

Index

1. Sorting	2 - 12
i. Task 1: Compare the execution times of Insertion Sort, Counting Sort, and Merge Sort on datasets of varying sizes to analyze their efficiency.	3 - 6
ii. Task 2: Compare and analyze Hybrid Sort(Insertion) and Merge Sort Performance with Varying Input Sizes and Thresholds.	6 - 9
iii. Task 3: Compare and analyze Hybrid Sort(Bubble) and Merge Sort Performance with Varying Input Sizes and Thresholds.	9 - 11
iv. Task 4: Fraudulent Activity Notifications.	11 - 12
2. Convex Hull	13 - 15
a) Task 1: Implement and Compare the Quickhull and Graham Scan Algorithms to Compute the Convex Hull of 2D Points, Visualize the Results, and Analyze Execution Times on Randomly Generated Datasets of Varying Sizes.	13 - 15
3. Fractional Knapsack	16 - 18
a) Task 1: Given a set of items, each with a weight and a value, determine the maximum total value that can be achieved by selecting items to include in a knapsack of fixed capacity. (Fraction allowed)	16 - 18
4. Merge and Quick	19 - 23
a) Task 1: Compare the number of swaps required during the sorting process between Merge Sort and Quick Sort.	19 - 23

List of Tables

1.1: Analysis of Sorting algorithms by various input size	05
1.2: Finding Optimum Threshold for Hybrid(insertion) Sort	08
1.3: Comparison between Merge & Hybrid(insertion) Sort	08
1.4: Finding Optimum Threshold for Hybrid(bubble) Sort	10
1.5: Comparison between Merge & Hybrid(bubble) Sort10	10
2.1: Execution Time Comparison Between Quick Hull and Graham Scan Algorithms	14

List of Figures

1.1: Visualization of Input Size Impact on Sorting Algorithms	06
1.2: Determining the Optimum Threshold for Hybrid (Insertion) Sort	09
1.3: Performance Comparison: Merge Sort vs. Hybrid (Insertion) Sort	09
1.4: Determining the Optimum Threshold for Hybrid(Bubble) Sort	11
1.5: Performance Comparison: Merge Sort vs. Hybrid(Bubble) Sort	11
2.1: Scatter Plot of Randomly Generated Points	14
2.2: Convex Hull of Random Points	14
2.3: Performance Analysis: Quick Hull vs. Graham Scan	15

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

Table 1.1: Analysis of Sorting algorithms by various input size

Result Analysis: The figure (Figure 1.1) 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 figure 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.

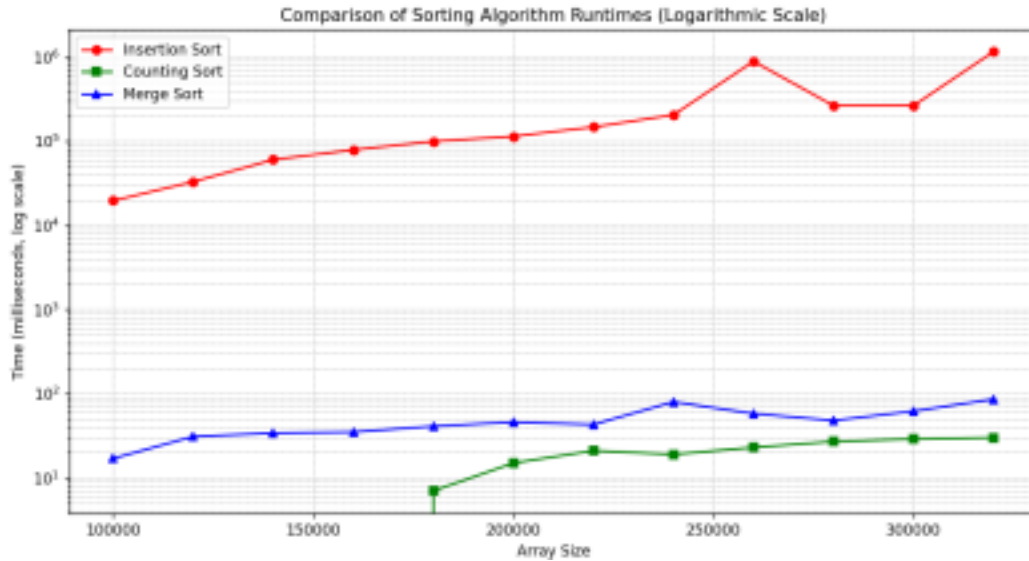


Figure 1.1: Visualization of Input Size Impact on Sorting Algorithms

The figure 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 figure 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 $O(n \log n)$, 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
15	9097
20	9194
25	10633
30	10619
35	10610
40	10645
45	10623
50	11684
55	11669
60	11709

Table 1.2: Finding Optimum Threshold for Hybrid(insertion) Sort.

