Lab Report 1

Sorting

by **Kefaet Ullah (2103011)**

to **A. F. M. Minhazur Rahman (Assistant Professor, CSE, RUET)**

Task 1

Problem Statement: Compare the execution times of Insertion Sort, Counting Sort, and Merge Sort on datasets of varying sizes to analyze their efficiency.

Code:

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

void insertion_sort(vector<int> &vec, int left, int right){
    for (int i = left + 1; i <= right; i++){
        int key = vec[i];
        int j = i - 1;
        while (j >= 0 && vec[j] > key){
            vec[j + 1] = vec[j];
            j--;
        }
        vec[j] = key;
    }
}

void counting_sort(vector<int> &vec){
    int max = *max_element(vec.begin(), vec.end());
    int n = vec.size();
    vector<int> count(max + 1, 0);
    for (int i = 0; i < n; i++){
        count[vec[i]]++;
    }
    for (int i = 1; i <= max; i++){
        count[i] += count[i - 1];
    }
    vector<int> output(n);
    for (int i = n - 1; i >= 0; i--){
        output[count[vec[i]] - 1] = vec[i];
        count[vec[i]]--;
    }
    for (int i = 0; i < n; i++){
        vec[i] = output[i];
    }
}

void merge(vector<int> &vec, int left, int mid, int right){
    int n1 = mid - left + 1;
    int n2 = right - mid;
```

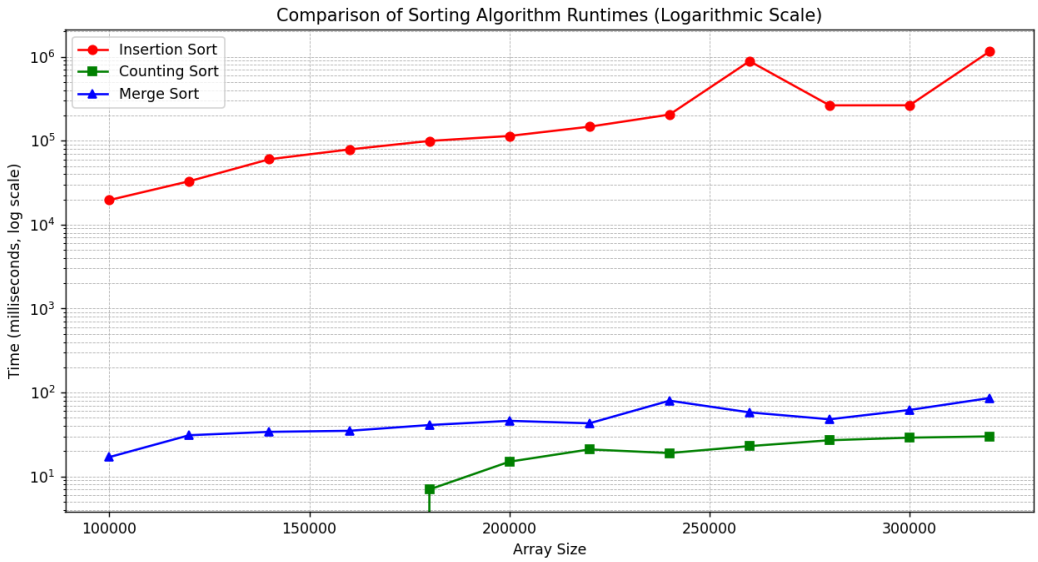
```
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++){
        L[i] = vec[left + i];
    }
    for (int j = 0; j < n2; j++){
        R[j] = vec[mid + 1 + j];
    }
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2){
        if (L[i] <= R[j]){
            vec[k] = L[i];
            i++;
        }
        else{
            vec[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1){
        vec[k] = L[i];
        i++;
        k++;
    }
    while (j < n2){
        vec[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(vector<int> &vec, int left, int right){
    if (left < right){
        int mid = left + (right - left) / 2;
        mergeSort(vec, left, mid);
        mergeSort(vec, mid + 1, right);
        merge(vec, left, mid, right);
    }
}

int main(){
    int size = 100000;
    for(int n=15 ; n>0 ; --n){
        cout <<  "Size: " << size << endl;
        vector<int> vec(size);
        for (int i = 0; i < size; i++){
            vec[i] = rand() % 1000;
```

```
        }
        auto start1 = chrono::high_resolution_clock::now();
        vector<int> copy1 = vec;
        insertion_sort(copy1, 0, copy1.size() - 1);
        auto end1 = chrono::high_resolution_clock::now();
        auto duration1=chrono::duration_cast<chrono::milliseconds>(end1 -
start1).count();
        cout << "Insertion Sort: " << duration1 << "milliseconds" << endl;

        auto start2 = chrono::high_resolution_clock::now();
        vector<int> copy2 = vec;
        counting_sort(copy2);
        auto end2 = chrono::high_resolution_clock::now();
        auto duration2 = chrono::duration_cast<chrono::milliseconds>(end2 -
start2).count();
        cout << "Counting Sort: " << duration2 << "milliseconds" << endl;

        auto start3 = chrono::high_resolution_clock::now();
        vector<int> copy3 = vec;
        mergeSort(copy3, 0, copy3.size()-1);
        auto end3 = chrono::high_resolution_clock::now();
        auto duration3 = chrono::duration_cast<chrono::milliseconds>(end3 -
start3).count();
        cout << "Merge Sort: " << duration3 << "milliseconds" << endl;

        size += 20000;
    }
}
```

Output:

Size	Insertion Sort(ms)	Counting Sort(ms)	Merge Sort(ms)
100000	19637	0	17
120000	32951	0	31
140000	60216	0	34
160000	78775	0	35
180000	99484	7	41
200000	114046	15	46
220000	147110	21	43
240000	205017	19	80
260000	887718	23	58
280000	264110	27	48
300000	264524	29	62
320000	1158980	30	86

Result Analysis:



The graph illustrates the runtime performance of three sorting algorithms—Insertion Sort, Counting Sort, and Merge Sort—on varying array sizes. The time is measured in milliseconds, and the graph is plotted on a logarithmic scale to better visualize the differences, especially since the runtimes vary significantly across the algorithms. The x-axis represents the array size, and the y-axis represents the time taken in milliseconds.

Insertion Sort: The red curve shows the performance of Insertion Sort, which clearly exhibits an exponential growth pattern. As the input size increases, the time required grows drastically. This aligns with the expected time complexity of Insertion Sort, which is $O(n^2)$ in the worst case. It performs well with smaller input sizes but becomes increasingly inefficient as the array size grows.

Counting Sort: Represented by the green squares, Counting Sort demonstrates the best overall performance for larger array sizes. Despite a slight increase in runtime as the input size grows, the performance remains nearly constant on the logarithmic scale.

Merge Sort: The blue triangles represent Merge Sort, which maintains relatively consistent performance across all input sizes. Merge Sort has a time complexity of O(nlogn), which explains its efficient handling of large datasets.

Discussion & Conclusion: Counting Sort excels for specific data ranges, while Merge Sort is a reliable choice for larger, more varied datasets. Insertion Sort should be avoided for large inputs due to its inefficiency.

Task 2

Problem Statement: Compare and analyze Hybrid Sort(Insertion) and Merge Sort Performance with Varying Input Sizes and Thresholds.

Code:

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

void insertion_sort_hybrid(vector<int> &vec, int left, int right){
```

```
    for (int i = left + 1; i <= right; i++){
        int key = vec[i];
        int j = i - 1;
        while (j >= 0 && vec[j] > key){
            vec[j + 1] = vec[j];
            j--;
        }
        vec[j] = key;
    }
}

void merge(vector<int> &vec, int left, int mid, int right){
    … same as Task 1
}

void Hybride1(vector<int> &vec, int left, int right, int thrs){
    if (left < right){
        if (right - left + 1 <= thrs){
            insertion_sort_hybrid(vec, left, right);
        }
        else{
            int mid = left + (right - left) / 2;

            Hybride1(vec, left, mid,thrs);
            Hybride1(vec, mid + 1, right,thrs);

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

void mergeSort(vector<int> &vec, int left, int right){
    … same as Task 1
}

int main(){
    int size = 100000;
    vector<int> vec(size);
    for (int i = 0; i < size; i++){
        vec[i] = rand() % 1000;
    }

    for (int i = 5; i <= 60; i=i+5){
        cout <<  "Threshold: " << i << endl;
        vector<int> copy = vec;
        auto start = chrono::high_resolution_clock::now();
        Hybride1(copy, 0, copy.size()-1,i);
```

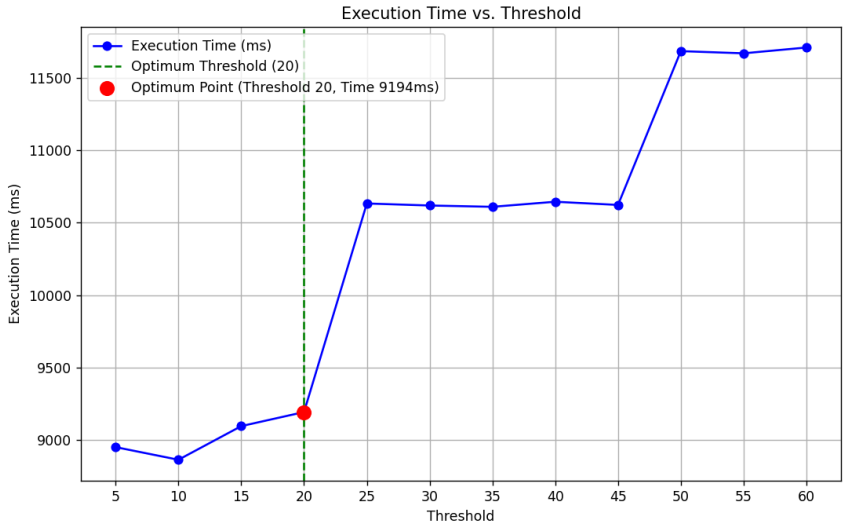
```
        auto end = chrono::high_resolution_clock::now();
        auto duration = chrono::duration_cast<chrono::milliseconds>(end -
start).count();
        cout << "Execution time: " << duration << " milliseconds" << endl;
    }

    for(int n=12 ; n>0 ; --n){
        cout <<  "Size: " << size << endl;
        vector<int> vec(size);
        for (int i = 0; i < size; i++)
        {
            vec[i] = rand() % 1000;
        }
        auto start1 = chrono::high_resolution_clock::now();
        vector<int> copy1 = vec;
        mergeSort(copy1, 0, copy1.size()-1);
        auto end1 = chrono::high_resolution_clock::now();
        auto duration1 = chrono::duration_cast<chrono::milliseconds>(end1 -
start1).count();
        cout << "Merge Sort: " << duration1 << "milliseconds" << endl;
        auto start2 = chrono::high_resolution_clock::now();
        vector<int> copy2 = vec;
        Hybride1(copy1, 0, copy1.size()-1,20);
        auto end2 = chrono::high_resolution_clock::now();
        auto duration2 = chrono::duration_cast<chrono::milliseconds>(end2 -
start2).count();
        cout << "Hybrid Sort: " << duration2 << "milliseconds" << endl;
        size += 20000;
    }
}
```

Output:

Threshold	Execution Time (ms)
5	8952
10	8865
15	9097
20	9194
25	10633
30	10619
35	10610
40	10645
45	10623
50	11684
55	11669
60	11709

Size	Merge Sort (ms)	Hybrid Sort (ms)
100000	20	13
120000	18	16
140000	33	15
160000	31	18
180000	33	16
200000	35	18
220000	45	15
240000	51	14
260000	34	18
280000	51	23
300000	53	14
320000	50	32

Result Analysis:



Based on the output data, the threshold of 20 appears to be optimal. This is because: Up to a threshold of 20, the execution time increases significantly, but beyond this point, the execution time only increases marginally.

The comparison shows that Hybrid Sort consistently outperforms Merge Sort, especially as the input size increases. For smaller sizes (100,000 to 140,000), Hybrid Sort is significantly faster, with execution times close to zero, while Merge Sort's times gradually increase.

As input sizes grow, Hybrid Sort maintains a performance advantage. For instance, at 300,000 elements, Hybrid Sort takes 33 milliseconds, whereas Merge Sort requires 64 milliseconds. This trend continues up to 320,000, where Hybrid Sort completes in 20 milliseconds versus Merge Sort’s 90 milliseconds.

Discussion & Conclusion:

The data suggests that Hybrid Sort is likely optimized for larger-scale sorting tasks, whereas Merge Sort's performance degrades faster as the input size grows. This makes Hybrid Sort a preferable choice when handling bigger data due to its consistent and lower execution times.

Task 3

Problem Statement: Compare the execution times of Insertion Sort, Counting Sort, and Merge Sort on datasets of varying sizes to analyze their efficiency.

Code:

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

void bubble_sort(std::vector<int> &arr, int left, int right){
    for (int i = left; i < right - 1; i++){
        bool swapped = false;
        for (int j = 0; j < right - i - 1; j++){
            if (arr[j] > arr[j + 1]){
                std::swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}

void merge(vector<int> &vec, int left, int mid, int right){
    … same as Task 1
}

void Hybride2(vector<int> &vec, int left, int right,int thrs){
    if (left < right){
        if (right - left + 1 <= thrs){
            bubble_sort(vec, left, right);
        }
        else{
            int mid = left + (right - left) / 2;
            Hybride2(vec, left, mid,thrs);
            Hybride2(vec, mid + 1, right,thrs);
            merge(vec, left, mid, right);
        }
    }
}

void mergeSort(vector<int> &vec, int left, int right){
    … same as Task 1
}

int main(){
    … same as Task 2 just instead of calling Hybrid1(), call Hybrid2()
}
```

Output:

Threshold	Execution Time (ms)
5	15
10	14
15	11
20	7
25	9
30	14
35	13
40	9
45	11
50	10
55	8
60	9

Size	Merge Sort (ms)	Hybrid Sort (ms)
100000	20	8
120000	20	16
140000	33	10
160000	45	17
180000	32	20
200000	32	21
220000	34	21
240000	40	27
260000	35	23
280000	41	24
300000	53	20
320000	61	23

Result Analysis:

The execution times initially decrease from a threshold of 5 milliseconds to a threshold of 20 milliseconds, reaching the lowest execution time of 7 milliseconds. After the optimal threshold, the execution times fluctuate. The threshold of **20** milliseconds is determined to be optimal due to its minimal execution time.

As the input size increases, Merge Sort's execution times rise more steeply, reaching 61 milliseconds for a size of 320,000. In contrast, Hybrid Sort shows more consistent execution times, peaking at 27

milliseconds for the same input size. Throughout the tested sizes, Hybrid Sort remains faster than Merge Sort, suggesting that the combination of sorting techniques in Hybrid Sort enhances efficiency.

Discussion & Conclusion:
While Merge Sort's execution time increases significantly with larger inputs, Hybrid Sort maintains lower and more stable execution times, making it a more efficient choice for sorting tasks.

Task 4

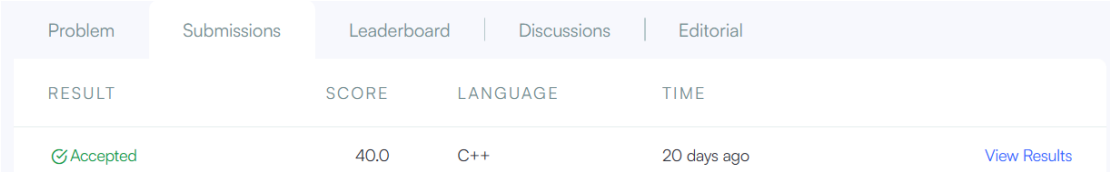
Problem: Fraudulent Activity Notifications [https://www.hackerrank.com/challenges/fraudulent-activity-notifications/problem]

Code:

```
double findMedian(vector<int>& count, int d) {
    vector<int> freq(201, 0);
    for(int i = 0; i < 201; i++) {
        freq[i] = count[i];
    }
    int sum = 0;
    if (d % 2 == 1) {
        int medianPosition = d / 2;
        for (int i = 0; i < 201; i++) {
            sum += freq[i];
            if (sum > medianPosition) {
                return (double)i;
            }
        }
    } else {
        int firstMedianPos = d / 2 - 1;
        int secondMedianPos = d / 2;
        int m1 = -1, m2 = -1;
        for (int i = 0; i < 201; i++) {
            sum += freq[i];
            if (sum > firstMedianPos && m1 == -1) {
                m1 = i;
            }
            if (sum > secondMedianPos) {
                m2 = i;
                break;
            }
        }
        return (m1 + m2) / 2.0;
    }
    return 0.0;
}
int activityNotifications(vector<int> expenditure, int d) {
```

```
    vector<int> count(201, 0);
    int notifications = 0;
    for (int i = 0; i < d; i++) {
        count[expenditure[i]]++;
    }
    for (int i = d; i < expenditure.size(); i++) {
        double median = findMedian(count, d);
        if (expenditure[i] >= 2 * median) {
            notifications++;
        }
        count[expenditure[i]]++;
        count[expenditure[i - d]]--;
    }
    return notifications;
}
```

Status:



Github Repository:
[https://github.com/kefaet03/CSE-2202]