CCDSALG S13 Group 7
October 20, 2023
Clavano, Angelica Therese I. (Jack)
Homssi, Yazan M.
Young, Henzley Emmanuel S.
Jimenez, Yves Alvin Andrei A.

Major Course Output 1: Sorting Algorithms

I. Introduction

A brief introduction of the project and an outline of the contents of the report.(Henzley)

This project serves to compare and contrast different sorting algorithms, mainly when it comes to their running time and use cases. The sorting algorithms chosen were Insertion sort, Selection sort, Merge sort, and Comb sort. Each algorithm is benchmarked in order to get each of their average runtime, as well as have posteriori analysis performed to get their empirical frequency counts. Results are then compared to each other with insights from the group as well as a summary of the group's findings.

II. Sorting Algorithms

The list of sorting algorithms implemented (including the required algorithms), along with a concise description of each algorithm and its implementation. (All group members)

Insertion sort

Insertion sort is an in-place, stable sorting algorithm. Insertion sort splits a given array into two sections, unsorted and sorted. The first element of the given array is considered as the "sorted list". Each subsequent element of the "unsorted list" will then be compared to each element of the "sorted list" via incrementation and will be placed into the sorted list accordingly. Insertion sort is an in-place sorting algorithm, meaning that the output of the function is stored in the input function, so that no additional memory is needed to create temporary variables except for the ones used for for loops. This algorithm was modified in order to support the needed data structures.

Selection Sort

Selection sort is an in-place, not stable sorting algorithm. Selection sort also does not take up any additional/extra memory, so it is known for its simplicity and memory efficiency. This algorithm works by dividing the input array into two parts: "sorted" & "unsorted", the algorithm repeatedly selects the smallest element (or largest depending on the implementation) from the unsorted part of the array, and moves it to the sorted part of the array. Despite the selection sort and insertion sort being similar in a way that they both make use of nested loops– the difference between them is what the inner loop does. For Selection sort, the inner loop iterates over the unsorted elements (unlike insertion sort), each pass selects one element, and moves it to its final location (the current end of the sorted region).

Merge Sort

Merge sort is an out-of-place sorting algorithm and retains the relative positions of the data with the arrays, as such it is a stable sorting algorithm. The way it works is basically divide and conquer, an array is partitioned into two halves, and the array is sorted recursively until the entire data set is sorted and then merged back. Although the partition is arbitrarily set, the rule of thumb is to split the array which was also used in the implementation.

Comb Sort

Comb Sort is an iterative sorting algorithm as it uses a for-loop nested within a while-loop instead of calling itself recursively. It uses a temporary variable to store the element in the array to be swapped but does not make an entirely new data structure making it an in-place sorting algorithm. Similarly to Bubble Sort, Comb Sort makes use of a “gap” to compare elements but instead of a gap of 1 like in BS, it uses an even larger gap which shrinks per iteration. First, the gap is set to the size of the entire array. The gap functions as the number of spaces or distance between the current element and another element to be compared. Another variable “swapped” that functions as a flag is also declared. In the first iteration, this gap is divided by 1.3, the shrink factor. (Note that if the gap ends up becoming less than 1, it is set to 1) Then, the current element, which is stored in the temp variable, is then compared to the other one. If it’s less than the current element, they would switch places and the swapped flag would be placed to true and the while-loop would continue running until the gap is 1 or the swapped flag remained as false.

III. Description of Processes

A clear description of the process/es used in running the algorithms on the datasets, including the process/es used to verify the correctness of the results for the given data, and the process/es used to benchmark the execution times. Show supporting code if necessary. (All group members) (Jack)

All the files, filereader.c, record.c, sortingalgorithms.c, and timer.c were all held in one folder as our group used Github to track changes in our code. Within this folder are two other folders holding the datasets — “data” held the provided datasets and “custom” held three other files “extremelysmall10.txt”, “limtationscheck.txt” which was a modified version of random2500.txt, and “custom500duplicates.txt” which was a modified version of the “random100.txt” file provided to us. Due to these datasets being held in different folders compared to the rest of the program, for ease of use, we declared define directives to access these easier. (See Figure 1) Every relative file path was defined. **To change to another dataset, we only change one line which is the value within the currentFilePath definition.** This allows us to switch between file paths faster instead of changing multiple lines in main.c. To better navigate through the text printed to the console, color escape codes were also used as defined in Figure 1.

```c
#include "record.c"
#include "sortingalgorithms.c"
#include "filereader.c"
#include "timer.c"
#include <stdio.h>
#include <string.h>
//added
#include <stdlib.h>

// defining file relative paths, to be switched out per test
#define almostSortedFilePath "data\\almostsorted.txt"       // unable to run (100000) update: able to run
#define random100FilePath "data\\random100.txt"             // able to run (100)
#define random25000FilePath "data\\random25000.txt"         // unable to run (25000) update: able to run
#define random50000FilePath "data\\random50000.txt"         // unable to run (50000) update: able to run
#define random75000FilePath "data\\random75000.txt"         // unable to run (75000) update: able to run
#define random100000FilePath "data\\random100000.txt"       // unable to run (100000) update: able to run
#define totallyReversedFilePath "data\\totallyreversed.txt" // unable to run (100000) update: able to run
// custom test files
#define extremelySmall10FilePath "custom\\extremelysmall10.txt"       // able to run (10)
#define custom500duplicatesFilePath "custom\\custom500duplicates.txt" // able to run (500)
#define limitationscheckFilePath "custom\\limitationscheck.txt"       // able to run any amount of lines

//CURRENT FILE PATH - CHANGE THIS LINE TO CHANGE FILES
#define currentFilePath extremelySmall10FilePath

// colors for console
#define ANSI_COLOR_RED "\x1b[31m"
#define ANSI_OFF "\x1b[0m"
#define ANSI_YELLOW "\x1b[33m"
```

Figure 1. Define directives for the paths

At one point, our group was using different compilers (GCC provided by MSYS2 project and the TDM GCC Compiler from DevC++), so we ran into an issue where the relative path was not being recognized in the TDM GCC Compiler. Since the DevC++ compiler was outdated, we switched to MSYS2 and Visual Studio Code, allowing us to be able to handle the same errors, if there are any.

In the main.c file, we call a function called getFileLineNumber() (See Figure 2.1) and pass the currentFilePath variable into it. **getFileLineNumber() acts both as a checking function to see if the file can be located and read as well as a function that returns n containing the number of lines in the file.** The n variable is used when calling the sorting algorithm functions later on. This was checked by a printf statement within main.c to see if it got the right number. **Additionally, this function allows us to confirm that the previously mentioned define directives work.** getFileLineNumber() acts similarly to the readFile() function within filereader.c.

Summarized main.c breakdown:

1. Declares a dynamic array currentRecord containing Record structs to be passed into the sorting algorithms.
2. Gets the number of lines in the file to be passed into sorting algorithms (See getFileLineNumber() function above).
3. The address of the record struct (passing an array without & gives the address anyway) and the number of lines n in the file are passed into the sorting algorithms.
4. Individual sorting algorithms are called 5 times and the average time is printed to the console in main.

One of our first struggles was getting the program to run. We are aware that we are not allowed to modify the filereader file but we had to change the parameter of char path[500] to char path[] because if not, it would show an error and refuse to compile properly. (See Figure 1.1) **Note, it would still compile and create an exe file if this change wasn't made but the error will still appear.**

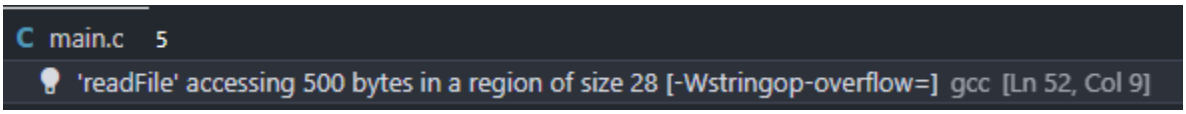


Figure 1.1. Error displayed whenever we called readFile with no changes. The region size displayed depended on the currentFilePath.

After solving the issue with compilation, we were working with an array declared as Record currentRecord[n]; but ran into an issue where the files were not able to be read past 5000 lines. This was concluded by running the datasets within the custom folder. Limitationstest.txt was used in this process by manually reducing lines in the text by chunks. The current state of limitationstest.txt is at 5001 lines. **When debugging the code, it shows that error lied within the declaration of the array saying it was a Segmentation fault.** (See Figure 2.2) According to Geeks4Geeks, this was caused by accessing memory that does not exist or unauthorized so we decided to declare currentRecord as a pointer where it is assigned the address of the first element. What the malloc() function does here is that it allocates memory on the heap for n instances of the Record structs. We used this dynamic memory allocation method in order to minimize data leaks and to make handling big chunks of data easier.

```c
int main()
{
    int n = getFileLineNumber(currentFilePath);
    /* For checking if file can be read and if it actually gets the first number in line */
    //printf("Current File Total line Number: %d\n", n);

    /* Replaced Record currentRecord[n]; with below*/
    Record *currentRecord = malloc(n * sizeof(Record)); //pointer is declared, stores the address
    of a dynamically allocated memory block.
    if (currentRecord == NULL) {
        printf("Error: failed to allocate memory for currentRecord\n");
        return 1;
    }

    readFile(currentRecord, currentFilePath);

    /* For checking if it actually reads the file and stores it into the array */
    //printf("FIRST LINE OF RECORD ARRAY: %d%s\n", currentRecord[0].idNumber, currentRecord[0].
    name);
    //printf("LAST LINE OF RECORD ARRAY: %d%s\n", currentRecord[n - 1].idNumber, currentRecord[n -
    1].name);

    /* Uncomment depending on which should run */
    printf(ANSI_COLOR_RED "Average execution time for Insertion Sort %ld milliseconds\n" ANSI_OFF,
    runInsertionSortTest(currentRecord, n));
    printf(ANSI_COLOR_RED "Average execution time for Selection Sort %ld milliseconds\n" ANSI_OFF,
    runSelectionSortTest(currentRecord, n));
    printf(ANSI_COLOR_RED "Average execution time for Merge Sort %ld milliseconds\n" ANSI_OFF,
    runMergeSortTest(currentRecord, n));
    printf(ANSI_COLOR_RED "Average execution time for Comb Sort %ld milliseconds\n" ANSI_OFF,
    runCombSortTest(currentRecord, n));

    free(currentRecord);
    return 0;
}
```

Figure 2. Main.c

```c
/*
    This function returns the number of lines in a file.
    It is used to determine the size of the array to be passed into the sorting algorithms.
*/
int getFileLineNumber(char path[])
{
    FILE *fp;
    int n = 0;
    fp = fopen(path, "r");

    if (fp == NULL)
    {
        printf("Could not open file %s", path);
        return 0;
    }

    fscanf(fp, "%d", &n); // N is the first line in the notepad file which denotes the amount of
    lines.
    fclose(fp);
    return n;
}
```

Figure 2.1. getFileLineNumber

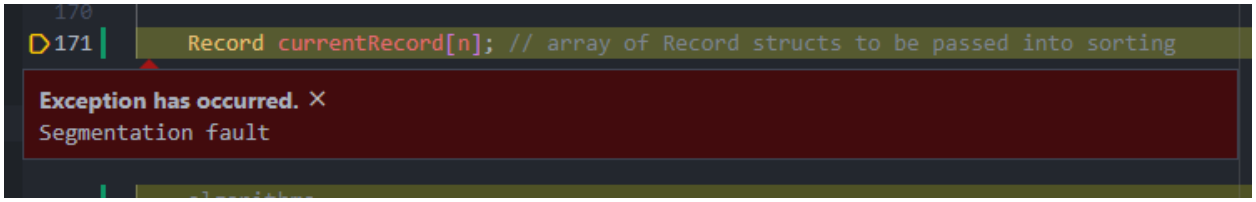


Figure 2.2. Segmentation Fault

To benchmark the algorithms at least 5 times, we created a template that every sorting algorithm follows. (See Figure 3) After importing the array and the n number of lines from main, we declare four new variables within this function: start (which contains the start time), end (which contains the end time), executionTime (which holds the difference between the start and end time), and averageExecutionTime (which holds the total of all the executionTimes). To run a certain algorithm 5 times, it is placed within a for loop leading from 0 to 5. At every iteration, start, end, and executionTime are set to zero. Next, the name of the sorting algorithm and current iteration is printed to the console for easy navigation. **To assure that the executionTime variable holds correct information, the start and end variables are printed exactly after currentTimeMillis() is called.** This is done before and after the sorting algorithm is called and allows us to manually compute and confirm the value held within averageExecutionTime.

Another issue we went through was handling the sorted array, we wondered if the array was actually being sorted. So in sortingalgorithms.c, a function was created called printRecord. (See Figure 4) What printRecord() does is take in the current state of the record that was passed into the sorting algorithm as well as the number of lines and prints the state into a file that is created called sorted.txt. **For every sorting algorithm, this is called within the function before and after the sorting process, therefore printing to the file the unsorted and sorted array into the file per iteration.** Before implementing this function, we noticed that there was a huge jump in milliseconds for any of the sorting algorithms from the first iteration to the second. **So through this check, we were able to pinpoint that this was because the second iteration was getting the sorted array from the first one**. To overcome this, we added readFile() to the end of the for loop (alongside the readFile() in main which stores the record into the array from earlier) within the template in Figure 3 so that it overwrites the array's contents. We did this to avoid making 4 more arrays to store copies of the unsorted one and to save memory.

```c
long runCombSortTest(Record *records, int n)
{
    long averageExecutionTime = 0;
    long start, end, executionTime;

    // benchmark algorithm at least 5 times and average the results
    for (int i = 0; i < 5; i++)
    {
        start = 0;
        end = 0;
        executionTime = 0;
        printf("COMB SORT ITERATION %d\n", i);
        start = currentTimeMillis(); // store the current time
        printf("Start Time: %ld\n", start);
        combSort(records, n);      // run sorting alg
        end = currentTimeMillis(); // store the current time
        printf("End Time: %ld\n", end);
        executionTime = end - start;
        printf(ANSI_YELLOW "Comb Sort took %ld milliseconds to execute.\n" ANSI_OFF,
        executionTime);
        averageExecutionTime += executionTime; // add to overall average
        printf("Current Total Average Execution Time: %ld\n", averageExecutionTime);
        readFile(records, currentFilePath);
    }
    return averageExecutionTime / 5.0;
}
```

Figure 3. Example of the template for running the algorithms five times.

```c
/* FOR CHECKING: prints the current array state into a file sorted.txt */
void printRecord(Record *currentRecord, int n) {
    FILE *fp = fopen("sorted.txt", "a+");

    for (int i = 0; i < n; i++) {
        fprintf(fp, "%d %s\n", currentRecord[i].idNumber, currentRecord[i].name);
    }
    fprintf(fp, "\n");

    fclose(fp);
}
```

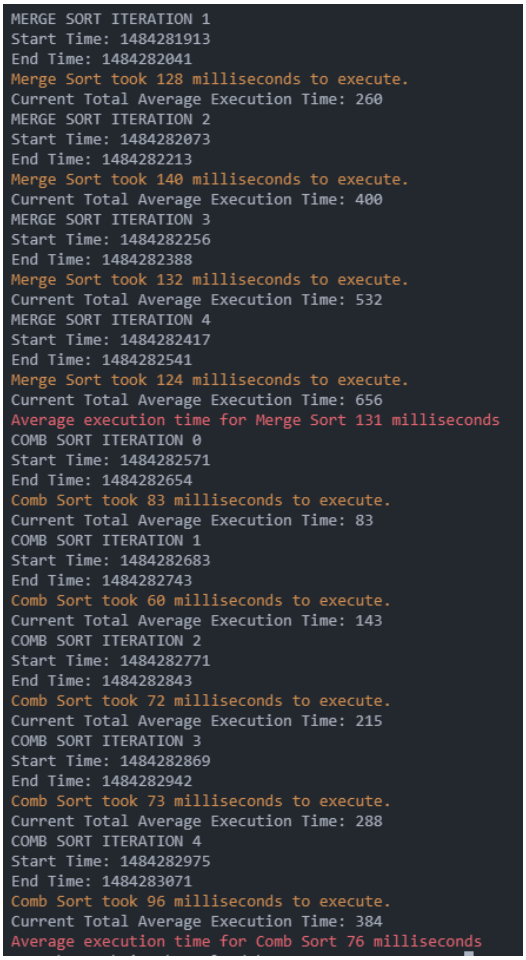
Figure 4. printRecord function

IV. Execution Times and Frequency Count

The execution times and frequency counts of the different algorithms on the given datasets. You can use tables and visualizations to convey the information as clearly as possible. (All)

A. EXECUTION TIMES

As the specs required our group to run the algorithm five times, what is shown below are the averages of these runs. Below is an example of what shows in our console.



RAN BY: JIMENEZ

Processor : Intel(R) Core(TM) i5-11400H @ 2.70GHz

Installed RAM: 16.0 GB (15.8 GB usable)

DATASET	INSERTION SORT	SELECTION SORT	MERGE SORT	COMB SORT
Almost Sorted (100000 size)	**26607ms**	**40682ms**	**237ms**	**246ms**
Random 100	**2ms**	**2ms**	**2ms**	**2ms**
Random 25000	**5858ms**	**983ms**	**58ms**	**57ms**
Random 50000	**31051ms**	**5451ms**	**121ms**	**127ms**
Random 75000	**72770ms**	**18362ms**	**203ms**	**210ms**
Random 100000	**144740ms**	**41086ms**	**245ms**	**281ms**
Totally Reversed	**284100ms**	**41107ms**	**245ms**	**100ms**
Custom Datasets				
Random 10	**2ms**	**2ms**	**2ms**	**2ms**

Random 500	**4ms**	**2ms**	**2ms**	**3ms**
Limit Check	**188ms**	**30ms**	**8ms**	**6ms**

RAN BY: CLAVANO

Processor: AMD Ryzen 7 5825U with Radeon Graphics 2.00 GHz
Installed RAM: 16.0 GB (14.8 GB usable)

DATASET	INSERTION SORT	SELECTION SORT	MERGE SORT	COMB SORT
Almost Sorted (100000 size)	**51059ms**	**9749ms**	**438ms**	**317ms**
Random 100	**0ms**	**0ms**	**1ms**	**1ms**
Random 25000	**5893ms**	**673ms**	**132ms**	**82ms**
Random 50000	**51984ms**	**2510ms**	**183ms**	**144ms**
Random 75000	**168320ms**	**6070ms**	**310ms**	**234ms**
Random 100000	**299094ms**	**13623ms**	**451ms**	**351ms**
Totally Reversed	**683017ms**	**15043ms**	**470ms**	**130ms**
Custom Datasets				
Random 10	**1ms**	**1ms**	**0ms**	**0ms**
Random 500	**2ms**	**1ms**	**1ms**	**1ms**
Limit Check	**175ms**	**27ms**	**6ms**	**6ms**

RAN BY: YOUNG

Processor: 12th Gen Intel(R) Core(TM) i5-12500H 3.10 GHz
Installed RAM: 8.00 GB (7.71 GB usable)

DATASET	INSERTION SORT	SELECTION SORT	MERGE SORT	COMB SORT
Almost Sorted (100000 size)	**24968ms**	**12686ms**	**279ms**	**186ms**
Random 100	**1ms**	**1ms**	**1ms**	**1ms**
Random 25000	**5918ms**	**660ms**	**82ms**	**62ms**
Random 50000	**24743ms**	**2786ms**	**138ms**	**108ms**
Random 75000	**95290ms**	**13856ms**	**273ms**	**220ms**
Random 100000	**170607ms**	**12084ms**	**271ms**	**218ms**
Totally Reversed	**246970ms**	**12421ms**	**278ms**	**93ms**
Custom Datasets				

Random 10	**0ms**	**0ms**	**0ms**	**0ms**
Random 500	**1ms**	**1ms**	**1ms**	**1ms**
Limit Check	**158ms**	**17ms**	**5ms**	**4ms**

RAN BY: HOMSSI

PROCESSOR: AMD Ryzen 5 3600 6-Core Processor 3.60 GHz

Installed RAM: 16.0 GB

DATASET	INSERTION SORT	SELECTION SORT	MERGE SORT	COMB SORT
Almost Sorted (100000 size)	**45007ms**	**11672ms**	**249ms**	**213ms**
Random 100	**3ms**	**1ms**	**2ms**	**2ms**
Random 25000	**6411ms**	**617ms**	**55ms**	**35ms**
Random 50000	**42045ms**	**2635ms**	**115ms**	**109ms**
Random 75000	**130014ms**	**6248ms**	**186ms**	**173ms**
Random 100000	**267137ms**	**10948ms**	**245ms**	**239ms**
Totally Reversed	**214420ms**	**23921ms**	**252ms**	**73ms**
Custom Datasets				
Random 10	**0ms**	**0ms**	**0ms**	**0ms**
Random 500	**2ms**	**1ms**	**1ms**	**1ms**
Limit Check	**196ms**	**20ms**	**4ms**	**4ms**

AVERAGES ON PROVIDED DATASETS (sum of all/number of members)

DATASET	INSERTION SORT	SELECTION SORT	MERGE SORT	COMB SORT
Almost Sorted (100000 size)	**36910.25ms**	**18697.25ms**	**300.75ms**	**240.5ms**
Random 100	**1.5ms**	**1ms**	**1.5ms**	**1.5ms**
Random 25000	**6020ms**	**733.25ms**	**81.75ms**	**59ms**
Random 50000	**37455.75ms**	**3345.5ms**	**139.25ms**	**122ms**
Random 75000	**116598.5ms**	**11134ms**	**243ms**	**209.25ms**
Random 100000	**220394.5ms**	**19435.25ms**	**303ms**	**272.25ms**
Totally Reversed	**357126.75ms**	**23123ms**	**311.25ms**	**99ms**

B. FREQUENCY COUNT

To get the frequency count, additional code was added to the sorting algorithms which displays the total count in the console. The table below shows the number of steps the algorithm has taken per dataset. (Jack)

DATASET	INSERTION SORT	SELECTION SORT	MERGE SORT	COMB SORT
Almost Sorted (100000 size)	429499030389162	429516740029862	429492435232807	598773342869
Random 100	2462937528773	2462937537722	546	5960
Random 25000	2421712632569	2422338696188	125098	4483414
Random 50000	2060817059989	2063314081029	250102	10310024
Random 75000	1802731727249	1808334548774	375106	14598893
Random 100000	1802731727249	1808334548774	375106	14598893
Totally Reversed	2828484965653	2828485015657	500106	16043890
Custom Datasets				
Random 10	1773483986067	1773483986177	79	239
Random 500	1700963677813	1700963924647	2549	41318
Limit Check	2759743384046	2759768094449	25080	707326

The updated code is shown below:

```c
/* Step counting variable was renamed to count due to step already being used as a variable name */
void insertionSortWithStepsAsCount(Record *arr, int n)
{
    long long int count; //added
    int step; count++;
    for (count++, step = 1;count++, step < n;count++, step++)
    {
        Record key = arr[step]; count++;
        int j = step - 1; count++;
        while (count+=2, j >= 0 && key.idNumber < arr[j].idNumber)
        {
            arr[j + 1] = arr[j]; count++;
            j = j - 1; count++;
        }
        arr[j + 1] = key; count++;
    }

    count++; //for return
    printf("Steps: %lld\n", count); //added
    return;
}
```

Insertion Sort Updated Code

```c
void selectionSortWithStep(Record *arr, int n)
{
    long long int step; //added
    int i, j, min; step+=3;
    Record temp; step++;

    for (step++, i = 0; step++, i < (n - 1);step++, i++)
    {
        min = i; step++;
        for (step++, j = i + 1;step++, j < n;step++, j++)
        {
            if (step++, arr[j].idNumber < arr[min].idNumber)
                min = j; step++;
        }
        if (step++, min != i)
        {
            temp = arr[i]; step++;
            arr[i] = arr[min]; step++;
            arr[min] = temp; step++;
        }
    }
    step++; //for return
    printf("Steps: %lld\n", step); //added
    return;
}
```

Selection Sort Updated Code

```c
/* mergeSortWithStep helper function */
 void mergeWithStep(Record *arr, int left, int mid, int right, Record *temp, long long int *step) {
    int i = left, j = mid + 1, k = left; step+=3;

    while (step++, i <= mid && j <= right) {
        if (step++, arr[i].idNumber <= arr[j].idNumber) {
            temp[k++] = arr[i++]; step++;
        } else {
            temp[k++] = arr[j++]; step++;
        }
    }

    while (step++, i <= mid) {
        temp[k++] = arr[i++]; step++;
    }

    while (step++, j <= right) {
        temp[k++] = arr[j++]; step++;
    }

    for (step++, i = left;step++, i <= right;step++, i++) {
        arr[i] = temp[i]; step++;
    }
}

void mergeSortWithStep(Record *arr, int n) {

    long long int step; //added
    Record *temp = (Record *)malloc(n * sizeof(Record)); step++;
    int current_size, left_start; step++; //added

    for (step++, current_size = 1;step++, current_size <= n - 1; step++, current_size *= 2) {
        for (step++, left_start = 0;step++, left_start < n - 1;step++, left_start += 2 * current_size) {
            int mid = (left_start + current_size - 1) < (n - 1) ? (left_start + current_size - 1) : (n - 1); step++;
            int right_end = (left_start + 2 * current_size - 1) < (n - 1) ? (left_start + 2 * current_size - 1) : (n - 1); step++;

            mergeWithStep(arr, left_start, mid, right_end, temp, &step); step++;
        }
    }
    free(temp); step++;
    printf("Steps: %lld\n", step); //added
}
```

Merge Sort Updated Code

```c
/* combSortWithStep helper function */
void combSortShrinkWithStep(int *gap, float shrink, long long int *step)
{
    *gap /= shrink; step++;

    if (step++, *gap < 1)
        *gap = 1; step++;

}

void combSortWithStep(Record *arr, int n)
{
    long long int step; //added
    int gap = n; step++;
    int i; step++;
    float shrink = 1.3; step++;
    int swapped = 1; step++;

    while (step++, gap > 1 || swapped == 1)
    {
        combSortShrinkWithStep(&gap, shrink, &step); step++;
        swapped = 0; step++;

        for (step++, i = 0;step++, i < (n - gap); step++, i++)
        {
            Record temp; step++;

            if (step++, arr[i].idNumber > arr[i + gap].idNumber)
            {
                temp = arr[i]; step++;
                arr[i] = arr[i + gap]; step++;
                arr[i + gap] = temp; step++;
                swapped = 1; step++;
            }
        }
    }
    printf("Steps: %lld\n", step); //added
}
```

Comb Sort Updated Code

V. Comparative Analysis

A comparative analysis of the different algorithms across the different datasets, including theoretical bases that support the observed performance of the algorithms with respect to the datasets. Feel free to use additional datasets to validate the results or generate even more insights. (Yazan)

1. In a data set of 10 - 500 lines,

All 4 sorting algorithms performed similarly in terms of execution time, this is because the data set is relatively small, and time complexity of these sorting algorithms has not become a significant factor yet.

2. In a data set of 25000 lines,

The difference in performance starts to show. Selection sort, despite being one of the less efficient sorting algorithms with a worst-case time complexity of O(n^2), outperforms Insertion sort. Merge sort and Comb sort are considerably faster, as they have better average and worst-case time complexities

3. In a data set if 50000 lines

The differences in performance starts to become even more noticeable than before, as Insertion and Selection sort have significantly higher execution times. Merge sort and Comb sort are still relatively efficient, with Merge sort being notably faster, thanks to its O(nlogn) time complexity.

4. In a data set if 100000 lines

The differences in performances MOST apparent. Insertion and Selection sort become very slow due to their quadratic time complexity. Merge and Comb sort still maintain their efficiency, with Merge sort being faster due to its consistent O(nlogn) time complexity.

VI. Summary of Findings

Summary of findings and the group's learnings, insights, and realizations with respect to sorting algorithms after accomplishing this project. (Yazan + Yves)

To summarize what we got from the different data sets using different sorting algorithms:

- Insertion Sort: Efficient for small data sets but becomes very slow for larger data sets due to its O(n^2) time complexity.
- Selection Sort: Efficient for small data sets but becomes slow for larger data sets due to its O(n^2) time complexity, although during testing with different data sets- Selection sort proved to be a bit quicker than Insertion Sort.
- Merge Sort: The most efficient and consistent sorting algorithm for all data set sizes due to its O(nlogn) time complexity.
- Comb Sort: It is very efficient due to its O(nlogn) time complexity, but Merge sort proved to be superior.

- Stability (Merge vs. Comb)
    - We would like to point out that Merge sort is indeed a stable sorting algorithm. Comb sort on the other hand is not a stable algorithm. We would argue that stability can potentially impact execution time, but its effect is typically minor in most cases. It is also worth noting that because of the nature of merge sort being that it is an out of place sorting algorithm, although it was extremely efficient, memory allocation, specifically larger dataset handling was and may be considered an issue in implementing this algorithm. This is further reinforced by the use of the iterative version of Merge sort rather than the recursive version. Thus, makes it evident that the application of merge sort may be best implemented on linked lists rather than arrays.

- Divide & Conquer Strategy:
    - O(nlogn) algorithms like Merge sort make use of the Divide and Conquer approach. They divide the data into smaller subsets, sort them recursively, and then combine the result.

- Nested Loops:
    - O(n^2) algorithms such as Insertion and Selection sort make use of nested loops, leading to a quadratic growth in execution time. As the data size increases, the number of comparisons and swaps increases quadratically, making them much slower.

VII. Members and Contributions

Member	Contributions
Clavano, Angelica Therese I. (Jack)	1. Comb Sort implementation and description 2. printRecord() in sortingalgorithms.c 3. getFileLineNumber() and code template for running tests 5 times and getting average in main.c 4. Debugging 5. Description of processes (See part III) 6. All modified functions for getting empirical count + running (See IV-B) 7. Running and recording of algorithms and their execution times (See IV-A) including averages on datasets
Homssi, Yazan M.	1. Selection sort implementation and description 2. Comparison of the sorting algorithms' execution time (See V) 3. Debugging 4. Running and recording of algorithms and their execution times (See IV-A) 5. Summary of Findings (See VI)
Young, Henzley Emmanuel S.	1. Insertion sort implementation and description 2. Report introduction (See I) 3. Debugging 4. Running and recording of algorithms and their execution times (See IV-A)
Jimenez, Yves Alvin Andrei A.	1. Merge sort implementation and description 2. Running and recording of algorithms and their execution times (See IV-A) 3. Debugging 4. Summary of Findings (See VI)

VIII. References (APA 7th):

Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell. (n.d.). https://www.bigocheatsheet.com/

Comb Sort Algorithm in C - Sanfoundry. (2022, May 16). Sanfoundry; Sanfoundry. https://www.sanfoundry.com/c-program-implement-comb-sort/

freeCodeCamp.org. (2018). Stability in Sorting Algorithms—A Treatment of Equality. freeCodeCamp.org. https://www.freecodecamp.org/news/stability-in-sorting-algorithms-a-treatment-of-equality-fa3140a5a539/#:~:text=A%20stable%20sorting%20algorithm%20maintains,after%20the%20collection%20is%20sorted

GeeksforGeeks. (2023a). Iterative merge sort. GeeksforGeeks. https://www.geeksforgeeks.org/iterative-merge-sort/

GeeksforGeeks. (2023b). Selection sort data structure and algorithm tutorials. GeeksforGeeks. https://www.geeksforgeeks.org/selection-sort/

GeeksforGeeks. (2023c). Stable and unstable sorting algorithms. GeeksforGeeks. https://www.geeksforgeeks.org/stable-and-unstable-sorting-algorithms/

Khandelwal, V. (2023). What is Merge Sort Algorithm: How does it work, and More. Simplilearn.com. https://www.simplilearn.com/tutorials/data-structure-tutorial/merge-sort-algorithm

Merge Sort-Segmentation fault [Online forum post]. (2014, July 14). Stack Overflow. Retrieved October 23, 2023, from https://stackoverflow.com/questions/24982409/merge-sort-segmentation-fault

Siddiqi. (2021). Types of sorting algorithms (Comparison, Recursive, inplace). CodersLegacy. https://coderslegacy.com/types-of-sorting-algorithms/

Singh, S. (2019, March 17). Comb Sort. OpenGenus; OpenGenus.

https://iq.opengenus.org/comb-sort/

Sort Visualizer. (n.d.). Comb Sort; Comb Sort. Retrieved October 22, 2023, from

https://sortvisualizer.com/

Sorting algorithm. (n.d.). https://www.programiz.com/dsa/sorting-algorithm