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

Table 1.3: Comparison between Merge & Hybrid(insertion) Sort.

Result Analysis: From Figure 1.2, 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.

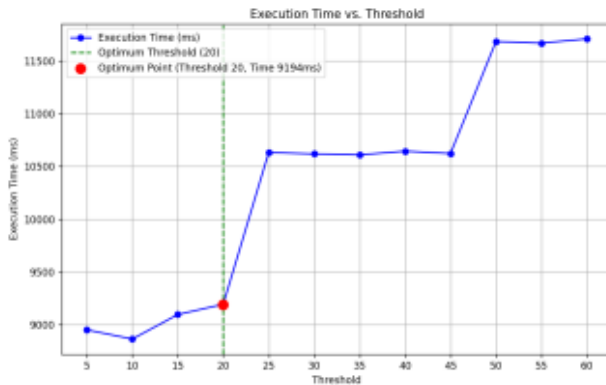


Figure 1.2: Determining the Optimum Threshold for Hybrid (Insertion) Sort

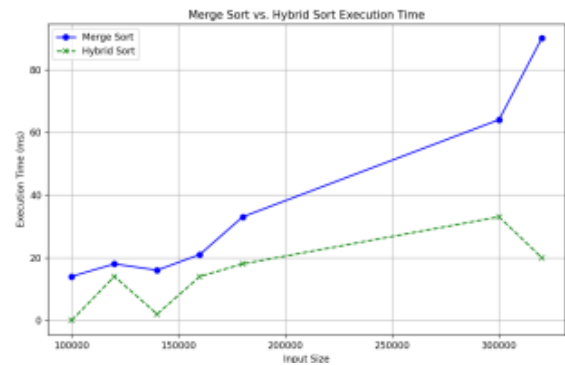


Figure 1.3: Performance Comparison: Merge Sort vs. Hybrid (Insertion) Sort

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 and analyze Hybrid Sort(Bubble) and Merge Sort Performance with Varying Input Sizes and Thresholds.

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

Table 1.4: Finding Optimum Threshold for Hybrid(insertion) Sort.

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

Table 1.5: Comparison between Merge & Hybrid(insertion) Sort.

Result Analysis:

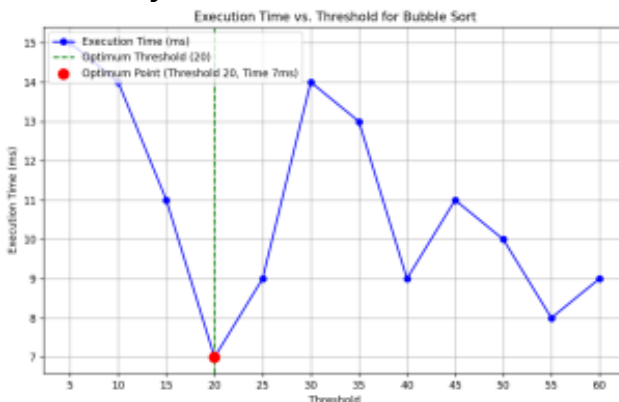


Figure 1.4: Determining the Optimum Threshold for Hybrid (Bubble) Sort

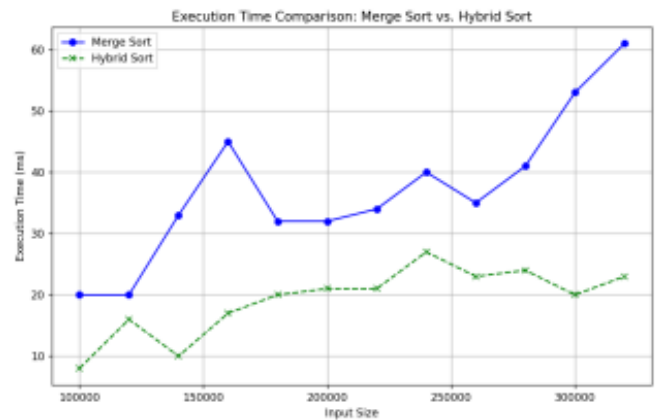


Figure 1.5: Performance Comparison: Merge Sort vs. Hybrid (Insertion) Sort

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:

Problem	Submissions	Leaderboard	Discussions	Editorial
RESULT		SCORE	LANGUAGE	TIME
 Accepted		40.0	C++	20 days ago

[View Results](#)

Lab Report 2

Convex Hull

by Kefaet Ullah (2103011)

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

Task 1

Problem Statement: Implement and compare the Quickhull and Graham Scan algorithms to compute the convex hull of 2D points, visualize the results, and analyze execution times on randomly generated datasets of varying sizes.

Code: (Google Colab Code)

Quick Hull:



Graham Scan:



Output:

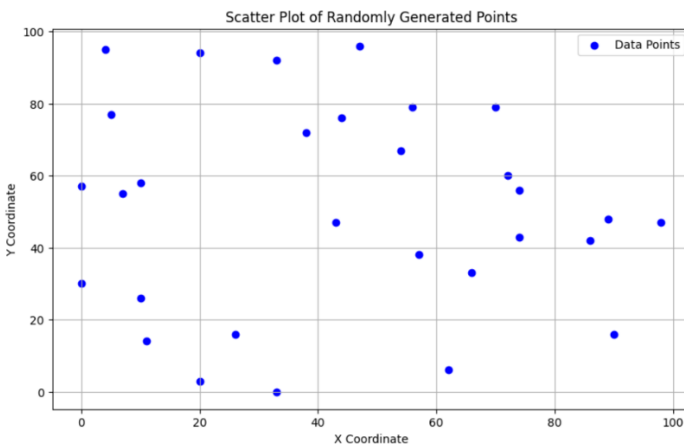


Figure 2.1: Scatter Plot of Randomly Generated Points

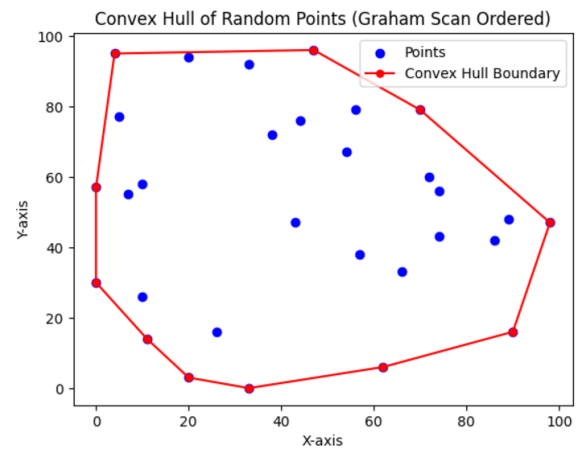


Figure 2.2: Convex Hull of Random Points

Comparison:

Size	Quick Hull (s)	Graham Scan (s)
50	0.002534	0.000347
100	0.005983	0.000686
250	0.011022	0.000934
500	0.034236	0.002739
1000	0.048835	0.004252
1500	0.113808	0.006608
2500	0.138793	0.014129
5000	0.182765	0.018683

Table 2.1: Execution Time Comparison Between Quick Hull and Graham Scan Algorithms.

Result Analysis: Overview of the Algorithms:

- Quick Hull Algorithm: Quick Hull is a divide-and-conquer algorithm that computes the convex hull by recursively finding the farthest points from a dividing line and constructing hulls for subproblems. It operates similarly to QuickSort in partitioning the problem.
- Graham Scan Algorithm: Graham Scan uses sorting and a stack-based approach. It begins by selecting the point with the lowest y-coordinate (and lowest x in case of ties), sorts the remaining points by polar angle, and constructs the convex hull by traversing the sorted list while maintaining the convexity condition.

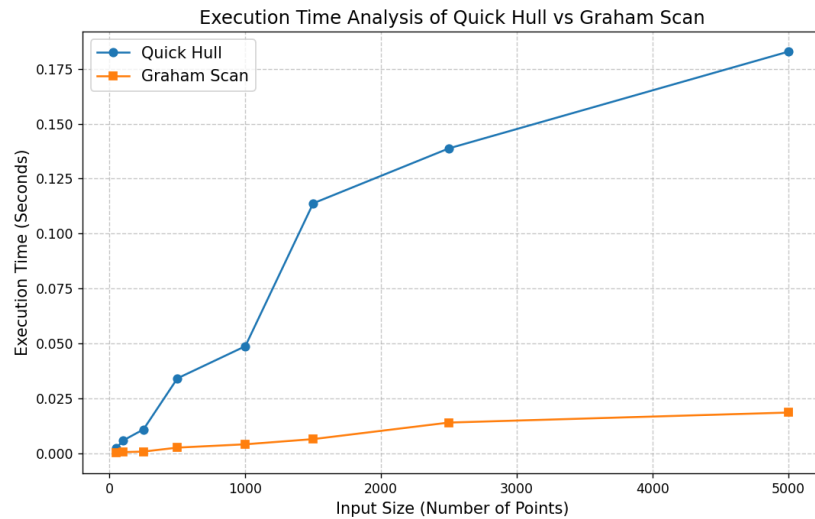


Figure 2.3: Performance Analysis: Quick Hull vs. Graham Scan

Observation from the Figure: The execution time of Quick Hull algorithm increases more sharply with the input size compared to Graham Scan, especially for larger datasets. Graham Scan algorithm demonstrates consistently better performance due to its more predictable and efficient operations, particularly for large datasets.

Theoretical Analysis:

- Quick Hull:

Steps: a) Finding the farthest point from a line: $O(n)$. b) Dividing the points into two subsets relative to this line: $O(n)$. c) Recursively solving each subset.

Recurrence Relation: Let $T(n)$ be the time complexity for n points. $T(n) = 2T(n/2) + O(n)$
This is similar to QuickSort's recurrence.

Using the Master Theorem: $T(n) = O(n \log n)$ in the average case.

In worst Case, $T(n) = T(n-1) + O(n)$

Solving this: $T(n) = O(n^2)$

- Graham Scan:

Steps: a) Finding the starting point (lowest y-coordinate): $O(n)$. b) Sorting all points by their polar angle relative to the starting point: $O(n \log n)$. c) Constructing the convex hull by traversing the sorted points and using a stack: $O(n)$.

The sorting step dominates, so the overall time complexity is $O(n \log n)$.

$$\text{Ratio} = \frac{\text{Quick Hull Worst Case}}{\text{Graham Scan}} = \frac{n^2}{n \log n} = O\left(\frac{n}{\log n}\right)$$

This shows that as n increases, Quick Hull's worst-case time grows significantly faster than Graham Scan.

Discussion & Conclusion: Graham Scan has a predictable time complexity of $O(n \log n)$, making it robust for practical use. Quick Hull's divide-and-conquer approach is efficient on average, but its $O(n^2)$ worst-case complexity makes it unsuitable for large datasets with unfavorable distributions. For real-world scenarios with large and complex datasets, Graham Scan is the preferred choice due to its consistent performance. Quick Hull can still be useful for smaller datasets or cases where input distribution favors divide-and-conquer strategies.

Lab Report 3

Fractional Knapsack

by Kefaet Ullah (2103011)

to Dr. Md. Ali Hossain(Professor, CSE, RUET)

Task 1

Problem Statement: Given a set of items, each with a weight and a value, determine the maximum total value that can be achieved by selecting items to include in a knapsack of fixed capacity. (Fraction allowed)

Introduction: The Fractional Knapsack Problem is a classic optimization problem where the goal is to maximize the total value of items that can be placed in a knapsack with a fixed weight capacity. Unlike the 0/1 Knapsack Problem, fractional knapsack allows taking fractions of an item, enabling a greedy approach. Items are selected based on their value-to-weight ratio to achieve the highest total value within the given constraints.

Algorithm:

Step 1: Read input values.

- Initialize an array `items[]` where each element is a pair (weight, value).
- Calculate the value-to-weight ratio for each item.

Step 2: Sort `items[]` in descending order of value-to-weight ratio.

Step 3: Initialize `total_weight = 0` and `total_benefit = 0`.

Step 4: Iterate through the sorted items:

For each item `i`:

If `total_weight < k`:

- Calculate the allowable weight to add: `temp = min(k - total_weight, weight[i])`.
- Add the corresponding benefit to `total_benefit`: `total_benefit += temp * (value[i] / weight[i])`.
- Update `total_weight += temp`.

Else:

Break the loop.

Step 5: Print `total_benefit` as the result.

Code:

```
#include <bits/stdc++.h>
using namespace std;
int main(){
    freopen("knapsack.txt", "r", stdin);
    int n;
    int k;
    cin >> n >> k;
    vector<pair<double, double>> per(n);
    for (int i = 0; i < n; i++){
        cin >> per[i].first;
    }
    for (int i = 0; i < n; i++){
        cin >> per[i].second;
    }
    sort(per.begin(), per.end(), [&](pair<double, double> &a, pair<double, double> &b)
        { return (a.second / a.first) > (b.second / b.first); });

    double total_weight = 0, total_benefit = 0;
    for (int i = 0; i < n; i++){
```

```

    if (total_weight < k){
        int temp = min(k - total_weight, per[i].first);
        double ratio = per[i].second / per[i].first;
        total_benefit += temp *10, ratio;
        cout << temp << " " << temp * ratio << "\n";
        total_weight += temp;
    }
    else
        break;
}
cout << total_benefit << endl;
}

```

Sample I/O:

Input

```

5 20
2 4 5 3 9
3 5 8 4 10

```

Output

```

5 8
2 3
3 4
4 5
6 6.66667
26.6667

```

Discussion & Conclusion: The Fractional Knapsack Problem, as implemented, efficiently uses a **greedy algorithm** to maximize the total benefit. The primary strength of this approach is its simplicity and speed, with a time complexity of $O(n \log n)$, dominated by the sorting step. This makes it suitable for scenarios where a quick solution is required for a large number of items. However, the reliance on item divisibility limits its real-world applicability in scenarios where items cannot be split. For such cases, dynamic programming approaches, such as the 0/1 Knapsack algorithm, may be more appropriate.

Lab Report 4

Merge and Quick

by Kefaet Ullah (2103011)

to Dr. Md. Ali Hossain(Professor, CSE, RUET)

Task 1

Problem Statement: Compare the **number of swaps** required during the sorting process between **Merge Sort** and **Quick Sort**.

Introduction: Sorting is a fundamental operation in computer science, with algorithms like Merge Sort and Quick Sort widely used for their efficiency and adaptability. While these algorithms are often compared based on their time complexity, analyzing the **number of swaps** provides a unique perspective on their internal workings and computational cost.

- **Quick Sort:** An efficient divide-and-conquer algorithm, where swaps occur during the partitioning phase to organize elements relative to a pivot.
- **Merge Sort:** A stable sorting algorithm that divides data into smaller subarrays and merges them while ensuring order. In this study, a "swap" in Merge Sort refers to moving elements from the left subarray to the right subarray during the merging process.

By quantifying and comparing the swap counts, this analysis aims to provide insights into the hidden costs of these algorithms, which can have implications in environments where memory operations or data movement are critical.

Algorithm:

Step 1: Generate Input Data

Generate an array `nums[]` of size `n` with random integers.

Step 2: Initialize Swap Counters

Initialize counters `mSwaps = 0` (for Merge Sort) and `qSwaps = 0` (for Quick Sort).

Step 3: Implement Merge Sort with Swap Counting

Count a swap whenever an element is placed in its new position during merging.

Update the `mSwaps` counter during each merge.

Step 4: Implement Quick Sort with Swap Counting

Partition the array into two parts:

Elements less than the pivot on the left.

Elements greater than the pivot on the right.

Count a swap whenever two elements are exchanged during partitioning.

Recursively apply the above steps to the left and right subarrays.

Step 5: Compare Swaps and Display Results

Code:

```
#include <bits/stdc++.h>
using namespace std;
void generateFile(const string &filename, int n){
    ofstream outFile(filename);
    if (!outFile.is_open()){
        cerr << "Error " << filename << endl;
        return;
    }
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dist(1, 100000);
```

```

        for (int i = 0; i < n; ++i){
            outFile << dist(gen) << " ";
        }
        outFile.close();
    }

void merge(vector<int> &arr, int left, int mid, int right, long long &swaps){
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; ++i)
        L[i] = arr[left + i];
    for (int i = 0; i < n2; ++i)
        R[i] = arr[mid + 1 + i];
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2){
        if (L[i] <= R[j]){
            arr[k++] = L[i++];
        }
        else{
            arr[k++] = R[j++];
        }
        swaps++;
    }
    while (i < n1){
        arr[k++] = L[i++];
        swaps++;
    }
    while (j < n2){
        arr[k++] = R[j++];
        swaps++;
    }
}

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

int partition(vector<int> &arr, int low, int high, long long &swaps){
    int pivot = arr[low];
    int i = low - 1;
    int j = high + 1;
    while (true){

```

```

        do{
            ++i;
            ++swaps;
        } while (arr[i] < pivot);
        do{
            --j;
            ++swaps;
        } while (arr[j] > pivot);
        if (i >= j)
            return j;

        swap(arr[i], arr[j]);
    }
}

void quickSort(vector<int> &arr, int low, int high, long long &swaps){
    if (low < high){
        int pivot = partition(arr, low, high, swaps);
        quickSort(arr, low, pivot - 1, swaps);
        quickSort(arr, pivot + 1, high, swaps);
    }
}

int main(){
    string file = "number.txt";
    int n;
    cout << "n: ";
    cin >> n;
    generateFile(file, n);
    ifstream inFile(file);
    if (!inFile.is_open()){
        cerr << "Error" << file << endl;
        return 1;
    }
    vector<int> nums;
    int num;
    while (inFile >> num){
        nums.push_back(num);
    }
    inFile.close();
    vector<int> mSortNums = nums;
    long long mSwaps = 0;
    mergeSort(mSortNums, 0, mSortNums.size() - 1, mSwaps);
    cout << "Merge: " << mSwaps << endl;
    vector<int> qSortNums = nums;
    long long qSwaps = 0;
    quickSort(qSortNums, 0, qSortNums.size() - 1, qSwaps);
    cout << "Quick: " << qSwaps << endl;
}

```

```

    cout << "Worst Case: " << endl;
    sort(mSortNums.begin(), mSortNums.end());
    reverse(mSortNums.begin(), mSortNums.end());
    mSwaps = 0;
    mergeSort(mSortNums, 0, mSortNums.size() - 1, mSwaps);
    cout << "Merge (descending): " << mSwaps << endl;
    sort(qSortNums.begin(), qSortNums.end());
    qSwaps = 0;
    quickSort(qSortNums, 0, qSortNums.size() - 1, qSwaps);
    cout << "Quick (ascending): " << qSwaps << endl;
    return 0;
}

```

Sample I/O:

Input

n: 1000

Output

Merge: 9976

Quick: 12771

Worst Case:

Merge (descending): 9976

Quick (ascending): 501503

Discussion & Conclusion: The results demonstrate distinct characteristics of Merge Sort and Quick Sort with respect to the number of swaps, particularly under normal and worst-case scenarios:

1. **Merge Sort:** The number of swaps in Merge Sort remains **relatively consistent** regardless of the input order (normal or descending). For $n = 1000$, Merge Sort records **9976 swaps**, showing no significant deviation even in its worst-case input (descending order). This suggests that the swap count depends primarily on the size of the array, not the initial arrangement of elements.
2. **Quick Sort:** In a normal scenario (random input), Quick Sort records **12771 swaps**, which aligns with its average-case complexity of $O(n \log n)$. However, in its worst-case scenario (ascending order), Quick Sort performs **501,503 swaps**, a staggering increase that reflects its $O(n^2)$ behavior when the pivot selection is poor. The worst-case swap count approximates $\frac{n^2}{2}$ as expected theoretically, making Quick Sort highly sensitive to the input order.

This analysis underscores the trade-offs between the two algorithms, emphasizing Merge Sort's stability and Quick Sort's sensitivity to input order.