

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 $r - p$ where 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 $\text{int } i = \text{heapSize} / 2 - 1$ in the for loop iteration.

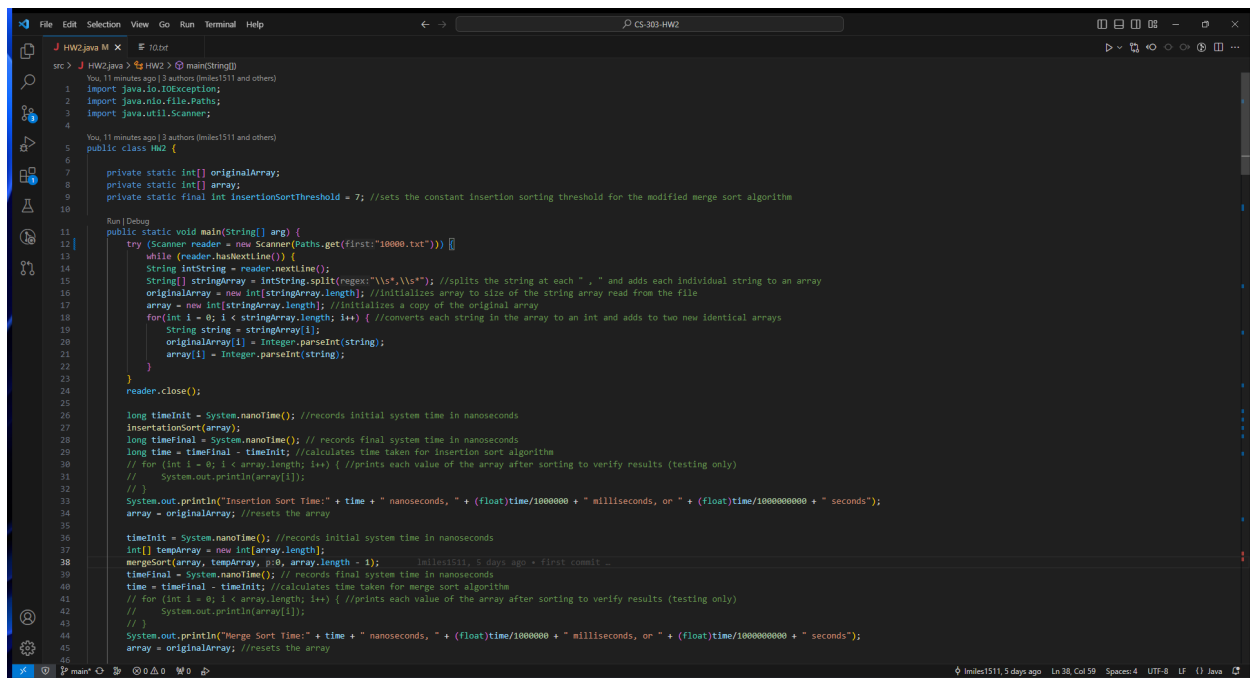
maxHeapify()

This function maintains the max heap property during each recursion 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.



```
src > J HW2.java M X F 10.txt
You, 11 minutes ago [3 authors (lmlc1511 and others)]
1 import java.io.IOException;
2 import java.nio.file.Paths;
3 import java.util.Scanner;
4
5 You, 11 minutes ago [3 authors (lmlc1511 and others)]
6 public class HW2 {
7
8     private static int[] originalArray;
9     private static final int insertionSortThreshold = 7; //sets the constant insertion sorting threshold for the modified merge sort algorithm
10
11     Run [Debug]
12     public static void main(String[] arg) {
13         try (Scanner reader = new Scanner(Paths.get("10000.txt"))) {
14             while (reader.hasNextLine()) {
15                 String intString = reader.nextLine();
16                 String[] stringArray = intString.split(regex: "\\s+"); //splits the string at each " " and adds each individual string to an array
17                 originalArray = new int[stringArray.length]; //initializes array to size of the string array read from the file
18                 array = new int[stringArray.length]; //initializes a copy of the original array
19                 for(int i = 0; i < stringArray.length; i++) { //converts each string in the array to an int and adds to two new identical arrays
20                     String string = stringArray[i];
21                     originalArray[i] = Integer.parseInt(string);
22                     array[i] = Integer.parseInt(string);
23                 }
24             }
25             reader.close();
26
27             long timeInit = System.nanoTime(); //records initial system time in nanoseconds
28             insertionSort(array);
29             long timeFinal = System.nanoTime(); // records final system time in nanoseconds
30             long time = timeFinal - timeInit; //calculates time taken for insertion sort algorithm
31             // for (int i = 0; i < array.length; i++) { //prints each value of the array after sorting to verify results (testing only)
32             //     System.out.println(array[i]);
33             // }
34             System.out.println("Insertion Sort Time: " + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");
35             array = originalArray; //resets the array
36
37             timeInit = System.nanoTime(); //records initial system time in nanoseconds
38             int[] tempArray = new int[array.length];
39             mergeSort(array, tempArray, 0, array.length - 1); //lmlc1511, 5 days ago • first commit ...
40             timeFinal = System.nanoTime(); // records final system time in nanoseconds
41             time = timeFinal - timeInit; //calculates time taken for merge sort algorithm
42             // for (int i = 0; i < array.length; i++) { //prints each value of the array after sorting to verify results (testing only)
43             //     System.out.println(array[i]);
44             // }
45             System.out.println("Merge Sort Time: " + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");
46             array = originalArray; //resets the array
47         }
48     }
49 }
```

The screenshot shows a Java IDE with two files open: HW2.java and HW2.java. The HW2.java file contains a Merge Sort implementation. The code is as follows:

```
41 //
42
43 System.out.println("Merge Sort Time: " + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");
44 array = originalArray; //resets the array
45
46
47 timeInit = System.nanoTime(); //records initial system time in nanoseconds
48 modifiedMergeSort(array, tempArray, 0, array.length - 1);
49 timeFinal = System.nanoTime(); // records final system time in nanoseconds
50 time = timeFinal - timeInit; //calculates time taken for modified merge sort algorithm
51 // for (int i = 0; i < array.length; i++) { //prints each value of the array after sorting to verify results (testing only)
52 //     System.out.println(array[i]);
53 // }
54 System.out.println("Modified Merge Sort Time: " + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");
55 array = originalArray; //resets the array
56
57 timeInit = System.nanoTime(); //records initial system time in nanoseconds
58 heapSort(array);
59 timeFinal = System.nanoTime(); // records final system time in nanoseconds
60 time = timeFinal - timeInit; //calculates time taken for modified heap sort algorithm
61 for (int i = 0; i < array.length; i++) { //prints each value of the array after sorting to verify results (testing only)
62     System.out.println(array[i]);
63 }
64 System.out.println("Heap Sort Time: " + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");
65 } catch (IOException e) {
66     e.printStackTrace();
67 }
68 //
69
70
71 Description: InsertionSort() iterates through each element in the array and compares the value of the current index to
72 that of the previous index moving the value back until the value of the previous index is less than the current
73 Parameters: int[] array is an array of integers read from the text file in main()
74 Returns: nothing
75
76 Citation:
77 https://chat.openai.com/share/e8124443-6828-4959-9853-b91958b9b172
78 https://www.geeksforgeeks.org/insertion-sort/
79
80 public static void insertionSort(int[] array) {
81     int i, j, k; //initializes count variables
82     for (i = 0; i < array.length; i++) {
83         k = array[i]; //set the key to the value of the current index
84         j = i - 1; //j is the previous index
85         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
86             array[j + 1] = array[j]; //shifts value at index one position to the right, making space for key
87             j = j - 1; //the j index is moved one left for further comparisons
88         }
89         array[j + 1] = k; //places the k value at the correct sorted position
90     }
91 }
```

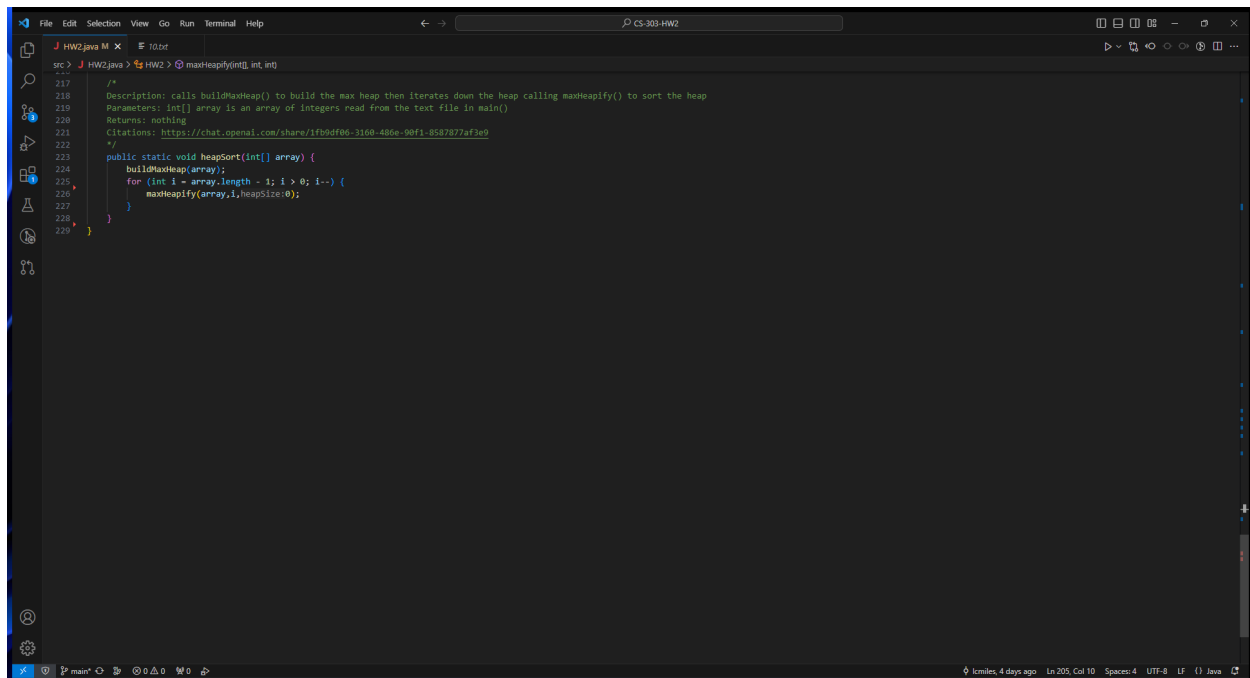
```
File Edit Selection View Go Run Terminal Help
J HW2.java M X E |0:0r
src> J HW2.java HW2> @ main(String[] args) {
91
92
93 Description: mergeSort() calculates the middle index, if the recursively calls mergeSort() on the left and right subarrays in order to sort them,
94 then calls merge() to merge the newly sorted subarrays.
95 Parameters:
96 Int[] array is an array of integers read from the text file in main()
97 Int[] tempArray is a temporary copy of array for iteration purposes
98 Int p is a pointer for starting index of left subarray
99 Int r is a pointer for ending index of right subarray
100 Returns: nothing
101 Citations:
102 https://chat.openai.com/share/c8f53e73-fb52-483b-846c-c756faa3bc6e
103 https://www.geeksforgeeks.org/merge-sort/
104
105 public static void mergeSort(Int[] array, Int[] tempArray, int p, int r) {
106     if (p < r) {
107         int q = (p + r) / 2; //calculate middle index
108         mergeSort(array, tempArray, p, q); //recursively sort left subarray
109         mergeSort(array, tempArray, q + 1, r); //recursively sort right subarray
110         merge(array, tempArray, p, q, r); //merge the two sorted subarrays
111     }
112 }
113
114
115 Description: modifyMergeSort() works similarly to mergeSort(), only the algorithm implements InsertionSort() when the array size has been reduced to less
116 than or equal to InsertionSortThreshold
117 Parameters:
118 Int[] array is an array of integers read from the text file in main()
119 Int[] tempArray is a temporary copy of array for iteration purposes
120 Int p is a pointer for starting index of left subarray
121 Int r is a pointer for ending index of right subarray
122 Returns: nothing
123 Citations: https://chat.openai.com/share/c8f53e73-fb52-483b-846c-c756faa3bc6e
124
125 public static void modifiedMergeSort(Int[] array, Int[] tempArray, int p, int r) {
126     if (p < r) {
127         if (r - p <= InsertionSortThreshold) { //runs insertion sort algorithm if array size is less than or equal to the threshold
128             InsertionSort(array);
129         }
130         else { //else runs mergeSort algorithm
131             int q = (p + r) / 2; //calculate middle index
132             mergeSort(array, tempArray, p, q); //recursively sort left subarray
133             mergeSort(array, tempArray, q + 1, r); //recursively sort right subarray
134             merge(array, tempArray, p, q, r); //merge the two sorted subarrays
135         }
136     }
137 }
138
```

```
File Edit Selection View Go Run Terminal Help
CS-303-HW2

J HW2.java M X F 10:01
src > J HW2.java > HW2 > @ main(String[])
139
140 /*
141 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
142 were divided from until the whole array is sorted
143 Parameters:
144 int[] array is an array of integers read from the text file in main()
145 int[] tempArray is a temporary copy of array for iteration purposes
146 int p is a pointer for starting index of left subarray
147 int r is a pointer for ending index of right subarray
148 int q is a pointer for middle index of the original array
149 Returns: nothing
150 Citations: https://chat.openai.com/share/c8f53e73-fb52-483b-846c-c756faa3bcde
151 */
152 public static void merge(int[] array, int[] tempArray, int p, int q, int r) {
153     int i = p; //initializing pointer to starting index of left subarray
154     int j = q + 1; //initializing pointer to starting index of right subarray
155     for (int k = p; k <= r; k++) { //copy all values in array to tempArray
156         tempArray[k] = array[k];
157     }
158     for (int k = p; k <= r; k++) {
159         if (i > q) { //if left subarray is empty, copy the element from the right subarray to the main array
160             array[k] = tempArray[j];
161             j++;
162         }
163         else if (j > r) { //if right subarray is empty, copy the element the left subarray to the main array
164             array[k] = tempArray[i];
165             i++;
166         }
167         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
168             array[k] = tempArray[j];
169             j++;
170         }
171         else { //else copy the element from the left subarray to the main array
172             array[k] = tempArray[i];
173             i++;
174         }
175     }
176 }
177
178 /*
179 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
180 Parameters: int[] array is an array of integers read from the text file in main()
181 Returns: nothing
182 Citations: https://chat.openai.com/share/1fb9df06-3168-486e-90f1-8587877af3e9
183 */
184 public static void buildMaxHeap(int[] array) {
185     int heapSize = array.length;
186     for (int i = heapSize / 2 - 1; i >= 0; i--) { //initializes i to the last non-leaf node
187         maxHeapify(array, heapSize, i);
188     }
189 }
190
191 /*
192 Description: maxHeapify() maintains the max heap properties during each recursion of heapSort()
193 Parameters:
194 int[] array is an array of integers read from the text file in main()
195 int i is the parent iterated over in buildMaxHeap()
196 int heapSize is the number of elements in the heap
197 Returns: nothing
198 Citations: https://chat.openai.com/share/1fb9df06-3168-486e-90f1-8587877af3e9
199 */
200 public static void maxHeapify(int[] array, int i, int heapSize) {
201     int largest = i; //index of the parent
202     int left = 2 * i + 1; //index of the left leaf
203     int right = 2 * i + 2; //index of the right leaf
204     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
205         largest = left; //set left leaf to parent
206     }
207     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
208         largest = right; //set right leaf to parent
209     }
210     if (largest != i) { //if the largest element is not the root of the maxheap, swap then and recursively heapify the affected sub-tree
211         int temp = array[i];
212         array[i] = array[largest];
213         array[largest] = temp;
214         maxHeapify(array, heapSize, largest);
215     }
216 }
217
218 /*
219 Description: calls buildMaxHeap() to build the max heap then iterates down the heap calling maxHeapify() to sort the heap
220 Parameters: int[] array is an array of integers read from the text file in main()
221 Returns: nothing
222 Citations: https://chat.openai.com/share/1fb9df06-3168-486e-90f1-8587877af3e9
223 */
224 public static void heapSort(int[] array) {
225     buildMaxHeap(array);
226     for (int i = array.length - 1; i > 0; i--) {
227         swap(array, 0, i);
228         maxHeapify(array, 0, i);
229     }
230 }
```

```
File Edit Selection View Go Run Terminal Help
CS-303-HW2

J HW2.java M X F 10:01
src > J HW2.java > HW2 > @ maxHeapify(int[], int, int)
177
178 /*
179 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
180 Parameters: int[] array is an array of integers read from the text file in main()
181 Returns: nothing
182 Citations: https://chat.openai.com/share/1fb9df06-3168-486e-90f1-8587877af3e9
183 */
184 public static void buildMaxHeap(int[] array) {
185     int heapSize = array.length;
186     for (int i = heapSize / 2 - 1; i >= 0; i--) { //initializes i to the last non-leaf node
187         maxHeapify(array, heapSize, i);
188     }
189 }
190
191 /*
192 Description: maxHeapify() maintains the max heap properties during each recursion of heapSort()
193 Parameters:
194 int[] array is an array of integers read from the text file in main()
195 int i is the parent iterated over in buildMaxHeap()
196 int heapSize is the number of elements in the heap
197 Returns: nothing
198 Citations: https://chat.openai.com/share/1fb9df06-3168-486e-90f1-8587877af3e9
199 */
200 public static void maxHeapify(int[] array, int i, int heapSize) {
201     int largest = i; //index of the parent
202     int left = 2 * i + 1; //index of the left leaf
203     int right = 2 * i + 2; //index of the right leaf
204     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
205         largest = left; //set left leaf to parent
206     }
207     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
208         largest = right; //set right leaf to parent
209     }
210     if (largest != i) { //if the largest element is not the root of the maxheap, swap then and recursively heapify the affected sub-tree
211         int temp = array[i];
212         array[i] = array[largest];
213         array[largest] = temp;
214         maxHeapify(array, heapSize, largest);
215     }
216 }
217
218 /*
219 Description: calls buildMaxHeap() to build the max heap then iterates down the heap calling maxHeapify() to sort the heap
220 Parameters: int[] array is an array of integers read from the text file in main()
221 Returns: nothing
222 Citations: https://chat.openai.com/share/1fb9df06-3168-486e-90f1-8587877af3e9
223 */
224 public static void heapSort(int[] array) {
225     buildMaxHeap(array);
226     for (int i = array.length - 1; i > 0; i--) {
227         swap(array, 0, i);
228         maxHeapify(array, 0, i);
229     }
230 }
```



```
File Edit Selection View Go Run Terminal Help
src > HW2.java M X 10.txt
src > HW2.java > HW2 > maxHeapify(int[], int, int)
217
218 /*
219 Description: calls buildMaxHeap() to build the max heap then iterates down the heap calling maxHeapify() to sort the heap
220 Parameters: int[] array is an array of integers read from the text file in main()
221 Returns: nothing
222 Citations: https://chat.openai.com/share/1f30df06-3168-486e-90f1-8587877af3e9
223 */
224 public static void heapSort(int[] array) {
225     buildMaxHeap(array);
226     for (int i = array.length - 1; i > 0; i--) {
227         maxHeapify(array, i, heapSize-1);
228     }
229 }
```

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.


```
File Edit Selection View Go Run Terminal Help
CS-303-HW2

EXPLOSER  PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS
OPEN EDITORS
X HW2.java src M
CS-303-HW2
> bin
> lib
> src
J HW2.java M
> .gitignore
> java.projects
> OUTLINE
main.c
main.c
CS-303 - HW2 Project Report...
README.zmd
README.md

VCS-303\HW2\CS-303-HW2> & 'C:\Program Files\Java\jdk-19\bin\java.exe' %*
"C:\Users\logan\AppData\Local\Temp\jckd9hjc9d9spatacs6f8dx_argfile" %*
3,3,4,4,6,7,11,12,14,15,16,16,18,18,19,19,20,22,22,26,28,29,29,30,31,32,33,36,36,37,37,38,39,40,40,41,42,42,45,45,48,49,49,50,58,58,51,51,52,53,55,57,58,58,59,61,61,64,65,66,67,70,70,70,71,72,74,75,76,76,79,81,
82,84,84,86,86,86,87,88,88,89,91,92,92,92,93,95,95,95,96,96,96,97,98,99,101,101,102,102,103,103,103,104,104,106,106,106,106,107,110,112,113,114,118,119,122,123,127,127,127,128,129,129,131,132,134,134,1
38,138,139,140,141,141,142,143,145,145,147,148,148,148,151,151,152,152,154,154,158,159,160,162,162,163,165,165,166,167,168,169,170,170,173,173,175,178,178,178,179,180,181,182,186,187,187,188,189,194,197,197,198,20
2,203,205,207,207,207,208,208,208,210,210,210,211,213,214,215,216,217,218,219,219,219,220,220,221,221,222,225,227,229,231,231,232,232,233,235,235,237,237,237,238,239,239,240,242,243,244,244,247,247,247,247,248,248,248,249,250,255,255,255,257,258,259,260,260,260,264,265,265,266,267,267,268,270,270,273,274,274,275,275,277,280,281,281,284,284,286,286,286,289,290,291,291,293,294,294,294,295,295,295,297,298,298,
298,299,300,303,304,305,308,312,312,312,313,313,314,314,315,315,315,317,319,320,321,321,322,323,324,326,327,330,332,335,338,340,342,343,345,345,346,346,347,348,348,348,350,352,353,354,357,359,359,363,363,3
65,366,367,368,369,369,369,373,375,375,376,376,377,378,378,380,380,382,383,383,384,384,385,385,385,386,387,388,388,388,392,392,392,393,394,394,398,400,402,403,404,407,407,408,409,410,410,411,411,412,41
3,416,416,417,417,419,419,420,420,421,423,424,425,428,429,430,431,431,432,432,432,434,434,436,436,437,437,438,438,440,442,442,444,446,446,447,448,452,452,453,454,454,455,458,459,460,460,461,462,465,466
467,469,469,470,470,471,472,472,472,475,475,477,478,479,483,485,485,486,489,489,489,491,492,494,494,495,496,496,499,499,500,500,500,502,502,505,506,506,509,509,514,514,516,516,517,518,518,518,520,521,523,524,524,524,
524,525,525,525,527,528,528,529,529,530,531,532,532,533,533,534,534,535,536,536,536,537,538,540,540,541,541,541,545,545,546,547,547,548,548,548,548,550,551,551,551,553,555,555,556,556,558,559,560,561,561,562,562,563,5
63,563,564,564,565,565,565,567,568,569,570,570,572,573,573,575,575,576,576,577,578,579,581,583,585,587,587,588,589,590,590,592,594,594,595,595,596,596,600,601,601,601,603,604,607,609,610,614,61
7,617,618,619,620,621,621,622,624,625,626,627,628,628,631,634,635,635,635,640,641,641,641,642,642,643,647,650,651,651,651,653,655,655,656,656,657,658,659,660,660,661,661,662,662,664,664,665,666,666
667,670,670,670,671,671,676,676,677,679,680,680,682,682,683,683,683,686,686,689,690,693,693,694,695,697,698,699,702,703,703,704,705,706,707,707,707,708,708,708,709,710,711,712,713,714,715,716,717,718,718,720,
722,723,725,726,726,727,729,729,729,730,731,732,732,733,734,736,737,737,738,740,740,740,740,741,742,744,744,746,748,749,750,751,752,754,755,755,756,756,757,758,760,764,765,766,766,766,767,771,7
73,774,775,776,776,777,777,778,778,779,779,780,782,785,786,786,786,789,791,791,791,793,794,795,796,796,798,799,800,801,801,802,803,804,804,807,807,808,808,809,810,811,812,812,813,813,814,815,815,817,819,819,820,82
1,823,823,824,824,825,826,826,826,827,829,830,832,833,833,834,835,835,836,836,838,839,839,840,842,842,842,843,843,844,845,845,845,846,847,847,847,848,848,848,851,851,857,857,858,858,859,859,860,861,861,863,863
865,866,866,868,869,869,871,872,874,874,875,875,875,876,877,878,878,879,880,881,883,883,884,884,885,885,886,887,888,889,891,893,893,894,897,898,898,899,899,900,901,902,905,908,910,913,913,914,916,917,917,
918,918,920,922,924,924,925,926,930,931,932,933,933,934,938,939,941,942,943,944,944,944,946,947,948,948,948,948,948,955,955,957,957,958,964,965,965,967,967,967,969,969,970,970,970,970,971,972,974,975,975,978,9
79,979,980,980,980,980,982,986,987,987,988,992,997,998,1000,1000
Insertion Sort Time:0.001900 nanoseconds, 0.0019 milliseconds, or 0.0000019 seconds
Merge Sort Time:5.878000 nanoseconds, 0.5878 milliseconds, or 5.878E-4 seconds
Modified Merge Sort Time:1.888000 nanoseconds, 0.1888 milliseconds, or 1.888E-4 seconds
Heap Sort Time:14.265000 nanoseconds, 0.1426 milliseconds, or 1.426E-4 seconds
PS C:\Users\logan\OneDrive - UAB - The University of Alabama at Birmingham\UAB Files\VA 2023\CS-303\HW2>
```

Example output of linear search algorithm.

```
File Edit Selection View Go Run Terminal Help
CS-303-HW2

EXPLOSER  PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS
OPEN EDITORS
X HW2.java src M
CS-303-HW2
> bin
> lib
> src
J HW2.java M
> .gitignore
> java.projects
> OUTLINE
main.c
main.c
CS-303 - HW2 Project Report...
README.zmd
README.md

VCS-303\HW2\CS-303-HW2> & 'C:\Program Files\Java\jdk-19\bin\java.exe' %*
"C:\Users\logan\AppData\Local\Temp\jckd9hjc9d9spatacs6f8dx_argfile" %*
3,3,4,4,6,7,11,12,14,15,16,16,18,18,19,19,20,22,22,26,28,29,29,30,31,32,33,36,36,37,37,38,39,40,40,41,42,42,45,45,48,49,49,50,58,58,51,51,52,53,55,57,58,58,59,61,61,64,65,66,67,70,70,70,71,72,74,75,76,76,79,81,
82,84,84,86,86,87,88,88,89,91,92,92,92,93,95,95,95,96,96,96,97,98,99,101,101,102,102,103,103,103,104,104,106,106,106,106,107,110,112,113,114,118,119,122,123,127,127,127,128,129,129,131,132,134,134,1
38,138,139,140,141,141,142,143,145,145,147,148,148,148,151,151,152,152,154,154,158,159,160,162,162,163,165,165,166,167,168,169,170,170,173,173,175,178,178,178,179,180,181,182,186,187,187,188,189,194,197,197,198,20
2,203,205,207,207,207,208,208,208,210,210,210,211,213,214,215,216,217,218,219,219,219,220,220,221,221,222,225,227,229,231,231,232,232,233,235,235,237,237,237,238,239,239,240,242,243,244,244,247,247,247,248,248,248,249,250,255,255,255,257,258,259,260,260,264,265,265,266,267,267,268,270,270,273,274,274,275,275,277,280,281,281,284,284,286,286,286,289,290,291,291,293,294,294,294,295,295,295,297,298,298,
298,299,300,303,304,305,308,312,312,312,313,313,314,314,315,315,315,317,319,320,321,321,322,323,324,326,327,330,332,335,338,340,342,343,345,345,346,346,347,348,348,348,350,352,353,354,357,359,359,363,363,3
65,366,367,368,369,369,369,373,375,375,376,376,377,378,378,380,380,382,383,383,384,384,385,385,385,386,387,388,388,388,392,392,392,393,394,394,398,400,402,403,404,407,407,408,409,410,410,411,411,412,41
3,416,416,417,417,419,419,420,421,423,424,425,428,429,430,430,431,431,432,432,432,434,434,436,436,437,437,438,438,440,442,442,444,446,446,447,448,452,452,453,454,454,455,458,459,460,460,461,462,465,466
467,469,469,470,470,471,472,472,472,475,475,477,478,479,483,485,485,486,489,489,489,491,492,494,494,495,496,496,499,499,500,500,500,502,502,505,506,506,509,509,514,514,516,516,517,518,518,518,520,521,523,524,524,524,
524,525,525,525,527,528,528,529,529,530,531,532,532,533,533,534,534,535,536,536,536,537,538,540,540,541,541,541,545,545,546,547,547,548,548,548,548,550,551,551,551,553,555,555,556,556,558,559,560,561,561,562,562,563,5
63,563,564,564,565,565,565,567,568,569,570,570,572,573,573,575,575,576,576,577,578,579,581,583,585,587,587,588,589,590,590,592,594,594,595,595,596,596,600,601,601,601,603,604,607,609,610,614,61
7,617,618,619,620,621,621,622,624,625,626,627,628,628,631,634,635,635,635,640,641,641,641,642,642,643,647,650,651,651,651,653,655,655,656,656,657,658,659,660,660,661,661,662,662,664,664,665,666,666
667,670,670,670,671,671,676,676,677,679,680,680,682,682,683,683,683,686,686,689,690,693,693,694,695,697,698,699,702,703,703,704,705,706,707,707,707,708,708,708,709,710,711,712,713,714,715,716,717,718,718,720,
722,723,725,726,726,727,729,729,729,730,731,732,732,733,734,736,737,737,738,740,740,740,740,741,742,744,744,746,748,749,750,751,752,754,755,755,756,756,757,758,760,764,765,766,766,766,767,771,7
73,774,775,776,776,777,777,778,778,779,779,780,782,785,786,786,786,789,791,791,791,793,794,795,796,796,798,799,800,800,801,802,803,804,804,807,807,808,808,809,810,811,812,812,813,813,814,815,815,817,819,819,820,82
1,823,823,824,824,825,826,826,826,827,829,830,832,833,833,834,835,835,836,836,838,839,839,840,842,842,842,843,843,844,845,845,845,846,847,847,847,848,848,848,851,851,857,857,858,858,859,859,860,861,861,863,863
865,866,866,868,869,869,871,872,874,874,875,875,875,876,877,878,878,879,880,881,883,883,884,884,885,885,886,887,888,889,891,893,893,894,897,898,898,899,899,900,901,902,905,908,910,913,913,914,916,917,917,
918,918,920,922,924,924,925,926,930,931,932,933,933,934,938,939,941,942,943,944,944,944,946,947,948,948,948,948,948,948,955,955,957,957,958,964,965,965,967,967,967,969,969,970,970,970,970,971,972,974,975,975,978,9
79,979,980,980,980,980,982,986,987,987,988,992,997,998,1000,1000
Merge Sort Time:6.887000 nanoseconds, 0.6887 milliseconds, or 6.887E-4 seconds
Modified Merge Sort Time:9.080000 nanoseconds, 0.908 milliseconds, or 0.1E-3 seconds
Heap Sort Time:24.580000 nanoseconds, 0.0245 milliseconds, or 7.43E-5 seconds
PS C:\Users\logan\OneDrive - UAB - The University of Alabama at Birmingham\UAB Files\VA 2023\CS-303\HW2>
```

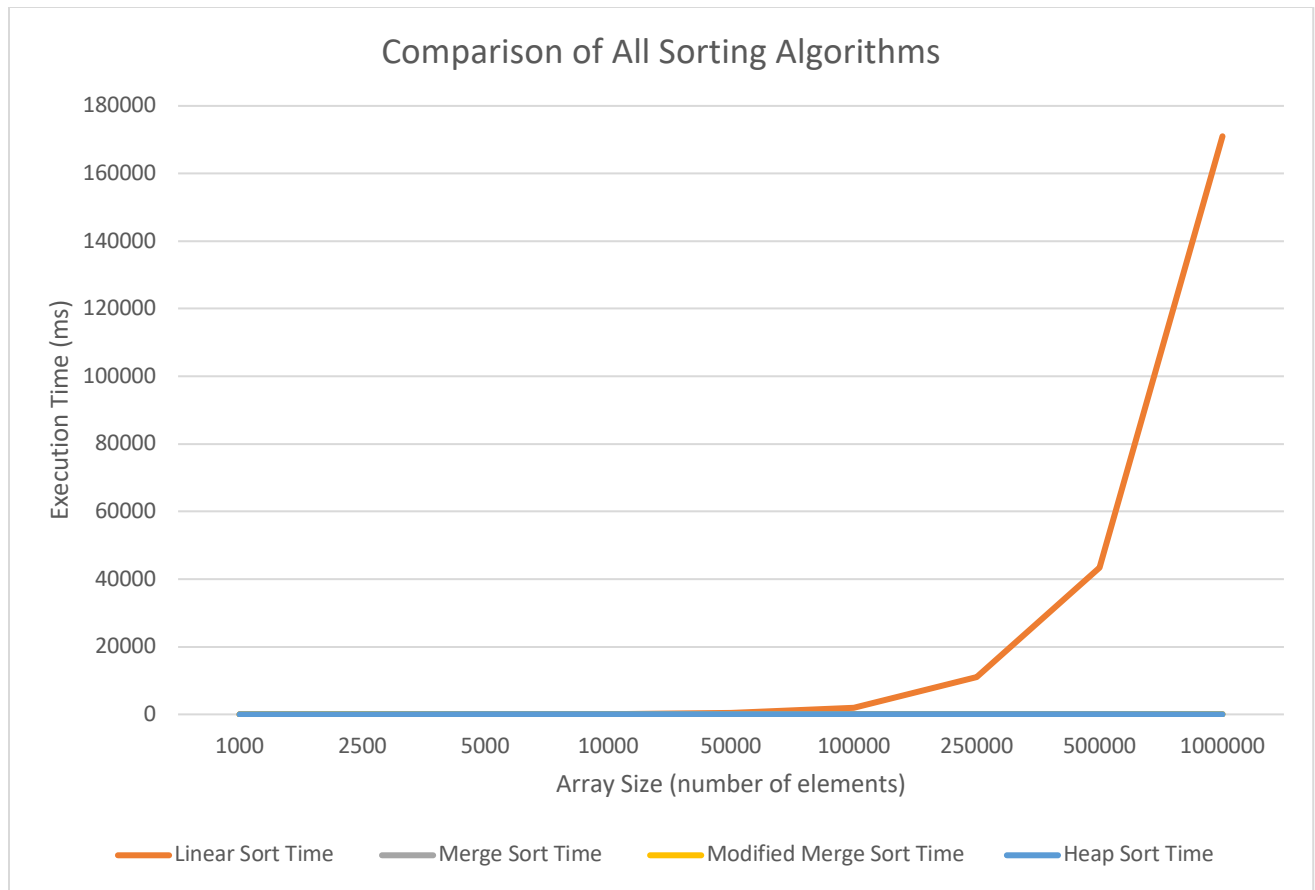
Example output of merge sort algorithm.

```
File Edit Selection View Go Run Terminal Help
CS-303-HW2
EXPLORER
OPEN EDITORS
HW2.java src M
bin
src
HW2.java M
.gitignore
1000.txt
2500.txt
5000.txt
10000.txt
50000.txt
100000.txt
250000.txt
500000.txt
1000000.txt
CS-303 - HW2 Project Report...
README.zmd
README.md
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
CS-303\VMCS-303-HW2: & 'C:\Program Files\Java\jdk-19\bin\java.exe' -D%USERLOG%\AppData\Local\Temp\jckd9jckd9\spatacs\of\sbx_argfile' %H2
Insertion Sort Time:3039000 nanoseconds, 3.0396 milliseconds, or 0.0030396 seconds
Merge Sort Time:973700 nanoseconds, 0.9737 milliseconds, or 0.713E-4 seconds
Modified Merge Sort Time:233000 nanoseconds, 0.233 milliseconds, or 2.33E-4 seconds
Heap Sort Time:162400 nanoseconds, 0.1624 milliseconds, or 1.624E-4 seconds
PS C:\Users\logan\OneDrive - UAB - The University of Alabama at Birmingham\UAB Files\FA 2023\CS-303\VMCS-303-HW2:
3,3,4,4,6,7,11,12,14,15,16,18,18,19,20,20,22,22,26,28,29,29,30,31,32,33,36,36,37,37,38,39,40,40,41,42,42,45,45,48,49,49,50,50,50,51,51,52,53,55,57,58,58,59,61,61,64,65,66,67,70,70,70,71,72,74,75,76,76,79,81,82,84,84,85,86,86,87,88,88,89,91,92,92,92,93,95,95,96,96,96,97,98,99,101,101,102,102,103,103,104,104,104,105,106,106,106,107,110,112,113,114,118,119,122,123,125,127,127,128,128,129,129,129,131,132,134,134,138,138,139,140,141,141,142,143,145,145,147,148,148,148,151,151,152,152,154,154,155,159,160,162,163,165,165,166,166,166,169,169,170,173,173,175,178,178,178,179,180,181,182,186,187,187,188,189,194,197,197,198,202,203,205,207,207,208,208,208,210,210,211,213,214,215,216,217,218,219,219,220,220,221,221,222,225,227,229,231,231,232,232,233,235,235,237,237,237,238,239,239,240,242,243,243,244,244,247,247,247,247,248,248,248,249,250,255,255,256,257,258,259,260,260,260,264,265,265,266,267,267,268,270,270,273,274,274,274,275,275,277,280,281,281,284,284,286,286,286,288,288,290,291,291,293,294,294,294,295,295,296,297,298,298,299,300,303,304,305,308,312,312,312,313,313,314,314,315,315,317,319,320,321,321,322,323,324,326,327,330,330,332,335,338,340,342,343,345,345,346,346,347,348,349,350,350,352,353,354,357,359,359,363,363,65,366,367,368,369,369,369,373,375,375,376,376,377,378,378,380,380,380,382,383,383,384,384,385,385,386,387,388,388,388,388,392,392,393,394,394,398,400,402,403,404,407,407,408,409,410,410,411,411,412,414,3,416,416,417,417,419,419,420,420,421,423,424,425,428,429,430,430,431,431,432,432,432,432,434,436,436,437,437,438,438,440,442,442,442,444,446,446,447,448,452,452,453,454,454,455,458,459,459,459,461,462,465,466,467,469,469,470,470,471,472,472,472,475,475,477,478,479,483,485,485,486,486,489,489,490,491,492,494,494,495,496,496,499,499,500,500,502,502,505,506,509,514,514,516,516,517,518,518,520,521,523,524,524,524,524,525,525,525,527,528,528,529,530,530,531,532,533,533,534,534,535,536,536,537,538,540,541,541,541,545,545,546,547,547,548,548,548,550,551,551,551,553,555,555,556,558,559,561,561,562,562,563,63,563,564,564,565,565,566,567,568,569,570,570,572,573,573,575,575,576,576,577,578,579,581,583,585,585,587,587,588,589,589,590,590,592,594,594,595,595,596,596,600,601,601,601,601,604,607,609,609,610,610,614,616,617,617,618,619,620,621,621,622,624,625,626,627,628,628,631,634,635,635,636,636,641,641,641,642,642,643,647,650,651,651,651,651,655,655,656,656,657,658,659,660,660,661,662,662,664,664,666,666,667,670,670,670,671,671,676,676,677,679,680,680,682,682,683,683,684,686,689,690,693,693,694,695,697,698,699,702,703,703,704,705,706,707,707,707,707,708,708,709,710,711,712,713,714,715,717,717,718,718,720,722,723,725,726,726,727,729,729,729,730,731,732,732,733,734,736,737,737,738,740,740,740,740,741,742,744,744,746,746,748,750,751,752,754,755,755,756,756,757,758,760,764,765,766,766,766,767,767,771,773,774,775,776,776,777,777,778,779,779,780,782,785,786,786,786,789,791,791,791,793,794,795,796,796,799,800,800,801,802,803,804,804,805,807,808,808,809,810,811,812,812,813,814,815,815,817,817,819,819,820,823,823,823,824,824,825,826,826,827,829,830,832,833,833,834,835,835,836,836,838,839,839,840,842,842,843,843,844,845,845,845,846,847,847,847,848,848,848,851,853,857,857,858,859,859,860,861,863,863,863,863,865,866,866,868,869,869,871,872,873,874,874,875,875,875,876,877,878,878,879,879,880,881,883,883,884,884,885,885,886,887,888,888,891,891,893,893,894,897,898,898,899,900,901,902,905,908,910,913,913,914,916,917,917,918,919,920,922,924,924,925,928,930,931,932,933,933,934,938,939,941,942,943,944,944,944,944,946,947,948,948,948,950,955,955,957,957,957,958,964,965,965,965,967,967,967,969,969,970,970,970,971,972,974,975,975,978,979,979,980,980,980,980,982,986,987,988,992,997,999,1000
Modified Merge Sort Time:233000 nanoseconds, 0.233 milliseconds, or 2.33E-4 seconds
Heap Sort Time:162400 nanoseconds, 0.1624 milliseconds, or 1.624E-4 seconds
PS C:\Users\logan\OneDrive - UAB - The University of Alabama at Birmingham\UAB Files\FA 2023\CS-303\VMCS-303-HW2:
11m15s, 6 hours ago Ln 58, Col 1 (4 selected) Spaces 4 UTF-8 LF ( ) Java
```

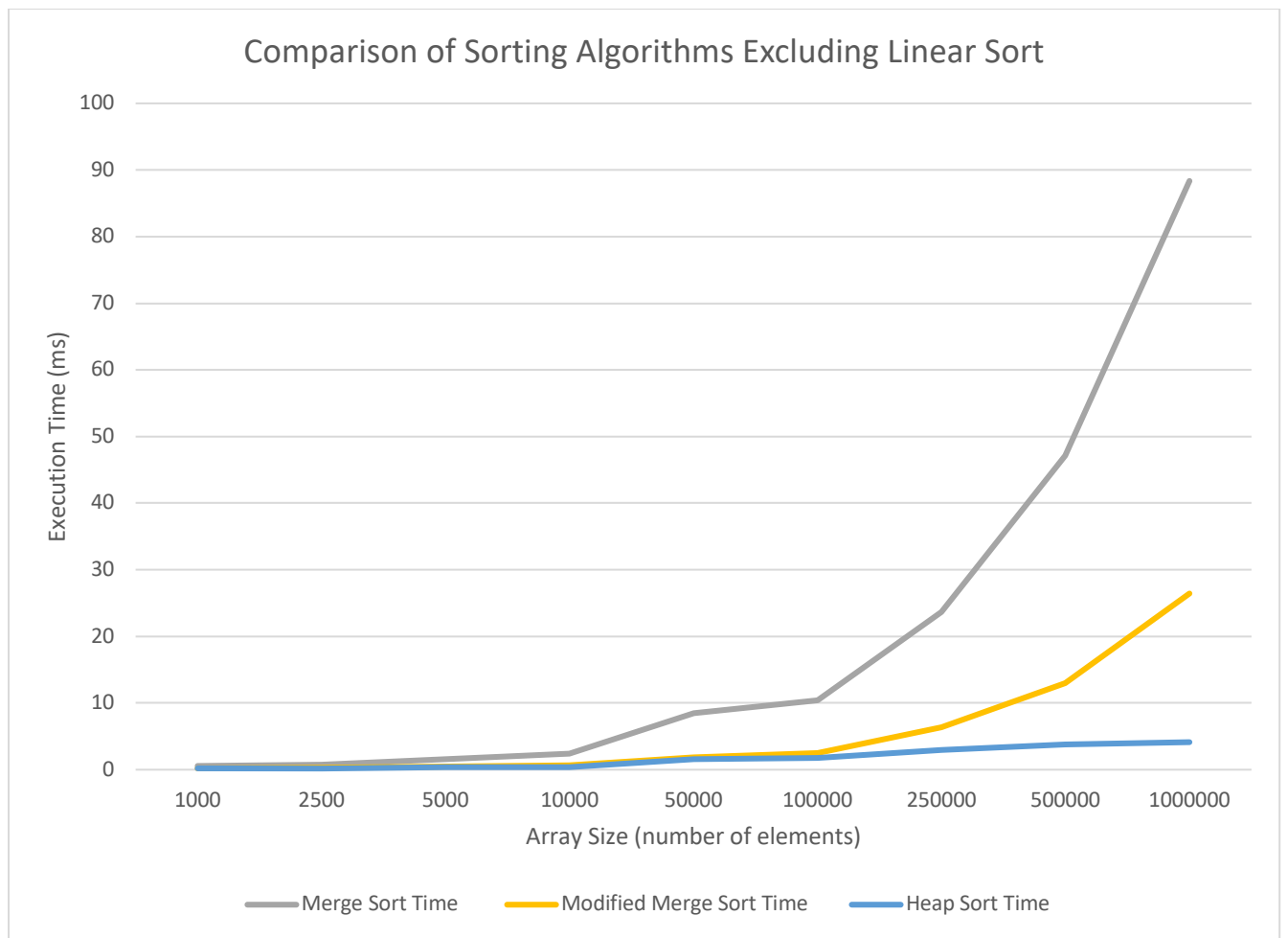
Example output of modified merge sort algorithm.

```
File Edit Selection View Go Run Terminal Help
CS-303-HW2
EXPLORER
OPEN EDITORS
HW2.java src M
bin
src
HW2.java M
.gitignore
1000.txt
2500.txt
5000.txt
10000.txt
50000.txt
100000.txt
250000.txt
500000.txt
1000000.txt
CS-303 - HW2 Project Report...
README.zmd
README.md
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
CS-303\VMCS-303-HW2: & 'C:\Program Files\Java\jdk-19\bin\java.exe' -D%USERLOG%\AppData\Local\Temp\jckd9jckd9\spatacs\of\sbx_argfile' %H2
Insertion Sort Time:3137200 nanoseconds, 3.1372 milliseconds, or 0.0031372 seconds
Merge Sort Time:436000 nanoseconds, 0.436 milliseconds, or 4.36E-4 seconds
Modified Merge Sort Time:90300 nanoseconds, 0.0903 milliseconds, or 9.03E-5 seconds
Heap Sort Time:13300 nanoseconds, 0.133 milliseconds, or 1.33E-4 seconds
PS C:\Users\logan\OneDrive - UAB - The University of Alabama at Birmingham\UAB Files\FA 2023\CS-303\VMCS-303-HW2:
3,3,4,4,6,7,11,12,14,15,16,18,18,19,20,20,22,22,26,28,29,29,30,31,32,33,36,36,37,37,38,39,40,40,41,42,42,45,45,48,49,49,50,50,50,51,51,52,53,55,57,58,58,59,61,61,64,65,66,67,70,70,70,71,72,74,75,76,76,79,81,82,84,84,85,86,86,87,88,88,89,91,92,92,92,93,95,95,96,96,96,97,98,99,101,101,102,102,103,103,104,104,104,105,106,106,106,107,110,112,113,114,118,119,122,123,125,127,127,128,128,129,129,129,131,132,134,134,138,138,139,140,141,141,142,143,145,145,147,148,148,148,151,151,152,152,154,154,155,159,160,162,163,165,165,166,166,166,169,169,170,173,173,175,178,178,178,179,180,181,182,186,187,187,188,189,194,197,197,198,202,203,205,207,207,208,208,208,210,210,211,213,214,215,216,217,218,219,219,220,220,221,221,222,225,227,229,231,231,232,233,235,235,237,237,237,238,239,239,240,242,243,243,244,244,247,247,247,247,248,248,248,249,250,255,255,256,257,258,259,260,260,260,264,265,265,266,267,267,268,270,270,273,274,274,274,275,275,277,280,281,281,284,284,286,286,286,290,291,291,293,294,294,294,295,295,296,297,298,298,299,300,303,304,305,308,312,312,313,313,314,314,315,315,317,319,320,321,321,322,323,324,326,327,330,330,332,335,338,340,342,343,345,346,346,347,348,349,350,352,353,354,357,359,359,363,363,65,366,367,368,369,369,369,373,375,375,376,376,377,778,778,380,380,382,383,383,384,384,385,385,386,387,388,388,388,392,392,393,394,394,398,400,402,403,404,407,407,408,409,410,410,411,411,412,414,3,416,416,417,417,419,419,420,420,421,423,424,425,428,429,430,430,431,431,432,432,432,432,434,436,436,437,437,438,438,440,442,442,442,444,446,446,447,448,452,452,453,454,454,455,458,459,459,459,461,462,465,466,467,469,469,470,470,471,472,472,475,475,477,478,479,483,485,485,486,486,489,489,490,491,492,494,494,495,496,496,499,499,500,500,502,502,505,506,509,514,514,516,516,517,518,518,520,521,523,524,524,524,525,525,525,527,528,528,529,530,530,531,532,533,533,534,534,535,536,536,537,538,540,541,541,541,545,545,546,547,547,548,548,548,550,551,551,551,553,555,555,556,558,559,561,561,562,562,563,63,563,564,564,565,565,566,567,568,569,570,570,572,573,573,575,575,576,576,577,578,579,581,583,585,585,587,587,588,589,589,590,590,592,594,594,595,595,596,596,600,601,601,601,601,604,607,609,609,610,610,614,616,617,617,618,619,620,621,621,622,624,625,626,627,628,628,631,634,635,635,636,636,641,641,641,642,642,643,647,650,651,651,651,651,655,655,656,656,657,658,659,660,660,661,662,662,664,664,666,666,667,670,670,670,671,671,676,676,677,679,680,680,682,682,683,683,684,686,689,690,693,693,694,695,697,698,699,702,703,703,704,705,706,707,707,707,707,708,708,709,710,711,712,713,714,715,717,717,718,718,720,722,723,725,726,726,727,729,729,729,730,731,732,732,733,734,736,737,737,738,740,740,740,741,742,744,744,746,746,748,750,751,752,754,755,755,756,756,757,758,760,764,765,766,766,766,767,767,771,773,774,775,776,776,777,777,778,779,779,780,782,785,786,786,786,789,791,791,791,793,794,795,796,796,799,800,800,801,802,803,804,804,805,807,808,808,809,810,811,812,812,813,814,815,815,817,817,819,819,820,823,823,823,824,824,825,826,826,827,829,830,832,833,833,834,835,835,836,836,838,839,839,840,842,842,843,843,844,845,845,845,846,847,847,847,848,848,848,851,853,857,857,858,859,859,860,861,863,863,863,863,865,866,866,868,869,869,871,872,873,874,874,875,875,875,876,877,878,878,879,879,880,881,883,883,884,884,885,885,886,887,888,888,891,891,893,893,894,897,898,898,899,900,901,902,905,908,910,913,913,914,916,917,917,918,919,920,922,924,924,925,928,930,931,932,933,933,934,938,939,941,942,943,944,944,944,944,946,947,948,948,948,950,955,955,957,957,957,958,964,965,965,965,967,967,967,969,969,970,970,970,971,972,974,975,975,978,979,979,980,980,980,980,982,986,987,988,992,997,999,1000
Modified Merge Sort Time:90300 nanoseconds, 0.0903 milliseconds, or 9.03E-5 seconds
Heap Sort Time:13300 nanoseconds, 0.133 milliseconds, or 1.33E-4 seconds
PS C:\Users\logan\OneDrive - UAB - The University of Alabama at Birmingham\UAB Files\FA 2023\CS-303\VMCS-303-HW2:
13m10s, 13 minutes ago Ln 91, Col 1 (225 selected) Spaces 4 UTF-8 LF ( ) Java
```

Example output of heap sort algorithm.

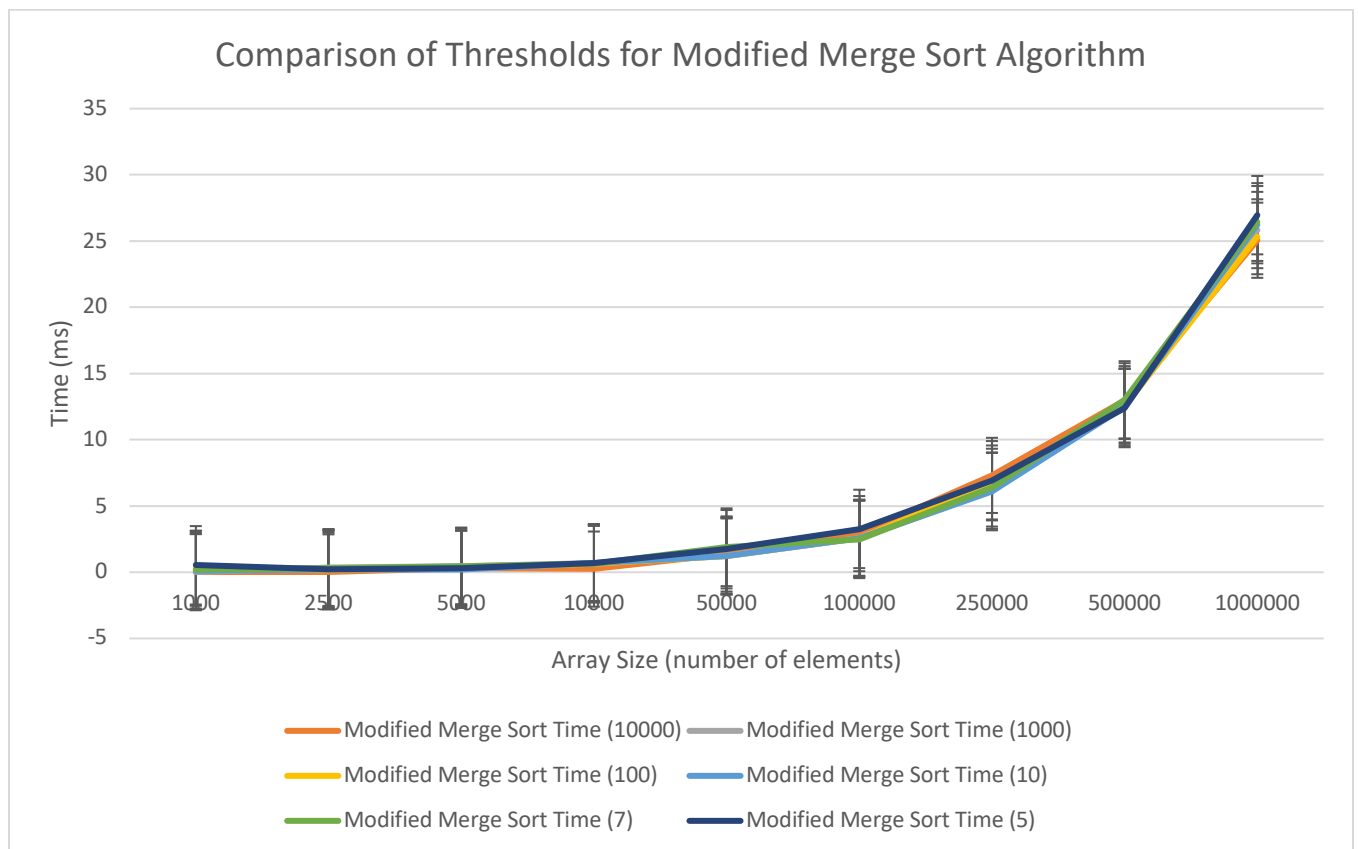


A comparison of all the search algorithms reveals the linear time complexity of $O(n)$ of linear sort. The graph shows that the execution time is manageable when dealing with smaller arrays, but when the number of elements in the array increases, so too does the execution time of the linear search algorithm. This is most apparent with the datapoint at $n = 1000000$. The time complexity of the other algorithms is $O(n\log(n))$. 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 $O(n\log(n))$. 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 $O(n\log(n))$ 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/>