Program Structures and Algorithms

Spring 2023(SEC – 1)

NAME: Venkat Pavan Munaganti

NUID: 002722397

Task: **Assignment 6: 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).

There is no *config.ini* entry for heapsort. You will have to work that one out for yourself.

The number of runs is actually determined by the problem sizes using a fixed formula.

One more thing: the sizes of the experiments are actually defined in the command line (if you are running in IntelliJ/IDEA then, under *Edit Configurations* for the *SortBenchmark*, enter 10000 20000 etc. in the box just above *CLI arguments to your application).*

You will also need to edit the SortBenchmark class. Insert the following lines before the introsort section:

```
if (isConfigBenchmarkStringSorter("heapsort")) {
    Helper<String> helper = HelperFactory.create("Heapsort", nWords, config);
    runStringSortBenchmark(words, nWords, nRuns, new HeapSort<>(helper), timeLoggersLinearithmic);
}
```

Then you can add the following extra line into the config.ini file (again, before the introsort line (which is 25 for me):

```
heapsort = true
```

Remember that your job is to determine the best predictor: that will mean the graph of the appropriate observation will match the graph of the timings most closely.

Answer:

Below are the results from the experiment

1. **Heap Sort:** Output for the heap sort experiment is tabulated below. Run time for each input size is gathered without the instrumentation and swaps, compares, copies, hits are gathered with instrumentation

Heap Sort	Run time	Hits	Swaps	Compares
10000	1.5	967,542	124,203	235,365
20000	3.21	2,095,001	268,388	510,725
40000	6.93	4,510,097	576,792	1,101,464
80000	15.31	9,660,282	1,233,586	2,362,970
160000	35.15	20,600,741	2,627,199	5,045,972
320000	100.27	43,761,160	5,574,372	10,731,836
640000	276.95	92,643,216	11,788,870	22,743,868
1280000	823.68	195,526,186	24,857,677	48,047,738

2. **Merge Sort:** Below is the output for merge sort experiment in tabular form. Runtime is gathered without the instrumentation and copies, hits, compares are gathered with instrumentation turned on

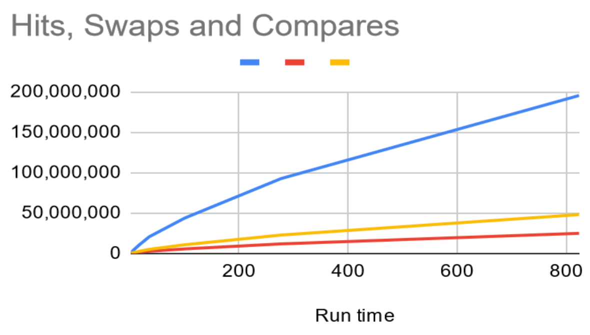
Merge Sort	Run time	Hits	Copies	Swaps	Compares
10000	1.8	269,720	110,000	9,741	121,502
20000	3.43	579,559	240,000	19,514	263,008
40000	7.53	1,239,028	520,000	39,010	565,995
80000	16.67	2,638,245	1,120,000	78,070	1,212,049
160000	38.19	5,596,744	2,400,000	156,212	2,584,120
320000	89.76	11,833,177	5,120,000	312,347	5,488,104
640000	222.88	24,946,051	10,880,000	624,619	11,616,094
1280000	555.54	52,451,910	23,040,000	1,249,199	24,512,277

3. **Quick Sort:** Output for the Quick sort experiment is tabulated below. Run time for each input size is gathered without the instrumentation and swaps, compares, copies, hits are gathered with instrumentation

Quick Sort	Run time	Hits	Swaps	Compares
10000	1.49	415,758	64,386	155,464
20000	2.07	906,129	140,200	339,841
40000	4.6	1,963,051	303,522	737,926
80000	9.78	4,206,082	651,553	1,577,825
160000	19.83	9,018,377	1,397,748	3,383,250
320000	55.79	19,217,988	2,974,243	7,232,758
640000	132.8	40,649,444	6,283,477	15,338,904
1280000	347.98	85,668,383	13,230,258	32,394,141

