Department of Computer Science & Engineering

Practical file submitted in partial fulfillment for the evaluation of

DESIGN AND ANALYSIS OF ALGORITHM (CIC-359)

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS

Grade A++ Accredited Institution by NAAC

NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE
An ISO 9001:2015 Certified Institution

SCHOOL OF ENGINEERING & TECHNOLOGY

INDEX

S.No	EXP.	Date	Marks			Remark	Updated Marks	Faculty Signature
			Laboratory Assessment (15 Marks)	Class Participation (5 Marks)	Viva (5 Marks)			
1	To implement following algorithm using array as a data structure and analyse its time complexity. a) Merge sort b) Quick sort c) Bubble sort d) Selection sort e) Heap sort							
2	To implement Linear search and Binary search and analyse its time complexity.							

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:

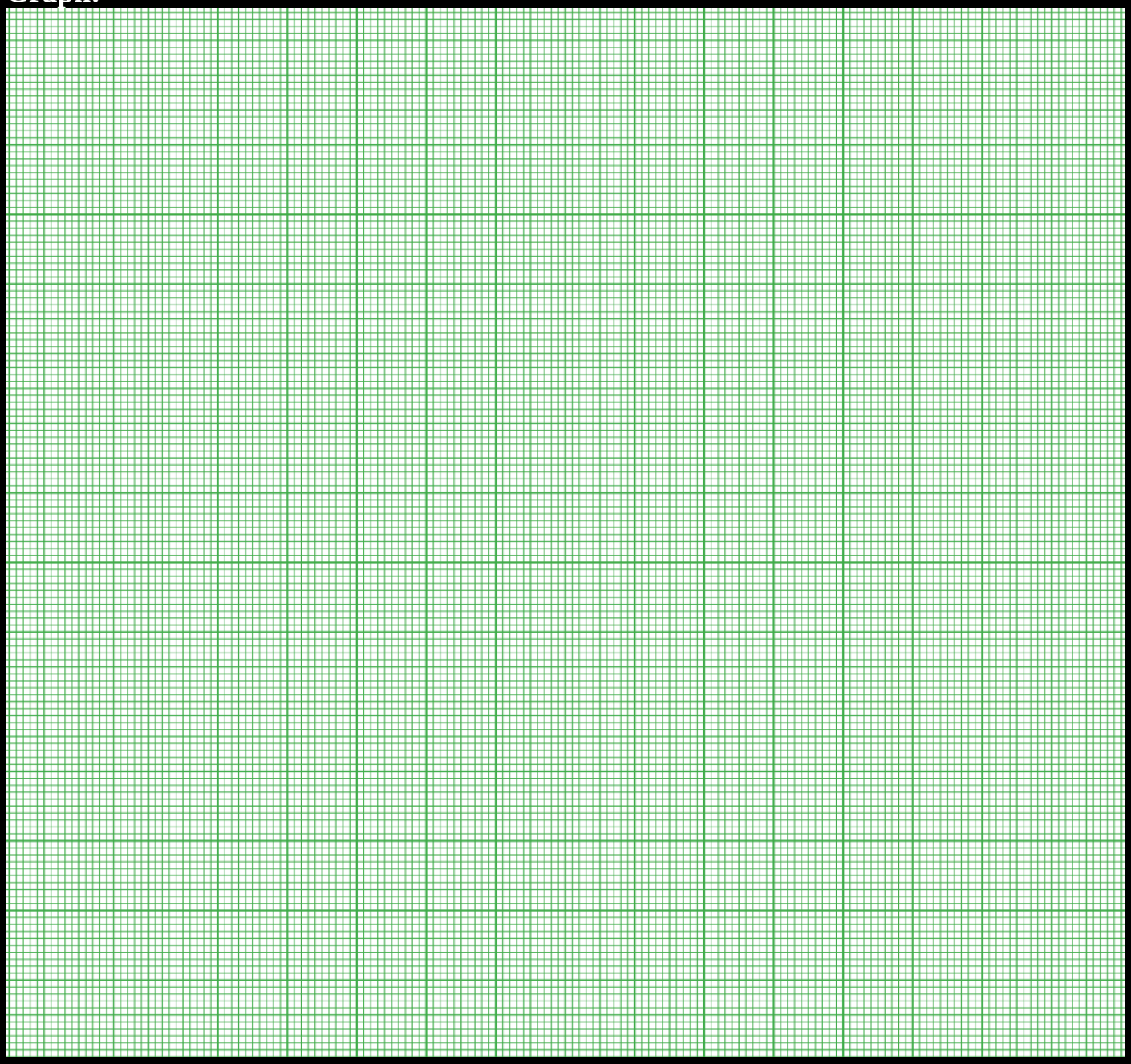
```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;
void insertionSort(int arr[], int n) {
for (int i = 1; i < n; ++i) {
int key = arr[i];
int j = i - 1;
//move one position right if greater than key
while (j >= 0 && arr[j] > key) {
arr[j + 1] = 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];
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;
}
```

Output:

```
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
```

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:

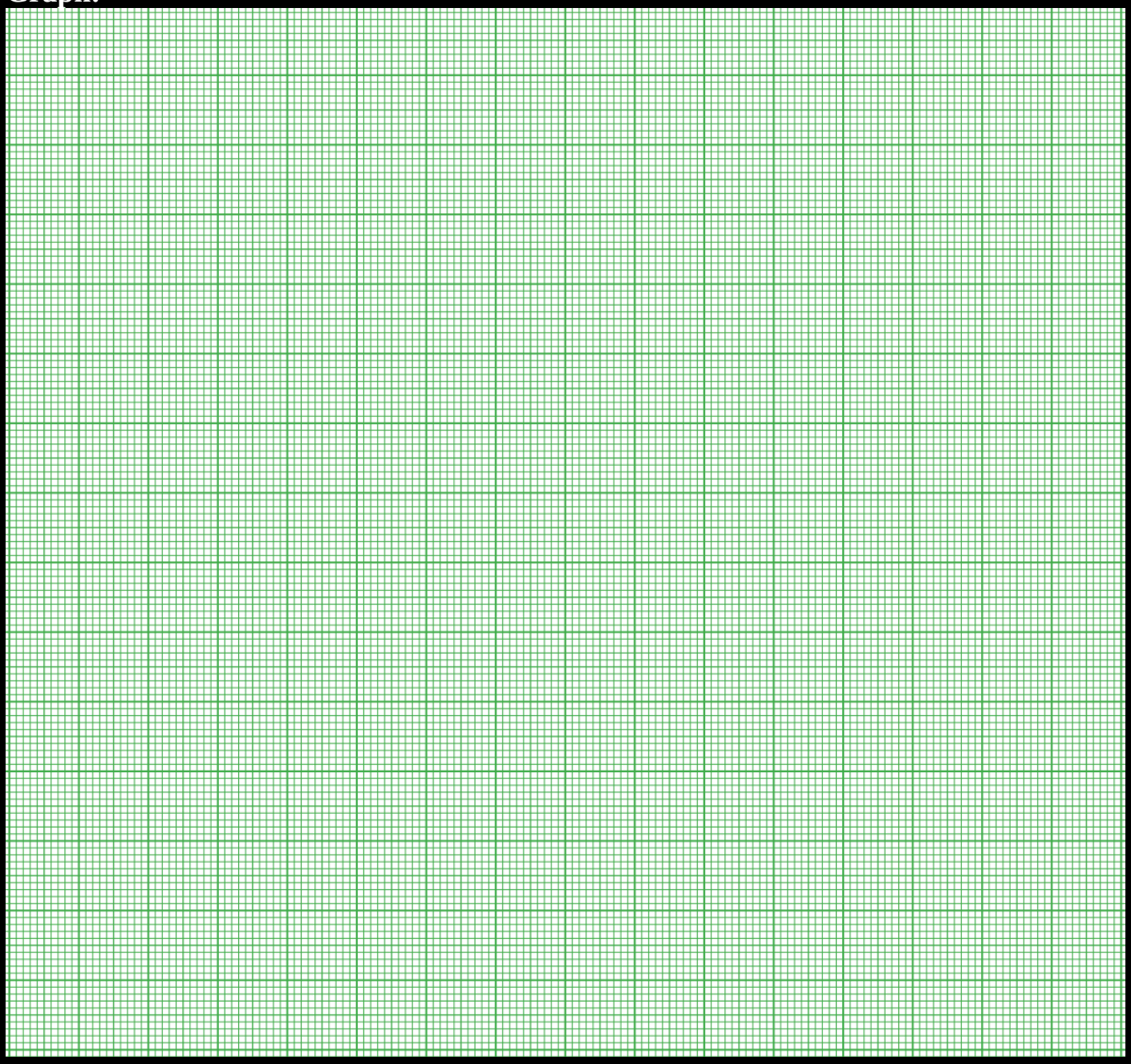
```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;
void selectionSort(int arr[], int n)
{
for (int i = 0; i < n - 1; ++i) {

// 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];
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.

D

Program:

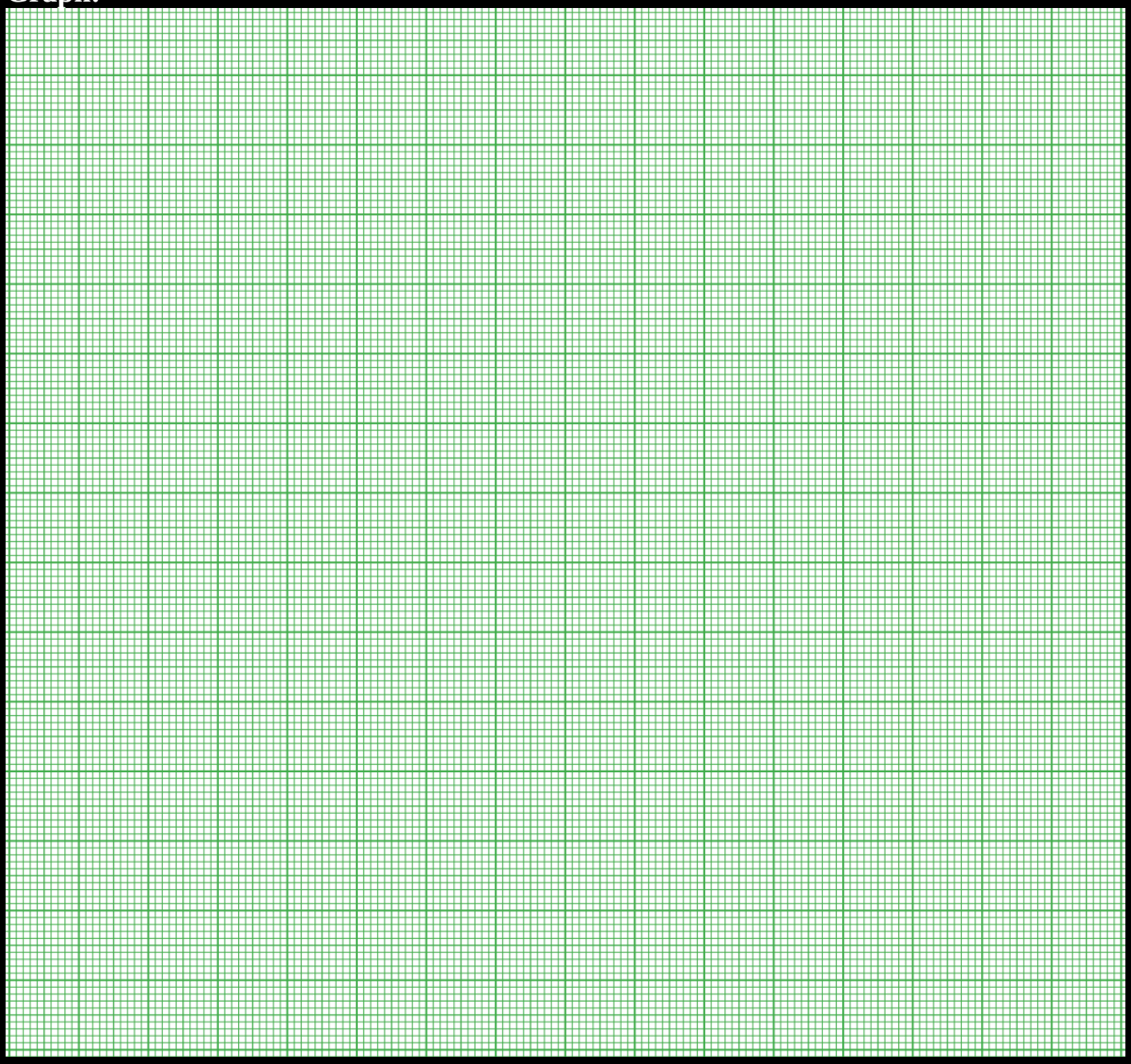
```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;
void bubbleSort(int arr[], int n) {
int flag=0;
for (int i = 0; i < n - 1; ++i) {
flag=0;
for (int j = 0; j < n - i - 1; ++j) {
if (arr[j] > arr[j + 1]) {
// Swap if the element found is greater 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;
}
```

Output:

```
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
```

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:

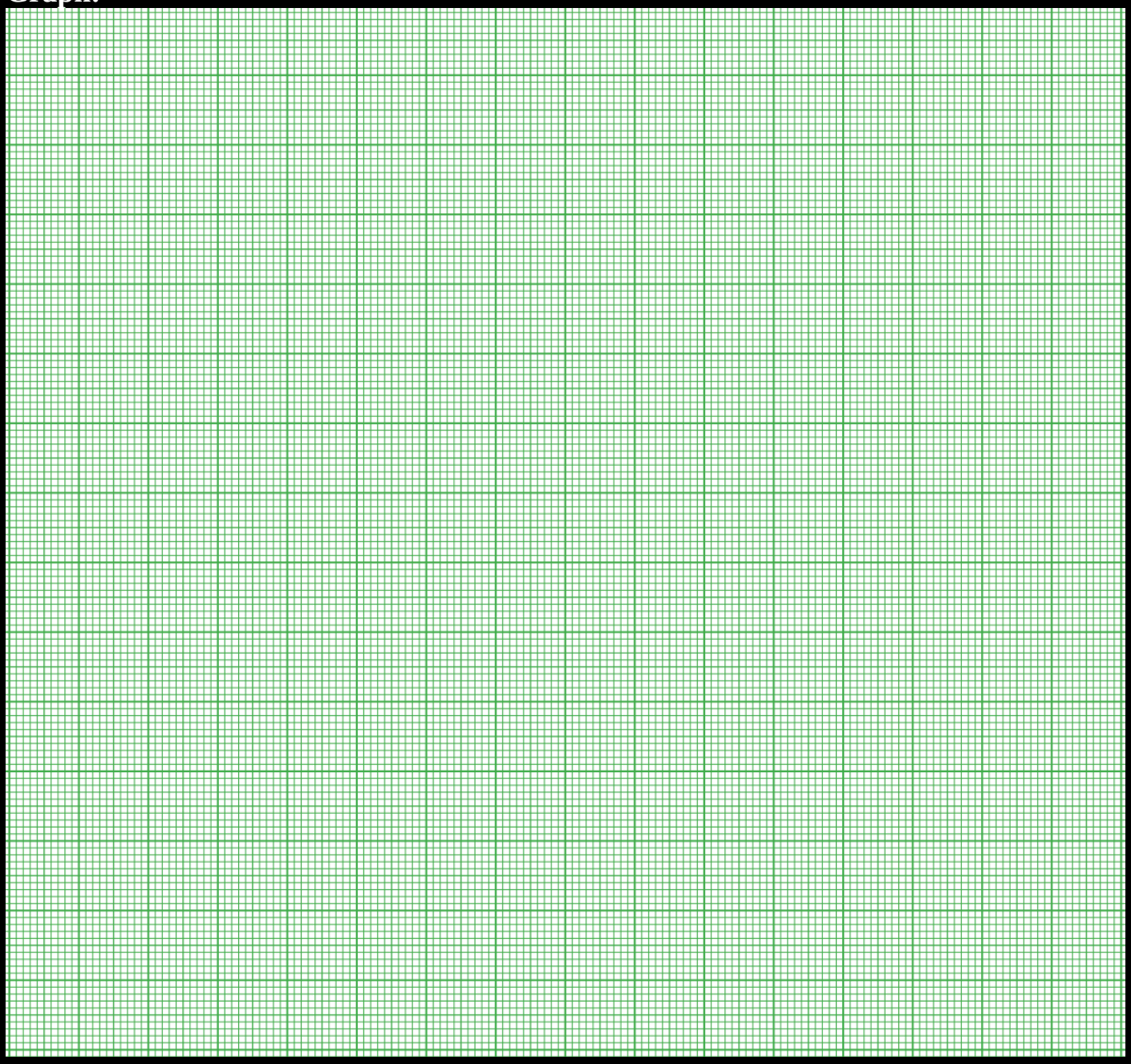
```
#include <iostream>
#include <cstdlib>
#include <chrono>
using namespace std;
using namespace std::chrono;
int partition(int arr[], int low, int high) {
int pivot = arr[high];
int i = low - 1;
for (int j = low; j < high; ++j) {
if (arr[j] <= pivot) {
++i;
swap(arr[i], arr[j]);
}
}
swap(arr[i + 1], 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;
}
```

Output:

```
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
```

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:

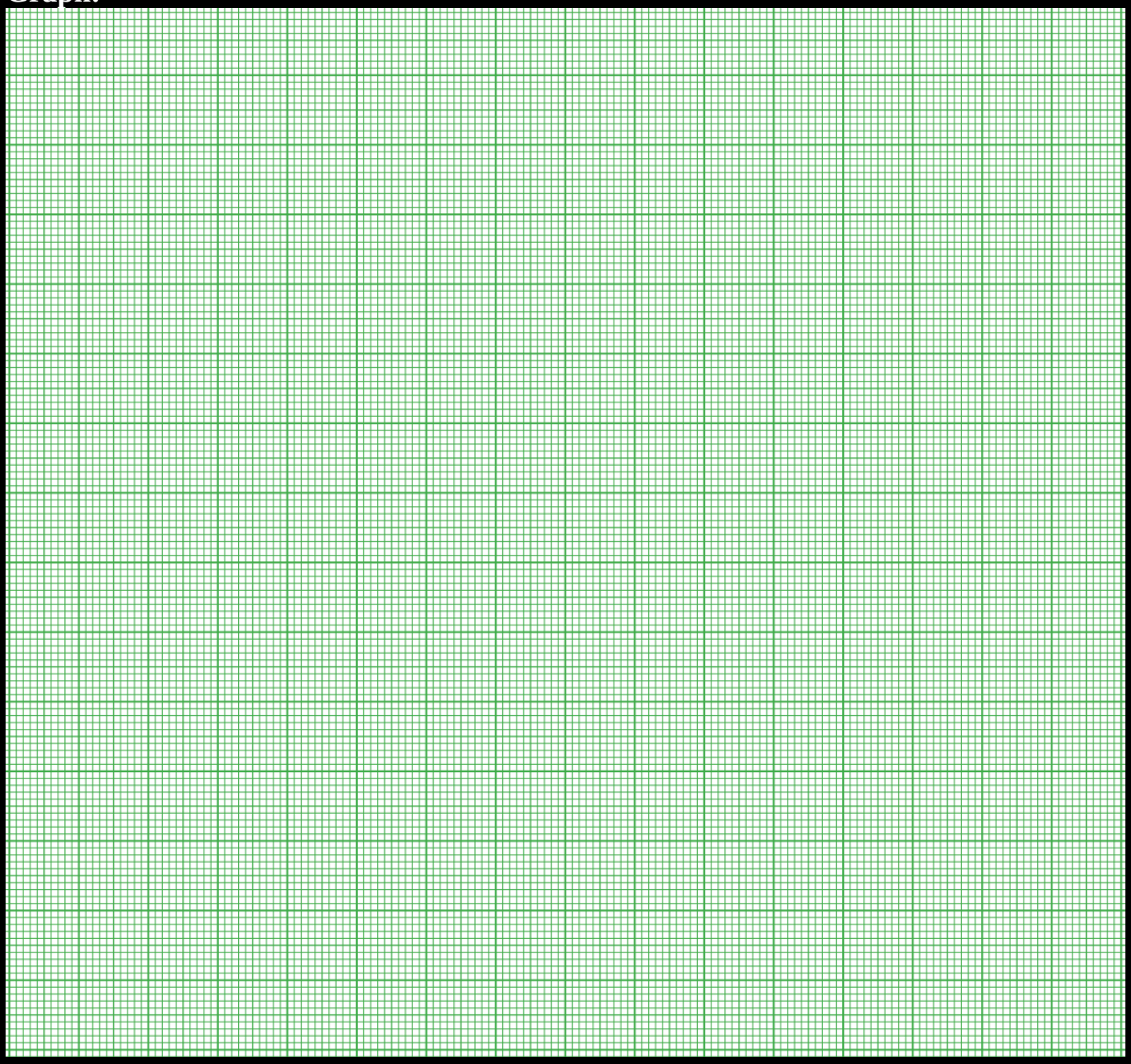
```
#include <iostream>
#include <cstdlib>
#include <chrono>
using namespace std;
using namespace std::chrono;
void merge(int arr[], int left, int mid, int right) {
int n1 = mid - left + 1;
int n2 = right - mid;
int* L = new int[n1];
int* R = new int[n2];
for (int i = 0; i < n1; ++i)
L[i] = arr[left + 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; }
```

Output:

```
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
```

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 **e**qual 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:

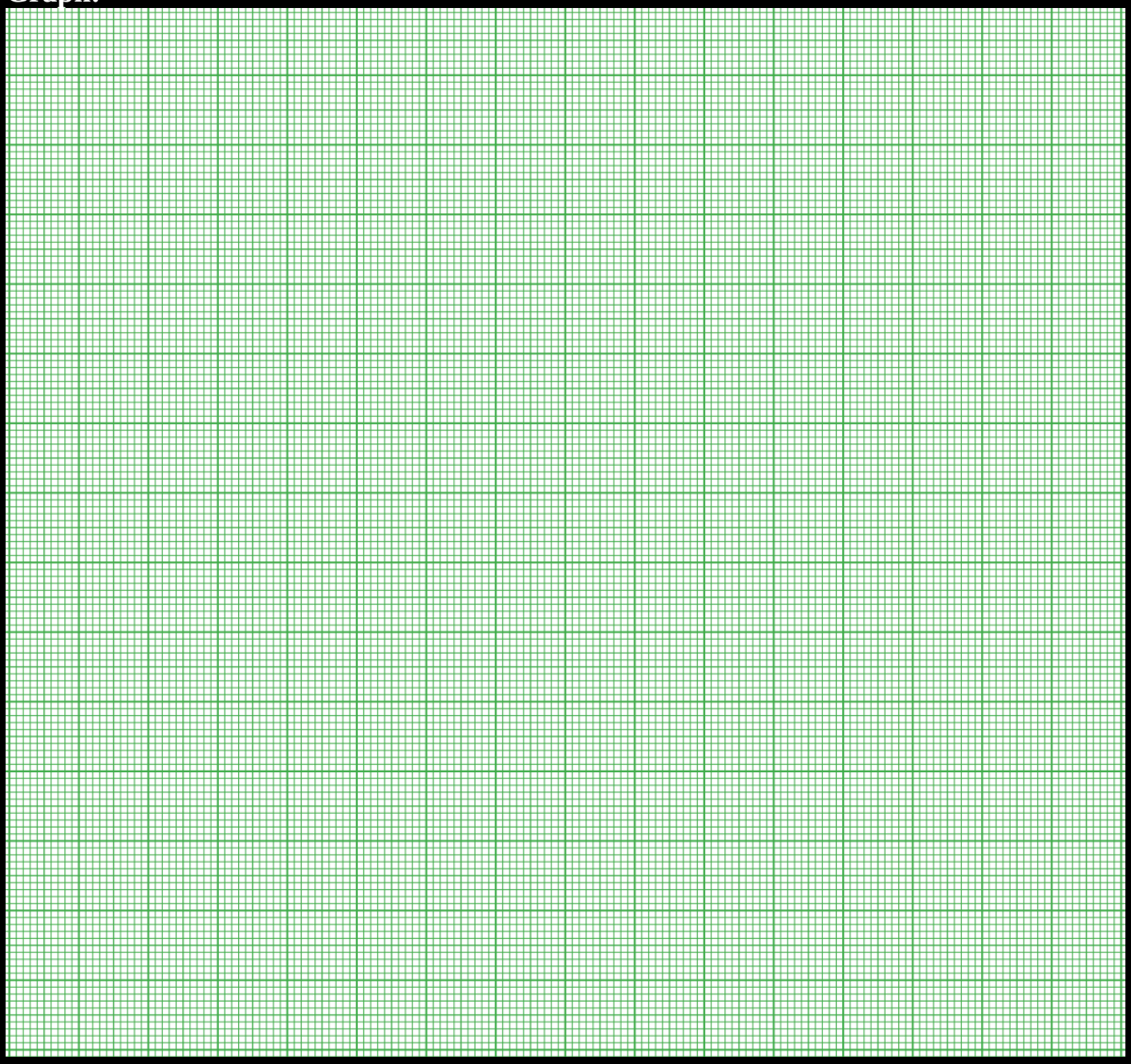
```
#include <iostream>
#include <cstdlib>
#include <chrono>
using namespace std;
using namespace std::chrono;
void heapify(int arr[], int n, int i) {
int largest = i;
int left = 2 * i + 1;
int right = 2 * i + 2;
if (left < n && arr[left] > arr[largest])
largest = left;
if (right < n && arr[right] > 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;
}
```

Output:

```
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
```

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:

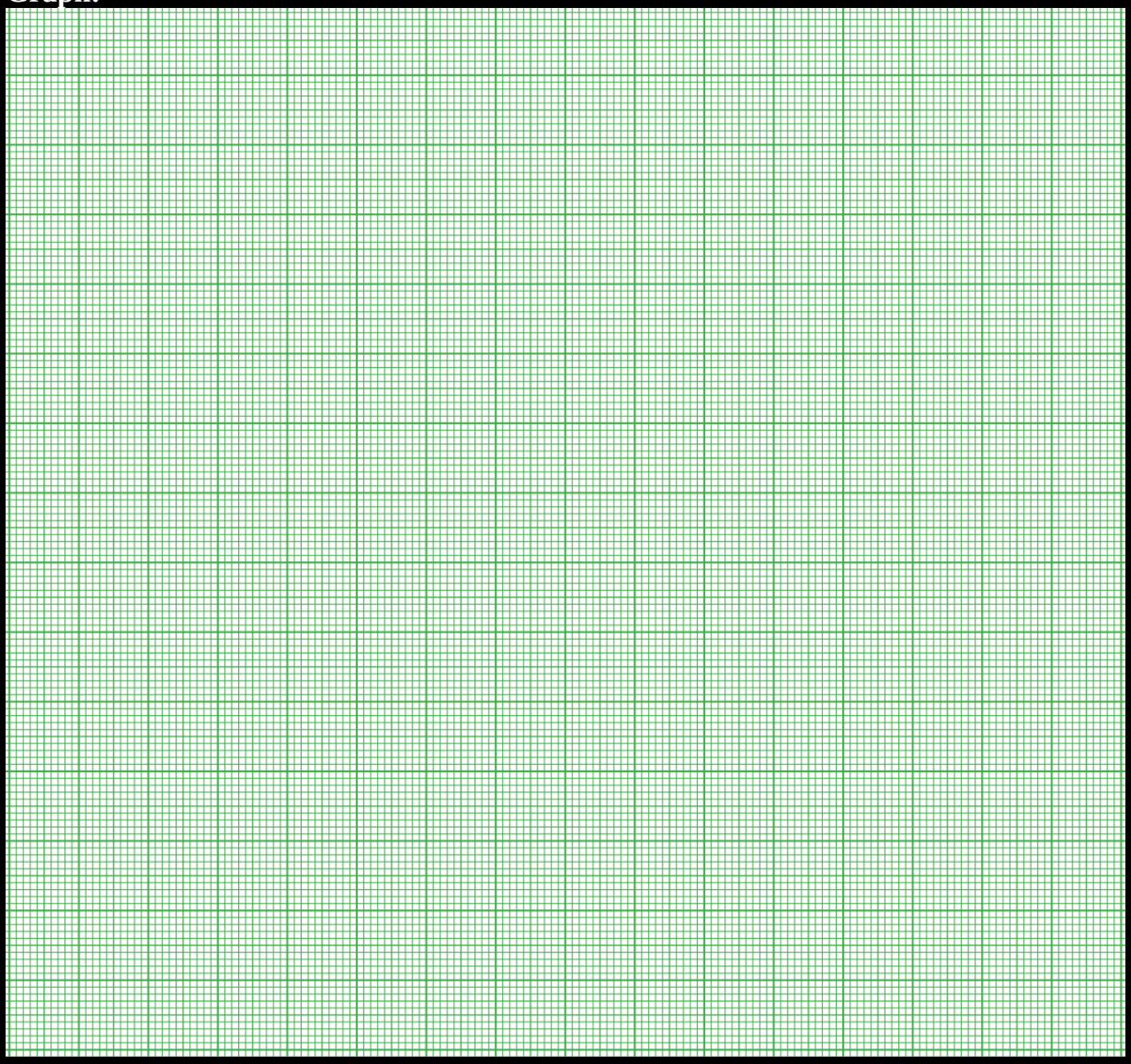
```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;
int linearSearch(int arr[], int n, int key)
{
for (int i = 0; i < n; i++)
{
if (arr[i] == key)
{
```

```
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;
}
```

Output:

```
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
```

Graph:



Learning Outcomes:

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:

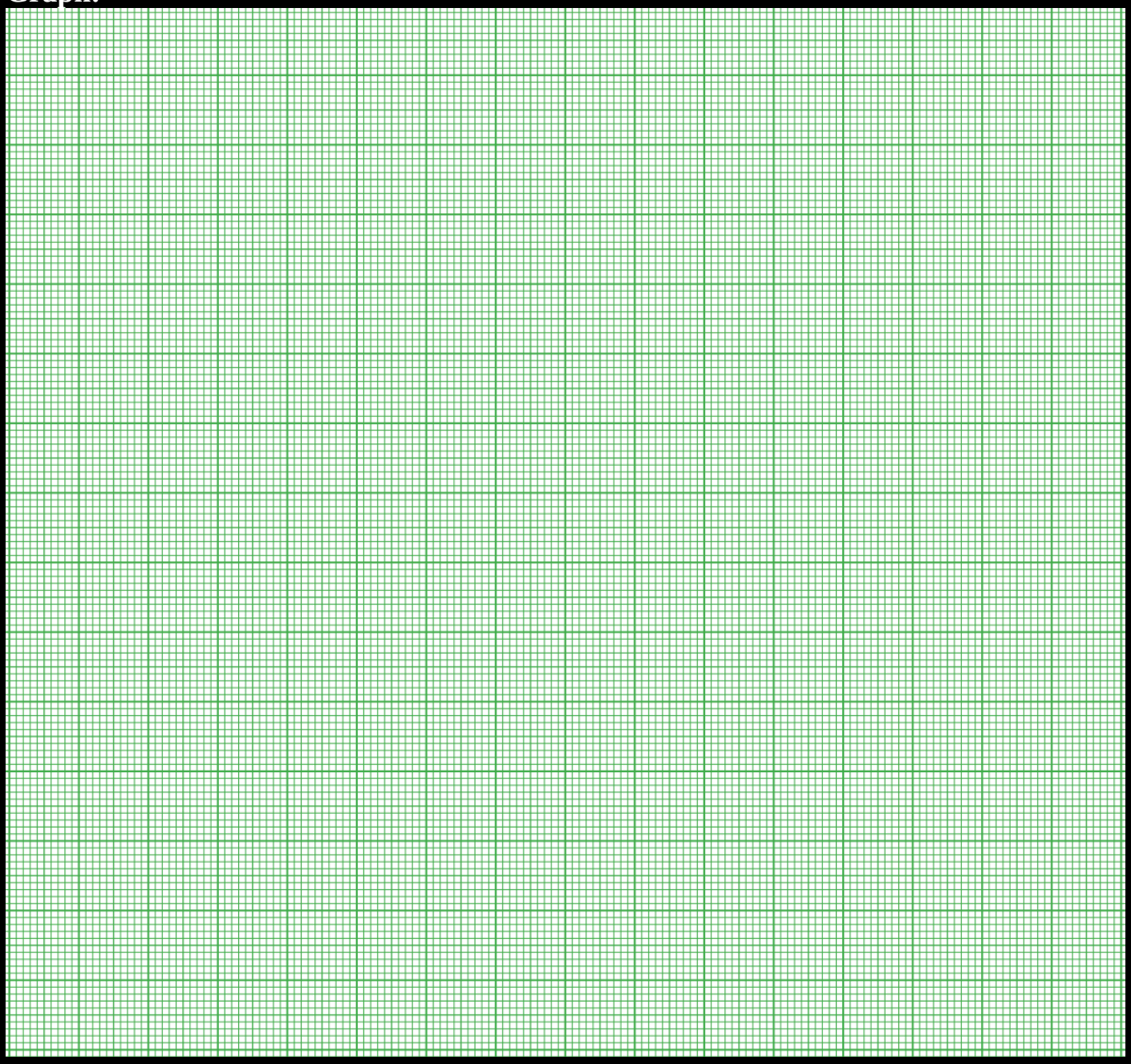
```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;
int binarySearch(int arr[], int n, int key)
{
int low = 0;
int high = n - 1;
```

```
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; }
```

Output:

```
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
```

Graph:



Learning Outcomes:

PROGRAM 3

AIM :- To implement Huffman Coding and analyze its time complexity

THEORY:

Huffman coding is a lossless compression algorithm that assigns variable-length codes to characters based on their frequencies. More frequent characters get shorter codes, while less frequent ones get longer codes. The steps are:

1. **Frequency Analysis:** Count character frequencies in the input.
2. **Priority Queue:** Build a min-heap where each node represents a character and its frequency.
3. **Huffman Tree:**
    - Repeatedly extract the two nodes with the lowest frequencies.
    - Combine them into a new node and insert it back into the heap.
    - Continue until one node remains, which becomes the root of the Huffman Tree.
4. **Generate Codes:** Traverse the tree to assign binary codes to characters.

Time Complexity Analysis

1. **Frequency Analysis: O(n),** where n is the input length.
2. **Building the Priority Queue and Huffman Tree: O(m log m),** where m is the number of unique characters.
3. **Generating Codes: O(m).**

Overall, the time complexity is dominated by the frequency analysis, making it:

O(n)

This makes Huffman coding efficient for compressing data with skewed frequency distributions.

CODE:-

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

using namespace std;
using namespace std::chrono;

// A Huffman tree node
struct MinHeapNode {
    char data;
```

```
    int freq;
    MinHeapNode *left, *right;

    MinHeapNode(char data, int freq) {
        left = right = nullptr;
        this->data = data;
        this->freq = freq;
    }
};

// For comparison of two heap nodes (needed for min heap)
struct compare {
    bool operator()(MinHeapNode* l, MinHeapNode* r) {
        return (l->freq > r->freq);
    }
};

// Print the codes of each character from the root of Huffman tree
void printCodes(struct MinHeapNode* root, string str, unordered_map<char, string>&
huffmanCode) {
    if (!root)
        return;

    // If this is a leaf node
    if (!root->left && !root->right)
        huffmanCode[root->data] = str;

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

// Build the Huffman tree and print codes
void HuffmanCodes(unordered_map<char, int>& freqMap) {
    struct MinHeapNode *left, *right, *top;

    // Create a min heap & inserts all characters of data[]
    priority_queue<MinHeapNode*, vector<MinHeapNode*>, compare> minHeap;

    for (auto pair: freqMap)
        minHeap.push(new MinHeapNode(pair.first, pair.second));

    // Iterate until size of heap doesn't become 1
    while (minHeap.size() != 1) {
        // Extract the two minimum freq items from heap
        left = minHeap.top();
        minHeap.pop();
        right = minHeap.top();
        minHeap.pop();
```

```
        // Create a new internal node with frequency equal to the sum of the two nodes'
frequencies.
        top = new MinHeapNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        minHeap.push(top);
    }

    // Store Huffman codes
    unordered_map<char, string> huffmanCode;
    printCodes(minHeap.top(), "", huffmanCode);

    // Print the Huffman codes
    cout << "\nCharacter\tHuffman Code\n";
    for (auto pair: huffmanCode)
        cout << pair.first << "\t\t" << pair.second << '\n';
}

int main() {
    string input;
    cout << "Enter a string: ";
    getline(cin, input);

    // Step 1: Calculate frequency of each character
    unordered_map<char, int> freqMap;
    for (char ch : input) {
        freqMap[ch]++;
    }

    // Measure time for Huffman coding
    auto start = high_resolution_clock::now();

    // Step 2: Generate Huffman Codes
    HuffmanCodes(freqMap);

    auto end = high_resolution_clock::now();
    auto time_spent = duration_cast<nanoseconds>(end - start).count();
    cout << "\nTime taken for Huffman Coding: " << time_spent << " nanoseconds\n";

    return 0;
}
```

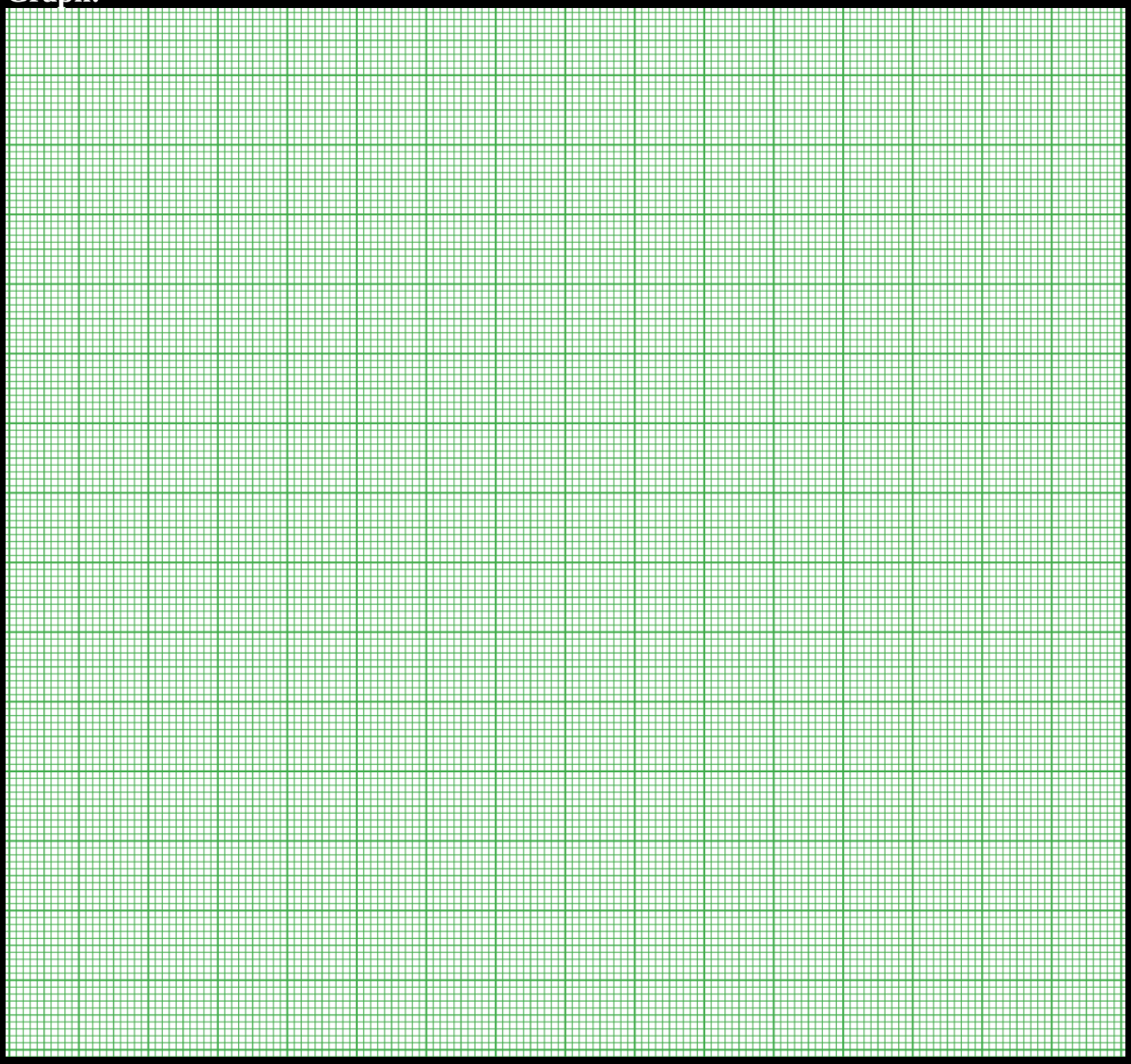
OUTPUT:-

```
Enter a string: hello

Character   Huffman Code
l       11
h       10
e       01
o       00

Time taken for Huffman Coding: 76260 nanoseconds


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

```
Enter a string: ninety

Character   Huffman Code
n       11
e       101
y       100
i       01
t       00

Time taken for Huffman Coding: 80480 nanoseconds


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

```
Enter a string: upgrade

Character   Huffman Code
a       110
u       101
p       100
r       111
e       011
d       010
g       00

Time taken for Huffman Coding: 98770 nanoseconds


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

Graph:



Learning Outcomes:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // Include for clock() function
// Comparator function to use in sorting
int comparator(const void *p1, const void *p2)
{
    const int(*x)[3] = p1;
    const int(*y)[3] = p2;
    return (*x)[2] - (*y)[2];
}
// Initialization of parent[] and rank[] arrays
void makeSet(int parent[], int rank[], int n)
{
    for (int i = 0; i < n; i++)
```

```
    {
        parent[i] = i;
        rank[i] = 0;
    }
}
// Function to find the parent of a node
int findParent(int parent[], int component)
{
    if (parent[component] == component)
        return component;
    return parent[component] = findParent(parent, parent[component]);
}

// Function to unite two sets
void unionSet(int u, int v, int parent[], int rank[], int n)
{
    // Finding the parents
    u = findParent(parent, u);
    v = findParent(parent, v);
    if (rank[u] < rank[v])
    {
        parent[u] = v;
    }
    else if (rank[u] > rank[v])
    {
        parent[v] = u;
    }
    else
    {
        parent[v] = u;
        // Since the rank increases if
        // the ranks of two sets are same
        rank[u]++;
    }
```

```
}
// Function to find the MST
void kruskalAlgo(int n, int edge[n][3])
{
    // First we sort the edge array in ascending order
    // so that we can access minimum distances/cost
    qsort(edge, n, sizeof(edge[0]), comparator);
    int parent[n];
    int rank[n];
    // Function to initialize parent[] and rank[]
    makeSet(parent, rank, n);
    // To store the minimum cost
    int minCost = 0;
    printf("Following are the edges in the constructed MST\n");
    for (int i = 0; i < n; i++)
    {
        int v1 = findParent(parent, edge[i][0]);
        int v2 = findParent(parent, edge[i][1]);
        int wt = edge[i][2];
        // If the parents are different that
        // means they are in different sets so
        // union them
        if (v1 != v2)
        {
            unionSet(v1, v2, parent, rank, n);
            minCost += wt;
            printf("%d -- %d == %d\n", edge[i][0], edge[i][1], wt);
        }
    }
    printf("Minimum Cost Spanning Tree: %d\n", minCost);
}

// Driver code
int main()
```

```
{
    // Input edges: {node1, node2, weight}
    int edge[5][3] = {{0, 1, 10},
                      {0, 2, 6},
                      {0, 3, 5},
                      {1, 3, 15},
                      {2, 3, 4}};
    // Start the clock before running the algorithm
    clock_t start_time = clock();
    // Call the Kruskal's algorithm to find the MST
    kruskalAlgo(5, edge);
    // End the clock after running the algorithm
    clock_t end_time = clock();
    // Calculate the time taken in milliseconds
    double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC * 1000;
    // Print the time taken to execute the program
    printf("Time taken: %.2f ms\n", time_taken);
    return 0;
}
```

Output:

```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
Time taken: 0.10 ms
```

Learning Outcome:

The Matrix Chain Multiplication problem aims to find the most efficient way to multiply a given sequence of matrices. The goal is to minimize the total number of scalar multiplications required.

Algorithm:
1. Matrix Dimensions: Let the dimensions of the matrices be represented as an array, where the i-th matrix has dimensions p[i-1] x p[i].
2. Dynamic Programming Table: Create a 2D table `m` where m[i][j] represents the minimum number of scalar multiplications needed to multiply matrices from index i to j.
3. Filling the Table:
  - For chain lengths from 2 to n, compute the minimum cost of multiplying matrices from i to j by trying all possible splits.
  - For each split point k between i and j, calculate the cost as m[i][k] + m[k+1][j] + (dimensions of the resulting matrix). Update m[i][j] with the minimum cost found.
4. Result: The minimum number of multiplications for the entire chain is stored in m[1][n].
Time Complexity: The time complexity of this algorithm is O(n^3), where n is the number of matrices.
Space Complexity: The space complexity is O(n^2) due to the storage of the dynamic programming table.

Program:

```
#include <iostream>
#include <climits>
using namespace std;
#define MAX 30

int matrixChainOrder(int p[], int n) {
   int m[MAX][M AX];  // m[i][j] holds the minimum multiplication cost for matrices i to j
   for (int i = 1; i < n; i++) {
      m[i][i] = 0}

   for (int L = 2; L < n; L++) {
      for (int i = 1; i <= n - L; i++) {
         int j = i + L - 1;
         m[i][j] = INT_MAX;

         // Find minimum cost for multiplying matrices i to j
         for (int k = i; k < j; k++) {
            int cost = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
            if (cost < m[i][j]) {
               m[i][j] = cost;}
         }
      }
   }
   return m[1][n - 1];}
```

```
int main() {
   int n;
   cout << "Enter the number of matrices: ";
   cin >> n;
   int arr[MAX];
   cout << "Enter the dimensions of matrices: ";
   for (int i = 0; i <= n; i++) {
      cin >> arr[i];}
   int minCost = matrixChainOrder(arr, n + 1);
   cout << "Minimum number of multiplications is: " << minCost << endl;
   return 0;}
```

Output:

```
Enter the number of matrices: 3
Enter the dimensions of matrices: 5 10 15 20
Minimum number of multiplications is: 2250
```

Code:

```
#include <limits.h>

#include <stdio.h>

#include <time.h> // Include time.h for clock()

int MatrixChainOrder(int p[], int i, int j) {

        if (i == j) return 0;

        int k, min = INT_MAX, count;

        for (k = i; k < j; k++) {

                count = MatrixChainOrder(p, i, k) + MatrixChainOrder(p, k + 1, j) + p[i - 1] * p[k] * p[j];

                if (count < min) min = count;

        }

        return min;

}

int main() {

        int arr[] = {1, 2, 3, 4, 3};

        int N = sizeof(arr) / sizeof(arr[0]);

        clock_t start = clock(); // Start clock
```

```
    printf("Minimum number of multiplications is %d ", MatrixChainOrder(arr, 1, N - 1));

    clock_t end = clock(); // End clock

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000; // Time in ms

    printf("\nTime taken: %.4f ms\n", time_taken); // Output time

    return 0;
}
```

Output:

```
Minimum number of multiplications is 30
Time taken: 0.0300 ms


...Program finished with exit code 0
Press ENTER to exit console.
```

Learning Outcome:

Code:

```
#include <stdio.h>
#include <limits.h>
#include <time.h> // Include time.h for clock()
#define MAX_VERTICES 100

int minDistance(int dist[], int sptSet[], int vertices) {
    int min = INT_MAX, minIndex;
    for (int v = 0; v < vertices; v++) {
        if (!sptSet[v] && dist[v] < min) {
            min = dist[v];
            minIndex = v;
        }
    }
    return minIndex;
}

void printSolution(int dist[], int vertices) {
    printf("Vertex \tDistance from Source\n");
    for (int i = 0; i < vertices; i++) {
        printf("%d \t%d\n", i, dist[i]);
    }
}

void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int src, int vertices) {
    int dist[MAX_VERTICES], sptSet[MAX_VERTICES];
    for (int i = 0; i < vertices; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = 0;
    }
    dist[src] = 0;
    for (int count = 0; count < vertices - 1; count++) {
        int u = minDistance(dist, sptSet, vertices);
        sptSet[u] = 1;
        for (int v = 0; v < vertices; v++) {
```

```
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }
    printSolution(dist, vertices);
}

int main() {
    int vertices;
    printf("Input the number of vertices: ");
    scanf("%d", &vertices);
    if (vertices <= 0 || vertices > MAX_VERTICES) {
        printf("Invalid number of vertices. Exiting...\n");
        return 1;
    }

    int graph[MAX_VERTICES][MAX_VERTICES];
    printf("Input the adjacency matrix for the graph (use INT_MAX for infinity):\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    int source;
    printf("Input the source vertex: ");
    scanf("%d", &source);
    if (source < 0 || source >= vertices) {
        printf("Invalid source vertex. Exiting...\n");
        return 1;
    }

    clock_t start = clock(); // Start clock
    dijkstra(graph, source, vertices);
    clock_t end = clock(); // End clock

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000; // Time in ms
    printf("Time taken: %.4f ms\n", time_taken); // Output time

    return 0;
}
```

Output:

```
Input the number of vertices: 5
Input the adjacency matrix for the graph (use INT_MAX for infinity):
0 3 2 0 0
3 0 0 1 0
2 0 0 1 4
0 1 1 0 2
0 0 4 2 0
Input the source vertex: 0
Vertex  Distance from Source
0       0
1       3
2       2
3       3
4       5
Time taken: 0.0410 ms
```

Learning Outcome:

Code:

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // Include time.h for clock()

#define MAX_VERTICES 1000
#define INF INT_MAX

void bellmanFord(int graph[MAX_VERTICES][MAX_VERTICES], int vertices, int edges, int source) {
  int distance[MAX_VERTICES];
  for (int i = 0; i < vertices; ++i) distance[i] = INF;
  distance[source] = 0;

  for (int i = 0; i < vertices - 1; ++i) {
    for (int j = 0; j < edges; ++j) {
      if (graph[j][0] != -1 && distance[graph[j][0]] != INF && distance[graph[j][1]] > distance[graph[j][0]] + graph[j][2])
        distance[graph[j][1]] = distance[graph[j][0]] + graph[j][2];
    }
  }

  for (int i = 0; i < edges; ++i) {
    if (graph[i][0] != -1 && distance[graph[i][0]] != INF && distance[graph[i][1]] > distance[graph[i][0]] + graph[i][2]) {
      printf("Negative cycle detected\n");
      return;
    }
  }

  printf("Vertex   Distance from Source\n");
  for (int i = 0; i < vertices; ++i)
    printf("%d \t\t %d\n", i, distance[i]);
}

int main() {
  int vertices = 6;
```

```
    int edges = 8;
    int graph[MAX_VERTICES][MAX_VERTICES] = {{0, 1, 5}, {0, 2, 7}, {1, 2, 3}, {1, 3, 4}, {1, 4, 6}, {3, 4, -1}, {3, 5, 2}, {4, 5, -3}};

    clock_t start = clock(); // Start clock
    bellmanFord(graph, vertices, edges, 0);
    clock_t end = clock(); // End clock

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000; // Time in ms
    printf("Time taken: %.4f ms\n", time_taken); // Output time

    return 0;
}
```

Output:

```
Vertex   Distance from Source
0                0
1                5
2                7
3                9
4                8
5                5
Time taken: 0.0970 ms
```

Learning Outcome:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h> // Include time.h for clock

int board[20], count = 0; // Initialize count to 0

void print(int n);
int place(int row, int column);
void queen(int row, int n);

int main() {
  int n;

  printf(" - N Queens Problem Using Backtracking -\n\n");
  printf("Enter number of Queens: ");
  scanf("%d", &n);

  clock_t start = clock(); // Start clock
  queen(1, n);
  clock_t end = clock(); // End clock

  double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000; // Time in ms
  printf("\nTime taken: %.4f ms\n", time_taken); // Output time

  return 0;
}

// Function for printing the solution
void print(int n) {
  int i, j;
  printf("\n\nSolution %d:\n\n", ++count);
  for (i = 1; i <= n; ++i) printf("\t%d", i);
  for (i = 1; i <= n; ++i) {
    printf("\n\n%d", i);
    for (j = 1; j <= n; ++j) { // for nxn board
      if (board[i] == j) printf("\tQ"); // queen at i,j position
      else printf("\t-"); // empty slot
    }
  }
}
```

```
// Function to check conflicts
int place(int row, int column) {
   int i;
   for (i = 1; i <= row - 1; ++i) {
      // Checking column and diagonal conflicts
      if (board[i] == column) return 0;
      else if (abs(board[i] - column) == abs(i - row)) return 0;
   }
   return 1; // no conflicts
}

// Function to check for proper positioning of queen
void queen(int row, int n) {
   int column;
   for (column = 1; column <= n; ++column) {
      if (place(row, column)) {
         board[row] = column; // No conflicts, so place queen
         if (row == n)        // All queens are placed
            print(n);         // Print the board configuration
         else                 // Try queen with the next position
            queen(row + 1, n);
      }
   }
}
```

Output:

```
  - N Queens Problem Using Backtracking -

Enter number of Queens: 4


Solution 1:

         1       2       3       4

1        -       Q       -       -

2        -       -       -       Q

3        Q       -       -       -

4        -       -       Q       -

Solution 2:

         1       2       3       4

1        -       -       Q       -

2        Q       -       -       -

3        -       -       -       Q

4        -       Q       -       -
Time taken: 0.1920 ms
```

Learning Outcome:

The Longest Common Subsequence (LCS) problem identifies the longest sequence of characters that appear in the same order in both input strings, not necessarily contiguously. This problem is efficiently solved using dynamic programming.

Algorithm :

1. Create a 2D array, LCS table, where LCS table[i][j] stores the length of the LCS for substrings S1[0...i-1] and S2[0...j-1].
2. Initialize the first row and column to 0 for cases where one string is empty.
3. For each character comparison:
    - If characters match (S1[i-1] == S2[j-1]), set LCS table[i][j] = LCS table[i-1][j-1] + 1.
    - If they don't match, set LCS table[i][j] = max(LCS table[i-1][j], LCS table[i][j-1]).
4. The length of the LCS is found in LCS table[m][n].

Traceback to Find LCS:

- Start from the bottom-right of LCS table and trace back to reconstruct the LCS:
    - If S1[i-1] == S2[j-1], add this character to the LCS and move diagonally up-left.
    - If not, move in the direction of the higher value (left or above) until reaching the top-left cell.

Time Complexity: O(m × n), where m and n are the lengths of the two strings, due to the nested loops filling LCS table.

Space Complexity: O(m × n), as LCS table stores intermediate values for each substring combination.

Program:

```
#include <iostream>
#include <cstring>
using namespace std;
void lcsAlgo(char *S1, char *S2, int m, int n) {
 int LCS_table[m + 1][n + 1];
 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];
 char lcsAlgo[index + 1];
 lcsAlgo[index] = '\0';
 int i = m, j = n;
 while (i > 0 && j > 0) {
  if (S1[i - 1] == S2[j - 1]) {
   lcsAlgo[index - 1] = S1[i - 1];
```

```
      i--;
      j--;
      index--;}
    else if (LCS_table[i - 1][j] > LCS_table[i][j - 1])
      i--;
    else
      j--;}
  cout << "S1 : " << S1 << "\nS2 : " << S2 << "\nLCS: " << lcsAlgo << "\n";}
int main() {
  char S1[50], S2[50];
  cout << "Enter the first string: ";
  cin >> S1;
  cout << "Enter the second string: ";
  cin >> S2;
  int m = strlen(S1);
  int n = strlen(S2);
  lcsAlgo(S1, S2, m, n);
  return 0;}
```

Output:

```
Enter the first string: Complex
Enter the second string: Complicated
S1 : Complex
S2 : Complicated
LCS: Comple
```

Code:

```
#include <stdio.h>
#include <string.h>
#include <time.h> // Include time.h for clock()

int max(int x, int y) { return x > y ? x : y; }

// Returns length of LCS for s1[0..m-1], s2[0..n-1]
int lcs(char *s1, char *s2, int m, int n) {
   if (m == 0 || n == 0) return 0; // Base case: If either string is empty, LCS length is 0
   if (s1[m - 1] == s2[n - 1]) // If last characters match
      return 1 + lcs(s1, s2, m - 1, n - 1); // Include character and recurse
   else
      return max(lcs(s1, s2, m, n - 1), lcs(s1, s2, m - 1, n)); // Recur without last character
}

int main() {
   char s1[] = "AGGTAB";
   char s2[] = "GXTXAYB";
   int m = strlen(s1);
   int n = strlen(s2);

   clock_t start = clock(); // Start clock
   printf("%d\n", lcs(s1, s2, m, n)); // Call LCS function and print result
   clock_t end = clock(); // End clock

   double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000; // Time in ms
```

```
    printf("Time taken: %.4f ms\n", time_taken); // Output time

    return 0;
}
```

Output:

```
4
Time taken: 0.0670 ms
```

Code:

1. Naive String-Matching algorithm

```
#include <stdio.h>
#include <string.h>
#include <time.h> // Include time.h for clock()

void search(char *pat, char *txt) {
   int M = strlen(pat);
   int N = strlen(txt);

   // A loop to slide pat[] one by one
   for (int i = 0; i <= N - M; i++) {
      int j;

      // For current index i, check for pattern match
      for (j = 0; j < M; j++) {
         if (txt[i + j] != pat[j]) {
            break;
         }
      }

      // If pattern matches at index i
      if (j == M) {
         printf("Pattern found at index %d\n", i);
      }
   }
}

int main() {
   char txt1[] = "AABAACAADAABAABA";
   char pat1[] = "AABA";
   char txt2[] = "agd";
   char pat2[] = "g";
```

```
    clock_t start = clock(); // Start clock
    printf("Example 1:\n");
    search(pat1, txt1);
    printf("\nExample 2:\n");
    search(pat2, txt2);
    clock_t end = clock(); // End clock

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000; // Time in ms
    printf("\nTime taken: %.4f ms\n", time_taken); // Output time

    return 0;
}
```

Output:

```
Example 1:
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12

Example 2:
Pattern found at index 1

Time taken: 0.1080 ms
```

```
#include <stdio.h>
#include <string.h>
#include <time.h> // Include time.h for clock()

// d is the number of characters in the input alphabet
#define d 256

/* pat -> pattern
    txt -> text
    q -> A prime number
*/
void search(char pat[], char txt[], int q) {
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0; // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;

    // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < M - 1; i++)
        h = (h * d) % q;

    // Calculate the hash value of pattern and first window of text
    for (i = 0; i < M; i++) {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }

    // Slide the pattern over text one by one
    for (i = 0; i <= N - M; i++) {
```

```
      // Check the hash values of current window of text and pattern
      if (p == t) {
         /* Check for characters one by one */
         for (j = 0; j < M; j++) {
            if (txt[i + j] != pat[j])
               break;
         }

         // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
         if (j == M)
            printf("Pattern found at index %d \n", i);
      }

      // Calculate hash value for next window of text: Remove leading digit, add trailing digit
      if (i < N - M) {
         t = (d * (t - txt[i] * h) + txt[i + M]) % q;

         // We might get negative value of t, converting it to positive
         if (t < 0)
            t = (t + q);
      }
   }
}

/* Driver Code */
int main() {
   char txt[] = "GEEKS FOR GEEKS";
   char pat[] = "GEEK";
   int q = 101;

   clock_t start = clock(); // Start clock
   search(pat, txt, q); // Call search function
   clock_t end = clock(); // End clock

   double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000; // Time in ms
   printf("\nTime taken: %.4f ms\n", time_taken); // Output time

   return 0;
}
```

Output:

```
Pattern found at index 0
Pattern found at index 10

Time taken: 0.0500 ms
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h> // Include time.h for clock()

void constructLps(char *pat, int *lps) {
   int len = 0; // Length of the previous longest prefix suffix
   int i = 1;
```

```
    lps[0] = 0; // lps[0] is always 0

    // Build the lps array
    while (i < strlen(pat)) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        } else { // Mismatch
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

int *search(char *pat, char *txt, int *resCount) {
    int n = strlen(txt);
    int m = strlen(pat);
    int *lps = (int *)malloc(m * sizeof(int));
    int *res = (int *)malloc(n * sizeof(int)); // Max possible matches
    *resCount = 0;

    constructLps(pat, lps);

    int i = 0; // Index for txt
    int j = 0; // Index for pat

    while (i < n) {
        if (txt[i] == pat[j]) {
            i++;
            j++;

            if (j == m) {
                res[(*resCount)++] = i - j;
                j = lps[j - 1];
            }
        } else {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }

    free(lps);
    return res;
}

int main() {
    char txt[] = "aabaacaadaabaaba";
    char pat[] = "aaba";

    int resCount;

    clock_t start = clock(); // Start clock
    int *res = search(pat, txt, &resCount); // Call search function
    clock_t end = clock(); // End clock
```

```
    printf("Pattern matched at indexes: ");
    for (int i = 0; i < resCount; i++) {
        printf("%d ", res[i]);
    }

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC * 1000; // Time in ms
    printf("\nTime taken: %.4f ms\n", time_taken); // Output time

    free(res);
    return 0;
}
```

Output:

```
Pattern matched at indexes: 0 9 12
Time taken: 0.0180 ms
```

Learning Outcome: