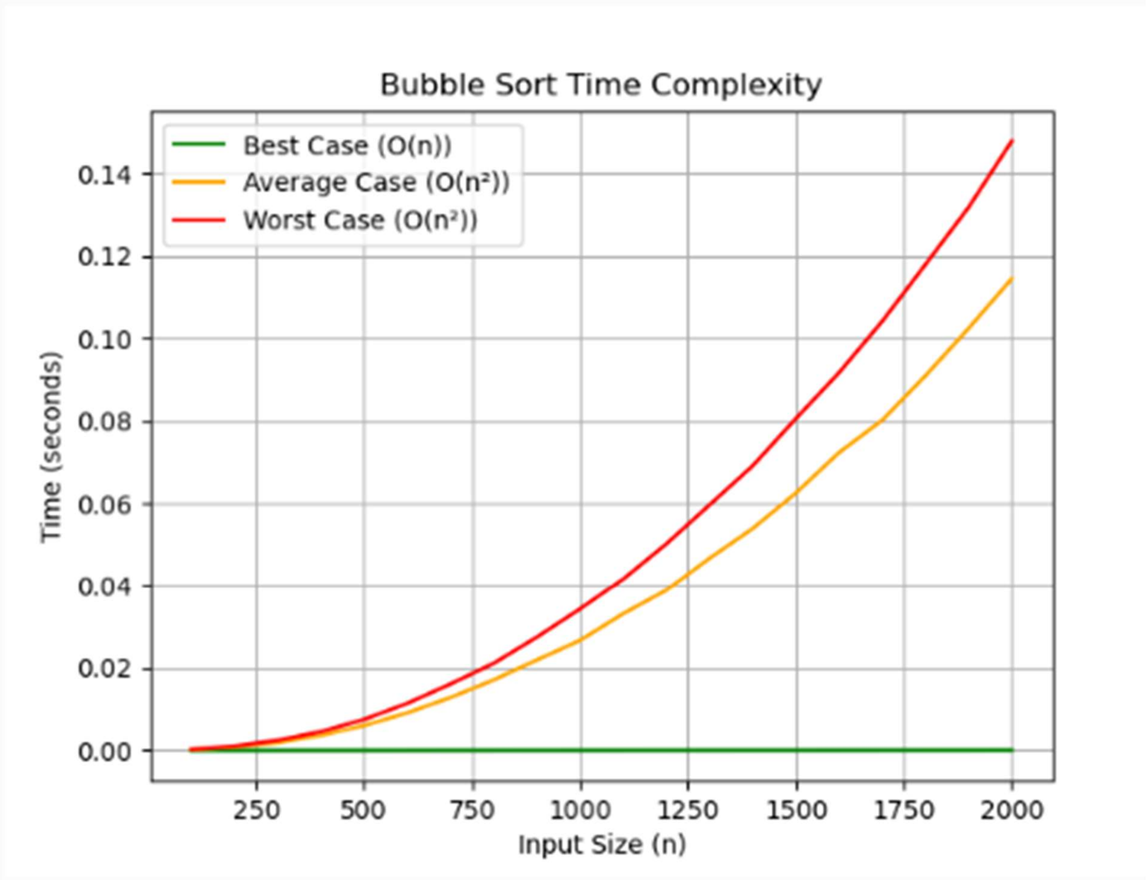
The algorithm gets its name because smaller elements “bubble” to the top (beginning of the list) with each pass.

Code:-

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <chrono>

using namespace std;
using namespace std::chrono;

void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    bool swapped;
    for (int i = 0; i < n - 1; ++i) {
        swapped = false;
        for (int j = 0; j < n - 1 - i; ++j) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        // If no two elements were swapped by inner loop, then break
        if (!swapped)
            break;
    }
}

void printArray(const vector<int>& arr) {
    for (int x : arr) cout << x << " ";
    cout << "\n";
}

int main() {
    srand(time(0));

    // Best case: already sorted array
    vector<int> bestCase = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Average case: random array
    vector<int> averageCase(10);
    for (int &x : averageCase)
        x = rand() % 100 + 1;

    // Worst case: reverse sorted array
    vector<int> worstCase = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

    cout << "Best Case (Already Sorted):\nBefore: ";
    printArray(bestCase);
    auto start = high_resolution_clock::now();
```

```
bubbleSort(bestCase);
auto end = high_resolution_clock::now();
cout << "After:  ";
printArray(bestCase);
cout << "Time taken: "
    << duration<double, milli>(end - start).count()
    << " ms\n\n";

cout << "Average Case (Random Array):\nBefore: ";
printArray(averageCase);
start = high_resolution_clock::now();
bubbleSort(averageCase);
end = high_resolution_clock::now();
cout << "After:  ";
printArray(averageCase);
cout << "Time taken: "
    << duration<double, milli>(end - start).count()
    << " ms\n\n";

cout << "Worst Case (Reverse Sorted):\nBefore: ";
printArray(worstCase);
start = high_resolution_clock::now();
bubbleSort(worstCase);
end = high_resolution_clock::now();
cout << "After:  ";
printArray(worstCase);
cout << "Time taken: "
    << duration<double, milli>(end - start).count()
    << " ms\n";
    return 0;
}
```

OUTPUT:-



Program – 2(a)

AIM: To implement Linear search and analyze its time complexity**.**

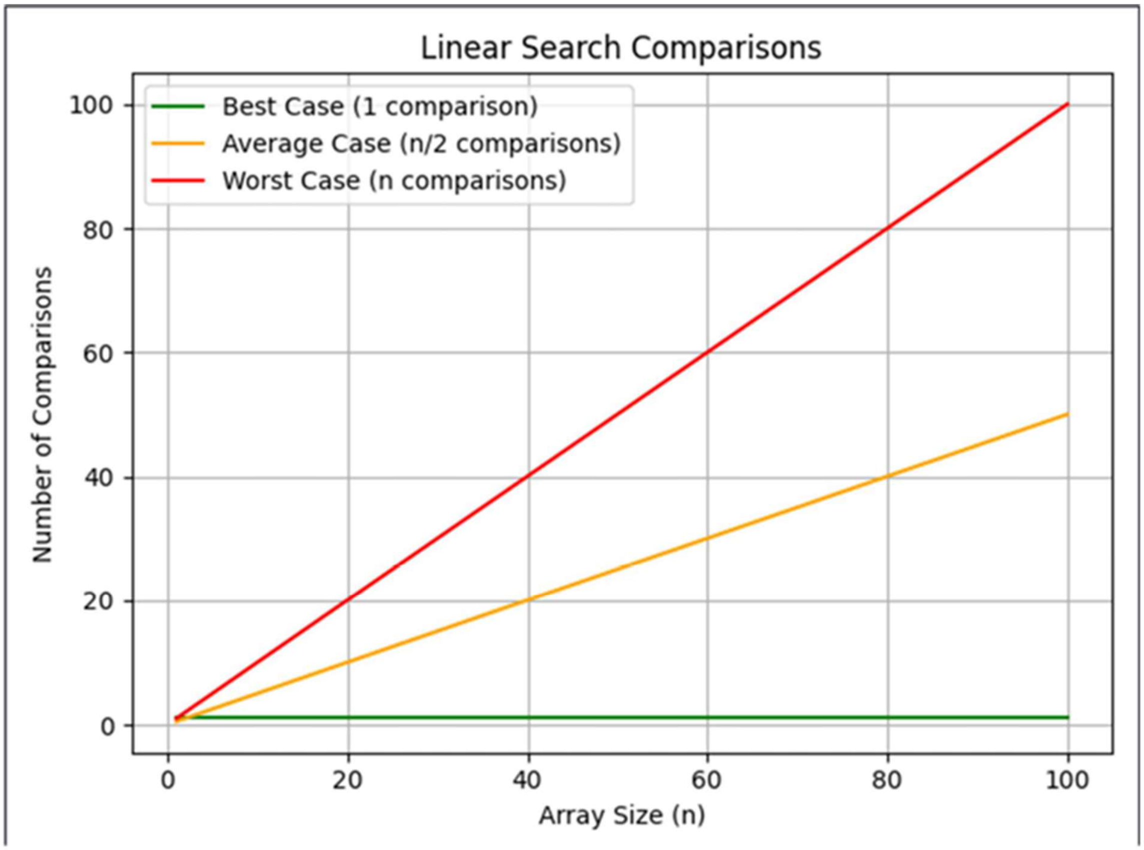
THEORY

Linear search is a simple algorithm that checks each element in a list sequentially until the target is found or the list ends. It works on both sorted and unsorted data, is easy to implement, but inefficient for large datasets due to its **O(n)** time complexity.

CODE

