Name: Ankush Singh
Group: 5C13
Roll No.: 040

Date: 12-09-2023

EXPERIMENT- 1

Aim:

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

Code:

```
#include <bits/stdc++.h>
#include <ctime>

using namespace std;

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

    int leftArr[n1];
    int rightArr[n2];

    for (int i = 0; i < n1; i++) {
        leftArr[i] = arr[left + i];
    }

    for (int i = 0; i < n2; i++) {
        rightArr[i] = arr[mid + 1 + i];
    }

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

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

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

```
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
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);
    }
}
// Quick Sort
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; 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);
    }
}
// Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}
// Selection Sort
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        swap(arr[i], arr[minIndex]);
    }
}
```

Name: Ankush Singh
Group: 5C13
Roll No.: 040

Date: 12-09-2023

```
// Heap Sort
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);
    }
}
int main() {
    const int arrSize = 1000;
    int arr[arrSize], arrCopy[arrSize];

    // Fill the arrays with random values
    srand(time(NULL));
    for (int i = 0; i < arrSize; i++) {
        arr[i] = rand() % 1000;
    }

    int numTrials = 5;

    double totalMergeSortTime = 0.0;
    double totalQuickSortTime = 0.0;
    double totalBubbleSortTime = 0.0;
    double totalSelectionSortTime = 0.0;
    double totalHeapSortTime = 0.0;

    for (int trial = 0; trial < numTrials; ++trial) {
        copy(begin(arr), end(arr), begin(arrCopy));

        // Merge Sort
        clock_t start_time = clock();
        mergeSort(arrCopy, 0, arrSize - 1);
        clock_t end_time = clock();
        totalMergeSortTime += (end_time - start_time) * 1000.0 / CLOCKS_PER_SEC;

        // Quick Sort
        copy(begin(arr), end(arr), begin(arrCopy));
        start_time = clock();
        quickSort(arrCopy, 0, arrSize - 1);
        end_time = clock();
```

```
        totalQuickSortTime += (end_time - start_time) * 1000.0 / CLOCKS_PER_SEC;

        // Bubble Sort
        copy(begin(arr), end(arr), begin(arrCopy));
        start_time = clock();
        bubbleSort(arrCopy, arrSize);
        end_time = clock();
        totalBubbleSortTime += (end_time - start_time) * 1000.0 / CLOCKS_PER_SEC;

        // Selection Sort
        copy(begin(arr), end(arr), begin(arrCopy));
        start_time = clock();
        selectionSort(arrCopy, arrSize);
        end_time = clock();
        totalSelectionSortTime += (end_time - start_time) * 1000.0 / CLOCKS_PER_SEC;

        // Heap Sort
        copy(begin(arr), end(arr), begin(arrCopy));
        start_time = clock();
        heapSort(arrCopy, arrSize);
        end_time = clock();
        totalHeapSortTime += (end_time - start_time) * 1000.0 / CLOCKS_PER_SEC;
    }

    double avgMergeSortTime = totalMergeSortTime / numTrials;
    cout << "Merge Sort Execution time: " << avgMergeSortTime << " milliseconds" << endl;

    double avgQuickSortTime = totalQuickSortTime / numTrials;
    cout << "Quick Sort Execution time: " << avgQuickSortTime << " milliseconds" << endl;

    double avgBubbleSortTime = totalBubbleSortTime / numTrials;
    cout << "Bubble Sort Execution time: " << avgBubbleSortTime << " milliseconds" <<
endl;

    double avgSelectionSortTime = totalSelectionSortTime / numTrials;
    cout << "Selection Sort Execution time: " << avgSelectionSortTime << " milliseconds"
<< endl;

    double avgHeapSortTime = totalHeapSortTime / numTrials;
    cout << "Heap Sort Execution time: " << avgHeapSortTime << " milliseconds" << endl;

    return 0;
}
```

Output:

```
Merge Sort Execution time: 0.1172 milliseconds
Quick Sort Execution time: 0.0976 milliseconds
Bubble Sort Execution time: 2.688 milliseconds
Selection Sort Execution time: 0.8404 milliseconds
Heap Sort Execution time: 0.1664 milliseconds
```

Name: Ankush Singh
Group: 5C13
Roll No.: 040

Date: 19-09-2023

EXPERIMENT- 2

Aim:

To implement Linear search and Binary search and analyse its time complexity.

Code:

```
#include <bits/stdc++.h>
#include <ctime>

using namespace std;

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

int binarySearch(int arr[], int n, int target) {
    int left = 0;
    int right = n - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1;
}

void print(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl << endl;
}

int main() {
    const int arraySize = 1000;
    int arr[arraySize];

