

# 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.

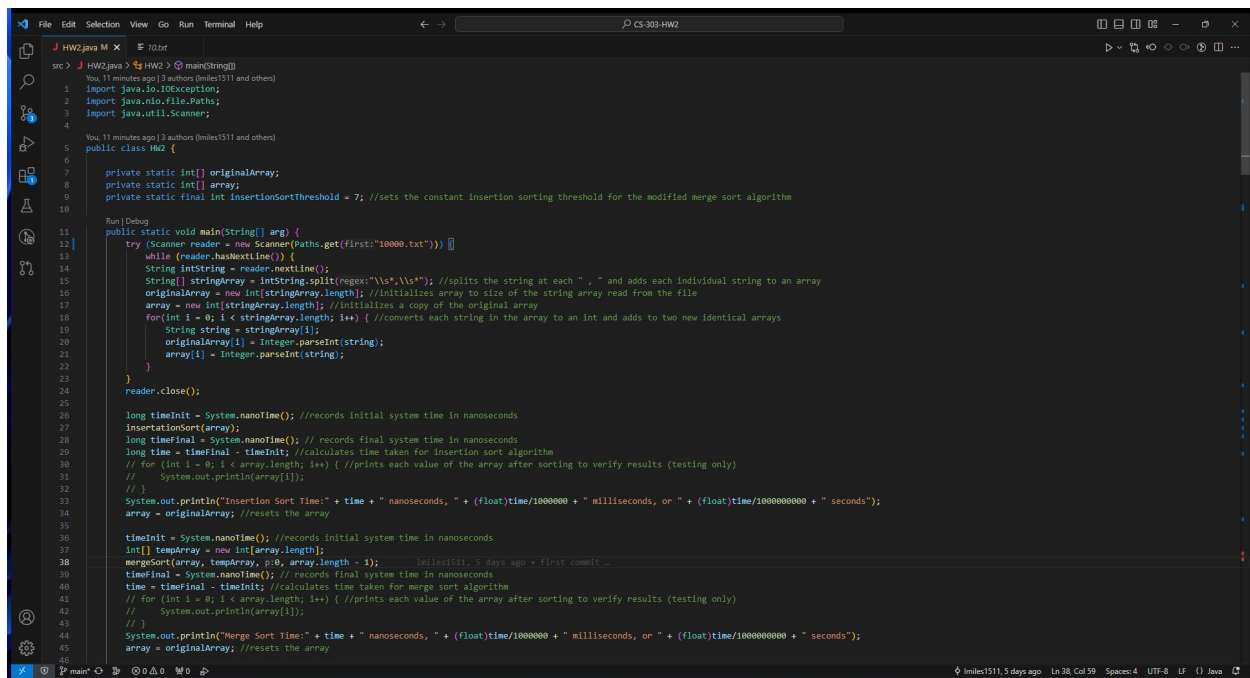
### **maxHeapify()**

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 E 10.txt
You, 11 minutes ago | 3 authors (lmiles1511 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 (lmiles1511 and others)
6 public class HW2 {
7
8     private static int[] originalArray;
9     private static int[] array;
10     private static final int insertionSortThreshold = 7; //sets the constant insertion sorting threshold for the modified merge sort algorithm
11
12     Run [Debug]
13     public static void main(String[] arg) {
14         try (Scanner reader = new Scanner(Paths.get(first("10000.txt")))) {}
15         while (reader.hasNextLine()) {
16             String intString = reader.nextLine();
17             String[] stringArray = intString.split(regex("\\s+")); //splits the string at each " " and adds each individual string to an array
18             originalArray = new int[stringArray.length]; //initializes array to size of the string array read from the file
19             array = new int[stringArray.length]; //initializes a copy of the original array
20             for(int i = 0; i < stringArray.length; i++) { //converts each string in the array to an int and adds to two new identical arrays
21                 String string = stringArray[i];
22                 originalArray[i] = Integer.parseInt(string);
23                 array[i] = Integer.parseInt(string);
24             }
25         }
26         reader.close();
27
28         long timeInit = System.nanoTime(); //records initial system time in nanoseconds
29         insertionSort(array);
30         long timeFinal = System.nanoTime(); // records final system time in nanoseconds
31         long time = timeFinal - timeInit; //calculates time taken for insertion sort algorithm
32         // for (int i = 0; i < array.length; i++) { //prints each value of the array after sorting to verify results (testing only)
33         //     System.out.println(array[i]);
34         // }
35         System.out.println("Insertion Sort Time: " + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");
36         array = originalArray; //resets the array
37
38         timeInit = System.nanoTime(); //records initial system time in nanoseconds
39         int[] tempArray = new int[array.length];
40         mergeSort(array, tempArray, 0, array.length - 1);
41         timeFinal = System.nanoTime(); // records final system time in nanoseconds
42         time = timeFinal - timeInit; //calculates time taken for merge sort algorithm
43         // for (int i = 0; i < array.length; i++) { //prints each value of the array after sorting to verify results (testing only)
44         //     System.out.println(array[i]);
45         // }
46         System.out.println("Merge Sort Time: " + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");
47         array = originalArray; //resets the array
48     }
49 }
```

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

J HW2.java M X E 10:12r

src > J HW2.java > HW2 > main(String[])
43 //
44 System.out.println("Merge Sort Time:" + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");
45 array = originalArray; //resets the array
46
47 timeInit = System.nanoTime(); //records initial system time in nanoseconds
48 modifiedMergeSort(array, tempArray, p=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 Citations:
76 https://chat.openai.com/share/e8124443-6028-4959-9653-b91b956b0b12
77 https://www.geeksforgeeks.org/insertion-sort/
78 //
79 public static void insertionSort(int[] array) {
80     int i, j, k; //initializes count variables
81     for (i = 0; i < array.length; i++) {
82         k = array[i]; //set the key to the value of the current index
83         j = i - 1; //j is the previous index
84         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
85             array[j + 1] = array[j]; //shifts value at index one position to the right, making space for key
86             j = j - 1; //the j index is moved one left for further comparisons
87         }
88         array[j + 1] = k; //places the k value at the correct sorted position
89     }
90 }
91 //
92 //
93 //
94 Description: mergeSort() calculates the middle index, if the recursively calls mergeSort() on the left and right subarrays in order to sort them,
95 then calls merge() to merge the newly sorted subarrays
96 Parameters:
97 int[] array is an array of integers read from the text file in main()
98 int[] temparray is a temporary copy of array for iteration purposes
99 int p is a pointer for starting index of left subarray
100 int r is a pointer for ending index of right subarray
101 Returns: nothing
102 Citations:
103 https://chat.openai.com/share/c8f53e73-fb52-483b-846c-c756faa3bc6e
104 https://www.geeksforgeeks.org/merge-sort/
105 //
106 public static void mergeSort(int[] array, int[] tempArray, int p, int r) {
107     if (p < r) {
108         int q = (p + r) / 2; //calculate middle index
109         mergeSort(array, tempArray, p, q); //recursively sort left subarray
110         mergeSort(array, tempArray, q + 1, r); //recursively sort right subarray
111         merge(array, tempArray, p, q, r); //merge the two sorted subarrays
112     }
113 }
114 //
115 Description: modifiedMergeSort() 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 //
139 //
140 //
141 //
142 //
143 //
144 //
145 //
146 //
147 //
148 //
149 //
150 //
151 //
152 //
153 //
154 //
155 //
156 //
157 //
158 //
159 //
160 //
161 //
162 //
163 //
164 //
165 //
166 //
167 //
168 //
169 //
170 //
171 //
172 //
173 //
174 //
175 //
176 //
177 //
178 //
179 //
180 //
181 //
182 //
183 //
184 //
185 //
186 //
187 //
188 //
189 //
190 //
191 //
192 //
193 //
194 //
195 //
196 //
197 //
198 //
199 //
200 //
201 //
202 //
203 //
204 //
205 //
206 //
207 //
208 //
209 //
210 //
211 //
212 //
213 //
214 //
215 //
216 //
217 //
218 //
219 //
220 //
221 //
222 //
223 //
224 //
225 //
226 //
227 //
228 //
229 //
230 //
231 //
232 //
233 //
234 //
235 //
236 //
237 //
238 //
239 //
240 //
241 //
242 //
243 //
244 //
245 //
246 //
247 //
248 //
249 //
250 //
251 //
252 //
253 //
254 //
255 //
256 //
257 //
258 //
259 //
260 //
261 //
262 //
263 //
264 //
265 //
266 //
267 //
268 //
269 //
270 //
271 //
272 //
273 //
274 //
275 //
276 //
277 //
278 //
279 //
280 //
281 //
282 //
283 //
284 //
285 //
286 //
287 //
288 //
289 //
290 //
291 //
292 //
293 //
294 //
295 //
296 //
297 //
298 //
299 //
300 //
301 //
302 //
303 //
304 //
305 //
306 //
307 //
308 //
309 //
310 //
311 //
312 //
313 //
314 //
315 //
316 //
317 //
318 //
319 //
320 //
321 //
322 //
323 //
324 //
325 //
326 //
327 //
328 //
329 //
330 //
331 //
332 //
333 //
334 //
335 //
336 //
337 //
338 //
339 //
340 //
341 //
342 //
343 //
344 //
345 //
346 //
347 //
348 //
349 //
350 //
351 //
352 //
353 //
354 //
355 //
356 //
357 //
358 //
359 //
360 //
361 //
362 //
363 //
364 //
365 //
366 //
367 //
368 //
369 //
370 //
371 //
372 //
373 //
374 //
375 //
376 //
377 //
378 //
379 //
380 //
381 //
382 //
383 //
384 //
385 //
386 //
387 //
388 //
389 //
390 //
391 //
392 //
393 //
394 //
395 //
396 //
397 //
398 //
399 //
400 //
401 //
402 //
403 //
404 //
405 //
406 //
407 //
408 //
409 //
410 //
411 //
412 //
413 //
414 //
415 //
416 //
417 //
418 //
419 //
420 //
421 //
422 //
423 //
424 //
425 //
426 //
427 //
428 //
429 //
430 //
431 //
432 //
433 //
434 //
435 //
436 //
437 //
438 //
439 //
440 //
441 //
442 //
443 //
444 //
445 //
446 //
447 //
448 //
449 //
450 //
451 //
452 //
453 //
454 //
455 //
456 //
457 //
458 //
459 //
460 //
461 //
462 //
463 //
464 //
465 //
466 //
467 //
468 //
469 //
470 //
471 //
472 //
473 //
474 //
475 //
476 //
477 //
478 //
479 //
480 //
481 //
482 //
483 //
484 //
485 //
486 //
487 //
488 //
489 //
490 //
491 //
492 //
493 //
494 //
495 //
496 //
497 //
498 //
499 //
500 //
501 //
502 //
503 //
504 //
505 //
506 //
507 //
508 //
509 //
510 //
511 //
512 //
513 //
514 //
515 //
516 //
517 //
518 //
519 //
520 //
521 //
522 //
523 //
524 //
525 //
526 //
527 //
528 //
529 //
530 //
531 //
532 //
533 //
534 //
535 //
536 //
537 //
538 //
539 //
540 //
541 //
542 //
543 //
544 //
545 //
546 //
547 //
548 //
549 //
550 //
551 //
552 //
553 //
554 //
555 //
556 //
557 //
558 //
559 //
560 //
561 //
562 //
563 //
564 //
565 //
566 //
567 //
568 //
569 //
570 //
571 //
572 //
573 //
574 //
575 //
576 //
577 //
578 //
579 //
580 //
581 //
582 //
583 //
584 //
585 //
586 //
587 //
588 //
589 //
590 //
591 //
592 //
593 //
594 //
595 //
596 //
597 //
598 //
599 //
600 //
601 //
602 //
603 //
604 //
605 //
606 //
607 //
608 //
609 //
610 //
611 //
612 //
613 //
614 //
615 //
616 //
617 //
618 //
619 //
620 //
621 //
622 //
623 //
624 //
625 //
626 //
627 //
628 //
629 //
630 //
631 //
632 //
633 //
634 //
635 //
636 //
637 //
638 //
639 //
640 //
641 //
642 //
643 //
644 //
645 //
646 //
647 //
648 //
649 //
650 //
651 //
652 //
653 //
654 //
655 //
656 //
657 //
658 //
659 //
660 //
661 //
662 //
663 //
664 //
665 //
666 //
667 //
668 //
669 //
670 //
671 //
672 //
673 //
674 //
675 //
676 //
677 //
678 //
679 //
680 //
681 //
682 //
683 //
684 //
685 //
686 //
687 //
688 //
689 //
690 //
691 //
692 //
693 //
694 //
695 //
696 //
697 //
698 //
699 //
700 //
701 //
702 //
703 //
704 //
705 //
706 //
707 //
708 //
709 //
710 //
711 //
712 //
713 //
714 //
715 //
716 //
717 //
718 //
719 //
720 //
721 //
722 //
723 //
724 //
725 //
726 //
727 //
728 //
729 //
730 //
731 //
732 //
733 //
734 //
735 //
736 //
737 //
738 //
739 //
740 //
741 //
742 //
743 //
744 //
745 //
746 //
747 //
748 //
749 //
750 //
751 //
752 //
753 //
754 //
755 //
756 //
757 //
758 //
759 //
760 //
761 //
762 //
763 //
764 //
765 //
766 //
767 //
768 //
769 //
770 //
771 //
772 //
773 //
774 //
775 //
776 //
777 //
778 //
779 //
780 //
781 //
782 //
783 //
784 //
785 //
786 //
787 //
788 //
789 //
790 //
791 //
792 //
793 //
794 //
795 //
796 //
797 //
798 //
799 //
800 //
801 //
802 //
803 //
804 //
805 //
806 //
807 //
808 //
809 //
810 //
811 //
812 //
813 //
814 //
815 //
816 //
817 //
818 //
819 //
820 //
821 //
822 //
823 //
824 //
825 //
826 //
827 //
828 //
829 //
830 //
831 //
832 //
833 //
834 //
835 //
836 //
837 //
838 //
839 //
840 //
841 //
842 //
843 //
844 //
845 //
846 //
847 //
848 //
849 //
850 //
851 //
852 //
853 //
854 //
855 //
856 //
857 //
858 //
859 //
860 //
861 //
862 //
863 //
864 //
865 //
866 //
867 //
868 //
869 //
870 //
871 //
872 //
873 //
874 //
875 //
876 //
877 //
878 //
879 //
880 //
881 //
882 //
883 //
884 //
885 //
886 //
887 //
888 //
889 //
890 //
891 //
892 //
893 //
894 //
895 //
896 //
897 //
898 //
899 //
900 //
901 //
902 //
903 //
904 //
905 //
906 //
907 //
908 //
909 //
910 //
911 //
912 //
913 //
914 //
915 //
916 //
917 //
918 //
919 //
920 //
921 //
922 //
923 //
924 //
925 //
926 //
927 //
928 //
929 //
930 //
931 //
932 //
933 //
934 //
935 //
936 //
937 //
938 //
939 //
940 //
941 //
942 //
943 //
944 //
945 //
946 //
947 //
948 //
949 //
950 //
951 //
952 //
953 //
954 //
955 //
956 //
957 //
958 //
959 //
960 //
961 //
962 //
963 //
964 //
965 //
966 //
967 //
968 //
969 //
970 //
971 //
972 //
973 //
974 //
975 //
976 //
977 //
978 //
979 //
980 //
981 //
982 //
983 //
984 //
985 //
986 //
987 //
988 //
989 //
990 //
991 //
992 //
993 //
994 //
995 //
996 //
997 //
998 //
999 //
1000 //
1001 //
1002 //
1003 //
1004 //
1005 //
1006 //
1007 //
1008 //
1009 //
1010 //
1011 //
1012 //
1013 //
1014 //
1015 //
1016 //
1017 //
1018 //
1019 //
1020 //
1021 //
1022 //
1023 //
1024 //
1025 //
1026 //
1027 //
1028 //
1029 //
1030 //
1031 //
1032 //
1033 //
1034 //
1035 //
1036 //
1037 //
1038 //
1039 //
1040 //
1041 //
1042 //
1043 //
1044 //
1045 //
1046 //
1047 //
1048 //
1049 //
1050 //
1051 //
1052 //
1053 //
1054 //
1055 //
1056 //
1057 //
1058 //
1059 //
1060 //
1061 //
1062 //
1063 //
1064 //
1065 //
1066 //
1067 //
1068 //
1069 //
1070 //
1071 //
1072 //
1073 //
1074 //
1075 //
1076 //
1077 //
1078 //
1079 //
1080 //
1081 //
1082 //
1083 //
1084 //
1085 //
1086 //
1087 //
1088 //
1089 //
1090 //
1091 //
1092 //
1093 //
1094 //
1095 //
1096 //
1097 //
1098 //
1099 //
1100 //
1101 //
1102 //
1103 //
1104 //
1105 //
1106 //
1107 //
1108 //
1109 //
1110 //
1111 //
1112 //
1113 //
1114 //
1115 //
1116 //
1117 //
1118 //
1119 //
1120 //
1121 //
1122 //
1123 //
1124 //
1125 //
1126 //
1127 //
1128 //
1129 //
1130 //
1131 //
1132 //
1133 //
1134 //
1135 //
1136 //
1137 //
1138 //
1139 //
1140 //
1141 //
1142 //
1143 //
1144 //
1145 //
1146 //
1147 //
1148 //
1149 //
1150 //
1151 //
1152 //
1153 //
1154 //
1155 //
1156 //
1157 //
1158 //
1159 //
1160 //
1161 //
1162 //
1163 //
1164 //
1165 //
1166 //
1167 //
1168 //
1169 //
1170 //
1171 //
1172 //
1173 //
1174 //
1175 //
1176 //
1177 //
1178 //
1179 //
1180 //
1181 //
1182 //
1183 //
1184 //
1185 //
1186 //
1187 //
1188 //
1189 //
1190 //
1191 //
1192 //
1193 //
1194 //
1195 //
1196 //
1197 //
1198 //
1199 //
1200 //
1201 //
1202 //
1203 //
1204 //
1205 //
1206 //
1207 //
1208 //
1209 //
1210 //
1211 //
1212 //
1213 //
1214 //
1215 //
1216 //
1217 //
1218 //
1219 //
1220 //
1221 //
1222 //
1223 //
1224 //
1225 //
1226 //
1227 //
1228 //
1229 //
1230 //
1231 //
1232 //
1233 //
1234 //
1235 //
1236 //
1237 //
1238 //
1239 //
1240 //
1241 //
1242 //
1243 //
1244 //
1245 //
1246 //
1247 //
1248 //
1249 //
1250 //
1251 //
1252 //
1253 //
1254 //
1255 //
1256 //
1257 //
1258 //
1259 //
1260 //
1261 //
1262 //
1263 //
1264 //
1265 //
1266 //
1267 //
1268 //
1269 //
1270 //
1271 //
1272 //
1273 //
1274 //
1275 //
1276 //
1277 //
1278 //
1279 //
1280 //
1281 //
1282 //
1283 //
1284 //
1285 //
1286 //
1287 //
1288 //
1289 //
1290 //
1291 //
1292 //
1293 //
1294 //
1295 //
1296 //
1297 //
1298 //
1299 //
1300 //
1301 //
1302 //
1303 //
1304 //
1305 //
1306 //
1307 //
1308 //
1309 //
1310 //
1311 //
1312 //
1313 //
1314 //
1315 //
1316 //
1317 //
1318 //
1319 //
1320 //
1321 //
1322 //
1323 //
1324 //
1325 //
1326 //
1327 //
1328 //
1329 //
1330 //
1331 //
1332 //
1333 //
1334 //
1335 //
1336 //
1337 //
1338 //
1339 //
1340 //
1341 //
1342 //
1343 //
1344 //
1345 //
1346 //
1347 //
1348 //
1349 //
1350 //
1351 //
1352 //
1353 //
1354 //
1355 //
1356 //
1357 //
1358 //
1359 //
1360 //
1361 //
1362 //
1363 //
1364 //
1365 //
1366 //
1367 //
1368 //
1369 //
1370 //
1371 //
1372 //
1373 //
1374 //
1375 //
1376 //
1377 //
1378 //
1379 //
1380 //
1381 //
1382 //
1383 //
1384 //
1385 //
1386 //
1387 //
1388 //
1389 //
1390 //
1391 //
1392 //
1393 //
1394 //
1395 //
1396 //
1397 //
1398 //
1399 //
1400 //
1401 //
1402 //
1403 //
1404 //
1405 //
1406 //
1407 //
1408 //
1409 //
1410 //
1411 //
1412 //
1413 //
1414 //
1415 //
1416 //
1417 //
1418 //
1419 //
1420 //
1421 //
1422 //
1423 //
1424 //
1425 //
1426 //
1427 //
1428 //
1429 //
1430 //
1431 //
1432 //
1433 //
1434 //
1435 //
1436 //
1437 //
1438 //
1439 //
1440 //
1441 //
1442 //
1443 //
1444 //
1445 //
1446 //
1447 //
1448 //
1449 //
1450 //
1451 //
1452 //
1453 //
1454 //
1455 //
1456 //
1457 //
1458 //
1459 //
1460 //
1461 //
1462 //
1463 //
1464 //
1465 //
1466 //
1467 //
1468 //
1469 //
1470 //
1471 //
1472 //
1473 //
1474 //
1475 //
1476 //
1477 //
1478 //
1479 //
1480 //
1481 //
1482 //
1483 //
1484 //
1485 //
1486 //
1487 //
1488 //
1489 //
1490 //
1491 //
1492 //
1493 //
1494 //
1495 //
1496 //
1497 //
1498 //
1499 //
1500 //
1501 //
1502 //
1503 //
1504 //
1505 //
1506 //
1507 //
1508 //
1509 //
1510 //
1511 //
1512 //
1513 //
1514 //
1515 //
1516 //
1517 //
1518 //
1519 //
1520 //
1521 //
1522 //
1523 //
1524 //
1525 //
1526 //
1527 //
1528 //
1529 //
1530 //
1531 //
1532 //
1533 //
1534 //
1535 //
1536 //
1537 //
1538 //
1539 //
1540 //
1541 //
1542 //
1543 //
1544 //
1545 //
1546 //
1547 //
1548 //
1549 //
1550 //
1551 //
1552 //
1553 //
1554 //
1555 //
1556 //
1557 //
1558 //
1559 //
1560 //
1561 //
1562 //
1563 //
1564 //
1565 //
1566 //
1567 //
1568 //
1569 //
1570 //
1571 //
1572 //
1573 //
1574 //
1575 //
1576 //
1577 //
1578 //
1579 //
1580 //
1581 //
1582 //
1583 //
1584 //
1585 //
1586 //
1587 //
1588 //
1589 //
1590 //
1591 //
1592 //
1593 //
1594 //
1595 //
1596 //
1597 //
1598 //
1599 //
1600 //
1601 //
1602 //
1603 //
1604 //
1605 //
1606 //
1607 //
1608 //
1609 //
1610 //
1611 //
1612 //
1613 //
1614 //
1615 //
1616 //
1617 //
1618 //
1619 //
1620 //
1621 //
1622 //
1623 //
1624 //
1625 //
1626 //
1627 //
1628 //
1629 //
1630 //
1631 //
1632 //
1633 //
1634 //
1635 //
1636 //
1637 //
1638 //
1639 //
1640 //
1641 //
1642 //
1643 //
1644 //
1645 //
1646 //
1647 //
1648 //
1649 //
1650 //
1651 //
1652 //
1653 //
1654 //
1655 //
1656 //
1657 //
1658 //
1659 //
1660 //
1661 //
1662 //
1663 //
1664 //
1665 //
1666 //
1667 //
1668 //
1669 //
1670 //
1671 //
1672 //
1673 //
1674 //
1675 //
1676 //
1677 //
1678 //
1679 //
1680 //
1681 //
1682 //
1683 //
1684 //
1685 //
1686 //
1687 //
1688 //
1689 //
1690 //
1691 //
1692 //
1693 //
1694 //
1695 //
1696 //
1697 //
1698 //
1699 //
1700 //
1701 //
1702 //
1703 //
1704 //
1705 //
1706 //
1707 //
1708 //
1709 //
1710 //
1711 //
1712 //
1713 //
1714 //
1715 //
1716 //
1717 //
1718 //
1719 //
1720 //
1721 //
1722 //
1723 //
1724 //
1725 //
1726 //
1727 //
1728 //
1729 //
1730 //
1731 //
1732 //
1733 //
1734 //
1735 //
1736 //
1737 //
1738 //
1739 //
1740 //
1741 //
1742 //
1743 //
1744 //
1745 //
1746 //
1747 //
1748 //
1749 //
1750 //
1751 //
1752 //
1753 //
1754 //
1755 //
1756 //
1757 //
1758 //
1759 //
1760 //
1761 //
1762 //
1763 //
1764 //
1765 //
1766 //
1767 //
1768 //
1769 //
1770 //
1771 //
1772 //
1773 //
1774 //
1775 //
1776 //
1777 //
1778 //
1779 //
1780 //
1781 //
1782 //
1783 //
1784 //
1785 //
1786 //
1787 //
1788 //
1789 //
1790 //
1791 //
1792 //
1793 //
1794 //
1795 //
1796 //
1797 //
1798 //
1799 //
1800 //
1801 //
1802 //
1803 //
1804 //
1805 //
1806 //
1807 //
1808 //
1809 //
1810 //
1811 //
1812 //
1813 //
1814 //
1815 //
1816 //
1817 //
1818 //
1819 //
1820 //
1821 //
1822 //
1823 //
1824 //
1825 //
1826 //
1827 //
1828 //
1829 //
1830 //
1831 //
1832 //
1833 //
1834 //
1835 //
1836 //
1837 //
1838 //
1839 //
1840 //
1841 //
1842 //
1843 //
1844 //
1845 //
1846 //
1847 //
1848 //
1849 //
1850 //
1851 //
1852 //
1853 //
1854 //
1855 //
1856 //
1857 //
1858 //
1859 //
1860 //
1861 //
1862 //
1863 //
1864 //
1865 //
1866 //
1867 //
1868 //
1869 //
1870 //
1871 //
1872 //
1873 //
1874 //
1875 //
1876 //
1877 //
1878 //
1879 //
1880 //
1881 //
1882 //
1883 //
1884 //
1885 //
1886 //
1887 //
1888 //
1889 //
1890 //
1891 //
1892 //
1893 //
1894 //
1895 //
1896 //
1897 //
1898 //
1899 //
1900 //
1901 //
1902 //
1903 //
1904 //
1905 //
1906 //
1907 //
1908 //
1909 //
1910 //
1911 //
1912 //
1913 //
1914 //
1915 //
1916 //
1917 //
1918 //
1919 //
1920 //
1921 //
1922 //
1923 //
1924 //
1925 //
1926 //
1927 //
1928 //
1929 //
1930 //
1931 //
1932 //
1933 //
1934 //
1935 //
1936 //
1937 //
1938 //
1939 //
1940 //
1941 //
1942 //
1943 //
1944 //
1945 //
1946 //
1947 //
1948 //
1949 //
1950 //
1951 //
1952 //
1953 //
1954 //
1955 //
1956 //
1957 //
1958 //
1959 //
1960 //
1961 //
1962 //
1963 //
1964 //
1965 //
1966 //
1967 //
1968 //
1969 //
1970 //
1971 //
1972 //
1973 //
1974 //
1975 //
1976 //
1977 //
1978 //
1979 //
1980 //
1981 //
1982 //
1983 //
1984 //
1985 //
1986 //
1987 //
1988 //
1989 //
1990 //
1991 //
1992 //
1993 //
1994 //
1995 //
1996 //
1997 //
1998 //
1999 //
2000 //
2001 //
2002 //
2003 //
2004 //
2005 //
2006 //
2007 //
2008 //
2009 //
2010 //
2011 //
2012 //
2013 //
2014 //
2015 //
2016 //
2017 //
2018 //
2019 //
2020 //
2021 //
2022 //
2023 //
2024 //
2025 //
2026 //
2027 //
2028 //
2029 //
2030 //
2031 //
2032 //
2033 //
2034 //
2035 //
2036 //
2037 //
2038 //
2039 //
2040 //
2041 //
2042 //
2043 //
2044 //
2045 //
2046 //
2047 //
2048 //
2049 //
2050 //
2051 //
2052 //
2053 //
2054 //
2055 //
2056 //
2057 //
2058 //
2059 //
2060 //
2061 //
2062 //
2063 //
2
```

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

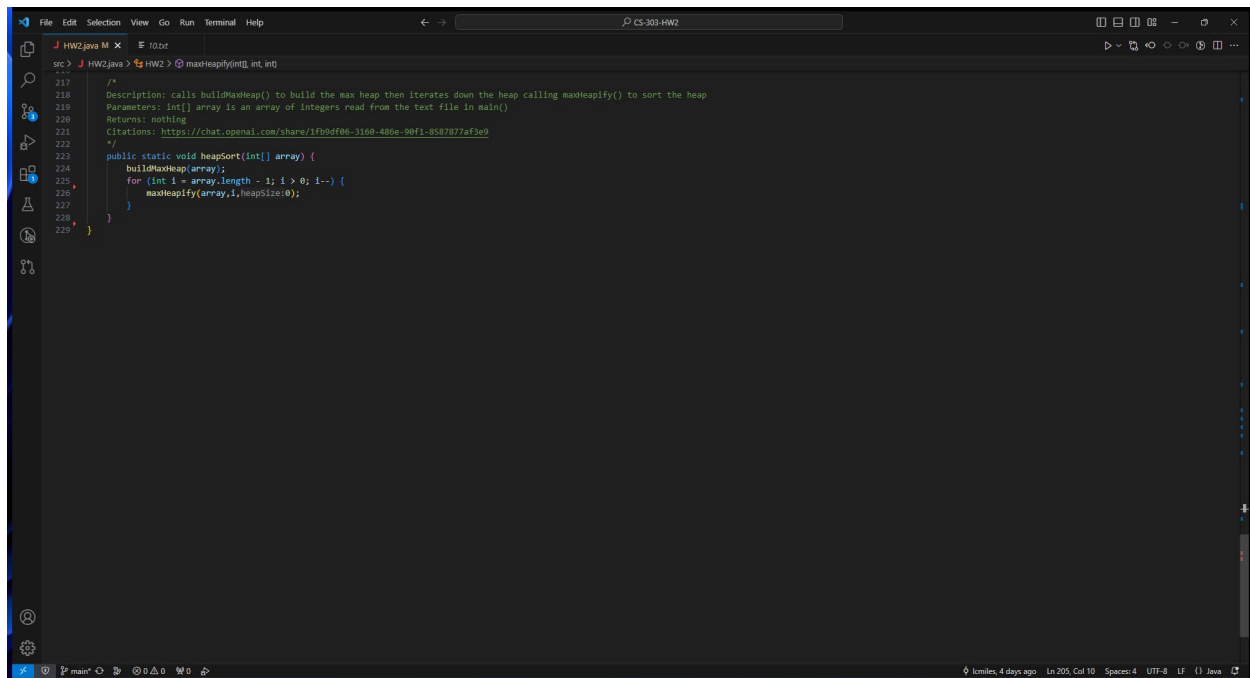
J HW2.java M X E 10.txt
src > J HW2.java > HW2 > @ main(String[])

139 /*
140 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
141 were divided from until the whole array is sorted
142 Parameters:
143 int[] array is an array of integers read from the text file in main()
144 int[] tempArray is a temporary copy of array for iteration purposes
145 int p is a pointer for starting index of left subarray
146 int r is a pointer for ending index of right subarray
147 int q is a pointer for middle index of the original array
148 Returns: nothing
149 Citations: https://chat.openai.com/share/c8f53e73-fb52-483b-846c-c756faa3bc6e
150 */
151 public static void merge(int[] array, int[] tempArray, int p, int q, int r) {
152     int i = p; //initializing pointer to starting index of left subarray
153     int j = q + 1; //initializing pointer to starting index of right subarray
154     for (int k = p; k <= r; k++) { //copy all values in array to tempArray
155         tempArray[k] = array[k];
156     }
157     for (int k = p; k <= r; k++) {
158         if (i > q) { //if left subarray is empty, copy the element from the right subarray to the main array
159             array[k] = tempArray[j];
160             j++;
161         }
162         else if (j > r) { //if right subarray is empty, copy the element the left subarray to the main array
163             array[k] = tempArray[i];
164             i++;
165         }
166         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
167             array[k] = tempArray[j];
168             j++;
169         }
170         else { //else copy the element from the left subarray to the main array
171             array[k] = tempArray[i];
172             i++;
173         }
174     }
175 }
176
177 /*
178 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
179 Parameters: int[] array is an array of integers read from the text file in main()
180 Returns: nothing
181 Citations: https://chat.openai.com/share/1fb9df68-3168-486e-98f1-8587877af3e9
182 */
183 public static void buildMaxHeap(int[] array) {
184     int heapSize = array.length;
185     for (int i = heapSize / 2 - 1; i >= 0; i--) { //initializes i to the last non-leaf node
186         maxHeapify(array, heapSize, i);
187     }
188 }
189
190 /*
191 Description: maxHeapify() maintains the max heap properties during each recursion of heapSort()
192 Parameters:
193 int[] array is an array of integers read from the text file in main()
194 int i is the parent iterated over in buildMaxHeap()
195 int heapSize is the number of elements in the heap
196 Returns: nothing
197 Citations: https://chat.openai.com/share/1fb9df68-3168-486e-98f1-8587877af3e9
198 */
199 public static void maxHeapify(int[] array, int i, int heapSize) {
200     int largest = i; //index of the parent
201     int left = 2 * i + 1; //index of the left leaf
202     int right = 2 * i + 2; //index of the right leaf
203     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
204         largest = left; //set left leaf to parent
205     }
206     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
207         largest = right; //set right leaf to parent
208     }
209     if (largest != i) { //if the largest element is not the root of the maxheap, swap then and recursively heapify the affected sub-tree
210         int temp = array[i];
211         array[i] = array[largest];
212         array[largest] = temp;
213         maxHeapify(array, heapSize, largest);
214     }
215 }
216
217 /*
218 Description: call buildMaxHeap() to build the max heap then iterates down the heap calling maxHeapify() to sort the heap
219 Parameters: int[] array is an array of integers read from the text file in main()
220 Returns: nothing
221 Citations: https://chat.openai.com/share/1fb9df68-3168-486e-98f1-8587877af3e9
222 */
223 public static void heapSort(int[] array) {
224     buildMaxHeap(array);
225     for (int i = array.length - 1; i > 0; i--) {
226         swap(array, 0, i);
227         maxHeapify(array, 0, i);
228     }
229 }
```

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

J HW2.java M X E 10.txt
src > J HW2.java > HW2 > @ main(String[])

177 /*
178 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
179 Parameters: int[] array is an array of integers read from the text file in main()
180 Returns: nothing
181 Citations: https://chat.openai.com/share/1fb9df68-3168-486e-98f1-8587877af3e9
182 */
183 public static void buildMaxHeap(int[] array) {
184     int heapSize = array.length;
185     for (int i = heapSize / 2 - 1; i >= 0; i--) { //initializes i to the last non-leaf node
186         maxHeapify(array, heapSize, i);
187     }
188 }
189
190 /*
191 Description: maxHeapify() maintains the max heap properties during each recursion of heapSort()
192 Parameters:
193 int[] array is an array of integers read from the text file in main()
194 int i is the parent iterated over in buildMaxHeap()
195 int heapSize is the number of elements in the heap
196 Returns: nothing
197 Citations: https://chat.openai.com/share/1fb9df68-3168-486e-98f1-8587877af3e9
198 */
199 public static void maxHeapify(int[] array, int i, int heapSize) {
200     int largest = i; //index of the parent
201     int left = 2 * i + 1; //index of the left leaf
202     int right = 2 * i + 2; //index of the right leaf
203     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
204         largest = left; //set left leaf to parent
205     }
206     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
207         largest = right; //set right leaf to parent
208     }
209     if (largest != i) { //if the largest element is not the root of the maxheap, swap then and recursively heapify the affected sub-tree
210         int temp = array[i];
211         array[i] = array[largest];
212         array[largest] = temp;
213         maxHeapify(array, heapSize, largest);
214     }
215 }
216
217 /*
218 Description: call buildMaxHeap() to build the max heap then iterates down the heap calling maxHeapify() to sort the heap
219 Parameters: int[] array is an array of integers read from the text file in main()
220 Returns: nothing
221 Citations: https://chat.openai.com/share/1fb9df68-3168-486e-98f1-8587877af3e9
222 */
223 public static void heapSort(int[] array) {
224     buildMaxHeap(array);
225     for (int i = array.length - 1; i > 0; i--) {
226         swap(array, 0, i);
227         maxHeapify(array, 0, i);
228     }
229 }
```



```
File Edit Selection View Go Run Terminal Help
src > HW2.java M X F 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/1fb9df06-3168-480e-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.



The image shows a screenshot of the Visual Studio Code editor interface. The top menu bar includes File, Edit, Selection, View, Go, Run, and Terminal. The Explorer sidebar on the left shows the project structure for 'CS-303-HW2', with folders for 'HW2', 'bin', 'lib', 'src', and 'hw2'. The main editor area displays the file 'hw2.cpp', which contains a large number of lines of C++ code. The code includes a main function and a large array of numbers. The bottom status bar shows the file path 'PS C:\Users\lgan\OneDrive\Local\Temp\cs\_dnd\cs303\cs303\hw2.cpp' and the file encoding 'UTF-8'. The status bar also shows 'You, 1 second ago', 'Ln 91, Col 34', 'Spaces: 4', and 'UTF-8'.

Example output of linear search algorithm.

The image shows a Windows File Explorer window. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, and Help. The address bar shows the path: C:\Users\lgan\OneDrive\UAB Files\Fall 2023\CS-363 HW2\CS-363 HW2. The left sidebar has a tree view with 'EXPLORER', 'OPEN EDITORS', 'CS-363 HW2', and 'HW2 Java'. The main pane displays a list of files and folders, including 'CS-363 HW2', 'HW2 Java src', and 'HW2 Java'. The right pane shows the contents of the 'HW2 Java' folder, which includes a large text file named 'HW2.java'. The status bar at the bottom indicates the current location is 'C:\Users\lgan\OneDrive\UAB Files\Fall 2023\CS-363 HW2\CS-363 HW2' and shows various icons for file operations and sharing.