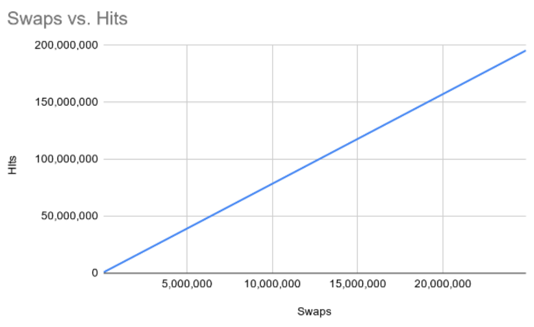
Analysis:

Heap Sort: For heap sort algorithm you can notice numbers from the above experiment that as hits are more compared to swaps and compares. Plotting a graph between the runtime and operations as below shows that higher runtimes have a greater number of hits than swaps or compares



From this graph we can see that as the runtime increases the number of hits, number of swaps and number of compares are also increasing linearly. Observing the rate of increase we can say that the number of hits has more correlation with the runtime, and it can be a best predictor of the runtime.

Although hits play an important role in the performance of the algorithm, it is not the only deciding factor that affects the runtime. Swaps and compares has a linear relationship with the hits. For a simple swap operation, we know that we will have to access the array 4 times which increases the hits by 4. Similarly, the compare also requires 2 array accesses for each compare increasing the hits by 2. For example, consider a case where we must optimize the algorithm to reduce the runtime. Reducing one swap lowers the runtime of the algorithm more than reducing one single hit or one comparison. This is true because one swap has 4 array accesses. We can see a linear relationship between the number of swaps and the number of hits in heap sort. Below is the graph to prove the same relation

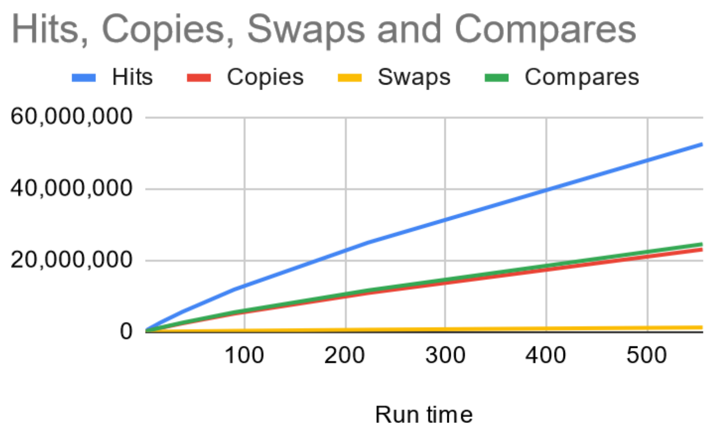


With the above data, graphs, and the experimental output, we can say that the number of ***swaps can*** be a best predictor to predict the runtime of heap sort algorithm. But we can't rely on just number of swaps to anticipate the runtime accurately. We must take multiple factors into consideration like the type of data to sort, hardware, and compares too.

To conclude,

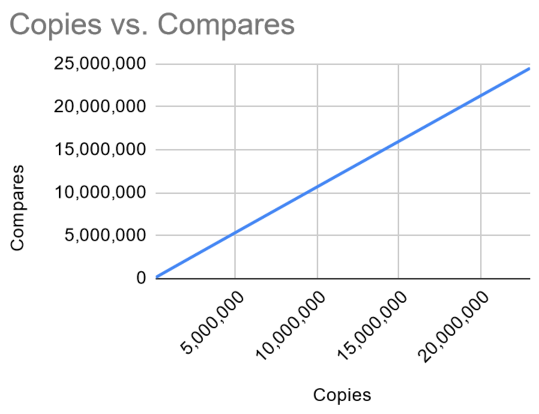
1. Hit can be a generalized predictor to predict the overall runtime of the heap sort but it is not the only factor that affects the runtime
2. Swaps has more correlation on the runtime of the algorithm than hits or compares since it involves higher number of operations, higher number of array accesses
3. We can use multiple predictors like swaps and compares together to get a better and reliable estimate of the runtime