```
#include <iostream>
using namespace std;

int linearSearch(int arr[], int n, int x, int &comparisons) {
    for (int i = 0; i < n; i++) {
        comparisons++;
        if (arr[i] == x)
            return i;
    }
    return -1;
}

int main() {
    const int n = 10;
    int arr[n];

    for (int i = 0; i < n; i++)
        arr[i] = i + 1;

    int comparisons, index;

    comparisons = 0;
    index = linearSearch(arr, n, 1, comparisons);
    cout << "Best Case - Found at index: " << index << ", Comparisons: " << comparisons << endl;
    comparisons = 0;
    index = linearSearch(arr, n, 5, comparisons);
    cout << "Average Case - Found at index: " << index << ", Comparisons: " << comparisons << endl;
comparisons = 0;
index = linearSearch(arr, n, 100, comparisons);
cout << "Worst Case - Found at index: " << index << ", Comparisons: " << comparisons << endl;
return 0;
}
```

OUTPUT



PROGRAM – 2(b)

AIM: To implement Binary Search and analyze its time complexity.

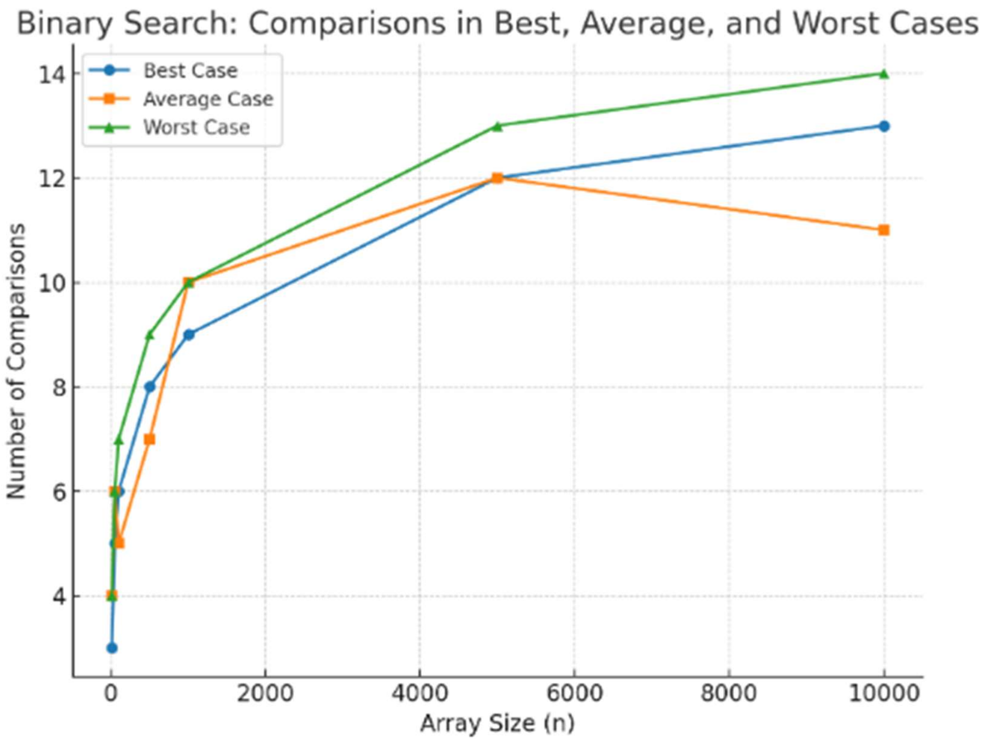
THEORY

Binary search is an efficient algorithm that works only on sorted arrays or lists. It repeatedly compares the target with the middle element and reduces the search space to either the left or right half. This process continues until the element is found or the search space becomes empty. Its time complexity is O(log n), making it much faster than linear search for large datasets.

CODE

```
#include <iostream>
using namespace std;

int binarySearch(int arr[], int n, int x, int &comparisons) {
    int left = 0, right = n - 1;
    while (left <= right) {
        comparisons++;
        int mid = left + (right - left) / 2;
        if (arr[mid] == x)
            return mid;
        else if (arr[mid] < x)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}

int main() {
    const int n = 15;
    int arr[n];

    for (int i = 0; i < n; i++)
        arr[i] = i + 1;
    int comparisons, index;

comparisons = 0;
index = binarySearch(arr, n, arr[n/2], comparisons);
cout << "Best Case - Found at index: " << index << ", Comparisons: " << comparisons << endl;
comparisons = 0;
index = binarySearch(arr, n, 3, comparisons);
cout << "Average Case - Found at index: " << index << ", Comparisons: " << comparisons << endl;
comparisons = 0;
index = binarySearch(arr, n, 100, comparisons);
cout << "Worst Case - Found at index: " << index << ", Comparisons: " << comparisons << endl;
return 0;
}
```

OUTPUT



Program 3(a)

Aim: Merge Sort

Theory:

- Merge Sort is a Divide and Conquer algorithm.
- It works by splitting the array into two halves, sorting each half recursively, and then merging the two sorted halves into one sorted array.
- This makes it a stable sorting algorithm (keeps the relative order of equal elements).

  Best Case (Ω(n log n))

  Average Case (Θ(n log n))

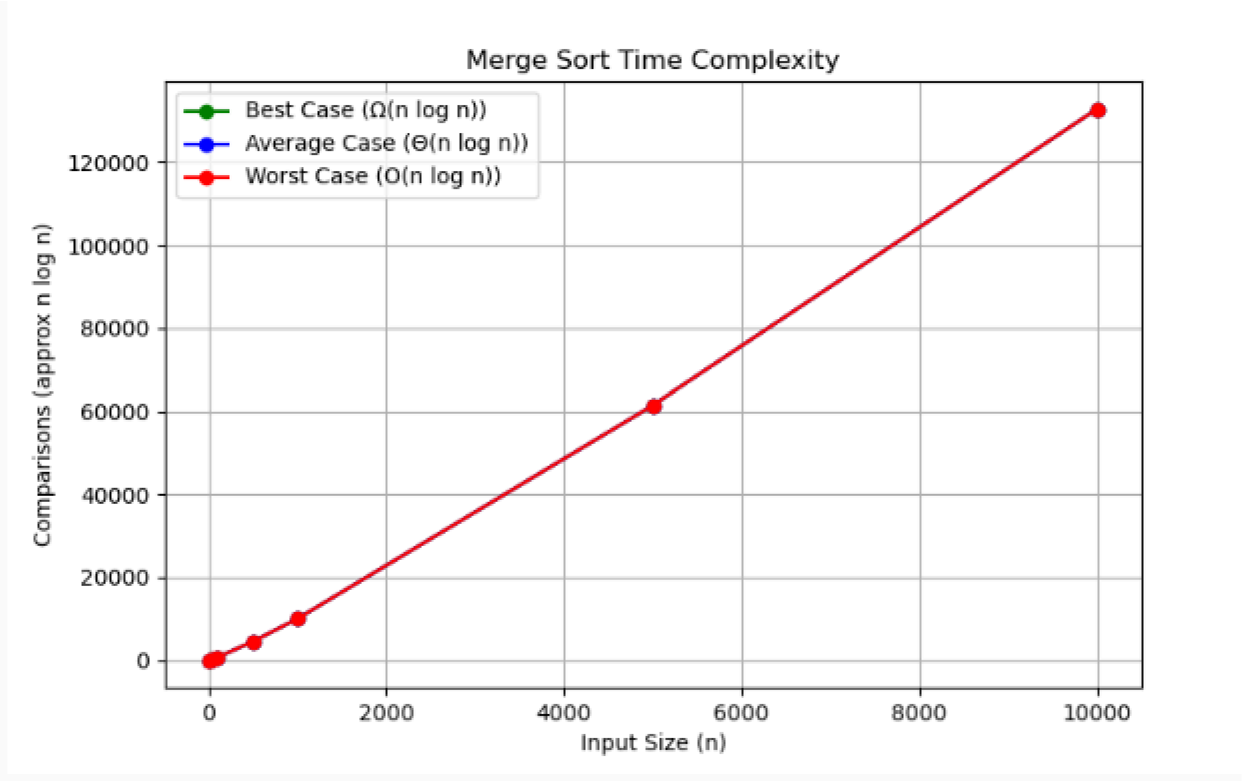
  Worst Case (O(n log n))

Code:

```
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

void merge(vector<int> &arr, int l, int m, int r) {
   int n1 = m - l + 1, n2 = r - m;
   vector<int> L(n1), R(n2);
   for (int i = 0; i < n1; i++) L[i] = arr[l + i];
   for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

   int i = 0, j = 0, k = l;
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) arr[k++] = L[i++];
      else arr[k++] = R[j++];
   }
   while (i < n1) arr[k++] = L[i++];
   while (j < n2) arr[k++] = R[j++];
}
```

```
void mergeSort(vector<int> &arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

int main() {
    vector<int> sizes = {10, 50, 100, 1000, 5000, 10000};
    srand(time(0));

    for (int n : sizes) {
        vector<int> arr(n);
        for (int i = 0; i < n; i++) arr[i] = rand() % 100000;

        auto start = high_resolution_clock::now();
        mergeSort(arr, 0, n - 1);
        auto end = high_resolution_clock::now();
        auto elapsed = duration_cast<nanoseconds>(end - start).count();

        cout << "Merge Sort | n=" << n
             << " | Time=" << elapsed << " ns\n";
    }
    return 0;
}
```

Output:


Merge Sort Time Complexity
Best Case (Ω(n log n))
Average Case (Θ(n log n))
Worst Case (O(n log n))
Comparisons (approx n log n)
0
20000
40000
60000
80000
100000
120000
Input Size (n)
0
2000
4000
6000
8000
10000

Program 3(b)

Aim: Quick Sort

Theory:

Definition:
Quick Sort is a divide and conquer sorting algorithm. It selects a pivot element, partitions the array into two subarrays (elements smaller and greater than the pivot), and recursively sorts them.

Algorithm Steps:

1. Choose a pivot.
2. Partition the array around the pivot.
3. Recursively apply Quick Sort to left and right partitions.
4. Combine results.

Complexity:

- Best Case: O(n log n) – balanced partition.
- Average Case: O(n log n).
- Worst Case: O(n^2) – unbalanced partition (e.g., sorted input with poor pivot).

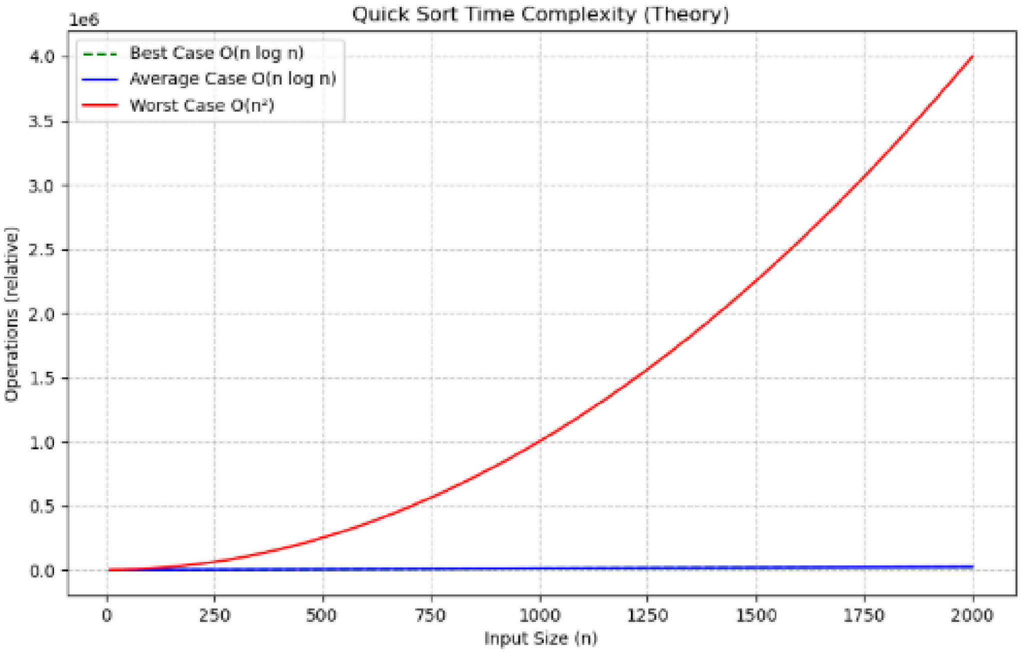
Space Complexity: O(log n) (recursion stack).

Code:

```
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

// Quick Sort implementation
int partition(vector<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(vector<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);
    }
}

int main() {
    vector<int> sizes = {10, 50, 100, 500, 1000, 5000, 10000, 50000};

    for (int n : sizes) {
        vector<int> arr(n);
        for (int i = 0; i < n; i++) arr[i] = rand() % 100000; // random input

        auto start = high_resolution_clock::now();
        quickSort(arr, 0, n - 1);
        auto stop = high_resolution_clock::now();

        auto duration = duration_cast<microseconds>(stop - start);
        cout << "n = " << n << " | Time = " << duration.count() << " microseconds" << endl;
    }
    return 0;
}
```

Output:


Quick Sort Time Complexity (Theory)
1e6
Best Case O(n log n)
Average Case O(n log n)
Worst Case O(n²)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Operations (relative)
0
250
500
750
1000
1250
1500
1750
2000
Input Size (n)

PROGRAM – 4

AIM: Strassen matrix multiplication.

THEORY:

Strassen’s Matrix Multiplication is an efficient divide-and-conquer algorithm for multiplying two square matrices. The classical method requires $O(n^3)$ time in best, average, and worst cases. Strassen reduced the number of multiplications from 8 to 7 by using additions and subtractions, which gives the recurrence $T(n) = 7T(n/2) + O(n^2)$. This solves to O(n^2.81), which is faster than the naive approach. Since the algorithm is deterministic, the best, average, and worst cases are the same. Strassen works best for large matrices, while for small matrices the classical method is often faster. If the matrix size is not a power of 2, the matrices can be padded with zeros to the next power of 2 before applying the algorithm.

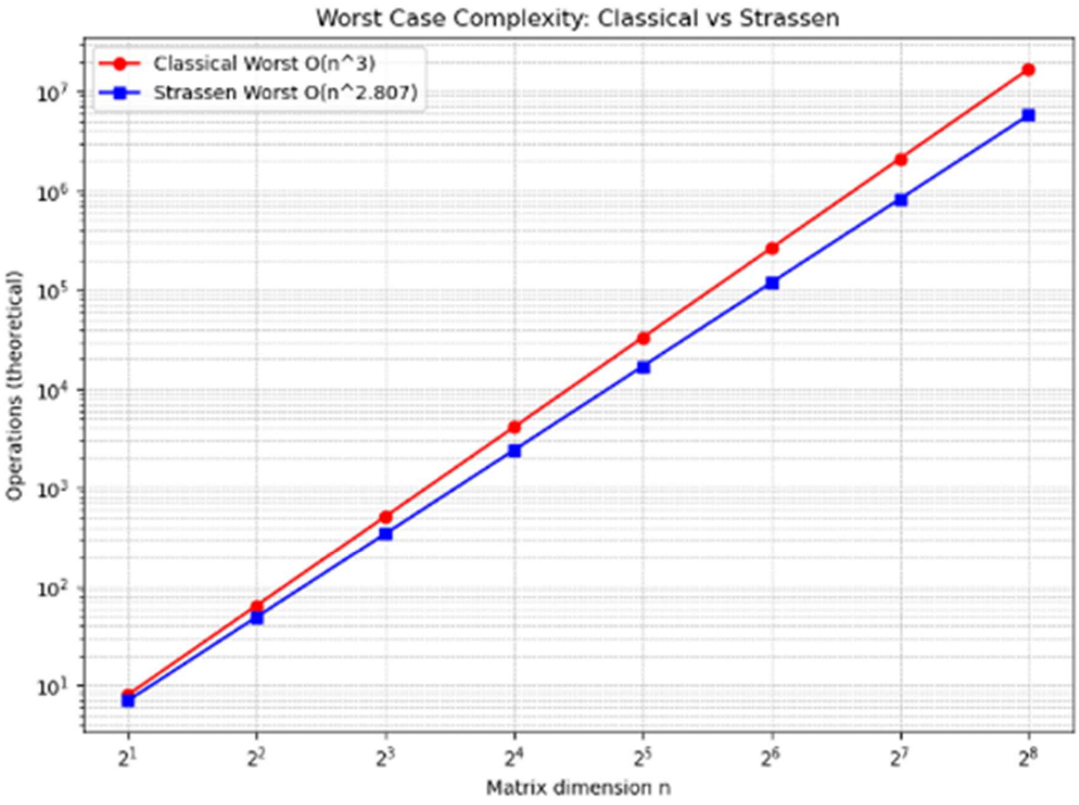
CODE:

```
#include <bits/stdc++.h>
using namespace std;

// Function to add two matrices
vector<vector<int>> add(vector<vector<int>> &A, vector<vector<int>> &B) {
    int n = A.size();
    vector<vector<int>> C(n, vector<int>(n, 0));
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            C[i][j] = A[i][j] + B[i][j];
    return C;
}

// Function to subtract two matrices
vector<vector<int>> sub(vector<vector<int>> &A, vector<vector<int>> &B) {
    int n = A.size();
    vector<vector<int>> C(n, vector<int>(n, 0));
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            C[i][j] = A[i][j] - B[i][j];
    return C;
}
```

```
// Strassen's Algorithm
vector<vector<int>> strassen(vector<vector<int>> A, vector<vector<int>> B) {
    int n = A.size();
    if(n == 1) {
        return {{A[0][0] * B[0][0]}};
    }
    int k = n/2;

    // Splitting matrices into 4 submatrices
    vector<vector<int>> A11(k, vector<int>(k)), A12(k, vector<int>(k)), A21(k,
vector<int>(k)), A22(k, vector<int>(k));
    vector<vector<int>> B11(k, vector<int>(k)), B12(k, vector<int>(k)), B21(k,
vector<int>(k)), B22(k, vector<int>(k));

    for(int i=0; i<k; i++){
        for(int j=0; j<k; j++){
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j+k];
            A21[i][j] = A[i+k][j];
            A22[i][j] = A[i+k][j+k];

            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j+k];
            B21[i][j] = B[i+k][j];
            B22[i][j] = B[i+k][j+k];
        }
    }

    // Strassen's 7 multiplications
    auto M1 = strassen(add(A11, A22), add(B11, B22));
    auto M2 = strassen(add(A21, A22), B11);
    auto M3 = strassen(A11, sub(B12, B22));
    auto M4 = strassen(A22, sub(B21, B11));
    auto M5 = strassen(add(A11, A12), B22);
    auto M6 = strassen(sub(A21, A11), add(B11, B12));
    auto M7 = strassen(sub(A12, A22), add(B21, B22));

    // Combining results
    auto C11 = add(sub(add(M1, M4), M5), M7);
    auto C12 = add(M3, M5);
    auto C21 = add(M2, M4);
    auto C22 = add(sub(add(M1, M3), M2), M6);

    // Final result
    vector<vector<int>> C(n, vector<int>(n));
    for(int i=0; i<k; i++){
        for(int j=0; j<k; j++){
            C[i][j] = C11[i][j];
            C[i][j+k] = C12[i][j];
            C[i+k][j] = C21[i][j];
            C[i+k][j+k] = C22[i][j];
        }
    }
    return C;
}
 int main() {
    int n;
    cout << "Enter matrix size (power of 2): ";
    cin >> n;

    vector<vector<int>> A(n, vector<int>(n)), B(n, vector<int>(n));
    cout << "Enter elements of Matrix A:\n";
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            cin >> A[i][j];

    cout << "Enter elements of Matrix B:\n";
```

```
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      cin >> B[i][j];
 auto C = strassen(A, B);

  cout << "Resultant Matrix:\n";
  for(int i=0; i<n; i++) {
    for(int j=0; j<n; j++)
      cout << C[i][j] << " ";
    cout << "\n";
  }
  return 0;
}
```

OUTPUT:



Experiment no. 6

Aim : To implement Huffman Coding and analyze its time complexity.

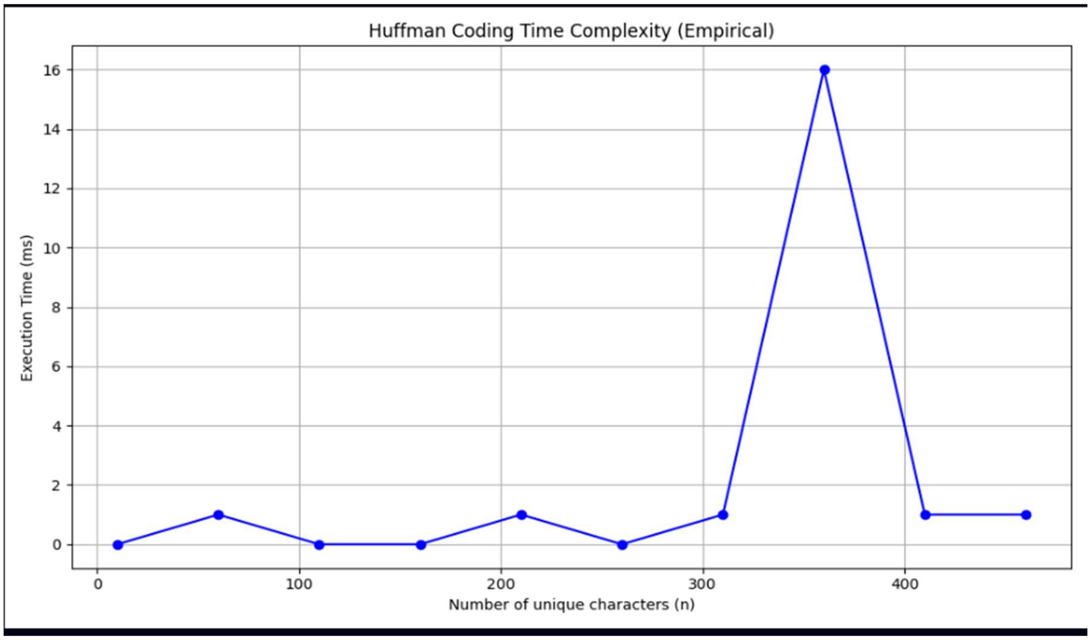
Theory: Huffman Coding is a popular lossless data compression algorithm. It assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters. The idea is to reduce the total number of bits used to represent a string. Characters that occur more frequently have shorter codes while that occur less frequently have longer codes.

CODE:-

```
#include <iostream>
#include <queue>
#include <unordered_map>
#include <vector>
#include <chrono> // For time complexity analysis

using namespace std;
using namespace std::chrono;

// Huffman Tree Node
struct HuffmanNode
{  char ch;
 int freq;
 HuffmanNode *left, *right;

 HuffmanNode(char character, int frequency) {  ch
= character;
 freq = frequency;
 left = right = nullptr;  }
};
struct Compare {
 bool operator()(HuffmanNode* l, HuffmanNode* r)
{  return l->freq > r->freq;
 } };

// Recursive function to generate Huffman codes
void generateCodes(HuffmanNode* root, string code, unordered_map<char, string>&
huffmanCode) {
    if (!root) return;

    // Leaf node
    if (!root->left && !root->right) {
        huffmanCode[root->ch] = code;
    }

    generateCodes(root->left, code + "0", huffmanCode);
    generateCodes(root->right, code + "1", huffmanCode);
}

// Huffman Encoding Algorithm
void huffmanCoding(const unordered_map<char, int>& frequencies) {
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> pq;

    // Step 1: Create a leaf node for each character
    for (auto pair : frequencies) {
        pq.push(new HuffmanNode(pair.first, pair.second));
    }

    // Step 2: Build the Huffman Tree
    while (pq.size() > 1) {
        HuffmanNode* left = pq.top(); pq.pop();
        HuffmanNode* right = pq.top(); pq.pop();

        HuffmanNode* sum = new HuffmanNode('\0', left->freq + right->freq);
        sum->left = left;
        sum->right = right;

        pq.push(sum);
    }
    HuffmanNode* root = pq.top();

    // Step 3: Traverse the tree to generate codes
    unordered_map<char, string> huffmanCode;
    generateCodes(root, "", huffmanCode);

    // Print Huffman Codes
    cout << "\nHuffman Codes:\n";
    for (auto pair : huffmanCode) {
        cout << pair.first << ": " << pair.second << "\n";
    }
}
```

```
int main() {

    unordered_map<char, int> frequencies = {
        {'A', 5}, {'B', 9}, {'C', 12}, {'D', 13}, {'E', 16}, {'F', 45}
    };

    cout << "Building Huffman Tree and Generating Codes...\n";
    auto start = high_resolution_clock::now();
    huffmanCoding(frequencies);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);

    cout << "\nTime taken: " << duration.count() << " microseconds\n";

    return 0;
}
```

Output:-



PROGRAM 6(B)

AIM: To implement Minimum Spanning Tree and analyse its time complexity (Kruskal and Prim's algorithms).

THEORY:

A minimum spanning tree (MST) connects all vertices in a weighted graph with the minimum total edge weight.

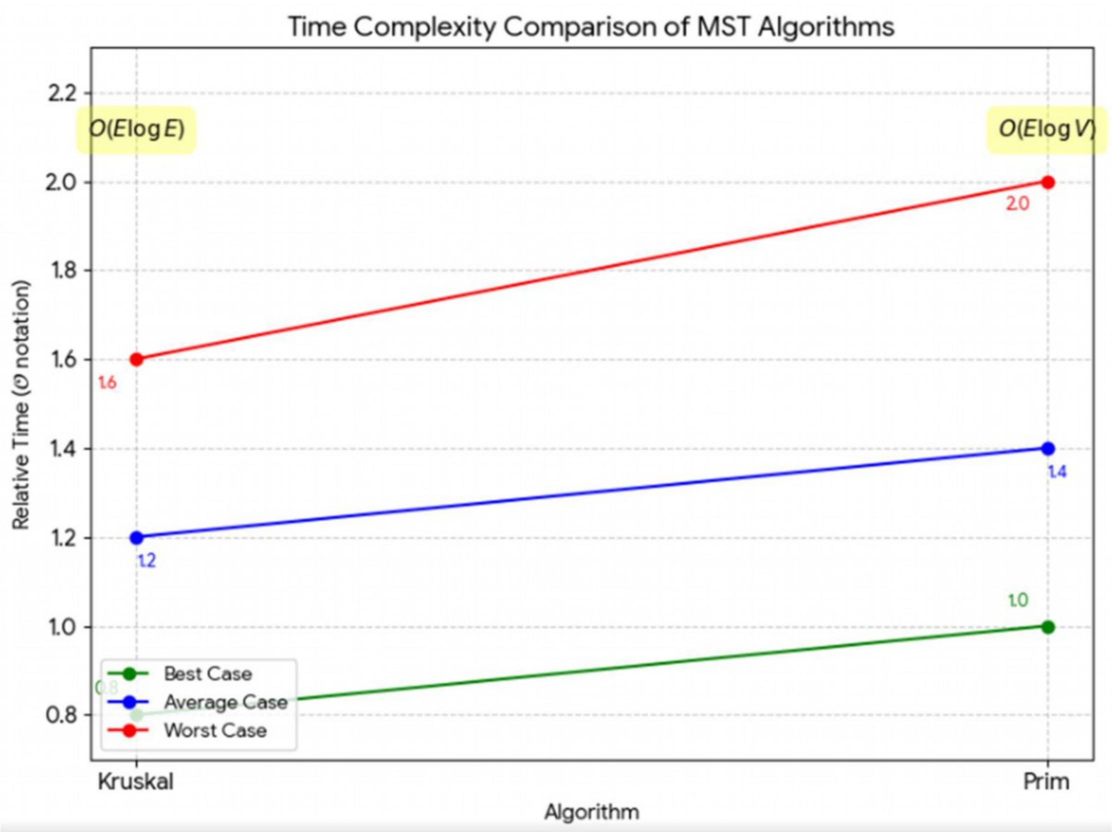
Two common algorithms are Kruskal's and Prim's algorithms.
Kruskal's algorithm sorts all edges and adds them one by one while avoiding cycles.
Prim's algorithm grows the spanning tree starting from an arbitrary vertex by adding the minimum weight edge at each step.

CODE:

```
#include <iostream>

#include <vector>

#include <algorithm>

#include <climits> using

namespace std; struct

Edge {

 int u, v, w;

 bool operator<(Edge const& other) {  return
w < other.w;


 } };


int find(vector<int>& parent, int i) {  if
(parent[i] == i) return i;


 return parent[i] = find(parent, parent[i]); }




void unite(vector<int>& parent, vector<int>& rank, int u, int v) {  u =
find(parent, u);
```

```
    v = find(parent, v);
    if (u != v) {
        if (rank[u] < rank[v]) parent[u] = v;
        else if (rank[u] > rank[v]) parent[v] = u;
        else parent[v] = u, rank[u]++;
    }
}

void kruskalMST(vector<Edge>& edges, int V) {
    sort(edges.begin(), edges.end());
    vector<int> parent(V), rank(V, 0);
    for (int i = 0; i < V; i++) parent[i] = i;
    int cost = 0;
    cout << "Edges in MST (Kruskal):\n";
    for (auto& e : edges) {
        if (find(parent, e.u) != find(parent, e.v)) {
            cost += e.w;
            cout << e.u << " - " << e.v << " : " << e.w << endl;
            unite(parent, rank, e.u, e.v);
        }
    }
    cout << "Total cost: " << cost << endl;
}

void primMST(vector<vector<int>>& graph, int V) {
    vector<int> key(V, INT_MAX), parent(V, -1);
    vector<bool> inMST(V, false);
    key[0] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = -1;
        for (int i = 0; i < V; i++)
            if (!inMST[i] && (u == -1 || key[i] < key[u])) u = i;

        inMST[u] = true;
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && !inMST[v] && graph[u][v] < key[v]) {
                key[v] = graph[u][v];
                parent[v] = u;
            }
        }
    }

    cout << "Edges in MST (Prim):\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << " : " << graph[i][parent[i]] << endl;
}

int main() {
    int V = 4;
    vector<Edge> edges = {{0,1,10}, {0,2,6}, {0,3,5}, {1,3,15}, {2,3,4}};
    kruskalMST(edges, V);

    vector<vector<int>> graph = {
        {0, 10, 6, 5},
        {10, 0, 0, 15},
 {6, 0, 0, 4},


 {5, 15, 4, 0}  };


 primMST(graph, V);
return 0;

}
```

OUTPUT:



PROGRAM 5

AIM: Implement and analyse the time complexity of Heap Sort.

THEORY:

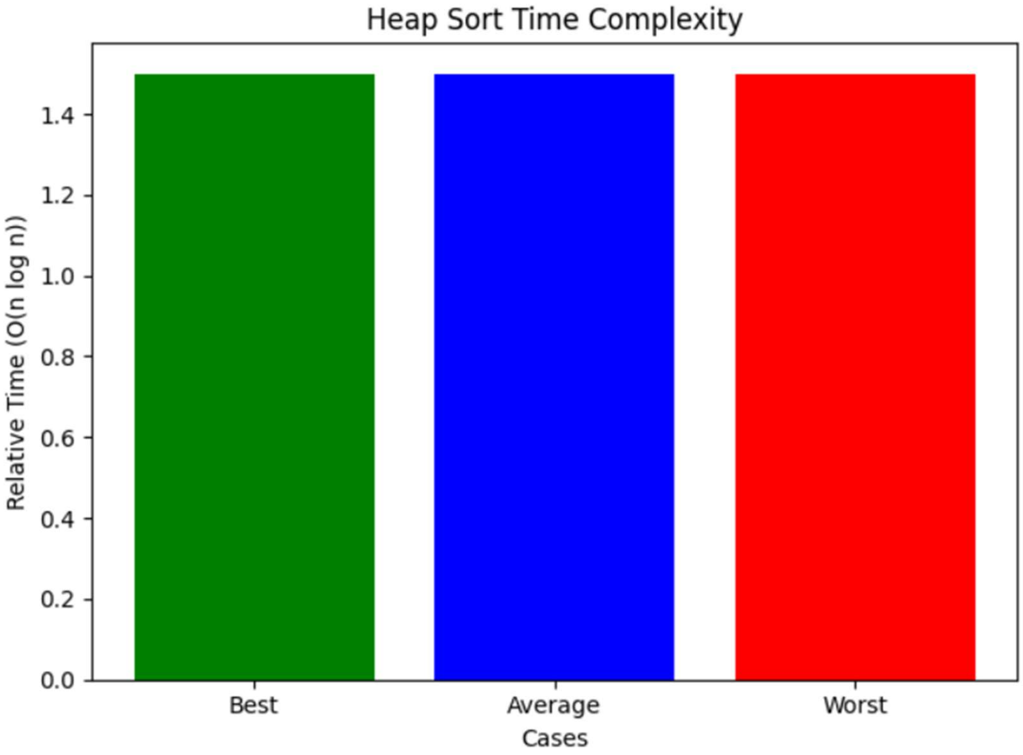
Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure.

It first builds a max-heap and then repeatedly extracts the maximum element to sort the array. Heap Sort has O(n log n) time complexity for all cases.

CODE:

```
#include       <iostream>
using namespace std;

void heapify(int arr[], int n, int i) {
 int largest = i, l = 2*i + 1, r = 2*i + 2;
 if (l < n && arr[l] > arr[largest]) largest = l;
 if (r < n && arr[r] > arr[largest]) largest = r;  if
(largest != i) {
 swap(arr[i], arr[largest]);
heapify(arr, n, largest);

 } }


void heapSort(int arr[], int n) {
 for (int i = n/2 - 1; i >= 0; i--) heapify(arr, n, i);  for (int
i = n - 1; i > 0; i--) {
 swap(arr[0], arr[i]);
heapify(arr, i, 0);

 } }


int main() {
 int arr[] = {12, 11, 13, 5, 6, 7};  int n
= 6;
 heapSort(arr, n);
 cout << "Sorted array: ";
 for (int i = 0; i < n; i++) cout << arr[i] << " ";  return
0;
}
```

OUTPUT:


Heap Sort Time Complexity
Relative Time (O(n log n))
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
Best
Average
Worst
Cases

PROGRAM 7

AIM: To implement Dijkstra's and Bellman-Ford algorithms and analyse their time complexities.

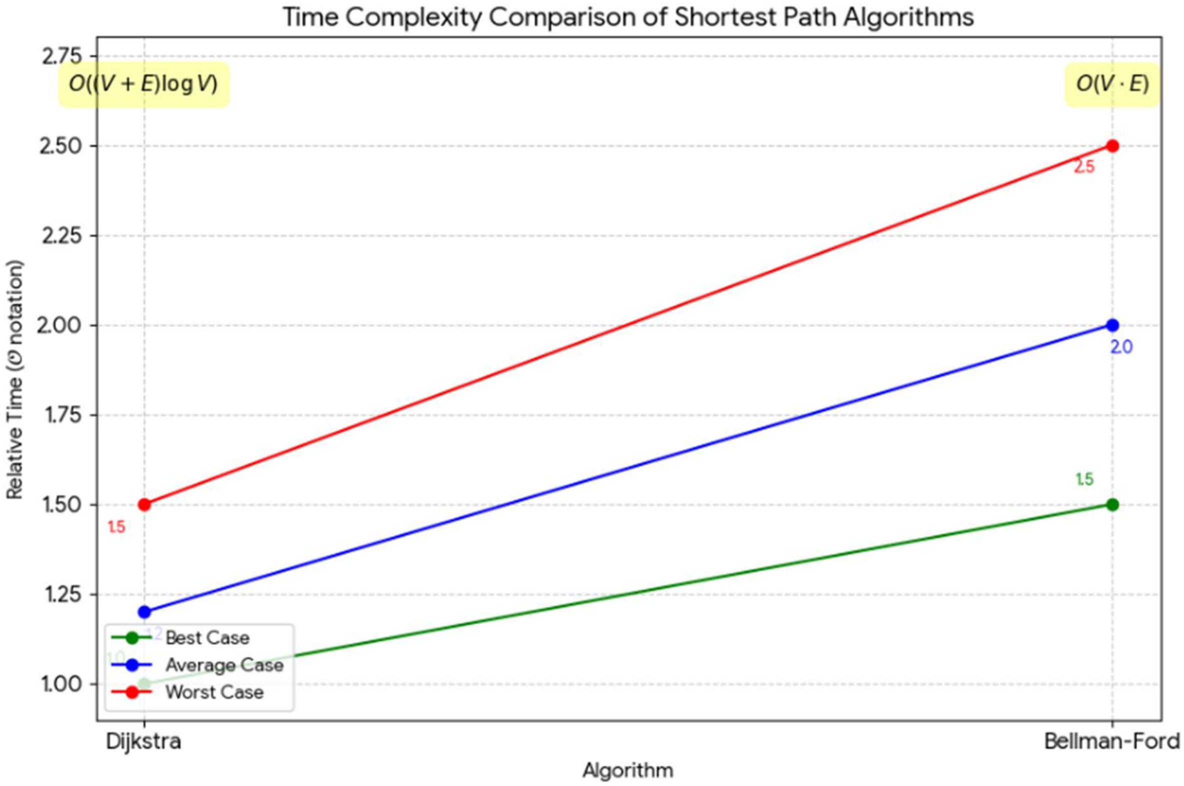
THEORY:

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with non-negative weights.
Bellman-Ford algorithm also finds shortest paths but can handle negative edge weights.
Dijkstra's algorithm has O(V^2) or O(E + V log V) with a priority queue, while Bellman-Ford has O(VE) complexity.

CODE:

```
#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;


void dijkstra(int V, vector<vector<pair<int,int>>>& adj, int src) {

    vector<int> dist(V, INT_MAX);

    dist[src] = 0;

    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;

    pq.push({0, src});


    while (!pq.empty()) {

        int u = pq.top().second;

        pq.pop();


        for (auto [v, w] : adj[u]) {

            if (dist[u] + w < dist[v]) {

                dist[v] = dist[u] + w;

                pq.push({dist[v], v});
            }
        }
    }

    cout << "Dijkstra’s Shortest Distances:\n";
    for (int i = 0; i < V; i++)
        cout << i << " : " << dist[i] << endl;
}
void bellmanFord(int V, vector<vector<int>>& edges, int src) {
    vector<int> dist(V, INT_MAX);
    dist[src] = 0;
    for (int i = 0; i < V - 1; i++)
        for (auto e : edges)
            if (dist[e[0]] != INT_MAX && dist[e[0]] + e[2] < dist[e[1]])
                dist[e[1]] = dist[e[0]] + e[2];

    cout << "Bellman-Ford Shortest Distances:\n";
    for (int i = 0; i < V; i++)
        cout << i << " : " << dist[i] << endl;

int main() {
    int V = 5;
    vector<vector<pair<int,int>>> adj(V);
    adj[0] = {{1, 2}, {2, 4}};
    adj[1] = {{2, 1}, {3, 7}};
    adj[2] = {{4, 3}};
    adj[3] = {{4, 1}};
    dijkstra(V, adj, 0);
    vector<vector<int>> edges = {{0,1,2},{0,2,4},{1,2,1},{1,3,7},{2,4,3},{3,4,1}};

    bellmanFord(V, edges, 0);

    return 0;