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

EXPLORER
  CS-303-HW2
    HW2.java src
    bin
    src
    HW2.java
    .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

TERMINAL
  CS-303-HW2
    Insertion Sort Time:3039000 nanoseconds, 3.0396 milliseconds, or 0.0030396 seconds
    Merge Sort Time:197100 nanoseconds, 0.1971 milliseconds, or 0.19714 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 - The University of Alabama at Birmingham\UAB Files\FA 2023\CS-303\HW\CS-303-HW2>

3,3,4,4,6,6,11,12,14,15,16,18,18,19,20,20,22,22,26,28,29,29,30,31,32,13,36,36,37,37,38,39,40,40,41,42,42,45,45,48,49,49,50,50,50,51,51,52,53,55,57,58,58,59,61,61,64,65,66,67,70,70,70,71,72,74,75,76,76,79,81,82,84,84,85,86,86,87,88,88,89,91,92,92,92,93,95,95,96,96,96,97,98,99,101,101,102,102,103,103,104,104,104,105,106,106,106,107,110,112,113,114,118,119,122,123,125,127,127,128,128,129,129,130,131,132,134,134,138,138,139,140,141,141,142,143,145,145,147,148,148,148,151,151,152,152,154,155,159,160,162,162,163,165,166,167,168,168,170,173,173,175,178,178,178,179,180,181,182,186,187,187,188,189,194,197,197,198,202,203,205,207,207,208,208,208,210,210,211,213,214,215,216,217,218,219,219,220,220,221,221,222,225,227,229,231,231,232,232,233,235,235,237,237,237,238,239,240,242,243,243,244,244,247,247,247,248,248,248,249,250,255,255,256,257,258,259,260,260,260,264,265,266,267,267,268,270,270,273,274,274,274,275,275,277,280,281,281,284,284,286,286,288,288,289,291,291,293,294,294,294,294,295,295,296,297,298,298,299,300,303,304,305,305,312,312,312,313,313,314,314,315,315,315,317,319,320,321,321,322,323,324,326,327,330,330,332,335,338,340,341,343,345,345,346,346,347,348,348,349,350,350,352,353,354,357,359,359,363,363,65,366,367,368,369,369,369,373,375,375,376,376,377,378,378,380,380,380,382,383,383,384,384,385,385,386,387,388,388,388,388,392,392,393,393,394,394,398,400,402,403,403,407,407,408,409,410,410,411,411,412,41,416,416,417,417,419,419,420,420,421,423,424,425,428,429,430,430,431,431,432,432,432,432,434,436,436,437,437,438,438,440,442,442,442,444,446,446,447,448,452,452,453,454,454,455,458,459,459,459,461,462,465,466,467,469,469,470,470,471,472,472,474,475,475,477,478,479,483,485,485,486,486,489,489,491,492,494,494,495,496,496,499,499,500,500,500,502,502,505,506,509,510,514,514,516,516,517,518,518,519,521,523,524,524,524,524,525,525,525,527,528,528,529,530,530,531,532,533,533,534,534,535,536,536,537,538,540,541,541,541,545,545,546,547,547,548,548,548,550,551,551,551,553,555,555,556,558,559,561,561,562,562,563,63,63,63,564,564,565,565,566,567,568,569,570,570,572,573,573,575,575,576,576,577,578,579,581,583,585,585,587,587,588,589,589,590,590,592,594,594,595,595,596,596,600,601,601,601,601,601,604,604,606,606,614,61,7,617,618,619,620,621,621,622,624,625,626,627,628,628,631,634,635,635,636,640,641,641,642,642,643,647,650,651,651,651,651,655,655,656,656,657,658,659,660,660,661,662,662,664,664,666,666,666,667,670,670,670,671,671,676,676,677,679,680,680,682,682,683,683,684,686,686,689,691,693,694,695,697,698,699,702,703,703,704,705,706,707,707,707,707,708,708,709,710,711,712,713,714,715,716,717,717,718,718,720,722,723,725,726,726,727,729,729,729,730,731,732,732,733,734,736,736,737,737,738,740,740,740,740,741,742,744,744,746,746,748,748,750,751,752,754,755,755,756,756,757,758,760,766,766,766,767,767,771,7,73,774,775,776,776,777,777,778,779,779,780,782,786,786,786,789,791,791,793,794,795,796,796,798,799,800,800,801,802,803,804,804,805,807,808,809,810,811,812,812,813,813,814,815,815,817,817,819,819,820,82,8,823,823,824,824,825,826,826,827,829,830,832,833,833,834,835,835,836,838,839,839,840,842,842,843,843,844,845,845,846,846,847,847,848,848,849,851,853,857,857,858,859,859,860,861,863,863,863,863,865,866,866,868,869,869,871,872,873,874,874,875,875,875,876,877,878,879,879,881,883,883,884,884,885,885,886,887,888,889,891,891,893,894,897,898,898,899,900,901,902,905,908,910,913,913,914,916,917,917,918,919,920,922,924,924,925,928,930,931,932,933,933,934,938,939,941,942,943,944,944,944,944,946,947,948,948,948,950,955,955,957,957,958,964,965,965,965,967,967,967,969,969,970,970,970,971,972,974,975,975,978,9,79,79,800,800,800,800,800,802,806,807,808,892,997,998,1000,1000
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 - The University of Alabama at Birmingham\UAB Files\FA 2023\CS-303\HW\CS-303-HW2>
```

Example output of modified merge sort algorithm.

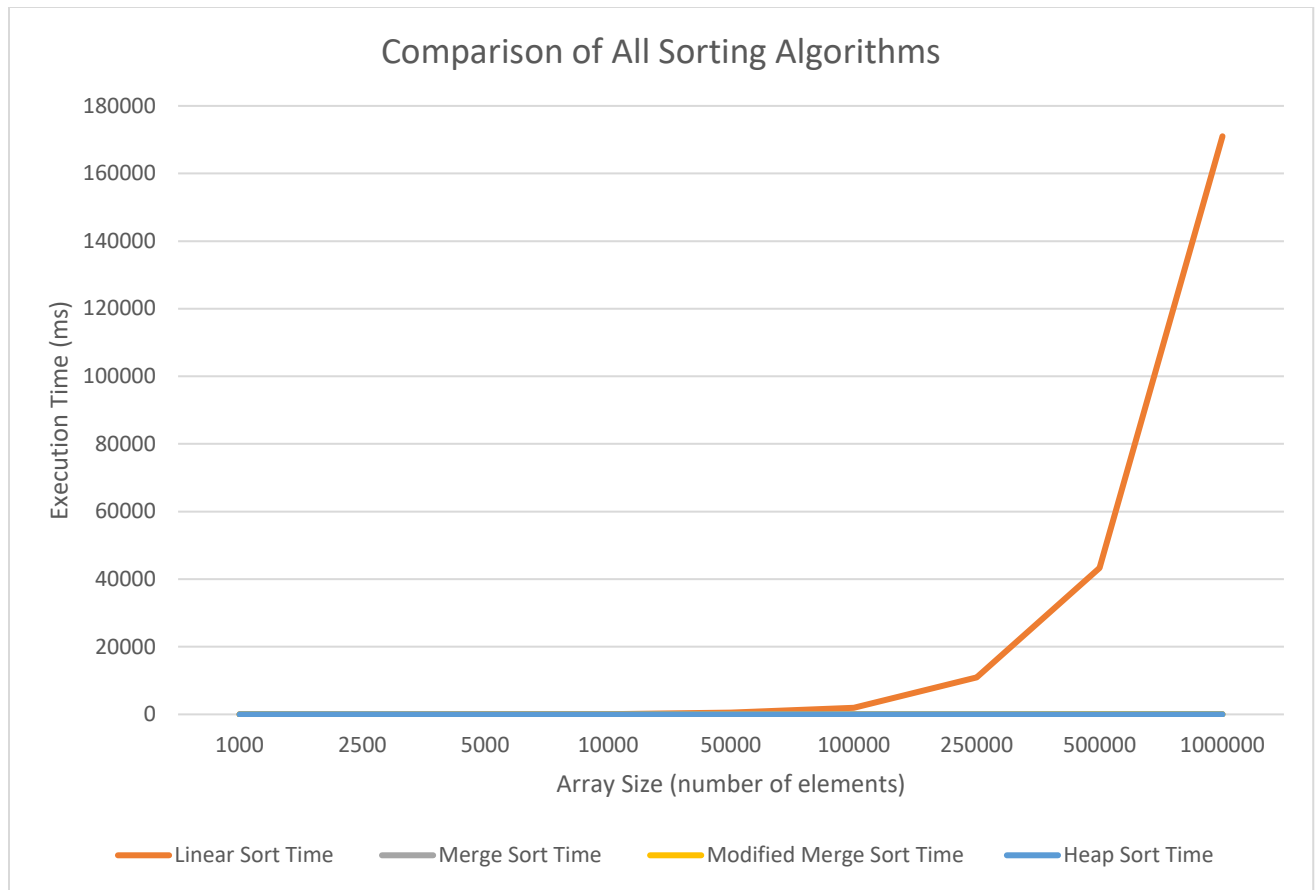
```
File Edit Selection View Go Run Terminal Help
CS-303-HW2

EXPLORER
  CS-303-HW2
    HW2.java src
    bin
    src
    HW2.java
    .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

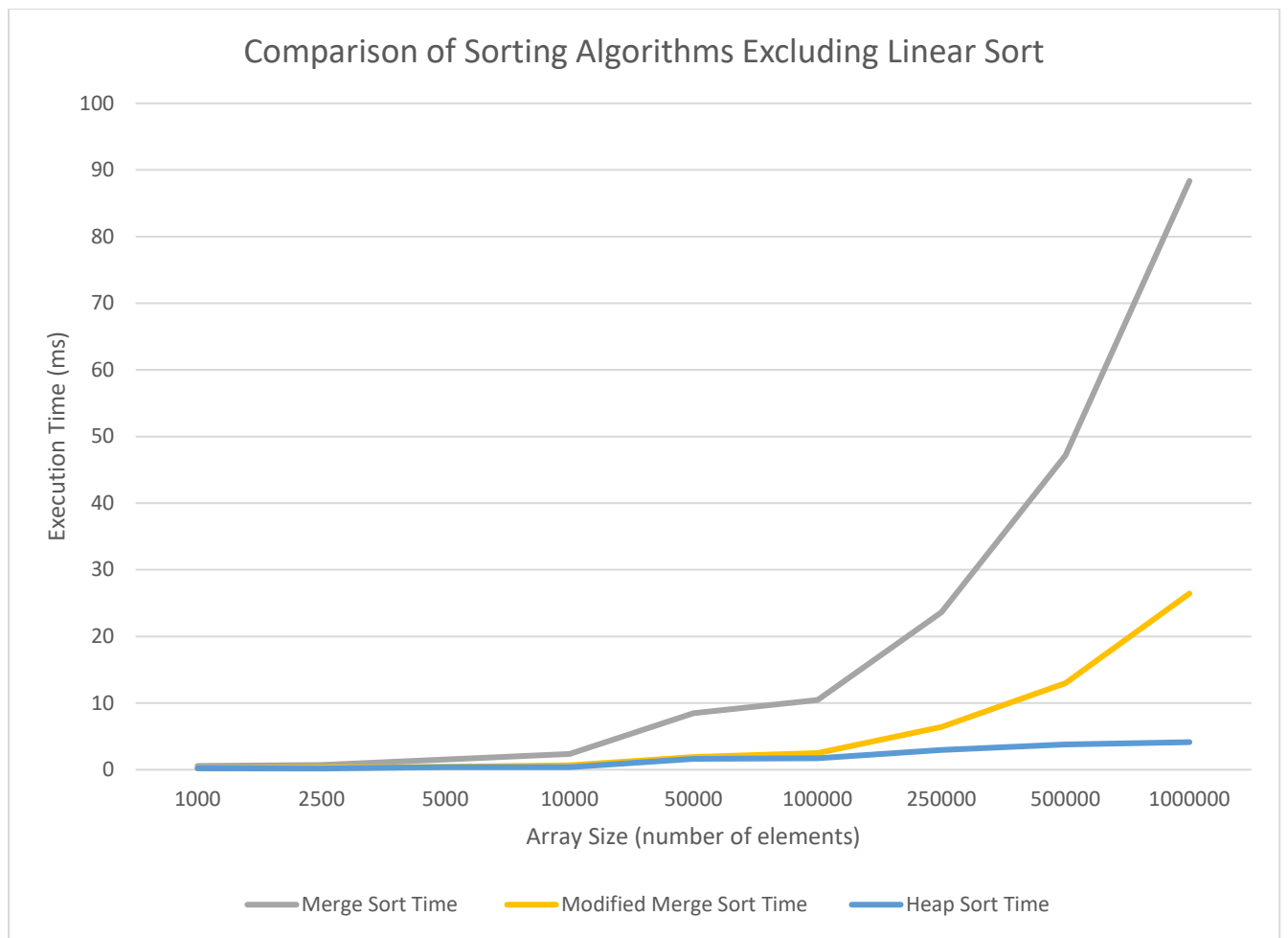
TERMINAL
  CS-303-HW2
    Insertion Sort Time:3117200 nanoseconds, 3.1172 milliseconds, or 0.0031172 seconds
    Merge Sort Time:436000 nanoseconds, 0.436 milliseconds, or 4.36E-4 seconds
    Modified Merge Sort Time:307000 nanoseconds, 0.307 milliseconds, or 0.307E-3 seconds
    Heap Sort Time:133000 nanoseconds, 0.133 milliseconds, or 1.33E-4 seconds
    PS C:\Users\logan\OneDrive - The University of Alabama at Birmingham\UAB Files\FA 2023\CS-303\HW\CS-303-HW2>

3,3,4,4,6,6,11,12,14,15,16,18,18,19,19,20,20,22,22,26,28,29,29,30,31,32,13,36,36,37,37,38,39,40,40,41,42,42,45,45,48,49,49,50,50,50,51,51,52,53,55,57,58,58,59,61,61,64,65,66,67,70,70,70,71,72,74,75,76,76,79,81,82,84,84,85,86,86,87,88,88,89,91,92,92,92,93,95,95,96,96,96,97,98,99,101,101,102,102,103,103,104,104,104,105,106,106,106,107,110,112,113,114,118,119,122,123,125,127,127,128,128,129,129,130,131,132,134,134,138,138,139,140,141,141,142,143,145,145,147,148,148,148,151,151,152,152,154,155,159,160,162,162,163,165,166,167,168,168,170,173,173,175,178,178,179,180,181,182,186,187,187,188,189,194,197,197,198,202,203,205,207,207,208,208,208,210,210,211,213,214,215,216,217,218,219,219,220,220,221,221,222,225,227,229,231,231,232,232,233,235,235,237,237,237,238,239,239,240,242,243,243,244,244,247,247,247,248,248,248,249,250,255,255,256,257,258,259,260,260,260,264,265,266,267,267,268,270,270,273,274,274,274,275,275,277,280,281,281,284,284,286,286,288,288,289,291,291,293,294,294,294,294,295,295,296,297,298,298,299,300,303,304,305,305,312,312,313,313,314,314,315,315,315,317,319,320,321,321,322,323,324,326,327,330,330,332,333,338,340,341,343,345,346,346,347,348,348,349,350,352,353,354,357,359,359,363,65,366,367,368,369,369,369,373,375,375,376,376,377,378,378,380,380,380,382,383,383,384,384,385,385,386,387,388,388,388,388,392,392,393,393,394,394,398,400,402,403,403,407,407,408,409,410,410,411,411,412,41,416,416,417,417,419,419,420,420,421,423,424,425,428,429,430,430,431,431,432,432,432,432,434,436,436,437,437,438,438,440,442,442,442,444,446,446,447,448,452,452,453,454,454,455,458,459,459,459,461,462,465,466,467,469,469,470,470,471,472,472,474,475,475,477,478,479,483,485,485,486,486,489,489,491,492,494,494,495,496,496,499,499,500,500,500,502,502,505,506,509,510,514,514,516,516,517,518,518,519,520,521,523,524,524,524,525,525,525,527,528,528,529,530,530,531,532,533,533,534,534,535,536,536,537,538,540,541,541,541,545,545,546,547,547,548,548,548,550,551,551,551,553,555,555,556,558,559,561,561,562,562,563,63,63,63,564,564,565,565,566,567,568,569,570,570,572,573,573,575,575,576,576,577,578,579,581,583,585,585,587,587,588,589,589,590,590,592,594,594,595,595,596,596,600,601,601,601,601,601,604,604,606,606,614,61,7,617,618,619,620,621,621,622,624,625,626,627,628,628,631,634,635,635,636,640,641,641,642,642,643,647,650,651,651,651,651,655,655,656,656,657,658,659,660,660,661,662,662,664,664,666,666,666,667,670,670,670,671,671,676,676,677,679,680,680,682,682,683,683,684,686,686,689,691,693,694,695,697,698,699,702,703,703,704,705,706,707,707,707,707,708,708,709,710,711,712,713,714,715,716,717,717,718,718,720,722,723,725,726,726,727,729,729,729,730,731,732,732,733,734,736,736,737,737,738,740,740,740,740,741,742,744,744,746,746,748,748,750,751,752,754,755,755,756,756,757,758,760,766,766,766,767,767,771,7,73,774,775,776,776,777,777,778,779,779,780,782,786,786,786,789,791,791,793,794,795,796,796,798,799,800,800,801,802,803,804,804,805,807,808,809,810,811,812,812,813,813,814,815,815,817,817,819,819,820,82,8,823,823,824,824,825,826,826,827,829,830,832,833,833,834,835,835,836,838,839,839,840,842,842,843,843,844,845,845,846,846,847,847,848,848,849,851,853,857,857,858,859,859,860,861,863,863,863,863,865,866,866,868,869,869,871,872,873,874,874,875,875,875,876,877,878,879,879,881,883,883,884,884,885,885,886,887,888,889,891,891,893,894,897,898,898,899,900,901,902,905,908,910,913,913,914,916,917,917,918,919,920,922,924,924,925,928,930,931,932,933,933,934,938,939,941,942,943,944,944,944,944,946,947,948,948,948,950,955,955,957,957,958,964,965,965,965,967,967,967,969,969,970,970,970,971,972,974,975,975,978,9,79,79,800,800,800,800,800,802,806,807,808,892,997,998,1000,1000
Modified Merge Sort Time:307000 nanoseconds, 0.307 milliseconds, or 0.307E-3 seconds
Heap Sort Time:133000 nanoseconds, 0.133 milliseconds, or 1.33E-4 seconds
PS C:\Users\logan\OneDrive - The University of Alabama at Birmingham\UAB Files\FA 2023\CS-303\HW\CS-303-HW2>
```

Example output of heap sort algorithm.

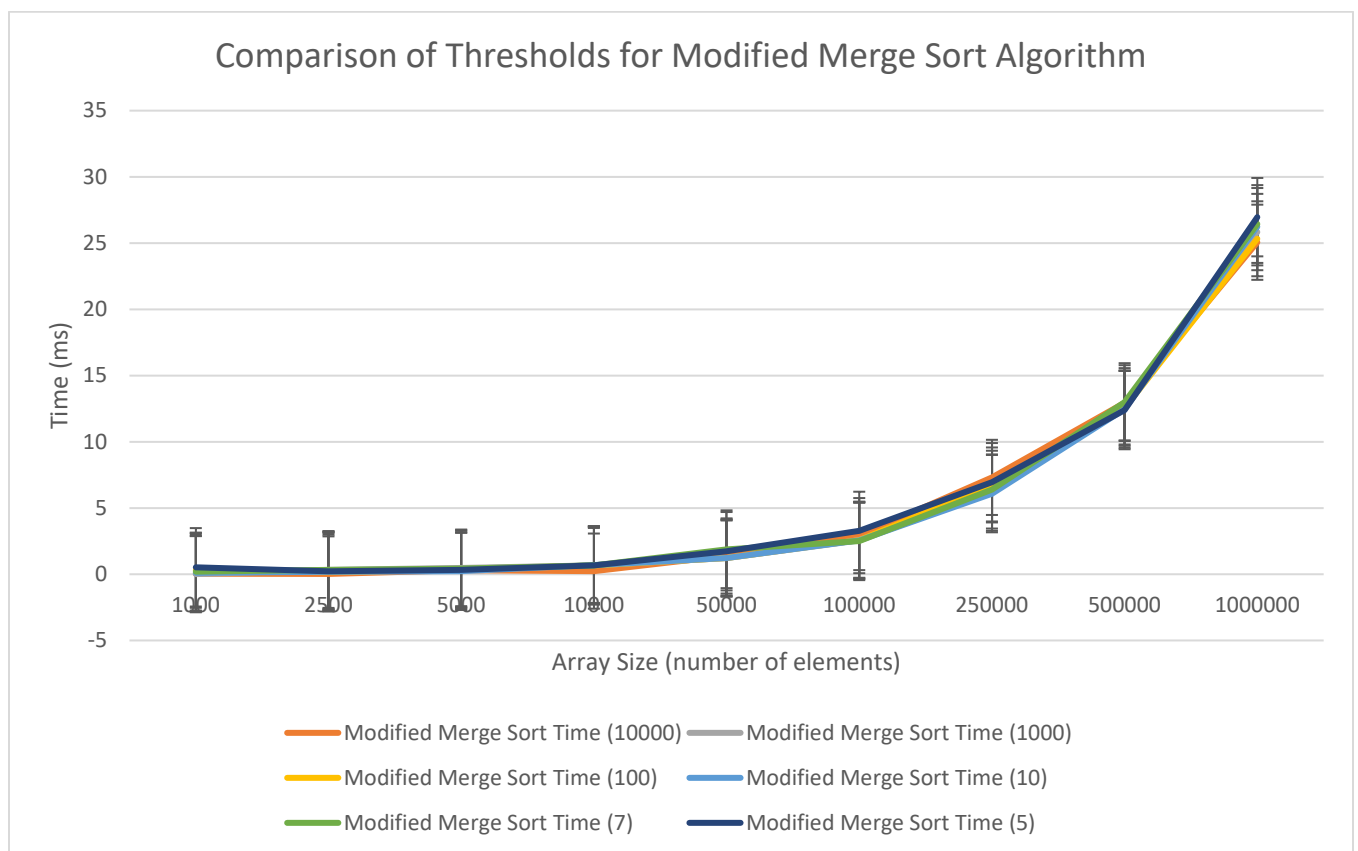


A comparison of all the search algorithms reveals the linear time complexity of  $O(n)$  of linear sort. The graph shows that the execution time is manageable when dealing with smaller arrays, but when the number of elements in the array increases, so too does the execution time of the linear search 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/>