    // Fill the array with numbers from 1 to 1000
    for (int i = 0; i < arraySize; i++) {
        arr[i] = i + 1;
    }

    int target = 920;
    int numTrials = 5;
```

```
    cout << "Target value to search for: " << target << endl << endl;

    double totalLinearExecutionTime = 0.0;
    double totalBinaryExecutionTime = 0.0;

    for (int trial = 0; trial < numTrials; ++trial) {
        // Linear Search
        clock_t start_time = clock();
        int linearResult = linearSearch(arr, arraySize, target);
        clock_t end_time = clock();

        double linearExecutionTime = (end_time - start_time) * 1000.0 / CLOCKS_PER_SEC;
        totalLinearExecutionTime += linearExecutionTime;

        // Binary Search
        start_time = clock();
        int binaryResult = binarySearch(arr, arraySize, target);
        end_time = clock();

        double binaryExecutionTime = (end_time - start_time) * 1000.0 / CLOCKS_PER_SEC;
        totalBinaryExecutionTime += binaryExecutionTime;
    }

    double avgLinearExecutionTime = totalLinearExecutionTime / numTrials;
    cout << "Linear Search Execution time: " << avgLinearExecutionTime << " milliseconds"
<< endl;

    double avgBinaryExecutionTime = totalBinaryExecutionTime / numTrials;
    cout << "Binary Search Execution time: " << avgBinaryExecutionTime << " milliseconds"
<< endl;

    return 0;
}
```

Output:

```
Target value to search for: 920
Linear Search Execution time: 0.0038 milliseconds
Binary Search Execution time: 0.0002 milliseconds
```

Name: Ankush Singh
Group: 5C13
Roll No.: 040

Date: 03-10-2023

EXPERIMENT- 3

Aim:

To implement Huffman Coding and analyse its time complexity.

Code:

```
#include <bits/stdc++.h>

using namespace std;

struct HuffmanNode {
    char data;
    int frequency;
    HuffmanNode* left;
    HuffmanNode* right;

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

struct CompareNodes {
    bool operator()(HuffmanNode* a, HuffmanNode* b) {
        return a->frequency > b->frequency;
    }
};

HuffmanNode* buildHuffmanTree(map<char, int>& frequencies) {
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, CompareNodes> minHeap;

    for (auto pair : frequencies) {
        minHeap.push(new HuffmanNode(pair.first, pair.second));
    }

    while (minHeap.size() > 1) {
        HuffmanNode* left = minHeap.top();
        minHeap.pop();
        HuffmanNode* right = minHeap.top();
        minHeap.pop();

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

        minHeap.push(mergedNode);
```

Date: 03-10-2023

```
    }

    return minHeap.top();
}

void generateHuffmanCodes(HuffmanNode* root, string code, map<char, string>&
huffmanCodes) {
    if (!root)
        return;

    if (root->data != '\0') {
        huffmanCodes[root->data] = code;
    }

    generateHuffmanCodes(root->left, code + "0", huffmanCodes);
    generateHuffmanCodes(root->right, code + "1", huffmanCodes);
}

int main() {
    map<char, int> frequencies;
    int num_characters;

    cout << "Enter the number of characters: ";
    cin >> num_characters;

    for (int i = 0; i < num_characters; i++) {
        char character;
        int frequency;

        cout << "Enter character " << i + 1 << ": ";
        cin >> character;

        cout << "Enter frequency for character " << character << ": ";
        cin >> frequency;

        frequencies[character] = frequency;
    }
    clock_t start_time = clock();

    HuffmanNode* root = buildHuffmanTree(frequencies);

    map<char, string> huffmanCodes;
    generateHuffmanCodes(root, "", huffmanCodes);

    cout << "Huffman Codes:" << endl;
    for (auto pair : huffmanCodes) {
        cout << pair.first << ": " << pair.second << endl;
    }

    clock_t end_time = clock();
```

```
    double execution_time = (double)(end_time - start_time) * 1000.0 /
CLOCKS_PER_SEC;
    cout << "\nExecution time: " << execution_time << " milliseconds" << endl;

    return 0;
}
```

Output:

```
Enter the number of characters: 7
Enter character 1: a
Enter frequency for character a: 10
Enter character 2: e
Enter frequency for character e: 15
Enter character 3: i
Enter frequency for character i: 12
Enter character 4: o
Enter frequency for character o: 3
Enter character 5: u
Enter frequency for character u: 4
Enter character 6: s
Enter frequency for character s: 13
Enter character 7: t
Enter frequency for character t: 1
Huffman Codes:
a: 111
e: 10
i: 00
o: 11011
s: 01
t: 11010
u: 1100

Execution time: 0.073 milliseconds
```

Name: Ankush Singh
Group: 5C13
Roll No.: 040

EXPERIMENT- 4

Aim:

To implement Minimum Spanning Tree and analyse its time complexity.

Code:

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

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

bool compareEdges(const Edge& a, const Edge& b) {
    return a.weight < b.weight;
}

class Graph {
public:
    int V, E;
    vector<Edge> edges;
    Graph(int v, int e) {
        V = v;
        E = e;
    }
    void addEdge(int src, int dest, int weight) {
        Edge edge = {src, dest, weight};
        edges.push_back(edge);
    }
    int findParent(vector<int>& parent, int i) {
        if (parent[i] == -1)
            return i;
        return findParent(parent, parent[i]);
    }
    void kruskalMST() {
        vector<Edge> result;
        int i = 0;
        int edgeCount = 0;
        vector<int> parent(V, -1);
        int totalCost = 0;
        sort(edges.begin(), edges.end(), compareEdges);

        clock_t startTime = clock();
        while (edgeCount < V - 1 && i < E) {
            Edge nextEdge = edges[i];
            i++;
            int x = findParent(parent, nextEdge.src);
            int y = findParent(parent, nextEdge.dest);
```

```
            if (x != y) {
                result.push_back(nextEdge);
                edgeCount++;
                parent[x] = y;
                totalCost += nextEdge.weight; // Update the total cost
            }
        }
        clock_t endTime = clock();

        double executionTime = double(endTime - startTime) * 1000.0 / CLOCKS_PER_SEC;
        cout << "Edges in the Minimum Spanning Tree:" << endl;
        for (const Edge& edge : result) {
            cout << edge.src << " - " << edge.dest << " : " << edge.weight << endl;
        }
        cout << "Total Cost of Minimum Spanning Tree: " << totalCost << endl;
        cout << "Execution time: " << executionTime << " milliseconds" << endl;
    }
};

int main() {
    int V = 5;
    int E = 7;
    Graph graph(V, E);

    graph.addEdge(0, 1, 1);
    graph.addEdge(0, 2, 7);
    graph.addEdge(0, 3, 10);
    graph.addEdge(0, 4, 5);
    graph.addEdge(1, 2, 3);
    graph.addEdge(2, 3, 4);
    graph.addEdge(3, 4, 2);

    graph.kruskalMST();

    return 0;
}
```

Output:

```
Edges in the Minimum Spanning Tree:
0 - 1 : 1
3 - 4 : 2
1 - 2 : 3
2 - 3 : 4
Total Cost of Minimum Spanning Tree: 10
Execution time: 0.002 milliseconds
```

Name: Ankush Singh
Group: 5C13
Roll No.: 040

Date: 10-10-2023

EXPERIMENT- 5

Aim:

To implement Dijkstra's algorithm and analyse its time complexity.

Code:

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

#define INF INT_MAX

class Graph {
public:
    int vertices;
    vector<vector<int>> adjMatrix;

    Graph(int V) {
        vertices = V;
        adjMatrix.resize(V, vector<int>(V, INF));
    }

    void addEdge(int src, int dest, int weight) {
        adjMatrix[src][dest] = weight;
        adjMatrix[dest][src] = weight; // For undirected graph
    }

    int minDistance(vector<int>& dist, vector<bool>& sptSet) {
        int minDist = INF, minIndex;

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

    void dijkstra(int src) {
        vector<int> dist(vertices, INF);
        vector<bool> sptSet(vertices, false);

        dist[src] = 0;

        for (int count = 0; count < vertices - 1; ++count) {
            int u = minDistance(dist, sptSet);
            sptSet[u] = true;
```

```
            for (int v = 0; v < vertices; ++v) {
                if (!sptSet[v] && adjMatrix[u][v] != INF &&
                    dist[u] != INF && dist[u] + adjMatrix[u][v] < dist[v]) {
                    dist[v] = dist[u] + adjMatrix[u][v];
                }
            }
        }
        cout << "Vertex   Distance from Source\n";
        for (int i = 0; i < vertices; ++i) {
            cout << i << "\t\t\t\t" << dist[i] << endl;
        }
    }
};
int main() {
    int vertices = 6;

    Graph graph(vertices);

    graph.addEdge(0, 1, 5);
    graph.addEdge(0, 2, 2);
    graph.addEdge(1, 3, 4);
    graph.addEdge(2, 4, 7);
    graph.addEdge(3, 5, 3);
    graph.addEdge(4, 5, 1);

    int source = 0;
    cout<<"Source: "<<source<<endl;
    clock_t start_time = clock();

    graph.dijkstra(source);

    clock_t end_time = clock();
    double execution_time = static_cast<double>(end_time - start_time)*1000.0 / CLOCKS_PER_SEC;

    cout << "Execution Time: " << execution_time << " milliseconds" << endl;

    return 0;
}
```

Output:

```
Source: 0
Vertex   Distance from Source
0                0
1                5
2                2
3                9
4                9
5                10
Execution Time: 0.046 milliseconds
```

Name: Ankush Singh
Group: 5C13
Roll No.: 040

Date: 17-10-2023

EXPERIMENT- 6

Aim:

Write a program to implement N-Queen Problem using backtracking and analyze its time complexity.

Code:

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

using namespace std;

bool isSafe(int row, int col, const vector<int>& placement) {
    for (int i = 0; i < row; ++i) {
        if (placement[i] == col || abs(i - row) == abs(placement[i] - col)) {
            return false;
        }
    }
    return true;
}

void printBoard(const vector<int>& placement) {
    int N = placement.size();
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            if (placement[i] == j) {
                cout << "Q" << "\t";
            } else {
                cout << "." << "\t";
            }
        }
        cout << endl;
    }
    cout << endl;
}

bool solveNQueens(int row, int N, vector<int>& placement) {
    if (row == N) {
        printBoard(placement);
        return true;
    }

    for (int col = 0; col < N; ++col) {
        if (isSafe(row, col, placement)) {
            placement[row] = col;
            if (solveNQueens(row + 1, N, placement)) {
                return true; // Stop after finding the first solution
            }
```

```
        }
    }

    return false;
}

int main() {
    int N;
    cout << "Enter the size of the chessboard (N): ";
    cin >> N;

    vector<int> placement(N, -1); // Initializing with -1, indicating no queen placed yet

    clock_t start = clock();

    if (!solveNQueens(0, N, placement)) {
        cout << "No solution found." << endl;
    }

    clock_t end = clock();
    double executionTime = double(end - start) * 1000.0/ CLOCKS_PER_SEC;

    cout << "Execution time: " << executionTime << " milliseconds" << endl;

    return 0;
}
```

Output:

```
Enter the size of the chessboard (N): 8
Q   .   .   .   .   .   .   .
.   .   .   .   Q   .   .   .
.   .   .   .   .   .   .   Q
.   .   .   .   .   Q   .   .
.   .   Q   .   .   .   .   .
.   .   .   .   .   .   Q   .
.   Q   .   .   .   .   .   .
.   .   .   Q   .   .   .   .


Execution time: 0.136 milliseconds
```

Name: Ankush Singh
Group: 5C13
Roll No.: 040

Date: 28-11-2023

EXPERIMENT- 7

Aim:

To implement Matrix Multiplication and analyse its time complexity.

Code:

```
#include <iostream>
#include <ctime>

using namespace std;

int main() {
    int m, n, p, q, i, j, k;
    cout << "Enter the number of rows and columns of the first matrix: ";
    cin >> m >> n;
    cout << "Enter the number of rows and columns of the second matrix: ";
    cin >> p >> q;

    if (n != p) {
        cout << "The matrices can't be multiplied with each other.";
        return 0;
    }

    int first[m][n], second[p][q], multiply[m][q];

    cout << endl << "Enter the elements of the first matrix:" << endl;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            cin >> first[i][j];
        }
    }

    cout << endl << "Enter the elements of the second matrix:" << endl;
    for (i = 0; i < p; i++) {
        for (j = 0; j < q; j++) {
            cin >> second[i][j];
        }
    }

clock_t start = clock();
    for (i = 0; i < m; i++) {
        for (j = 0; j < q; j++) {
            multiply[i][j] = 0;
            for (k = 0; k < p; k++) {
                multiply[i][j] += first[i][k] * second[k][j];
            }
        }
    }
```

```
clock_t end = clock();
    double executionTime = double(end - start) * 1000.0 / CLOCKS_PER_SEC;


    cout << endl << "Product of the matrices:" << endl;
    for (i = 0; i < m; i++) {
        for (j = 0; j < q; j++) {
            cout << multiply[i][j] << "\t";
        }
        cout << endl;
    }
    cout << "\nExecution time: " << executionTime << " milliseconds" << endl;

    return 0;
}
```

Output:

```
Enter the number of rows and columns of the first matrix: 2 3
Enter the number of rows and columns of the second matrix: 3 2
Enter the elements of the first matrix:
1 2 5
3 6 5
Enter the elements of the second matrix:
2 2
4 7
8 9
Product of the matrices:
50  61
70  93

Execution time: 0.001 milliseconds
```