}
```

OUTPUT:



PROGRAM 8

AIM: Implement N Queen's problem using Backtracking.

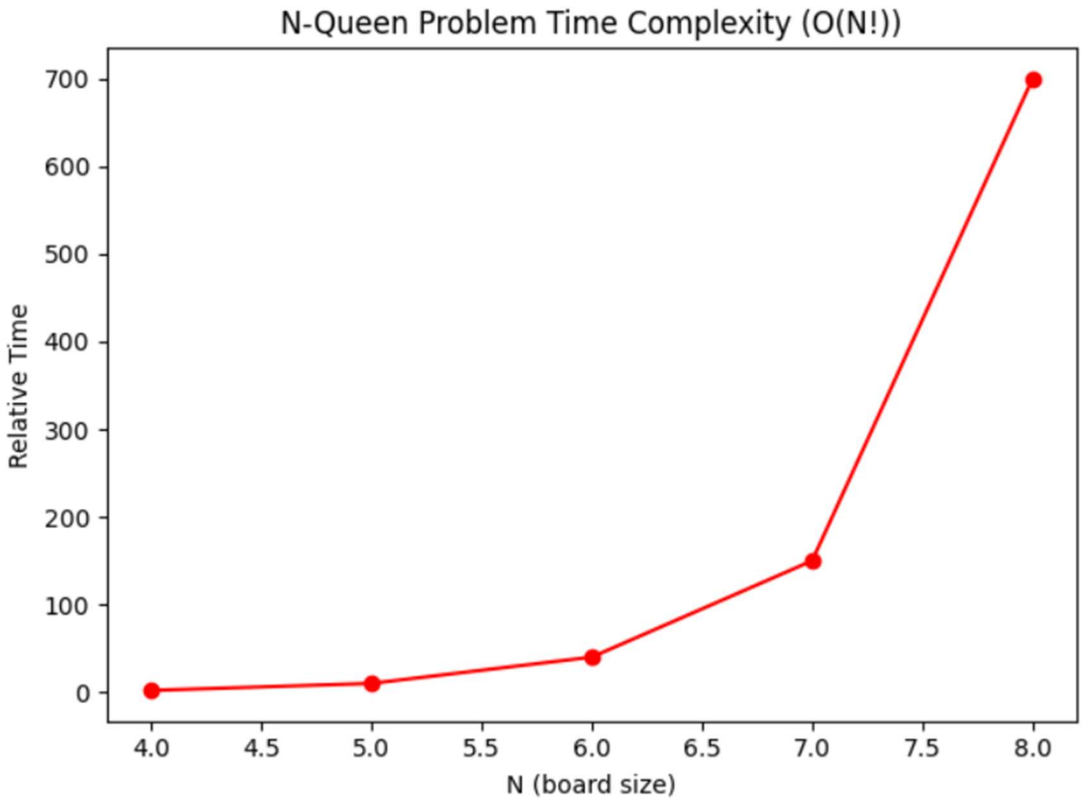
THEORY:

The N Queen's problem aims to place N queens on an N×N chessboard such that no two queens threaten each other.
A backtracking algorithm incrementally places queens column by column and backtracks when no valid position is found.

CODE:

```
#include <iostream>
using namespace std;

#define N 4

void printSolution(int board[N][N]) {
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++)
         cout << board[i][j] << " ";
      cout << endl;
   }
}

bool isSafe(int board[N][N], int row, int col) {
   for (int i = 0; i < col; i++)
      if (board[row][i]) return false;
   for (int i=row,j=col; i>=0 && j>=0; i--,j--)
      if (board[i][j]) return false;
   for (int i=row,j=col; i<N && j>=0; i++,j--)
      if (board[i][j]) return false;
   return true;
}

bool solveNQUtil(int board[N][N], int col) {
   if (col >= N) return true;
   for (int i = 0; i < N; i++) {
      if (isSafe(board, i, col)) {
         board[i][col] = 1;
         if (solveNQUtil(board, col + 1)) return true;
         board[i][col] = 0;
      }
   }
   return false;
}

bool solveNQ() {
    int board[N][N] = {0};
    IF (!SOLVENQUTIL(BOARD, 0)) {
        cout << "Solution does not exist";
        return false;
    }
    printSolution(board);
    return true;
}

int main() {
    solveNQ();
    return 0;
}
```

OUTPUT:



PROGRAM 9(A)

AIM: To implement Matrix Chain Multiplication and analyse its time complexity.

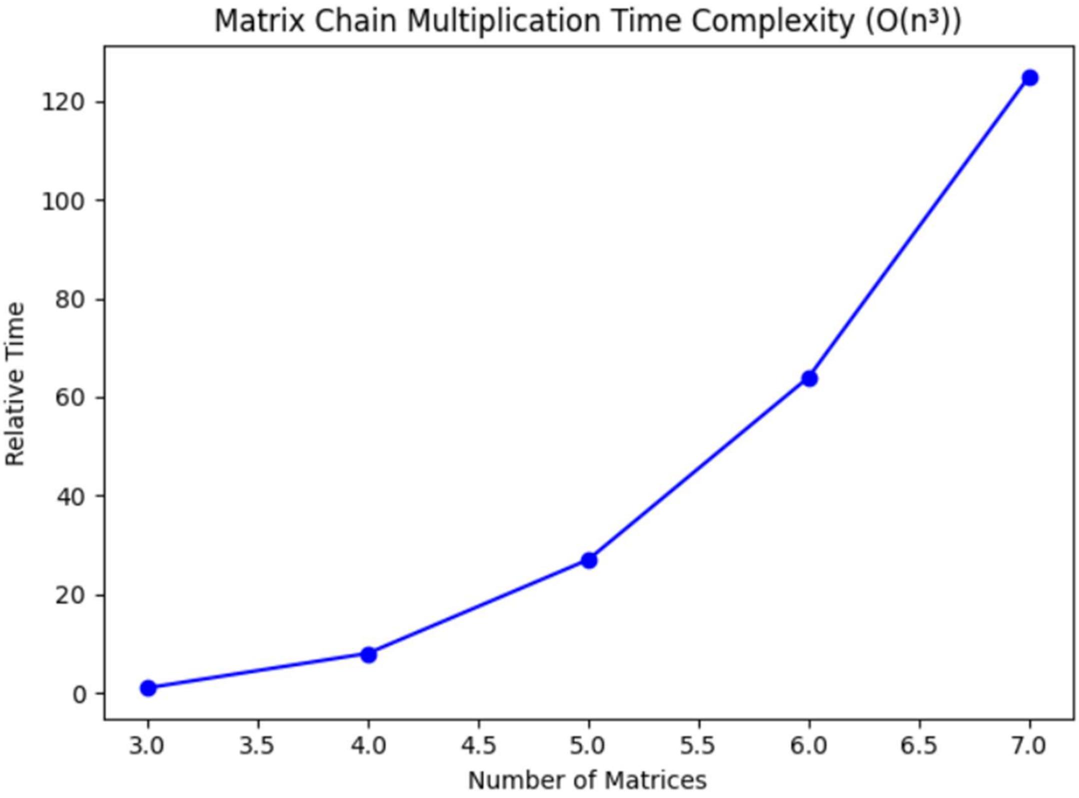
THEORY:

Matrix Chain Multiplication problem determines the most efficient way to multiply a chain of matrices.
Dynamic programming is used to find the minimum number of scalar multiplications needed.
The time complexity is O(n^3).

CODE:

```
#include <iostream>
#include <climits>
using namespace std;

