

**Program structures and algorithms  
Spring 2023 (Section-01)**

**BY: PRANAV KAPOOR – NUID: 002998253**

**Assignment 6 (Hits as time predictor)**

In this assignment, your task is to determine--for sorting algorithms--what is the best predictor of total execution time: comparisons, swaps/copies, hits (array accesses), or something else.

You will run the benchmarks for merge sort, (dual-pivot) quick sort, and heap sort. You will sort randomly generated arrays of between 10,000 and 256,000 elements (doubling the size each time). If you use the *SortBenchmark*, as I expect, the number of runs is chosen for you. So, you can ignore the instructions about setting the number of runs.

For each experiment (a sort method of a given size), you will run it twice: once for the instrumentation, once (without instrumentation) for the timing.

Of course, you will be using the *Benchmark* and/or *Timer* classes, as you did in a previous assignment.

You must support your (clearly stated) conclusions with evidence from the benchmarks (you should provide log/log charts and spreadsheets typically).

All of the code to count comparisons, swaps/copies, and hits, is already implemented in the *InstrumentedHelper* class. You can see examples of the usage of this kind of analysis in:

- `src/main/java/edu/neu/coe/info6205/util/SorterBenchmark.java`
- `src/test/java/edu/neu/coe/info6205/sort/linearithmic/MergeSortTest.java`
- `src/test/java/edu/neu/coe/info6205/sort/linearithmic/QuickSortDualPivotTest.java`
- `src/test/java/edu/neu/coe/info6205/sort/elementary/HeapSortTest.java` (you will have to refresh your repository for HeapSort).

The configuration for these benchmarks is determined by the *config.ini* file.

## Solution:

### MergeSort.java

```
1  @Override
2  public X[] sort(X[] xs, boolean makeCopy) {
3      .....getHelper().init(xs.length);
4      .....X[] result = makeCopy ? Arrays.copyOf(xs, xs.length) : xs;
5      .....sort(result, 0, result.length);
6      .....return result;
7  }
8
9  @Override
10 public void sort(X[] a, int from, int to) {
11     .....// CONSIDER don't copy but just allocate according to the xs/aux interchange optimization
12     .....X[] aux = Arrays.copyOf(a, a.length);
13     .....sort(a, aux, from, to);
14 }
15
16 private void sort(X[] a, X[] aux, int from, int to) {
17     .....final Helper<X> helper = getHelper();
18     .....Config config = helper.getConfig();
19     .....boolean insurance = config.getBoolean(MERGESORT, INSURANCE);
20     .....boolean noCopy = config.getBoolean(MERGESORT, NOCOPY);
21     .....if (to <= from + helper.cutoff()) {
22         .....insertionSort.sort(a, from, to);
23         .....return;
24     }
25
26     .....int mid = from + (to - from) / 2;
27
28     .....checkNoCopy(a, aux, from, to, insurance, mid, helper, noCopy);
29 }
30
31 private void checkNoCopy(X[] a, X[] aux, int from, int to, boolean insurance, int mid, final Helper<X> helper, boolean noCopy) {
32     .....if (noCopy) {
33         .....isNoCopy(a, aux, from, to, insurance, mid, helper);
34     }
35     .....else {
36         .....isNotNoCopy(a, aux, from, to, insurance, mid, helper);
37     }
38 }
39 }
```

### HeapSort.java

```
1  package edu.neu.coe.info6205.sort.elementary;
2
3  import edu.neu.coe.info6205.sort.Helper;
4  import edu.neu.coe.info6205.sort.SortWithHelper;
5
6  public class HeapSort<X> extends Comparable<X> extends SortWithHelper<X> {
7
8      .....public HeapSort(Helper<X> helper) {
9          .....super(helper);
10     }
11
12     @Override
13     public void sort(X[] array, int from, int to) {
14         .....if (array == null || array.length <= 1) return;
15
16         .....// XXX construction phase
17         .....buildMaxHeap(array);
18
19         .....// XXX sort-down phase
20         .....Helper<X> helper = getHelper();
21         .....for (int i = array.length - 1; i >= 1; i--) {
22             .....helper.swap(array, 0, i);
23             .....maxHeap(array, i, 0);
24         }
25
26         .....private void buildMaxHeap(X[] array) {
27             .....int half = array.length / 2;
28             .....for (int i = half; i >= 0; i--) maxHeap(array, array.length, i);
29         }
30
31         .....private void maxHeap(X[] array, int heapSize, int index) {
32             .....Helper<X> helper = getHelper();
33             .....final int left = index * 2 + 1;
34             .....final int right = index * 2 + 2;
35             .....int largest = index;
36             .....if (left < heapSize && helper.compare(array, largest, left) < 0) largest = left;
37             .....if (right < heapSize && helper.compare(array, largest, right) < 0) largest = right;
38             .....if (index != largest) {
39                 .....helper.swap(array, index, largest);
40                 .....maxHeap(array, heapSize, largest);
41             }
42         }
43     }
44 }
```

```

40 private static CompletableFuture runHeapSort(int start, int end, Config config, FileWriter fileWriter) {
41     return CompletableFuture.runAsync(
42         () -> {
43             for (int n = start; n <= end; n += 2) {
44                 Helper<Integer> helper = HelperFactory.create("HeapSort", n, config);
45                 HeapSort<Integer> sort = new HeapSort<>(helper);
46                 final int val = n;
47                 Integer[] arr = helper.random(Integer.class, r -> r.nextInt(val));
48                 SorterBenchmark sorterBenchmark = new SorterBenchmark<>(Integer.class,
49                     (Integer[] array) -> {
50                         for (int i = 0; i < array.length; i++) {
51                             array[i] = array[i];
52                         }
53                     },
54                     return array;
55                 );
56                 double time = sort.sort(arr, 1, timeLoggersLinearithmic);
57                 try {
58                     fileWriter.write(createCsvString(n, time, ((InstrumentedHelper) helper).getStatPack(), config.isInstrumented()));
59                     //System.out.println(((InstrumentedHelper) helper).getStatPack());
60                 } catch (Exception e) {
61                     System.out.println("error while writing file Heap" + e);
62                 }
63             }
64             try {
65                 fileWriter.flush();
66             } catch (Exception e) {
67                 System.out.println("error while closing file Heap" + e);
68             }
69         }
70     );
71 }
72
73 private static CompletableFuture runMergeSort(int start, int end, Config config, FileWriter fileWriter) {
74     return runAsync(
75         () -> {
76             for (int n = start; n <= end; n += 2) {
77                 Helper<Integer> helper = HelperFactory.create("MergeSort", n, config);
78                 MergeSort<Integer> sort = new MergeSort<>(helper);
79                 final int val = n;
80                 Integer[] arr = helper.random(Integer.class, r -> r.nextInt(val));
81                 SorterBenchmark sorterBenchmark = new SorterBenchmark<>(Integer.class,
82                     (Integer[] array) -> {
83                         for (int i = 0; i < array.length; i++) {
84                             array[i] = array[i];
85                         }
86                     },
87                     return array;
88                 );
89             }
90         }
91     );
92 }

```

## Main.java

```

1 package edu.neu.coe.info6205.sort.linearithmic;
2 import edu.neu.coe.info6205.sort.Helper;
3
4 public class Main {
5     public static void main(String[] args) {
6         fileOperation();
7     }
8
9     private static void fileOperation() {
10         try {
11             File fHeap = new File("HeapBenchMark.csv");
12             File fMerge = new File("MergeBenchMark.csv");
13             File fQuick = new File("QuickBenchMark.csv");
14
15             createFile(fHeap);
16             createFile(fMerge);
17             createFile(fQuick);
18
19             FileWriter fileWriterHeap = new FileWriter(fHeap);
20             FileWriter fileWriterMerge = new FileWriter(fMerge);
21             FileWriter fileWriterQuick = new FileWriter(fQuick);
22
23             writeFile(fileWriterHeap);
24             writeFile(fileWriterMerge);
25             writeFile(fileWriterQuick);
26
27             runSort(fileWriterHeap, fileWriterMerge, fileWriterQuick);
28         } catch (Exception e) {
29             System.out.println("error while sorting main" + e);
30         }
31     }
32
33     private static void runSort(FileWriter f1, FileWriter f2, FileWriter f3) {
34         boolean instrumentation = true;
35
36         System.out.println("Degree of parallelism: " + ForkJoinPool.getCommonPoolParallelism());
37         Config config = Config.setupConfig("true", "1", "1", "1");
38         Config no_config = Config.setupConfig("false", "1", "1", "1");
39
40         int start = 10000;
41         int end = 256000;
42
43         CompletableFuture<FileWriter> heapSort = runHeapSort(start, end, config, f1);
44         CompletableFuture<FileWriter> quickSort = runQuickSort(start, end, config, f3);
45         CompletableFuture<FileWriter> mergeSort = runMergeSort(start, end, config, f2);
46
47         quickSort.join();
48         heapSort.join();
49         mergeSort.join();
50     }
51 }

```

```


INFO6205 > src/main/java > edu.neu.coe.info6205.sort.linearithmic > main > runMergeSort(int, int, Config, FileWriter) : CompletableFuture
87 ● private static CompletableFuture runHeapSort(int start, int end, Config config, FileWriter fileWriter) {
88     return CompletableFuture.runAsync(
89         () -> {
90             for (int n = start; n <= end; n *= 2) {
91                 Helper<Integer> helper = HelperFactory.create("HeapSort", n, config);
92                 HeapSort<Integer> sort = new HeapSort<>(helper);
93                 final int val = n;
94                 Integer[] arr = helper.random(Integer.class, r -> r.nextInt(val));
95                 SorterBenchmark sorterBenchmark = new SorterBenchmark<>(Integer.class,
96                     (Integer[] array) -> {
97                         for (int i = 0; i < array.length; i++) {
98                             array[i] = array[i];
99                         }
100                     });
101                 return array;
102             }
103             sort, arr, 1, timeLoggersLinearithmic);
104             double time = sorterBenchmark.rund(n);
105             try {
106                 fileWriter.write(createCsvString(n, time, ((InstrumentedHelper) helper).getStatPack(), config.isInstrumented()));
107                 //System.out.println(((InstrumentedHelper) helper).getStatPack());
108             } catch (Exception e) {
109                 System.out.println("error while writing file Heap" + e);
110             }
111         }
112     );
113     fileWriter.flush();
114     fileWriter.close();
115 } catch (Exception e) {
116     System.out.println("error while closing file Heap" + e);
117 }
118 }
119 }
120 }
121 }
122 }
123 }
124 ● private static CompletableFuture runMergeSort(int start, int end, Config config, FileWriter fileWriter) {
125     return runAsync(
126         () -> {
127             for (int n = start; n <= end; n *= 2) {
128                 Helper<Integer> helper = HelperFactory.create("MergeSort", n, config);
129                 MergeSort<Integer> sort = new MergeSort<>(helper);
130                 final int val = n;
131                 Integer[] arr = helper.random(Integer.class, r -> r.nextInt(val));
132                 SorterBenchmark sorterBenchmark = new SorterBenchmark<>(Integer.class,
133                     (Integer[] array) -> {
134                         for (int i = 0; i < array.length; i++) {
135                             array[i] = array[i];
136                         }
137                     });
138                 return array;
139             }
140             sort, arr, 1, timeLoggersLinearithmic);
141             double time = sorterBenchmark.rund(n);
142             try {
143                 if (helper instanceof InstrumentedHelper) {
144                     fileWriter.write(createCsvString(n, time, ((InstrumentedHelper) helper).getStatPack(), config.isInstrumented()));
145                 }
146             } catch (Exception e) {
147                 System.out.println("error while writing file Merge" + e);
148             }
149         }
150     );
151     fileWriter.flush();
152     fileWriter.close();
153 } catch (Exception e) {
154     System.out.println("error while closing file Merge" + e);
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 }






```

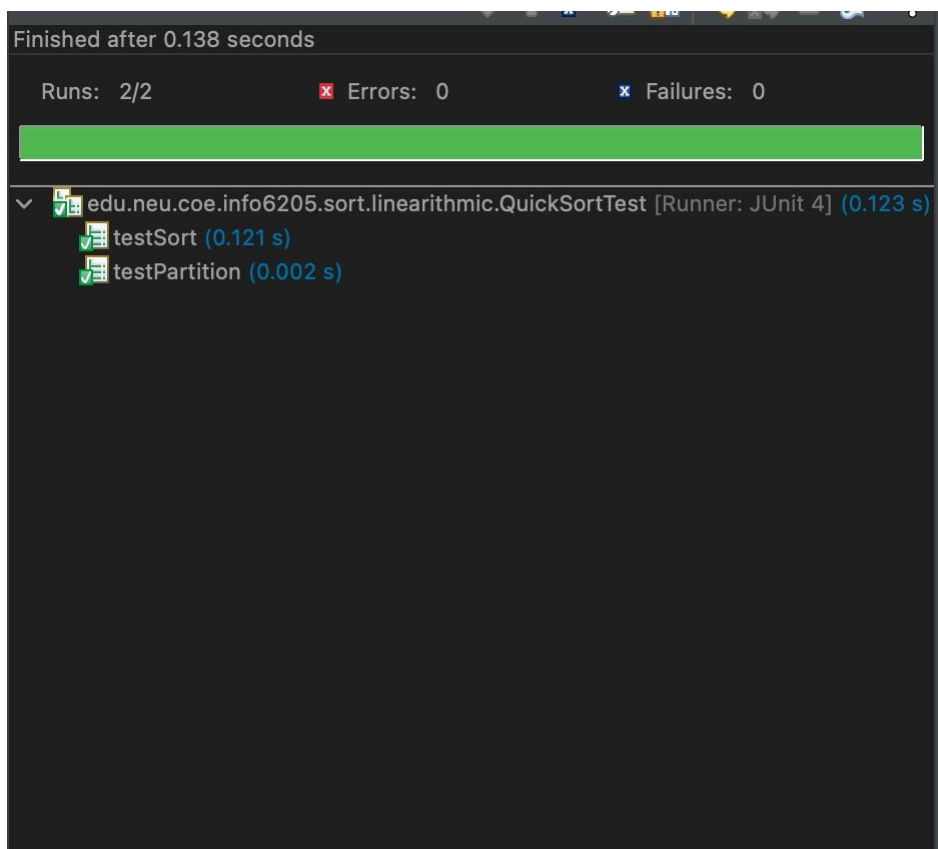
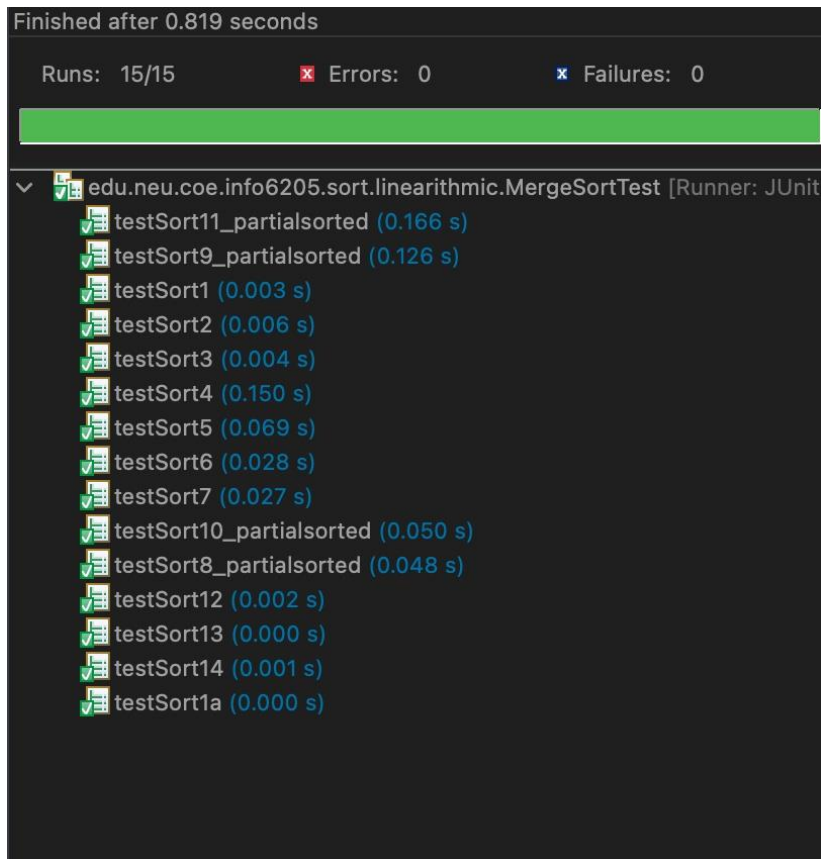
## Unit Tests

Finished after 0.07 seconds

Runs: 5/5 ✖ Errors: 0 ✖ Failures: 0

▼  edu.neu.coe.info6205.sort.linearithmic.HeapSortTest [Runner: JUnit 4]

-  testMutatingHeapSort (0.029 s)
-  sort0 (0.004 s)
-  sort1 (0.000 s)
-  sort2 (0.002 s)
-  sort3 (0.000 s)

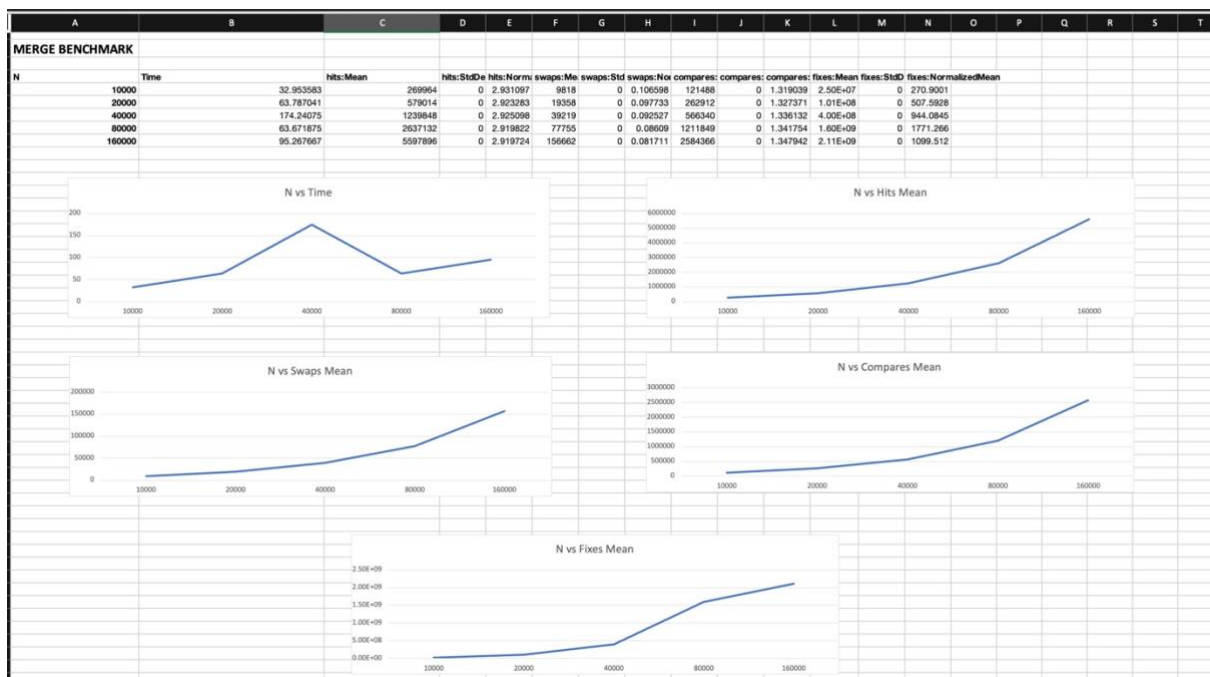




## Conclusion

```
main [Java Application] /Users/Pranavkapoor/Library/Java/JavaVirtualMachines/openjdk-17.0.2/Contents/Home/bin/java (12-Mar-2023, 7:51:15 pm) [pid: 80488]
Degree of parallelism: 7
2023-03-12 19:51:16 INFO SorterBenchmark - run: sort 10,000 elements using SorterBenchmark on class java.lang.Integer from 10,000 total elements and
2023-03-12 19:51:16 INFO SorterBenchmark - run: sort 10,000 elements using SorterBenchmark on class java.lang.Integer from 10,000 total elements and
2023-03-12 19:51:16 INFO SorterBenchmark - run: sort 10,000 elements using SorterBenchmark on class java.lang.Integer from 10,000 total elements and
2023-03-12 19:51:16 INFO Benchmark_Timer - Begin run: Instrumenting helper for QuickSort with 10,000 elements with 1 runs
2023-03-12 19:51:16 INFO Benchmark_Timer - Begin run: Instrumenting helper for MergeSort with 10,000 elements with 1 runs
2023-03-12 19:51:16 INFO Benchmark_Timer - Begin run: Instrumenting helper for HeapSort with 10,000 elements with 1 runs
2023-03-12 19:51:16 INFO TimeLogger - Raw time per run (mSec): 4.93
2023-03-12 19:51:16 INFO SorterBenchmark - run: sort 20,000 elements using SorterBenchmark on class java.lang.Integer from 20,000 total elements and
2023-03-12 19:51:16 INFO Benchmark_Timer - Begin run: Instrumenting helper for MergeSort with 20,000 elements with 1 runs
2023-03-12 19:51:16 INFO TimeLogger - Raw time per run (mSec): 22.19
2023-03-12 19:51:16 INFO SorterBenchmark - run: sort 40,000 elements using SorterBenchmark on class java.lang.Integer from 40,000 total elements and
2023-03-12 19:51:16 INFO Benchmark_Timer - Begin run: Instrumenting helper for MergeSort with 40,000 elements with 1 runs
2023-03-12 19:51:16 INFO TimeLogger - Raw time per run (mSec): 9.87
2023-03-12 19:51:16 INFO SorterBenchmark - run: sort 80,000 elements using SorterBenchmark on class java.lang.Integer from 80,000 total elements and
2023-03-12 19:51:16 INFO Benchmark_Timer - Begin run: Instrumenting helper for MergeSort with 80,000 elements with 1 runs
2023-03-12 19:51:16 INFO TimeLogger - Raw time per run (mSec): 15.03
2023-03-12 19:51:16 INFO SorterBenchmark - run: sort 160,000 elements using SorterBenchmark on class java.lang.Integer from 160,000 total elements a
2023-03-12 19:51:16 INFO Benchmark_Timer - Begin run: Instrumenting helper for MergeSort with 160,000 elements with 1 runs
2023-03-12 19:51:16 INFO TimeLogger - Raw time per run (mSec): 32.72
2023-03-12 19:51:16 INFO TimeLogger - Raw time per run (mSec): 155.01
2023-03-12 19:51:16 INFO SorterBenchmark - run: sort 20,000 elements using SorterBenchmark on class java.lang.Integer from 20,000 total elements and
2023-03-12 19:51:16 INFO Benchmark_Timer - Begin run: Instrumenting helper for QuickSort with 20,000 elements with 1 runs
2023-03-12 19:51:17 INFO TimeLogger - Raw time per run (mSec): 203.26
2023-03-12 19:51:17 INFO SorterBenchmark - run: sort 20,000 elements using SorterBenchmark on class java.lang.Integer from 20,000 total elements and
2023-03-12 19:51:17 INFO Benchmark_Timer - Begin run: Instrumenting helper for HeapSort with 20,000 elements with 1 runs
2023-03-12 19:51:19 INFO TimeLogger - Raw time per run (mSec): 706.28
2023-03-12 19:51:19 INFO SorterBenchmark - run: sort 40,000 elements using SorterBenchmark on class java.lang.Integer from 40,000 total elements and
2023-03-12 19:51:19 INFO Benchmark_Timer - Begin run: Instrumenting helper for QuickSort with 40,000 elements with 1 runs
2023-03-12 19:51:19 INFO TimeLogger - Raw time per run (mSec): 895.22
2023-03-12 19:51:19 INFO SorterBenchmark - run: sort 40,000 elements using SorterBenchmark on class java.lang.Integer from 40,000 total elements and
2023-03-12 19:51:19 INFO Benchmark_Timer - Begin run: Instrumenting helper for HeapSort with 40,000 elements with 1 runs
2023-03-12 19:51:28 INFO TimeLogger - Raw time per run (mSec): 3122.29
2023-03-12 19:51:28 INFO SorterBenchmark - run: sort 80,000 elements using SorterBenchmark on class java.lang.Integer from 80,000 total elements and
2023-03-12 19:51:28 INFO Benchmark_Timer - Begin run: Instrumenting helper for QuickSort with 80,000 elements with 1 runs
2023-03-12 19:51:31 INFO TimeLogger - Raw time per run (mSec): 3999.02
2023-03-12 19:51:31 INFO SorterBenchmark - run: sort 80,000 elements using SorterBenchmark on class java.lang.Integer from 80,000 total elements and
2023-03-12 19:51:31 INFO Benchmark_Timer - Begin run: Instrumenting helper for HeapSort with 80,000 elements with 1 runs
2023-03-12 19:52:09 INFO TimeLogger - Raw time per run (mSec): 13647.90
2023-03-12 19:52:09 INFO SorterBenchmark - run: sort 160,000 elements using SorterBenchmark on class java.lang.Integer from 160,000 total elements a
2023-03-12 19:52:09 INFO Benchmark_Timer - Begin run: Instrumenting helper for QuickSort with 160,000 elements with 1 runs
2023-03-12 19:52:20 INFO TimeLogger - Raw time per run (mSec): 16313.41
2023-03-12 19:52:20 INFO SorterBenchmark - run: sort 160,000 elements using SorterBenchmark on class java.lang.Integer from 160,000 total elements a
2023-03-12 19:52:20 INFO Benchmark_Timer - Begin run: Instrumenting helper for HeapSort with 160,000 elements with 1 runs
```

## MERGE BENCHMARK



Heap BENCHMARK



Quick BENCHMARK

