PROGRAM 1

Aim: To implement following algorithm using array as a data structure and analyse its time complexity.

a) Insertion Sort

Theory:

Insertion Sort is a simple and intuitive sorting algorithm. It builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, it has some advantages, such as simplicity, stability (it preserves the relative order of equal elements), and efficiency for small datasets or nearly sorted data.

Algorithm Steps:

1. Initialization: Start with the second element (index 1) in the array as the current key.
2. Comparison: Compare the current key with elements in the sorted portion of the array (i.e., the elements to its left).
3. Shifting: If the key element is smaller than its predecessor, compare it with the elements before. Shift all larger elements one position to the right to make space for the swapped element.
4. Insertion: Insert the key element at the correct position in the sorted part of the array.
5. Repeat: Move to the next element and repeat the process until the entire array is sorted.

Time Complexity Analysis:

- Best Case: O(n)

  Occurs when the array is already sorted. The inner loop is never executed, so each element is compared once, resulting in linear time complexity.

- Average Case: $O(n^2)$

  On average, half of the elements in the array will need to be shifted for each element being inserted. This results in a quadratic time complexity.

- Worst Case: $O(n^2)$

  The worst-case scenario occurs when the array is sorted in reverse order. Every insertion requires shifting all the previously sorted elements, leading to a quadratic time complexity.

Program:

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

using namespace std;
using namespace std::chrono;

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        //move one position right if greater than key
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand();
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    int sizes[] = {100, 500, 1000, 1500};

    cout<<"For Insertion Sort:"<<endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        int* arr = new int[size];
        //sri
        generateRandomArray(arr, size);

        auto start = high_resolution_clock::now();
        insertionSort(arr, size);
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();

        cout << "The elapsed time for " << size << " elements is " << time_spent << "
nanoseconds" << endl;

        delete[] arr;
    }
```

```
    return 0;
}
```

```
For Insertion Sort:
The elapsed time for 100 elements is 17333 nanoseconds
The elapsed time for 500 elements is 329708 nanoseconds
The elapsed time for 1000 elements is 1101333 nanoseconds
The elapsed time for 1500 elements is 2021542 nanoseconds
```

Output:

Graph:

Learning Outcomes:

b) Selection Sort

Theory:

Selection Sort is a simple comparison-based sorting algorithm. It works by dividing the input array into a sorted and an unsorted region, repeatedly selecting the smallest (or largest, depending on the order) element from the unsorted region, and moving it to the end of the sorted region.

Algorithm Steps:

1. Initialization: Start with the entire array as the unsorted region.
2. Find the Minimum Element: Iterate through the unsorted region to find the minimum element.
3. Swap: Swap the minimum element found with the first element of the unsorted region.
4. Repeat: Move the boundary between the sorted and unsorted regions one element to the right and repeat the process until the entire array is sorted.

Time Complexity Analysis:

- Best Case: $O(n^2)$

  Selection Sort always performs $n(n-1)/2$ comparisons, regardless of the initial arrangement of the array. Thus, the best-case time complexity is $O(n^2)$.

- Average Case: $O(n^2)$

  On average, the number of comparisons is the same as in the worst case, leading to a quadratic time complexity.

- Worst Case: $O(n^2)$

  The worst-case scenario also results in $O(n^2)$ time complexity, as the algorithm always performs the same number of comparisons and swaps regardless of the input array's order.

Program:

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

using namespace std;
using namespace std::chrono;

void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; ++i) {
        // Find the minimum element in the unsorted part of the array
        int minIndex = i;
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap the found minimum element with the first element of unsorted part
        if (minIndex != i) {
            swap(arr[i], arr[minIndex]);
        }
    }
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand();
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    int sizes[] = {100, 500, 1000, 1500};

    cout<<"\n\nFor Selection Sort:"<<endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        int* arr = new int[size];
        //sri
        generateRandomArray(arr, size);

        auto start = high_resolution_clock::now();
        selectionSort(arr, size);
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();

        cout << "The elapsed time for " << size << " elements is " << time_spent << "
nanoseconds" << endl;
```

```
        delete[] arr; // Deallocate the array
    }

    cout<<endl<<endl;
    return 0;
}
```

Output:

```
For Selection Sort:
The elapsed time for 100 elements is 29625 nanoseconds
The elapsed time for 500 elements is 601667 nanoseconds
The elapsed time for 1000 elements is 2192625 nanoseconds
The elapsed time for 1500 elements is 4595000 nanoseconds
```

Graph:

Learning Outcomes:

c) Bubble Sort

Theory:

Bubble Sort is one of the simplest sorting algorithms that works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. This process is repeated until the list is sorted. The name "Bubble Sort" comes from the way larger elements "bubble" to the top of the list.

Algorithm Steps:

1. Initialization: Start from the beginning of the array.
2. Pairwise Comparison: Compare each pair of adjacent elements in the array. If the current element is greater than the next element, swap them.
3. Pass Through the Array: After each pass through the array, the largest unsorted element is moved to its correct position at the end of the array.
4. Repeat: Repeat the process for the remaining unsorted portion of the array until no swaps are needed, indicating the array is sorted.

Time Complexity Analysis:

- Best Case: O(n)

  The best case occurs when the array is already sorted. The algorithm only needs one pass through the array to confirm that no swaps are needed, resulting in linear time complexity.

- Average Case: $O(n^2)$

  On average, the algorithm needs to perform $n(n-1)/2$ comparisons and a number of swaps, leading to quadratic time complexity.

- Worst Case: $O(n^2)$

  The worst case occurs when the array is sorted in reverse order. In this case, the algorithm must make the maximum number of comparisons and swaps, resulting in quadratic time complexity.

Program:

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

using namespace std;
using namespace std::chrono;

void bubbleSort(int arr[], int n) {
   int flag=0;
   for (int i = 0; i < n - 1; ++i) {
      flag=0;
      for (int j = 0; j < n - i - 1; ++j) {
         if (arr[j] > arr[j + 1]) {
            // Swap if the element found is greater than the next element
            swap(arr[j], arr[j + 1]);
            flag=1;
         }
      }
      // If no two elements were swapped by inner loop, then the array is sorted
      if (flag!=1) break;
   }
}

void generateRandomArray(int arr[], int size) {
   for (int i = 0; i < size; ++i) {
      arr[i] = rand();
   }
}

int main() {
   srand(static_cast<unsigned int>(time(0)));
   int sizes[] = {100, 500, 1000, 1500};

   cout<<"\n\nFor Bubble Sort:"<<endl;

   for (int i = 0; i < 4; ++i) {
      int size = sizes[i];
      int* arr = new int[size];
      //sri
      generateRandomArray(arr, size);

      auto start = high_resolution_clock::now();
      bubbleSort(arr, size);
      auto end = high_resolution_clock::now();

      auto time_spent = duration_cast<nanoseconds>(end - start).count();

      cout << "The elapsed time for " << size << " elements is " << time_spent << "
nanoseconds" << endl;
```

```
        delete[] arr; // Deallocate the array
    }

    cout<<endl<<endl;
    return 0;
}
```

```
For Bubble Sort:
The elapsed time for 100 elements is 47791 nanoseconds
The elapsed time for 500 elements is 938333 nanoseconds
The elapsed time for 1000 elements is 3726833 nanoseconds
The elapsed time for 1500 elements is 7206709 nanoseconds
```

Output:

Graph:

Learning Outcomes:

d) Quick Sort

Theory:

Quick Sort is a highly efficient and widely used sorting algorithm. It employs the divide-and-conquer strategy to sort elements, making it significantly faster than simpler algorithms like Bubble Sort or Selection Sort, especially for large datasets.

Algorithm Steps:

1. Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, middle element, or a random element).
2. Partitioning: Rearrange the array such that all elements less than the pivot are placed before it, and all elements greater than the pivot are placed after it. This step is known as partitioning. After partitioning, the pivot is in its final sorted position.
3. Recursion: Recursively apply the above steps to the sub-arrays formed by dividing the array at the pivot's position—one sub-array contains elements less than the pivot, and the other contains elements greater than the pivot.
4. Base Case: The recursion ends when the sub-array has one or zero elements, which are already sorted by definition.

Time Complexity Analysis:

- Best Case: O(n logn)

  The best-case scenario occurs when the pivot always divides the array into two equal halves. This leads to nlogn levels of recursion, with n comparisons at each level, resulting in O(nlogn) time complexity.

- Average Case: O(n logn)

  On average, Quick Sort performs O(nlogn) comparisons. The division of the array is typically balanced, making it an efficient sorting algorithm for most cases.

- Worst Case: $O(n^2)$

  The worst-case scenario occurs when the pivot chosen is always the smallest or largest element, resulting in an extremely unbalanced partitioning. This can happen if the array is already sorted or nearly sorted. The time complexity in this case is quadratic, $O(n^2)$.

Program:

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

using namespace std;
using namespace std::chrono;

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 generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand();
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    int sizes[] = {100, 500, 1000, 1500};

    cout<<"\n\nFor Quick Sort:"<<endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        int* arr = new int[size];
        //sri
```

```
        generateRandomArray(arr, size);

        auto start = high_resolution_clock::now();
        quickSort(arr, 0, size - 1);
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();

        cout << "The elapsed time for " << size << " elements is " << time_spent << "
nanoseconds" << endl;

        delete[] arr;
    }

    cout<<endl<<endl;
    return 0;
}
```

```
For Quick Sort:
The elapsed time for 100 elements is 21958 nanoseconds
The elapsed time for 500 elements is 76167 nanoseconds
The elapsed time for 1000 elements is 153625 nanoseconds
The elapsed time for 1500 elements is 233750 nanoseconds
```

Output:

Graph:

Learning Outcomes:

e) Merge Sort

Theory:

Merge Sort is an efficient, stable, and comparison-based sorting algorithm that uses the divide-and-conquer strategy. It works by recursively splitting the array into smaller sub-arrays until each sub-array has only one element, and then merging those sub-arrays in a sorted manner to produce the final sorted array.

