Department of Computer Science & Engineering

Practical file submitted in partial fulfillment for the evaluation of

DESIGN AND ANALYSIS OF ALGORITHM (CIC-359)

Submitted To:
Ms. Shikha Jain

Submitted By:
Saurav Malik
06017702722
CSE A

योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION

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 f) Insertion sort							
2	To implement Linear search and Binary search and analyse its time complexity.							
3	To implement Huffman Coding and analyze its time complexity							

4	To implement Minimum Spanning Tree using Kruskal's Algorithm and analyse its time complexity.							
5	To implement Matrix chain Multiplication program							
6	To implement Dijkstra algorithm and analyse its time complexity							
7	To implement Bellman Ford algorithm and analyse its time complexity.							
8	To implement n-Queen problem using backtracking.							
9	To implement Longest Common Subsequence problem and analyse its time complexity.							
10	To implement Naive String-Matching algorithm, Rabin Karp algorithm and knuth Morris Pratt algorithm 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:

Program:

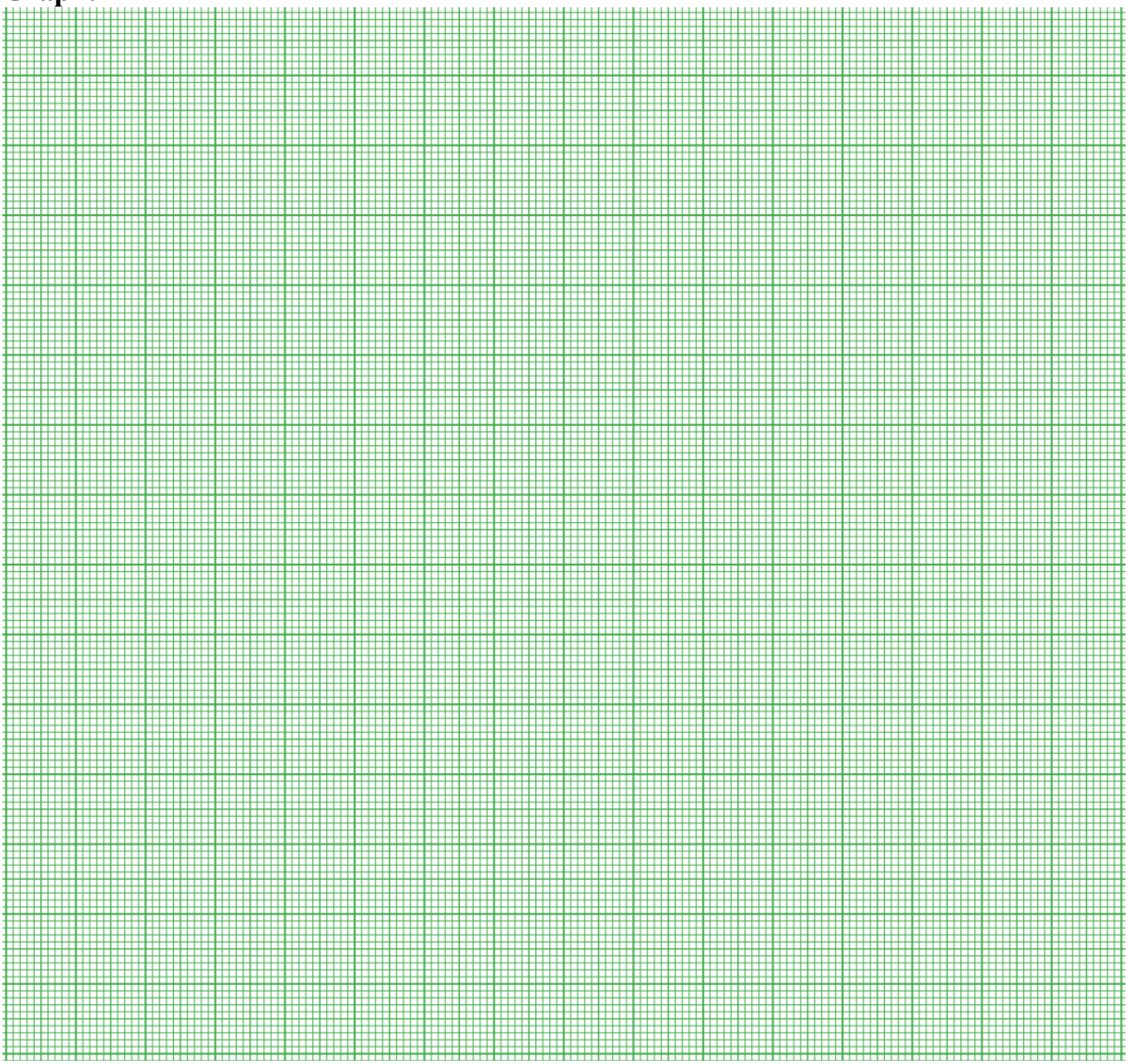
```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;
void insertionSort(int arr[], int n) {
for (int i = 1; i < n; ++i) {
int key = arr[i];
int j = i - 1;
//move one position right if greater than key
while (j >= 0 && arr[j] > key) {
arr[j + 1]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

Program:

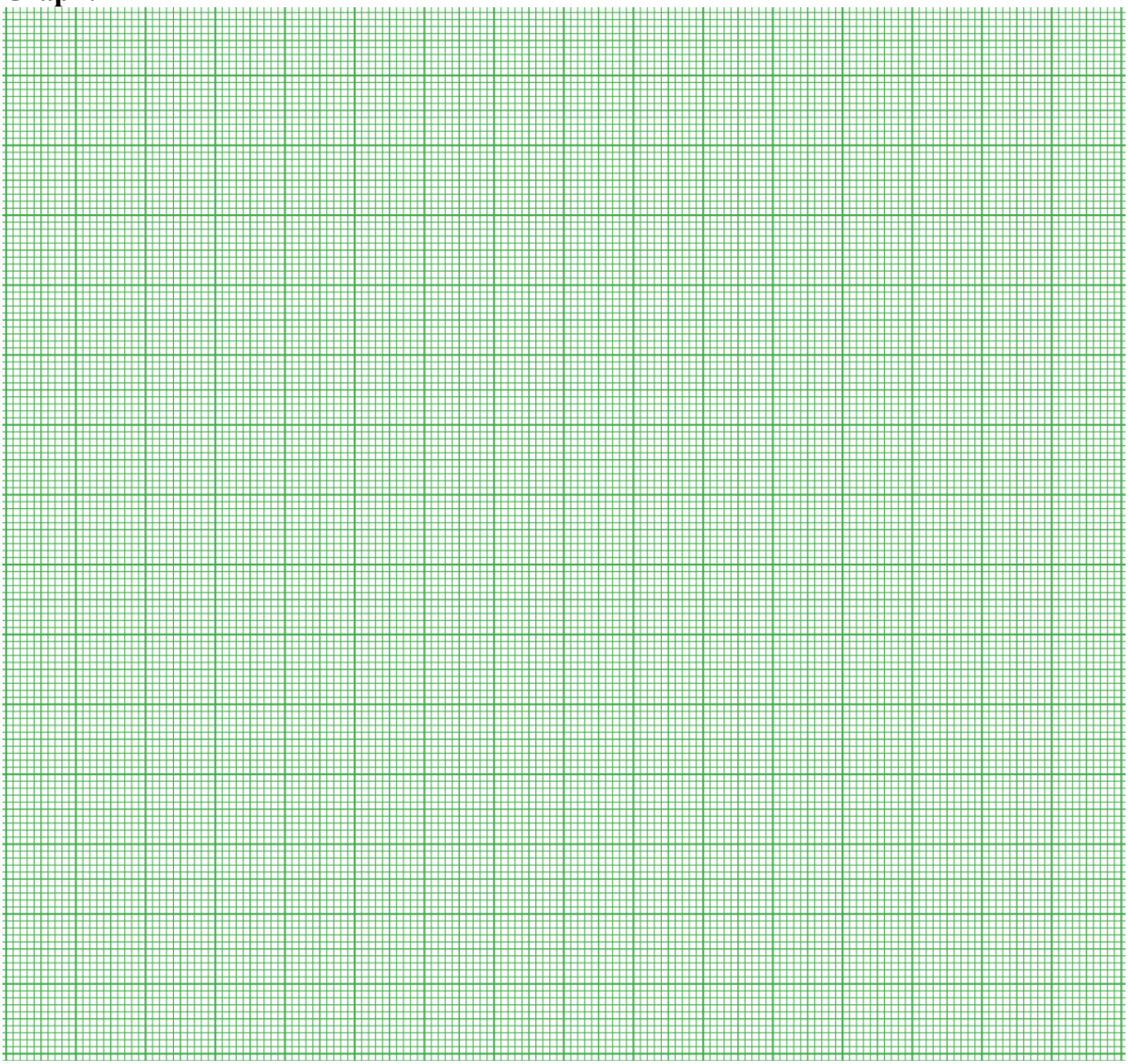
```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;
void selectionSort(int arr[], int n)
{
for (int i = 0; i < n - 1; ++i) {

// 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:

Program:

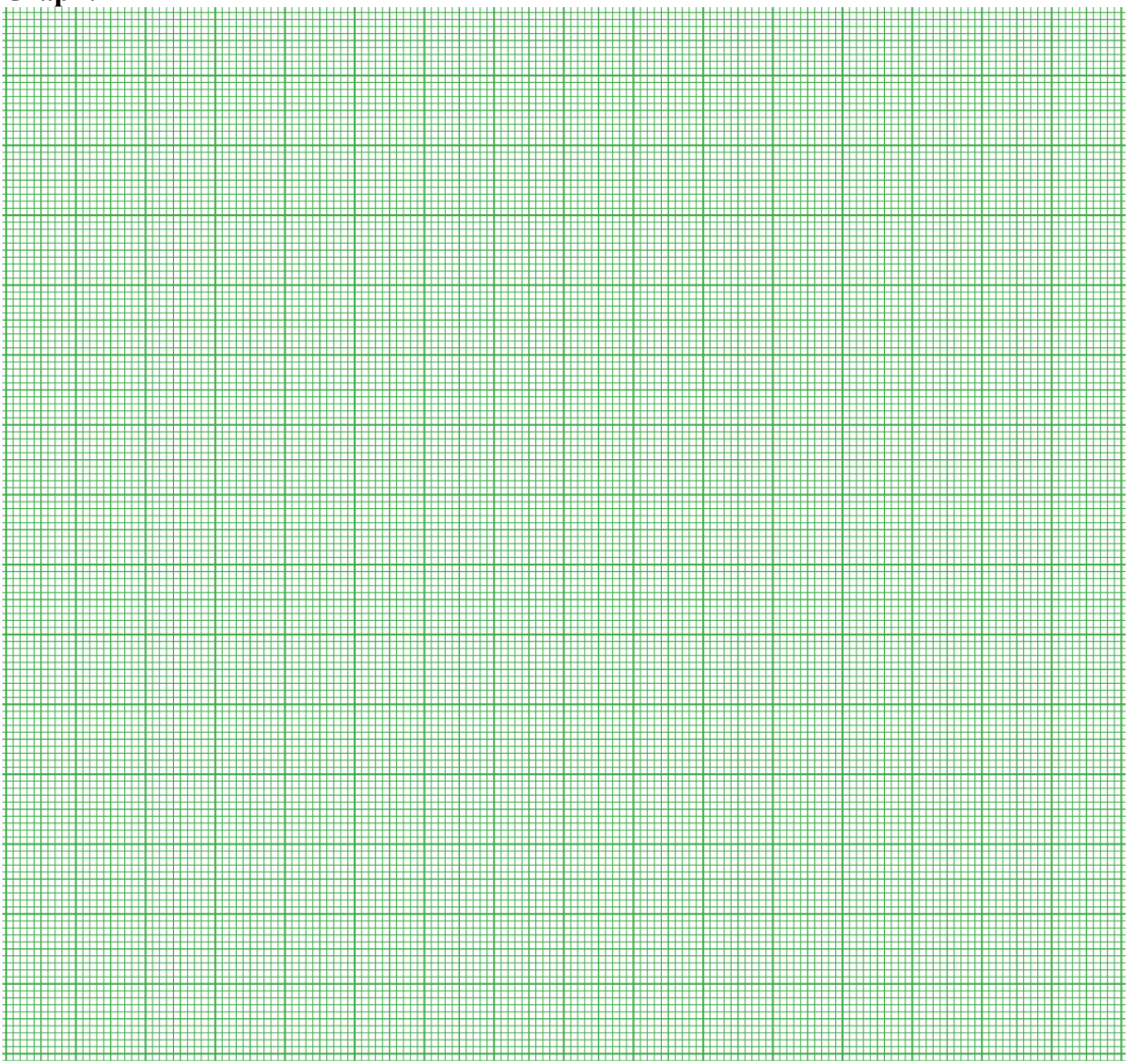
```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;
void bubbleSort(int arr[], int n) {
int flag=0;
for (int i = 0; i < n - 1; ++i) {
flag=0;
for (int j = 0; j < n - i - 1; ++j) {
if (arr[j] > arr[j + 1]) {
// Swap if the element found is greater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

Program:

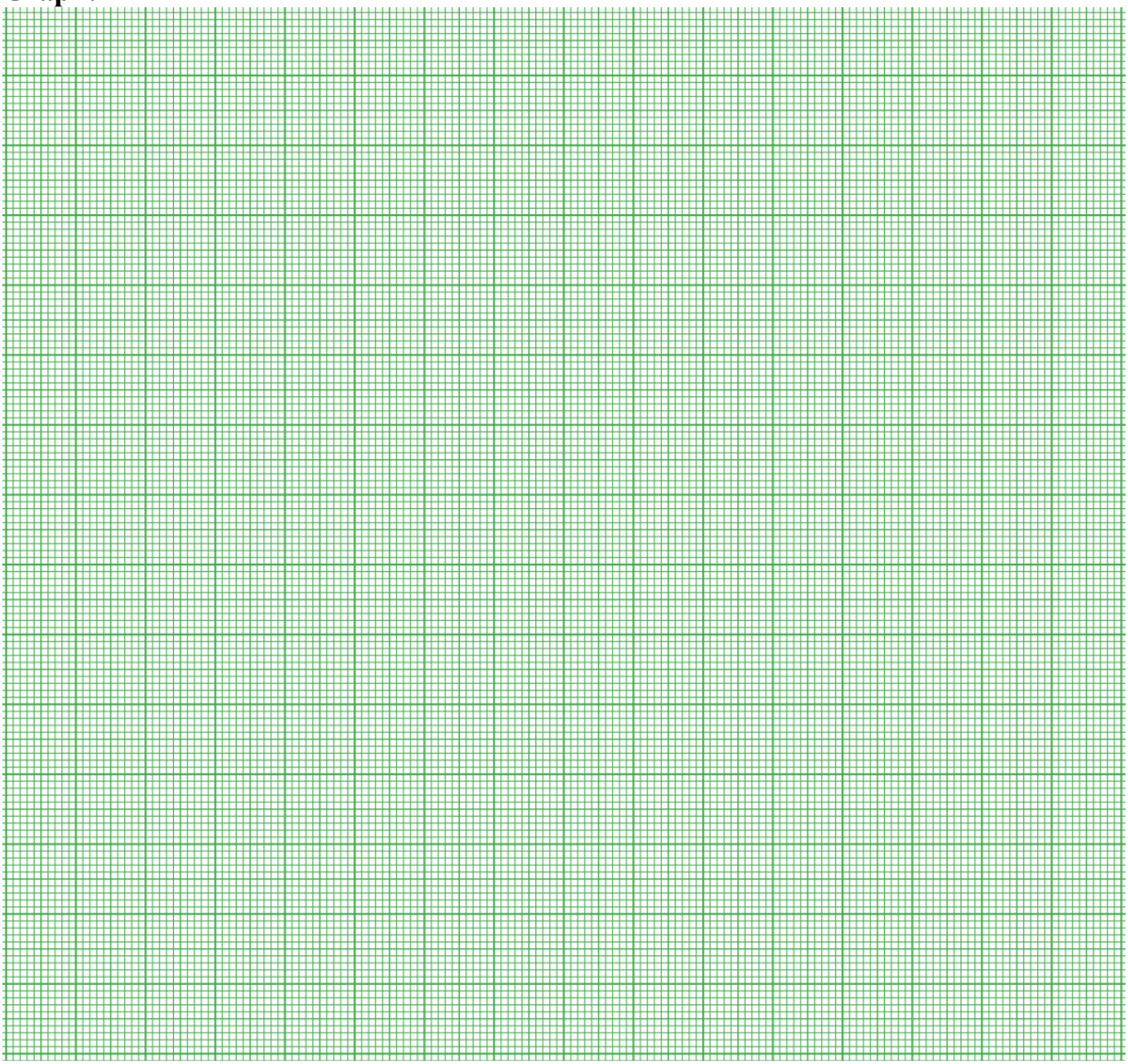
```
#include <iostream>
#include <cstdlib>
#include <chrono>
using namespace std;
using namespace std::chrono;
int partition(int arr[], int low, int high) {
int pivot = arr[high];
int i = low - 1;
for (int j = low; j < high; ++j) {
if (arr[j] <= pivot) {
++i;
swap(arr[i], arr[j]);
}
}
swap(arr[i + 1],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

Program:

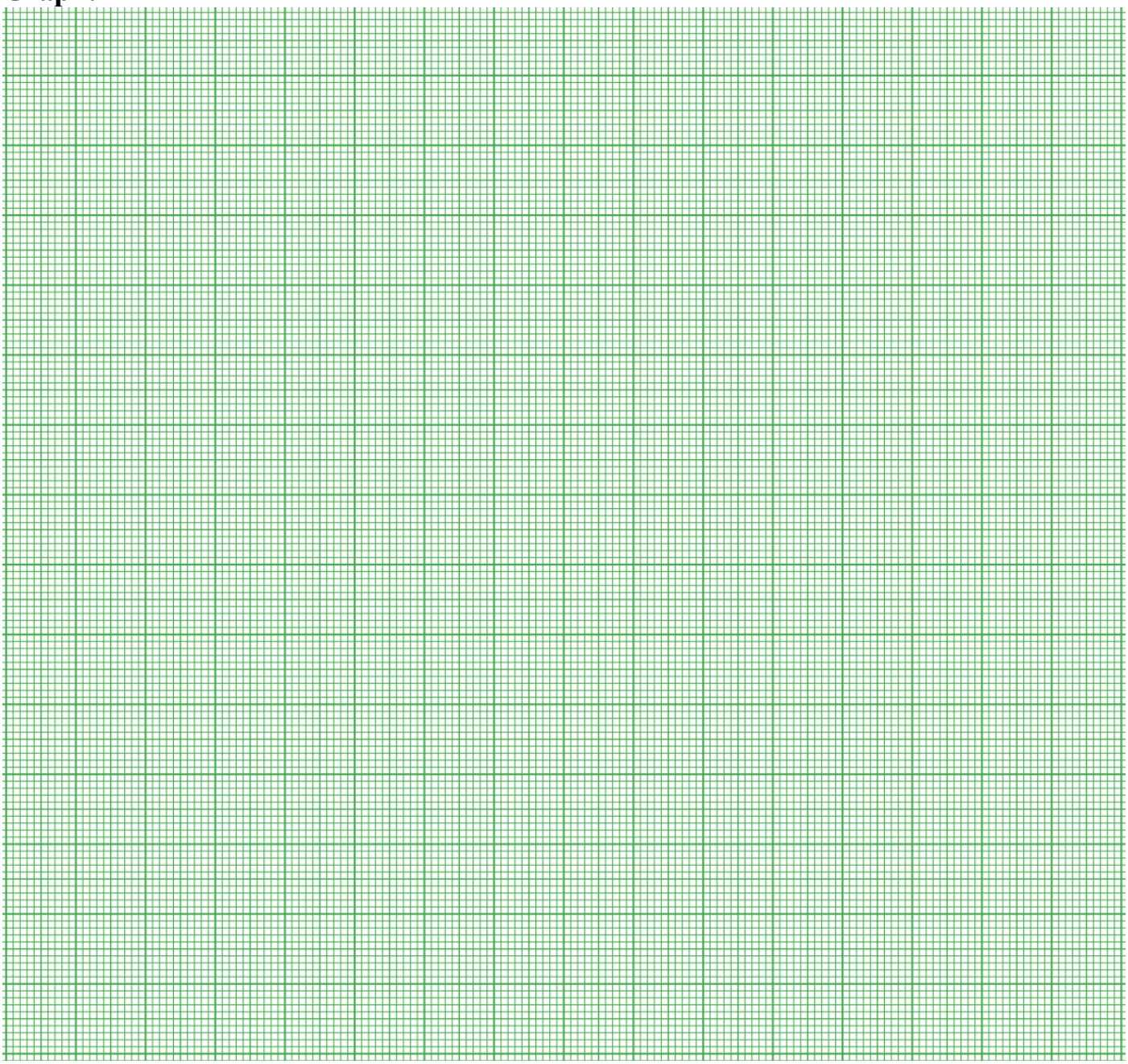
```
#include <iostream>
#include <cstdlib>
#include <chrono>
using namespace std;
using namespace std::chrono;
void merge(int arr[], int left, int mid, int right) {
int n1 = mid - left + 1;
int n2 = right - mid;
int* L = new int[n1];
int* R = new int[n2];
for (int i = 0; i < n1; ++i)
L[i] = arr[left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

Program:

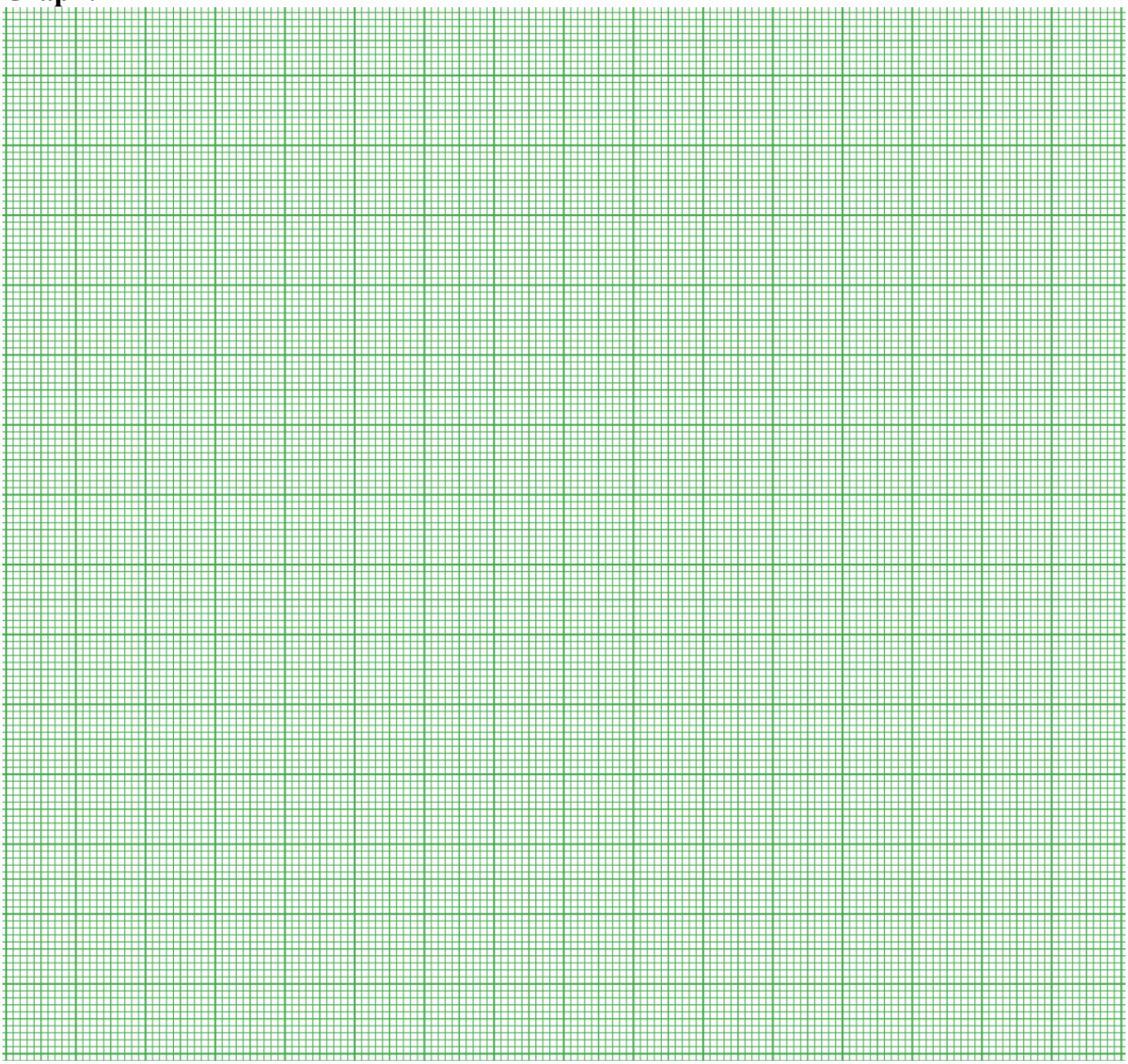
```
#include <iostream>
#include <cstdlib>
#include <chrono>
using namespace std;
using namespace std::chrono;
void heapify(int arr[], int n, int i) {
int largest = i;
int left = 2 * i + 1;
int right = 2 * i + 2;
if (left < n && arr[left] > arr[largest])
largest = left;
if (right < n && arr[right] >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

Program:

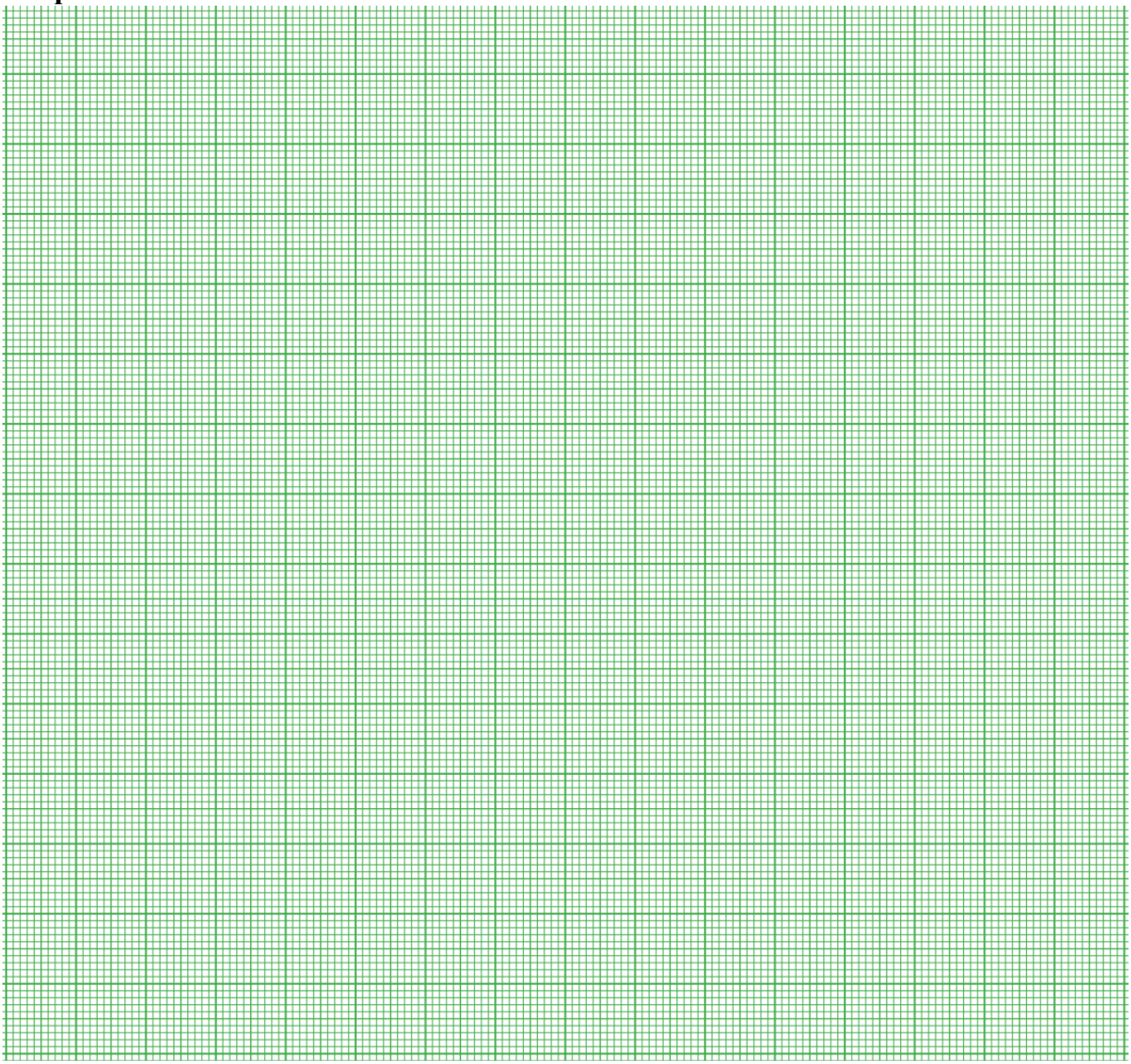
```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;
int linearSearch(int arr[], int n, int key)
{
for (int i = 0; i < n; i++)
{
if (arr[i] == key)
{
```

```
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:

Program:

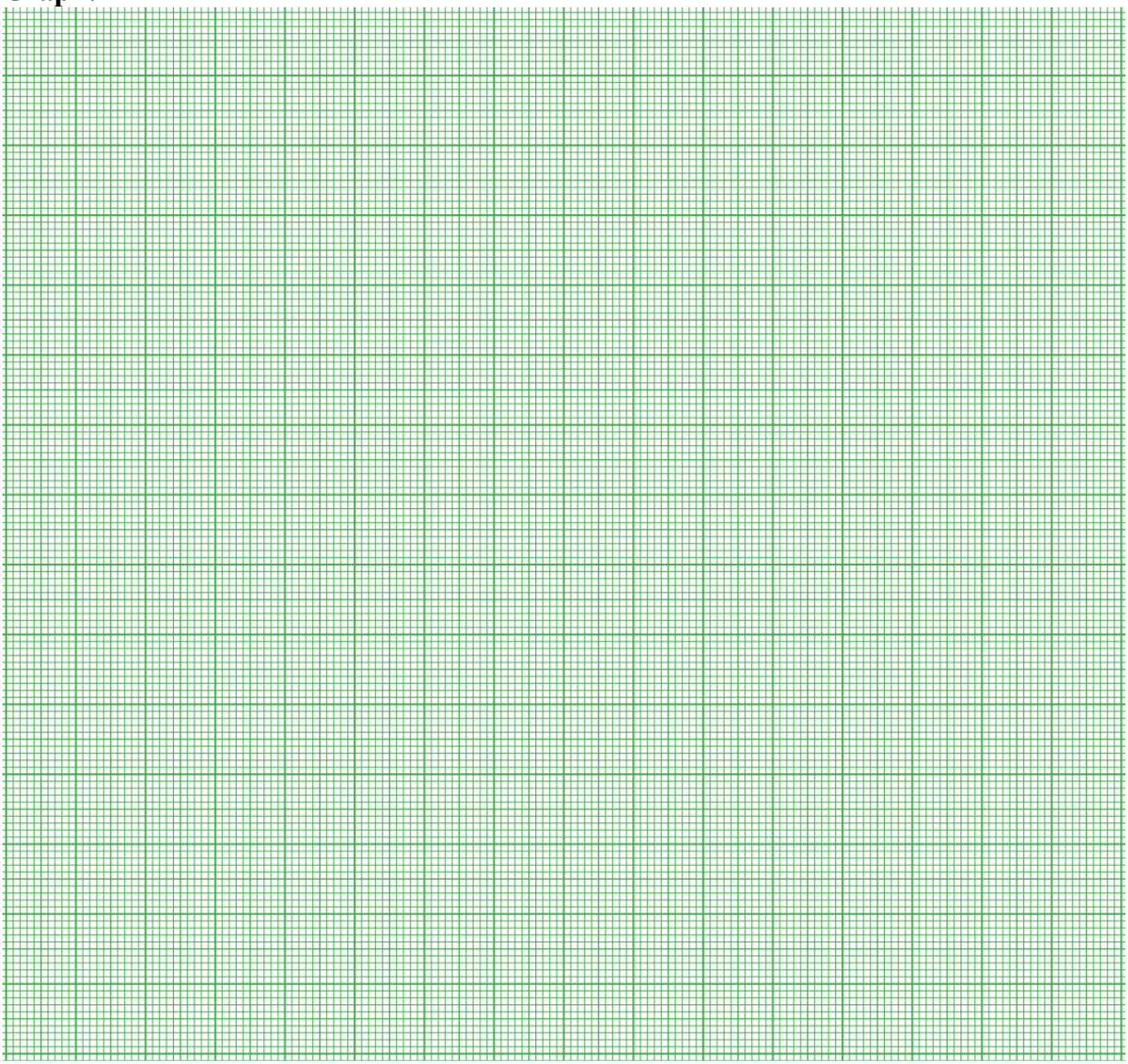
```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;
int binarySearch(int arr[], int n, int key)
{
int low = 0;
int high = n - 1;
while (low <= high)
{
int mid = (low + high) / 2;
if (arr[mid] == key)
```

```
{
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 analyse its time complexity

THEORY:

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 ===
```

Learning Outcomes:

PROGRAM 4

AIM : To implement Minimum Spanning Tree using Kruskal's Algorithm and analyse its time complexity.

THEORY:

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: 0.02 ms
```

Learning Outcome:

PROGRAM 5

AIM : To implement Matrix chain Multiplication program

THEORY:

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: 0.0210 ms
```

Learning Outcome:

PROGRAM 6

AIM : To implement Dijkstra algorithm and analyse its time complexity

THEORY:

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: 0.0520 ms
```

Learning Outcome:

PROGRAM 7

AIM : To implement Bellman Ford algorithm and analyse its time complexity.

THEORY:

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: 0.0600 ms
```

Learning Outcome:

PROGRAM 8

AIM : To implement n-Queen problem using backtracking.

THEORY:

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);
```

```c
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: 0.1500 ms
```

Learning Outcome:

PROGRAM 9

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

THEORY:

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: 0.0420 ms
```

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:

Code:

1. Naive String-Matching algorithm

```c
#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: 0.0510 ms
```

2. Rabin Karp algorithm

```
#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: 0.0420 ms
```

3. knuth Morris Pratt algorithm

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```c
#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: 0.0050 ms
```

Learning Outcome: