

Presented to the Department of Software Technology  
College of Computer Studies  
De La Salle University  
A.Y. 2023-2024

**CCDSALG (Data Structures and Algorithms)**  
**Major Course Output (MCO) 1**  
**Group Report**

Submitted by: Group 7  
**Aleandrino, Vince Alejson V.**  
**Clemente, Daniel Gavrie Y.**  
**Hernandez, Christa Ysabel T.**  
**Lingat, Carl Vincent Blix P.**

Submitted to:  
**Mr. Romualdo M. Bautista Jr.**

June 26, 2023

## **Chapter 1: A Brief Introduction of the Project and its Outline**

The project aims to compare the run time or efficiency of four different sorting algorithms, specifically insertion sort, selection sort, merge sort, and comb sort. These algorithms will be tested on the struct arrays of sizes 100, 25000, 50000, 75000, and 100000, which are all scanned from a text file. The struct is named *record*, and it contains a string of size 500 named *name* and an integer named *idNumber*. The algorithms will use the aforementioned integer to sort or arrange the data in ascending order.

To facilitate this exploration, our project will take an empirical approach. This involves running each algorithm multiple times on the aforementioned struct arrays and accurately measuring the time taken for each execution. This will empirically assess each algorithm's efficiency under varying load sizes. It is crucial to note that these tests will be performed with the same system or computer to ensure the accuracy of our results.

The sorting algorithms under investigation each have unique characteristics and principles of operation. The insertion sort, for example, operates on the principle of scanning, comparing, and inserting elements appropriately within a list. Selection sort takes a similar approach, but rather than inserting, it focuses on swapping the least element with the first element in the unsorted portion of the array. Merge sort, a more complex algorithm, divides the problem into smaller manageable parts, sorts these parts, and then merges them together. Comb sort, on the other hand, is an improvement of bubble sort that eliminates the problem of "turtles" by initially comparing elements that are far apart.

These diverse algorithms will be run against the struct arrays, which consist of a 'name' string and an 'idNumber' integer. The 'idNumber' will serve as the key on which the sorting operations will be performed. This approach aligns with practical applications of sorting algorithms, where struct arrays often comprise heterogeneous data types.

After running our tests, we will analyze the data to identify any patterns and differences in performance between the algorithms. These insights will be used to understand each algorithm's relative strengths and weaknesses and under what circumstances one might be preferred over another. By understanding these trade-offs, we can make informed decisions about which algorithm to use in a given context or for a specific application.

By the end of this project, we expect to have a comprehensive understanding of the performance characteristics of each of these sorting algorithms. This understanding will serve as a valuable resource for future projects and studies involving data sorting and manipulation.

The following chapters will go into more depth about the specific details of each algorithm, the testing methodology, the data analysis, and finally, the conclusions that can be drawn from this study.

Outline:

Chapter 1: *A Brief Introduction to the Project and its Outline*

Chapter 2: *The Sorting Algorithms Implemented by the Project*

Chapter 3: *The Implementation of the Algorithms in the Code*

Chapter 4: *The Performance of the Sorting Algorithms*

Chapter 5: *A Comparative Analysis of the Sorting Algorithms*

Chapter 6: *The Findings, Learnings, and Realizations of the Group*

## **Chapter 2: The Sorting Algorithms Implemented by the Project**

The following algorithms are implemented in the project and their descriptions:

### 1. Insertion Sort

Insertion sort divides the input into a sorted and an unsorted region. The sorted region starts as just the first element of the array. The algorithm then iteratively takes the first unsorted element and 'inserts' it into the correct position in the sorted region, shifting the other elements as necessary. According to an article by Dartmouth Computer Science professors Thomas Cormen and Devin Balkcom, the algorithm is akin to sorting a set of cards. To keep the cards sorted when receiving a new card, a person would insert it in between the cards that are of lower and higher value of the given card. Similarly, when the algorithm is given a new position in an array, it will insert the element into an already sorted subarray between an element of lower value and another of higher value, hence the name insertion sort. Its average and worst case is  $O(n^2)$ .

In the worst-case scenario, when the input array is sorted in reverse order, insertion sort has a time complexity of  $O(n^2)$  due to the nested loops. The best-case scenario is an already sorted input, resulting in a linear running time  $O(n)$  because the inner loop is never executed in this case. The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is efficient for relatively small data sets. It is more efficient in practice than most other simple quadratic (i.e.,  $O(n^2)$ ) algorithms, such as selection sort or bubble sort.

## 2. Selection Sort

The algorithm works by dividing the input into a sorted and an unsorted region. The sorted region is built up by repeatedly selecting the smallest (or largest, depending on the ordering) element of the unsorted array and swapping it with the first element in the unsorted part. Despite its simplicity, it is not typically suitable for large data sets due to its  $O(n^2)$  complexity.

Its main advantage is its simplicity and the fact that it performs well for smaller lists or for lists with large values where the cost of swapping is significantly less than the cost of comparison. It is an in-place sorting algorithm, meaning it sorts the list by changing its contents. Unlike some other sorting algorithms, there is no best-case scenario. Even if the list is already sorted, it will still go through all its steps.

## 3. Merge Sort

Merge sort is a comparison-based algorithm that focuses on how to merge two pre-sorted arrays such that the resulting array is also sorted. It is a classic example of a divide-and-conquer algorithm that breaks a problem down into smaller, more manageable sub-problems until each sub-problem is simple enough to solve directly.

In the case of merge sort, the algorithm divides the array into halves, sorts each half separately, and then merges the two sorted halves back together. This process is applied recursively unless the arrays are of size one or zero which are sorted by definition. The algorithm contains three processes: Divide, Conquer, and Combine. The array is divided into two halves by

the middle index. This step is repeated recursively until we are left with size one or zero subarrays. Each half is sorted recursively using merge sort. Lastly, the two sorted halves merge to create a larger array. Merge sort performs well with the time complexity of  $O(n \log n)$  in all cases (worst, average, and best).

#### 4. Comb Sort

Comb Sort is initially designed to improve upon the bubble sort, and is related to the shell sort. It was developed in 1980 by Włodzimierz Dobosiewicz and Artur Borowy, and later rediscovered and popularized by Stephen Lacey and Richard Box with a Byte Magazine publication in April 1991.

The key idea in Comb Sort is the elimination of turtles, or small values at the end of the list, which are a problem in bubble sort. It does this by comparing elements with a gap larger than 1, allowing smaller values to move to the beginning more quickly. The algorithm includes four main procedures. First, the gap size to the length of the array is initialized. Then the gap size is divided by the shrink factor at the beginning of each iteration. Each pair of elements separated by the gap is then compared and swapped if they are placed in the wrong order. The process is repeated until the gap size becomes one and the array is fully sorted. Its worst-case scenario is  $O(n^2)$ , similar to bubble sort.

### Chapter 3: The Implementation of the Algorithms in the Code

The four algorithms in the program are taken from various sources on the internet but modified in order for the structs to be sorted properly. For example, in Photo 1, it is an insertion sort algorithm obtained from [geeksforgeeks.org](https://www.geeksforgeeks.org/insertion-sort/) that sorts an array of integers while Photo 2 is the same algorithm but modified to sort an array of structs of type record.

#### Photo 1: The unaltered insertion sort.

```
void insertionSort(int *arr, int n)
{
    int i, j;
    int key;

    for (i = 1; i < n; i++)
    {
        key = arr[i];

        j = i - 1;

        // Compare key with each element on the left of it until an element smaller
        // than it is found.
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }

        arr[j + 1] = key;
    }
}
```



## Photo 2: The modified insertion sort.

```
void insertionSort(Record *arr, int n)
{
    int i, j;
    Record key;

    for (i = 1; i < n; i++)
    {
        key = arr[i];

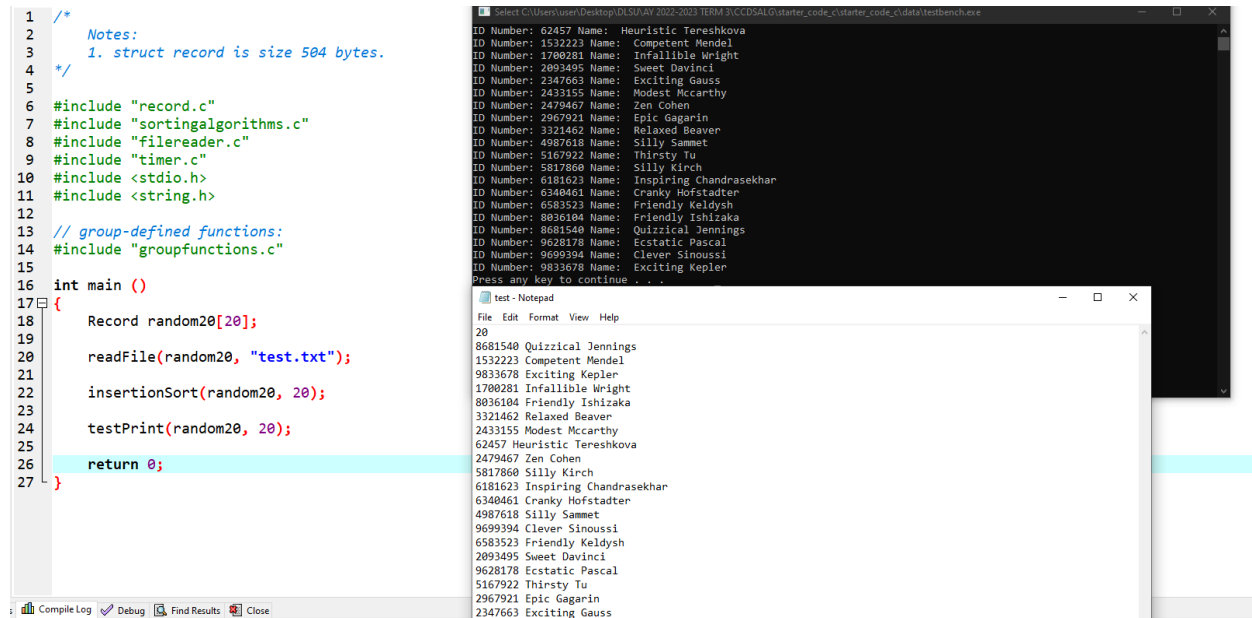
        j = i - 1;

        // Compare key with each element on the left of it until an element smaller
        // than it is found.
        while (j >= 0 && arr[j].idNumber > key.idNumber)
        {
            arr[j+1] = arr[j];
            j = j - 1;
        }

        arr[j+1] = key;
    }
}
```

In order to ensure that the modified algorithm sorts properly, it is tested separately on a different program using a text file that is of a significantly smaller size than the given but it still uses the same function that the project program is using. Once sorting is finished, the numbers and names are then visually inspected to verify that the given data is sorted correctly. Photo 3 shows the said procedure.

Photo 3: The test code and its corresponding inputs and outputs are shown.

The image shows a screenshot of a C program and its execution. On the left, a code editor displays the source code for a program that generates random records, reads them from a file, sorts them using insertion sort, and prints them. The code includes headers for record.c, sortingalgorithms.c, filereader.c, timer.c, stdio.h, and string.h. The main function declares an array of 20 records, calls readFile to populate it, calls insertionSort to sort it, and calls testPrint to display it. On the right, a terminal window shows the output of the program, which lists 20 records with their IDs and names. The records are: 62457 Heuristic Tereshkova, 153223 Competent Mendel, 1700281 Infallible Wright, 2093495 Sweet Davinci, 2347663 Exciting Gauss, 2433155 Modest Mccarthy, 2479467 Zen Cohen, 2967921 Epic Gagarin, 3321462 Relaxed Beaver, 4987618 Silly Sammet, 5167922 Thirsty Tu, 5817860 Silly Kirch, 6181623 Inspiring Chandrasekhar, 6340461 Cranky Hofstadter, 6583523 Friendly Keldysh, 8036104 Friendly Ishizaka, 8681540 Quizzical Jennings, 9628178 Ecstatic Pascal, 9699394 Clever Sinoussi, and 9833678 Exciting Kepler. Below the terminal, a Notepad window shows the same list of records.

In implementing the algorithms on the actual project program itself, an array of structs of type record was declared, and since some sizes are too large for C to handle, the function malloc had to be called to accommodate them. After declaration, all the declared arrays of structs will receive their contents according to their names from the function readFile in order to avoid confusion. A do-while loop is then used to make the program run until a certain option is chosen, which will terminate the program. This is done to make testing more efficient since it will eliminate the necessity to keep starting the program repeatedly. Since the program is on a loop, it is ideal that the arrays declared in main are not changed so that when it is passed to the functions of the menus of the sorting algorithms, it remains unsorted. The functions that have “Menu” on its name or identifier works by prompting the user which data set should the chosen algorithm be implemented on and calling the function that contains the algorithm. After the sorting is done,

the user is then shown the run time in milliseconds and given a choice to see the sorted data (although not all of the data will be shown due to the limitations of the command prompt). Another thing to note is that there are arrays of struct of type record declared inside the function “menus” and this is done in order to keep the array passed from main unsorted. Below are the pictures of the code and outputs of the actual project program.

Photo 4: The declaration of the arrays of struct record in main.

```
int main()
{
    int dChoice, dFlag = 0;

    Record random100[100]; // Unsorted array. DO NOT CHANGE/SORT ITS CONTENTS!
    Record *random25000 = malloc(sizeof(Record) * 25000);
    Record *random50000 = malloc (sizeof(Record) * 50000);
    Record *random75000 = malloc (sizeof(Record) * 75000);
    Record *random100000 = malloc (sizeof(Record) * 100000);
    Record *almostsorted = malloc (sizeof(Record) * 100000);
    Record *totallyreversed = malloc (sizeof(Record) * 100000);

    readFile(random100, "random100.txt");
    readFile(random25000, "random25000.txt");
    readFile(random50000, "random50000.txt");
    readFile(random75000, "random75000.txt");
    readFile(random100000, "random100000.txt");
    readFile(almostsorted, "almostsorted.txt");
    readFile(totallyreversed, "totallyreversed.txt");
```

Photo 5: A fragment of the menu function of insertion sort.

```
void InsertionMenu(Record *random100, Record *random25000, Record *random50000, Record *random75000, Record *random100000, Record *almostsorted, Record *totallyreversed)
{
    int dChoice;
    long startTime, endTime, executionTime;
    char dYesOrNo;
    Record data100[100];
    Record *data25000 = malloc(sizeof(Record) * 25000);
    Record *data50000 = malloc(sizeof(Record) * 50000);
    Record *data75000 = malloc(sizeof(Record) * 75000);
    Record *data100000 = malloc(sizeof(Record) * 100000);
    Record *dataAlmostsorted = malloc(sizeof(Record) * 100000);
    Record *dataTotallyreversed = malloc (sizeof(Record) * 100000);

    system("cls");
    printf("Which data set do you want to use insertion sort?\n");
    printf("[1] - Almost Sorted\n");
    printf("[2] - Random 100\n");
    printf("[3] - Random 25000\n");
    printf("[4] - Random 50000\n");
    printf("[5] - Random 75000\n");
    printf("[6] - Random 100000\n");
    printf("[7] - Totally Reversed\n");
    scanf("%d", &dChoice);
```

Photo 6: Another fragment of the menu function of insertion sort.

```
switch(dChoice)
{
    case 1:
        arrayAssignment(almostsorted, dataAlmostsorted, 100000);

        startTime = currentTimeMillis();
        insertionSort(dataAlmostsorted, 100000);
        endTime = currentTimeMillis();

        executionTime = endTime - startTime;

        printf("\nExecution Time: %ld\n\n", executionTime);
        printf("Do you want to see the sorted algorithm? (Press Y for Yes and any other for no)\n");

        fflush(stdin);

        scanf("%c", &dYesOrNo);
        if (dYesOrNo == 'Y' || dYesOrNo == 'y')
            testPrint(dataAlmostsorted, 100000);

        break;
```

Photo 7: The main menu of the program.

```
CCDSALG MC01
SORTING ALGORITHMS

CHOOSE YOUR ALGORITHM:
[1] - Insertion Sort
[2] - Selection Sort
[3] - Merge Sort
[4] - Comb Sort
[5] - Exit
```

Photo 8: The prompt for which dataset to implement the algorithm.

```
Which data set do you want to use insertion sort?  
[1] - Almost Sorted  
[2] - Random 100  
[3] - Random 25000  
[4] - Random 50000  
[5] - Random 75000  
[6] - Random 100000  
[7] - Totally Reversed
```

Photo 9: The event after running an algorithm.

```
Which data set do you want to use insertion sort?  
[1] - Almost Sorted  
[2] - Random 100  
[3] - Random 25000  
[4] - Random 50000  
[5] - Random 75000  
[6] - Random 100000  
[7] - Totally Reversed  
3  
  
Execution Time: 5477  
  
Do you want to see the sorted algorithm? (Press Y for Yes and any other for no)
```

Photo 10: The sorted algorithm in display.

```
ID Number: 9988855 Name:  Goofy Ptolemy
ID Number: 9988883 Name:  Dreamy Goldstine
ID Number: 9988991 Name:  Kind Pare
ID Number: 9989848 Name:  Bold Williamson
ID Number: 9990079 Name:  Recursing Hellman
ID Number: 9991266 Name:  Goofy Mccarthy
ID Number: 9992398 Name:  Lucid Dubinsky
ID Number: 9992454 Name:  Awesome Elgamal
ID Number: 9993044 Name:  Epic Tharp
ID Number: 9993090 Name:  Charming Colden
ID Number: 9993419 Name:  Magical Almeida
ID Number: 9993952 Name:  Intelligent Feistel
ID Number: 9994397 Name:  Nostalgic Huggle
ID Number: 9994482 Name:  Silly Lamarr
ID Number: 9994780 Name:  Sad Cannon
ID Number: 9994915 Name:  Elastic Lichterman
ID Number: 9995879 Name:  Magical Bohr
ID Number: 9996388 Name:  Distracted Shockley
ID Number: 9996788 Name:  Hopeful Neumann
ID Number: 9996930 Name:  Hopeful Hodgkin
ID Number: 9996935 Name:  Confident Mahavira
ID Number: 9997302 Name:  Strange Ritchie
ID Number: 9997373 Name:  Sweet Beaver
ID Number: 9997489 Name:  Sweet Gauss
ID Number: 9999312 Name:  Dazzling Kilby
ID Number: 9999700 Name:  Inspiring Feynman
ID Number: 9999749 Name:  Modest Wilbur
ID Number: 9999787 Name:  Romantic Mcclintock
ID Number: 9999881 Name:  Quizzical Williams
Press any key to continue . . .
```

#### **Chapter 4: The Performance of the Implemented Sorting Algorithms**

The following are the run times in milliseconds of the algorithms implemented in the program. They are the averages of three run times.

Algorithm Dataset	Insertion	Selection	Merge	Comb
Random 100	0	0	0	0
Random 25000	6486	945	87	42
Random 50000	35849	3862	191	106
Random 75000	82884	10977	315	165
Random 100000	158554	33450	411	232
Almost Sorted	27023	31214	389	206
Totally Reversed	317469	35126	414	97

The following is the frequency counts of the implemented sorting algorithms:

### Insertion Sort

<code>void insertionSort(Record *arr, int n){</code>		
<code>int i, j;</code>		
<code>Record key;</code>		
<code>for (i = 1; i &lt; n; i++){</code>	$n - 1 + 2$	$n + 1$
<code>key = arr[i];</code>	$n - 1 + 1$	$n$
<code>j = i - 1;</code>	$n - 1 + 1$	$n$
<code>while (j &gt;= 0 &amp;&amp; arr[j].idNumber &gt; key.idNumber){</code>	$(n - 1 + 1)(i - 1 - 0 + 2)$	$in + n$
<code>arr[j+1] = arr[j];</code>	$(n - 1 + 1)(i - 1 - 0 + 1)$	$in$
<code>j = j - 1;</code>	$(n - 1 + 1)(i - 1 - 0 + 1)$	$in$
<code>}</code>		
<code>arr[j+1] = key;</code>	$n - 1 + 1$	$n$
<code>}</code>		
	Frequency Count:	$3in + 5n + 1$

### Selection Sort

<code>void selectionSort(Record *arr, int n)</code>		
<code>{</code>		
<code>int i, j, min;</code>		
<code>Record temp;</code>		
<code>for (i = 0; i &lt; n-1; i++){</code>	$n - 1 - 0 + 2$	$n + 1$
<code>{</code>		
<code>// minimum index</code>		
<code>min = i;</code>	$n - 1 - 0 + 1$	$n$
<code>// traverse array to find minimum value</code>		
<code>for(j = i+1; j &lt; n; j++){</code>	$n - 1 - 0 + 1 (n - i + 1 + 2)$	$n^2 - in + 3n$
<code>{</code>		
<code>if(arr[j].idNumber &lt; arr[min].idNumber)</code>	$n - 1 - 0 + 1 (n - i + 1 + 1)$	$n^2 - in + 2n$
<code>min = j;</code>	$n - 1 - 0 + 1 (n - i + 1 + 1)$	$n^2 - in + 2n$
<code>}</code>		
<code>// swap if minimum value is not the same as starting value</code>		
<code>if (min != i)</code>	$n - 1 - 0 + 1$	$n$
<code>{</code>		
<code>temp = arr[i];</code>	$n - 1 - 0 + 1$	$n$
<code>arr[i] = arr[min];</code>	$n - 1 - 0 + 1$	$n$
<code>arr[min] = temp;</code>	$n - 1 - 0 + 1$	$n$
<code>}</code>		
<code>}</code>		
<code>}</code>		
	Frequency Count:	$3n^2 - 3in + 13n + 1$



## Merge Sort (DISCLAIMER: IT MAY BE INACCURATE)

Merge Sort:

<code>void mergeSort(Record arr[],</code>		
<code>int l, int r){</code>		
<code>if (l &lt; r){</code>		1
<code>int m = l + (r - l) / 2;</code>		1
<code>mergeSort(arr, l, m);</code>	$T(r-1)$	
<code>mergeSort(arr, m + 1, r);</code>	$T(r-1)+1$	
<code>merge(arr, l, m, r);</code>	$11n1+6n2+16$	
<code>}</code>		
<code>}</code>		
	Frequency Count:	$2T(r-1)+11n1+6n2+19$

## Merge Function:

void merge(Record *arr, int l, int m, int r)		
{		
int i, j, k;		
int n1 = m - l + 1;	1	
int n2 = r - m;	1	
// Create temp arrays		
Record *L = malloc(sizeof(Record)*	1	
Record *R = malloc(sizeof(Record)*	1	
// Copy data to temp arrays		
// L[] and R[]		
for (i = 0; i < n1; i++)	n1+2	
L[i] = arr[l + i];	n1+1	
for (j = 0; j < n2; j++)	n2+2	
R[j] = arr[m + 1 + j];	n2+1	
// Merge the temp arrays back		
// into arr[l..r]		
// Initial index of first subarray		
i = 0;	1	
// Initial index of second subarray		
j = 0;	1	
// Initial index of merged subarray		
k = l;	1	
while (i < n1 && j < n2)	n1+1	
{		
if (L[i].idNumber <= R[j].idNumber)	n1	
{		
arr[k] = L[i];	n1	
i++;	n1	
}		
else		
{		
arr[k] = R[j];		
j++;		
}		
k++;	n1	
}		
// Copy the remaining elements		
// of L[], if there are any		
while (i < n1) {	n1+1	
arr[k] = L[i];	n1	
i++;	n1	
k++;	n1	
}		
// Copy the remaining elements of		
// R[], if there are any		
while (j < n2)	n2+1	
{		
arr[k] = R[j];	n2	
j++;	n2	
k++;	n2	
}		
}		
Frequency Count:	11n1+8n2+16	

## Comb Sort

void combSort(Record *arr, int n)		
{		
double gap = n;		1
bool swaps = true;		1
int igap, i;		
Record temp;		
while (gap > 1    swaps)	$n-1+1$	$n$
{		
gap /= 1.247330950103979;	$n-1$	$n-1$
if (gap < 1)	$n-1$	$n-1$
gap = 1;	$n-1$	$n-1$
i = 0;	$n-1$	$n-1$
swaps = false;	$n-1$	$n-1$
while (i + gap < n)	$n-1(2n+1)$	$2n^2-n-1$
{		
igap = i + (int) gap;	$n-1(2n)$	$2n^2-2n$
if (arr[i].idNumber > arr[igap].idNumber)	$n-1(2n)$	$2n^2-2n$
{		
temp = arr[i];	$n-1(2n)$	$2n^2-2n$
arr[i] = arr[igap];	$n-1(2n)$	$2n^2-2n$
arr[igap] = temp;	$n-1(2n)$	$2n^2-2n$
swaps = true;	$n-1(2n)$	$2n^2-2n$
}		
i++;	$n-1(2n)$	$2n^2-2n$
}		
}		
}		
Frequency Count:		$16n^2-9n-4$

## **Chapter 5: A Comparative Analysis of the Different Sorting Algorithms**

The sorting algorithms presented in the previous chapters show the different frequency count each has as well as their running time. This chapter aims to not only compare the algorithms, but also their theoretical concepts.

### *The Priori Analysis:*

Upon careful analysis and comparison of the provided sorting algorithms, it becomes apparent that their time complexities align with their commonly known values, confirming the validity of the a priori analysis. The Insertion Sort algorithm's time complexity of  $3in + 5n + 1$  corresponds to the expected  $\Omega(n)$  time complexity, indicating its efficiency for smaller input sizes. The Selection Sort algorithm's time complexity of  $3n^2 - 3in + 13n + 1$  aligns with the anticipated  $O(n^2)$  notation, highlighting its higher time requirements for larger datasets. Similarly, the Comb Sort algorithm's time complexity of  $16n^2 - 9n - 4$  also matches the established  $O(n^2)$  notation, emphasizing its performance characteristics. However, the time complexity of the Merge Sort algorithm remains unknown due to its difficulty in finding the frequency count.

### *The Posteriori Analysis:*

Upon observing the table in Chapter 4, it presents the average run times in milliseconds of four different algorithms (Insertion, Selection, Merge, and Comb) under different conditions (Random 100, Random 25000, Random 50000, Random 75000, Random 100000, Almost Sorted, and Totally Reversed).

Analyzing the data, we can make several observations:

1. **Performance across Datasets:** As expected, as the dataset size increases, all algorithms show an increase in run time. However, the rate at which the run time increases varies significantly across the different algorithms.
2. **Insertion Algorithm:** The Insertion algorithm's performance deteriorates most rapidly with the increase in dataset size. This is particularly noticeable with the "Random 100000" and "Totally Reversed" datasets, where it has the worst performance of all four algorithms. Interestingly, it performs relatively better in the "Almost Sorted" case, implying that it may be more suitable for already somewhat sorted datasets.
3. **Selection Algorithm:** The Selection Sort algorithm performs better than the Insertion algorithm but still shows a significant increase in run time as the dataset size increases. It performs especially poorly with the "Almost Sorted" dataset, suggesting that it does not take advantage of the pre-sorted nature of the dataset.
4. **Merge Algorithm:** The Merge algorithm outperforms the Insertion Sort and Selection Sort algorithms across all datasets. Its run time grows more slowly than

the other two with the increase in dataset size, indicating that it is a more efficient algorithm for larger datasets.

5. **Comb Algorithm:** The Comb algorithm has the best overall performance of the four algorithms. It has the shortest run time in almost all cases, showing only a modest increase as the dataset size grows. It is worth noting its superior performance even in the "Totally Reversed" dataset, where other algorithms particularly struggled.

Overall, these results demonstrate the efficiency of the Merge and Comb algorithms, particularly for larger or reversed datasets. The Selection and Insertion algorithms, while they may have uses in certain contexts, struggle with larger and more complex datasets. The Comb algorithm shows the most consistent performance across different types of datasets. These insights can help choose the most appropriate algorithm for specific contexts.

## **Chapter 6: The Findings, Learnings, and Realizations of the Group**

The heart of the project lies in its empirical and theoretical comparative analysis of these sorting algorithms, where it scrutinizes the runtime and theoretical concepts of each. This leads to a rich understanding of each algorithm's performance and efficiency under diverse data loads. In this chapter, the study concludes by summarizing the group's findings, lessons, and realizations. This section embodies the practical value of the study as the group reflects upon their journey, the challenges encountered, the solutions devised, and the broader implications of their findings on computational efficiency and algorithm selection.

The first realization is the coding part of the project. As mentioned earlier, the codes were from the internet, as almost, if not, all codes implement the same idea or principles for each sorting algorithm. However, they have to be modified since these codes only work for an array or list of integers and not structs. At first, modifying these codes was challenging because it would always end up with the ID numbers being sorted, but the names are not associated with their respective ID numbers. After a few trials and errors, the modified codes now work, and it is displaying the correct output. The realization is that as long as the basic idea of the algorithm is known, it will be much easier to understand what needs to be done. This is due to being able to visualize what is happening in the program and implement the necessary changes.

After coding, the group did an a priori analysis to determine if the frequency count matched what was being taught or in resources. The priori analysis took a significant amount of effort as the group could not be certain of the frequency count of each sorting algorithm. Though the process was daunting and time-consuming, the group was able to finalize and make certain that the frequency count was correct.

Aside from a priori analysis, a posteriori analysis was also done to determine the algorithms' efficiency. The analysis offers significant insights into the efficiency and scalability of the four evaluated algorithms. It is clear that the size and condition of datasets can greatly influence the algorithm's performance. The Insertion and Selection algorithms struggle with larger datasets, particularly if they are in a totally reversed state, indicating their limited scalability. Conversely, the Merge and Comb algorithms demonstrate more robust performance,

maintaining their efficiency even as the dataset size increases significantly. Remarkably, the Comb algorithm consistently outperforms all others across nearly all datasets, demonstrating its robustness and versatility. This brief analysis underscores the importance of understanding the characteristics of the dataset and the strengths of each algorithm to make informed decisions in algorithm selection for optimal performance.

After accomplishing the project, the group has made several insights and takeaways. The first insight is that it is important to understand how the sorting algorithm works in order to know the needed modifications in different contexts like the sorting of structs in this project. This has made the coding part of the project significantly less challenging, allowing for more creative implementation ideas like using functions to “restart” the array instead of opening and closing the program repeatedly. The second insight is to have patience as we are used to having our programs run and do their job immediately and the program for the project showed that it can take a while for some functions (in this case, the sorting algorithms) to do their job, taking as long as five minutes to complete. This is a valuable insight as not all programs will be instantaneous and this gives us caution to what may possibly happen to the programs we make for further College of Computer Studies courses. The third insight is to be meticulous even when struggling. It helps to be meticulous as it can help trace where the error is, establish more accurate findings, and even gain knowledge.

Overall, the project is very insightful and interesting despite the challenges and struggles along the way. It has enriched our practical knowledge of the sorting algorithms as well as our



coding, logical, and mathematical skills. There is no best sorting algorithm and for this reason, it is crucial to choose which one works best in different situations.

**Appendix:**

Group Member	Contributions
Alejandrino, Vince Alejson V.	Getting the frequency count and writing 2, 4, and 5.
Clemente, Daniel Gavrie Y.	Getting the frequency count and writing chapters 4, 5, 6.
Hernandez, Christa Ysabel T.	Wrote chapters 1, 2, 5, and 6.
Lingat, Carl Vincent Blix P.	Coding the program and writing chapters 1, 3, 4, and 6.

## References

- Alake, R. (2021, December 9). *Insertion Sort Explained—A Data Scientists Algorithm Guide* | *NVIDIA Technical Blog*. NVIDIA Developer. Retrieved June 18, 2023, from <https://developer.nvidia.com/blog/insertion-sort-explained-a-data-scientists-algorithm-guide/>
- Comb Sort Implementation in C*. (n.d.). Programming Algorithms. <https://www.programmingalgorithms.com/algorithm/comb-sort/c/>
- Cormen, T., Balkcom, D., & Khan Academy. (n.d.). *Insertion sort (article)* | *Algorithms*. Khan Academy. Retrieved June 18, 2023, from <https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort>
- Dobosiewicz, W. and Borowy, A. "More on a Natural Heapsort", *Inform. Process. Lett.*, 11, (1), (1980), pp. 1–4.
- GeeksforGeeks. (2016, October 1). *Comb Sort* | *GeeksforGeeks*. YouTube. <https://www.youtube.com/watch?v=n51GFZHXIYY>
- GeeksforGeeks. (2023, March 17). *C Program For Insertion Sort*. GeeksforGeeks. <https://www.geeksforgeeks.org/c-program-for-insertion-sort/>
- GeeksforGeeks. (2023, March 23). *C Program for Merge Sort*. GeeksforGeeks. <https://www.geeksforgeeks.org/c-program-for-merge-sort/>
- GeeksforGeeks. (2023, May 26). *Selection Sort – Data Structure and Algorithm Tutorials*. GeeksforGeeks. <https://www.geeksforgeeks.org/selection-sort>

Knuth, D.E. "The Art of Computer Programming, Volume 3: Sorting and Searching".  
Addison-Wesley. 1998.

Lacey, S. and Box, R. "Comb Sort – a fast improvement of bubble sort", Byte Magazine, (1991).

Programiz. (n.d.). *Insertion Sort (With Code in Python/C++/Java/C)*. Programiz.

<https://www.programiz.com/dsa/insertion-sort>

Sedgewick, R. "Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching".  
Addison-Wesley Professional. 1997.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. "Introduction to  
Algorithms, Third Edition". The MIT Press. 2009.