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 + 1; i++) {
         int j = i + L - 1;
         m[i][j] = INT_MAX;
         for (int k = i; k <= j - 1; 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[] = {1, 2, 3, 4};
   int n = 4;
   cout << "Minimum number of multiplications is " << matrixChainOrder(arr, n);
   return 0;
}
```

OUTPUT:


Matrix Chain Multiplication Time Complexity (O(n³))
Relative Time
Number of Matrices
0
20
40
60
80
100
120
3.0
3.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0

PROGRAM 9(B)

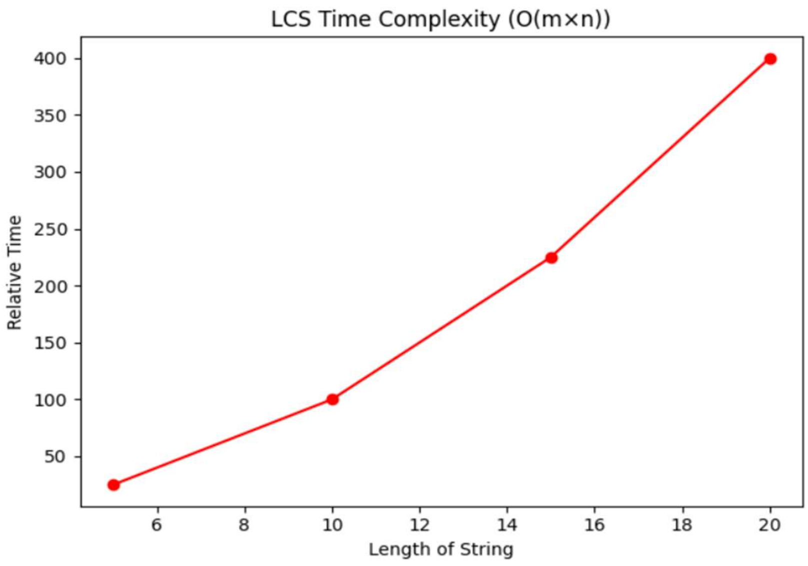
AIM: To implement Longest Common Subsequence (LCS) and analyse its time complexity.

THEORY: The Longest Common Subsequence (LCS) problem finds the longest sequence present in both given sequences in the same order.
Dynamic programming is used to compute LCS in O(mn) time complexity.

CODE:

```
#include <iostream>
#include <string>
using namespace std;

int lcs(string X, string Y, int m, int n) {
    int L[m+1][n+1];
    for (int i=0; i<=m; i++) {
        for (int j=0; j<=n; j++) {
            if (i==0 || j==0) L[i][j] = 0;
            else if (X[i-1] == Y[j-1]) L[i][j] = L[i-1][j-1] + 1;
            else L[i][j] = max(L[i-1][j], L[i][j-1]);
        }
    }
    return L[m][n];
}

int main() {
    string X = "AGGTAB", Y = "GXTXAYB";
    cout << "Length of LCS is " << lcs(X, Y, X.length(), Y.length());
    return 0;
}
```

OUTPUT:



PROGRAM 10

AIM: To implement Naïve String Matching, Rabin-Karp, and Knuth-Morris-Pratt (KMP) algorithms and analyse their time complexities.

THEORY:

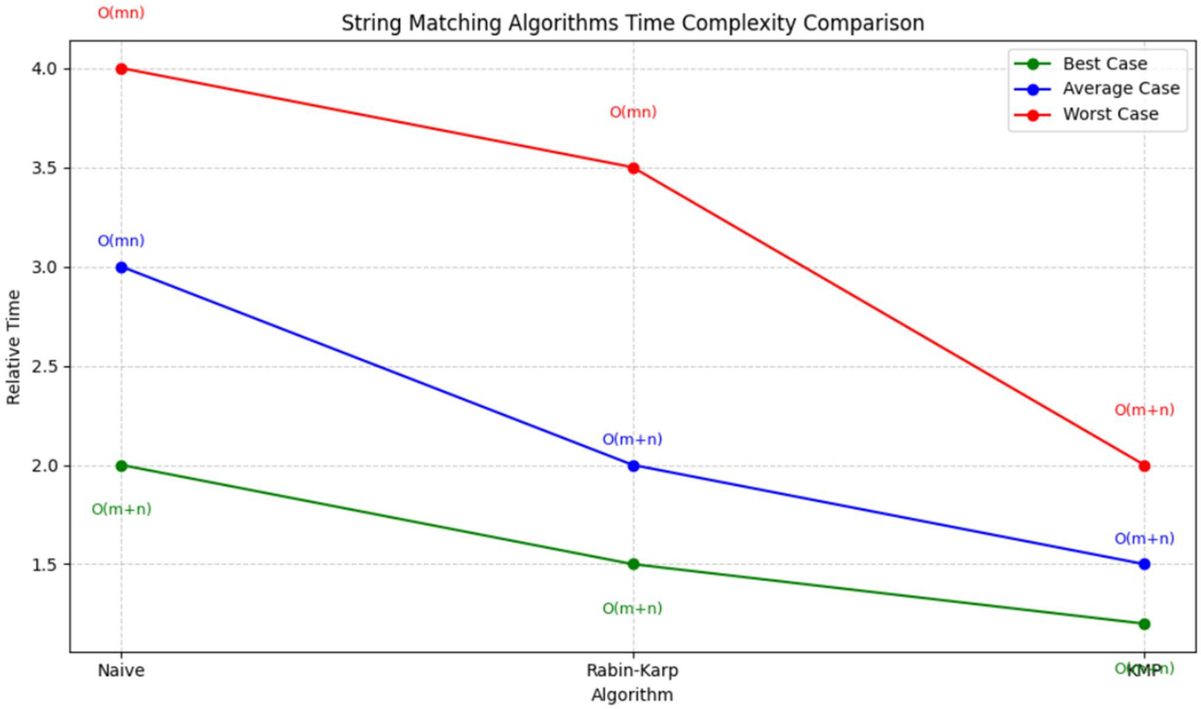
String Matching algorithms are used to find occurrences of a pattern in a text.
Naïve algorithm checks for the pattern at all positions.
Rabin-Karp uses hashing for efficiency, and KMP preprocesses the pattern to skip unnecessary comparisons.
Time complexities are: Naïve - O((n−m+1)m), Rabin-Karp - O(n+m), and KMP - O(n+m).

CODE:

```
#include <iostream>

#include <string>

using namespace std;

void naiveSearch(string txt, string pat) {

  int n = txt.size(), m = pat.size();

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

    int j;

    for (j = 0; j < m; j++)

      if (txt[i + j] != pat[j]) break;

    if (j == m) cout << "Pattern found at index " << i << endl;

  }

}

void rabinKarp(string txt, string pat, int q) {

  int n = txt.size(), m = pat.size();

  int p = 0, t = 0, h = 1;

  for (int i = 0; i < m - 1; i++)

    h = (h * d) % q;


  for (int i = 0; i < m; i++) {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }

    for (int i = 0; i <= n - m; i++) {
        if (p == t) {
            int j;
            for (j = 0; j < m; j++)
                if (txt[i + j] != pat[j]) break;
            if (j == m) cout << "Pattern found at index " << i << endl;
        }
        if (i < n - m) {
            t = (d * (t - txt[i] * h) + txt[i + m]) % q;
            if (t < 0) t += q;
        }
    }
}
void computeLPS(string pat, int m, int* lps) {
    int len = 0;
    lps[0] = 0;
    int i = 1;
    while (i < m) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0)
```

```
                len = lps[len - 1];
            else
                lps[i++] = 0;
        }
    }
}

void KMPSearch(string pat, string txt) {
    int m = pat.size(), n = txt.size();
    int lps[m];
    computeLPS(pat, m, lps);
    int i = 0, j = 0;
    while (i < n) {
        if (pat[j] == txt[i]) i++, j++;
        if (j == m) {
            cout << "Pattern found at index " << i - j << endl;
            j = lps[j - 1];
        } else if (i < n && pat[j] != txt[i]) {
            if (j != 0) j = lps[j - 1];
            else i++;
        }
    }
}
int main() {
    string txt = "AABAACAADAABAABA";
    string pat = "AABA";
    cout << "Naive Search:\n";
    naiveSearch(txt, pat);
    cout << "\nRabin-Karp:\n";

    rabinKarp(txt, pat, 101);

    cout << "\nKMP Algorithm:\n";

    KMPSearch(pat, txt);

    return 0;

}
```

OUTPUT:


String Matching Algorithms Time Complexity Comparison
Best Case
Average Case
Worst Case
Relative Time
O(mn)
O(mn)
O(mn)
O(m+n)
O(m+n)
O(m+n)
O(m+n)
O(m+n)
O(m+n)
Naive
Rabin-Karp
KMP
Algorithm