Algorithm Steps:

1. Divide: Split the array into two halves, typically at the midpoint.
2. Conquer (Recursion): Recursively apply Merge Sort to both halves of the array.
3. Merge: Merge the two sorted halves back together into a single sorted array. The merging process involves comparing elements from each half and arranging them in order.
4. Base Case: The recursion terminates when the sub-array has only one element, which is inherently sorted.

Time Complexity Analysis:

- Best Case: O(nlogn)

  Merge Sort consistently divides the array into two equal parts, performing a merge process for each division, leading to a time complexity of O(nlogn) in the best case.

- Average Case: O(nlogn)

  Regardless of the initial arrangement of elements, Merge Sort always requires O(nlogn) time, making it highly efficient even on average.

- Worst Case: O(nlogn)

  The worst-case time complexity is also O(nlogn) since the number of operations remains the same across all cases.

Program:

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

using namespace std;
using namespace std::chrono;

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int* L = new int[n1];
    int* R = new int[n2];

    for (int i = 0; i < n1; ++i)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[mid + 1 + j];

    int i = 0;
    int j = 0;
    int k = left;

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

    while (i < n1) {
        arr[k] = L[i];
        ++i;
        ++k;
    }

    while (j < n2) {
        arr[k] = R[j];
        ++j;
        ++k;
    }

    delete[] L;
    delete[] R;
}
```

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand();
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    int sizes[] = {100, 500, 1000, 1500};

    cout << "\n\nFor Merge Sort:" << endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        int* arr = new int[size];
              //sri
        generateRandomArray(arr, size);

        auto start = high_resolution_clock::now();
        mergeSort(arr, 0, size - 1);
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();

        cout << "The elapsed time for " << size << " elements is " << time_spent << "
nanoseconds" << endl;

        delete[] arr;
    }

    cout<<endl<<endl;
    return 0;
}
```

```
For Merge Sort:
The elapsed time for 100 elements is 48750 nanoseconds
The elapsed time for 500 elements is 253042 nanoseconds
The elapsed time for 1000 elements is 502250 nanoseconds
The elapsed time for 1500 elements is 780291 nanoseconds
```

Output:

Graph:

Learning Outcomes:

f) Heap Sort

Theory:

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to sort elements. It is an in-place algorithm, meaning it doesn't require additional memory, and it is not a stable sort. Heap Sort is particularly useful for its efficiency in sorting large datasets and its ability to guarantee a worst-case time complexity of O(nlogn).

Algorithm Steps:

1. Build a Max-Heap: Convert the array into a max-heap, a complete binary tree where the value of each node is greater than or equal to the values of its children. This ensures the largest element is at the root of the heap.
2. Swap and Reduce:
    - Swap the root (the largest element) with the last element in the array, effectively moving the largest element to its final sorted position.
    - Reduce the heap size by one (excluding the last element from the heap) and heapify the root to maintain the max-heap property.
3. Repeat: Continue the process of swapping the root with the last element of the heap and reducing the heap size until the heap is empty and the array is fully sorted.

Time Complexity Analysis:

- Best Case: O(nlogn)

  Even in the best case, Heap Sort requires O(nlogn)operations because each element must be inserted into the heap and extracted, both of which are O(logn) operations.

- Average Case: O(nlogn)

  On average, Heap Sort performs consistently at O(nlogn) since the heapification process ensures a balanced binary heap.

- Worst Case: O(nlogn)

  The worst-case time complexity is O(nlogn), as the operations required to maintain the heap structure are the same regardless of the initial order of elements.

Program:

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

using namespace std;
using namespace std::chrono;

void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    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);
    }
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand();
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    int sizes[] = {100, 500, 1000, 1500};

    cout << "\n\nFor Heap Sort:" << endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        int* arr = new int[size];
```

```
        //sri
        generateRandomArray(arr, size);

        auto start = high_resolution_clock::now();
        heapSort(arr, size);
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();

        cout << "The elapsed time for " << size << " elements is " << time_spent << "
nanoseconds" << endl;

        delete[] arr;
    }

    cout<<endl<<endl;
    return 0;
}
```

```
For Heap Sort:
The elapsed time for 100 elements is 15500 nanoseconds
The elapsed time for 500 elements is 113000 nanoseconds
The elapsed time for 1000 elements is 471291 nanoseconds
The elapsed time for 1500 elements is 702500 nanoseconds
```

Output:

Graph:

Learning Outcomes:

PROGRAM 2

Aim: To implement linear search and binary search and analyse its time complexity.

Linear Search

Theory:

Linear Search is the simplest search algorithm used to find a particular element in an array or list. The algorithm works by sequentially checking each element of the array until it finds the target element or reaches the end of the array.

Algorithm Steps:

1. Start from the Beginning: Begin at the first element of the array.
2. Compare Each Element: Compare the target element with the current element of the array.
3. Check for Match: If the current element matches the target, return the index of the element. If it doesn't match, move to the next element.
4. Continue Until Found or End: Repeat the comparison process until the element is found or until the end of the array is reached.
5. Return Result: If the target element is found, return its index. If the target element is not found after checking all elements, return an indication (like -1) that the element is not present in the array.

Time Complexity Analysis:

- Best Case: O(1)

  The best-case scenario occurs when the target element is the first element in the array. The algorithm only needs one comparison to find the target.

- Average Case: O(n)

  On average, the algorithm will have to search through half of the array before finding the target, leading to a time complexity of O(n)O(n)O(n), where nnn is the number of elements in the array.

- Worst Case: O(n)

  The worst-case scenario occurs when the target element is the last element in the array or is not present at all. The algorithm must check every element, resulting in O(n)O(n)O(n) time complexity.

Program:

```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;

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

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand();
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    int sizes[] = {100, 500, 1000, 1500};

    cout << "\n\nFor Linear Search:" << endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        int* arr = new int[size];

        generateRandomArray(arr, size);
        //sri
        auto start = high_resolution_clock::now();
        linearSearch(arr, size, rand());
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();

        cout << "The elapsed time for " << size << " elements is " << time_spent << "
nanoseconds" << endl;

        delete[] arr;
    }
```

```
    cout<<endl<<endl;
    return 0;
}
```

```
For Linear Search:
The elapsed time for 100 elements is 833 nanoseconds
The elapsed time for 500 elements is 2417 nanoseconds
The elapsed time for 1000 elements is 4292 nanoseconds
The elapsed time for 1500 elements is 6375 nanoseconds
```

Output:

Graph:
Binary Search

Theory
Binary Search is an efficient search algorithm that finds the position of a target value within a sorted array or list. It works by repeatedly dividing the search interval in half, which allows it to achieve a time complexity of O(logn), making it much faster than linear search for large datasets.

Algorithm Steps:

1. Initialize Pointers: Set two pointers, **low** and high, representing the start and end of the array segment being searched.
2. Calculate Midpoint: Compute the midpoint index as **mid = (low + high) // 2.**
3. Compare with Target: Compare the target value with the element at the midpoint index.

4. Adjust Search Range:
   - If the target value equals the element at the midpoint, return the midpoint index (target found).
   - If the target value is less than the element at the midpoint, adjust the **high** pointer to **mid - 1** (search the left half).
   - If the target value is greater than the element at the midpoint, adjust the **low** pointer to **mid + 1** (search the right half).
5. Repeat Until Found or Exhausted: Continue the process until the **low** pointer exceeds the **high** pointer.
6. Return Result: If the target value is not found, return an indication (like **-1**) that the target is not present in the array.

Time Complexity Analysis:

- Best Case: O(1)

  The best-case scenario occurs when the target element is at the midpoint index of the array. The algorithm finds the target in one comparison.
- Average Case: O(logn)

  On average, Binary Search performs nlogn comparisons, where n is the number of elements in the array. The array is divided in half at each step.
- Worst Case: O(logn)

  The worst-case time complexity occurs when the target is not present, and the search interval needs to be halved repeatedly until it is exhausted.
- Space Complexity: O(1)

  Binary Search is an in-place algorithm that requires only a constant amount of extra space, regardless of the input size.

Program:

```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;

int binarySearch(int arr[], int n, int key)
{
   int low = 0;
   int high = n - 1;
   while (low <= high)
   {
      int mid = (low + high) / 2;
      if (arr[mid] == key)
      {
         return mid;
```

```
        }
        else if (arr[mid] < key)
        {
            low = mid + 1;
        }
        else
        {
            high = mid - 1;
        }
    }
    return -1;
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand();
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    int sizes[] = {100, 500, 1000, 1500};

    cout << "\n\nFor Binary Search:" << endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        int* arr = new int[size];
        //sri
        generateRandomArray(arr, size);

        auto start = high_resolution_clock::now();
        binarySearch(arr, size, rand());
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();

        cout << "The elapsed time for " << size << " elements is " << time_spent << "
nanoseconds" << endl;

        delete[] arr;
    }

    cout<<endl<<endl;
    return 0;
}
```

```
For Binary Search:
The elapsed time for 100 elements is 125 nanoseconds
The elapsed time for 500 elements is 250 nanoseconds
The elapsed time for 1000 elements is 166 nanoseconds
The elapsed time for 1500 elements is 209 nanoseconds
```

Output:

Graph:

Learning Outcomes:

PROGRAM 3

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

Theory:

Huffman Coding is a popular algorithm used for lossless data compression. The key idea behind Huffman Coding is to assign variable-length codes to input characters based on their frequencies of occurrence. Characters with higher frequencies are assigned shorter codes, while characters with lower frequencies are assigned longer codes. This ensures that the overall size of the encoded data is minimized.

Steps to Implement Huffman Coding:

1. **Frequency Calculation:**
   - Calculate the frequency of occurrence of each character in the input data.
2. **Build a Min-Heap:**
   - Treat each character and its frequency as a leaf node in a binary tree. Create a min-heap (priority queue) where the key is the frequency of the character.
3. **Build the Huffman Tree:**
   - While there are more than one node in the heap, perform the following steps:
     - Extract the two nodes with the lowest frequency from the min-heap.
     - Create a new internal node with a frequency equal to the sum of the two nodes' frequencies. The two nodes become the left and right children of this new node.
     - Insert the new node back into the min-heap.
   - Repeat this process until only one node remains in the heap, which becomes the root of the Huffman tree.
4. **Generate Codes:**
   - Traverse the Huffman tree from the root, assigning `0` for left edges and `1` for right edges. This generates the Huffman codes for each character.
5. **Encode the Input Data:**
   - Replace each character in the input data with its corresponding Huffman code.
6. **Decode the Encoded Data:**
   - The encoded data can be decoded by traversing the Huffman tree using the binary Huffman codes.

Overall Time Complexity:

- Building the min-heap and Huffman tree takes **O(d log d)**.
- The frequency calculation and encoding take **O(n)**.

Thus, the overall time complexity of Huffman coding is:

- **O(n + d log d)** where `n` is the size of the input and `d` is the number of distinct characters.

In cases where the number of distinct characters (`d`) is much smaller than the length of the input (`n`), the time complexity is dominated by **O(n)**. However, if `d` is large, the term **O(d log d)** becomes significant.

Program:

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

using namespace std;
using namespace std::chrono;

int flag = 0;

struct Node {
    char ch;
    int freq;
    Node* left;
    Node* right;

    Node(char c, int f) {
        ch = c;
        freq = f;
        left = right = nullptr;
    }
};

struct Compare {
    bool operator()(Node* l, Node* r) {
        return l->freq > r->freq;
    }
};

void printHuffmanCodes(Node* root, string str, char characters[], string huffmanCode[], int
&index) {
    if (!root) return;

    if (!root->left && !root->right) {
        characters[index] = root->ch;
        huffmanCode[index] = str;
        index++;
    }

    printHuffmanCodes(root->left, str + "0", characters, huffmanCode, index);
    printHuffmanCodes(root->right, str + "1", characters, huffmanCode, index);
}

void buildHuffmanTree(string text) {

    int freq[128] = {0};
```

```
for (char ch : text) {
    freq[(int)ch]++;
}

priority_queue<Node*, vector<Node*>, Compare> pq;

for (int i = 0; i < 128; i++) {
    if (freq[i] > 0) {
        pq.push(new Node((char)i, freq[i]));
    }
}


while (pq.size() != 1) {

    Node* left = pq.top(); pq.pop();
    Node* right = pq.top(); pq.pop();

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

    pq.push(node);
}

Node* root = pq.top();

char characters[128];
string huffmanCode[128];
int index = 0;

if (flag == 0)
{
printHuffmanCodes(root, "", characters, huffmanCode, index);

cout << "Huffman Codes are:\n";
for (int i = 0; i < index; i++) {
    cout << characters[i] << ": " << huffmanCode[i] << "\n";
}

//size of original string in bits
int originalSize = text.length() * 8;

string encodedString;
int encodedSize = 0;
for (char ch : text) {
    for (int i = 0; i < index; i++) {
        if (characters[i] == ch) {
            encodedString += huffmanCode[i];
```

```
            encodedSize += huffmanCode[i].length(); // Each character's code length in bits
            break;
        }
      }
    }

    cout << "\nEncoded string is:\n" << encodedString << "\n";

    cout << "\nOriginal string size: " << originalSize << " bits\n";
    cout << "Encoded string size: " << encodedSize << " bits\n\n";
    }
    flag++;
}

string generateRandomString(int size) {
    string charset =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    string result;
    for (int i = 0; i < size; i++) {
        result += charset[rand() % charset.size()];
    }
    return result;
}

int main() {
    srand(static_cast<unsigned int>(time(0)));

    // Original code for user input and Huffman encoding
    string text;
    cout << "Enter a string to encode using Huffman Coding: ";
    getline(cin, text);

    buildHuffmanTree(text);

    // Benchmark Huffman encoding for random strings of different sizes
    int sizes[] = {500, 1000, 1500, 2000};

    cout << "\n\nFor Huffman Coding on Random Strings:" << endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        string randomString = generateRandomString(size);

        auto start = high_resolution_clock::now();
        buildHuffmanTree(randomString);
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();
        cout << "\nThe elapsed time for a string of " << size << " characters is " << time_spent
<< " nanoseconds" << endl;
```

```
    }

    cout<<endl;
    return 0;
}
```

```
Enter a string to encode using Huffman Coding: Sriyanshu Azad
Huffman Codes are:
A: 000
d: 001
 : 0100
S: 0101
i: 0110
z: 0111
y: 1000
h: 1001
r: 1010
n: 1011
u: 1100
s: 1101
a: 111

Encoded string is:
01011010011010001111011111011001110001000000111111001

Original string size: 112 bits
Encoded string size: 52 bits



For Huffman Coding on Random Strings:

The elapsed time for a string of 500 characters is 92000 nanoseconds

The elapsed time for a string of 1000 characters is 97000 nanoseconds

The elapsed time for a string of 1500 characters is 102000 nanoseconds

The elapsed time for a string of 2000 characters is 111000 nanoseconds
```

Output:

Graph:

Learning Outcomes:

PROGRAM 4

Aim: To implement Minimum Spanning Tree and analyse its time complexity.

Theory:

A **Minimum Spanning Tree (MST)** is a subset of the edges of a connected, undirected graph that connects all the vertices without any cycles and with the minimum possible total edge weight. In other words, it's a tree that spans all the vertices of the graph and has the smallest possible sum of edge weights.

Key Properties of MST:

1. **Connected**: All vertices of the graph are connected by the tree.
2. **Acyclic**: There are no cycles in the MST.
3. **Minimum Weight**: The sum of the edge weights in the MST is minimized.
4. **Unique for Distinct Weights**: If all edge weights are distinct, the MST is unique.

Kruskal's Algorithm

Kruskal's Algorithm is a greedy algorithm that finds an MST for a connected, weighted graph. The algorithm follows the principle of adding the shortest edge to the tree without forming a cycle, ensuring that the growing tree remains acyclic at each step.

Steps of Kruskal's Algorithm:

1. **Sort All Edges**: First, sort all edges in the graph in increasing order of their weights.
2. **Initialize Disjoint Sets**: Each vertex is initially its own parent in a disjoint-set (union-find) structure, meaning there are as many components as vertices.
3. **Iterate Through Edges**: Iterate through the sorted edges, and for each edge:
   - Check if the current edge connects two different components (i.e., vertices with different parents in the disjoint-set).
   - If it does, add the edge to the MST, and union the sets of the two vertices (this merges their components).
4. **Continue Until the MST is Formed**: Repeat until the MST contains exactly $V-1$ edges (where V is the number of vertices).

Time Complexity of Kruskal's Algorithm

The time complexity of **Kruskal's Algorithm** is determined by two main operations:

1. **Sorting the edges**:
   - Sorting all the edges by weight takes O(E log E), where:
     - E is the number of edges.
2. **Union-Find operations**:

- The `find` and `union` operations in the disjoint-set data structure (for cycle detection and merging sets) take O(α(V)) time each, where:
  - V is the number of vertices.
  - α(V) is the inverse Ackermann function, which grows extremely slowly and is almost constant in practice.

Since sorting the edges dominates, the **overall time complexity** of Kruskal's algorithm is: **O(E log E)**

Program:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_map>
#include <chrono>

using namespace std;
using namespace std::chrono;

struct Edge {
    char u, v;
    int weight;
};

bool compare(Edge a, Edge b) {
    return a.weight < b.weight;
}

char findParent(char u, unordered_map<char, char>& parent) {
    if (parent[u] == u) return u;
    return parent[u] = findParent(parent[u], parent);
}

void unionSets(char u, char v, unordered_map<char, char>& parent, unordered_map<char,
int>& rank) {
    u = findParent(u, parent);
    v = findParent(v, parent);

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

void kruskal(vector<Edge>& edges) {
```

```
    sort(edges.begin(), edges.end(), compare);

    unordered_map<char, char> parent;
    unordered_map<char, int> rank;

    for (char c = 'A'; c < 'A' + edges.size(); c++) {
        parent[c] = c;
        rank[c] = 0;
    }

    vector<Edge> mst;
    int mstWeight = 0;

    for (Edge edge : edges) {

        char rootU = findParent(edge.u, parent);
        char rootV = findParent(edge.v, parent);

        if (rootU != rootV) {
            mst.push_back(edge);
            mstWeight += edge.weight;
            unionSets(rootU, rootV, parent, rank);
        }
    }

    cout << "Minimum Spanning Tree (MST) contains the following edges:\n";
    for (Edge edge : mst) {
        cout << edge.u << " -- " << edge.v << " == " << edge.weight << "\n";
    }
    cout << "Total weight of MST: " << mstWeight << "\n";
}

int main() {
    int E;
    cout << "Enter the number of edges: ";
    cin >> E;

    vector<Edge> edges(E);

    cout << "Enter the edges in the format (u v weight):\n";
    for (int i = 0; i < E; i++) {
        cin >> edges[i].u >> edges[i].v >> edges[i].weight;
    }

    auto start = high_resolution_clock::now();
    kruskal(edges);
    auto end = high_resolution_clock::now();

    auto time_spent = duration_cast<nanoseconds>(end - start).count();
```

```
    cout << "\nThe elapsed time for " << E << " edges is " << time_spent << " nanoseconds"
<< endl;

    cout<<endl;
    return 0;
}
```

```
Enter the number of edges: 5
Enter the edges in the format (u v weight):
A B 4
A C 1
B C 2
B D 3
C D 5
Minimum Spanning Tree (MST) contains the following edges:
A -- C == 1
B -- C == 2
B -- D == 3
Total weight of MST: 6

The elapsed time for 5 edges is 194000 nanoseconds
```

Output:

```
Enter the number of edges: 10
Enter the edges in the format (u v weight):
A B 3
A C 8
A D 5
A E 7
B C 1
B D 4
B E 2
C D 6
C E 9
D E 10
Minimum Spanning Tree (MST) contains the following edges:
B -- C == 1
B -- E == 2
A -- B == 3
B -- D == 4
Total weight of MST: 10

The elapsed time for 10 edges is 1211000 nanoseconds
```

```
Enter the number of edges: 15
Enter the edges in the format (u v weight):
A B 4
A C 6
A D 3
A E 9
A F 7
B C 2
B D 5
B E 8
B F 1
C D 10
C E 7
C F 3
D E 6
D F 8
E F 4
Minimum Spanning Tree (MST) contains the following edges:
B -- F == 1
B -- C == 2
A -- D == 3
A -- B == 4
E -- F == 4
Total weight of MST: 14

The elapsed time for 15 edges is 1979000 nanoseconds
```

Graph:

Learning Outcomes:

PROGRAM 5

Aim: To implement Matrix Multiplication and analyse its time complexity.

Theory:

Matrix Chain Multiplication is a problem where we are given a sequence of matrices and need to find the most efficient way to multiply them. The objective is to minimize the total number of scalar multiplications. Although matrix multiplication is associative (the order doesn't affect the result), the order of operations can significantly impact the computational cost.

Problem Definition:

Given matrices A1, A2, ..., An, where matrix Ai has dimensions `p(i-1) x p(i)`, we need to determine the minimum number of scalar multiplications required to compute the product `A1 x A2 x ... x An`.

Dynamic Programming Approach:

1. **Subproblem**: Let `dp[i][j]` represent the minimum number of scalar multiplications needed to compute the product of matrices from Ai to Aj.
2. **Recursive Formula**: To compute `dp[i][j]`, split the chain at every possible position `k`, and choose the split that minimizes the total cost:

```
dp[i][j] = min(dp[i][k] + dp[k+1][j] + (dimensions[i-1]
     * dimensions[k] * dimensions[j]))
```

3. **Initialization**: For multiplying a single matrix, no operations are needed, so:`css`

```
dp[i][i] = 0
```

4. **Optimal Solution**: The result is found in `dp[1][n-1]`, where `n` is the number of matrices.

Using dynamic programming, we can calculate the minimum number of scalar multiplications required by recursively solving subproblems and storing the results in a table.

Time Complexity:

The time complexity of Matrix Chain Multiplication is O(n^3), where `n` is the number of matrices. This is because:

1. There are O(n^2) subproblems (one for each pair of matrices).
2. For each subproblem, we check all possible splits, which takes O(n) time.

Program:

```
#include <iostream>
#include <climits>
#include<chrono>

using namespace std;
using namespace std::chrono;

int MatrixChainMultiplication(int dimensions[], int n) {

    int dp[n][n];

    // cost is zero when multiplying one matrix
    for (int i = 1; i < n; i++)
        dp[i][i] = 0;

    // L is the chain length
    for (int L = 2; L < n; L++) {
        for (int i = 1; i < n - L + 1; i++) {
            int j = i + L - 1;
            dp[i][j] = INT_MAX;
            for (int k = i; k < j; k++) {
                // q = cost/scalar multiplications
                int q = dp[i][k] + dp[k + 1][j] + dimensions[i - 1] * dimensions[k] * dimensions[j];
                if (q < dp[i][j])
                    dp[i][j] = q;
            }
        }
    }

    // The minimum cost is found at dp[1][n-1]
    return dp[1][n - 1];
}

int main() {
    int n;
    cout<<"Enter the number of matrices: ";
    cin>>n;
    cout<<"Enter the dimensions of the matrices: ";
    int dimensions[n];
    for (int i = 0; i < n; i++)
        cin>>dimensions[i];

    auto start = high_resolution_clock::now();
    cout << "Minimum number of multiplications is: "
         << MatrixChainMultiplication(dimensions, n) << endl;
    auto end = high_resolution_clock::now();

    auto time_spent = duration_cast<nanoseconds>(end - start).count();
```

```
    cout << "\nThe elapsed time for " << n << " matrices is " << time_spent << "
nanoseconds" << endl;

    cout<<endl;
    return 0;
}
```

Output:

```
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./test
Enter the number of matrices: 5
Enter the dimensions of the matrices: 3 2 4 2 5
Minimum number of multiplications is: 58

The elapsed time for 5 matrices is 104000 nanoseconds

sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./test
Enter the number of matrices: 10
Enter the dimensions of the matrices: 2 4 6 8 10 12 14 16 18 20
Minimum number of multiplications is: 2624

The elapsed time for 10 matrices is 115000 nanoseconds

sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./test
Enter the number of matrices: 15
Enter the dimensions of the matrices: 2 3 5 7 11 13 17 19 23 29 31 37 39 41 43
Minimum number of multiplications is: 17538

The elapsed time for 15 matrices is 1481000 nanoseconds

sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./test
Enter the number of matrices: 20
Enter the dimensions of the matrices: 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
Minimum number of multiplications is: 332250

The elapsed time for 20 matrices is 107000 nanoseconds
```

Graph:

Learning Outcomes:

PROGRAM 6

Aim: To implement Dijkstra algorithm and analyse its time complexity.

Theory:

Dijkstra's Algorithm is a popular shortest path algorithm that finds the minimum distance from a starting vertex (or "source") to all other vertices in a weighted, directed or undirected graph. It was developed by Dutch computer scientist Edsger W. Dijkstra in 1956 and is particularly efficient for graphs without negative edge weights.
Key Concepts

1. **Graph Representation**: Dijkstra's algorithm is typically used with graphs represented as adjacency lists, where each vertex has a list of connected vertices and the respective edge weights.

2. **Priority Queue**: A priority queue (or min-heap) is used to keep track of the current minimum distances to unexplored vertices. The vertex with the smallest known distance is processed first.

3. **Relaxation**: As each vertex is processed, its neighboring vertices are evaluated, and the algorithm updates (or "relaxes") the shortest known distance to each neighbor if a shorter path is found.

Time Complexity of Dijkstra's Algorithm

The time complexity of Dijkstra's algorithm depends on the data structure used for the priority queue and the graph's representation:

Using a Binary Heap:

- **Vertex Extraction**: Each vertex extraction from the priority queue takes O(log V), where V is the number of vertices.
- **Edge Relaxation**: Each edge is considered once and relaxed if necessary, with the relaxation taking O(log V) using the heap.
- **Total Complexity**: O((V + E) log V), where E is the number of edges.

Using a Fibonacci Heap (less common but optimal):

- **Vertex Extraction**: With a Fibonacci heap, the complexity for extracting the minimum element becomes O(log V).
- **Edge Relaxation**: Fibonacci heaps allow a decrease-key operation in O(1) time, making each edge relaxation O(1).
- **Total Complexity**: O(E + V log V), which is faster than the binary heap version for dense graphs.

Program:

```
#include <iostream>
#include <vector>
#include <queue>
#include <utility>
#include <climits>
#include <chrono>

using namespace std;
using namespace chrono;

typedef pair<int, int> pii;

void addEdge(vector<vector<pii>>& graph, int u, int v, int weight) {
    graph[u].push_back({weight, v});
    graph[v].push_back({weight, u});
}

vector<int> dijkstra(const vector<vector<pii>>& graph, int source) {
    int V = graph.size();
    vector<int> dist(V, INT_MAX);
    priority_queue<pii, vector<pii>, greater<pii>> pq;

    dist[source] = 0;
    pq.push({0, source});

    while (!pq.empty()) {
        int u = pq.top().second;
        int uDist = pq.top().first;
        pq.pop();

        if (uDist > dist[u]) continue;

        for (const auto& edge : graph[u]) {
            int weight = edge.first;
            int v = edge.second;

            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}

int main() {
    int V, E;
    cout << "Enter the number of vertices: ";
```

```
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;

    vector<vector<pii>> graph(V);

    cout << "Enter the edges in the format (u v weight):" << endl;
    for (int i = 0; i < E; ++i) {
        int u, v, weight;
        cin >> u >> v >> weight;
        addEdge(graph, u, v, weight);
    }

    int source;
    cout << "Enter the source vertex: ";
    cin >> source;

    auto start = high_resolution_clock::now();
    vector<int> distances = dijkstra(graph, source);
    auto end = high_resolution_clock::now();

    auto duration = duration_cast<microseconds>(end - start).count();
    cout << "Time taken to execute Dijkstra's algorithm: " << duration << " microseconds" <<
endl;

    cout << "Vertex\tDistance from Source " << source << endl;
    for (int i = 0; i < V; ++i) {
        cout << i << "\t\t" << distances[i] << endl;
    }

    return 0;
}
```

```
Enter the number of vertices: 4
Enter the number of edges: 5
Enter the edges in the format (u v weight):
0 1 4
0 2 1
1 2 2
1 3 5
2 3 1
Enter the source vertex: 0
Time taken to execute Dijkstra's algorithm: 153 microseconds
Vertex  Distance from Source 0
0               0
1               3
2               1
3               2
```

Output:

```
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAADijkstras
Enter the number of vertices: 6
Enter the number of edges: 10
Enter the edges in the format (u v weight):
0 1 3
0 2 5
0 3 9
1 2 1
1 4 7
2 3 3
2 4 4
3 4 2
3 5 6
4 5 1
Enter the source vertex: 0
Time taken to execute Dijkstra's algorithm: 31 microseconds
Vertex  Distance from Source 0
0               0
1               3
2               4
3               7
4               8
5               9
```

```
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAADijkstras
Enter the number of vertices: 7
Enter the number of edges: 15
Enter the edges in the format (u v weight):
0 1 2
0 2 4
0 3 1
1 2 2
1 4 7
2 3 3
2 5 8
3 4 5
3 6 6
4 5 2
4 6 4
5 6 3
1 6 5
2 6 4
3 5 6
Enter the source vertex: 0
Time taken to execute Dijkstra's algorithm: 40 microseconds
Vertex  Distance from Source 0
0               0
1               2
2               4
3               1
4               6
5               7
6               7
```

```
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAADijkstras
Enter the number of vertices: 8
Enter the number of edges: 20
Enter the edges in the format (u v weight):
0 1 2
0 2 6
0 3 5
1 2 1
1 4 3
1 5 9
2 3 2
2 6 4
3 6 3
3 7 8
4 5 4
4 6 7
4 7 5
5 6 2
5 7 3
2 4 8
3 5 1
0 7 10
1 6 6
6 7 4
Enter the source vertex: 0
Time taken to execute Dijkstra's algorithm: 34 microseconds
Vertex  Distance from Source 0
0               0
1               2
2               3
3               5
4               5
5               6
6               7
7               9
```

Graph:

Learning Outcome:

PROGRAM 7

Aim: To implement Bellman Ford algorithm and analyse its time complexity

Theory:

The Bellman-Ford algorithm is a single-source shortest path algorithm for graphs that may contain edges with negative weights. It computes the shortest paths from a source vertex to all other vertices in a weighted graph. It is particularly useful for graphs with negative weight edges, unlike Dijkstra's algorithm, which cannot handle such cases correctly.

Key Concepts

1. **Relaxation**: The process of updating the distance to a vertex if a shorter path is found. This is a core concept in shortest path algorithms.
2. **Negative-Weight Cycles**: A cycle in a graph where the total weight is negative. If a negative-weight cycle is reachable from the source, the shortest path cannot be determined since the path can be made indefinitely shorter by traversing the cycle repeatedly.

Algorithm Steps

1. **Initialization**: Set the distance to the source vertex to 0 and all other vertices to infinity.
2. **Relaxation**: For each vertex, repeat the process of relaxing all edges V-1 times (where V is the number of vertices). This ensures that the shortest paths are found as every possible edge is checked.
3. **Cycle Detection**: After the V-1 iterations, perform one more iteration over all edges. If any distance can still be shortened, a negative-weight cycle exists in the graph.

Time Complexity: O(V * E)
The algorithm iterates over all edges for V - 1 times. In the worst case, this leads to V iterations through E edges.

Space Complexity: O(V)
The algorithm requires an array to store distances for all vertices, leading to linear space complexity relative to the number of vertices.

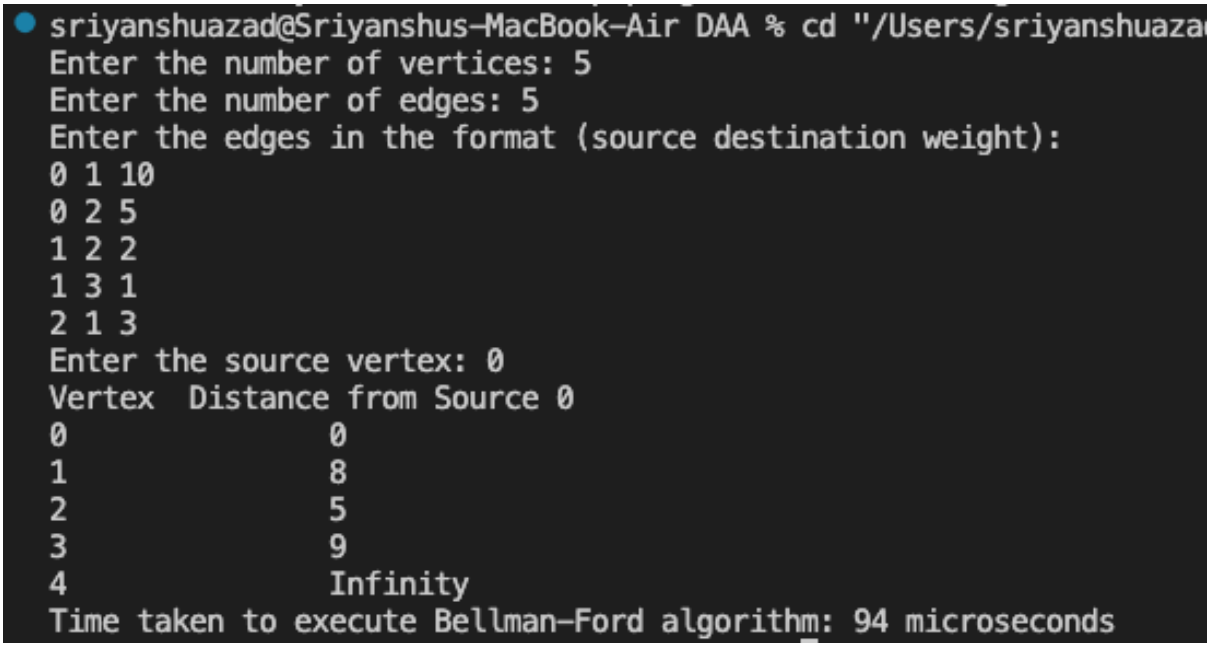
Program:

```
#include <iostream>
#include <vector>
#include <limits.h>
#include <chrono>

using namespace std;
using namespace chrono;

struct Edge {
    int src, dest, weight;
};

class BellmanFord {
    int vertices;
    vector<Edge> edges;

public:
    BellmanFord(int vertices) {
        this->vertices = vertices;
    }

    void addEdge(int src, int dest, int weight) {
        edges.push_back({src, dest, weight});
    }

    void bellmanFord(int source) {
        vector<int> distance(vertices, INT_MAX);
        distance[source] = 0;

        // Step 1: Relax all edges V-1 times
        for (int i = 1; i < vertices; ++i) {
            for (const auto& edge : edges) {
                int u = edge.src;
                int v = edge.dest;
                int weight = edge.weight;
                if (distance[u] != INT_MAX && distance[u] + weight < distance[v]) {
                    distance[v] = distance[u] + weight;
                }
            }
        }

        // Step 2: Check for negative-weight cycles
        for (const auto& edge : edges) {
            int u = edge.src;
            int v = edge.dest;
            int weight = edge.weight;
            if (distance[u] != INT_MAX && distance[u] + weight < distance[v]) {
                cout << "Graph contains a negative-weight cycle" << endl;
```

```
            return;
        }
    }

    // Print the distances from source
    printDistances(distance, source);
}

private:
void printDistances(const vector<int>& distance, int source) {
    cout << "Vertex\tDistance from Source " << source << endl;
    for (int i = 0; i < vertices; ++i) {
        if (distance[i] == INT_MAX)
            cout << i << "\t\tInfinity" << endl;
        else
            cout << i << "\t\t" << distance[i] << endl;
    }
}
};

int main() {
    int vertices, edges;
    cout << "Enter the number of vertices: ";
    cin >> vertices;

    BellmanFord graph(vertices);

    cout << "Enter the number of edges: ";
    cin >> edges;

    cout << "Enter the edges in the format (source destination weight):" << endl;
    for (int i = 0; i < edges; ++i) {
        int src, dest, weight;
        cin >> src >> dest >> weight;
        graph.addEdge(src, dest, weight);
    }

    int source;
    cout << "Enter the source vertex: ";
    cin >> source;

    // Start measuring time
    auto start = high_resolution_clock::now();
    graph.bellmanFord(source);
    auto end = high_resolution_clock::now();

    // Calculate and print elapsed time
    auto duration = duration_cast<microseconds>(end - start).count();
    cout << "Time taken to execute Bellman-Ford algorithm: " << duration << "
microseconds" << endl;
```

```
    return 0;
}
```

Output:



```
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAABellman
Enter the number of vertices: 5
Enter the number of edges: 10
Enter the edges in the format (source destination weight):
0 1 10
0 2 5
1 2 2
1 3 1
2 1 3
2 3 9
2 4 2
3 4 4
3 0 -2
4 3 6
Enter the source vertex: 0
Vertex  Distance from Source 0
0               0
1               8
2               5
3               9
4               7
Time taken to execute Bellman-Ford algorithm: 43 microseconds
```

```
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAABellman
Enter the number of vertices: 6
Enter the number of edges: 15
Enter the edges in the format (source destination weight):
0 1 10
0 2 5
1 2 2
1 3 1
2 1 3
2 3 9
2 4 2
3 4 4
3 0 -2
4 3 6
4 5 1
5 0 3
5 1 7
5 2 6
5 3 2
Enter the source vertex: 0
Vertex  Distance from Source 0
0               0
1               8
2               5
3               9
4               7
5               8
Time taken to execute Bellman-Ford algorithm: 56 microseconds
```

Graph:

Learning Outcome:

PROGRAM 8

Aim: To implement n-Queen problem using backtracking.

Theory:

The N-Queens problem is a classic problem in combinatorial optimization and computer science. The objective is to place **N** queens on an **N x N** chessboard in such a way that no two queens threaten each other. This means that no two queens can be in the same row, the same column, or on the same diagonal.

Solution Approach: Backtracking

The backtracking algorithm is a systematic way to explore all potential placements of the queens and find all valid solutions. The approach consists of:

1. **Recursive Exploration**:
    - Start from the first row and attempt to place a queen in each column of that row.
    - For each placement, recursively attempt to place queens in the subsequent rows.
2. **Safety Checks**:
    - Before placing a queen in a given position, ensure that the position is safe. This involves checking:
        - The current column for previous rows to ensure no other queens are placed there.
        - The diagonals (both left and right) to ensure no queens are attacking diagonally.
    - If a position is found to be unsafe, backtrack by removing the last placed queen and trying the next possible position.
3. **Base Case**:
    - If all queens are successfully placed (i.e., when the row index equals **N**), a solution has been found, and the current configuration of the board is printed.
4. **Backtracking**:
    - If placing a queen in a particular column leads to a dead end (no further queens can be placed), the algorithm backtracks by removing the queen and trying the next column in the previous row.

Complexity

- **Time Complexity**: The time complexity of the backtracking approach for the N-Queens problem is **O(N!)** in the worst case, as each queen can potentially be placed in each row and column.
- **Space Complexity**: The space complexity is **O(N)** due to the recursion stack and the storage for the board representation.

Program:

```
#include <iostream>
#include <vector>
#include <sstream>

using namespace std;

class NQueens {
private:
    int n;
    vector<vector<int>> board;

public:
    NQueens(int n) : n(n), board(n, vector<int>(n, 0)) {}

    void printBoard() {
        cout << "Board Representation:" << endl;
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (board[i][j]) {
                    cout << " Q "; // Mark the queen's position
                } else {
                    cout << " . ";
                }
            }
            cout << endl;
        }

        cout << "Column Indices: ";
        vector<int> solution; // To store the column positions of queens

        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (board[i][j]) {
                    solution.push_back(j); // Store the column index of the queen
                }
            }
        }

        for (size_t i = 0; i < solution.size(); ++i) {
            cout << solution[i];
            if (i < solution.size() - 1) {
                cout << ",";
            }
        }
        cout << endl << endl;
    }

    // Function to check if placing a queen is safe
```

```
    bool isSafe(int row, int col) {
        // Check the column
        for (int i = 0; i < row; i++) {
            if (board[i][col]) return false;
        }

        // Check the upper left diagonal
        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j]) return false;
        }

        // Check the upper right diagonal
        for (int i = row, j = col; i >= 0 && j < n; i--, j++) {
            if (board[i][j]) return false;
        }

        return true;
    }

    // Backtracking function to solve the n-Queens problem
    bool solveNQueensUtil(int row) {
        if (row == n) {
            printBoard();
            return true;
        }

        bool foundSolution = false; // To track if we found a solution

        for (int col = 0; col < n; col++) {
            if (isSafe(row, col)) {
                board[row][col] = 1;
                foundSolution = solveNQueensUtil(row + 1) || foundSolution;
                board[row][col] = 0;
            }
        }

        return foundSolution;
    }

    void solveNQueens() {
        if (!solveNQueensUtil(0)) {
            cout << "No solution exists for n = " << n << endl;
        }
    }
};

int main() {
    int n;
    cout << "Enter the number of queens (n): ";
    cin >> n;
```

```
    NQueens nQueens(n);
    nQueens.solveNQueens();

    return 0;
}
```

```
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAANQueen
Enter the number of queens (n): 4
Board Representation:
 .  Q  .  .
 .  .  .  Q
 Q  .  .  .
 .  .  Q  .
Column Indices: 1,3,0,2

Board Representation:
 .  .  Q  .
 Q  .  .  .
 .  .  .  Q
 .  Q  .  .
Column Indices: 2,0,3,1
```

Output:

Learning Outcome:

PROGRAM 9

Aim: To implement Longest Common Subsequence problem and analyse its time complexity.

Theory:

The Longest Common Subsequence (LCS) problem is a classical problem in computer science and bioinformatics. It focuses on finding the longest subsequence present in two sequences. Unlike substrings, the subsequence is not required to occupy consecutive positions within the original sequences; thus, characters in the subsequence may be non-contiguous.

Definition

Given two sequences X and Y:

- X = X1, X2, ..., Xm
- Y = Y1, Y2, ..., Yn

A subsequence is a sequence derived from another sequence by deleting some or none of the elements without changing the order of the remaining elements. The LCS is the longest subsequence that appears in both sequences in the same order.

Applications

The LCS problem has several applications, including:

- Version Control Systems: Identifying changes between file versions.
- Bioinformatics: Comparing DNA, RNA, or protein sequences.
- Text Comparison: Finding similarities between text documents or programming code.

Time and Space Complexity

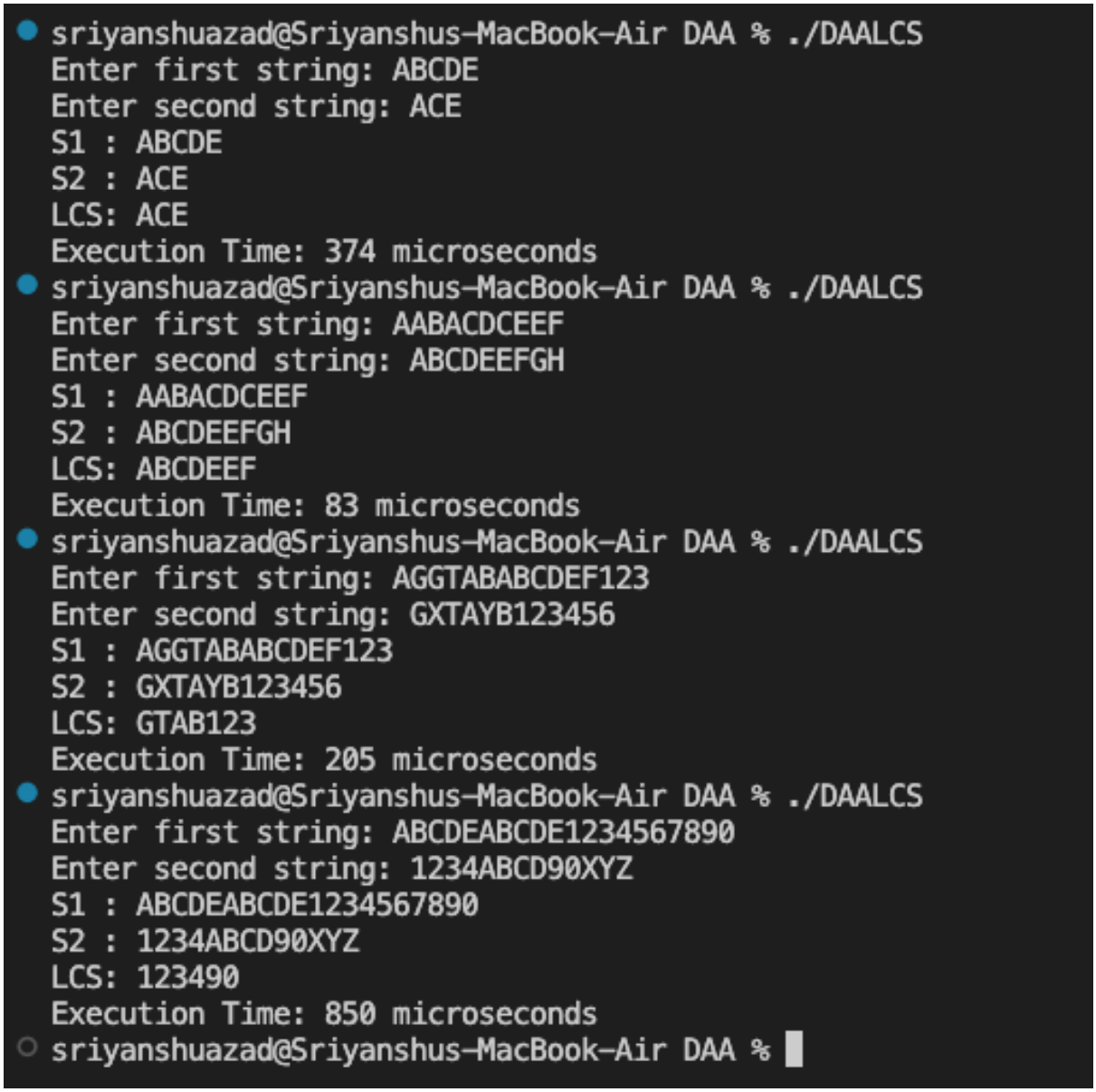
- **Time Complexity**: The time complexity of the LCS algorithm is **O(m × n)**, where m and n are the lengths of the two sequences.
- **Space Complexity**: The space complexity is also **O(m × n)** due to the storage of the DP table. However, it can be optimized to O(min(m, n)) by using only two rows of the DP table, as shown in the previous code examples.

Program:

