Kingdom of Saudi Arabia
Ministry of Education
Imam Abdulrahman bin Faisal University
Computer Science department

Algorithm Analysis and Design

Term 2 – 2023

CIS Year 4, G3

Supervised By:

Dr. Azza A. Ali

Student Name	Student ID	Role
Shahed Aljiaan	2200003861	Member
Fatimah Alrastem	2200003453	Leader
Layan Alghamdi	2200002543	Member
Renad Alshahrani	2200002949	Member
Ghaida Alotaibi	2200002466	Member

List of Contents

List of Figures

List of Tables

Introduction

This project aims on identifying and analyzing The performance of different sorting algorithms (Merge, Selection and Quick sort) in different sorting cases, increasing, decreasing and random generation of numbers. Each of these cases will be calculated using both, theoretical and experimental time and then all of the cases will be conducted.

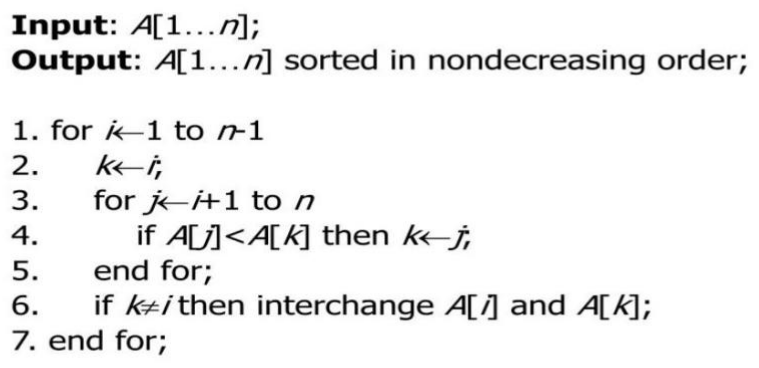
1. Theoretical Question

1.1 Selection Sort:

Definition:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty, and the unsorted part is the entire list. The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right [5].

Input:



```
Input: $A[1...n]$;
Output: $A[1...n]$ sorted in nondecreasing order;

1. for $i \leftarrow 1$ to $n-1$
2.     $k \leftarrow i$;
3.     for $j \leftarrow i+1$ to $n$
4.         if $A[j] < A[k]$ then $k \leftarrow j$;
5.     end for;
6.     if $k \neq i$ then interchange $A[i]$ and $A[k]$;
7. end for;
```

Figure 1: Selection Sort Algorithm

	Best Case	**Average Case**	**Worst Case**
Data Order	Data sorted increasingly	Randomly	Data sorted decreasingly
Time Complexity	$T(n) = \Theta(n^2)$	$T(n) = \Theta(n^2)$	$T(n) = \Theta(n^2)$

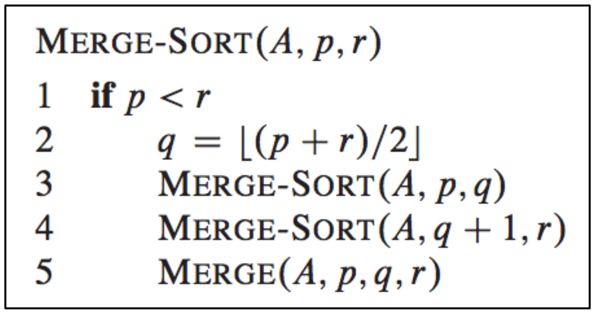
Table 1: Theoretical Question for Selection sort

1.2 Merge Sort:

Definition:

Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.
In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is
repeated until the entire array is sorted [1].

Input:



```
MERGE-SORT(A, p, r)
1  if $p < r$
2      $q = \lfloor (p + r)/2 \rfloor$
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Figure 2: Merge Sort Algorithm

	Best Case	**Average Case**	**Worst Case**
Data Order	Data sorted increasingly	Randomly	Data sorted decreasingly
Time Complexity	$T(n) = \Theta(n \log n)$	$T(n) = \Theta(n \log n)$	$T(n) = \Theta(n \log n)$

Table 2: Theoretical Question for Merge sort

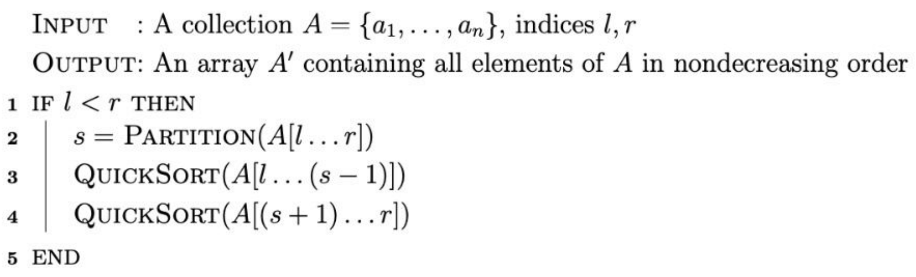
1.3 Quick Sort:

Definition:

Quicksort is a sorting algorithm based on the divide and conquer approach where
An array is divided into subarrays by selecting a pivot element (element selected from the array).
While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements

greater than pivot are on the right side of the pivot. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
At this point, elements are already sorted. Finally, elements are combined to form a sorted array [3].

Input:



```
INPUT  : A collection $A = \{a_1, \ldots, a_n\}$, indices $l, r$
OUTPUT: An array $A'$ containing all elements of $A$ in nondecreasing order
IF $l < r$ THEN
    $s = $ PARTITION($A[l \ldots r]$)
    QUICKSORT($A[l \ldots (s-1)]$)
    QUICKSORT($A[(s+1) \ldots r]$)
END
```

Figure 3:Quick Sort Algorithm

	Best Case	**Average Case**	**Worst Case**
Data Order	Data sorted increasingly.	Randomly	Data sorted decreasingly
Time Complexity	$T(n) = \Theta(n \log n)$	$T(n) = \Theta(n \log n)$	$T(n) = \Theta(n^2)$

Table 3: Theoretical Question for Quick sort

2. Generation and experimental setup

2.1 Description of the characteristics of the machine used:

Random Access Memory (RAM)	**Central processer unit (CPU)**	**Operating System (OS)**	**System Type**
8 GB	Core i7	Windows OS	64-bit

Table 4: characteristics of the device used in this experiment.

2.2 What timing mechanism?

We decided to use CurrentTimeMillis() method that returns the current time in milliseconds. Each value accuracy of this method relies on the operating system from device to another.

2.3 How many times did you repeat each experiment? Explain your choice.

Because the outcomes were always close, we conducted each experiment four times. Therefore, in the majority of sorting algorithms, four iterations were sufficient to arrive at a decent result. We have noticed that most algorithms take more time when the inputs to the algorithm are increasing. Due to this reason, each algorithm was run for a different input size.

2.4 What times are reported?

We used the execution time as the average of the four times it was obtained. Most of the time, the execution time disparity was steadily growing. Although the method's running duration was accurately calculated, we also determined how long each algorithm takes on average. The average time displays a distinct value that succinctly summarizes the running time.

2.5 What is the input to the algorithms? How did you select it?

The algorithms' input ranges from 100 to 100000. Starting at 100 this range was selected because it clearly demonstrates how the input size affects how long each method takes to run. We stopped at the number 100000 because several algorithms' running times were excessively long, causing the application to operate slowly and overheat the device.

2.6 Did you use the same inputs for all sorting algorithms?

Yes, to make it easier to compare the sorting algorithms and arrive at the most accurate conclusion, we did utilize the same input for all sorting algorithms. Furthermore, since all sorting algorithms utilize the same input, they should all yield accurate and dependable results. For a fair comparison of all the types, the same number of inputs were used.

3. The three sorts perform

The best, average, and worst cases of each sorting algorithm will be shown in this question as the experiment's outcome. The line graphs below show median time (in microseconds) and input size (on the X-axis).

3.1. Graph the best-case running time as a function of input size n for the three sorts (use the best case input you determined in each case in part 1)

The ideal scenarios for each sorting algorithm are:
selection sort: growing the list (Θ (n^2))
Merge sort: growing the list. (Θ (n log n))
Quicksort: Growing the list. (O(n log n))

The best cases for each sorting algorithm are shown in the figure. The Merge and Quick sort outperform the sorting algorithm in the best cases, as shown by the line graph, which demonstrates that they are the fastest sorting algorithm as it takes (n log n), while comparing the best cases. The others, however, the second-fastest sorting algorithm is the selection sort, which completes in (O n^2)[2].

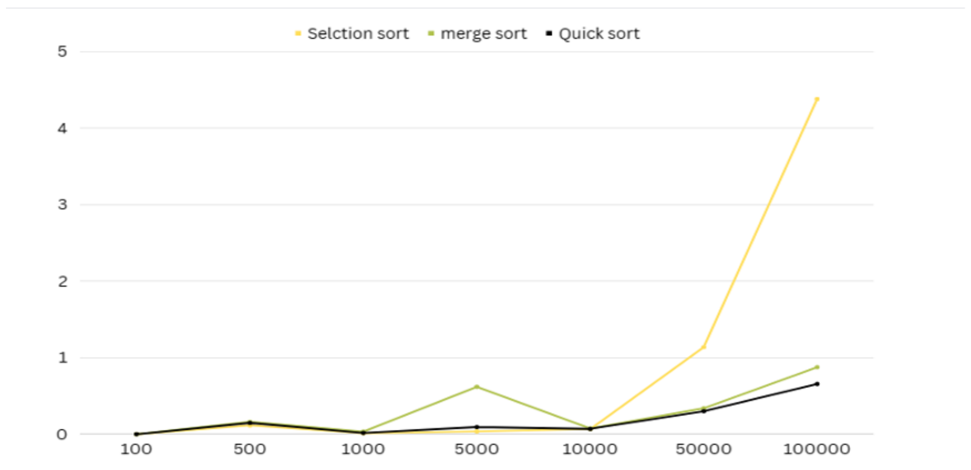


Figure 4: Best case of all sorting algorithm

3.2. Graph the worst-case running time as a function of input size n for the tree sorts (use the best case input you determined in each case in part 1)

Worst case: The worst-cases of all sorting algorithms:
− Selection Sort: Random list. (Θ (n^2))
− Merge sort: Random list. (Θ (n log n))
− Quick Sort: Random list. (Θ(n^2))

The worst cases for each sorting algorithm are shown in the figure. The line graph shows that the quick sort and Merge sort performs best, whereas selection sort is the worst sorting algorithms[2].

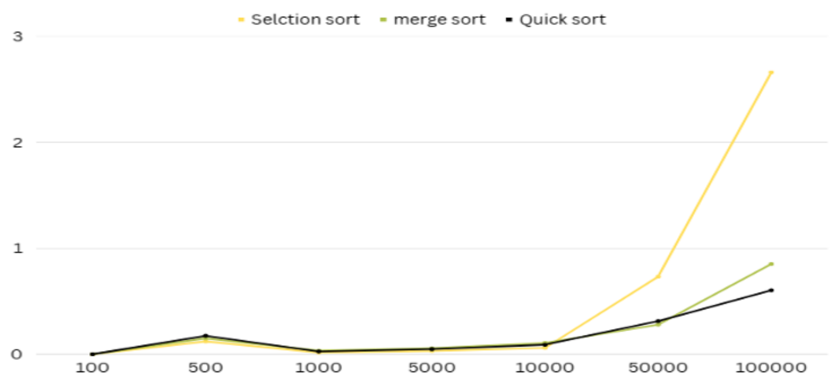


Figure 5: Worst case of all sorting algorithm

3.3. Graph the average case running time as a function of input size n for the three sorts

The average case:
selection sort: Random list (Θ (*n*^2))
Merge sort: growing the list. (Θ (n lg n))
Quicksort: Growing the list. (O(n lg n))

The average cases for each sorting algorithm are shown in the figure. The line graph demonstrates that the merge Quick sorting algorithms used is the best. (Θ (n lg n)). because of how poorly the selection sort performs on average across all sorting (Θ (*n*^2))[2].

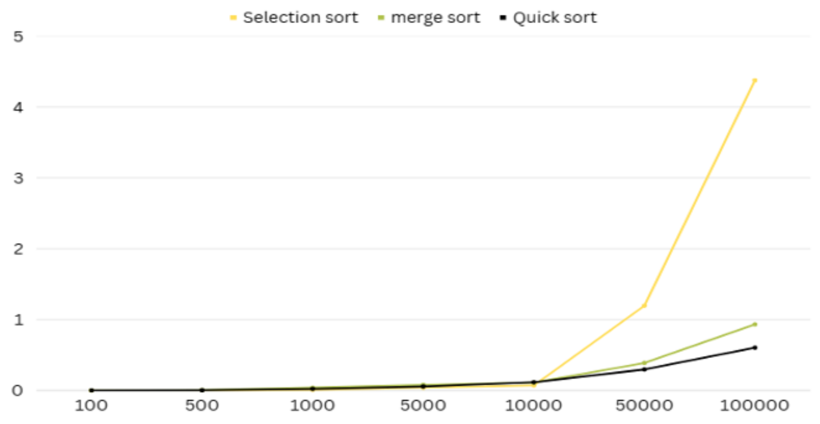


Figure 6: Average case case of all sorting algorithm

3.4. Discussion of the best performance sorting algorithm

when determining the optimal running time. We conclude that merging sort is the fastest sorting algorithm by always taking the worst-case scenario into account. Merge sort, on the other hand, is quicker than Quick sort in the worst case. The merge sort is therefore the most effective

4. To what extent does the best, average, and worst-case analyses (from class/textbook) of each sort agree with the experimental results?

In this question, the ratio of theoretical running time to experimental running time is used to compare the asymptotic order of theoretical running time to that of experimental running time. The outcomes of experiments sometimes deviate from those of theory. The usage of the CPU, the speed of the compiler, and the programming language are just a few variables that may have an impact on the running time. The x-axis shows the input size, while the y-axis shows the ratio between theoretical and experimental run times.

4.1. For each sort, and for each case (best, average, and worst), determine whether the observed experimental running time is of the same order as predicted by the asymptotic analysis

4.1.1 Selection sort

Cases	Best case		Average case		Worst case	
Input size	Theoretical Time	Practical Time	Theoretical Time	Practical Time	Theoretical Time	Practical Time
100	10000	0.3678	10000	0.3776	10000	0.5835
500	250000	2.3404	250000	11.1503	250000	2.8195
1000	1000000	106.8826	1000000	16.2546	1000000	34.3706
5000	25000000	66.5106	25000000	60.4879	25000000	55.9923
10000	100000000	90.8322	100000000	287.0152	100000000	118.441
50000	2500000000	563.04448	2500000000	1785.56352	2500000000	1878.73293
100000	10000000000	1921.44486	10000000000	6491.2481	10000000000	5942.0099

Table 5: selection Sort Algorithm's Time Complexity Case

Cases	Best case	Average case	Worst case
Input size (n)	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$
100	1.226	0.03776	0.05835
500	2.3404	1.48670667	0.3759333333
1000	0.01068826	0.54182	0.343707
5000	0.664106	0.604879	0.559923
10000	1.816644	0.22961216	0.7896066667
50000	2.25217792	1.428450816	1.502986344
100000	0.384288972	1.29824962	1.18840198

Table 6: Error Ratio for the selection Sort Algorithm's cases

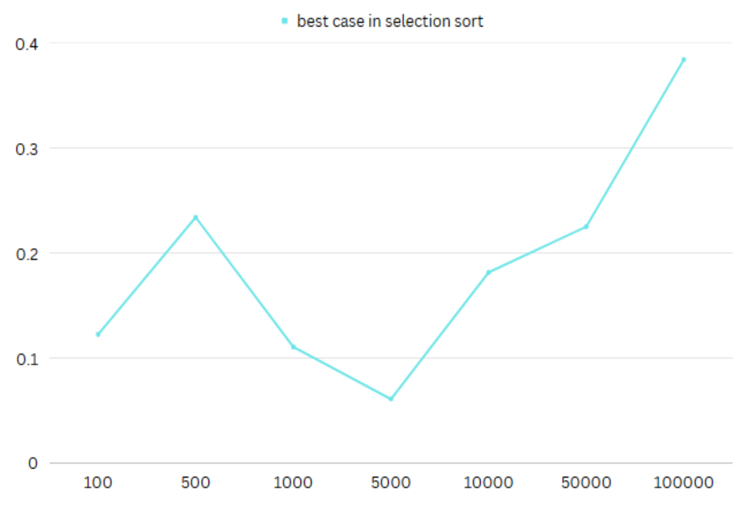


Figure 7: Best case of Selection sort

The figure shows that the best-case scenario for the above line graph's lack of a clearly defined horizontal line is shown via selection sort. Therefore, a horizontal line can show up if we use different input sizes.

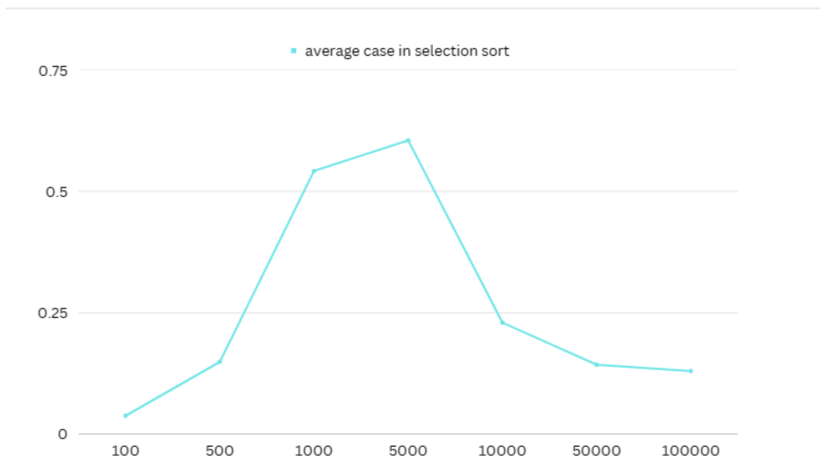


Figure 8: Average case of Selection sort

The picture illustrates how selection sort is used to display the average-case scenario for the above line graph's lack of a distinct horizontal line. Therefore, if we use larger input sizes, a horizontal line may appear.

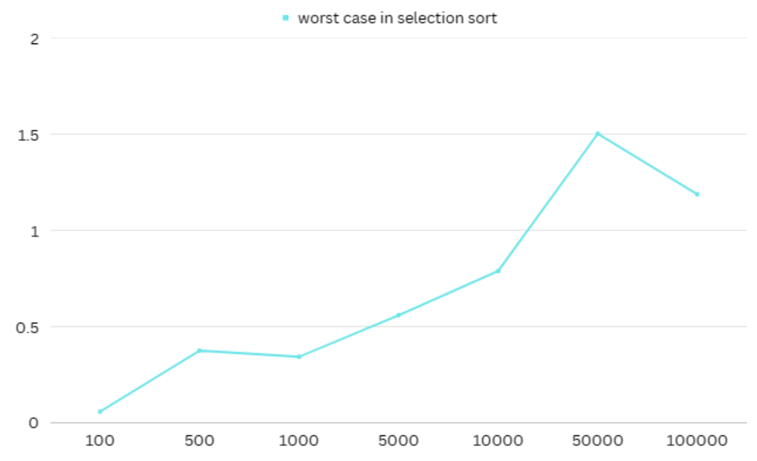


Figure 9: Worst case of Selection sort

The worst-case scenario for the above line graph's lack of a clearly defined horizontal line is shown via selection sort. Therefore, a horizontal line can show up if we use bigger input sizes.

4.1.2 Merge sort

Cases	Best case		Average case		Worst case	
Input size	Theoretical Time	Practical Time	Theoretical Time	Practical Time	Theoretical Time	Practical Time
100	200	1.1832	200	0.6874	200	812234960
500	1349.485002	1.4155	1349.485002	2.1782	1349.485002	812262940
1000	3000	23.7872	3000	20.2924	3000	812295690
5000	18494.85002	72.284704	18494.85002	48.1811	18494.85002	812329980
10000	40000	104.9732	40000	101.884704	40000	812354340
50000	234948.5002	373.298912	234948.5002	315.849696	234948.5002	812376020
100000	500000	669.42208	500000	667.8953	500000	812402260

Table 7: Merge Sort Algorithm's Time Complexity Case

Cases	Best case	Average case	Worst case
Input size (n)	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$
100	0.5916	0.3437	40617480
500	0.104891866	0.1614097227	60190586.69
1000	2.643022222	0.6764133333	4002955244
5000	0.39083693	0.2605108987	4392195.552
10000	0.262433	0.25471176	20308858.5
50000	0.15888854203	0.1344335868	345767.6977
100000	0.133884416	0.13357906	162480.452

Table 8: Error Ratio for the Merge Sort Algorithm's cases

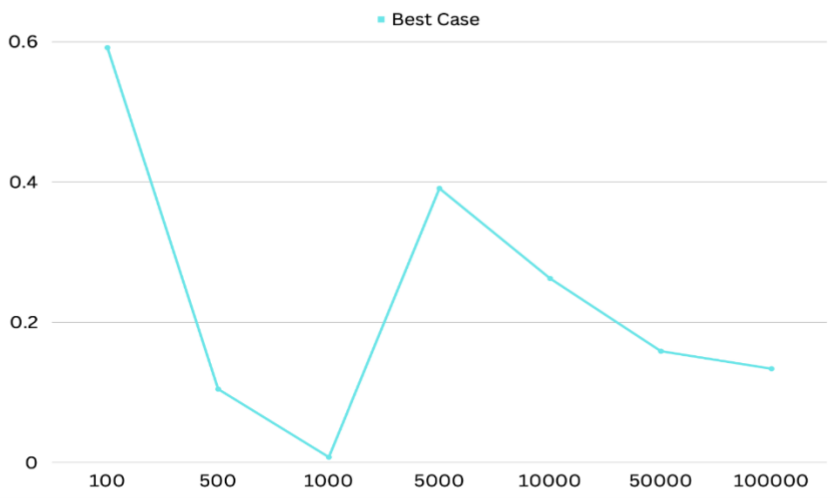


Figure 10: Best case of Merge sort

The graphic illustrates how merge sort is used to display the best-case scenario for the absence of a distinct horizontal line in the line graph above. Considering this, using larger input sizes may result in the appearance of a horizontal line.

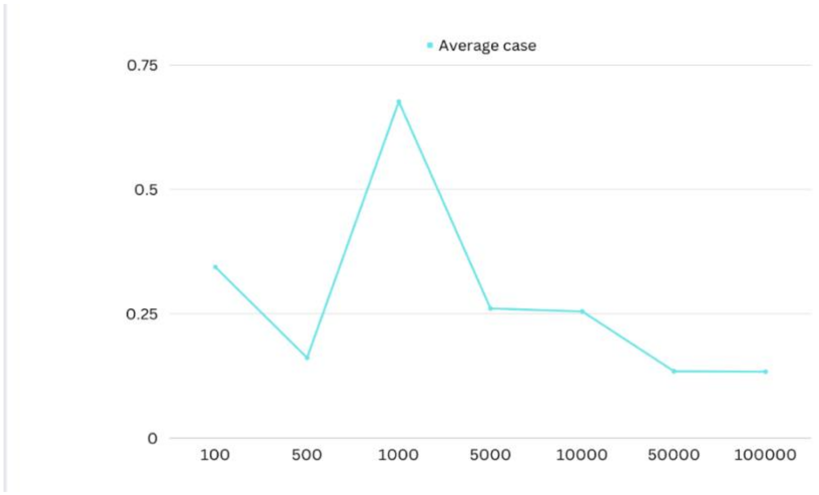


Figure 11: Average case of Merge sort

Like a Best case, There does not appear to be a distinct horizontal line in the line graph above. Therefore, if we had used bigger input sizes, a horizontal line might have shown up.

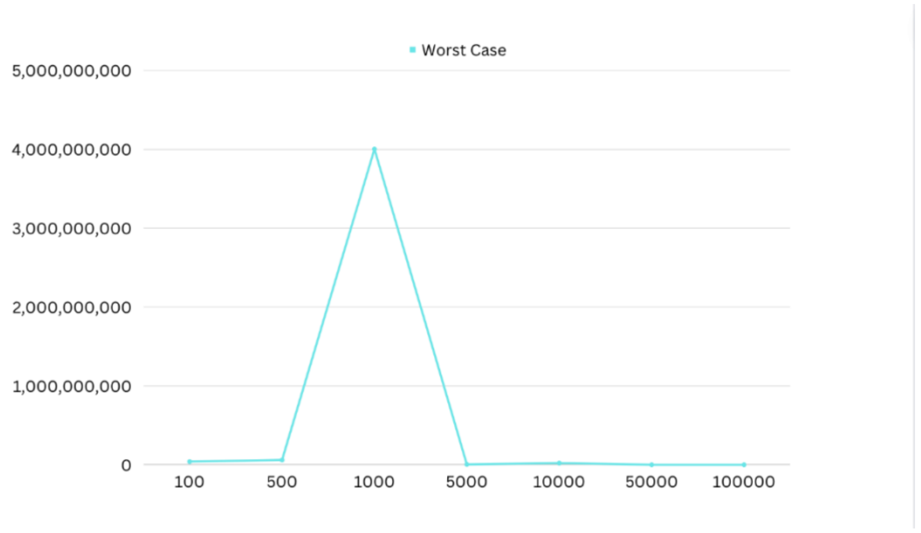


Figure 12: Worst case of Merge sort

It displays a merging sort's worst-case scenario. We notes a horizontal line between=100 and n=500 Then a horizontal line can be seen in n=5000. These findings allow us to draw the conclusion that the experiment supports the hypothesis, which is (n log n).

4.1.3 Quick sort

Cases	Best case		Average case		Worst case	
Input size	Theoretical Time	Practical Time	Theoretical Time	Practical Time	Theoretical Time	Practical Time
100	200	0.4655	200	0.6157	10000	809282700
500	1349.485002	1.0675	1349.485002	1.9257	250000	809323170
1000	3000	14.1009	3000	8.0233	1000000	809357730
5000	18494.85002	50.1287	18494.85002	44.0026	25000000	809401020
10000	40000	125.298	40000	97.438096	100000000	809430410
50000	234948.5002	292.411008	234948.5002	313.8768	2500000000	809458060
100000	500000	617.80122	500000	592.97619	10000000000	809498660

Table 9: Quick Sort Algorithm's Time Complexity Case

Cases	Best case	Average case	Worst case
Input size (n)	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$
100	0.23275	0.30785	8092827
500	0.07910425076	0.1426988812	323729.268
1000	0.47003	0.267443333	80935.773
5000	0.2710413977	0.2379181229	3237.60408
10000	0.313245	0.24359524	809.43041
50000	0.1244574908	0.1335938726	32.3783224
100000	0.123560244	0.118595238	8.0945806

Table 10: Error Ratio for the Quick Sort Algorithm's cases

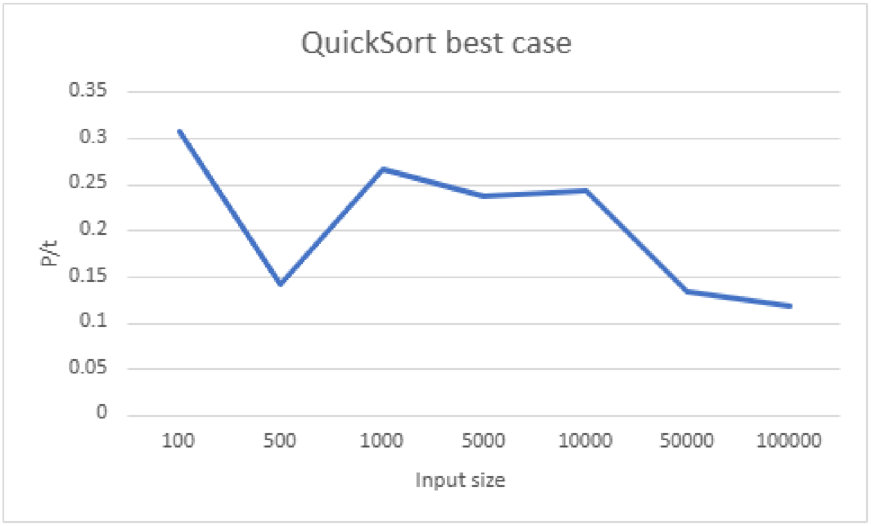


Figure 13: Best case of Quick sort

The figure illustrates the best-case situation for the absence of a specifically stated horizontal line in the above line graph. Therefore, if we use new input sizes, a horizontal line may result.

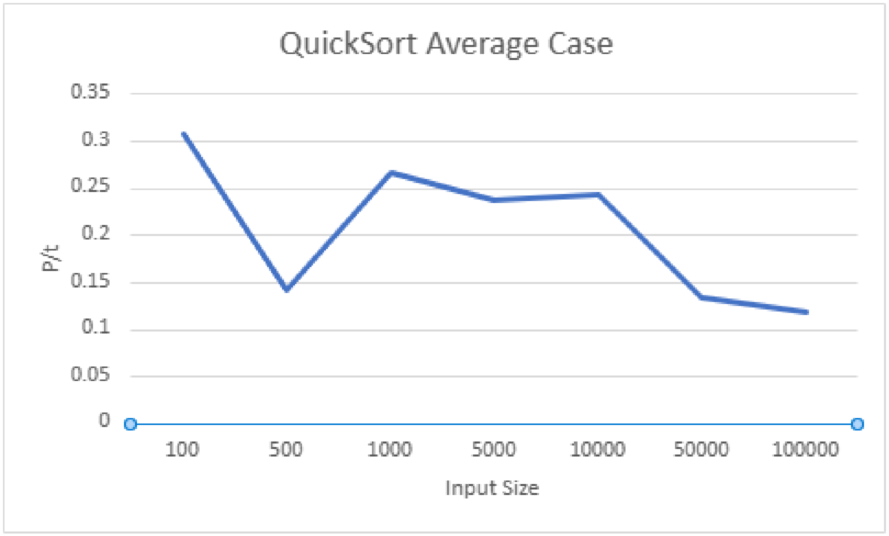


Figure 14: Average case of Quick sort

Like the best case, The figure illustrates the best-case situation for the absence of a specifically stated horizontal line in the above line graph. Therefore, if we use new input sizes, a horizontal line may result.

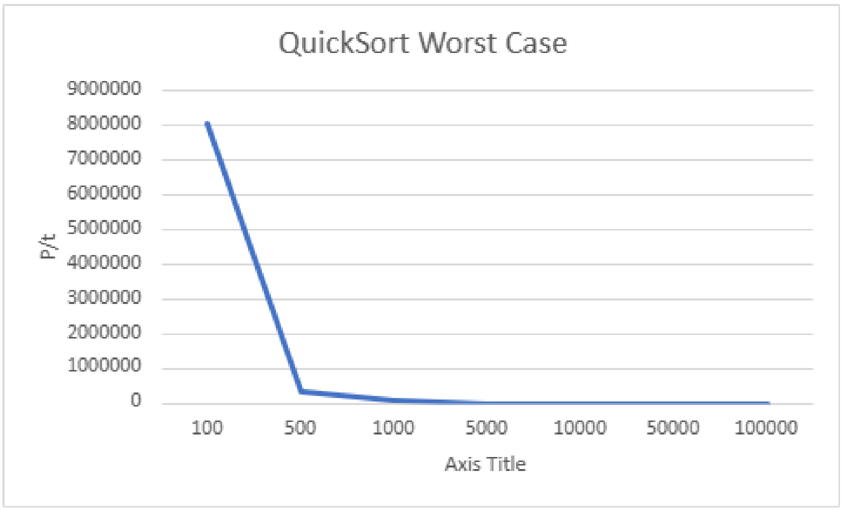


Figure 15: Worst case of Quick sort

demonstrates the worst-case rapid sort analysis. There does not appear to be a distinct horizontal line in the line graph above. In contrast to the difference between the previous subsequent inputs, the experimental running time difference at input 500 began to converge. These findings allow us to draw the conclusion that the experiment supports the hypothesis, which is (n2).

5. For the comparison sorts, is the number of comparisons really a good predictor of the execution time? In other words, is a comparison a good choice of basic operation for analyzing these algorithms?

Yes, this project uses different input sizes of the same numbers for each of the three sorting algorithms (100, 500, 1000, 5000, 10000, 50000, 100000). All of the algorithms will be run and compared in the cases of increasing, random, and decreasing order for the all of the input sizes. The execution time will be accurate and correct if all sorting algorithms use the same input.

5.1. Selection Sort Algorithm

The table below shows the number of comparisons performed during the sorting algorithm using the equation (n^ 2 /2) and the execution time calculated above using the selection sort code, and the graph shows the representation of the values.

Input Size (n)	Number of comparisons	Execution Time (µs)
100	664.4	2.1636ms
500	4483	5.69ms
1000	9965.8	29.6679ms
5000	61438.6	194.44ms
10000	132877.1	534.72ms
50000	780482	2637.16ms
100000	3521928	5107.87ms

Table 11: Selection Sort Algorithm

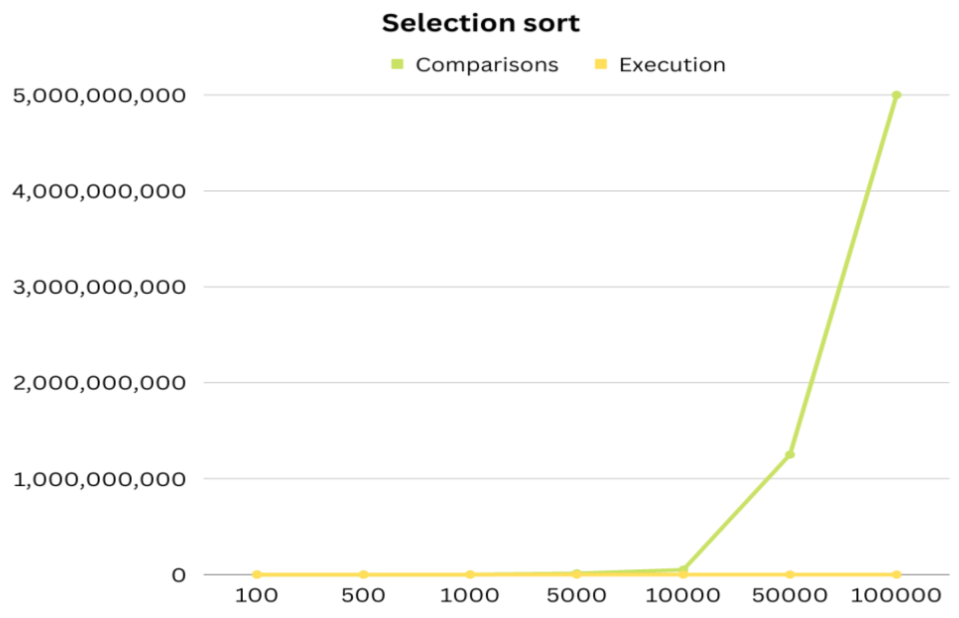


Figure 16 Number of Comparison for Selection Sort

As shown in the graph, the number of comparisons has faster growth than the execution time.

5.2. Merge Sort Algorithm

The table below shows the Number of Comparisons performed during the sorting algorithm using the equation (n log n) and the execution time calculated above using merge sort code, and the graph shows the representation of the values.

Input Size (n)	Number of comparisons	Execution Time (µs)
100	**5000**	**329.77ms**
500	**125000**	**530 ms**
1000	**500000**	**378.46ms**
5000	**12500000**	**674.06ms**
10000	**50000000**	**1170.88ms**
50000	**1250000000**	**7124.7ms**
100000	**5000000000**	**17512.28ms**

Table 12: Merge Sort Algorithm

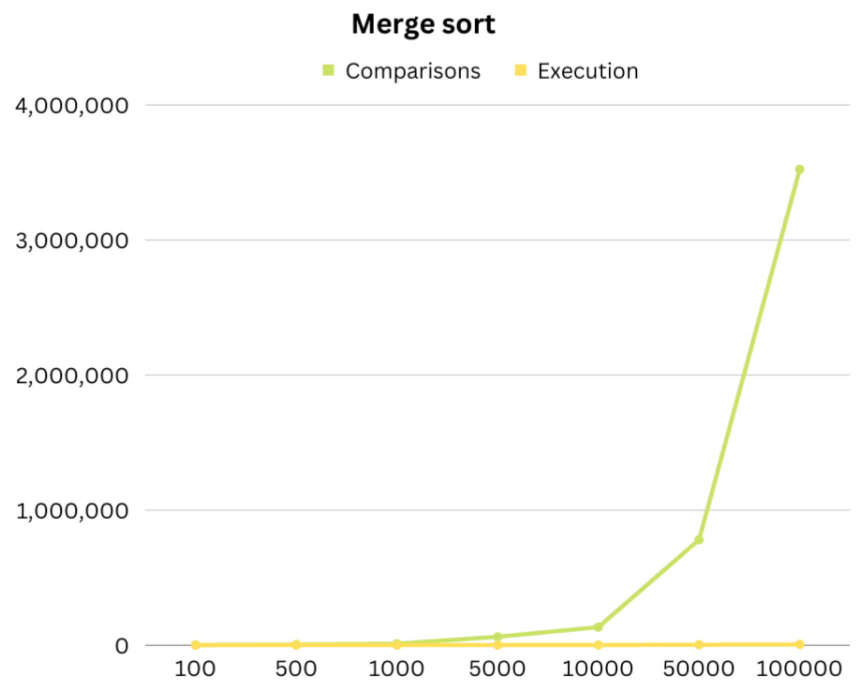


Figure 17 Number of Comparison for Merge Sort

As shown in the graph, the number of comparisons has faster growth than the execution time.

5.3. Quick Sort Algorithm

The table below shows the Number of Comparisons performed during the sorting process using the equation (n^2) and the execution time calculated above using Quick sort code. And the graph shows the representation of the values.

Input Size (n)	Number of comparisons	Execution Time (µs)
100	**5000**	**329.77ms**
500	**125000**	**530 ms**

1000	**500000**	**378.46ms**
5000	**12500000**	**674.06ms**
10000	**50000000**	**1170.88ms**
50000	**1250000000**	**7124.7ms**
100000	**5000000000**	**17512.28ms**

Table 13: Quick Sort Algorithm

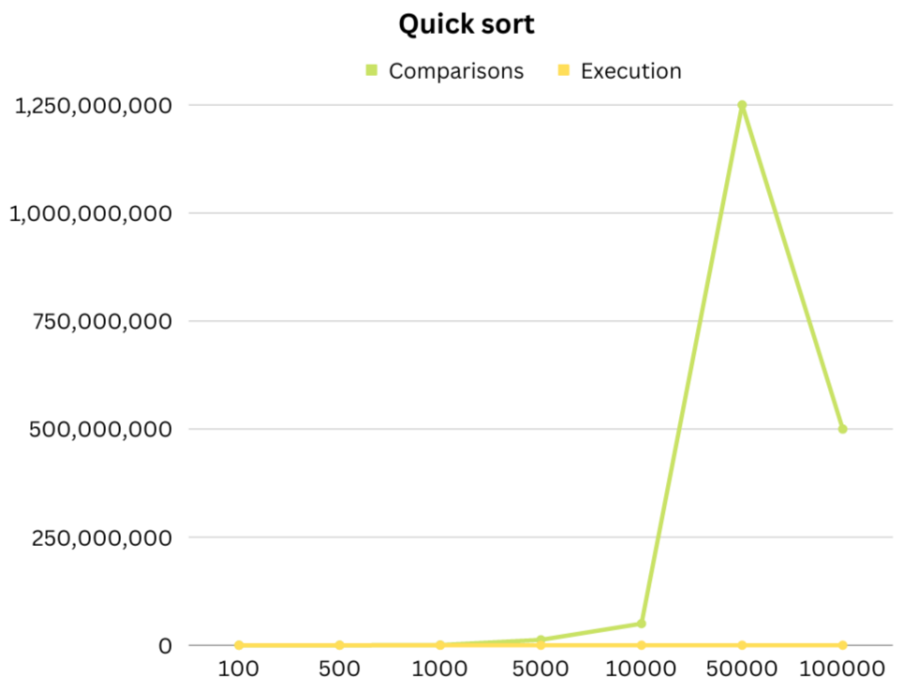


Figure 18 Number of Comparison for Quick Sort

As shown in the graph, the number of comparisons has faster growth than the execution time

6. Design and analysis of an improved Divide-and-conquer algorithm to compute matrix multiplication [1]

6.1. Solve the problem in a brute-force manner

Brute force algorithm: This kind of algorithm is the most fundamental and straightforward, is a simple solution to a problem, or the first solution that springs to mind when we perceive the problem. Technically speaking, it is equivalent to iterating through all of the options available to resolve that issue.

```
def multiply(A, B, n):
 C = []
 for i in range(n):
 for j in range(n):
 for k in range(n):
 C[i][j] += A[i][k]*B[k][j]
 return C
```

Running time analysis:

code	**constant**	**time**
C = []	**c1**	**1**
for i in range(n):	**c2**	**(n+1)**
for j in range(n):	**c3**	**n(n+1)**
for k in range(n):	**c4**	**(n+1)(n(n+1))-1**
C[i][j] += A[i][k]*B[k][j]	**c5**	**(n+1)(n(n+1))-1)-1**

Table 14: Running time analysis (1)

Time complexity:

c1 + c2 ×(n+1) + c3 × (n(n+1) + c4 × (n+1)(n(n+1))-1 + c5 × (n+1)(n(n+1))-1-1 =

T(n) = O(n^3).

6.2. Efficient algorithm and time complexity

Practical code in java:

```
static int MatrixMultyply(int arrayOne[][size], int arrayTwo[][size], int
arrayThree[][size])
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            arrayThree [i][j] = 0;
            for (int k = 0; k < size; k++)
            {
                arrayThree [i][j] += arrayOne[i][k]* arrayTwo[k][j];
            }
        }
    }
}
```

Running time analysis [4]:

code	constant	time
for (int i = 0; i < size; i++)	**c1**	**n+1**
for (int j = 0; j < size; j++)	**c2**	**n(n+1)**
arrayThree [i][j] = 0;	**c3**	**n(n+1)-1**
for (int k = 0; k < size; k++)	**c4**	**(n+1)(n(n+1))-1**

Table 15: Running time analysis (2)

Time complexity:

c1 × (n+1) + c2 ×n(n+1) + c3 × (n(n+1)-1 + c4 × (n+1)(n(n+1))-1 =

T(n) = O(n^3).

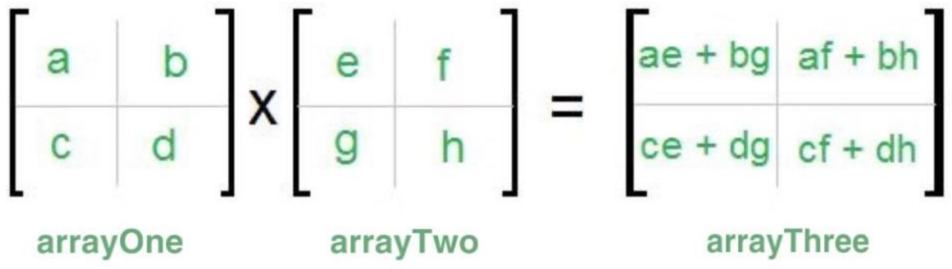


Figure 19: Matrix multiplication

7-References:

[1] *GeeksForGeeks*. (2022, November 21). Retrieved November 10, 2023, from https://www.geeksforgeeks.org/strassens-matrix-multiplication/

[2] *Great Learning*. (2022, November 7). Retrieved February 10, 2023, from https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/#time-complexity-of-sorting-algorithms

[3] *Khan Academy*. (2023). Retrieved February 10, 2023, from https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort

[4] Strassen's Matrix Multiplication | Divide and Conquer. (2017, July 2). In *YouTube*. Retrieved February 10, 2023, from https://youtu.be/E-QtwPi620I

[5] B.Hayfron-Acquah, J., Appiah, O., & Riverson, K. (2015, January 16). Improved Selection Sort Algorithm. *International Journal of Computer Applications*, *110*(5), 29–33. https://doi.org/10.5120/19314-0774

In-Text Citation: (B.Hayfron-Acquah et al., 2015)