Implementing Linear, Merge, and Heap Sorting Algorithms

Homework #1

By Logan Miles

1. Objectives

The goal of this assignment was to implement the following sorting algorithms: linear sort, merge sort, a modified merge sort, and heap sort to evaluate their respective performance and compare their time complexities utilizing a high-level programming language.

2. Program Design

This assignment required several functions to achieve the desired outputs and functionality. These consist of 4 functions containing the required sorting algorithms, and several functions to support them. These are:

1) main() – The function containing the driver code
2) insertionSort() – The function containing the insertion sort algorithm
3) mergeSort() – The function containing the merge sort algorithm
4) modifiedMergeSort() – The function containing the modified merge sort algorithm
5) merge() – A function called in mergeSort() and modifiedMergeSort() to merge subarrays
6) heapSort() – The function containing the heap sort algorithm
7) maxHeapify() – A function called in heapSort() that maintains the max heap properties
8) buildMaxHeap() – A function called in heapSort() that builds the max heap from the original array

main()

The driver code found in main() is responsible to reading the text files, initializing the array, running the sorting algorithms, recording the execution time, and printing the results. Several different text files containing integers separated by commas were given as the elements that were meant to be sorted inside of an array within the program. This meant that the text files must be found using the Paths class and then read using the Scanner class. The strings are then split at each comma and space using a regex and added to an array of strings. The strings are then converted to integers using parseInt() and added to an array and a copy of that array for iteration purposes. The function then calls each function for each sorting algorithm. Before and after each call the system time is recorded in nanoseconds using nanoTime() to give the local variables timeInit and timeFinal respectively. timeInit is then subtracted from timeFinal to yield time, which represents the execution time and is printed after each algorithm function call. Time is represented in nanoseconds, milliseconds, and seconds using simple conversion in the print statement.

insertionSort()

The insertion sort function utilizes a for loop to iterate forward one-by-one through each element in the array. The algorithm then uses a while loop to move the current element of iteration back one index if the value of the current index is less than the value of the previous index and the previous index is greater than or equal to zero.

mergeSort()

The merge sort algorithm calculates the middle index of the array, defined as int q, then calls recursively calls itself on the left and right subarrays by using four different pointers as parameters for where the subarrays begin and in:

1) int p – The pointer for the beginning of the left subarray
2) int q – The pointer for the end of the left subarray
3) int q + 1 – The pointer for the beginning of the right subarray
4) int r – The pointer for the end of the right subarray

The function then recursively calls itself on these subarrays to further split them into smaller subarrays, so long as the if condition is met, which states that the pointer for the left subarray is less than the pointer for the right subarray. This prevents the arrays from being divided to a size smaller than one. After the subarrays have been divided down to the smallest size possible, the merge() function is called on each subarray recursively until the full array has been sorted.

merge()

The merge function uses a for loop to create a copy of each subarray; tempArray(). The function then iterates over that array using a for loop starting at the starting index of the left subarray and ending at the ending index of the right subarray. A series of if statements check what order to add the elements of the subarray back into the main array using the following conditions:

1) If the starting index of the left subarray is greater than the middle index of the main array, then the left subarray is empty and the element from the right subarray is copied to the main array.

2) If the starting index of the right subarray is greater than the ending index, then the right subarray is empty and the element from the left subarray is copied to the main array.
3) If the value from the right subarray at the index is less than the value in the left subarray, then the element from the right subarray is copied to the main array.
4) Else the value from the right subarray is copied to the main array.

The proper pointer is also iterated after each if statement so that the correct two elements are compared. This series of if statements sort the integer elements from the array into the proper sequence and merges them back into the main array.

modifiedMergeSort()

This algorithm essentially functions the same as mergeSort(), except for one key difference. There is an if statement that checks the size of the array after each recursion. The array size is defined by r – p were r is the ending index of the right subarray and p is the starting index of the left subarray. If the array size is less than or equal to the insertionSortThreshold defined as a static global variable, then the function calls insertionSort() on the subarray during that recursion before merge() is called, which merges the subarray with the main array.

buildMaxHeap()

This function is called in heapSort(). It is responsible for building the initial max heap for the heap sort algorithm. The function accomplishes this by iterating down the array and calling the maxHeapify() function on every e, defined by int i = heapSize / 2 – 1 in the for loop iteration.

maxHeapify()

This function maintains the max heap property during each recurion of heapSort(). It does so using a series of if statements:

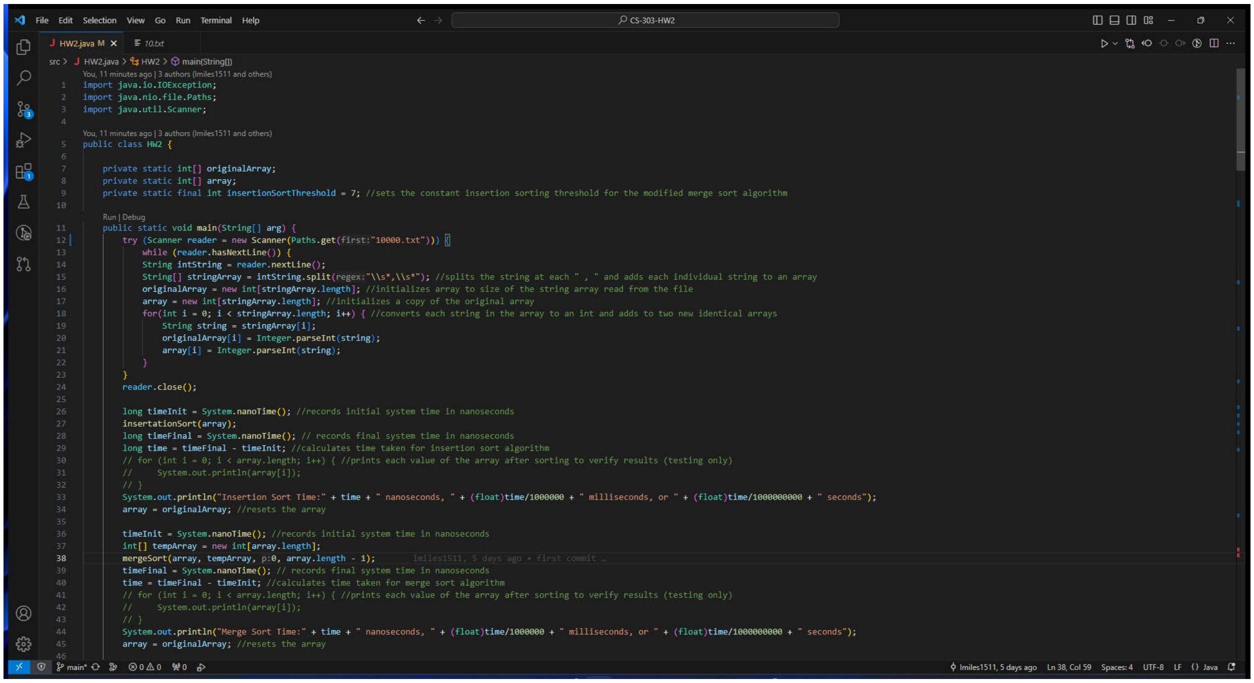
1) If the index of the left leaf is smaller than the heap size and the value of the left leaf is greater than the parent value, set the left leaf to parent.

2) If the index of the right leaf is smaller than the heap size and the value of the left leaf is greater than the parent value, set the right leaf to parent.

3) If the largest element is not the root of the maxheap, swap them and recursively heapify the affected sub-tree.

heapSort()

This function first calls the buildMaxHeap() to initialize the max heap. It then iterates down the array using a for loop and calls the maxHeapify() function, which sorts the array.

