# DESIGN AND ANALYSIS OF ALGORITHMS: GROUP PROJECT REPORT

**Members:**
**Jan Wilhelm T. Sy**
**Chris John Borigas**
**Russel Pispis**
**Jovic Francis B. Rayco**

**BSCS-2A**

# CONTENTS

# Documentation and Structure of Source Code

**INCLUDES**

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>
#include <string.h>

/* PROJECT IN DESIGN AND ANALYSIS OF ALGORITHMS: SOURCE CODE
AUTHORS
- Jan Wilhelm T. Sy
- Chris John Borigas
- Russel Pispis
- Jovic Francis B. Rayco
*/
```

**Header files included in our source code are:**

**stdio.h** - Mainly for input/output.

**stdlib.h** - Mainly for several macros and functions like exit(), rand(), etc.

**time.h** - For use of clock().

**limits.h** - To use ULONG_MAX as the Max Range (Max Range for Unsigned Long Integer).

**METHOD PROTOTYPES**

```c
// ----------------------------- FUNCTION DECLARATIONS -----------------------------
unsigned long int randomNumberGenerator(unsigned long int min, unsigned long int max);
void auxArrInsertion(unsigned long int auxArr[], unsigned long int numArr[], int arrLength);

void selectionSort(unsigned long int auxArr[], int arrLength);
void bubbleSort(unsigned long int auxArr[], int maxRange);
void insertionSort(unsigned long int auxArr[], int arrLength);

void callMergeSort(unsigned long int auxArr[], int arrLength);
void merge(unsigned long int auxArr[], int start, int mid, int end);
void mergeSort(unsigned long int auxArr[], int start, int end);

void heapSort(unsigned long int auxArr[], int n);
void maxHeapify(unsigned long int auxArr[], int i, int n);
void buildMaxHeap(unsigned long int auxArr[], int n);

void callQuickSort(unsigned long int auxArr[], int arrLength);
void quickSort(unsigned long int auxArr[], int left, int right);
int partition(unsigned long int auxArr[], int left, int right);
int medianOfThree(unsigned long int auxArr[], int left, int right);

void swap(unsigned long int *a, unsigned long int *b);
void print(unsigned long int auxArr[], int arrLength);

void runSortingAlgorithm(unsigned long int numArr[], int arrLength, int option);
```

Nearly all of our method prototypes **require** an unsigned long integer array, in order to accommodate for the max range of unsigned long integers.

**MAIN METHOD**

The **Main Method** is split into several sections, one for file handling, another for input handling, and another for array generation and passing.

```
FILE *file;
// File Handling for creating outputfile
if (fopen("sortingFile.txt","r") == NULL)
{
    // Create new file if expected file does not exist
    file = fopen("sortingFile.txt", "w");
}
else
{
    do
    {   // Give the user the option to replace or just append to existing file
        printf("Replace File? [1->Yes | 0->No]\n");
        scanf("%d", &option);
    }
    while(option != 1 && option != 0);
    if (option == 1)
    {
        file = fopen("sortingFile.txt", "w");
    }
    else
    {
        file = fopen("sortingFile.txt", "a");
    }
}
```

The File Handling portion is in charge of creating or opening the file “sortingFile.txt” for appending. The if-else and do while ensures that the user has to decide to keep or replace the file “sortingFile.txt” if said file already exists in the directory. Once the other sections of the program are finished, the file is closed.

```
while (1) {

    // ASK FOR ARRAY LENGTH
    // Label for goto operation
    arrayLength:
    printf("---------------------------------------\n");
    printf("Enter Array Length: ");
    scanf("%d", &arrLength);

    // Input Error Handling
    if (arrLength <= 0 || arrLength > INT_MAX) {
        printf("!! Invalid Array Length !!\n\n");
        // If Invalid Input, goto arrayLength: label
        goto arrayLength;
    }
```

The Input Handling portion is in charge of ensuring that the user inputs accepted inputs. This is achieved with the use of either goto, while, or do-while segments.

The Array Generation portion is in charge of creating the original number array depending on user decision. If the user opts for generating random values, the random number generator method is called with the range being [0,MAXRANGE] where MAXRANGE is ULONG_MAX or the max range of unsigned long integers. The array is also initialized with malloc instead of the usual initialization (numArr[NUMBER]), in order to account for the generation of 1 million size arrays. The group has encountered errors if we do not initialize the array with malloc. The generated arrays in both cases are passed to the sorter.

```
if (option == 1)   {      // a) Values are randomly generated.

    // Set rand() seed as current time.
    srand(time(NULL));

    // Add random numbers into the array.
    for (int i = 0; i < arrLength; i++) {
        numArr[i] = randomNumberGenerator(0, ULONG_MAX);
    }

    // Run the sorting algorithm
    runSortingAlgorithm(numArr, arrLength, 1);

} else if (option == 2)  {  // b) Values are sorted using the formula: N+X, N+2X, N+kX

    printf("Enter 'X' Number: ");
    do{
        scanf("%lu", &xNum);
    } while (xNum < 1);
    printf("\n");

    // Increasing numbers using formula: N+X, N+2X, N+NX, X >= 1
    for (int i = 0; i < arrLength; i++) {
        numArr[i] = arrLength + ((i+1) * xNum);
    }

    // Run the sorting algorithm
    runSortingAlgorithm(numArr, arrLength, 2);
```

A while(1) is used so that the user can generate new arrays once the sorting algorithms are finished, for easier testing. The user has the option to exit the program once an array size is entered.

**RUNNING SORTING ALGORITHMS**

Running the sorting algorithms is delegated to **runSortingAlgorithm()**, which starts with an auxiliary array for array reusability. This method has a file handling section, a sorting algorithm run loop, and the clock integration.

Once the auxiliary array is created, the file opened from the main method is opened for appending. Then ‘fprintf’ is used to append text, unsorted, and sorted arrays in the file.

```
// Run all sorting algorithms
for(int i = 0; i < 6; i++){
    file = fopen("sortingFile.txt", "a");

    clock_t start, end;
    double cpu_time_used;

    auxArrInsertion(auxArr, numArr, arrLength);

    switch (i) {
        case 0:
            fprintf(file,"\n\n[ SELECTION SORT ]\n");
            start = clock();                    // CLOCK START
            selectionSort(auxArr, arrLength);   // Implementing SELECTION SORT
            end = clock();                      // CLOCK END
            printf("Selection Sort Done Processing! \n");
            break;
        case 1:
            fprintf(file,"\n\n[ BUBBLE SORT ]\n");
            start = clock();                    // CLOCK START
            bubbleSort(auxArr, arrLength);      // Implementing SELECTION SORT
            end = clock();                      // CLOCK END
            printf("Bubble Sort Done Processing! \n");
            break;
        case 2:
            fprintf(file,"\n\n[ INSERTION SORT ]\n");
            start = clock();                    // CLOCK START
            insertionSort(auxArr, arrLength);   // Implementing SELECTION SORT
            end = clock();                      // CLOCK END
            printf("Insertion Sort Done Processing! \n");
            break;
```

A for loop is then executed to run all 6 sorting algorithms with the use of a switch case. The switch case is used to slightly reduce redundancy in the code.

All throughout the for loop, the start clock is initialized right before the algorithm call, and the end clock is initialized right after the algorithm call, in order to ensure that the runtime being calculated is that of the sorting algorithm only. The file is opened and closed throughout the loop as well for assurance.

## HELPER METHODS I

**auxArrInsertion** is a function or method whose purpose is to copy elements from an original array to an auxiliary array. auxArrInsertion is used throughout the program to “reinitialize” the array that is passed to the sorting algorithms, ensuring that the sorting algorithms of **1 run** of runSortingAlgorithm are sorting the same array, because as the runSortingAlgorithm() ends, and the user inputs a new array size, and chooses random generation, the next run of runSortingAlgorithm looks at a different array.

```
void auxArrInsertion(unsigned long int auxArr[], unsigned long int numArr[], int arrLength)
{
    for(int i = 0; i < arrLength; i++){
        auxArr[i] = numArr[i];
    }
}
```

**randomNumberGenerator()** returns a random number of range min-max which is made possible with min + rand() % (max-min). max-min determines a limit (or range) for the random numbers (as being within min and max). The mentioned expression assumes that min is 0 when max is a maxRange (which results in a range of 0 to maxRange). The min + portion is to act as an initial value (as min is the smallest possible element).

```
unsigned long int randomNumberGenerator(unsigned long int min, unsigned long int max)
{
    // NOTE: min is assumed to be 0. If The min parameter is more than 0, using the max
    return min + rand() % (max - min);
}
```

**SORTING ALGORITHMS**

**Iterative Algorithms**

**Selection Sort:**

The **selectionSort** function calls the entire comparison for sorting using selection sort. This function takes two parameters: auxArr, which is the array to be sorted, and arrLength, which is the length of the array.

Inside the selectionSort, two loops are present which sorting takes place. Once a new minimum value is found, it is swapped to the element of index i (outer loop). In addition, the variable index holds the value to be used for a condition if a new minimum value is found or not.

```
void selectionSort(unsigned long int auxArr[], int arrLength)
{
    unsigned long int min;
    int index;
    int i;
    int j;
    unsigned long int temp;

    for(i = 0; i < arrLength; i++) {

        // Setting the min as the i'th element.
        min = auxArr[i];
        index = i;

        for(j = i; j < arrLength; j++) {

            // Changing the value of min if another lowest value is found.
            if(auxArr[j] < min) {
                min = auxArr[j];
                index = j;
            }

        }

        // If the index of the min value is different, swap.
        if(index != i) {
            temp = auxArr[i];
            auxArr[i] = min;
            auxArr[index] = temp;
        }

    }

}
```

**Bubble Sort:**

The **bubbleSort** function calls the entire comparison for sorting using bubble sort. This function takes two parameters: auxArr, which is the array to be sorted, and arrLength, which is the length of the array.

The bubbleSort function takes values starting from index 0 to maxRange – 1. Inside the inner loop, for each auxArr[j], it is compared to the index added by 1 (auxArr[j] + 1) if it is greater than or not. If found to be greater than: (auxArr[j] > auxArr[j + 1]), swapping takes place. In addition, a variable named 'checker' is added to build a modified bubbleSort. The variable 'checker' holds a value 0 if bubble sort did not take place and >= 1 values if bubble sort was applied. Once 'checker' is found to be 0, the outer loop will terminate. Finally, the array is sorted.

```
void bubbleSort(unsigned long int auxArr[], int maxRange)
{
    int i = 0;
    int j = 0;
    int checker = 0;

    // Swapping adjacent elements based on comparison result
    for(i = 0; i < maxRange - 1; i++) {
        for(j = 0; j < maxRange - i - 1; j++) {

            if(auxArr[j] > auxArr[j + 1]) {
                swap( a: &auxArr[j],  b: &auxArr[j+1]);

                checker++;
            }

        }
        // To check if array is already sorted
        if(checker == 0) {break;}

    }
}
```

**Insertion Sort:**

The **insertionSort** function calls the entire comparison for sorting using insertion sort. This function takes two parameters: auxArr, which is the array to be sorted, and arrLength, which is the length of the array.

The insertion sort works like playing cards. Using the outer loop [while(i < arrLength - 1)] where 'i' starts at index 1, the inner loop index 'j' will have the value of (i - 1). Index 'j' will move to the left making sure that the value of 'key' is correct according to the values on the left of the index in that given array. If the values on the left are found to be greater than 'key', the value of auxArr[j + 1] will have the value of auxArr[j]. This will continue until index 'j' is not anymore >= 0 && auxArr[j] is not anymore > key. Finally, the value of the key is inserted to the index of auxArr[j + 1] and the array is finally sorted.

```
void insertionSort(unsigned long int auxArr[], int arrLength)
{
    int i = 1;
    unsigned long int key = 0;
    int j = 0;

    // Main Loop
    while(i <= arrLength-1)
    {
        key = auxArr[i];
        j = i - 1;

        // Find correct place for the key
        while(j >= 0 && auxArr[j] > key)
        {
            auxArr[j + 1] = auxArr[j];
            j--;
        }
        auxArr[j+1] = key;
        i++;
    }
}
```

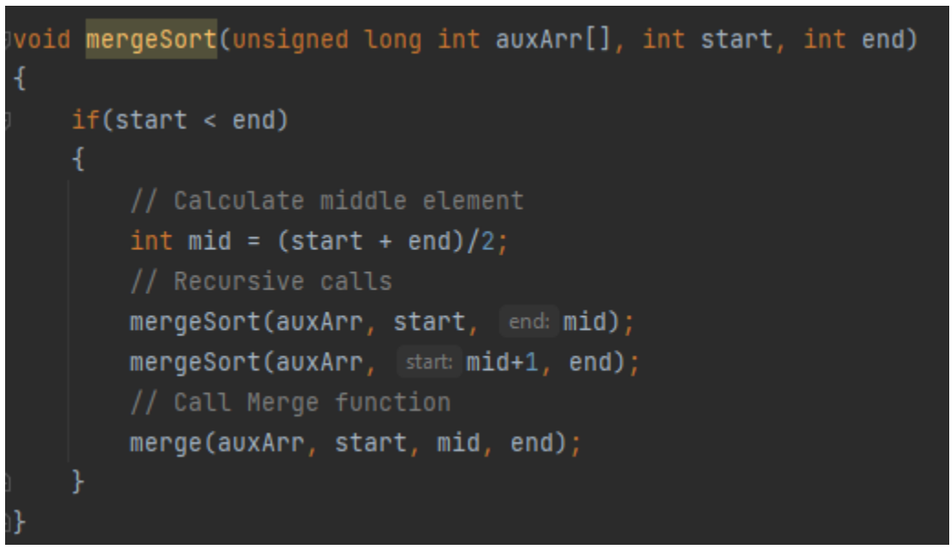
**Recursive Algorithms**

**Merge Sort:**

The **callMergeSort** function calls the mergeSort function. This function takes two parameters: auxArr, which is the array to be sorted, and arrLength, which is the length of the array.

```
void callMergeSort(unsigned long int auxArr[], int arrLength)
{
    // Call main mergesort algorithm
    mergeSort(auxArr, start: 0, end: arrLength - 1);
}
```

The **mergeSort** function takes three parameters: auxArr, which is the array to be sorted, start, which is the starting index of the sub-array to be sorted, and end, which is the ending index of the sub-array to be sorted. This function recursively calls itself on the left and right sub-arrays until the sub-arrays have a length of one or less. When the sub-arrays are of length one or less, the merge function is called to merge the two sub-arrays.




The **merge** function takes four parameters: auxArr, start, mid, and end. The function dynamically allocates memory to two sub-arrays, one for the left sub-array (with the length of mid – start + 1) and one for the right sub-array (with the length of end – mid) The function then merges the two sub-arrays into the auxArr array. To merge the two sub-arrays, the function then iterates over the auxArr array and compares the elements in the left and right sub-arrays. The smaller element is added to auxArr, and the index of the sub-array from which the element was taken is incremented. This process is repeated until all elements have been added to auxArr.

```
void merge(unsigned long int auxArr[], int start, int mid, int end)
{
    int leftLength = mid - start + 1;
    int rightLength = end - mid;

    // Initialized auxillary arrays
    unsigned long int * left = malloc( Size: leftLength*sizeof(unsigned long int));
    unsigned long int * right = malloc( Size: rightLength*sizeof(unsigned long int));
    int i;

    // Place elements
    for(i = 0; i < leftLength; i++)
    {
        left[i] = auxArr[start + i];
    }
    for(i = 0; i < rightLength; i++)
    {
        right[i] = auxArr[mid + 1 + i];
    }

    // Place auxillary array elements to auxArr
    for(int x = 0, y = 0, i = start; i <= end; i++)
    {
        if(x < leftLength && (y >= rightLength || left[x] <= right[y]))
        {
            auxArr[i] = left[x];
            x++;
        }
        else
        {
            auxArr[i] = right[y];
            y++;
        }
    }

    // Free Array Memory
    free( Memory: left);
    free( Memory: right);
}
```

**Quick Sort:**

The source code for the Quick Sort is partially based on the given lecture, but is also based on Hoare's partition scheme after encountering runtime errors when testing the taught quick sort pseudocode.

**callQuickSort()** calls the main function quickSort() and places the appropriate arguments. This function takes the input array and the array length, and passes the appropriate algorithms to the proper quicksort method.

```
void callQuickSort(unsigned long int auxArr[], int arrLength)
{
    // Call main quicksort function
    quickSort(auxArr, left: 0, right: arrLength - 1);

}
```

The **quicksort()** method, if the start index is less than the end index, first calls the partition() method in order to get the pivotIndex used for its two recursive calls. It takes the array, first index (as left) and last index (as right) Its first recursive call includes the pivotIndex in its parameters, meaning that the pivot is still included in the recursive calls, which means that the pivot is not excluded from the swapping phase and that it will not be placed in the i'th position. The pivot in this algorithm is used as reference for the comparisons and swapping in the partition() method.

```
void quickSort(unsigned long int auxArr[], int left, int right)
{
    if (left < right)
    {
        // Partition and get the pivotIndex
        int pivotIndex = partition(auxArr, left, right);
        // Recursive Calls
        // First recursive call includes pivotIndex: Pivot is not excluded in the swapping.
        quickSort(auxArr, left, right: pivotIndex);
        quickSort(auxArr, left: pivotIndex + 1, right);
    }
}
```

The **partition()** method has two sections, initialization and the swapping. In the initialization section, the values of i and j are calculated to be beyond the bounds of left and right (in preparation for the do while loops), and the index of the median-of-three is calculated in order to initialize the pivot. The swapping mostly follows the lecture material's pseudo code logic, but with the notion that the pivot is not excluded in swapping, there was no = condition in the incrementing of i and j. Instead, strictly < or strictly > are used in the conditions of incrementing i and j, including the pivot in the comparisons. When i exceeds j, the index j is returned as the pivotIndex, meaning that the pivot from median-of-three is not necessarily the pivot for the recursive calls. This is an implementation of the Hoare's partition scheme that took a while to dissect and understand, but it resulted in very fast quicksort runtimes.

```
int partition(unsigned long int auxArr[], int left, int right)
{
    // Find Index of Median of Three
    int pivotIndex = medianOfThree(auxArr, left, right);
    // Assign pivot
    unsigned long int pivot = auxArr[pivotIndex];
    // Places arrays out of bounds for the do while iterations
    int i = left - 1;
    int j = right + 1;

    // Will repeat until i >= j after the two do while loops
    while (1)
    {
        // Do while method ensures that an infinite loop is avoided
        // Strict > and < is used as pivot is not excluded in the swapping.
        do
        {
            i++;
        } while (auxArr[i] < pivot);
        do
        {
            j--;
        } while (auxArr[j] > pivot);

        // Return the index j as the pivot index for the recursive calls
        if (i >= j)
        {
            return j;
        }

        swap( a: &auxArr[i],  b: &auxArr[j]);
    }
}
```

The **medianOfThree()** method is used as an attempt to maximize the efficiency of partition(). It returns the index of the array element that fits the median-of-three criteria.

```
int medianOfThree(unsigned long int auxArr[], int left, int right)
{
    // Set mid as the middle index
    int mid = (left + right) / 2;

    if (auxArr[left] > auxArr[right] && auxArr[left] < auxArr[mid])
    {
        // Returns index of left when left is median of three
        return left;
    }
    else if (auxArr[right] > auxArr[left] && auxArr[right] < auxArr[mid])
    {
        // Returns index of right when right is median of three
        return right;
    }
    else
    {
        // Returns index of mid when mid is median of three
        return mid;
    }
}
```

**Heap Sort:**

The source code for the **Heap Sort** is completely based on the given pseudo code from the heap sort lecture. The Heap Sort source code is split into three separate functions: heapSort(), maxHeapify(), and buildMaxHeap().

The first function, **heapSort()** is the main function that calls the other heap sort functions (buildMaxHeap and maxHeapify) while swapping the last element to the root element to sort the array. The function takes two arguments, the array to be sorted (which is auxArray) and the length of the passed array. Then, the buildMaxHeap function is called to convert the array into a max heap. After the heap is built, the root node and the last element are swapped (since the root node is the largest value in the heap), the heap size is decremented and is converted into a max heap, and then the above two steps repeat until the heap only has a size of 1.

```
void heapSort(unsigned long int auxArr[], int n)
{
    // Call buildMaxHeap to convert the auxArr into a maxHeap
    buildMaxHeap(auxArr, n: n-1);

    int i = n-1;
    // Swap the root and last child values of the heap, then heapify
    while (i>0)
    {
        swap( a: &auxArr[0], b: &auxArr[i]);
        maxHeapify(auxArr, i: 0, n: i-1);
        i--;
    }
}
```

The **maxHeapify()** function is the core of the Heap Sort algorithm, and it deviates slightly from the lecture's given pseudo code. It takes the array (auxArr), the index of the node to turn into a heap (i), and the size of the array (n). In the beginning of the code, the main deviation was how the left and right children of a node are calculated. The left child (represented by the 'l') is calculated with 2i + 1 instead of 2i because array indexing in C starts from 0, not from 1. The right child (or 'r') is calculated with 2i + 2 instead of 2i + 1 due to the same reason. After initializing l, r, and the largest variables, the following if-else and if statements determine if the parent node, the left child, or the right child is the one with the largest value (the largest node being stored in the largest variable). The conditional statements that compare l or r to the arraySize are there in order to make sure that the left

and right child are not out of bounds. After the if-else and if statements are finished, if the largest value is not in the parent node, the parent node's value is swapped with the largest child's value, and then the child that had the largest value will be passed to a recursive call to maxHeapify again.

```
void maxHeapify(unsigned long int auxArr[], int i, int n)
{
    int l = 2*i + 1; // left of i
    int r = 2*i + 2; // right of i
    int largest = 0;

    // Comparison of parent value to child values
    if (l <= n && auxArr[l] > auxArr[i])
    {
        largest = l;
    }
    else
    {
        largest = i;
    }
    if (r <= n && auxArr[r] > auxArr[largest])
    {
        largest = r;
    }
    // Swapping when largest node is not the parent node, and heapifying (recursive)
    if (largest != i)
    {
        swap( a: &auxArr[largest], b: &auxArr[i]);
        maxHeapify(auxArr, i: largest, n);
    }
}
```

Finally, **the buildMaxHeap()** function takes the array (auxArr) and the last index of the array (size of the array - 1). First, the last internal node index (or parent node index) is calculated. The calculation is achieved with n/2. There was no need to call a floor function since decimal places that result from an integer division are removed. Once the internal node counter is calculated, all the internal nodes (and root) are maxHeapified (starting from the last internal node) through a while loop and the decrementing internal node counter. After the while loop, a maxHeap is built.

```
void buildMaxHeap(unsigned long int auxArr[], int n)
{
    // maxHeapify the floor(n/2) elements
    int internalCount = (n/2);
    while (internalCount >= 0)
    {
        maxHeapify(auxArr, i: internalCount, n);
        internalCount--;
    }
}
```

**HELPER METHODS II**

**swap()** is used in almost every sorting algorithm that involves swapping (except selection sort). It takes two unsigned long integer addresses and swaps the values of the two unsigned long integers.

```
void swap(unsigned long int*a, unsigned long int*b)
{
    unsigned long int temp = *a;
    *a = *b;
    *b = temp;
}
```

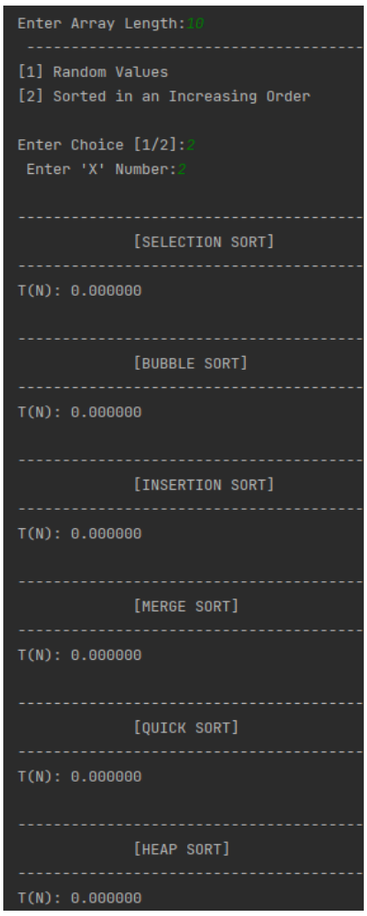
**print()** is a function used to print the given array (and its length) to the output file.

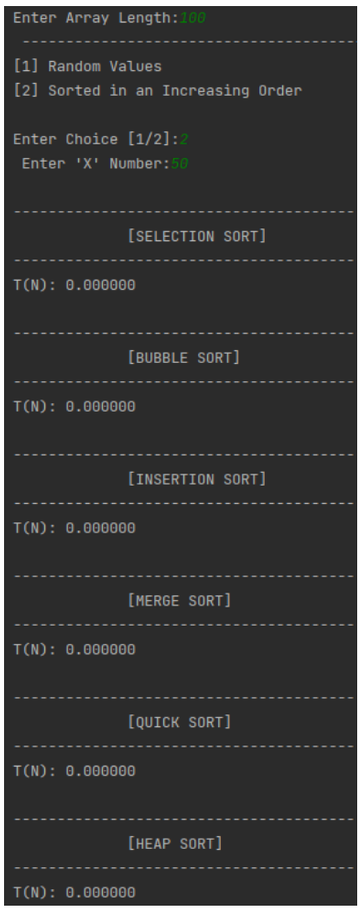
```
void print(unsigned long int auxArr[], int arrLength)
{
    FILE *file;
    file = fopen( Filename: "sortingFile.txt",  Mode: "a");

    for (int i = 0; i < arrLength; ++i)
    {
        fprintf( stream: file,  format: "%lu ", auxArr[i]);
    }
    fprintf( stream: file,  format: "\n");
    fclose( File: file);
}
```

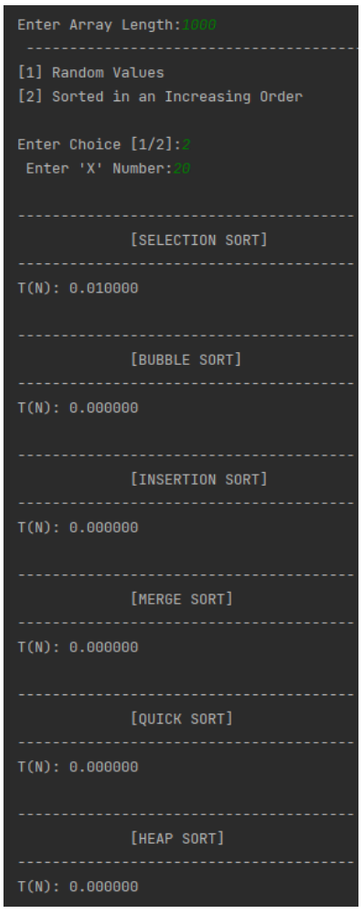
# Screenshots of Actual Execution

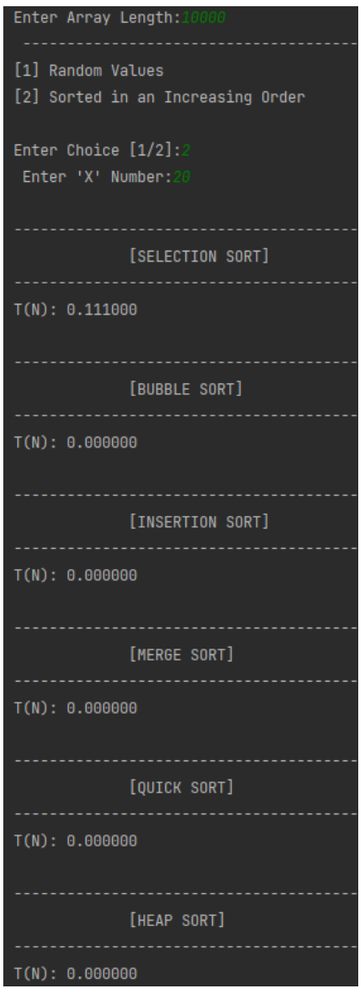
## Old Versions of the Source Code Run

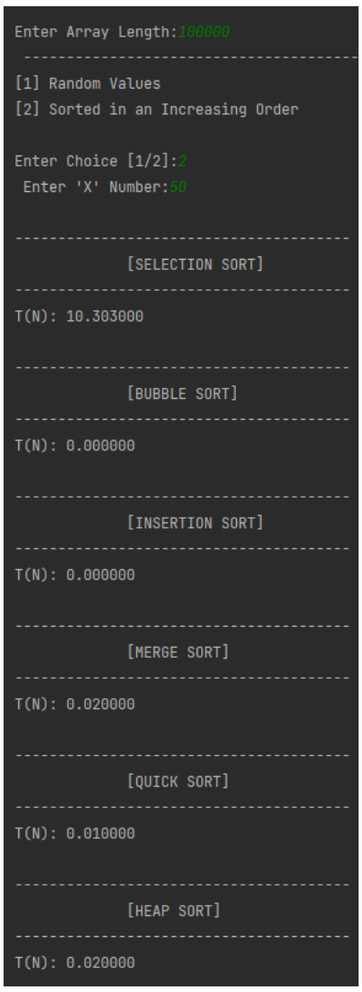
### SORTED ARRAYS

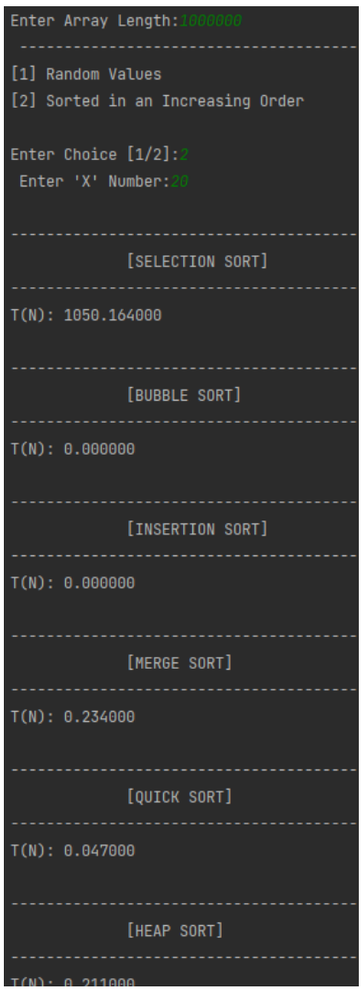


## UNSORTED ARRAY

```
nter Array Length:10
[1] Random Values
[2] Sorted in an Increasing Order

nter Choice [1/2]:1

           [SELECTION SORT]
(N): 0.000000

           [BUBBLE SORT]
(N): 0.000000

           [INSERTION SORT]
(N): 0.000000

           [MERGE SORT]
(N): 0.000000

           [QUICK SORT]
(N): 0.000000

           [HEAP SORT]
(N): 0.000000
```

```
Enter Array Length:100
[1] Random Values
[2] Sorted in an Increasing Order

Enter Choice [1/2]:1

           [SELECTION SORT]
T(N): 0.000000

           [BUBBLE SORT]
T(N): 0.000000

           [INSERTION SORT]
T(N): 0.000000

           [MERGE SORT]
T(N): 0.000000

           [QUICK SORT]
T(N): 0.000000

           [HEAP SORT]
T(N): 0.000000
```

```
Enter Array Length:1000
[1] Random Values
[2] Sorted in an Increasing Order

Enter Choice [1/2]:1

           [SELECTION SORT]
T(N): 0.000000

           [BUBBLE SORT]
T(N): 0.010000

           [INSERTION SORT]
T(N): 0.000000

           [MERGE SORT]
T(N): 0.000000

           [QUICK SORT]
T(N): 0.000000

           [HEAP SORT]
T(N): 0.000000
```

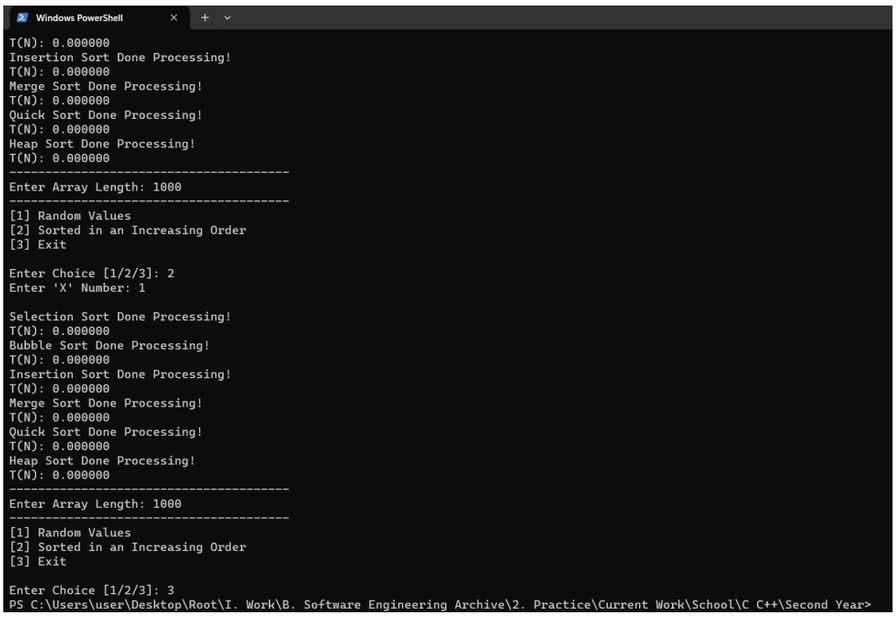
```
Enter Array Length:10000
[1] Random Values
[2] Sorted in an Increasing Order

Enter Choice [1/2]:1

           [SELECTION SORT]
T(N): 0.090000

           [BUBBLE SORT]
T(N): 0.256000

           [INSERTION SORT]
T(N): 0.060000

           [MERGE SORT]
T(N): 0.000000

           [QUICK SORT]
T(N): 0.000000

           [HEAP SORT]
T(N): 0.000000
```

```
nter Array Length:100000
1] Random Values
2] Sorted in an Increasing Order

nter Choice [1/2]:1

           [SELECTION SORT]
(N): 11.326000

           [BUBBLE SORT]
(N): 31.801000

           [INSERTION SORT]
(N): 6.085000

           [MERGE SORT]
(N): 0.025000

           [QUICK SORT]
(N): 0.010000

           [HEAP SORT]
(N): 0.020000
```

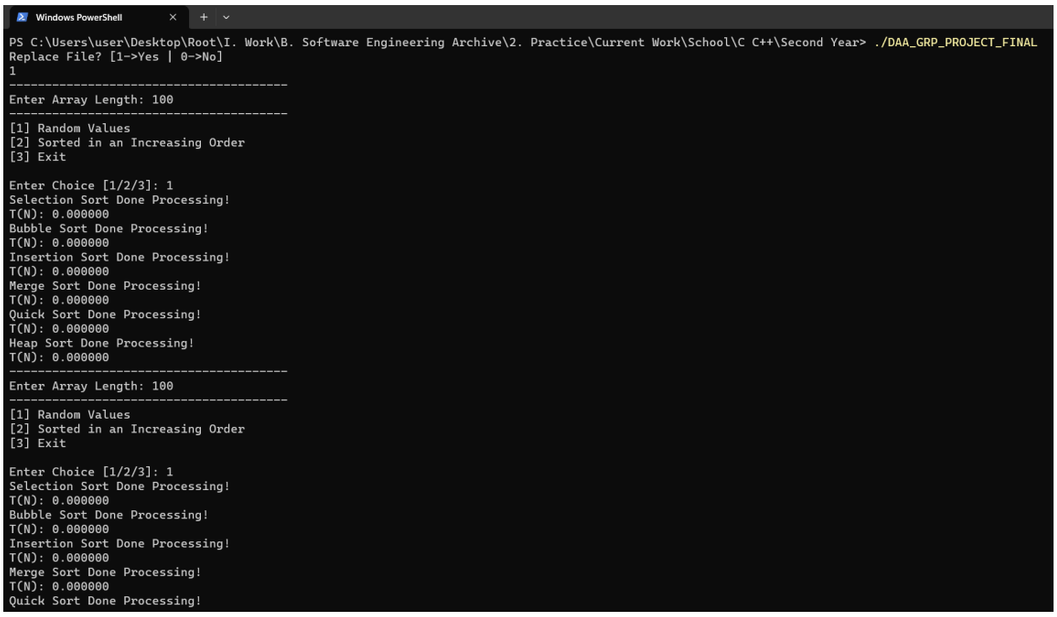
```
nter Array Length:1000000
1] Random Values
2] Sorted in an Increasing Order

nter Choice [1/2]:1

           [SELECTION SORT]
(N): 1072.332000

           [BUBBLE SORT]
(N): 3118.600000

           [INSERTION SORT]
(N): 593.242000

           [MERGE SORT]
(N): 0.318000

           [QUICK SORT]
(N): 0.141000

           [HEAP SORT]
(N): 0.335000
```

# Example Runs of the Final Source Code

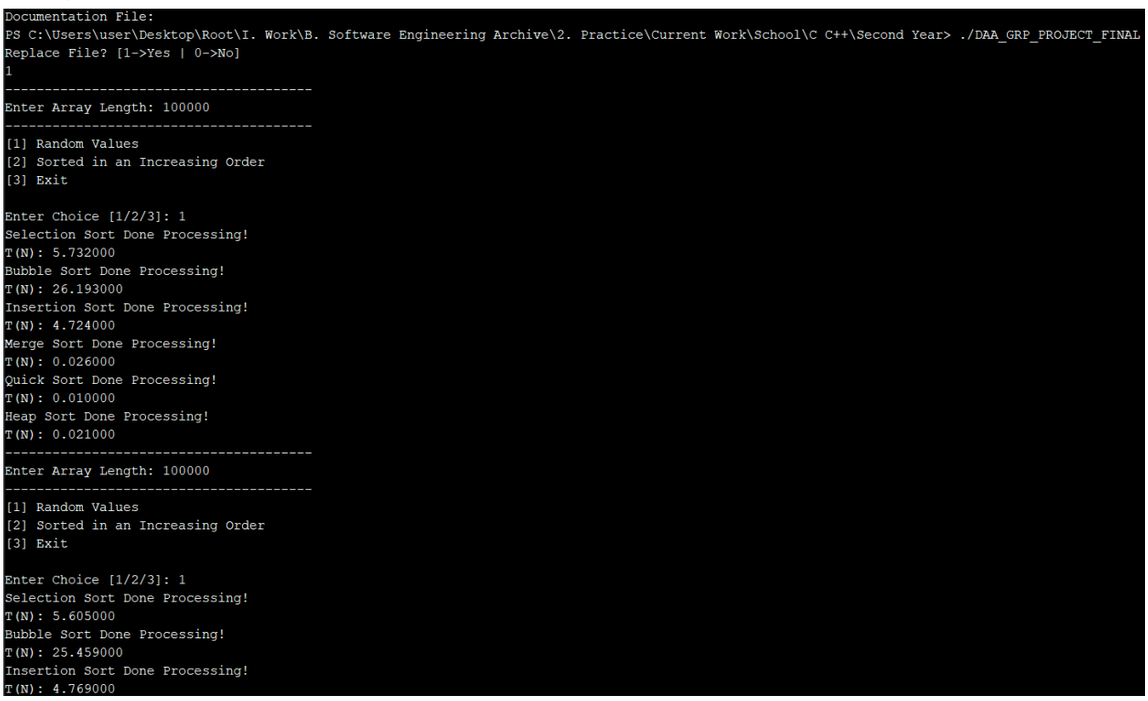
## Testing for Changes in Runtime



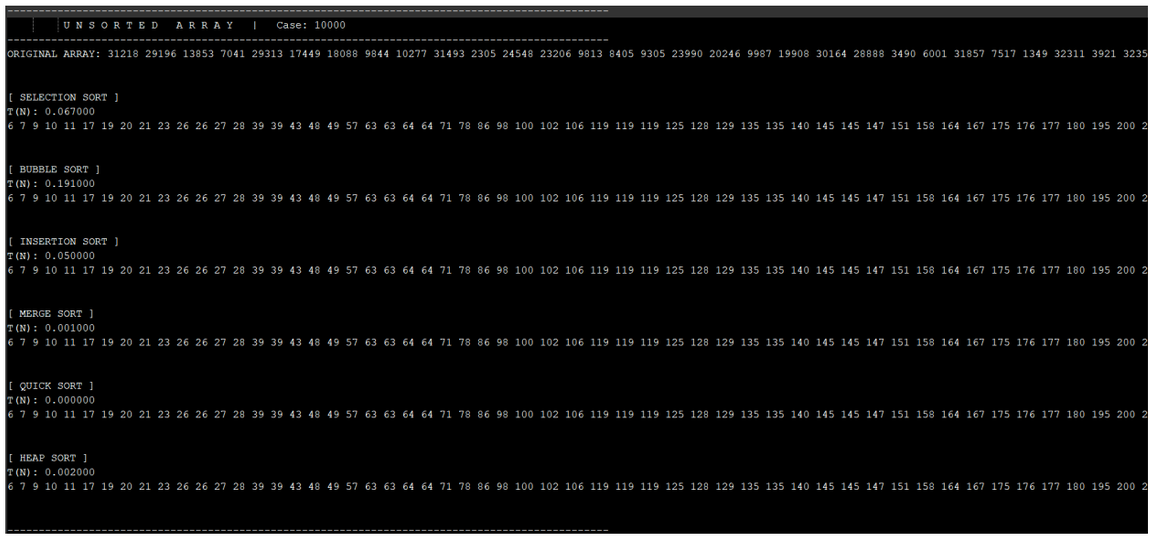

## Example run on 100



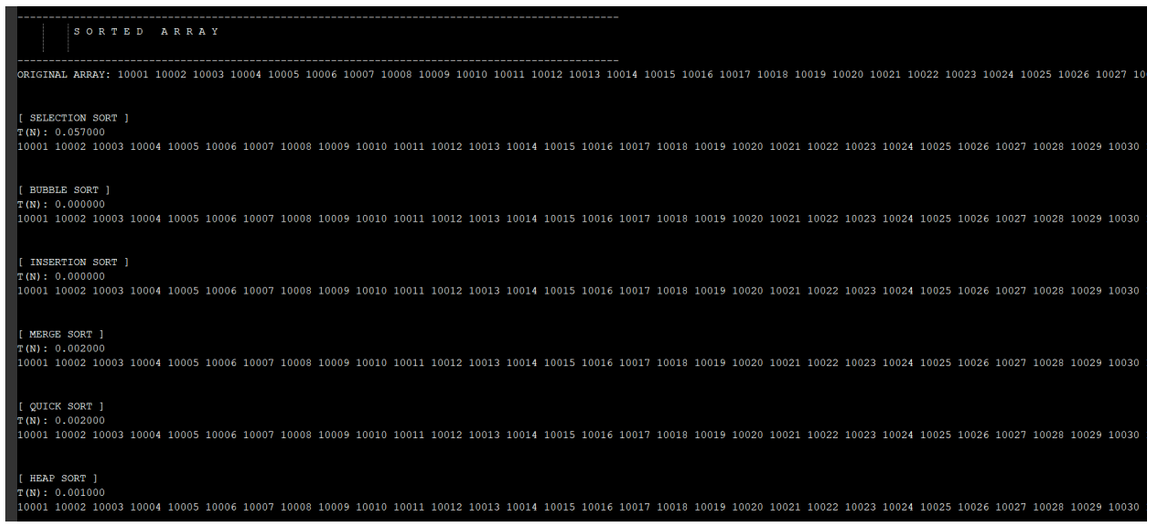

## Example run on 100000



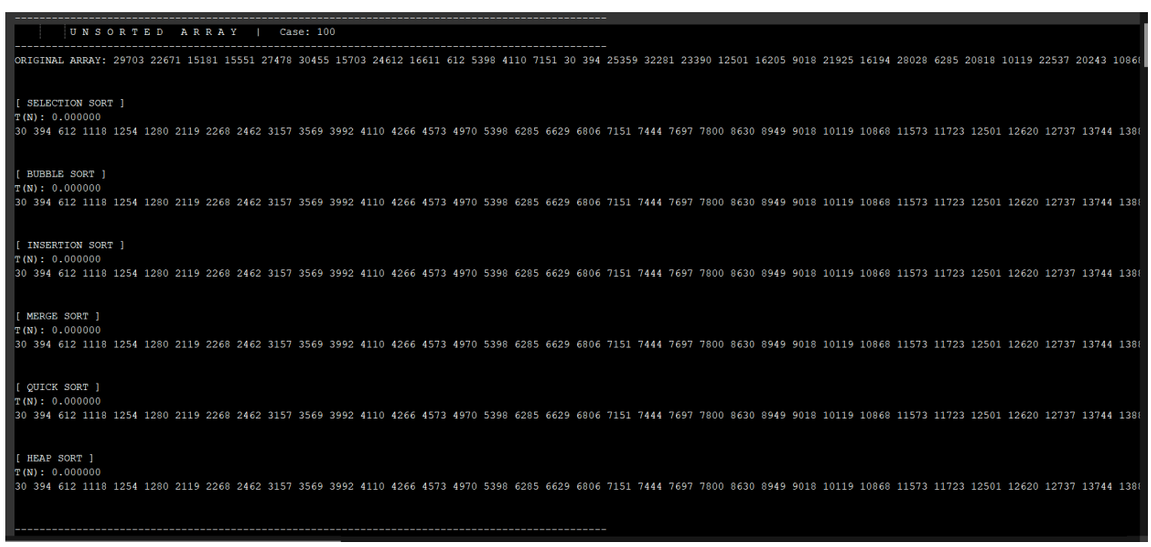
## sortingFile.txt Sample Screenshots



```
--------------------------------------------------------------------------------------------
        U N S O R T E D   A R R A Y   |   Case: 10000
--------------------------------------------------------------------------------------------
ORIGINAL ARRAY: 31218 29196 13853 7041 29313 17449 18088 9844 10277 31493 2305 24548 23206 9813 8405 9305 23990 20246 9987 19908 30164 28888 3490 6001 31857 7517 1349 32311 3921 3235


[ SELECTION SORT ]
T(N): 0.067000
6 7 9 10 11 17 19 20 21 23 26 26 27 28 39 39 43 48 49 57 63 63 64 64 71 78 86 98 100 102 106 119 119 119 125 128 129 135 135 140 145 145 147 151 158 164 167 175 176 177 180 195 200 2


[ BUBBLE SORT ]
T(N): 0.191000
6 7 9 10 11 17 19 20 21 23 26 26 27 28 39 39 43 48 49 57 63 63 64 64 71 78 86 98 100 102 106 119 119 119 125 128 129 135 135 140 145 145 147 151 158 164 167 175 176 177 180 195 200 2


[ INSERTION SORT ]
T(N): 0.050000
6 7 9 10 11 17 19 20 21 23 26 26 27 28 39 39 43 48 49 57 63 63 64 64 71 78 86 98 100 102 106 119 119 119 125 128 129 135 135 140 145 145 147 151 158 164 167 175 176 177 180 195 200 2


[ MERGE SORT ]
T(N): 0.001000
6 7 9 10 11 17 19 20 21 23 26 26 27 28 39 39 43 48 49 57 63 63 64 64 71 78 86 98 100 102 106 119 119 119 125 128 129 135 135 140 145 145 147 151 158 164 167 175 176 177 180 195 200 2


[ QUICK SORT ]
T(N): 0.000000
6 7 9 10 11 17 19 20 21 23 26 26 27 28 39 39 43 48 49 57 63 63 64 64 71 78 86 98 100 102 106 119 119 119 125 128 129 135 135 140 145 145 147 151 158 164 167 175 176 177 180 195 200 2


[ HEAP SORT ]
T(N): 0.002000
6 7 9 10 11 17 19 20 21 23 26 26 27 28 39 39 43 48 49 57 63 63 64 64 71 78 86 98 100 102 106 119 119 119 125 128 129 135 135 140 145 145 147 151 158 164 167 175 176 177 180 195 200 2


--------------------------------------------------------------------------------------------
```



```
--------------------------------------------------------------------------------------------
        S O R T E D   A R R A Y
--------------------------------------------------------------------------------------------
ORIGINAL ARRAY: 10001 10002 10003 10004 10005 10006 10007 10008 10009 10010 10011 10012 10013 10014 10015 10016 10017 10018 10019 10020 10021 10022 10023 10024 10025 10026 10027 10


[ SELECTION SORT ]
T(N): 0.057000
10001 10002 10003 10004 10005 10006 10007 10008 10009 10010 10011 10012 10013 10014 10015 10016 10017 10018 10019 10020 10021 10022 10023 10024 10025 10026 10027 10028 10029 10030


[ BUBBLE SORT ]
T(N): 0.000000
10001 10002 10003 10004 10005 10006 10007 10008 10009 10010 10011 10012 10013 10014 10015 10016 10017 10018 10019 10020 10021 10022 10023 10024 10025 10026 10027 10028 10029 10030


[ INSERTION SORT ]
T(N): 0.000000
10001 10002 10003 10004 10005 10006 10007 10008 10009 10010 10011 10012 10013 10014 10015 10016 10017 10018 10019 10020 10021 10022 10023 10024 10025 10026 10027 10028 10029 10030


[ MERGE SORT ]
T(N): 0.002000
10001 10002 10003 10004 10005 10006 10007 10008 10009 10010 10011 10012 10013 10014 10015 10016 10017 10018 10019 10020 10021 10022 10023 10024 10025 10026 10027 10028 10029 10030


[ QUICK SORT ]
T(N): 0.002000
10001 10002 10003 10004 10005 10006 10007 10008 10009 10010 10011 10012 10013 10014 10015 10016 10017 10018 10019 10020 10021 10022 10023 10024 10025 10026 10027 10028 10029 10030


[ HEAP SORT ]
T(N): 0.001000
10001 10002 10003 10004 10005 10006 10007 10008 10009 10010 10011 10012 10013 10014 10015 10016 10017 10018 10019 10020 10021 10022 10023 10024 10025 10026 10027 10028 10029 10030


--------------------------------------------------------------------------------------------
```



```
--------------------------------------------------------------------------------------------
        U N S O R T E D   A R R A Y   |   Case: 100
--------------------------------------------------------------------------------------------
ORIGINAL ARRAY: 29703 22671 15181 15551 27478 30455 15703 24612 16611 612 5398 4110 7151 30 394 25359 32281 23390 12501 16205 9018 21925 16194 28028 6285 20818 10119 22537 20243 1086


[ SELECTION SORT ]
T(N): 0.000000
30 394 612 1118 1254 1280 2119 2268 2462 3157 3569 3992 4110 4266 4573 4970 5398 6285 6629 6806 7151 7444 7697 7800 8630 8949 9018 10119 10868 11573 11723 12501 12620 12737 13744 138


[ BUBBLE SORT ]
T(N): 0.000000
30 394 612 1118 1254 1280 2119 2268 2462 3157 3569 3992 4110 4266 4573 4970 5398 6285 6629 6806 7151 7444 7697 7800 8630 8949 9018 10119 10868 11573 11723 12501 12620 12737 13744 138


[ INSERTION SORT ]
T(N): 0.000000
30 394 612 1118 1254 1280 2119 2268 2462 3157 3569 3992 4110 4266 4573 4970 5398 6285 6629 6806 7151 7444 7697 7800 8630 8949 9018 10119 10868 11573 11723 12501 12620 12737 13744 138


[ MERGE SORT ]
T(N): 0.000000
30 394 612 1118 1254 1280 2119 2268 2462 3157 3569 3992 4110 4266 4573 4970 5398 6285 6629 6806 7151 7444 7697 7800 8630 8949 9018 10119 10868 11573 11723 12501 12620 12737 13744 138


[ QUICK SORT ]
T(N): 0.000000
30 394 612 1118 1254 1280 2119 2268 2462 3157 3569 3992 4110 4266 4573 4970 5398 6285 6629 6806 7151 7444 7697 7800 8630 8949 9018 10119 10868 11573 11723 12501 12620 12737 13744 138


[ HEAP SORT ]
T(N): 0.000000
30 394 612 1118 1254 1280 2119 2268 2462 3157 3569 3992 4110 4266 4573 4970 5398 6285 6629 6806 7151 7444 7697 7800 8630 8949 9018 10119 10868 11573 11723 12501 12620 12737 13744 138


--------------------------------------------------------------------------------------------
```

# Output Analysis

### Tables

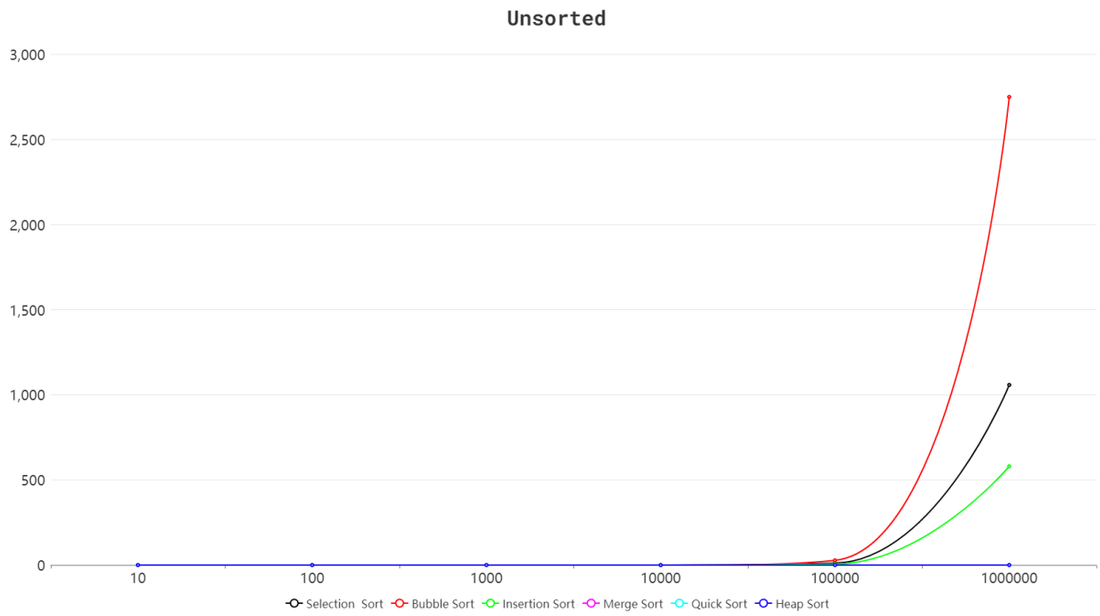
Average Running Time for an Input Array that is Random (Unsorted)

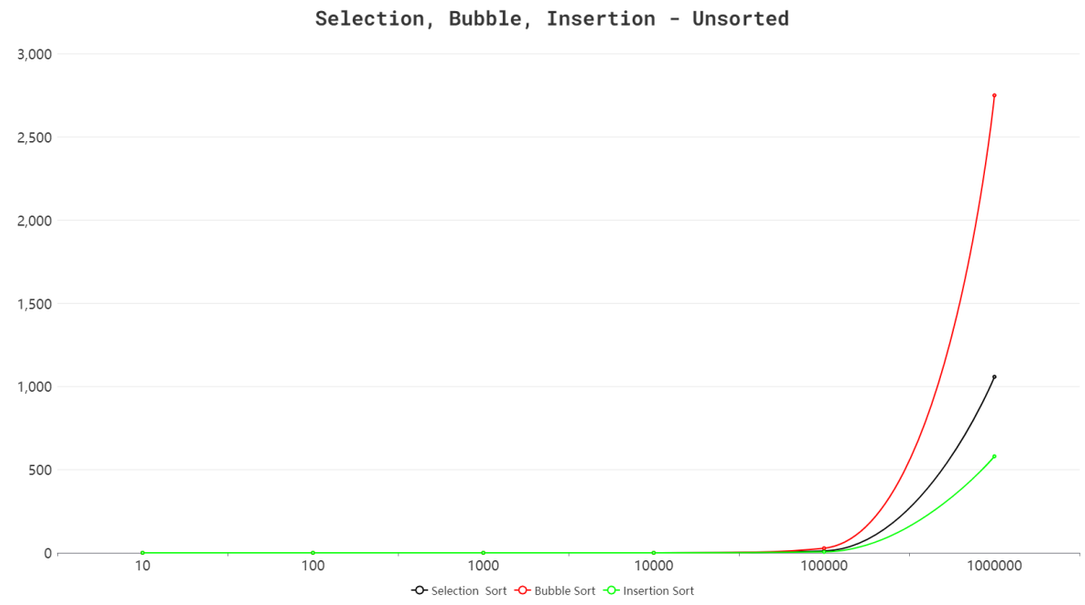
| N | Selection Sort | Bubble Sort | Insertion Sort | Merge Sort | Quick Sort | Heap Sort |
|---|---|---|---|---|---|---|
| 10 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 100 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 1000 | 0.004000 | 0.002000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 10000 | 0.143200 | 0.279200 | 0.069200 | 0.002000 | 0.000000 | 0.000000 |
| 100000 | 10.891600 | 27.671600 | 5.706200 | 0.024000 | 0.006000 | 0.019400 |
| 1000000 | 1058.019200 | 2750.394000 | 580.099000 | 0.279400 | 0.090600 | 0.199600 |

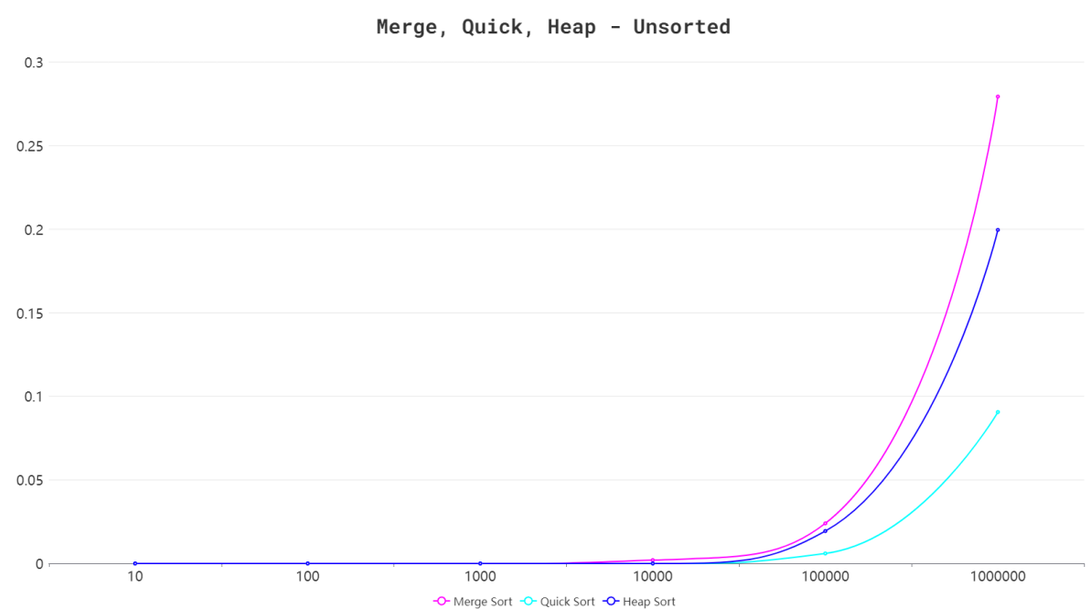
Average Running Time for an Input Array that is Sorted

| N | Selection Sort | Bubble Sort | Insertion Sort | Merge Sort | Quick Sort | Heap Sort |
|---|---|---|---|---|---|---|
| 10 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 100 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 1000 | 0.002000 | 0.002000 | 0.000000 | 0.002000 | 0.000000 | 0.000000 |
| 10000 | 0.131200 | 0.000000 | 0.000000 | 0.002000 | 0.000000 | 0.006000 |
| 100000 | 10.704800 | 0.000000 | 0.000000 | 0.028000 | 0.006000 | 0.020000 |
| 1000000 | 1067.348600 | 0.001800 | 0.004000 | 0.241200 | 0.126800 | 0.194000 |

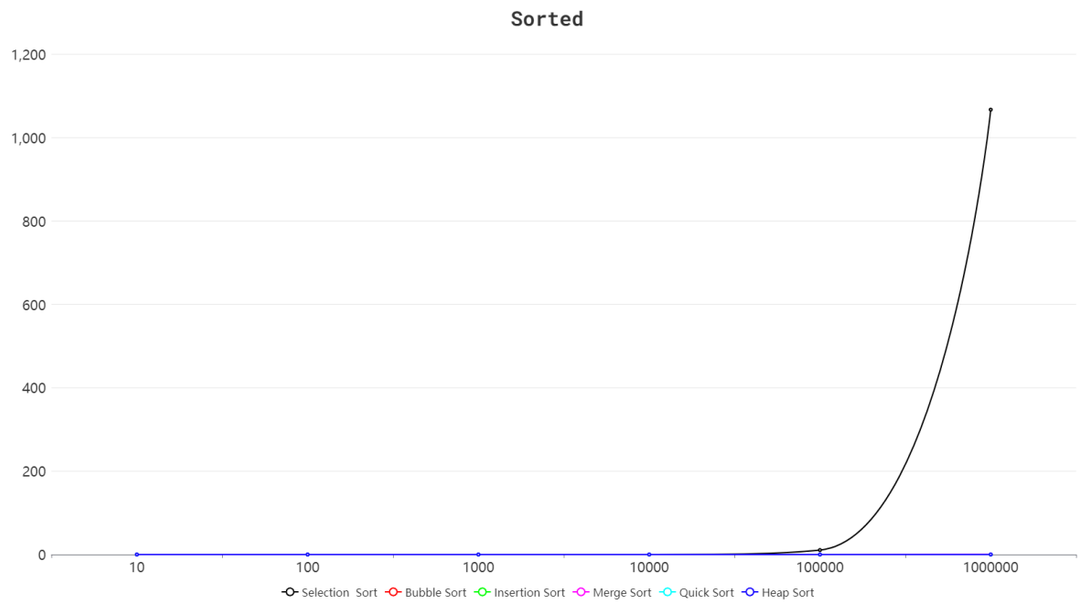
## Graphs
## Unsorted:

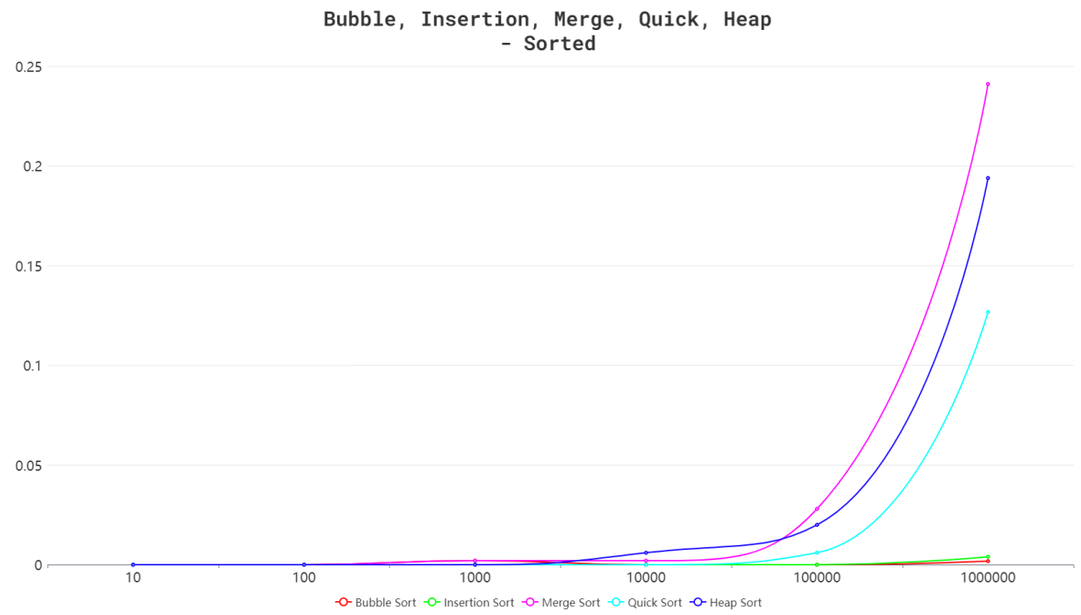


## Sorted:


Sorted
1,200
1,000
800
600
400
200
0
10
100
1000
10000
100000
1000000
Selection Sort
Bubble Sort
Insertion Sort
Merge Sort
Quick Sort
Heap Sort


Bubble, Insertion, Merge, Quick, Heap
- Sorted
0.25
0.2
0.15
0.1
0.05
0
10
100
1000
10000
100000
1000000
Bubble Sort
Insertion Sort
Merge Sort
Quick Sort
Heap Sort

**Discussion**

**SELECTION SORT**

In both unsorted and sorted arrays, selection sort is consistent in its runtime. Its runtime at a 1 million-sized unsorted array is nearly or approximately the same as its runtime at a 1 million-sized sorted array. That said, we have observed that when the array size increases the runtime of the selection sort algorithm significantly increases, which is within the expectations of an O(n^2) algorithm and is further evidenced by the graph of selection sort.

For example, its runtime differences between 10000 and 100000. Its average runtime at N = 100000 is approximately 100 times slower than its average runtime at N = 10000, clearly showing that selection sort will be significantly slower at N = 1000000. Its average runtime at N = 1000000 is approximately more than 16 minutes, and is also around 100 times slower than at N = 100000.

At small size arrays, selection sort in the output table stops having approximately 0 average runtime starting at N = 1000, which is a subtle sign that its runtime gets significantly worse at higher array sizes.

**BUBBLE SORT**

The behavior of our bubble sort algorithm (modified bubble sort) is different depending on if the array is already sorted or not. However, its average runtime across unsorted arrays is still not desirable (due to how time consuming it is). In an unsorted array of size N = 1000000, bubble sort runs for around 45+ minutes, which is approximately triple the runtime of selection sort. Running bubble sort and selection sort can take 1 hour per run, which is not good compared to the other sorting algorithms. Bubble sort and the selection sort algorithms are also seen to be the fastest growing among all algorithms as array size increases, even when it seems to be slower growing than selection sort at N = below 1000.

In unsorted arrays, bubble sort is similar to selection sort (a fellow O(n^2) algorithm) in behavior. As the array size grows from 100000 to 1000000, the runtime becomes approximately 100 times slower. This is shown with bubble sort’s runtime of approximately 27 seconds at N = 100000, and a runtime of

approximately 45 minutes (approximately 2700 seconds) at N = 1000000.

When the array is already sorted, bubble sort has near 0 runtime at N = 1000000 and at lower sizes, which makes sense due to our bubble sort algorithm stopping when the array is already sorted (with the use of a checker). But, in unsorted arrays, it is worse than selection sort, especially at N = 10000 and above array sizes. This was the most difficult algorithm to run due to its runtime. According to a groupmate, bubble sort on his machine ran for 4500 seconds (more than an hour), which is quite severe. It is suspected that bubble sort took too long due to the many adjacent element swapping that occurred in the approximately 1 million (or half a million) passes of bubble sort.

**INSERTION SORT**

Though the insertion sort is an O(n^2) algorithm, its runtime is different (and better) than selection and bubble sort. For once it runs twice as fast than selection sort at N = 1000000, only taking approximately 7-8 minutes on average. And it needed no modifications (unlike bubble sort) to have its runtime be near 0 when the array is sorted. In the graph, it is also the slowest growing among the iterative / non-recursive sorting algorithms.

However, it is still an O(n^2) algorithm because its runtime increases or becomes 100 times slower when the size increases by 10 times. From N = 10000 to N = 100000, the runtime becomes 100 times slower.

It is suspected that insertion sort is faster than bubble sort and selection sort as insertion sort does not necessarily scan the entire array when it sorts an array, which is unlike selection sort that finds a minimum at the entire unsorted portion of the array, and bubble sort that will swap as soon as adjacent elements are not in the proper order. Insertion sort mostly looks at its sorted array, only taking the adjacent element in the unsorted array and inserting it at the right place in the sorted array.

**MERGE SORT**

Merge Sort consistently ran for less than 1 second for all N sizes, including 1 Million. However, in sorted arrays, it had nearly the same average runtime (similar to selection sort’s

behavior in both sorted and unsorted arrays). This shows that while merge sort is a quick sorting algorithm, when compared to the other recursive sorting algorithms, merge sort seems to be the slowest. Merge sort in this case is also the fastest growing among the recursive sorting algorithms.

As the array size changes from N = 10000 to N = 100000, the runtime of merge sort only becomes 10 times slower, instead of the iterative algorithms' having 100 times slower runtime when the size increases in size. This shows that merge sort is not the same as the iterative sorting algorithms as an O(nlogn) algorithm.

**QUICK SORT**

Quick Sort, with the median-of-three pivot, was unexpectedly fast. It is the fastest recursive sorting algorithm in the table and the graph (as it is the slowest growing). It ran significantly faster than merge and heap sort in both sorted and unsorted arrays. However, quicksort has a 'hiccup' in its runtime in the table, where it had a slightly slower average runtime in the sorted array table. This might be a sign that a bad pivot was chosen during the sorting process, which is likely inevitable even with the median-of-three pivot.

Similar to merge sort and selection sort, its behavior remains rather consistent regardless if the array is already sorted or not, and at our 10 times of running quick sort (5 for sorted and 5 for unsorted), it only ran slow at 1 out of those 10 attempts (the final run for sorted arrays). This shows that on average, quick sort is still the fastest even when it has 'hiccups' sometimes.

A quick sort is also an O(nlogn) algorithm due to how its runtime only becomes 10 times slower when the runtime increases by 10 times. However, the difference between quick sort and merge sort is significant as it ran approximately 10 times faster than merge sort (and heap sort) at N = 1000000.

**HEAP SORT**

Heap sort behaved similarly to merge sort, but it was faster than merge sort on average. Like the other O(nlogn) algorithms, its runtime becomes 10 times slower when the array size increases by 10. It has no 'hiccups' unlike quick sort, but it is still considered slower on average compared to quick sort.

This is also evidenced by the graph, since heap sort is slower growing than merge sort but faster growing than quick sort.

Heap sort also behaves the same regardless of whether the array is sorted or not, showing that it is a consistent algorithm that is faster than merge sort, yet slower than quick sort.

**Conclusions**

Because of this, we conclude that the sorting algorithms from fastest to slowest are quick sort, heap sort, merge sort, insertion sort, selection sort, and bubble sort once the array size is over 1000. Below 1000, all sorting algorithms ran at below 1 seconds, showing that all algorithms at small sizes are alright.

# Challenges Encountered and Member Participation

**Challenges Encountered**

**Coding**

- On the coding aspect, most of the challenge came from implementing the main functions and implementing the recursive algorithms. The iterative algorithms were straightforward to implement, but the recursive algorithms urged us to really study the given pseudo code, or study the algorithm’s concept on the internet.

- Challenges encountered when implementing the main functions revolved around having incorrect assumptions regarding the instruction. We first thought that the max range is a user input, so rereading the instruction and realizing that the max range is the max range of an unsigned long integer, made the group change the typings throughout the source code. Another problem was not realizing that the source program was supposed to produce an output file, which the first iterations of our program did not do. These issues regarding the instructions may have appeared since we mostly handled the project during a week of no classes (Holy Week) where most of us had a rest from school.

- For Heap Sort, the implementer had to run the pseudo code on his paper to understand that the calculation of left and right child nodes will be slightly different due to starting from 0 instead of 1. Implementing the heap sort mostly had minor index-related problems

- Quick Sort is where some of our members had challenges, as one did not feel comfortable with deviating from the pseudocode given in the lecture. However, after attempts to implement quick sort with the given pseudo code, certain realizations occurred, and that using a quicksort concept from the internet (the Hoare’s partition scheme) followed the logic given in the lecture much more clearly and has less logic errors. Quick sort was much more difficult to “run-on-paper” however, which contributed to the challenge of implementing the algorithm. However, finding the median of three was unexpectedly quick.

**Runtime**

- In truth, there was one true challenge while running the program, and it was the extremely slow runtime of bubble sort and selection sort. There was one time where bubble sort and selection sort ran for an extremely long time (2500 for selection and 4900 for bubble). Theoretically, one can attend a 1-2 hour webinar, and it would still not be enough to have both bubble and selection sort finish. The time consuming bubble sort and selection sort made N = 1000000 really tedious to calculate, which was especially challenging when our source code is sometimes adjusted or changed for bug fixing in the main (not sorting) algorithms. Thankfully, the runtime of the sorting algorithms are completely unaffected by our bug fixing/adjustments on the main functions (like implementing file handling).

**Group Cooperation**

- Since we handled the project during Holy Week, it was difficult to have the members cooperate at the same time. Some were participating in holy week events and others were busy with backlogs from other subjects.

## Member Participation Notes

**Jan Wilhelm T. Sy**

- Initiated the creation of the source code
- Managed to run all sorting algorithms 5 times
- Helped in emergency adjustments (e.g. added file handling)
- Main Implementer of Merge Sort
- Helped in the documentation

**Chris John Borigas**

- Main Implementer of Quick Sort
- Attempted to run the algorithms for the runtime
- Explained Quick Sort to members (e.g. why do-while instead of while)

**Russel Pispis**

- Graphed the table results for a successful 5 runs of the algorithm
- Almost ran all algorithms 5 times, but had trouble with large N sizes (specifically with bubble and selection sort)
- Main Implementer of the three iterative algorithms (Selection, Insertion, Bubble)

**Jovic Francis B. Rayco**

- Helped in Emergency Adjustments (e.g. adjusted max range to being the max range of unsigned long int as instructed by the instruction at A.2a)
- Main Implementer of Heap Sort
- Added additional comments on all parts of the source code
- Managed to run all sorting algorithms 5 times
- Wrote majority of the documentation

**ADDITIONAL - Algorithm References**

- Since we used the Hoare's Partition Scheme as our main reference for the quick sort algorithm, here is the link of the concept, just for reference purposes: https://www.geeksforgeeks.org/hoares-vs-lomuto-partition-scheme-quicksort/

**ADDITIONAL - Table Outputs**

- This is the output table used for the analysis:

| UNSORTED ARRAYS | | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| MERGE SORT | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.030000 | 0.276000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.020000 | 0.270000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.020000 | 0.275000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.010000 | 0.020000 | 0.290000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.030000 | 0.286000 |
| | | | | | | | |
| QUICK SORT | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.010000 | 0.090000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.010000 | 0.090000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.093000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.090000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.010000 | 0.090000 |
| | | | | | | | |
| HEAP SORT | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.020000 | 0.200000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.020000 | 0.196000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.020000 | 0.200000 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.020000 | 0.208000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.017000 | 0.194000 |
| | | | | | | | |
| **INSERTION SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.070000 | 5.745000 | 589.333000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.080000 | 5.673000 | 579.476000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.070000 | 5.741000 | 585.335000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.056000 | 5.630000 | 577.430000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.070000 | 5.742000 | 568.921000 |
| | | | | | | | |
| **SELECTION SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.140000 | 10.580000 | 1057.167000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.140000 | 10.627000 | 1059.135000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.120000 | 10.887000 | 1058.130000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.156000 | 11.378000 | 1055.499000 |
| | | 0.000000 | 0.000000 | 0.020000 | 0.160000 | 10.986000 | 1060.165000 |
| | | | | | | | |
| **BUBBLE SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.368000 | 28.706000 | 2730.049000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.290000 | 27.515000 | 2759.874000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.262000 | 27.287000 | 2741.308000 |
| | | 0.000000 | 0.000000 | 0.010000 | 0.250000 | 27.600000 | 2767.067000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.226000 | 27.250000 | 2753.672000 |
| **SORTED ARRAYS** | | **10** | **100** | **1000** | **10000** | **100000** | **1000000** |
| | | | | | | | |
| **MERGE SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.030000 | 0.251000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.030000 | 0.245000 |
| | | 0.000000 | 0.000000 | 0.010000 | 0.000000 | 0.030000 | 0.230000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.010000 | 0.020000 | 0.240000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.030000 | 0.240000 |
| | | | | | | | |
| **QUICK SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.010000 | 0.050000 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.010000 | 0.060000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.050000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.054000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.010000 | 0.420000 |
| | | | | | | | |
| **HEAP SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.020000 | 0.200000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.010000 | 0.020000 | 0.180000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.020000 | 0.190000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.010000 | 0.020000 | 0.210000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.010000 | 0.020000 | 0.190000 |
| | | | | | | | |
| **INSERTION SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.010000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.010000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| | | | | | | | |
| **SELECTION SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.126000 | 10.854000 | 1086.569000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.120000 | 10.711000 | 1066.148000 |
| | | 0.000000 | 0.000000 | 0.010000 | 0.144000 | 10.710000 | 1063.189000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.140000 | 10.700000 | 1059.543000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.126000 | 10.549000 | 1061.294000 |
| | | | | | | | |
| **BUBBLE SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.002000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.003000 |
| | | 0.000000 | 0.000000 | 0.010000 | 0.000000 | 0.000000 | 0.002000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.002000 |

**ADDITIONAL - Extra Tables**

- Another member successfully ran the algorithms, resulting in the following output table. However, upon looking at the results of the table, the behavior of the algorithms as the input size increased is mostly the same, so the difference in runtime here is most likely caused by the different computer specifications:

| UNSORTED ARRAYS | | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| MERGE SORT | | 0.000000 | 0.000000 | 0.000000 | 0.001000 | 0.026000 | 0.290000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.004000 | 0.028000 | 0.284000 |
| | | 0.000000 | 0.000000 | 0.001000 | 0.003000 | 0.025000 | 0.281000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.002000 | 0.024000 | 0.284000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.026000 | 0.277000 |
| | | | | | | | |
| QUICK SORT | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.010000 | 0.114000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.002000 | 0.010000 | 0.112000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.001000 | 0.012000 | 0.114000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.002000 | 0.014000 | 0.111000 |
| | | 0.000000 | 0.000000 | 0.002000 | 0.005000 | 0.011000 | 0.111000 |
| | | | | | | | |
| HEAP SORT | | 0.000000 | 0.000000 | 0.000000 | 0.002000 | 0.021000 | 0.258000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.002000 | 0.019000 | 0.257000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.003000 | 0.018000 | 0.253000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.023000 | 0.243000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.020000 | 0.255000 |
| | | | | | | | |
| INSERTION SORT | | 0.000000 | 0.000000 | 0.000000 | 0.050000 | 4.724000 | 495.208000 |
| | | 0.000000 | 0.000000 | 0.001000 | 0.050000 | 4.769000 | 487.830000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.049000 | 4.728000 | 489.543000 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0.000000 | 0.000000 | 0.000000 | 0.051000 | 4.764000 | 500.399000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.050000 | 4.703000 | 493.941000 |
| | | | | | | | |
| **SELECTION SORT** | | 0.000000 | 0.000000 | 0.001000 | 0.067000 | 5.732000 | 725.381000 |
| | | 0.000000 | 0.000000 | 0.002000 | 0.056000 | 5.605000 | 722.325000 |
| | | 0.000000 | 0.000000 | 0.001000 | 0.060000 | 5.769000 | 734.364000 |
| | | 0.000000 | 0.000000 | 0.002000 | 0.055000 | 5.778000 | 754.391000 |
| | | 0.000000 | 0.000000 | 0.001000 | 0.061000 | 5.749000 | 794.549000 |
| | | | | | | | |
| **BUBBLE SORT** | | 0.000000 | 0.000000 | 0.003000 | 0.191000 | 26.193000 | 2601.799000 |
| | | 0.000000 | 0.000000 | 0.002000 | 0.184000 | 25.459000 | 2607.063000 |
| | | 0.000000 | 0.000000 | 0.001000 | 0.186000 | 25.663000 | 2601.001000 |
| | | 0.000000 | 0.000000 | 0.001000 | 0.182000 | 25.733000 | 2621.842000 |
| | | 0.000000 | 0.000000 | 0.003000 | 0.185000 | 25.888000 | 2593.563000 |
| **SORTED ARRAYS** | | **10** | **100** | **1000** | **10000** | **100000** | **1000000** |
| | | | | | | | |
| **MERGE SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.002000 | 0.019000 | 0.229000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.021000 | 0.206000 |
| | | 0.000000 | 0.000000 | 0.001000 | 0.000000 | 0.019000 | 0.202000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.018000 | 0.214000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.017000 | 0.202000 |
| | | | | | | | |
| **QUICK SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.002000 | 0.003000 | 0.038000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.004000 | 0.038000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.004000 | 0.039000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.004000 | 0.007000 | 0.039000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.003000 | 0.036000 |
| | | | | | | | |
| **HEAP SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.001000 | 0.016000 | 0.173000 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.015000 | 0.178000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.002000 | 0.015000 | 0.187000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.001000 | 0.015000 | 0.184000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.003000 | 0.016000 | 0.173000 |
| | | | | | | | |
| **INSERTION SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.003000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.003000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.002000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.003000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.002000 |
| | | | | | | | |
| **SELECTION SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.057000 | 5.823000 | 713.910000 |
| | | 0.000000 | 0.000000 | 0.001000 | 0.057000 | 5.780000 | 722.835000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.054000 | 5.614000 | 725.022000 |
| | | 0.000000 | 0.000000 | 0.001000 | 0.060000 | 5.659000 | 703.917000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.058000 | 5.803000 | 735.874000 |
| | | | | | | | |
| **BUBBLE SORT** | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.002000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.002000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.001000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.003000 |
| | | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.003000 | 0.002000 |