```
#include <iostream>
#include <vector>
#include <chrono>
#include <string>

using namespace std;
using namespace chrono;

void lcsAlgo(const string &S1, const string &S2, int m, int n) {
    vector<vector<int>> LCS_table(m + 1, vector<int>(n + 1));

    // Building the matrix in bottom-up way
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                LCS_table[i][j] = 0;
            else if (S1[i - 1] == S2[j - 1])
                LCS_table[i][j] = LCS_table[i - 1][j - 1] + 1;
            else
                LCS_table[i][j] = max(LCS_table[i - 1][j], LCS_table[i][j - 1]);
        }
    }

    int index = LCS_table[m][n];
    string lcsString(index, '\0'); // Preallocate string with size index

    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (S1[i - 1] == S2[j - 1]) {
            lcsString[index - 1] = S1[i - 1];
            i--;
            j--;
            index--;
        } else if (LCS_table[i - 1][j] > LCS_table[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }

    // Printing the subsequences
    cout << "S1 : " << S1 << "\nS2 : " << S2 << "\nLCS: " << lcsString << "\n";
}

int main() {
    string S1, S2;

    // Taking input from user
```

```
    cout << "Enter first string: ";
    cin >> S1;
    cout << "Enter second string: ";
    cin >> S2;

    int m = S1.length();
    int n = S2.length();

    auto start = high_resolution_clock::now();
    lcsAlgo(S1, S2, m, n);
    auto end = high_resolution_clock::now();

    auto duration = duration_cast<microseconds>(end - start).count();
    cout << "Execution Time: " << duration << " microseconds" << endl;

    return 0;
}
```

Output:


sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAALCS
Enter first string: ABCDE
Enter second string: ACE
S1 : ABCDE
S2 : ACE
LCS: ACE
Execution Time: 374 microseconds
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAALCS
Enter first string: AABACDCEEF
Enter second string: ABCDEEFGH
S1 : AABACDCEEF
S2 : ABCDEEFGH
LCS: ABCDEEF
Execution Time: 83 microseconds
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAALCS
Enter first string: AGGTABABCDEF123
Enter second string: GXTAYB123456
S1 : AGGTABABCDEF123
S2 : GXTAYB123456
LCS: GTAB123
Execution Time: 205 microseconds
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAALCS
Enter first string: ABCDEABCDE1234567890
Enter second string: 1234ABCD90XYZ
S1 : ABCDEABCDE1234567890
S2 : 1234ABCD90XYZ
LCS: 123490
Execution Time: 850 microseconds
sriyanshuazad@Sriyanshus-MacBook-Air DAA %

Graph:

Learning Outcome:

PROGRAM 10

Aim: To implement Naive String-Matching algorithm, Rabin Karp algorithm and knuth Morris Pratt algorithm and analyse its time complexity

Theory:

String matching is a fundamental problem in computer science that involves finding occurrences of a substring (pattern) within a larger string (text). Various algorithms have been developed to solve this problem efficiently. Here are three well-known string matching algorithms: the Naive String Matching Algorithm, Rabin-Karp Algorithm, and Knuth-Morris-Pratt (KMP) Algorithm.

1. Naive String Matching Algorithm

The naive approach checks every possible position in the text for a match with the pattern. This algorithm works by sliding the pattern over the text and checking for matches at each position.

- **Algorithm**:
  - For each position `i` in the text, check if the substring starting from `i` matches the pattern.
  - If all characters match, report the position `i`.
- **Time Complexity**:
  - **Best Case**: O(n) when the pattern does not exist in the text and characters do not match at the start.
  - **Worst Case**: O(m * n) where `m` is the length of the pattern and `n` is the length of the text, particularly when the text contains repeated characters.

2. Rabin-Karp Algorithm

The Rabin-Karp algorithm uses hashing to find any one of a set of pattern strings in a text. It calculates a hash value for the pattern and for each substring of the text of the same length.

- **Algorithm**:
  - Compute the hash value of the pattern.
  - Compute the hash value for the first substring of the text with the same length as the pattern.
  - Slide the window one character at a time, updating the hash value efficiently (using subtraction and addition) to compare with the pattern's hash value.
  - If the hash values match, a direct comparison is performed to confirm the match (to avoid hash collisions).
- **Time Complexity**:
  - **Best Case**: O(n) when the pattern is found early in the text.
  - **Worst Case**: O(m * n) if there are many hash collisions, but on average, it performs well with O(n + m).

3. Knuth-Morris-Pratt (KMP) Algorithm

The KMP algorithm improves upon the naive approach by avoiding unnecessary comparisons using preprocessed information about the pattern itself.

- **Algorithm**:
  - Preprocess the pattern to create a longest prefix-suffix (LPS) array, which contains the lengths of the longest proper prefix that is also a suffix for each substring of the pattern.
  - Use the LPS array to skip comparisons in the text when a mismatch occurs.
  - When a mismatch occurs after some matches, use the LPS array to find the next positions in the pattern to compare instead of starting from scratch.
- **Time Complexity**:
  - **Best Case**: O(n) when the pattern is found quickly.
  - **Worst Case**: O(n + m), where `m` is the length of the pattern and `n` is the length of the text.

Program:

```
#include <iostream>
#include <vector>
#include <string>
#include <chrono>

using namespace std;
using namespace chrono;

// Naive String Matching Algorithm
void naiveStringMatch(const string& text, const string& pattern) {
    int m = pattern.length();
    int n = text.length();
    for (int i = 0; i <= n - m; i++) {
        int j;
        for (j = 0; j < m; j++) {
            if (text[i + j] != pattern[j]) {
                break;
            }
        }
        if (j == m) {
            cout << "Naive: Pattern found at index " << i << endl;
        }
    }
}

// Rabin-Karp Algorithm
void rabinKarp(const string& text, const string& pattern) {
```

```
    int m = pattern.length();
    int n = text.length();
    const int d = 256; // Number of characters in the input alphabet
    const int q = 101; // A prime number for hashing
    int h = 1;
    int p = 0; // Hash value for pattern
    int t = 0; // Hash value for text

    // Calculate the value of h = d^(m-1) % q
    for (int i = 0; i < m - 1; i++) {
        h = (h * d) % q;
    }

    // Calculate initial hash values for pattern and first window of text
    for (int i = 0; i < m; i++) {
        p = (d * p + pattern[i]) % q;
        t = (d * t + text[i]) % q;
    }

    for (int i = 0; i <= n - m; i++) {
        if (p == t) {
            // Check for characters one by one
            int j;
            for (j = 0; j < m; j++) {
                if (text[i + j] != pattern[j]) {
                    break;
                }
            }
            if (j == m) {
                cout << "Rabin-Karp: Pattern found at index " << i << endl;
            }
        }
        // Calculate hash value for next window
        if (i < n - m) {
            t = (d * (t - text[i] * h) + text[i + m]) % q;
            if (t < 0) {
                t += q;
            }
        }
    }
}

// Knuth-Morris-Pratt (KMP) Algorithm
void computeLPSArray(const string& pattern, vector<int>& lps) {
    int len = 0; // Length of previous longest prefix suffix
    lps[0] = 0; // LPS[0] is always 0
    int i = 1;
    int m = pattern.length();

    while (i < m) {
```

```
        if (pattern[i] == pattern[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

void kmp(const string& text, const string& pattern) {
    int n = text.length();
    int m = pattern.length();
    vector<int> lps(m);

    computeLPSArray(pattern, lps);

    int i = 0; // Index for text
    int j = 0; // Index for pattern

    while (i < n) {
        if (pattern[j] == text[i]) {
            i++;
            j++;
        }
        if (j == m) {
            cout << "KMP: Pattern found at index " << i - j << endl;
            j = lps[j - 1];
        } else if (i < n && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
}

int main() {
    string text, pattern;
    cout << "Enter text: ";
    cin >> text;
    cout << "Enter pattern: ";
    cin >> pattern;
```

```
    // Naive String Matching
    auto start = high_resolution_clock::now();
    naiveStringMatch(text, pattern);
    auto end = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(end - start).count();
    cout << "Naive execution time: " << duration << " microseconds" << endl << endl;

    // Rabin-Karp Algorithm
    start = high_resolution_clock::now();
    rabinKarp(text, pattern);
    end = high_resolution_clock::now();
    duration = duration_cast<microseconds>(end - start).count();
    cout << "Rabin-Karp execution time: " << duration << " microseconds" << endl << endl;

    // KMP Algorithm
    start = high_resolution_clock::now();
    kmp(text, pattern);
    end = high_resolution_clock::now();
    duration = duration_cast<microseconds>(end - start).count();
    cout << "KMP execution time: " << duration << " microseconds" << endl << endl;

    return 0;
}
```

Output:

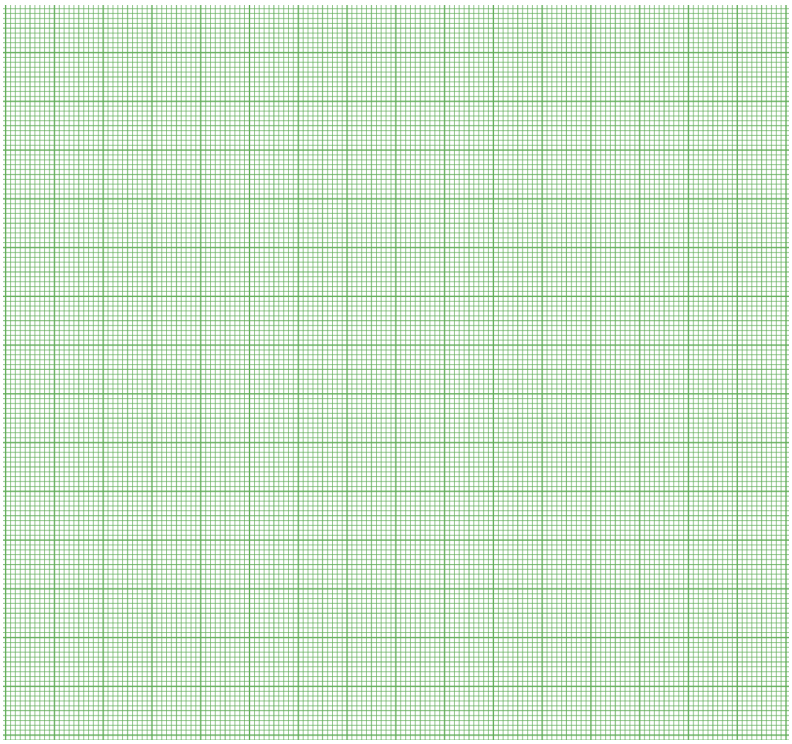
```
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAA10
Enter text: ABCDABEFGH
Enter pattern: AB
Naive: Pattern found at index 0
Naive: Pattern found at index 4
Naive execution time: 1389 microseconds

Rabin-Karp: Pattern found at index 0
Rabin-Karp: Pattern found at index 4
Rabin-Karp execution time: 4 microseconds

KMP: Pattern found at index 0
KMP: Pattern found at index 4
KMP execution time: 339 microseconds
```

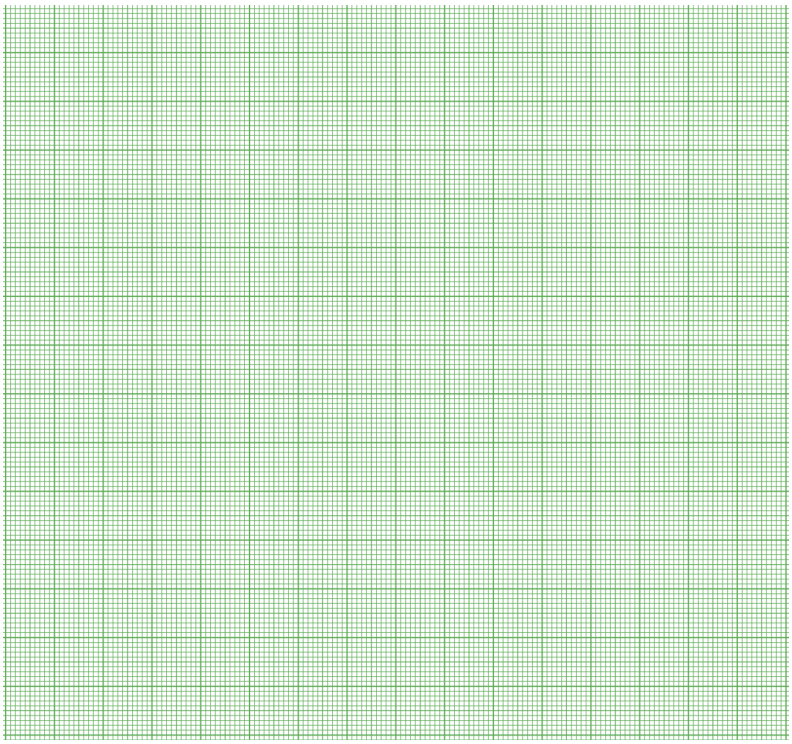
```
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAA10
Enter text: HELLOTHEREWORLD
Enter pattern: THERE
Naive: Pattern found at index 5
Naive execution time: 1105 microseconds

Rabin-Karp: Pattern found at index 5
Rabin-Karp execution time: 3 microseconds

KMP: Pattern found at index 5
KMP execution time: 42 microseconds
```

```
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAA10
Enter text: ABCDEFGHIJKLABCDEFGHIJKL
Enter pattern: DEFGHI
Naive: Pattern found at index 3
Naive: Pattern found at index 15
Naive execution time: 1878 microseconds

Rabin-Karp: Pattern found at index 3
Rabin-Karp: Pattern found at index 15
Rabin-Karp execution time: 5 microseconds

KMP: Pattern found at index 3
KMP: Pattern found at index 15
KMP execution time: 28 microseconds
```

```
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAA10
Enter text: THISISATESTSTRINGFORTESTING
Enter pattern: TESTING
Naive: Pattern found at index 20
Naive execution time: 1822 microseconds

Rabin-Karp: Pattern found at index 20
Rabin-Karp execution time: 3 microseconds

KMP: Pattern found at index 20
KMP execution time: 42 microseconds
```

Graph:





Learning Outcome:

PROGRAM 11

Aim: To implement Sorting Network.

Theory:

A sorting network is a mathematical model for sorting a sequence of elements using a fixed sequence of comparisons. Each comparison is made between two elements, and based on their order, they may be swapped. Sorting networks are particularly useful in parallel processing and hardware implementations of sorting algorithms.

Key Characteristics:

- **Fixed Sequence**: The sequence of comparisons in a sorting network is predetermined and does not depend on the input data. This makes it suitable for hardware implementations where the circuit needs to be fixed at design time.
- **Comparators**: The fundamental unit of a sorting network is a comparator, which compares two elements and swaps them if they are out of order. In hardware implementations, this corresponds to a physical circuit that can compare and swap values.
- **Layered Structure**: Sorting networks can be visualized as a series of layers, with each layer consisting of several comparators operating simultaneously. This parallel nature allows for efficient sorting.

Example of a Sorting Network

One of the simplest sorting networks is the one for sorting four elements, which can be represented as follows:

1. Compare and swap the first two elements.
2. Compare and swap the last two elements.
3. Compare and swap the first element with the third.
4. Compare and swap the second element with the fourth.
5. Finally, compare and swap the second and third elements.

This method ensures that the four elements are sorted after a fixed number of operations.

Complexity Analysis

1. **Time Complexity**:
   - The time complexity of a sorting network can be considered O(1) for a fixed number of elements, as the number of comparisons and swaps does not change with the size of the input. However, if we consider variable sizes, the time complexity can vary depending on the design of the sorting network. For larger inputs, the complexity may grow logarithmically with the number of elements.
2. **Space Complexity**:
   - The space complexity is O(1) since sorting networks operate in place and do not require additional storage that scales with input size. They use a constant amount of space regardless of how many elements are sorted.

Applications

- **Parallel Computing**: Sorting networks are particularly useful in parallel computing environments where operations can be executed simultaneously, allowing for efficient sorting of large datasets.
- **Hardware Implementation**: They are used in the design of hardware circuits for sorting algorithms, especially where low latency and predictable performance are required.

Program:

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

void compareAndSwap(int &a, int &b) {
    if (a > b) {
        swap(a, b);
    }
}

// Function to implement the sorting network for 4 elements
void sortingNetwork4(int &a, int &b, int &c, int &d) {
    compareAndSwap(a, b);
    compareAndSwap(c, d);

    compareAndSwap(a, c);
    compareAndSwap(b, d);

    compareAndSwap(b, c);
}

void printArray(const vector<int> &arr) {
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
}

int main() {
    vector<int> arr(4);

    cout << "Enter 4 integers to sort: ";
    for (int i = 0; i < 4; i++) {
        cin >> arr[i];
    }

    sortingNetwork4(arr[0], arr[1], arr[2], arr[3]);
```

```
    cout << "Sorted array: ";
    printArray(arr);

    return 0;
}
```

```
sriyanshuazad@Sriyanshus-MacBook-Air DAA % g++ DAASortingNet.cpp -o DAASortingNet
sriyanshuazad@Sriyanshus-MacBook-Air DAA % ./DAASortingNet
Enter 4 integers to sort: 9 5 1 6
Sorted array: 1 5 6 9
sriyanshuazad@Sriyanshus-MacBook-Air DAA %
```

Output:

Learning Outcome: