Kingdom of Saudi Arabia
Ministry of Education
Imam Abdulrahman bin Faisal University
College of Science and Humanities in Jubail
Computer Science Department

المملكة العربية السعودية
وزارة التعليم
جامعة الإمام عبدالرحمن بن فيصل
كلية العلوم والدراسات الإنسانية بالجبيل
قسم علوم الحاسب

Algorithm Analysis and Design Course Project

First Semester 2022/2023

Group Members:

	Student name	ID
1	Warood Alzayer	2190004986
2	Hana Alomran	2190004931
3	Reem Almualem	2190000429
4	Danah AlMuzel	2190001201
5	Areej Ahmed Saleh	2190006001
6	Luluwah Waleed	2190004563

Supervised by:

Dr. Azza A. Ali

Kingdom of Saudi Arabia
Ministry of Education
Imam Abdulrahman bin Faisal University
College of Science and Humanities in Jubail
Computer Science Department

المملكة العربية السعودية
وزارة التعليم
جامعة الإمام عبدالرحمن بن فيصل
كلية العلوم والدراسات الإنسانية بالجبيل
قسم علوم الحاسب

Table of Content

Table of Figures

Table of Tables

Kingdom of Saudi Arabia
Ministry of Education
Imam Abdulrahman bin Faisal University
College of Science and Humanities in Jubail
Computer Science Department

المملكة العربية السعودية
وزارة التعليم
جامعة الإمام عبدالرحمن بن فيصل
كلية العلوم والدراسات الإنسانية بالجبيل
قسم علوم الحاسب

PART 1

1. INTRODUCTION

This report will discuss 3 types of sorting algorithms; insertion, merge and heap sort by analyzing them empirically and theoretically using these algorithms to sort data in ascending and descending order.

1.1 Sorting Algorithm Importance

Sorting Algorithms are important as it reduces the complexity of problems such as, reducing the complexity of searching since it is easy to locate items in sorted lists. Moreover, they rearrange lists, and increase the efficiency of performance. Furthermore, Table 1 shows where some of these algorithms can be used in real-world implementation, these are examples and not necessarily the main purpose of using these algorithms.

Algorithm	Real-world implementation example
Insertion	Shopping on a website and sorting the price from low to high or vice versa.
Merge	Databases to sort out data that is large.
Heap	Manufacturing to decide what project to work on next.
Quick	Numerical computations and scientific researches.

Table 1 Introduction to Sorting Algorithms

Admittedly, sorting algorithms are important in our everyday lives it is used in Google search, social media, public transportation schedules, Spotify, online shopping, etc.

1.2 Device Characteristics

This section shows the characteristics of the device used in this experiment.

Random Access Memory (RAM)	**Central processer unit (CPU)**	**Operating System (OS)**	**System Type**
8.00 GB (7.78 GB usable)	11th Gen Intel(R) Core (TM) i5-1135G7 @ 2.40GHz 2.42 GHz	Windows 11	64-bit operating system, x64-based processor

Table 2 Device Characteristics

1.3 Project's Objectives

This report shall yield the best, average, and worst cases for insertion, merge, and heap sorting algorithms according to the device's characteristics, input size, and rate of growth of the running time.

1.4 Timing Machine

In this project, nanoTime () built in method from Java Object Oriented Programming has been used to measure the start and stop time that each algorithm has spent sorting the array, then the time has been converted to milliseconds. The comparison was made based on the time that was received from the code.

1.5 Input Selection

In this project, multiple input sizes have been used for all algorithms that were implemented in this project, 100, 500, 1000, 5000, 10000, 50000, 100000, 200000. These input sizes were used to prove the correctness of all algorithms especially the insertion sort algorithm which has the lowest running time in the best case, because at the beginning it got the highest running time, but after expanding the input sizes and at a specific point it got the lowest running time as it should be. In this project multiple types of input have been used, which are:

- **Increasing ordered**

By using the Random class, a random array was generated and after using Array.sort() built in method the array was arranged in the ascending order and used to measure the best case for all algorithms.

- **Decreasing ordered**

By using the Random class, a random array was generated and after writing a specific code to arrange the array in a descending order, the array was ordered and used to measure the worst case for all algorithms.

- **Random ordered**

By using the Random class, a random array was generated and used to measure the average case of all algorithms.

The range of the array that was generated by the Random class was from 0 to 99.

2. IMPLEMENTATION

2.1 Insertion Sort Algorithm

Insertion sort is a sorting algorithm that places an unsorted element in the proper location after each iteration. Insertion sort functions similarly to how we sort the cards in our hands when playing cards. Given that we know the first card is already sorted, we choose an unsorted card.

2.1.1 Insertion Sort algorithm

Figure 1 shows the insertion sort algorithm and it's time complexity analysis, and Table 2 shows the time complexity for each case.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted
           sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Figure 1 Insertion Sort algorithm

Cases	Best Case	Average Case	Worst Case
The order of data	Data sorted increasingly.	Between best and worst case.	Data sorted decreasingly.
Time complexity	$T(N) = \emptyset(n)$	$T(N) = \emptyset(n^2)$	$T(N) = \emptyset(n^2)$

Table 3 Insertion Sort Algorithm's Cases and its Time Complexity

2.1.2 Insertion Sort Code

```
class Insertion{

    public Insertion(int[] numbers) {


    // Print the unsorted array //
    System.out.println("Before Insertion Sort:");
    printArray(numbers);


    insertionsort(numbers); // Calling the sort funct
    // Print the sorted array //
    System.out.println("\n After Insertion Sort:'");
    printArray (numbers);
    }

      public void printArray(int[] numbers)
    {
      for (int i=0; i < numbers.length; i++)
      {
      System.out.println(numbers[i]);
      }
    }
```

```
public void insertionsort (int[] inputArray)
{
  for (int i=1 ; i < inputArray.length; i++) // Loop
  {
      int currentValue = inputArray[i]; // Copy the cu
      int j = i-1; // Walk back towards the beginning


      // The while loop will keep testing until the co
      while(j>=0 && inputArray[j] > currentValue ) //
      {
      inputArray[j+1]=inputArray[j]; // Shifting //
      j--; // Decrement to keep walk back towards the
      }
   inputArray[j+1]=currentValue; // Put the temp value
  }
}
```

Figure 2 Insertion Sort Code

2.1.3 Insertion Sort Theoretical and Practical Analyze

Cases	Best Case		Average Case		Worst case	
Input size (n)	Theoretical Time	Practical Time (Actual Time)	Theoretical Time	Practical Time (Actual Time)	Theoretical Time	Practical Time (Actual Time)
100	100	3.353400ms	10000	1.950500ms	10000	1.399800ms
500	500	4.064600ms	250000	2.658800ms	250000	7.462600ms
1000	1000	5.933700ms	1000000	7.932100ms	1000000	13.781100ms
5000	5000	99.635400ms	25000000	106.165300ms	25000000	141.161700ms
10000	10000	294.527500ms	100000000	249.206700ms	100000000	461.818800ms
50000	50000	6165.683300ms	2500000000	2083.697300ms	2500000000	3557.153800ms
100000	100000	3564.733500ms	10000000000	5442.379400ms	10000000000	8168.329900ms
200000	200000	14541.174200ms	40000000000	30956.249700ms	40000000000	31047.511200ms

Table 4 Insertion Sort Algorithm's Time Complexity Cases

Case	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	3.5%	0.019%	0.014%
500	0.8%	0.0017%	0.0029%
1000	0.5%	0.00079%	0.0014%
5000	1.9%	0.00042%	0.00056%
10000	2.9%	0.00025%	0.00046%
50000	12.3%	0.000083%	0.00014%
100000	3.5%	0.000054%	0.000082%
200000	7.2%	0.000077%	0.00000077%

Table 5 Error Ratio for the Insertion Sort Algorithm's case

2.1.4 Insertion Sort Algorithm Comparison Graph (Theoretical & Practical Time Values)

- **Best case graph**

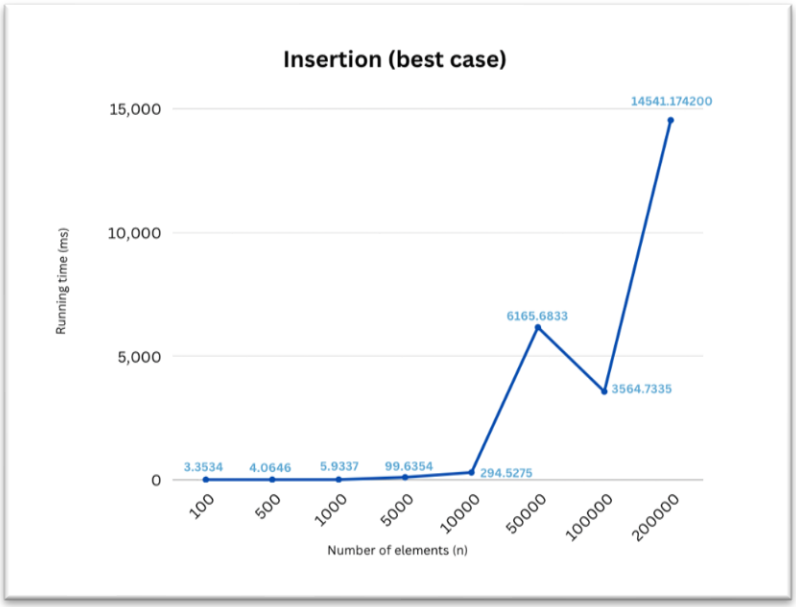

Figure 3 Insertion Sort Best Case Graph

- **Average case graph**

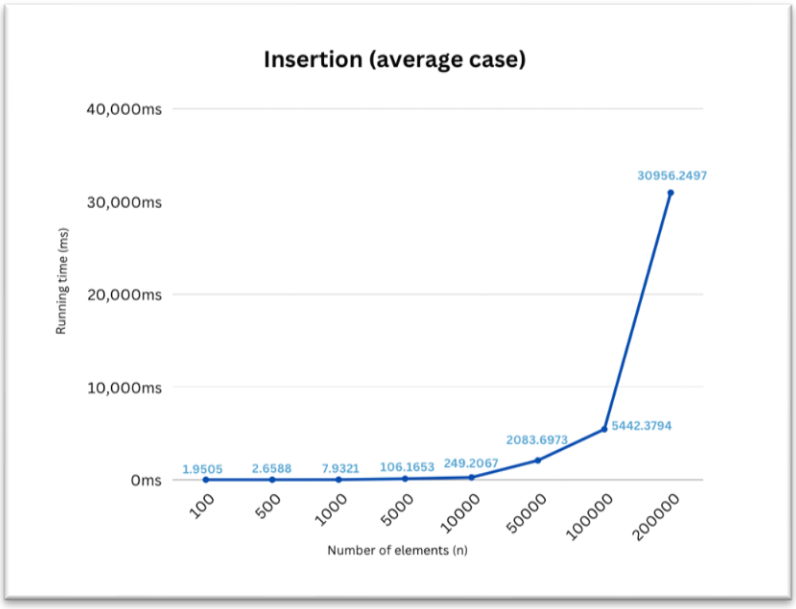

Figure 4 Insertion Sort Best Case Graph

- **Worst case graph**

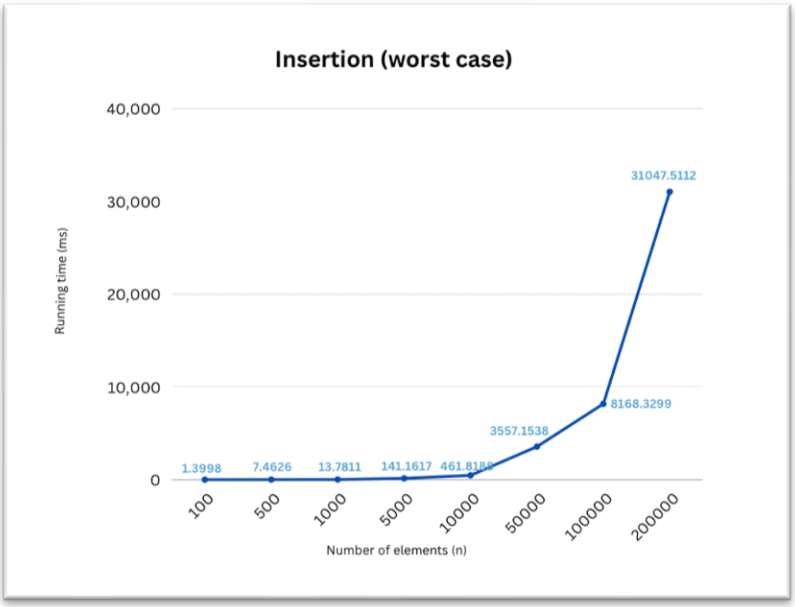

Figure 5 Insertion Sort Worst Case Graph

- **Best, worst, and average graph**

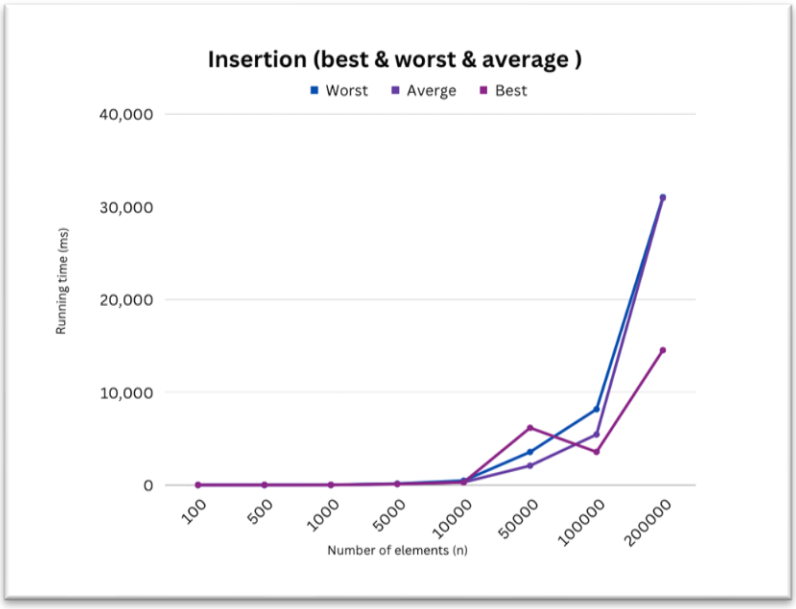

Figure 6 Insertion Sort All Cases Graph

The graphs illustrate how the insertion sort algorithm reacts to different-sized sets of numbers arranged once as best case/increasing order, worst case / decreasing order, and finally average case / random order based on the total execution time formula (Finishing time – Starting time). As shown in the graphs. Smaller sets of numbers such as 100 and 500 cannot show clear visualization of the process as their total execution times are too close to each other, unlike bigger sets such as 10,000 and 100,000, their total execution times are distanced which helps visualize how different the insertion sort algorithm reacts based on the case size and order. Furthermore, as mentioned in Table 2 it can be shown that the worst case increases significantly due to the x^2 expression.

2.2 Merge Sort Algorithm

Merge sort algorithm is a method of sorting that sequentially joins list items to order data. Each item in the first unordered list is combined with every other item to form groups of two. There is one ordered list once all two-item groups have been combined, creating groups of four and so on.

2.2.1 Merge Sort Algorithm

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Figure 7 Merge Sort Algorithm

Cases	Best Case	Average Case	Worst Case
The order of data	Data sorted increasingly.	Between best and worst case.	Data sorted decreasingly.
Time complexity	$T(N) = \emptyset(n \log n)$	$T(N) = \emptyset(n \log n)$	$T(N) = \emptyset(n \log n)$

Table 6 Merge Sort Algorithm's Cases and its Time Complexity

2.2.2 Merge Sort Code

```
class Merge{

    public Merge(int[] numbers) {

        // array before mergeSort function
        System.out.println("Before Merge Sort");
         for (int element: numbers) {
         System.out.println(element);
        }
        // merging function call
        mergeSort(numbers);

         // array after mergeSort function
        System.out.println("\n After Merge Sort:");
        for (int element: numbers) {
         System.out.println(element);
        }
    }
```

```
 public void mergeSort (int[] inputArray) {
 int inputLength = inputArray.length;

 if (inputLength < 2) { // array with 1 element //
  return;
 }
 int midIndex = inputLength / 2; // divide the length
 int[] leftHalf = new int[midIndex]; // create a suba
 int[] rightHalf = new int[inputLength - midIndex]; /
 // note; the midIndex was subtracted from inputLengt

 for (int i = 0; i < midIndex; i++) {
      leftHalf[i] = inputArray[i]; // copy from origi
 }

 for (int i = midIndex; i < inputLength; i++) {
     rightHalf[i - midIndex] = inputArray[i]; // copy
 }

mergeSort(leftHalf); // merge function call for left
mergeSort(rightHalf); // merge function call for righ
merge(inputArray, leftHalf, rightHalf); // merge two
```

```
public void merge (int[] inputArray, int[] leftHalf, int[] rightHalf) {
  int leftSize = leftHalf.length; // get left subarray size //
  int rightSize = rightHalf.length; // get right subarray size //
  int i = 0, j = 0, k = 0; // the looping keys for each array (merged,
 while (i < leftSize && j < rightSize) { // loop until the last indez /
   if (leftHalf[i] <= rightHalf[j]) { //  the element in left subarray
   inputArray[k] = leftHalf[i];  // therfore elementt is added to merge
   i++; // increment for next element in keft subarray\
   }
   else {
   inputArray[k] = rightHalf[j]; // therfore elementt is added to merge
   j++;  // increment for next element in right subarray //
   }
   k++;
 }
 while (i < leftSize) { // if there are no eleemnts in left subarray we
  inputArray[k] = leftHalf[i];
  i++;
  k++;
  }
 while (j < rightSize) { // if there are no eleemnts in right subarray
  inputArray[k] = rightHalf[j];
  j++;
  k++;
 }
```

Figure 8 Merge Sort Code

2.2.3 Merge Sort Theoretical and Practical Analyze

Cases	Best Case		Average Case		Worst case	
Input size (n)	Theoretical Time	Practical Time (Actual Time)	Theoretical Time	Practical Time (Actual Time)	Theoretical Time	Practical Time (Actual Time)
100	200	2.914600ms	200	1.204600ms	200	1.134200ms
500	1349.485002	5.843700ms	1349.485002	5.205600ms	1349.485002	4.404100ms
1000	3000	20.230300ms	3000	7.754600ms	3000	11.043600ms
5000	18494.85002	261.885600ms	18494.85002	277.852000ms	18494.85002	293.611700ms
10000	40000	1895.638400 ms	40000	620.920700ms	40000	2371.687600 ms
50000	234948.5002	3568.189700 ms	234948.5002	3372.414600 ms	234948.5002	3429.722200 ms
100000	500000	5037.861800 ms	500000	5130.939100 ms	500000	5202.545700 ms
200000	1060205.999 132796	15002.870700 ms	1060205.999 132796	14316.786600ms	1060205.999 132796	14565.421800ms

Table 7 Merge Sort Algorithm's Running Time Cases

Case Input size (n)	Best case $\frac{P}{T} \times 100$	Average Case $\frac{P}{T} \times 100$	Worst Case $\frac{P}{T} \times 100$
100	1.4573%	0.6%	0.6%
500	0.43%	0.38%	0.3%
1000	0.67%	0.26%	0.4%
5000	1.4%	1.5%	1.6%
10000	4.7%	1.5%	5.9%
50000	1.5%	1.4%	1.5%
100000	1%	1%	1%
200000	1.41%	1.35%	1.37%

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

2.2.4 Merge Sort Algorithm Comparison Graph (Theoretical & Practical Time Values)

- **Best case graph**

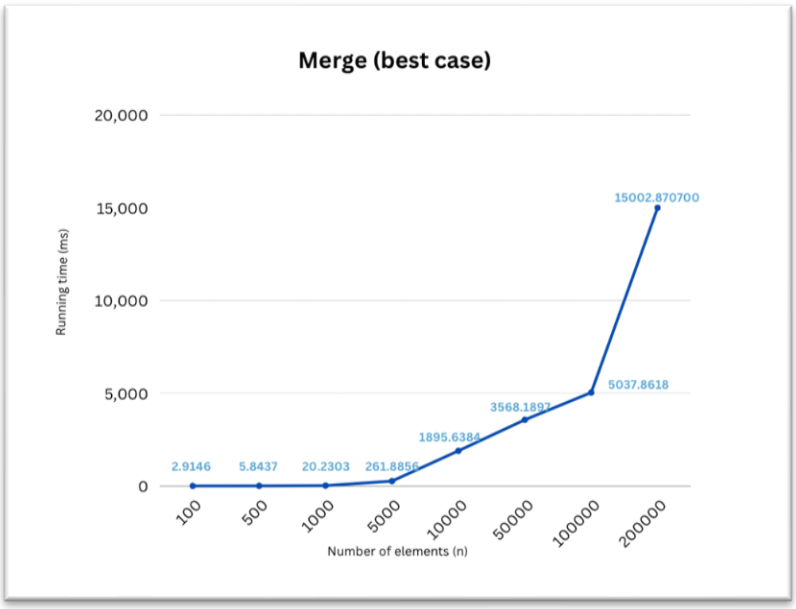

Figure 9 Merge Sort Best Case

- **Average case graph**

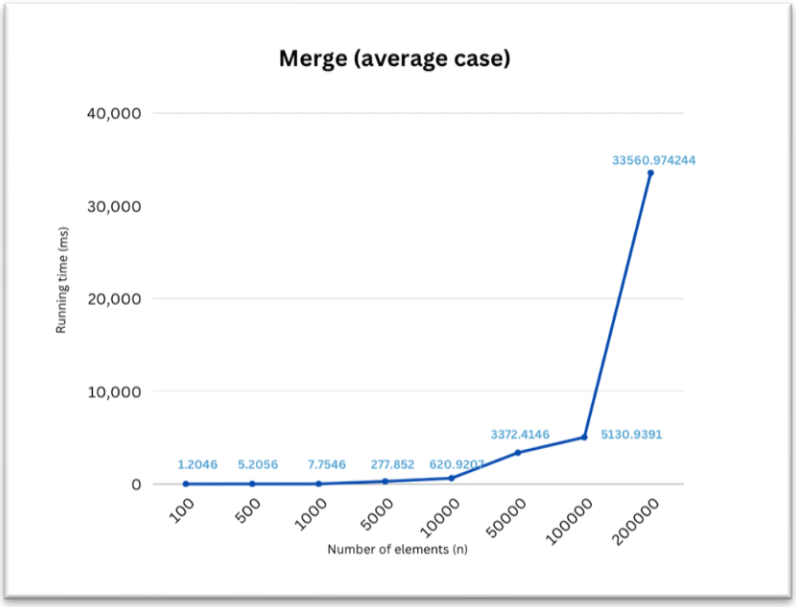

Figure 10 Merge Sort Average Case

- **Worst case graph**

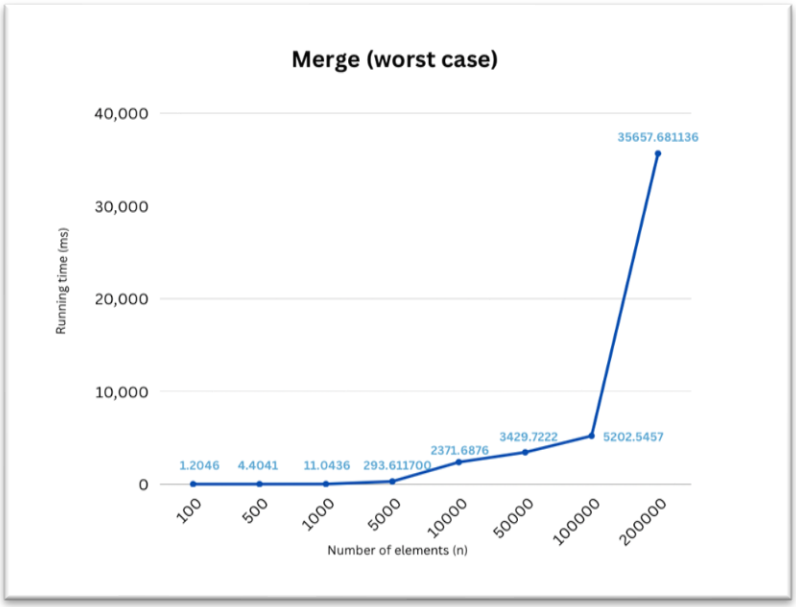

Figure 11 Merge Sort Average Case

- **Best, worst, and average graph**

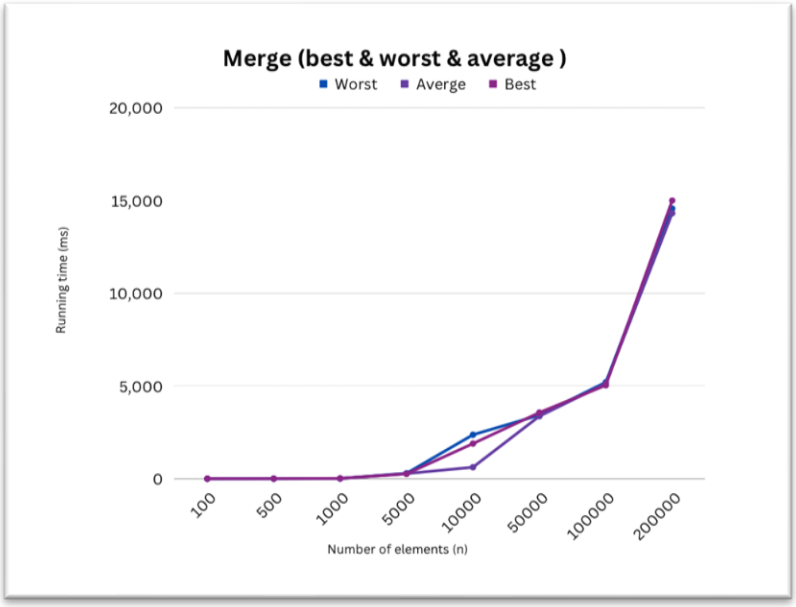

Figure 12 Merge Sort Best, Worst, Average Case

The graphs illustrate how the merge sort algorithm reacts to different-sized sets of numbers arranged once as best case/increasing order, worst case /decreasing order, and finally average case / random order based on the total execution time formula (Finishing time – Starting time). As shown in the graphs. Smaller sets of numbers such as 100 and 500 cannot show clear visualization of the process as their total execution times are too close to each other, unlike bigger sets such as (10,000 to 200,000) their total execution times are approximately the same which helps visualize that the merge sort algorithm for the best, worst and average cases are approximately equal based on the case size and order.

2.3 Heap Sort Algorithm

Heap sorting uses comparisons to sort data and is based on the Binary Heap data structure. The minimum element is first identified, and it is then positioned at the start. The remaining elements should be treated similarly. A built-in algorithm is heap sort.

2.3.1 Heap Sort Algorithm

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3       MAX-HEAPIFY(A, i)
```

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3       exchange A[1] with A[i]
4       A.heap-size = A.heap-size − 1
5       MAX-HEAPIFY(A, 1)
```

Figure 13 Heap Sort Algorithm

Cases	Best Case	Average Case	Worst Case
The order of data	Data sorted increasingly.	Between best and worst case.	Data sorted decreasingly.
Time complexity	$T(N) = \emptyset(n \log n)$	$T(N) = \emptyset(n \log n)$	$T(N) = \emptyset(n \log n)$

Table 9 Heap Sort Algorithm's Cases and its Time Complexity

2.3.2 Heap Sort Code

```java
public void sort(int arr[])
{
    int N = arr.length;

    // Build heap (rearrange array)
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);

    // One by one extract an element from heap
    for (int i = N - 1; i > 0; i--) {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

```java
class Heap{

    public Heap(int[] numbers) {

    // Print the unsorted array //
    System.out.println("Before Heap Sort:");
    printArray(numbers);

        // Function call
        sort(numbers);

        System.out.println("After Heap Sort");
        printArray(numbers);
```

```java
    void heapify(int arr[], int N, int i)
    {
        int largest = i; // Initialize largest as root
        int l = 2 * i + 1; // left = 2*i + 1
        int r = 2 * i + 2; // right = 2*i + 2

        // If left child is larger than root
        if (l < N && arr[l] > arr[largest])
            largest = l;

        // If right child is larger than largest so far
        if (r < N && arr[r] > arr[largest])
            largest = r;

        // If largest is not root
        if (largest != i) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

            // Recursively heapify the affected sub-tree
            heapify(arr, N, largest);
        }
    }
```

```java
/* A utility function to print array of size n */
static void printArray(int arr[])
{
    int N = arr.length;

    for (int i = 0; i < N; ++i)
        System.out.println(arr[i]);
    System.out.println();
}
```

Figure 14 Heap Sort Code

2.3.3 Heap Sort Theoretical and Practical Analyze

Cases	Best Case		Average Case		Worst case	
Input size (n)	Theoretical Time	Practical Time (Actual Time)	Theoretical Time	Practical Time (Actual Time)	Theoretical Time	Practical Time (Actual Time)
100	200	2.980600ms	200	1.510900ms	200	1.232500 ms
500	1349.485002	5.184900ms	1349.485002	4.342900ms	1349.485002	3.955500 ms
1000	3000	8.223900ms	3000	7.888900ms	3000	9.698800 ms
5000	18494.85002	287.877300ms	18494.85002	292.063700ms	18494.85002	334.993900ms
10000	40000	982.057600ms	40000	3089.529200 ms	40000	599.646400ms
50000	234948.5002	3800.142400 ms	234948.5002	3858.311600 ms	234948.5002	3911.060300ms
100000	500000	5121.981100 ms	500000	5102.574600 ms	500000	5165.564800ms
200000	1060205.999132796	14956.240100 ms	1060205.999132796	13809.935800ms	1060205.999132796	13624.675100ms

Table 10 Heap Sort Algorithm's Running Time Cases

Case	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	1.49%	0.76%	0.62%
500	0.38%	0.32%	0.29%
1000	0.27%	0.26%	0.32%
5000	1.55%	1.58%	1.81%
10000	2.45%	7.72%	9%
50000	1.62%	1.64%	1.66%
100000	1.02%	1.02%	1.03%
200000	1.4%	1.3%	1.2%

Table 11 Error Ratio for the Heap Sort Algorithm's cases

2.3.4 Heap Sort Algorithm Comparison Graph (Theoretical & Practical Time Values)

- **Best case graph**

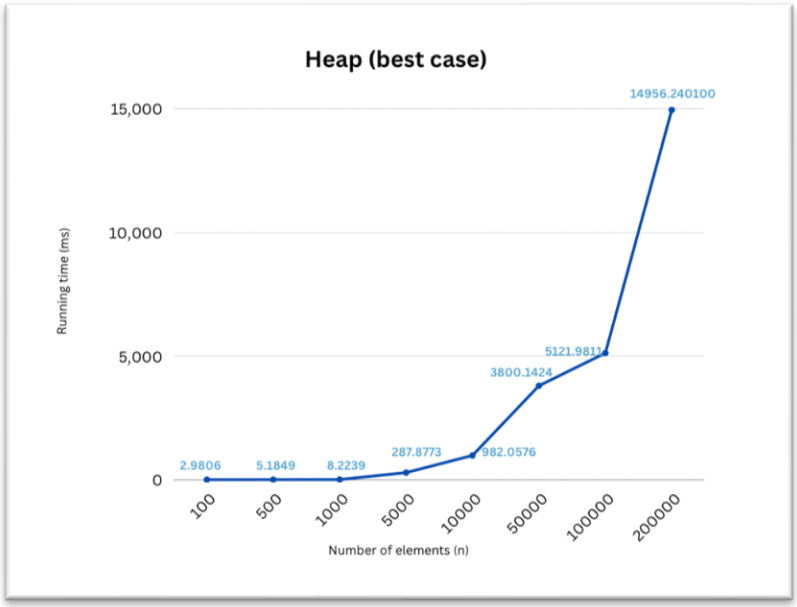

Figure 15 Heap Sort Best Case

- **Average case graph**

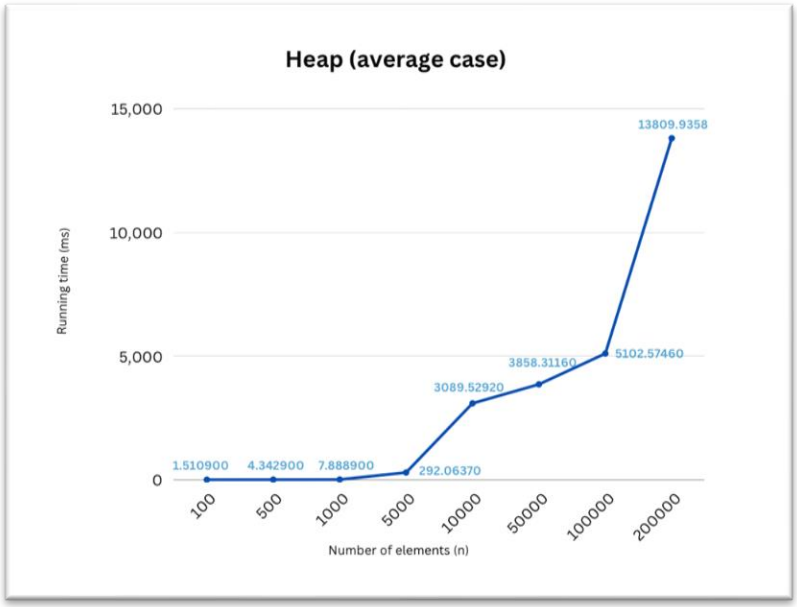

Figure 16 Heap Sort Average Case

- **Worst case graph**

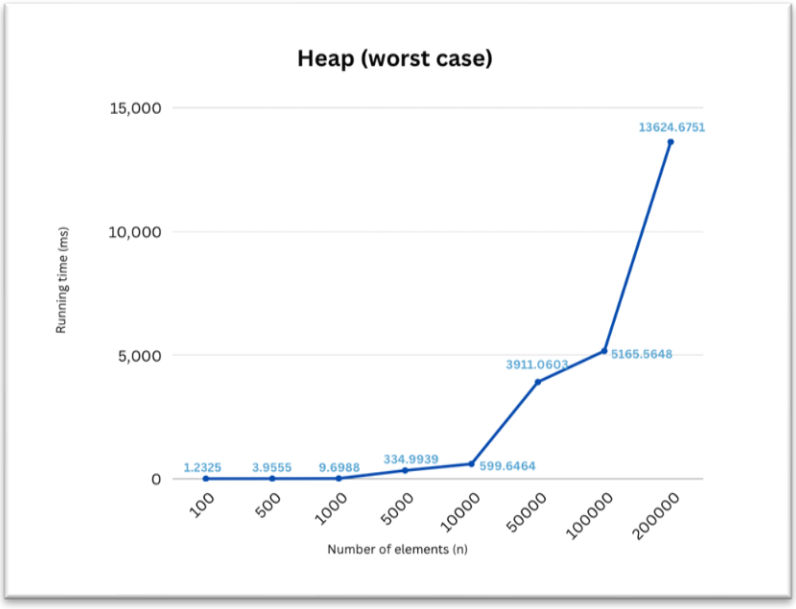

Figure 17 Heap Sort worst Case

- **Best, worst, and average graph**

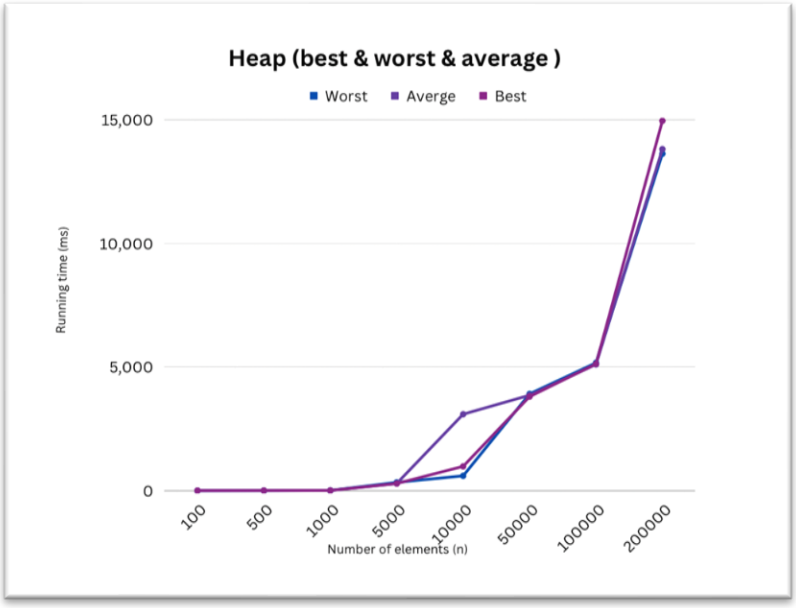

Figure 18 Heap Sort Best, Worst, Average Case

The graphs illustrate how the heap sort algorithm reacts to different-sized sets of numbers arranged once as best case/increasing order, worst case /decreasing order, and finally average case / random order based on the total execution time formula (Finishing time – Starting time). As shown in the graphs. Smaller sets of numbers such as 100 and 500 cannot show clear visualization of the process as their total execution times are too close to each other, unlike bigger sets such as 10,000 and 50,000, their total execution times are approximately the same which helps visualizing that the heap sort algorithm for the best, worst and average cases are approximately equal based on the case size and order.

2.4 Algorithms Analysis

2.4.1 Sorting Algorithms Comparison Table

Sort type	**Insertion Sort**			**Merge Sort**			**Heap Sort**		
case	Best case	Average Case	Worst Case	Best case	Average Case	Worst Case	Best case	Average Case	Worst Case
	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$	$\frac{P}{T} \times 100$
100	3.5%	0.019%	0.014%	1.4573%	0.6%	0.6%	1.49%	0.76%	0.62%
500	0.8%	0.0017%	0.0029%	0.43%	0.38%	0.3%	0.38%	0.32%	0.29%
1000	0.5%	0.00079%	0.0014%	0.67%	0.26%	0.4%	0.27%	0.26%	0.32%
5000	1.9%	0.00042%	0.00056%	1.4%	1.5%	1.6%	1.55%	1.58%	1.81%
10000	2.9%	0.00025%	0.00046%	4.7%	1.5%	5.9%	2.45%	7.72%	9%
50000	12.3%	0.000083%	0.00014%	1.5%	1.4%	1.5%	1.62%	1.64%	1.66%
100000	3.5%	0.000054%	0.000082%	1%	1%	1%	1.02%	1.02%	1.03%
200000	7.5%	0.000077%	0.00000077%	1.35%	1.35%	1.37%	1.3%	1.3%	1.2%

Table 12 Sorting Algorithms Comparison

2.4.2 Sorting Algorithm Comparison Graph

The figures below present the graphs for each best case, average case, and worst case of different input size. It is found that the best performance for the best case is insertion sort and for both average and worst cases merge sort and heap sort have close results.

- **Best case graph for all sorting algorithms**

  Figure 19 below represents the best cases of each sorting algorithms which are insertion sort, merge sort and heap sort. The best performance of the four sorts in the best case is insertion sort.

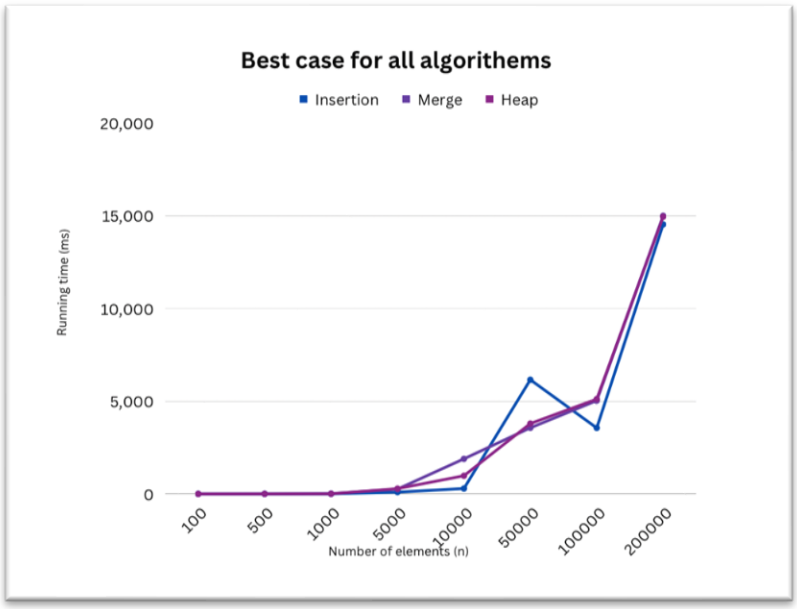

Figure 19 Best Case for All Algorithms

- **Average case graph for all sorting algorithms**

  The **Figure 20** below represents the average cases of each sorting algorithms which are insertion sort, merge sort and heap sort. When it comes to the best performance, both merge sort and heap sort have close results because their complexity is the same, while the insertion sort has the worst performance due to its high growth.

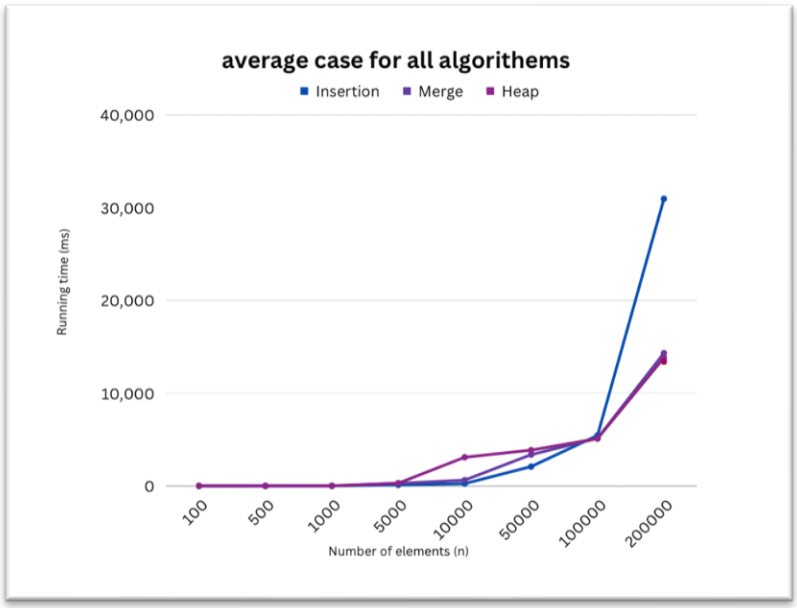

Figure 20 Average Case for All Algorithms

- **Worst case graph for all sorting algorithms**

  The **Figure 21** below represents the worst cases of each sorting algorithms which are insertion sort, merge sort and heap sort. When it comes to the best performance, both merge sort and heap sort have close results because their complexity is the same, while the insertion sort has the worst performance due to its high growth.

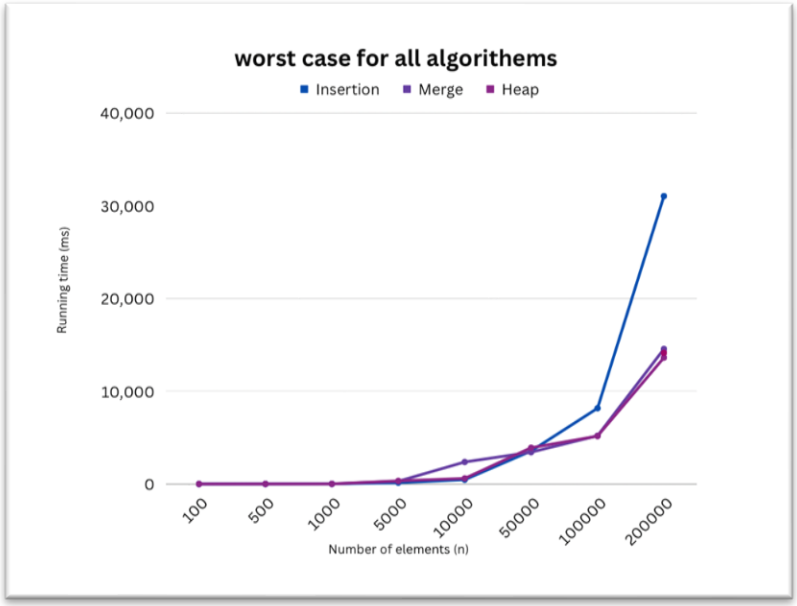

Figure 21 Worst Case for All Algorithms

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

By comparing the theoretical running time (T) with the practical running time (P), divide them to find the difference rate (P/T * 100) and finally draw the graphs. (Comparison graphs for all sorting algorithms have been well implemented in the previous sections).

2.4.4 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 multiple input sizes with the same numbers among all three sorting algorithms (100, 500, 1000, 5000, 10000, 50000, 100000, 200000), all will be executed and compared in the case of increasing, random, and decreasing order for the algorithm itself and for all. Using the same input for all sorting algorithms will give correct and accurate results for the execution time.

- **The Number of Comparisons and the Execution Time for Insertion Sort:**

The table below shows the number of comparisons performed during the sorting process using the equation ($n^2/2$) and the execution time calculated above using insertion sort programmed code. And the graph shows a visual representation of the values.

Input size (n)	Number of Comparisons	Execution Time
100	5000	1.399800ms
500	125000	7.462600ms
1000	500000	13.781100ms
5000	12500000	141.161700ms
10000	50000000	461.818800ms
50000	1250000000	3557.153800ms
100000	5000000000	8168.329900ms
200000	20000000000	31047.511200 ms

Table 13 Number of Comparisons and the Execution Time for Insertion Sort

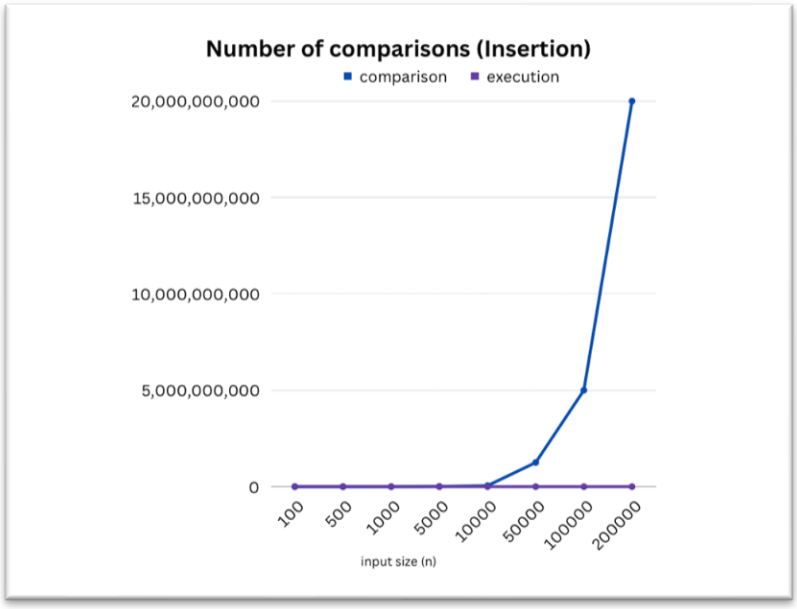

Figure 22 Number of Comparison for Insertion Sort

The graph represents the relationship between the number of comparison and the execution time of the insertion sort for each input size. As shown in the graph, the number of comparisons has faster growth than the execution time.

- **The Number of Comparisons and the Execution Time for Merge Sort:**

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

Input size (n)	Number of Comparisons	Execution Time
100	664.4	1.134200ms
500	4483	4.404100ms
1000	9965.8	11.043600ms
5000	61438.6	293.611700ms
10000	132877.1	2371.687600ms
50000	780482	3429.722200ms
100000	1660964	5202.545700ms
200000	3521928	14565.421800ms

Table 14 Number of Comparisons and the Execution Time for Merge Sort

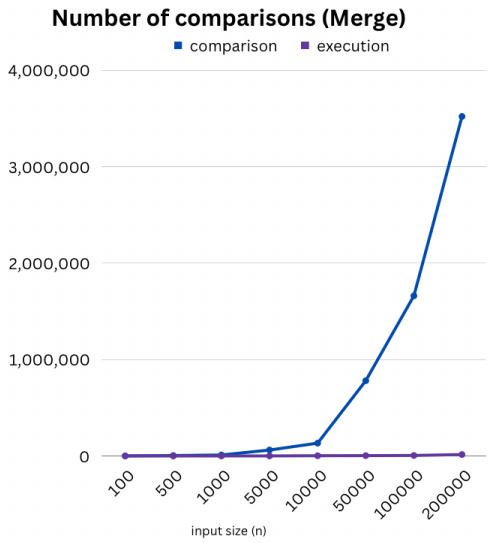

Figure 23 Number of Comparisons for Merge Sort

The graph represents the relationship between the number of comparison and the execution time of the merge sort for each input size. As shown in the graph, the number of comparisons has faster growth than the execution time.

- **The Number of Comparisons and the Execution Time for Heap Sort:**

The table below shows the Number of Comparisons performed during the sorting process using the equation (n log n) and the execution time calculated above using heap sort programmed code. And the graph shows a visual representation of the values.

Input size (n)	Number of Comparisons	Execution Time
100	664.4	1.232500ms
500	4483	3.955500ms
1000	9965.8	9.698800ms
5000	61438.6	334.993900ms
10000	132877.1	599.646400ms
50000	780482	3911.060300ms
100000	1660964	5165.564800ms
200000	3521928	13624.675100ms

Table 15 Number of Comparisons and the Execution Time for Heap Sort

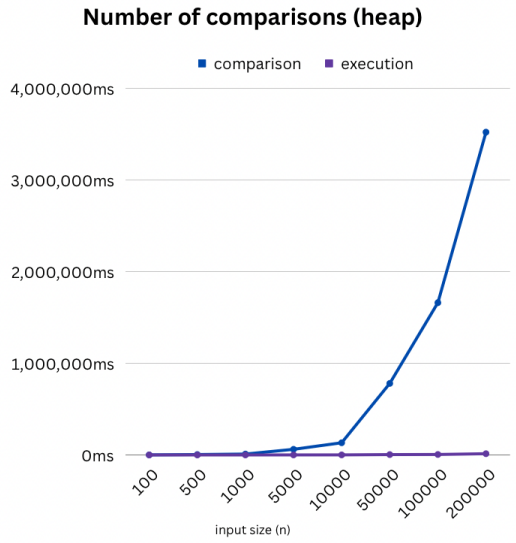

Figure 24 Number of Comparisons for Heap Sort

The graph represents the relationship between the number of comparison and the execution time of the heap sort for each input size. As shown in the graph, the number of comparisons has faster growth than the execution time.

Kingdom of Saudi Arabia
Ministry of Education
Imam Abdulrahman bin Faisal University
College of Science and Humanities in Jubail
Computer Science Department

المملكة العربية السعودية
وزارة التعليم
جامعة الإمام عبدالرحمن بن فيصل
كلية العلوم والدراسات الإنسانية بالجبيل
قسم علوم الحاسب

PART 2

3 PROBLEM SOLVING AND ANALYSIS

Design and analysis an improved Divide-and-conquer algorithm compute a^n , where n is natural number. Given two integers a and n, an algorithm is written to solve this problem and give the best running time possible.

3.1 Algorithm

- **3.1.1 Practical code in java (Improved Divide-And-Conquer):**

```
1-  static int power (int a, int n) {
2-   int temp;
3-   if (n == 0)
4-        return 1;
5-   temp = power (a, n / 2);
6-   if (n % 2 == 0)
7-        return temp * temp;
8-   else
9-        return a * temp * temp;}
```

3.1.2 Pseudocode

- Improved Divide-And-Conquer

```
Power (a.n)
If n= 0
Return 1
Let I = Power(an/2)
I 1%2=0
Return I°I|I
Else
Return a*I*I
```

3.1.3 Running time analysis

Improved Divide-And-Conquer:

- The improved algorithm for Divide-and-conquer would yield a running time of **O ($\log_2$ n)**, a logarithmic time complexity as shown in Table 6. By calculating each step of the algorithm.

T(n)= (1) + (1) + (1) + T(n/2) + (1) + (1) + (1) + (1) = O ($\log_2$ n)
T(n)= (log n) + (1) = O (log2 n)

Divide-And-Conquer:

- The algorithm for Divide-and-conquer would yield a running time of **O (n)**, a linear time complexity as shown in Table 6. By calculating each step of the algorithm.

T(n)= (1) + (1) + (1) + 2T(n/2) + (1) + 2T(n/2) = O (n)
T(n)= (1) + (log n) + (log n) = O (n)

improved Divide-and-conquer Steps	**Running time**	**Divide-and-conquer Steps**	**Running time**
static int power(int a, int n)	T(n)	**static int** power(**int** x, **int** y)	T(n)
int temp;	O(1)	**if** (y == 0)	O(1)
if (n == 0)	O(1)	**return** 1;	O(1)
return 1;	O(1)	**else if** (y % 2 == 0)	O(1)
temp = power(a, n / 2);	T(n/2)	**return** power(x, y / 2) * power(x, y / 2);	2T(n/2)
if (n % 2 == 0)	O(1)	**else**	O(1)
return temp * temp;	O(1)	**return** x * power(x, y / 2) * power(x, y / 2);	2T(n/2)
else	O(1)		
return a * temp * temp;	O(1)		
Recurrence relation	T(n)=T(n/2)+1	**Recurrence relation**	T(n)=2T(n/2)+1

Table 16 Running time analysis for the given problem in part 2

4 REFERENCES

1. *Introduction to algorithms, 3rd Edition (the MIT Press) 3rd edition* (no date).
2. Palak (2020) *Heaps and heap sort algorithm*, *Medium*. Medium. Available at: https://palak001.medium.com/heaps-and-heap-sort-algorithm-c6e96b16fc23 (Accessed: October 17, 2022).
3. "Heap sort," *GeeksforGeeks*, 22-Sep-2022. [Online]. Available: https://www.geeksforgeeks.org/heap-sort/. [Accessed: 17-Oct-2022].
4. *Insertion sort*. GeeksforGeeks. (2022, July 13). Retrieved October 17, 2022, from https://www.geeksforgeeks.org/insertion-sort/
5. *Merge sort algorithm*. GeeksforGeeks. (2022, September 23). Retrieved October 17, 2022, from https://www.geeksforgeeks.org/merge-sort/
6. *Quicksort*. GeeksforGeeks. (2022, September 27). Retrieved October 17, 2022, from https://www.geeksforgeeks.org/quick-sort/?ref=gcse
7. *Write program to calculate pow(x, n)*. Available at: https://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/amp/ (Accessed: October 31, 2022).
8. Invisibly (2022) 10 algorithm examples used in your daily life, Home. Available at: https://www.invisibly.com/learn-blog/algorithm-examples-everyday-life/ (Accessed: November 3, 2022).
9. Team, G.L. (2022) Insertion sort with a real-world example, Great Learning Blog: Free Resources what Matters to shape your Career! Available at: https://www.mygreatlearning.com/blog/insertion-sort-with-a-real-world-example/ (Accessed: November 3, 2022).