Group 8 : CCDSALG Slayers
MCO1 Documentation

Members :
Cansino, Joehanna Llarenas
Dy, Harmony Claire Cheng
Uy, Justine Nicole Chya

I. Introduction

The report is focused on getting the runtime of different sorting algorithms based on different input sizes. The goal of our report is to provide analysis, evaluation, and presentation based on the given datasets. We added 3 more text files with input sizes 5, 25, and 50, to show how the sorting algorithm reacts on different ranges of input sizes. Our report aims to provide insights, findings, and recommendations based on the conducted report and analysis.

II. List of Sorting Algorithms

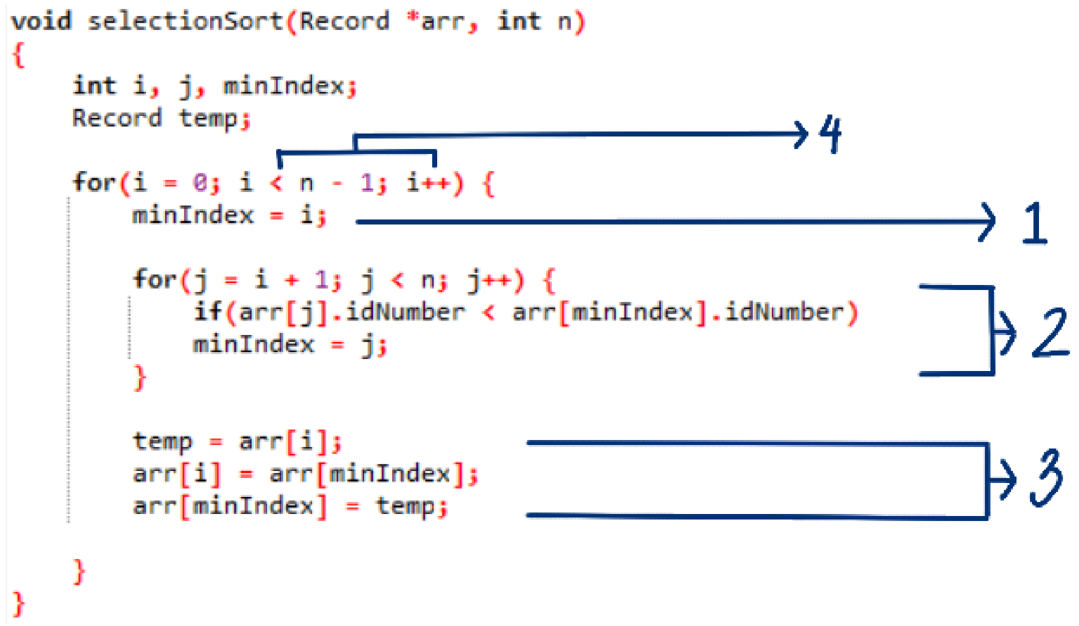
A. Selection Sort

a.1. Description of the Sorting Algorithm

Selection sort uses a variable (**minimum**) to store the index of the element with the smallest valued id number in the unsorted array. The initial minimum index is 0 but the minimum index is changed once an element with a smaller valued id number is found and will therefore change the value of the minimum variable to the index of the newly found minimum. Once the loop has gone through the whole unsorted array, the id number and name of the first element of the unsorted array is swapped with the id number and name of the element with the minimum index. After the swap, the initial minimum index is incremented by 1.

By using the selection sort algorithm, it can be **guaranteed** that after every run of the outer loop, the id number and name that are swapped to the start of the unsorted array are already at their correct position in the sorted array.

a.2 Implementation of the Sorting Algorithm



```
void selectionSort(Record *arr, int n)
{
    int i, j, minIndex;
    Record temp;

    for(i = 0; i < n - 1; i++) {
        minIndex = i;

        for(j = i + 1; j < n; j++) {
            if(arr[j].idNumber < arr[minIndex].idNumber)
            minIndex = j;
        }

        temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;

    }
}
```

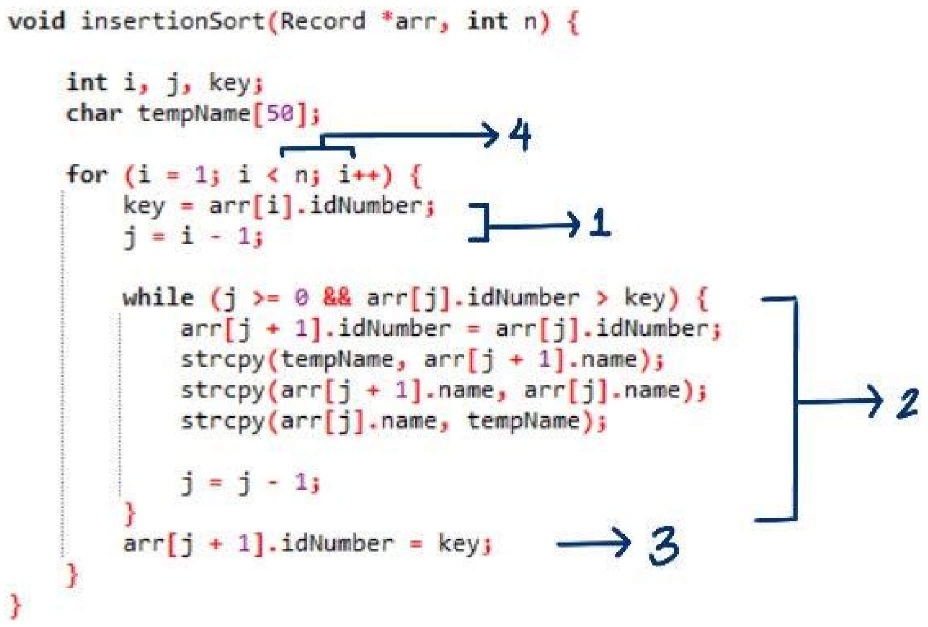
1. Initialize the minimum index to first element of the unsorted array
2. Search for the least valued id number & set minIndex to it's index
3. Swap elements in the first position of the unsorted array with the elements with the minimum index
4. Loop until i is less than n (size of the array) minus 1

B. Insertion Sort

b.1 Description of the Sorting Algorithm

Insertion sort uses a variable (**key**) to store the initial element with index 1 and compares the id number of key to the id number of the elements before it in the array. If the id number of the key is less than the id number of the element being compared to, the id number that key is being compared to will overwrite the id number in the key while swapping the name of both elements, then the key will be compared to the next element. Once the id number in key is greater than the id number of the element being compared to, the id number in key will be inserted into the position of the last element that has overwritten the key then the key's index will increment.

b.2 Implementation of the Sorting Algorithm



```
void insertionSort(Record *arr, int n) {

    int i, j, key;
    char tempName[50];

    for (i = 1; i < n; i++) {
        key = arr[i].idNumber;
        j = i - 1;

        while (j >= 0 && arr[j].idNumber > key) {
            arr[j + 1].idNumber = arr[j].idNumber;
            strcpy(tempName, arr[j + 1].name);
            strcpy(arr[j + 1].name, arr[j].name);
            strcpy(arr[j].name, tempName);

            j = j - 1;
        }
        arr[j + 1].idNumber = key;
    }
}
```

1. Mark first element as sorted and set key to first unsorted element
2. Overwrite the element in key if key is less than element in the sorted array and loop while sorted array has not been gone through or key is greater than element being compared to
3. Insert element in key in array position with index j
4. Loop until i is less than the size of the array

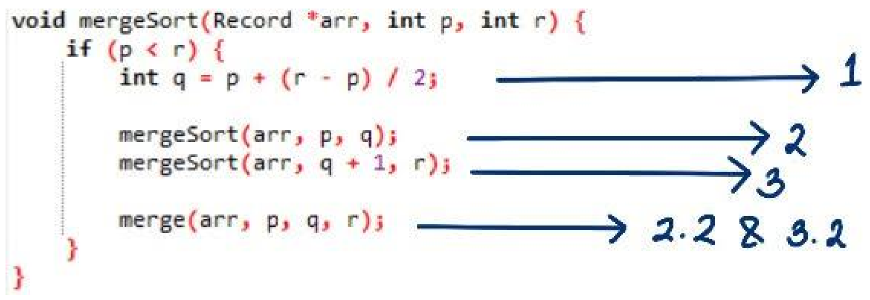
C. Merge Sort

c.1 Description of the Sorting Algorithm

Merge sort uses **recurrence** calls to split the unsorted arrays into half until the elements are in single element array lists. The first occurrence of recurrence calls the merge sort but passes only the left half of the initial array (leftmost element to the middle) which then splits the passed portion of the array. After the passed portion is split into two, merge sort is called again and repeats this cycle until the base case is reached which means that the passed array contains only one element. Once the base case has been reached, a helper function called merge is called which concatenates the two elements while sorting the elements from least to greatest. This repeats until the left half of the initial unsorted array is now sorted and merged back together. Then, the second recurrence line is called which goes through the same process as the left hand side but instead uses the right hand side (middle + 1 to the rightmost element). After the two halves have been sorted, merge is called again to concatenate both halves while sorting from least to greatest.

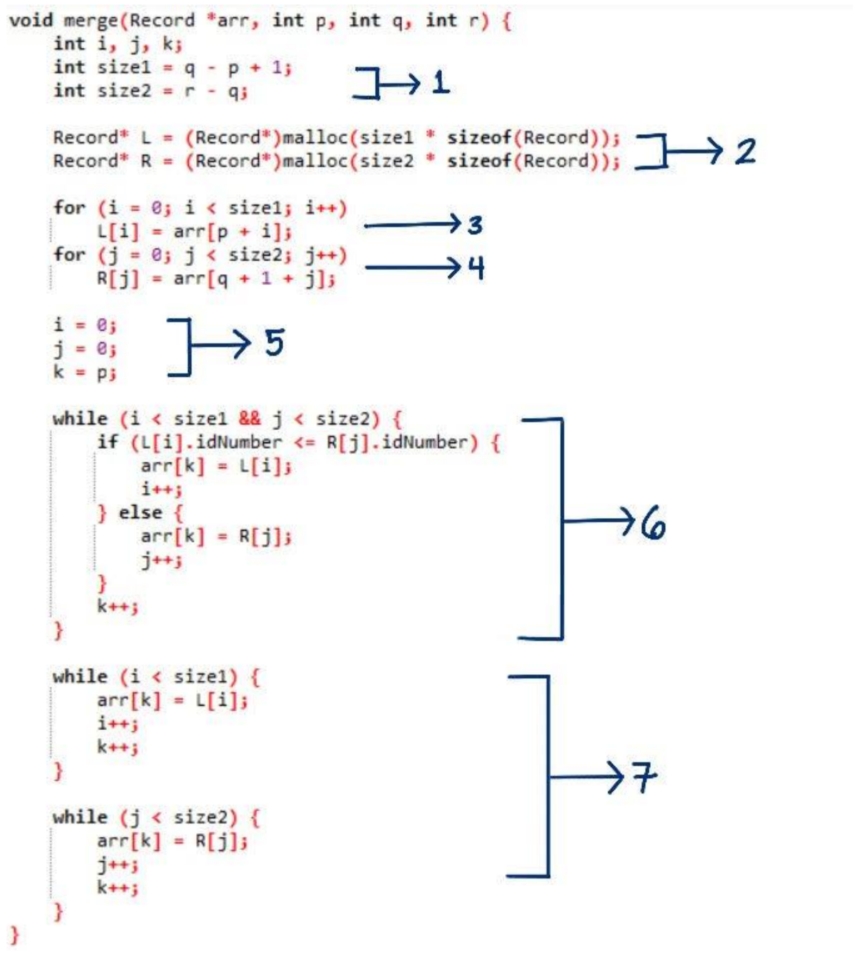
The helper function merge uses two pointers starting from the leftmost element of the two arrays and compares the two id numbers, the lesser valued id number is placed into the larger array and the pointer for the array it came from is incremented. This happens until one of the pointers goes beyond the size of the array it is assigned to.

c.2 Implementation of the Sorting Algorithm



```
void mergeSort(Record *arr, int p, int r) {
    if (p < r) {
        int q = p + (r - p) / 2;

        mergeSort(arr, p, q);
        mergeSort(arr, q + 1, r);

        merge(arr, p, q, r);
    }
}
```

1. Set q as middle of the array from parameters low(p) and high(r) passed
2. Call recursive function mergeSort and pass low(p) and high(q/middle)
   2.1 Step 1-2 is repeated until base case has been reached
   2.2 merge function is called to merge the left hand halve of the initial array while sorting the elements
3. Call recursive function mergeSort and pass low(middle + 1) and high (r)
   3.1 Step 1-2 is repeated until base case has been reached
   3.2 merge function is called to merge the right hand halve of the initial array while sorting the elements
4. End of function



```
void merge(Record *arr, int p, int q, int r) {
    int i, j, k;
    int size1 = q - p + 1;
    int size2 = r - q;

    Record* L = (Record*)malloc(size1 * sizeof(Record));
    Record* R = (Record*)malloc(size2 * sizeof(Record));

    for (i = 0; i < size1; i++)
        L[i] = arr[p + i];
    for (j = 0; j < size2; j++)
        R[j] = arr[q + 1 + j];

    i = 0;
    j = 0;
    k = p;

    while (i < size1 && j < size2) {
        if (L[i].idNumber <= R[j].idNumber) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < size1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < size2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

1. Set size1 and size2 equal to the size of both arrays
2. Allocate enough memory for both left(L) and right(R) arrays
3. Place elements from passed values low(p) and increment until i is equal to size1(L array is full)
4. Place elements from passed values middle(q) + 1 and increment until j is equal to size2(R array is full)
5. Reinitialize i = 0, j = 0 and k = low(p) (to use as pointers when merging L and R to one array)
6. Sorting : If element pointed to by the pointer(i) of L array is less than the element pointed to by pointer(j) of R array, first element of arr array is set to first element of L array and i is incremented; if otherwise, j is incremented – then k is incremented until L or R pointers are over their size
7. Sorting : last element of L or R is placed into arr array and pointer and k is incremented
8. End of function

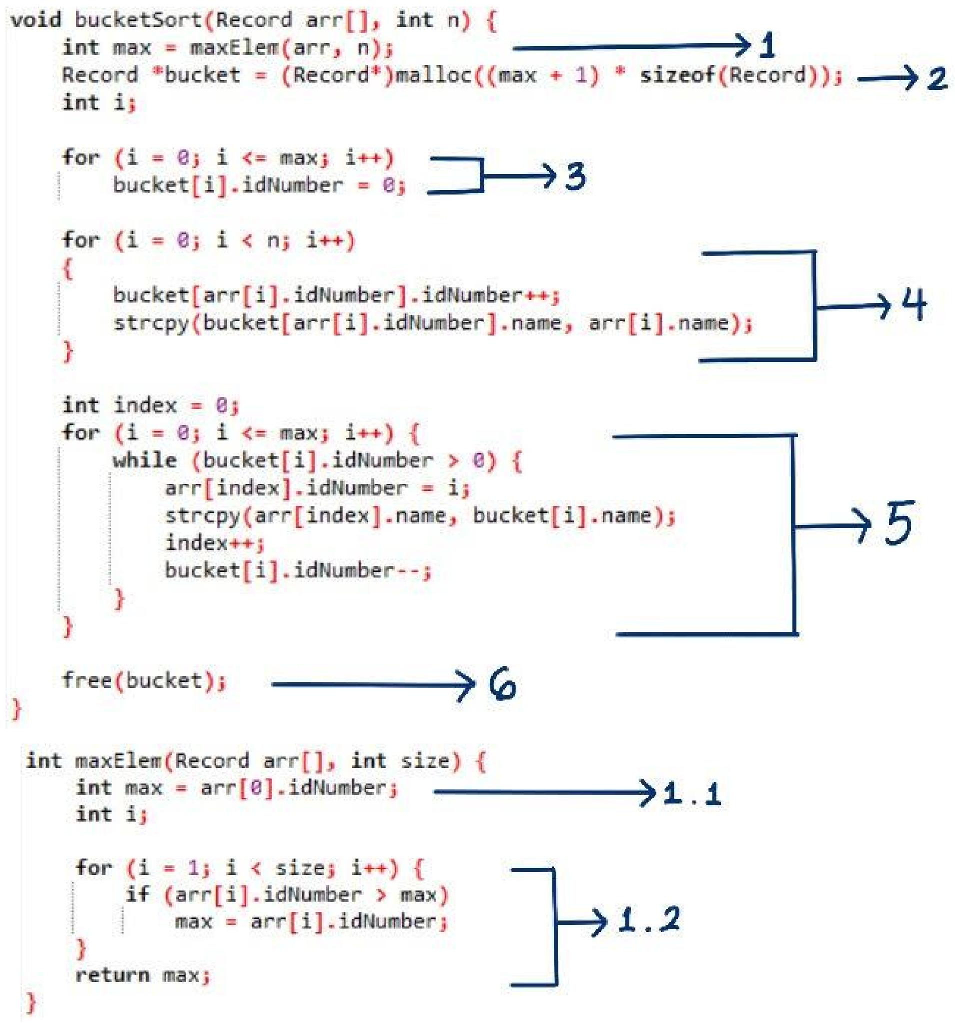
D. Bucket Sort

d.1 Description of the Sorting Algorithm

Bucket sort uses the **index** of the bucket array to place the id numbers in order. To do this, it creates a bucket array that can store an amount equal to the highest valued id number + 1. After, it increments the array elements with indexes equal to the id number which leaves the other bucket elements (indexes not equal to id numbers) still 0; this will be used to determine which elements to input into the final array and in what order as well. When inputting elements into the final array, the index should be equal to the id number which means that the elements with the smaller id number should go into the final array by order while also passing the corresponding name.

Note: The bucket sort algorithm used by this group uses an algorithm more similar to the bucket sort in c referenced from javaTpoint's Bucket Sorting Algorithm which does not use a smaller amount of buckets and sorts with ranges of values, unlike other usual bucket sorting algorithms of other references.

d.2 Implementation of the Sorting Algorithm



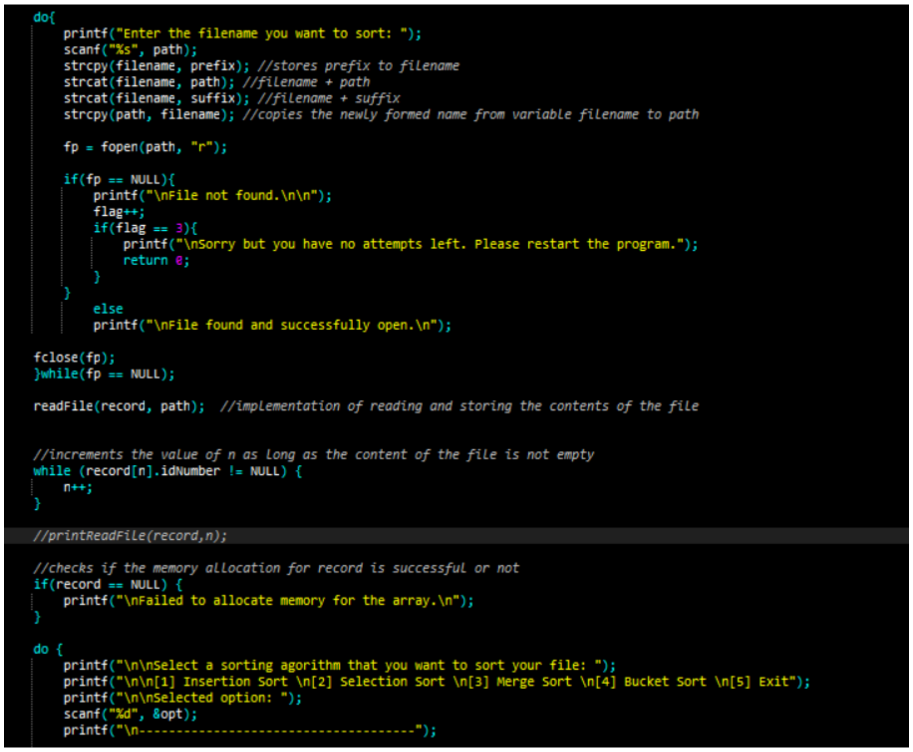
```
void bucketSort(Record arr[], int n) {
    int max = maxElem(arr, n);
    Record *bucket = (Record*)malloc((max + 1) * sizeof(Record));
    int i;

    for (i = 0; i <= max; i++)
        bucket[i].idNumber = 0;

    for (i = 0; i < n; i++)
    {
        bucket[arr[i].idNumber].idNumber++;
        strcpy(bucket[arr[i].idNumber].name, arr[i].name);
    }

    int index = 0;
    for (i = 0; i <= max; i++) {
        while (bucket[i].idNumber > 0) {
            arr[index].idNumber = i;
            strcpy(arr[index].name, bucket[i].name);
            index++;
            bucket[i].idNumber--;
        }
    }

    free(bucket);
}

int maxElem(Record arr[], int size) {
    int max = arr[0].idNumber;
    int i;

    for (i = 1; i < size; i++) {
        if (arr[i].idNumber > max)
            max = arr[i].idNumber;
    }
    return max;
}
```

1. maxElem helper function is called
   1.1 initialize max to the first element of the array
   1.2 Search for greatest valued id number and set that to max – return max
2. Allocate enough memory (max id value + 1); create bucket list
3. Initialize id numbers in bucket list to 0
4. Increment elements with indexes equal to id Number and copy corresponding name to id number
5. If id number in bucket is incremented, push id number and name onto arr array then increment index of arr array and decrement current bucket id number
6. Deallocate bucket memory

III. **Description of Processes**

A. The method(s) by which the algorithms are executed on the datasets

There are four sorting methods that can be used to execute the data sets, namely : selection sort, insertion sort, merge sort, and bucket sort. In order for the algorithms to be run on the datasets, the ‘main’ function or ‘int main’ asks the user for the filename they want to sort. If the file cannot be opened using the obtained path, the user is given three attempts to enter a valid file name. If all attempts fail, the program terminates, but if not, the file is successfully opened. Once the file is opened with the ‘readFile’ function, the number of elements in the file is counted until a null record is reached. The user is then prompted to choose one of the four sorting algorithms or exit the program. Depending on the user’s choice, the corresponding sorting algorithm function is called using the record array and the number of elements as its parameters. The sorting algorithm is then executed on the dataset. The array is then printed using the selected sorting technique. In addition to this, the execution time and the option to select another sorting array will be displayed which will be further explained in succeeding paragraphs. Until the user decides to terminate the program, the process is repeated. To summarize, the code allows its users to test out different sorting algorithms on the same or on different datasets by reading a file each time a sorting method is selected. These algorithms are executed on the dataset within the ‘main function’ as it controls how the sorting algorithms are applied to the dataset. The snippet of the code below is taken from the ‘int main’ or ‘main function of the code which will aid in understanding how the sorting algorithms were executed on the datasets.



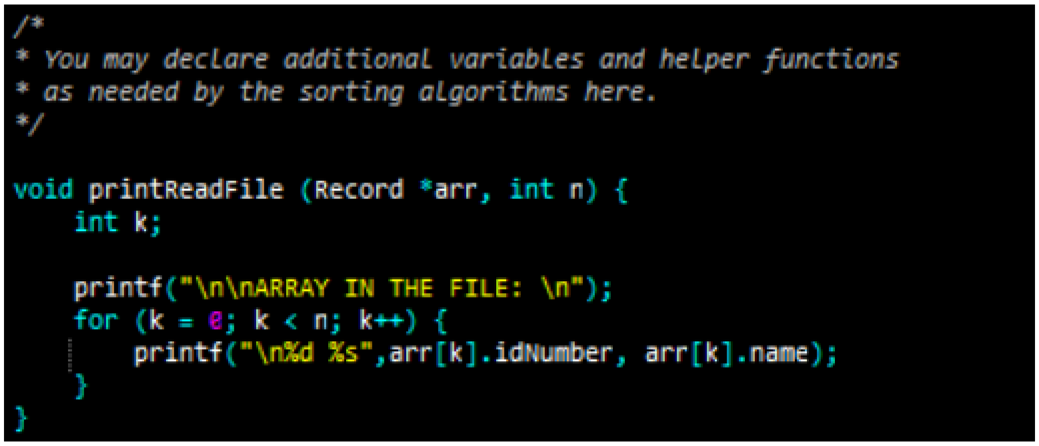
```c
do{
    printf("Enter the filename you want to sort: ");
    scanf("%s", path);
    strcpy(filename, prefix); //stores prefix to filename
    strcat(filename, path); //filename + path
    strcat(filename, suffix); //filename + suffix
    strcpy(path, filename); //copies the newly formed name from variable filename to path

    fp = fopen(path, "r");

    if(fp == NULL){
        printf("\nFile not found.\n\n");
        flag++;
        if(flag == 3){
            printf("\nSorry but you have no attempts left. Please restart the program.");
            return 0;
        }
    }
        else
        printf("\nFile found and successfully open.\n");

fclose(fp);
}while(fp == NULL);

readFile(record, path);  //implementation of reading and storing the contents of the file


//increments the value of n as long as the content of the file is not empty
while (record[n].idNumber != NULL) {
    n++;
}

//printReadFile(record,n);

//checks if the memory allocation for record is successful or not
if(record == NULL) {
    printf("\nFailed to allocate memory for the array.\n");
}

do {
    printf("\n\nSelect a sorting agorithm that you want to sort your file: ");
    printf("\n\n[1] Insertion Sort \n[2] Selection Sort \n[3] Merge Sort \n[4] Bucket Sort \n[5] Exit");
    printf("\n\nSelected option: ");
    scanf("%d", &opt);
    printf("\n-----------------------------------");
```

Snippet Code of int main().

In addition to the datasets already provided, smaller datasets were created (random5, random25, and random50) and used that were identical to the actual data These were utilized in order to execute the code with a smaller sample space and check the correctness of the code faster. The ability and efficacy of the algorithm can be understood by running the algorithms on various data sets with different sample sizes and comparing the performances of the results. It is also possible to look at how the algorithm execution times change as the size of the dataset increases or decreases. To conclude, the code shows a flexible framework for executing sorting algorithms on datasets and the production of varying datasets.

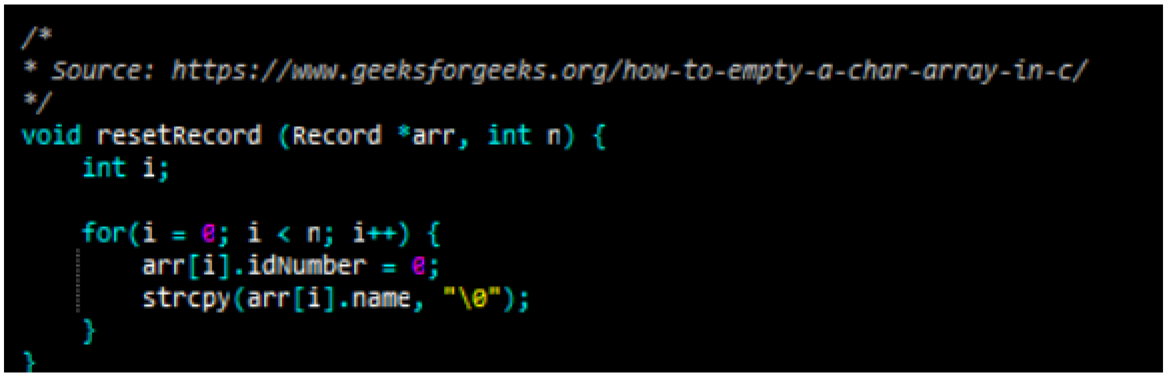
B. The method(s) to confirm the accuracy of the findings

The procedures used to validate the findings' accuracy were mainly done through trial and error and cross checking the datasets using other sorting algorithms. The print statements in the code enables printing and visual verification of the validity of the sorted arrays. With the use of this, the correctness of the code was tested repeatedly until it was determined that it was properly arranged. Another method involved was utilizing different sorting algorithms to cross-check the datasets. As shown in the code below, there is a 'resetRecord' and 'printReadFile' function, both of which function to return the array to its initial state and read the data from the file respectively. With the use of the 'resetRecord' function, the code can reset the array to its original state after using it with a specific algorithm, and then use another sorting algorithm in the same array. This allows the user to verify the validity by comparing the outputs of the several sorting algorithms and check if the outputs produce the same results with the same sample size.



```
/*
* You may declare additional variables and helper functions
* as needed by the sorting algorithms here.
*/

void printReadFile (Record *arr, int n) {
    int k;

    printf("\n\nARRAY IN THE FILE: \n");
    for (k = 0; k < n; k++) {
        printf("\n%d %s",arr[k].idNumber, arr[k].name);
    }
}
```

Code for printReadFile().



```
/*
* Source: https://www.geeksforgeeks.org/how-to-empty-a-char-array-in-c/
*/
void resetRecord (Record *arr, int n) {
    int i;

    for(i = 0; i < n; i++) {
        arr[i].idNumber = 0;
        strcpy(arr[i].name, "\0");
    }
}
```

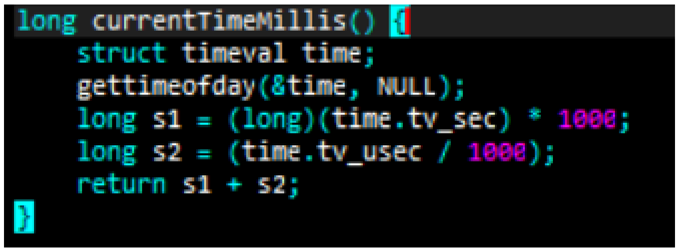
Code for resetRecord().

Here is the following step-by-step process used to verify the accuracy of the findings :

1. Using the ‘printReadFile’ function, the array is printed without any sorting algorithm applied. This shows the original array order.
2. Apply one of the sorting algorithm (insertionSort, selectionSort, mergeSort, or bucketSort)
3. Print the sorted array with the applied sorting algorithm.
4. The array is reseted using the ‘resetRecord’ function call and reverts it back to its original state. This ensures that the next sorting algorithm will be applied starting with the same initial array.
5. With the function ‘readFile’, the data from the file is read and its contents are added to the array. The array will be read in the same way it was initially read because of step 4.
6. Finally, to compare the outcomes with different sorting algorithms, the code is repeated through steps 2-5 but with a different sorting algorithm from the previous one chosen. This enables the accuracy of the findings to be cross-checked, and if an error is found, the code is updated accordingly through trial and error such that the results of the different sorting algorithms provide the same output when using the same sample size.

C. The method(s) to benchmark the execution time

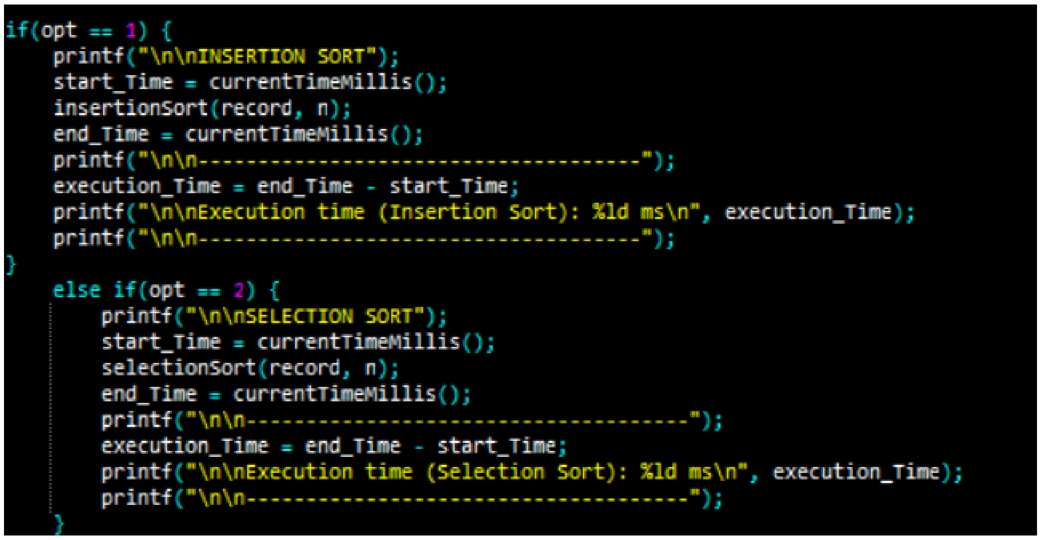
The code shown below is the method used to benchmark the execution duration. According to Tutorialspoint, It was originally based off Java’s System.currentTimeMillis and was converted to C in a stackoverflow question :