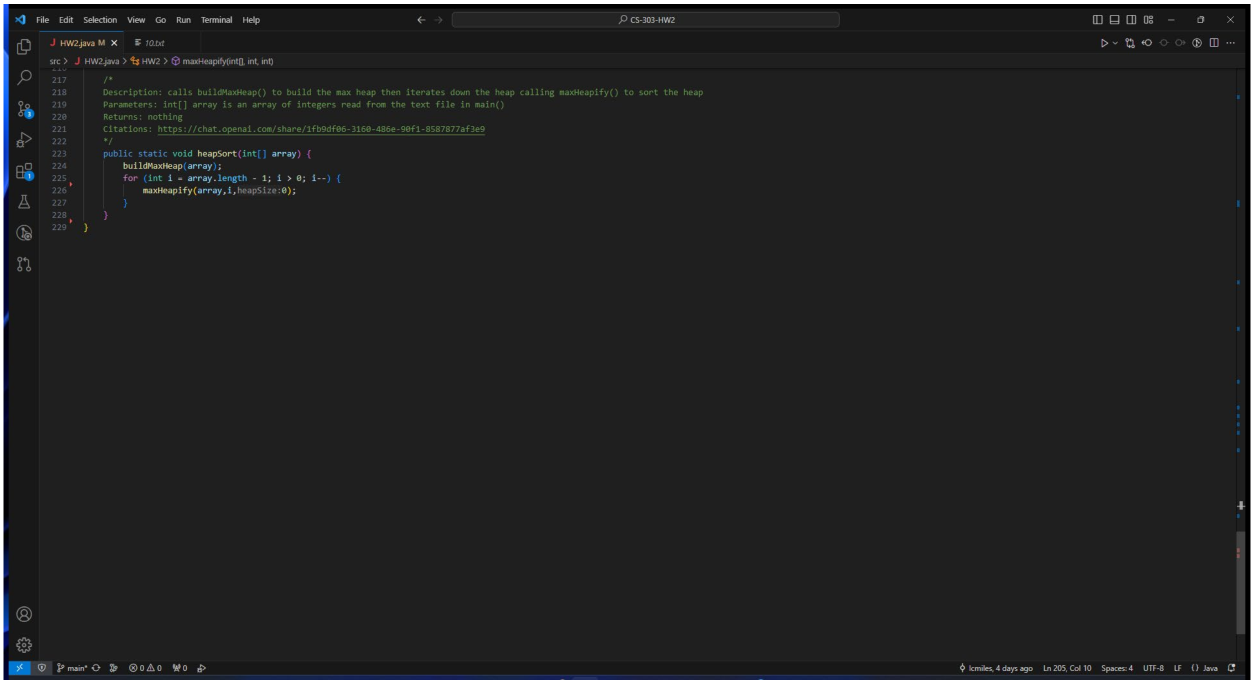


```java
            // }
            System.out.println("Merge Sort Time:" + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");
            array = originalArray; //resets the array

            timeInit = System.nanoTime(); //records initial system time in nanoseconds
            modifiedMergeSort(array, tempArray, p:0, array.length - 1);
            timeFinal = System.nanoTime(); // records final system time in nanoseconds
            time = timeFinal - timeInit; //calculates time taken for modified merge sort algorithm
            // for (int i = 0; i < array.length; i++) { //prints each value of the array after sorting to verify results (testing only)
            //     System.out.println(array[i]);
            // }
            System.out.println("Modified Merge Sort Time:" + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");
            array = originalArray; //resets the array

            timeInit = System.nanoTime(); //records initial system time in nanoseconds
            heapSort(array);
            timeFinal = System.nanoTime(); // records final system time in nanoseconds
            time = timeFinal - timeInit; //calculates time taken for modified heap sort algorithm
            for (int i = 0; i < array.length; i++) { //prints each value of the array after sorting to verify results (testing only)
                System.out.println(array[i]);
            }
            System.out.println("Heap Sort Time:" + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");
            } catch (IOException e) {
                e.printStackTrace();
            }     lmiles1511, 5 days ago • first commit
        }

        /*
        Description: insertionSort() iterates through each element in the array and compares the value of the current index to
        that of the previous index moving the value back until the value of the previous index is less than the current
        Parameters: int[] array is an array of integers read from the text file in main()
        Returns: nothing
        Citations:
        https://chat.openai.com/share/e0124443-6628-4959-9653-b91b956b9b12
        https://www.geeksforgeeks.org/insertion-sort/
        */
        public static void insertationSort(int[] array) {
            int i, j, k; //initializes count variables
            for (i = 0; i < array.length; i++) {
                k = array[i]; //set the key to the value of the current index
                j = i - 1; //j is the previous index
                while (j >= 0 && array[j] > k) { //while lowest index is above 0 and the value of the previous index is above the value of the current index
                    array[j + 1] = array[j]; //shifts value at index one position to the right, making space for key
                    j = j - 1; //the j index is moved one left for further comparisons
                }
                array[j + 1] = k; //places the k value at the correct sorted position
            }
        }
```

```java

        /*
        Description: mergeSort() calculates the middle index, if the recursively calls mergeSort() on the left and right subarrays in order to sort them,
        then calls merge() to merge the newly sorted subarrays
        Parameters:
        int[] array is an array of integers read from the text file in main()
        int[] temparray is a temporary copy of array for iteration purposes
        int p is a pointer for starting index of left subarray
        int r is a pointer for ending index of right subarray
        Returns: nothing
        Citations:
        https://chat.openai.com/share/c8f53e73-fb52-483b-846c-c756faa3bc6e
        https://www.geeksforgeeks.org/merge-sort/
        */
        public static void mergeSort(int[] array, int[] tempArray, int p, int r) {
            if (p < r) {
                int q = (p + r) / 2; //calculate middle index
                mergeSort(array, tempArray, p, q); //recursively sort left subarray
                mergeSort(array, tempArray, q + 1, r); //recursively sort right subarray
                merge(array, tempArray, p, q, r); //merge the two sorted subarrays
            }
        }

        /*
        Description: modifiedMergeSort() works similarly to mergeSort(), only the algorithm implements insertionSort() when the array size has been reduced to less
        than or equal to insertionSortThreshold
        Parameters:
        int[] array is an array of integers read from the text file in main()
        int[] temparray is a temporary copy of array for iteration purposes
        int p is a pointer for starting index of left subarray
        int r is a pointer for ending index of right subarray
        Returns: nothing
        Citations: https://chat.openai.com/share/c8f53e73-fb52-483b-846c-c756faa3bc6e
        */
        public static void modifiedMergeSort(int[] array, int[] tempArray, int p, int r) {
            if (p < r) {
                if (r - p <= insertionSortThreshold) { //runs insertion sort algorithm if array size is less than or equal to the threshold
                    insertationSort(array);
                }
                else { //else runs mergesort algorithm
                int q = (p + r) / 2; //calculate middle index
                mergeSort(array, tempArray, p, q); //recursively sort left subarray
                mergeSort(array, tempArray, q + 1, r); //recursively sort right subarray
                merge(array, tempArray, p, q, r); //merge the two sorted subarrays
                }
            }
        }
```

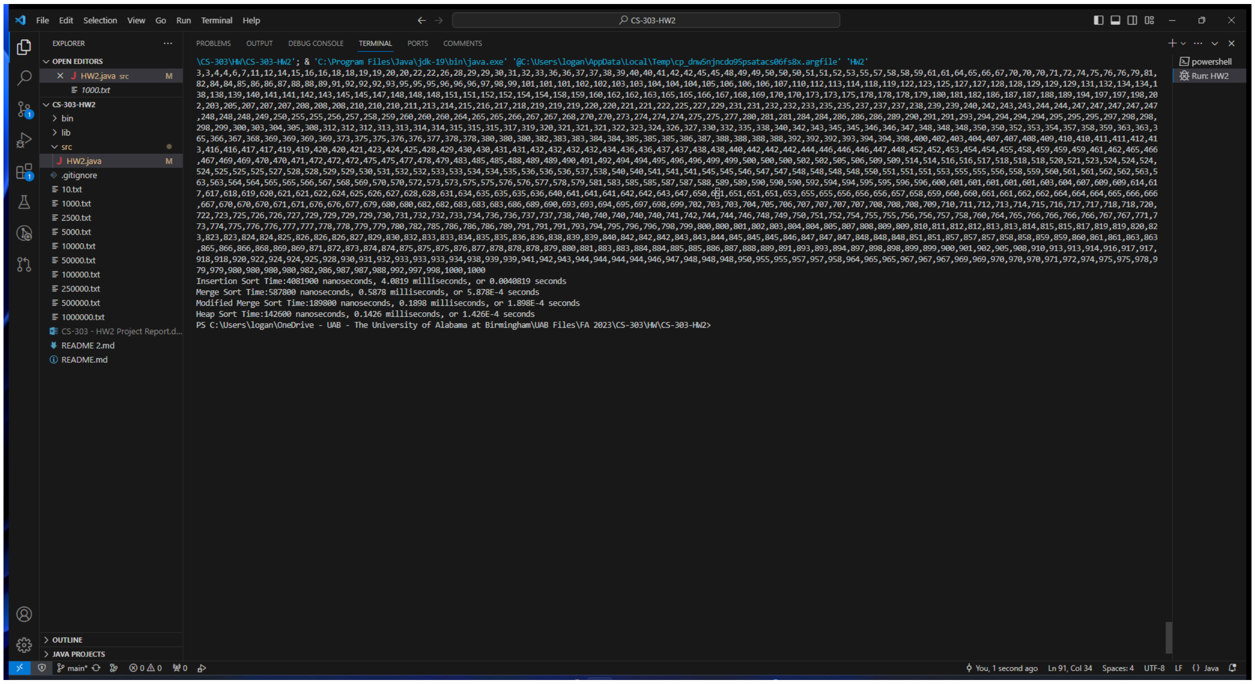
```java
    /*
    Description: merge() is called in mergeSort(), and it is responsible for adding the values in the correct order from the subarrays back into the array they
    were divided from until the whole array is sorted
    Parameters:
    int[] array is an array of integers read from the text file in main()
    int[] temparray is a temporary copy of array for iteration purposes
    int p is a pointer for starting index of left subarray
    int r is a pointer for ending index of right subarray
    int q is a pointer for middle index of the original array
    Returns: nothing
    Citations: https://chat.openai.com/share/c8f53e73-fb52-483b-846c-c756faa3bc6e
    */
    public static void merge(int[] array, int[] tempArray, int p, int q, int r) {
        int i = p; //initializing pointer to starting index of left subarray
        int j = q + 1; //initializing pointer to starting index of right subarray
        for (int k = p; k <= r; k++) { //copy all values in array to tempArray
            tempArray[k] = array[k];
        }
        for (int k = p; k <= r; k++) {
            if (i > q) { //if left subarray is empty, copy the element from the right subarray to the main array
                array[k] = tempArray[j];
                j++;
            }
            else if (j > r) { //if right subarray is empty, copy the element the left subarray to the main array
                array[k] = tempArray[i];
                i++;
            }
            else if (tempArray[j] < tempArray[i]) { //if the right element is smaller than the left element, copy the element from the right subarray to the main array
                array[k] = tempArray[j];
                j++;
            }
            else { //else copy the element from the left subarray to the main array
                array[k] = tempArray[i];
                i++;
            }
        }
    }

    /*
    Description: buildMaxHeap() builds the max heap by iterating down the array and calling maxHeapify() on non-leaf node to ensure that the subtree rooted at i is a valid max heap
    Parameters: int[] array is an array of integers read from the text file in main()
    Returns: nothing
    Citations: https://chat.openai.com/share/1fb9df06-3160-486e-90f1-8587877af3e9
    */
    public static void buildMaxHeap(int[] array) {
        int heapSize = array.length;
        for (int i = heapSize / 2 - 1; i >= 0; i--) { //initializes i to the last non-leaf node
            maxHeapify(array, heapSize, i);
```

```java
    /*
    Description: buildMaxHeap() builds the max heap by iterating down the array and calling maxHeapify() on non-leaf node to ensure that the subtree rooted at i is a valid max heap
    Parameters: int[] array is an array of integers read from the text file in main()
    Returns: nothing
    Citations: https://chat.openai.com/share/1fb9df06-3160-486e-90f1-8587877af3e9
    */
    public static void buildMaxHeap(int[] array) {
        int heapSize = array.length;
        for (int i = heapSize / 2 - 1; i >= 0; i--) { //initializes i to the last non-leaf node
            maxHeapify(array, heapSize, i);
        }
    }

    /*
    Description: maxHeapify() maintains the max heap properties during each recursion of heapSort()
    Parameters:
    int[] array is an array of integers read from the text file in main()
    int i is the parent iterated over in buildMaxHeap()
    int heapSize is the number of elements in the heap
    Returns: nothing
    Citations: https://chat.openai.com/share/1fb9df06-3160-486e-90f1-8587877af3e9
    */
    public static void maxHeapify(int[] array, int i, int heapSize) {
        int largest = i; //index of the parent
        int left = 2 * i + 1; //index of the left leaf
        int right = 2 * i + 2; //index of the right leaf
        if (left < heapSize && array[left] > array[largest]) { //if index of the left leaf is less than the heap size and the value of the left leaf is greater than the parent value
            largest = left; //set left leaf to parent
        }     lcmiles, 4 days ago • Committing changes
        if (right < heapSize && array[right] > array[largest]) { //if index of the right leaf is less than the heap size and the value of the right leaf is greater than the parent value
            largest = right; //set right leaf to parent
        }
        if (largest != i) { //if the largest element is not the root of the maxheap, swap them and recursively heapify the affected sub-tree
            int temp = array[i];
            array[i] = array[largest];
            array[largest] = temp;
            maxHeapify(array,heapSize,largest);
        }
    }

    /*
    Description: calls buildMaxHeap() to build the max heap then iterates down the heap calling maxHeapify() to sort the heap
    Parameters: int[] array is an array of integers read from the text file in main()
    Returns: nothing
    Citations: https://chat.openai.com/share/1fb9df06-3160-486e-90f1-8587877af3e9
    */
    public static void heapSort(int[] array) {
        buildMaxHeap(array);
```

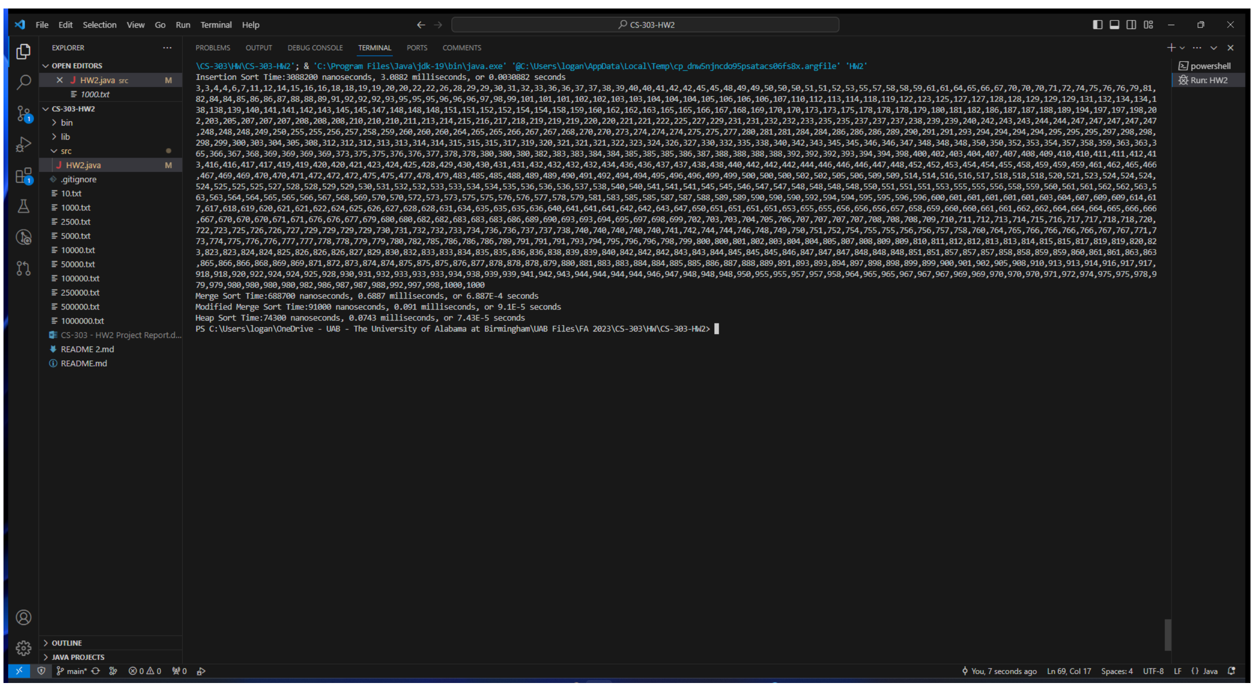


3. Testing

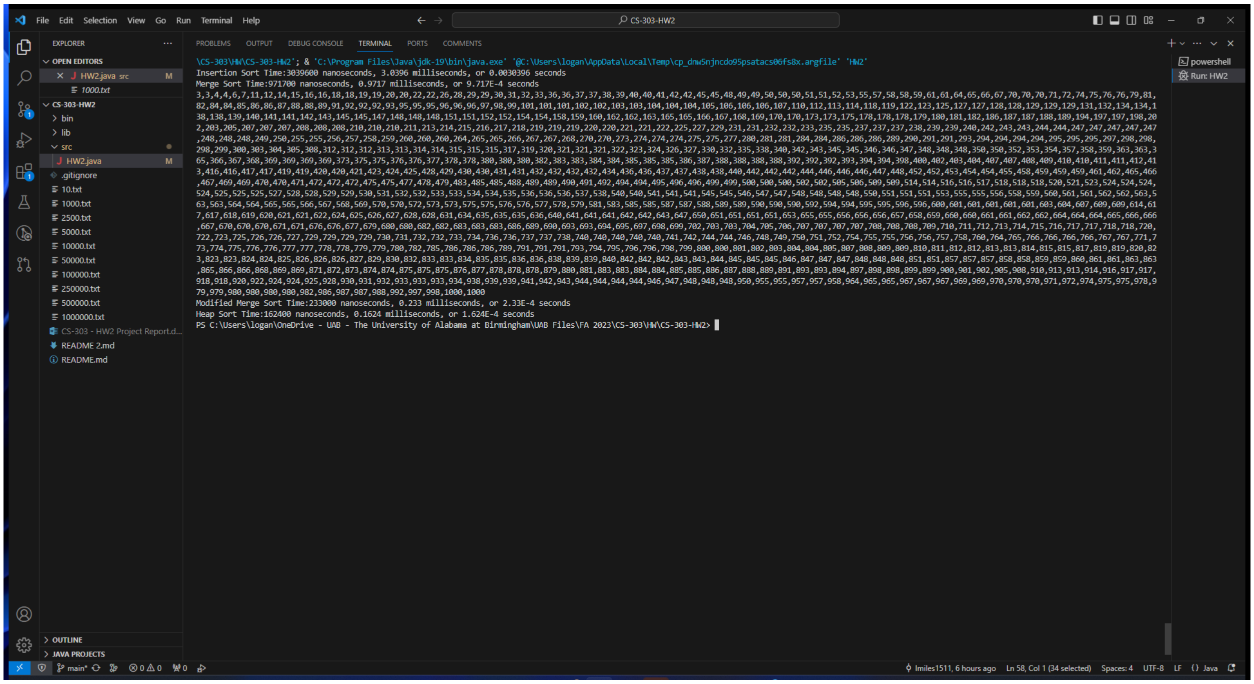
Testing of the algorithms and supporting functions was done using the several text files containing integers separated by commas ranging from 1000 integers to 1000000 integers. The output of each algorithm was verified by printing the sorted arrays. The time was recorded for each algorithm by printing the time in main() mentioned in the Program Design section. Each algorithm was tested using the provided set of integers in the text files. The modified merge sort algorithm threshold was defined as 7 for all comparisons between it and other sorting algorithms.



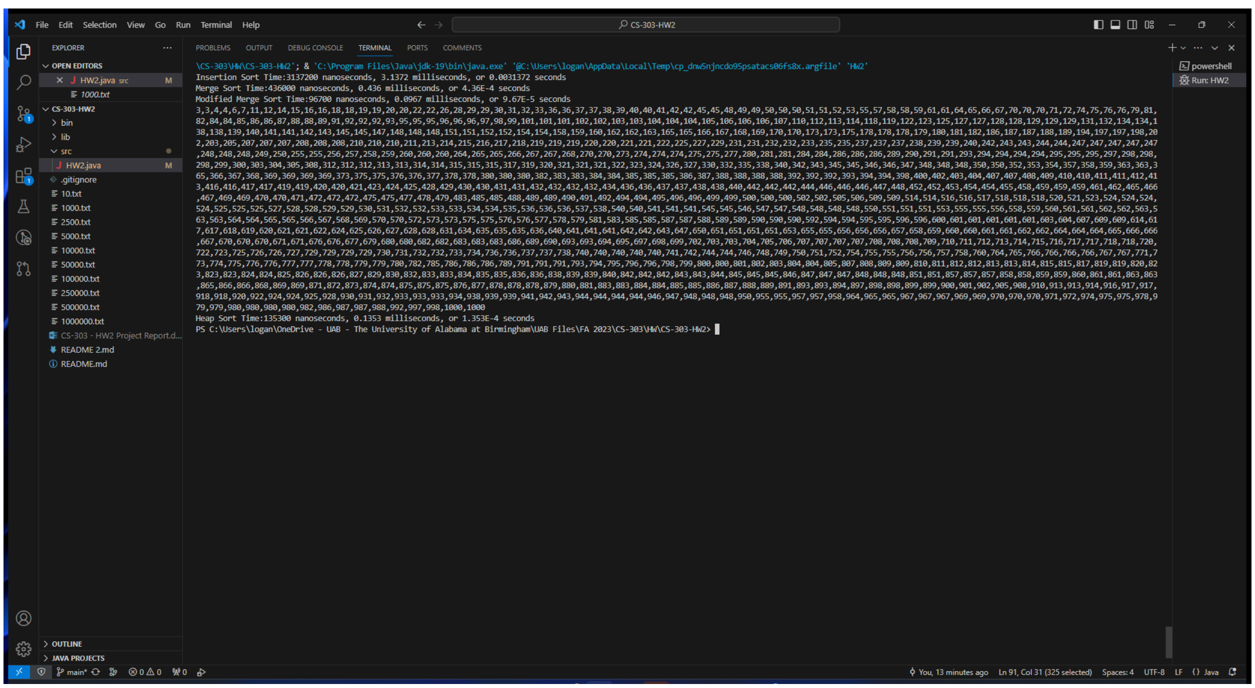
Example output of linear search algorithm.



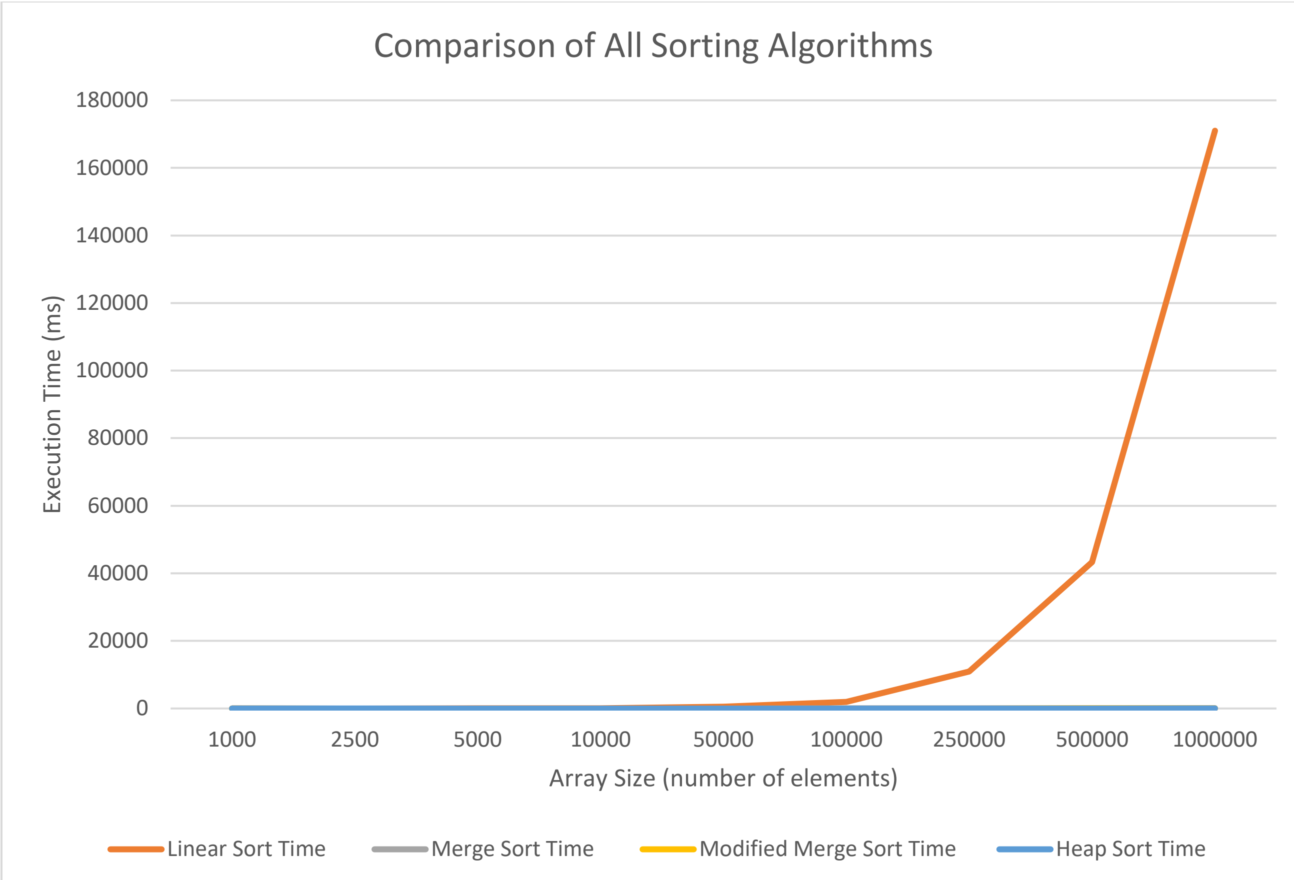
Example output of merge sort algorithm.



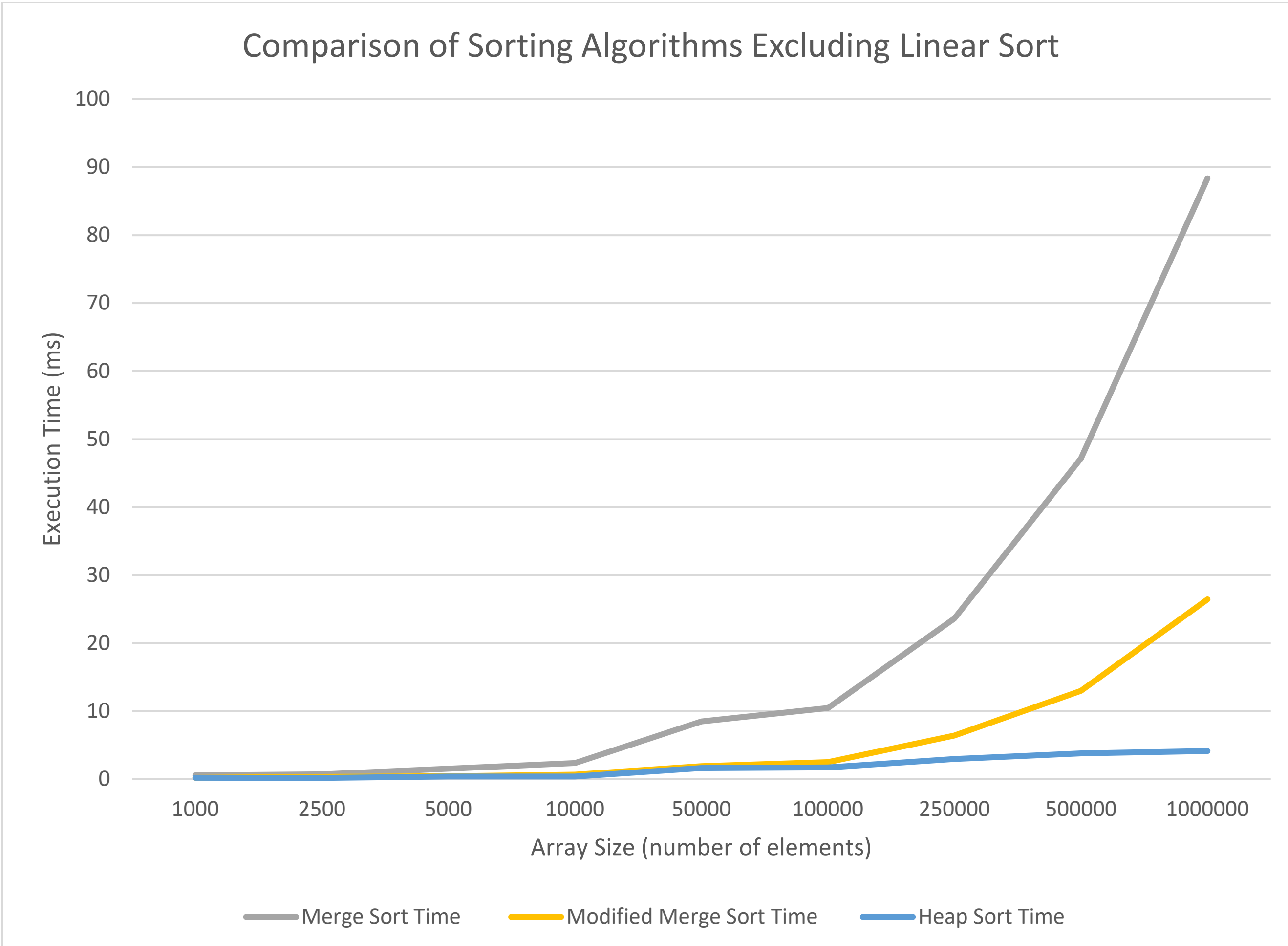
Example output of modified merge sort algorithm.



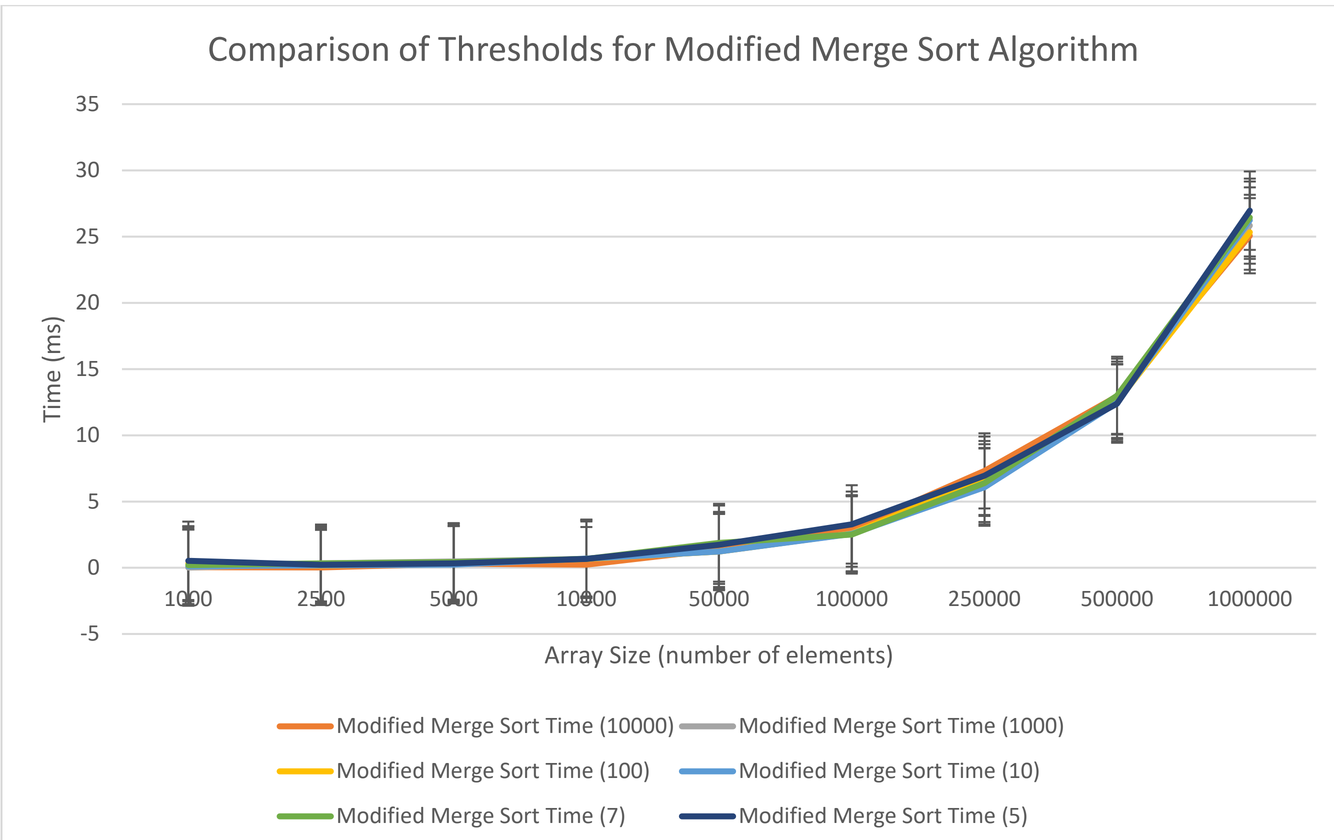
Example output of heap sort algorithm.



A comparison of all the search algorithms reveals the linear time complexity of O(n) of linear sort. The graph shows that the execution time is manageable when dealing with smaller arrays, but when the number of elements in the array increases, so too does the execution time of the linear search algorithm. This is most apparent with the datapoint at n = 1000000. The time complexity of the other algorithms is O(nlog(n)). In conclusion, linear search is an inefficient algorithm to use when dealing with large arrays as the time complexity varies linearly with the number of elements.



After scaling the graph down by removing the linear search algorithm, one can now see that there are 3 very distinct graphs for the other search algorithms. It seems that the merge sort algorithm, while much more efficient than the linear search algorithm, has the longest execution time in most cases here. While it is comparable to the modified merge sort and the heap sort algorithms at lower dataset sizes, it becomes more apparent the higher the number of elements becomes. This difference is barely noticeable in real time, as it is only a few milliseconds, however, when dealing with much larger arrays this could become more apparent. The time complexity of merge sort is O(nlog(n)). The modified merge sort algorithm seems to have a better time complexity, however, the time complexity is the same regardless of the implementation of insertion sort. In any case the reason that the execution time is lower when dealing with larger arrays is due to this implementation, as it sorts the smaller arrays faster than merge sort. Heap sort also has a time complexity of O(nlog(n)) but appears to be slightly faster than both merge sort algorithms. This could be due to several external factors, such as cache efficiency. All the differences shown above are within milliseconds of one another and are realistically negligible.



Testing of different thresholds for the modified merge sort algorithm revealed very little difference in most cases. All results were within the margin of error. This could be because the linear sort algorithm is most effective when dealing with small arrays and these thresholds all include smaller arrays.

4. References

https://jenkov.com/tutorials/java-date-time/system-currenttimemillis.html
https://chat.openai.com/share/e0124443-6628-4959-9653-b91b956b9b12
https://chat.openai.com/share/c8f53e73-fb52-483b-846c-c756faa3bc6e
https://chat.openai.com/share/1fb9df06-3160-486e-90f1-8587877af3e9
https://www.geeksforgeeks.org/insertion-sort/
https://www.geeksforgeeks.org/merge-sort/