****

**CZ2001 Algorithms Example Class 3A**

**Report**

**SSP4 Team 3**

**Tan Wei Xuan (U1721260J)**

**Datta Anusha (U1822948G)**

**Prashant Mohit (U1823630E)**

**Han Jun (U1820665L)**

**Ho Shu Peng Xavier (U1822670E)**

**Kumar Mehul (U1822146E)**

**Wang Yifan (U1821102B)**

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY

Contents

[1. INTRODUCTION 3](#_Toc22640138)

[1.1 Description of Problem 3](#_Toc22640139)

[2. IMPLEMENTATION 3](#_Toc22640140)

[2.1 Generating Input Data 3](#_Toc22640141)

[2.1.1 Integers (1,2, …, n) sorted in ascending order 3](#_Toc22640142)

[2.1.2 Integers (n, n-1, …, 1) in Descending Order 4](#_Toc22640143)

[2.1.3 Randomly generated datasets of Integers in the range [1 … n] 4](#_Toc22640144)

[2.2 Insertion Sort 5](#_Toc22640145)

[2.3 Merge Sort 6](#_Toc22640146)

[3. STATISTICS 7](#_Toc22640147)

[3.1 Insertion Sort 7](#_Toc22640148)

[3.2 Merge Sort 7](#_Toc22640149)

[3.3 Graphs 8](#_Toc22640150)

[4. ANALYSIS 13](#_Toc22640151)

[4.1 Theoretical Analysis 13](#_Toc22640152)

[4.2 Empirical Analysis 13](#_Toc22640153)

# 1. INTRODUCTION

*Description of the problem domain and data sets*

## 1.1 Description of Problem

In this example class, we will be showing the empirical comparison of time efficiency, both in terms of number of key comparisons and CPU Execution time for the two soring algorithms – Insertion Sort and Merge Sort. We will be performing these two algorithms on data sets with sizes ranging from 100 to 10000, with a step size of 100. For each of the data set sizes (n), the following types of data will be generated:

1) Integers 1, 2, …, n sorted in **ascending order (Ascending List)**

2) Integers n, n-1, …, 1 sorted in **descending order (Descending List)**

3) **Randomly generated** datasets of integers in the range [1 … n].

# 2. IMPLEMENTATION

## 2.1 Generating Input Data

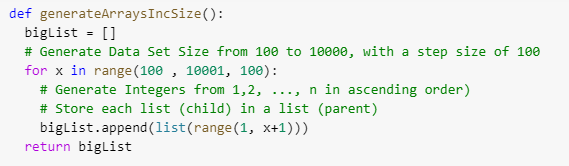
Fig 1 below illustrates how we generate data sets with size ranging from 100 to 1000, with a step size of 100. For each of the data set size (n), integers ranging from 1,2, …, n will be **generated in ascending order**, without any repeats, by default.

Figure 1 Function to generate different data set sizes with data in ascending order

### 2.1.1 Integers (1,2, …, n) sorted in ascending order

Fig 2 below illustrates how we generate the first type of data (Integers from 1 to n) for all the data set sizes.

Figure 2 Generating the Ascending List of Integers

We simply run the function *(generateArraysIncSize)* as shown in Figure 1

### 2.1.2 Integers (n, n-1, …, 1) in Descending Order

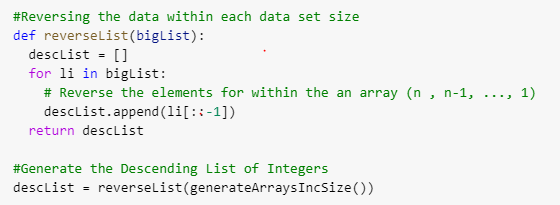
Fig 3 below illustrates how we generate the second type of data (Integers from n to 1 in descending order) for all the data set sizes.

Figure 3 Generating the Descending List of Integers

We first run first the function (*(generateArraysIncSize)* as shown in Figure 1 previously. With the data being generated from that function, we run the function (*reverseList)* as shown in Figure 3 to reverse the order data within each of the data set size.

### 2.1.3 Randomly generated datasets of Integers in the range [1 … n]

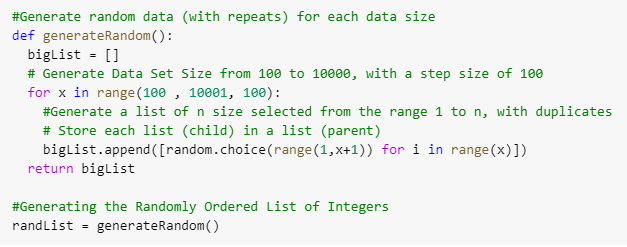
Figure 4 below illustrates how we generate the third type of data (Randomly generated datasets of integers from range 1 to n, with duplicates) for all the data sets sizes.

Figure 4 Generating the Randomly Ordered List of Integers (With Duplicates)

## 2.2 Insertion Sort

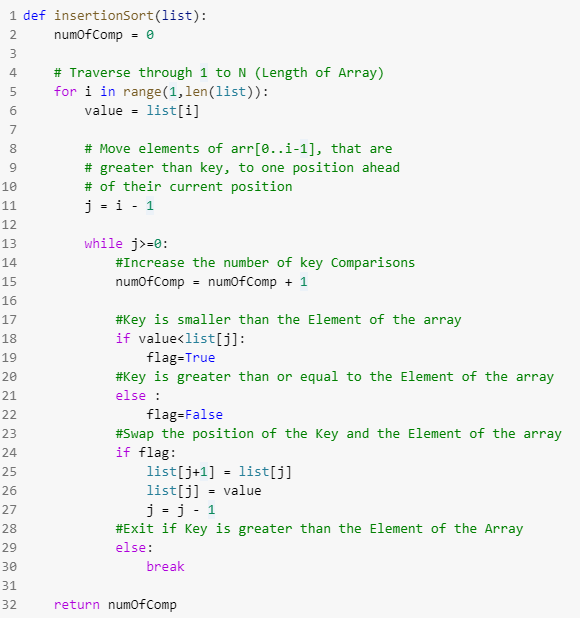
The Insertion Sort algorithm takes elements from the input array and places those elements in their correct place into a new array, shifting existing array elements as needed. Figure 5 below shows how we implement the Insertion Sort Algorithm. 

Figure 5 Insertion Sort Algorithm

For each of the data set size, we iterate through all the array elements by growing the sorted array at each iteration. At each iteration, we compare the current element with the elements in front of it. If the current element is smaller in value, it swaps its position with the element in front of it and it will continuously do this until either the current element is greater in value than the element in front of it, or it reaches the front of the array (lines 5 - 30).

## 2.3 Merge Sort

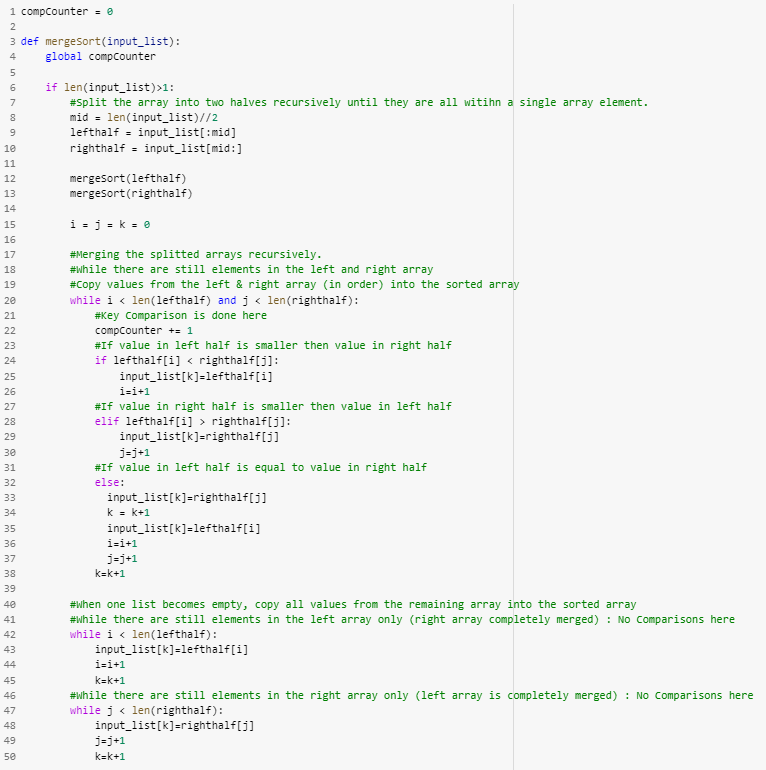
The Merge Sort algorithm is a divide-and-conquer algorithm. It takes input of an array and divides that array into sub arrays of single elements. A single element is already sorted, and so the elements are sorted back into sorted arrays, two sub-arrays at a time, until we are left with a final sorted array. Figure 6 below shows how we implement the Merge Sort Algorithm.  
For each of the data set size, the data set is broken up into a left half and right half, and the two halves are divided recursively until they are all within a single array element (lines 8 – 13). Then, the two halves’ elements are compared to determine how the two arrays should be arranged (lines 20 – 38). Should any one half contain elements not added to the sorted array after the comparisons are made, the remainder is added so no elements are lost (lines 42 - 50).

Figure 6 Merge Sort Algorithm

# 3. STATISTICS

The statistical results (number of key comparisons and CPU time) from using both Insertion and Merge Sort on the data sets (10 times for each of the data set sizes) of the three different types of data. Each data set size was run through the two algorithms 10 times and the number of key comparisons and average CPU Time for Data Set Size of [2000, 4000, 6000, 8000, 10000] are being recorded (as shown below).

## 3.1 Insertion Sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data Size | Ascending List | | Descending List | | Random List | |
| Number of Key Comparisons | Average CPU time (milliseconds) | Number of Key Comparisons | Average CPU time (milliseconds) | Number of Key Comparisons | Average CPU time (milliseconds) |
| 2000 | 1999 | 0.341364 | 1999000 | 376.156034 | 984080 | 184.718464 |
| 4000 | 3999 | 0.6715942 | 7998000 | 1542.534218 | 4136779 | 783.496369 |
| 6000 | 5999 | 1.0156708 | 17997000 | 3483.914153 | 8981658 | 1733.694997 |
| 8000 | 7999 | 1.3397 | 31996000 | 6182.961007 | 15939819 | 3055.641993 |
| 10000 | 9999 | 1.6526552 | 49995000 | 9745.170913 | 24893599 | 4754.887758 |

## 3.2 Merge Sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data Size | Ascending List | | Descending List | | Random List | |
| Number of Key Comparisons | Average CPU time (milliseconds) | Number of Key Comparisons | Average CPU time (milliseconds) | Number of Key Comparisons | Average CPU time (milliseconds) |
| 2000 | 10864 | 6.139191 | 11088 | 6.758364 | 18677 | 13.370617 |
| 4000 | 23728 | 13.673324 | 24176 | 14.729216 | 41337 | 18.656129 |
| 6000 | 36656 | 22.690663 | 39152 | 22.922517 | 65493 | 29.459557 |
| 8000 | 51456 | 28.212278 | 52353 | 31.034882 | 90643 | 39.173419 |
| 10000 | 64608 | 36.527133 | 69008 | 39.963441 | 116719 | 50.33941 |

## 3.3 Graphs

Figure 7 (Insertion Sort) Array Size vs No. of Key Comparisons



Figure 8 (Insertion Sort) Array Size vs CPU Execution Time (ms)

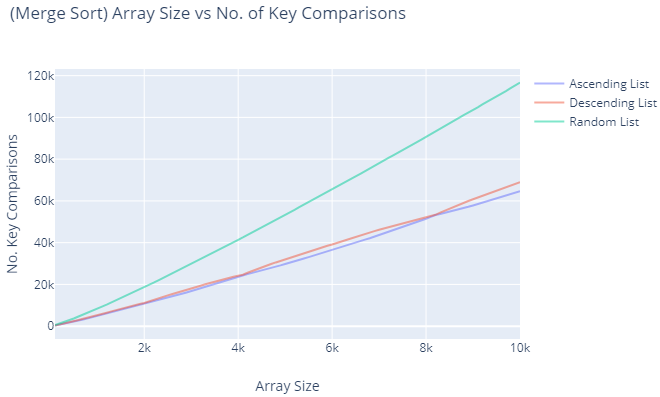
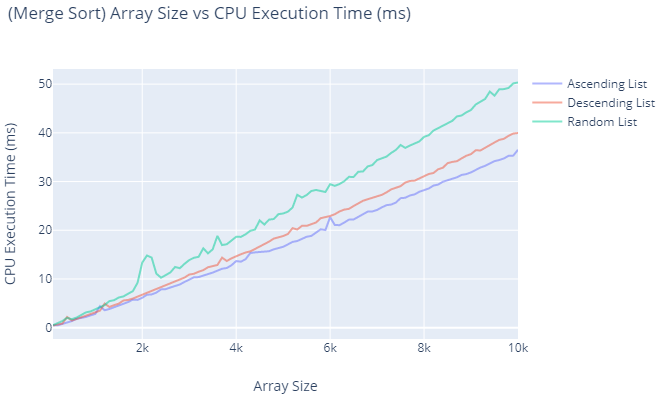


Figure 9 (Merge Sort) Array Size vs No. of Key Comparisons

Figure 10 (Merge Sort) Array Size vs CPU Execution Time (ms)

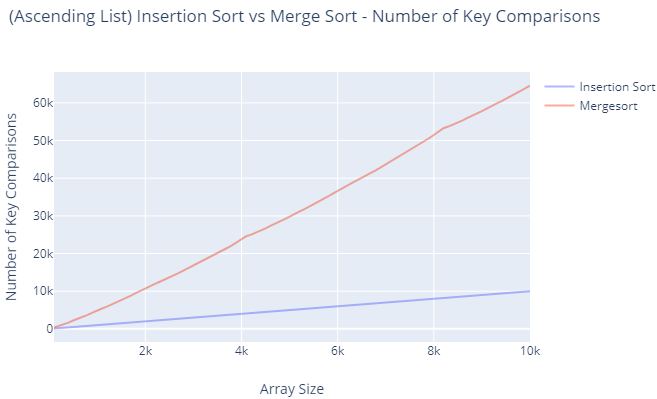
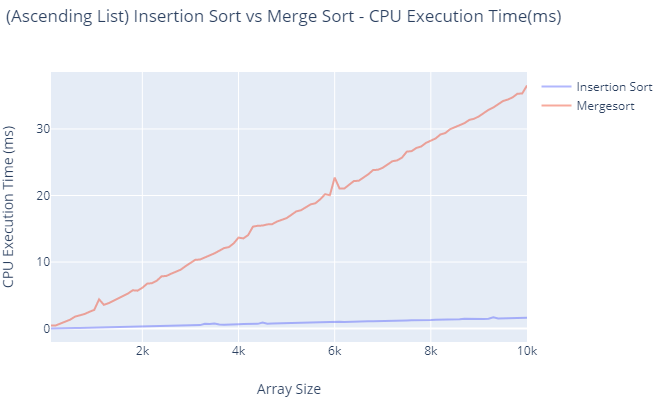


Figure 12 (Ascending List) Insertion Sort vs Merge Sort – CPU Execution Time (ms)

Figure 11 (Ascending List) Insertion Sort vs Merge Sort - No. of Key Comparisons

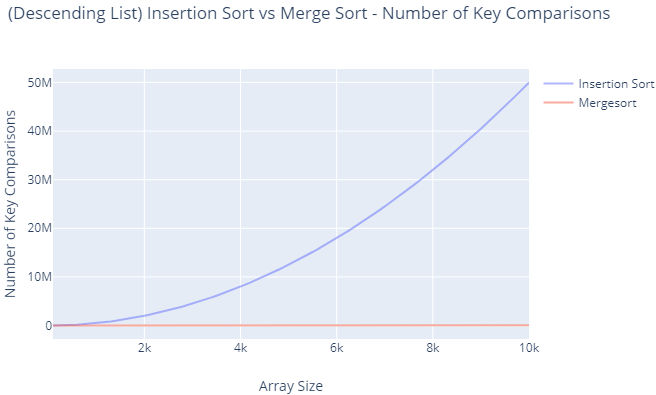
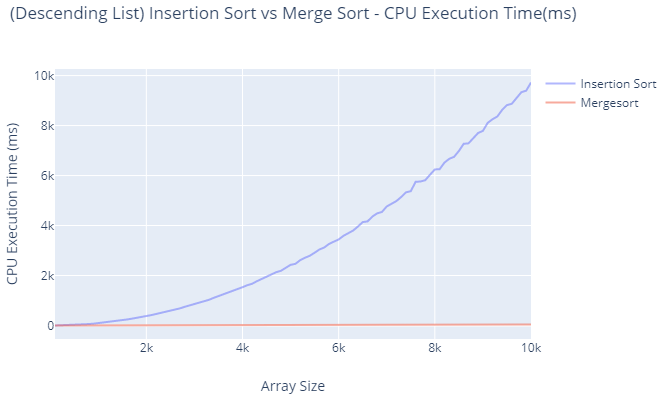


Figure 14 (Descending List) Insertion Sort vs Merge Sort – CPU Execution Time (ms)

Figure 13 (Descending List) Insertion Sort vs Merge Sort - No. of Key Comparisons

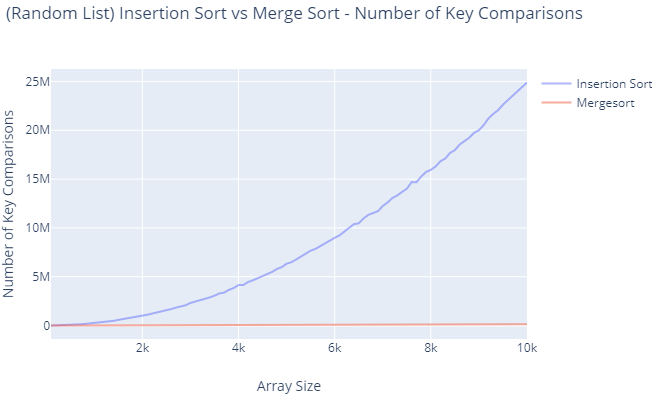
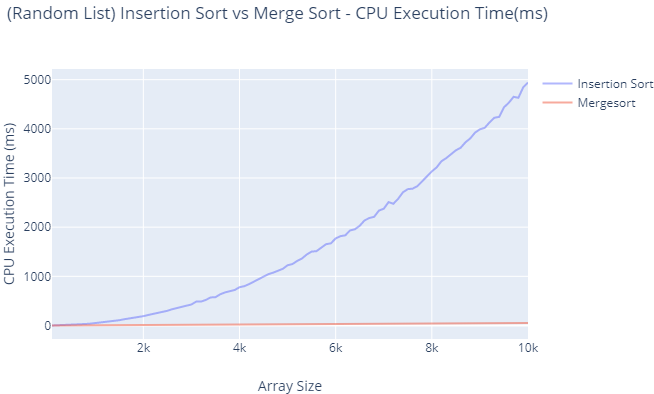


Figure 16 (Random List) Insertion Sort vs Merge Sort – CPU Execution Time (ms)

Figure 15 (Random List) Insertion Sort vs Merge Sort - No. of Key Comparisons

# 4. Analysis

## 4.1 Theoretical Analysis

For Insertion Sort, **best case** occurs when the input array is already sorted. In this scenario, each element will only be compared to its immediate predecessor. The number of comparisons required will be **n-1** as comparison is made from the 2nd element onwards till the last element. This results in a Big O time complexity of).

Insertion Sort’s **worst case** occurs when the input array is in descending order. In this case every iteration of the array elements will scan and shift the entire sorted region of the array before inserting the next element. The number of elements required will be . This results in a Big O time complexity of

For Merge Sort, the **best case** occurs when the largest element of one array is smaller than any element in the other. In this scenario, only ­ comparisons of array elements are made for recursive call of the merge step, giving the recurrence equation, . This results in a Big O Time complexity of .

Merge Sort’s **worst case** occurs when during each recursive call of the merge step, the two largest elements are located in different arrays. This forces the maximum number of comparisons to occur, giving the recurrence equation, . Though there are more comparisons being made, the overall Big O Time complexity is still .

## 4.2 Empirical Analysis

Based on the empirical results that we have obtained, from Fig 8, it can be seen using **Insertion Sort** on an array that is in ascending order (already sorted) is the fastest (best case of Insertion Sort). The time taken to sort an array in descending order is the slowest (worst case of Insertion Sort). For an array of random elements, certain elements in the array are either already sorted or do not require to scan and shift through the entire sorted region of the array. Though it is still slower than an array that is in ascending order, it is faster than an array in descending order. These results align with the theoretical analysis of the Insertion Sort algorithm.

From Figure 10, using **Merge Sort** of an array that is in descending and ascending order takes about the same time to run This is because for an array that is either in descending or ascending order, at every recursive call of the merge step, the largest element of one array will be smaller than any element in the order. This results in the Best Case of Merge Sort. For an array of random elements, there can be a chance that at each recursive of the merge step, the two largest elements are located in different arrays, and thus forcing a larger number of comparisons to occur. As such, using Merge Sort on an array of random elements will be slower than one that is in descending or ascending order. These results align with the theoretical analysis of the Merge Sort algorithm.

From Figure 12, it can be seen that the **Insertion Sort Algorithm takes much lesser time** to run on an **array in ascending order** than the Merge Sort Algorithm. This is because for an array in ascending order, the time complexity of running the Insertion Sort Algorithm will only be , which is much smaller than the time complexity of of running the Merge Sort Algorithm

From Figure 14, it can be seen that the **Merge Sort Algorithm takes much lesser time** to run on an array in **descending order** than the Insertion Sort Algorithm. This is because for an array in descending order, the time complexity of running the Merge Sort Algorithm will only be, which is much smaller than the time complexity of of running the Insertion Sort Algorithm

From Figure 16, similarly, it can be seen that the **Merge Sort Algorithm takes much lesser time** to run on an **array with random elements** than the Insertion Sort Algorithm. However, it is to note that there can exist a case whereby the array consists of elements that are already almost sorted. This will result in the Insertion Sort algorithm running faster than the Merge Sort Algorithm.

As such, we can conclude that:

1. **Insertion Sort** should be used for an **array in ascending order**
2. **Merge Sort** should be used for **an array in descending order**
3. **Insertion Sort** should be used for **an array of random elements if the elements are already almost sorted. Merge Sort should be used otherwise.**