```
long currentTimeMillis() {
    struct timeval time;
    gettimeofday(&time, NULL);
    long s1 = (long)(time.tv_sec) * 1000;
    long s2 = (time.tv_usec / 1000);
    return s1 + s2;
}
```

Code for currentTimeMillis().

To measure the execution duration of the sorting algorithms, the program uses a timer. Specifically the helper function in “timer.c” called the ‘currentTimeMillis’ function, given as a starter code specs, was used to record the time. The ‘currentTimeMillis’ function takes note of the current time before and after each sorting algorithm function is called. It is recorded using the start_Time and end_Time variables. The execution time is then given in milliseconds as the difference between the start and end durations. Using the printf statement, the code prints out the name of the sorting method along with the execution time. Here is a sample from the code for better understanding :



```
if(opt == 1) {
    printf("\n\nINSERTION SORT");
    start_Time = currentTimeMillis();
    insertionSort(record, n);
    end_Time = currentTimeMillis();
    printf("\n\n------------------------------------");
    execution_Time = end_Time - start_Time;
    printf("\n\nExecution time (Insertion Sort): %ld ms\n", execution_Time);
    printf("\n\n------------------------------------");
}
    else if(opt == 2) {
        printf("\n\nSELECTION SORT");
        start_Time = currentTimeMillis();
        selectionSort(record, n);
        end_Time = currentTimeMillis();
        printf("\n\n------------------------------------");
        execution_Time = end_Time - start_Time;
        printf("\n\nExecution time (Selection Sort): %ld ms\n", execution_Time);
        printf("\n\n------------------------------------");
    }
```

IV. Execution Times and Frequency Counts

A. Execution Times of Sorting Algorithms using data sets with different sizes

A. Random5

Sorting Algorithms	Average Execution Times
Selection Sort	2 ms
Insertion Sort	2 ms
Merge Sort	3.2 ms
Bucket Sort	866 ms

B. Random25

Sorting Algorithms	Average Execution Times
Selection Sort	7.8 ms
Insertion Sort	8.4 ms
Merge Sort	7 ms
Bucket Sort	3443.6 ms

C. Random50

Sorting Algorithms	Average Execution Times
Selection Sort	12.8 ms
Insertion Sort	13.4 ms
Merge Sort	12.2 ms
Bucket Sort	4419 ms

D. Random100

Sorting Algorithms	Average Execution Times
Selection Sort	19 ms
Insertion Sort	19.6 ms
Merge Sort	18.6 ms
Bucket Sort	3590.6 ms

E. Random25000

Sorting Algorithms	Average Execution Times
Selection Sort	1847.4 ms
Insertion Sort	8902.4 ms
Merge Sort	146.6 ms
Bucket Sort	4461 ms

F. Random50000

Sorting Algorithms	Average Execution Times
Selection Sort	13927.2 ms
Insertion Sort	44721.2 ms
Merge Sort	340.8 ms
Bucket Sort	4806.2 ms

G. Random75000

Sorting Algorithms	Average Execution Times
Selection Sort	38564.2 ms
Insertion Sort	138424.4 ms
Merge Sort	510.8 ms
Bucket Sort	4815.6 ms

H. Random100000

Sorting Algorithms	Average Execution Times
Selection Sort	65084.4 ms
Insertion Sort	217488.4 ms
Merge Sort	868.8 ms
Bucket Sort	4728 ms

I. AlmostSorted

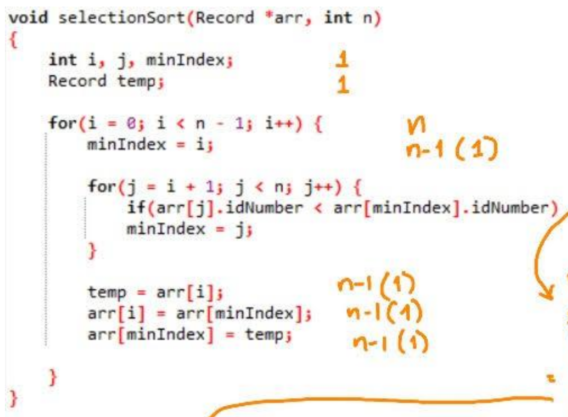
Sorting Algorithms	Average Execution Times
Selection Sort	64147.2 ms
Insertion Sort	39749.6 ms
Merge Sort	1137.8 ms
Bucket Sort	4156.4 ms

J. TotallyReversed

Sorting Algorithms	Average Execution Times
Selection Sort	65721.8 ms
Insertion Sort	457756.4 ms
Merge Sort	1149.2 ms
Bucket Sort	4546.8 ms

B. Frequency Counts of Each Sorting Algorithm

1. Selection Sort



```
void selectionSort(Record *arr, int n)
{
    int i, j, minIndex;                    1
    Record temp;                           1

    for(i = 0; i < n - 1; i++) {                   n
        minIndex = i;                              n-1 (1)

        for(j = i + 1; j < n; j++) {
            if(arr[j].idNumber < arr[minIndex].idNumber)
            minIndex = j;
        }

        temp = arr[i];                     n-1 (1)
        arr[i] = arr[minIndex];            n-1 (1)
        arr[minIndex] = temp;              n-1 (1)

    }
}
```

ex. n=5

	i=0	i=1	i=2	i=3
	5	4	3	2
	8	6	4	2

$$\sum_{i=0}^{n-1} n-i + \sum_{i=0}^{n-1} 8-2i$$

$$= \sum_{i=0}^{n-1} n+8-3i$$

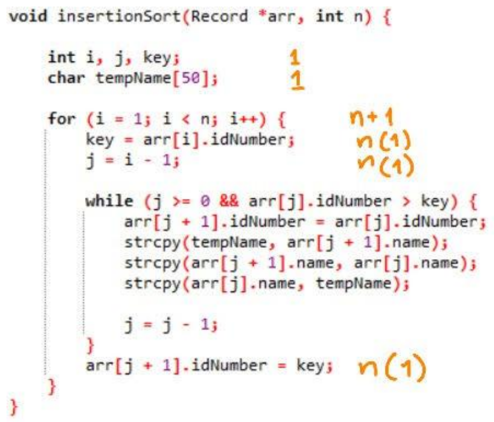
$$\sum_{i=0}^{n-1} n = n^2 \quad + \quad \sum_{i=0}^{n-1} 8 = n-1+0+8 \quad + \quad -3\sum_{i=0}^{n-1} i = \frac{n-1(n)}{2}$$

$$= 2 + 5n - 4 + n^2 + n + 7 - \frac{3n^2-n}{2}$$

$$= n^2 + 6n + 5 - \frac{3n^2-n}{2}$$

$$= \frac{2n^2 + 12n + 10 - 3n^2 - n}{2} = \boxed{\frac{-n^2+11n+10}{2}} \quad O(n^2) ✓$$

2. Insertion Sort



```
void insertionSort(Record *arr, int n) {

    int i, j, key;                1
    char tempName[50];            1

    for (i = 1; i < n; i++) {          n+1
        key = arr[i].idNumber;         n(1)
        j = i - 1;                     n(1)

        while (j >= 0 && arr[j].idNumber > key) {
            arr[j + 1].idNumber = arr[j].idNumber;
            strcpy(tempName, arr[j + 1].name);
            strcpy(arr[j + 1].name, arr[j].name);
            strcpy(arr[j].name, tempName);

            j = j - 1;
        }
        arr[j + 1].idNumber = key;     n(1)
    }
}
```

$$\begin{array}{ccc} j=0 & j=1 & j=2 \\ 2 & 3 & 4 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ \\ 1 & 2 & 3 \end{array}$$

$$\sum_{j=0}^{n} j+2 \qquad 5\sum_{j=0}^{n} j+1$$

$$\downarrow \qquad \downarrow$$

$$\frac{n(n+1)}{2} + n+2 + 5\left(\frac{n(n+1)}{2}\right)$$

$$= 4n+3+n+2 + 6\left(\frac{n^2-n}{2}\right)$$

$$= 5n+5 + \frac{6n^2-6n}{2}$$

$$= 5n+5 + 3n^2-3n$$

$$= \boxed{3n^2+2n+5} \qquad O(n^2) ✓$$

3. Merge Sort

$$T(n) = \begin{cases} 0 & n=1 \\ 2T(n/2)+n & \text{otherwise} \end{cases}$$

$i=1$

$2T(n/2)+n$

$i=2$

$2(2T(n/4)+n)+n$

$2(2(2T(n/8)+n)+n)+n$

$\hookrightarrow 2^i T(n/2^i)+ni$

set $n/2^i = 1$

$\log n(n) = (2^i)\log n$

$\log n = i$

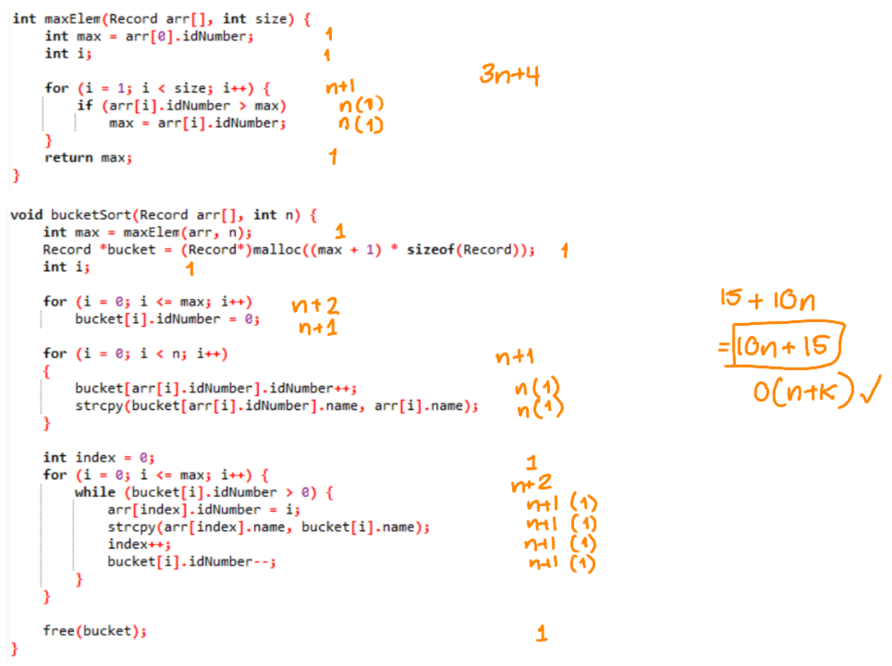
$$2^{\log n} T(n/2^{\log n}) + n\log n$$

$$\cancel{2^{\log n}} \overset{\downarrow}{(0)} + n\log n$$

$$= \boxed{n\log n}$$

$$O(n\log n) ✓$$

4. Bucket Sort



```
int maxElem(Record arr[], int size) {
    int max = arr[0].idNumber;          1
    int i;                              1

    for (i = 1; i < size; i++) {        n+1           3n+4
        if (arr[i].idNumber > max)        n(1)
            max = arr[i].idNumber;        n(1)
    }
    return max;                          1
}

void bucketSort(Record arr[], int n) {
    int max = maxElem(arr, n);             1
    Record *bucket = (Record*)malloc((max + 1) * sizeof(Record));   1
    int i;                 1

    for (i = 0; i <= max; i++)     n+2
        bucket[i].idNumber = 0;     n+1

    for (i = 0; i < n; i++)                               n+1
    {
        bucket[arr[i].idNumber].idNumber++;                n(1)
        strcpy(bucket[arr[i].idNumber].name, arr[i].name); n(1)
    }

    int index = 0;                                         1
    for (i = 0; i <= max; i++) {                          n+2
        while (bucket[i].idNumber > 0) {                    n+1 (1)
            arr[index].idNumber = i;                        n+1 (1)
            strcpy(arr[index].name, bucket[i].name);        n+1 (1)
            index++;                                        n+1 (1)
            bucket[i].idNumber--;
        }
    }

    free(bucket);                                           1
}
```

$15 + 10n$

$= \boxed{10n + 15}$

$O(n+k)$ ✓

V. Comparative Analysis

I. Purpose of the report

a. In this project we want to compare 4 different sorting algorithms. Namely insertion, selection, merge, and bucket sorting algorithm. Comparing them gives us the certainty on which is more efficient to use when it comes to different input sizes.

II. Introduction of entities

a. **Insertion sort** – is a straightforward sorting algorithm that operates similarly to how you would arrange playing cards in your hands. In a sense, the array is divided into sorted and unsorted parts. Values are selected and positioned correctly in the sorted section of the data by selecting them from the unsorted component.

b. **Selection sort** – is a quick and effective sorting method that functions by continually shifting the lowest (or largest) element from the list's unsorted section to the sorted section.

c. **Merge sort** – is a sorting method that divides an array into smaller subarrays, sorts each subarray, and then merges the sorted subarrays back into the original array to get the final sorted array.

d. **Bucket sort** – is a sorting algorithm that splits the items of an unsorted array into several groups known as buckets. Then, each bucket is sorted either by recursively utilizing the same bucket algorithm or by using any of the appropriate sorting algorithms.

III. Criteria

a. Time complexity - Time complexity in sorting algorithms is important because it helps us understand how efficient the algorithm is in terms of the amount of time it takes to sort a given dataset. By comparing the time complexity of sorting algorithms differently, we can select the best algorithm for a particular problem.

b. Space complexity - Space Complexity is vital for an algorithm because when huge data (in real-time) is searched or traversed through an algorithm, quite a large amount of space is needed to hold the inputs and variables along with the code that is being run.

c. Stability – On the basis of stability, a sorting algorithm can be stable or unstable. A stable sorting algorithm maintains the relative order of the items with equal sort keys. An unstable sorting algorithm does not. In other words, when a collection is sorted with a stable sorting algorithm, items with the same sort keys preserve their order after the collection is sorted.

d. Best use cases – Each sorting algorithm has its advantages and disadvantages.

e. Implementation complexity – Difficulty in implementing the sorting algorithm.

f. Runtime - A phase of the programming life cycle is runtime. It is the period of time when a program is simultaneously being executed by all external instructions required for proper operation. Some of these extraneous commands are built into programming languages and are known as runtime environments or runtime systems.

IV. Gathered Data

a. INSERTION SORT

i. Time complexity – O (n^2) both average and worst case, O(n) best case.

ii. Space complexity – O (1).

iii. Stability – Yes, because the Insertion Sort will keep the key value order if there are duplicate elements in the sorting array rather than moving one element in front of the other. The Insertion Sort is a stable sorting algorithm because of this feature.

iv. Best use cases – Insertion sort is best used when the input size is small.

v. Implementation complexity – Simple and easy.

vi. Runtime – get average. (Run it 5 times)

1. **Almostsorted –** (39406 + 39715 + 39879 + 39661 + 40087)/5 = 39749.6 ms
2. **Random5 –** (1 + 2 + 3 + 2 + 2)/5 = 2 ms
3. **Random25 –** (8 + 8 + 9 + 8 + 9)/5 = 8.4 ms
4. **Random50 –** (17 + 16 + 14 + 14 + 6)/5 = 13.4 ms
5. **Random100 –** (27 + 33 + 17 + 10 + 11)/5 = 19.6 ms
6. **Random25000 –** (8799 + 8825 + 8810 + 9007 + 9071)/5 = 8902.4 ms
7. **Random50000 –** (44048 + 45583 + 45047 + 43765 + 45163)/5 = 44721.2 ms
8. **Random75000** - (136052 + 138428 + 136452 + 138119 + 143071)/5 = 138424.4 ms
9. **Random100000 –** (217002 + 233879 + 222971 + 208706 + 204884)/5 = 217488.4 ms
10. **Totallyreversed –** (453616 + 458341 + 460763 + 447034 + 469028)/5 = 457756.4 ms

b. SELECTION SORT

i. Time complexity – O (n^2) best, average, and worst case.

ii. Space complexity – O (1).

iii. **Stability –** No, according to a source it is because it swaps adjacent elements.

iv. Best use cases – Does very well on small lists. Selection sort is useful for determining whether everything has already been sorted. It is particularly beneficial to utilize when memory is at a premium. This is because selection sort uses less temporary storage space since, in contrast to other sorting algorithms, it saves time by only switching items at the very end.

v. Implementation complexity – Easiest to understand.

vi. Runtime – get average. (Run it 5 times)

1. **Almostsorted –** (63733 + 64043 + 66370 + 64443 + 62147)/5 = 64147.2 ms
2. **Random5 –** (2 + 2 + 1 + 3 + 2)/5 = 2 ms
3. **Random25 –** (8 + 7 + 9 + 7 + 8)/5 = 7.8 ms
4. **Random50 –** (14 + 15 + 15 + 14 + 6)/5 = 12.8 ms
5. **Random100 –** (29 + 28 + 16 + 11 + 11)/5 = 19 ms
6. **Random25000 –** (1252 + 1766 + 1735 + 2359 + 2125)/5 = 1847.4 ms

7. **Random50000 –** (13673 + 13904 + 13849 + 13601 + 14609)/5 = 13927.2 ms

8. **Random75000** - (36702 + 37995 + 39581 + 36113 + 42430)/5 = 38564.2 ms

9. **Random100000 –** (65237 + 66299 + 64875 + 64782 + 64229)/5 = 65084.4 ms

10. **Totallyreversed –** (65123 + 66428 + 64805 + 66610 + 65643)/5 = 65721.8 ms

c. MERGE SORT

i. Time complexity – O (n log n) best, average, and worst case.

ii. Space complexity – O (n).

iii. Stability – Yes, because two equal-valued elements show up in the same order in the output sort as they did in the input unsorted array.

iv. Best use cases – When used on smaller lists, merge sort is slower than other sorting options, but it works well when sorting huge lists.

v. Implementation complexity – One of the most efficient sorting algorithms.

vi. Runtime – Get average. (Run it 5 times).

1. **Almostsorted –** (657 + 688 + 781 + 2454 + 1109)/5 = 1137.8 ms

2. **Random5 –** (0 + 0 + 0 + 16 + 0)/5 = 3.2 ms

3. **Random25 –** (0 + 16 + 4 + 15 + 0)/5 = 7 ms

4. **Random50 –** (16 + 15 + 0 + 15 + 15)/5 = 12.2 ms

5. **Random100 –** (31 + 31 + 15 + 0 + 16)/5 = 18.6 ms

6. **Random25000 –** (141 + 140 + 140 + 140 + 172)/5 = 146.6 ms

7. **Random50000 –** (330 + 365 + 328 + 328 + 353)/5 = 340.8 ms

8. **Random75000 -** (375 + 391 + 390 + 578 + 820)/5 = 510.8 ms

9. **Random100000 –** (828 + 734 + 797 + 1188 + 797)/5 = 868.8 ms

10. **Totallyreversed –** (640 + 524 + 816 + 2922 + 844)/5 = 1149.2 ms

d. BUCKET SORT

i. Time complexity – O (n + k) best and average case, O (n^2) worst case.

ii. **Space complexity –** O (n + k).

iii. **Stability –** It depends on the internal sorting algorithm. If the internal is stable then the bucket sort itself is stable.

iv. **Best use cases –** When inputs are uniformly distributed.

v. **Implementation complexity –** The idea of bucket sort is quite simple.

vi. **Runtime –** Get average. (Run it 5 times).

1. **Almostsorted –** (3985 + 4125 + 4687 + 3985 + 4000)/5 = 4156.4 ms
2. **Random5 -** (1047 + 1110 + 1157 + 1047 + 1016)/5 = 866 ms
3. **Random25 –** (3219 + 3656 + 3078 + 3437 + 3828)/5 = 3443.6 ms
4. **Random50 –** (4172 + 4859 + 5110 + 3375 + 4579)/5 = 4419 ms
5. **Random100 –** (2859 + 3156 + 3532 + 4266 + 4140)/5 = 3590.6 ms
6. **Random25000 –** (5000 + 4539 + 4469 + 3797 + 4500)/5 = 4461 ms
7. **Random50000 –** (4390 + 5000 + 4141 + 5672 + 4828)/5 = 4806.2 ms
8. **Random75000 -** (4828 + 4547 + 4312 + 5250 + 5141)/5 = 4815.6 ms
9. **Random100000 –** (4500 + 5140 + 5078 + 4875 + 4047)/5 = 4728 ms
10. **Totallyreversed –** (4906 + 4109 + 4422 + 4813 + 4484)/5 = 4546.8 ms

V. Compare and Analyze

Based on the given datasets and the ones we provided, we can see the difference of the runtime for each of the sorting algorithms we implemented. For the insertion and selection, we can see that the runtime of these two increases as the input size increases. We also noticed that on our implementation, selection sort seemed to have better runtimes than insertion sort this is because of how we implemented the insertion sort. Onto the next, we have the merge and bucket sort. We noticed that they seemed to be slower than the other two sorting algorithms mentioned earlier when it comes to sorting arrays with small input size. Nevertheless, they worked well as the input size increased. Although the runtime of merge and bucket sort also increases as the input sizes increase, we can't see a huge gap between each runtime for each dataset unlike insertion and selection sort where we can notice instantly that each runtime doubles or even triple in size.

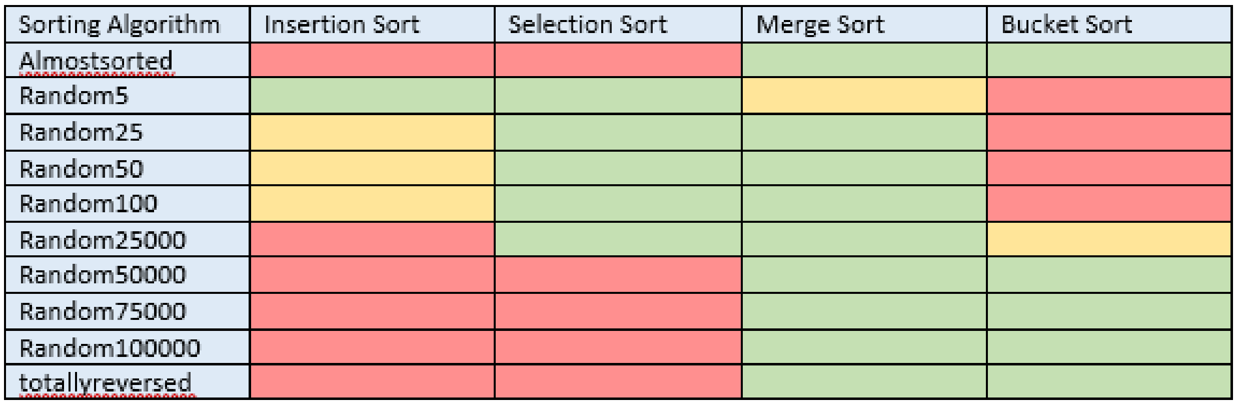
VI. Conclusion

According to our evaluation and analysis we therefore conclude that there are various factors we must consider in taking the runtime of each sorting algorithm. Their runtime can be affected by how we implement it, what kind of hardware we use, the interval time of each run of the program, how many times we run the program, and of course the input size. Based on the acquired runtimes we can say that the fastest and most efficient sorting algorithm for sorting large input sizes is the merge sort. Although we can also consider the bucket sort in sorting large input size because according to an article bucket sort also performs well in large input size. When it comes to small input sizes we can rely on insertion and selection sort. According to some articles, insertion and selection are reliable only on small input sizes.

VII. Result

Criteria:

Red shade – bad Yellow shade – average Green shade - good



Sorting Algorithm	Insertion Sort	Selection Sort	Merge Sort	Bucket Sort
Almostsorted	🟥	🟥	🟩	🟩
Random5	🟩	🟩	🟨	🟥
Random25	🟨	🟩	🟩	🟥
Random50	🟨	🟩	🟩	🟥
Random100	🟨	🟩	🟩	🟥
Random25000	🟥	🟩	🟩	🟨
Random50000	🟥	🟥	🟩	🟩
Random75000	🟥	🟥	🟩	🟩
Random100000	🟥	🟥	🟩	🟩
totallyreversed	🟥	🟥	🟩	🟩

As a result of our evaluation, we can say that in most cases the merge sort is the most reliable. We can also consider using bucket sort because it's most likely that we would sort a huge input size rather than small ones. However, bucket sort can consume much more memory if the elements are not uniformly distributed.

VI. Summary of Findings

After the implementation and the documentation of this project, our group has a more advanced understanding about the different sorting algorithms. One of the more earlier learnings of the group was the main difference of the implementation of the sorting algorithms. While selection sort and insertion sort is similar due to the fact that they both use pointers (pointer for the minimum value for selection and pointer for the position on where to insert the key for insertion), they both still are different when sorting the elements because insertion overwrites and selection swaps. Merge uses recursion to sort the algorithms because it needs to split the array until only it becomes an one element array so recursion is used instead of loops. Bucket sort, instead of using pointers and recursion to implement its sorting method, uses the indexes of the bucket array to arrange the values automatically. Other notable differences are that unlike selection sort and insertion sort which are not very efficient in terms of large samples, merge sort is more efficient with larger data sets. Since both selection sort and insertion sort have time complexities of O(n^2), their time requirement increases quadratically as the sample size grows. Bucket sort on the other hand is dependent on the distribution of elements, making it more variable in terms of performance. Understanding these differences has allowed our group to choose which algorithm is most suitable based on specific requirements and characteristics of the given dataset.

VII. References (APA Format)

Baeldung. (n.d.). Choose the Right Sorting Algorithm for Your Needs. Retrieved from https://www.baeldung.com/cs/choose-sorting-algorithm#:~:text=Introsort%20uses%20a%20suitable%20sorting,when%20the%20iterations%20get%20higher.

Bethune College. (2021). Srijoni-Maitra-CompSc-Q1-Selection-Sort [PDF document]. Retrieved June 24, 2023, from http://www.bethunecollege.ac.in/econtent/2021/srijoniMaitra-Computer/Srijoni-Maitra-CompSc-Q1-Selection-Sort.pdf

DataTrained. (n.d.). Time Complexity of Sorting Algorithms. Retrieved from https://datatrained.com/post/time-complexity-of-sorting-algorithms/#:~:text=The%20time%20complexity%20of%20sorting%20algorithms%20like%20Selection%20Sort%20is,2)%20in%20the%20worst%20case

freeCodeCamp. (n.d.). Stability in Sorting Algorithms: A Treatment of Equality. Retrieved from 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. (2022). *How to Empty a Char Array in C*. Retrieved from. https://www.geeksforgeeks.org/how-to-empty-a-char-array-in-c/

GeeksforGeeks.(n.d).*Insertion Sort*. Retrieved from. https://www.geeksforgeeks.org/insertion-sort/

GeeksforGeeks.(n.d.). *Merge Sort*. Retrieved From. https://www.geeksforgeeks.org/merge-sort/

GeeksforGeeks. (n.d.). Quick Sort vs Merge Sort. Retrieved from. https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/

GeeksforGeeks.(n.d).*Selection Sort*. Retrieved from. https://www.geeksforgeeks.org/selection-sort/

Horsmalahti, P. (2012, June 18). Comparison of Bucket Sort and RADIX Sort. Retrieved from. https://arxiv.org/pdf/1206.3511.pdf#:~:text=Bucket%20sort%20is%20stable%2C%20if,in%20order%20to%20each%20bucket.&text=Counting%20sort%20works%20by%20determining,in%20the%20input%20array%20A.

Interview Kickstart. (n.d.). Bucket Sort Algorithm. Retrieved from https://www.interviewkickstart.com/learn/bucket-sort-algorithm

Interview Kickstart. (n.d.). Stability in Sorting Algorithms. Retrieved from https://www.interviewkickstart.com/learn/stability-in-sorting-algorithms

Javatpoint (n.d.). *Bucket Sort*. Retrieved from.
https://www.javatpoint.com/bucket-sort

Java Revisited. (2017, January 1). Bucket Sort in Java with Example. Retrieved from. https://javarevisited.blogspot.com/2017/01/bucket-sort-in-java-with-example.html#axzz85chMudNC

Matcha. (n.d.). Selection Sort in JavaScript. Retrieved from. https://matcha.fyi/selection-sort-javascript/

NVIDIA Developer Blog. (n.d.). Insertion Sort Explained: A Data Scientist's Algorithm Guide. Retrieved from https://developer.nvidia.com/blog/insertion-sort-explained-a-data-scientists-algorithm-guide/

NVIDIA Developer Blog. (n.d.). Merge Sort Explained: A Data Scientist's Algorithm Guide. Retrieved from . https://developer.nvidia.com/blog/merge-sort-explained-a-data-scientists-algorithm-guide/

Opengenus. (n.d.). Time Complexity of Selection Sort. Retrieved from. https://iq.opengenus.org/time-complexity-of-selection-sort/

Programiz.(n.d).*Bucket Sort*. Retrieved from.
https://www.programiz.com/dsa/bucket-sort

Programiz. (n.d.). Sorting Algorithm - Data Structures & Algorithms. Retrieved from https://www.programiz.com/dsa/sorting-algorithm#:~:text=2.,apart%20from%20the%20input%20data.

ProstDev. (2021, October 29). Reviewing Sorting Algorithms: Insertion Sort. ProstDev. Retrieved from.
https://www.prostdev.com/post/reviewing-sorting-algorithms-insertion-sort

Rodriquez, L. (2018). *What is the time complexity of the counting and merge sort?*. Quora. Retrieved from.
https://www.quora.com/What-is-the-time-complexity-of-the-counting-and-merge-sort

Sehgal, K. (n.d.). An Introduction to Bucket Sort. Retrieved from https://medium.com/karuna-sehgal/an-introduction-to-bucket-sort-62aa5325d124#:~:text=The%20average%20time%20complexity%20for,O(n%2Bk).

Simplilearn. (n.d.). Merge Sort Algorithm. Retrieved from https://www.simplilearn.com/tutorials/data-structure-tutorial/merge-sort-algorithm#:~:text=Merge%20sort%20is%20one%20of,sublists%20into%20a%20sorted%20list.

StackOverflow. (2010). *Enter custom file name to be read?*. Retrieved from. https://stackoverflow.com/questions/2795382/enter-custom-file-name-to-be-read

StackOverflow. (2012). *Get the current time, in milliseconds, in C?*. Retrieved from. https://stackoverflow.com/questions/10098441/get-the-current-time-in-milliseconds-in-c

StudyMite. (n.d.). Space Complexity of an Algorithm - Data Structure. Retrieved from https://www.studymite.com/data-structure/space-complexity-of-an-algorithm

TechTarget. (2021). *Runtime*. Retrieved from. https://www.techtarget.com/searchsoftwarequality/definition/runtime#:~:text=Runtime%20is%20a%20stage%20of,parts%20of%20the%20programming%20language.

Tutorialspoint. (n.d.). *Java.lang.System.currentTimeMillis() Method*. Retrieved from. https://www.tutorialspoint.com/java/lang/system_currenttimemillis.htm

Vadakkanmarveettil, J. (2023, January 14). *What Is Bucket Sort? Simplified Overview In 3 Points | UNext |* UNext. *UNext*. Retrieved from. https://u-next.com/blogs/product-management/bucket-sort/#:~:text=Bucket%20sort%20is%20a%20useful,each%20bucket%20is%20individually%20sorted

VIII. Group Contribution Table

Members	**Contributions**
Joehanna	● Code for the Selection Sort ● Introduction ● Comparative Analysis ● Summary of Findings
Harmony	● Code for the Merge Sort ● Description of Process ● References ● Summary of Findings
Justine	● Code for the Insertion Sort ● Code for the Bucket Sort ● List of Sorting Algorithms

	● Execution times and Frequency Counts ● Summary of Findings