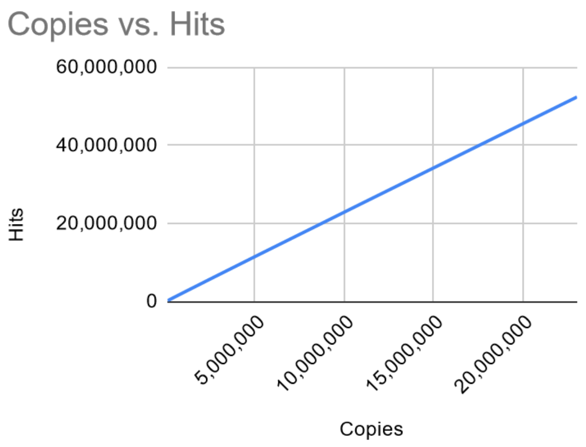
Merge Sort: Similarly in merge sort algorithm also we can notice that the number of hits is increasing as the input size is increasing and ultimately affects the performance of the algorithm. One important thing about merge sort is that there is no swapping involved in it. This algorithm has copies instead of swaps, where sub sorted arrays are copied while merging them. Thus we don't have to consider number of swaps as best predictor for merge sort algorithm.



From the graph we can notice that the number of swaps are relatively less compared to number of compares and copies for higher runtimes. Similarly the number of hits are more as the runtime increases. Number of compares and number od copies are almost similar. This is because copying in merge also involves comparison.

Below graph plots shows the relation between the number of copies and compares, and number of copies and number of hits



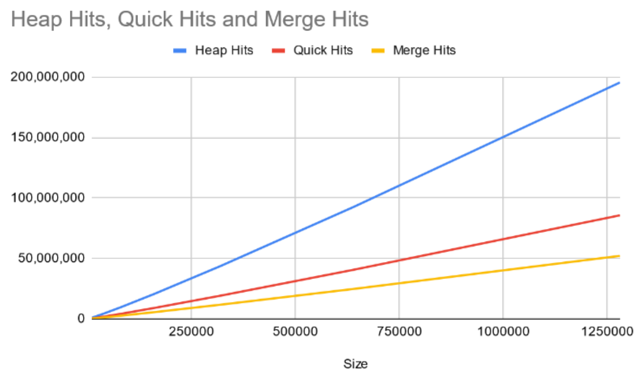


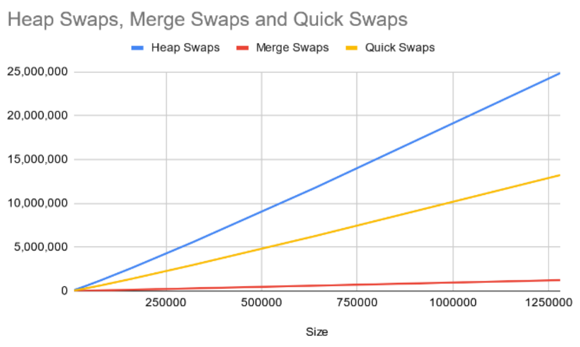
From the above graphs we can see that there is a linear relationship between the copies and compares, and copies and hits. This proves that copies has more correlation than any other factors in the merge sort algorithm. Increasing or decreasing in the number of copies has more impact on the performance of the algorithm, and increasing or decreasing the number of copies affects the compares and hits respectively.

To conclude,

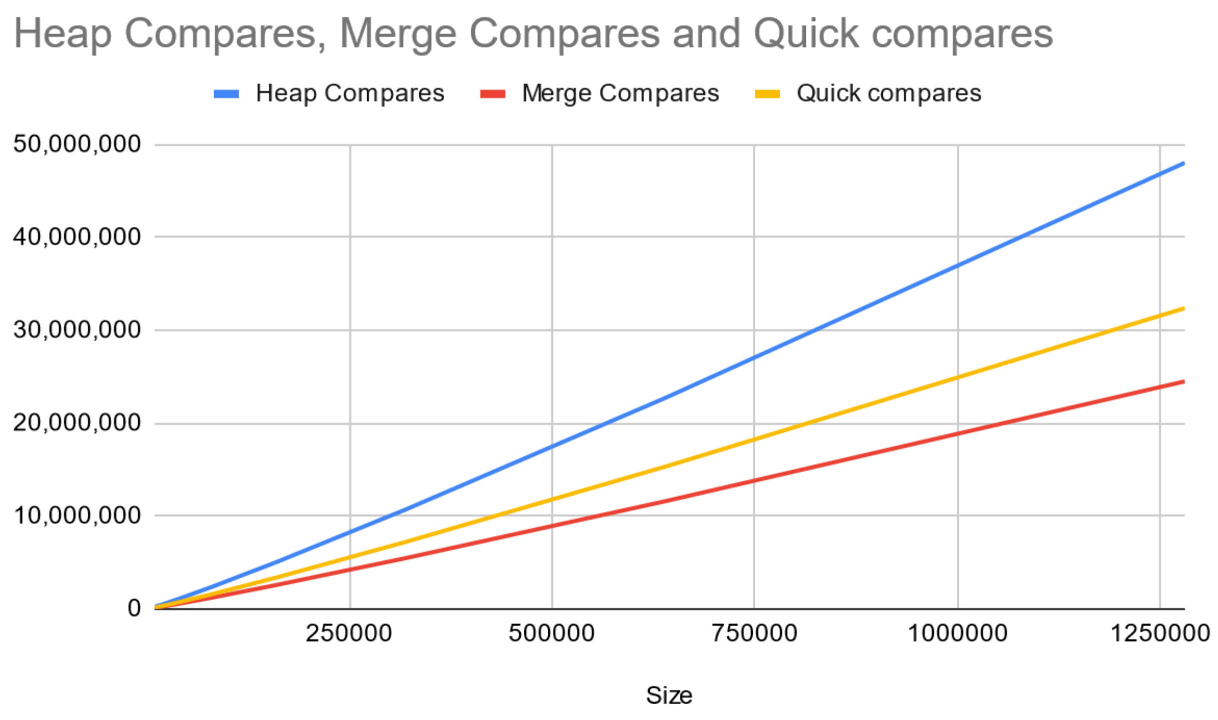
1. Hit can be a generalized predictor to predict the overall runtime of the merge sort but it is not the only factor that affects the runtime
2. Copies has more impact on the performance of the merge sort; hence it can be a best predictor of runtime of the algorithm
3. Swaps are negligible in merge sort
4. Hits and compares has a linear relationship with the number of copies

Quick Sort: Looking at the table above we can notice that for the quick sort algorithm swaps are way less compared to compares and the number of hits. Quick sort is best performing sort algorithm because the number of swaps is minimized ultimately reducing the number of hits. This is show in below graph between hits and swaps between all the algorithms





From the above graph we can notice that the number of swaps and number of hits are least for the quick sort algorithm. We can ignore the swaps for the merge sort, which are caused by the insertion sort after reaching the cut off value.



From the above graph for the compares, we can notice that the number of compares for the quick sort is more, but still relatively better than heap sort. Hence, from the above evidences we can say that compares are the best predictor for the quick sort.

To conclude,

1. Hit can be a generalized predictor to predict the overall runtime of the merge sort but it is not the only factor that affects the runtime
2. Number of compares is the best predictor for the runtime of quick sort algorithm

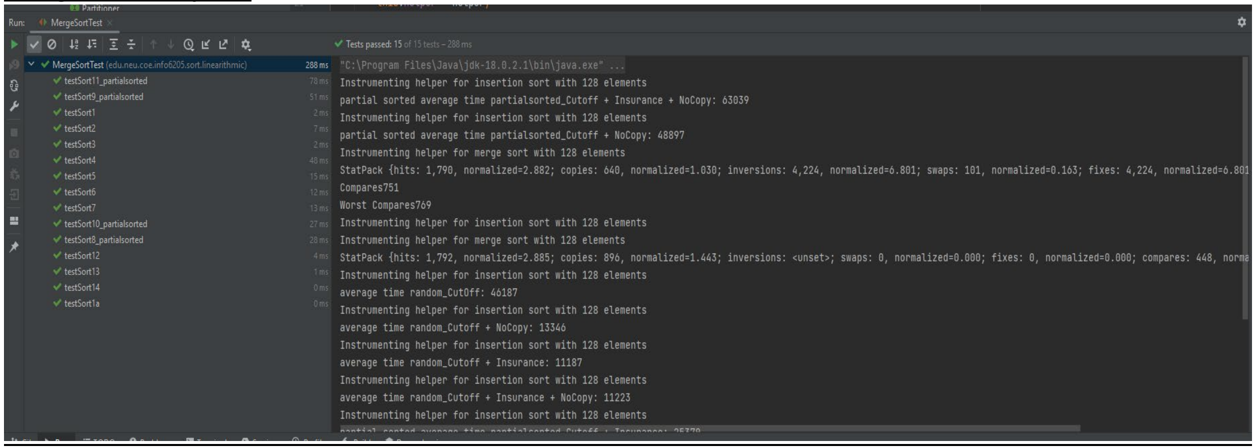
Conclusions:

1. When we don't know much about the type of algorithm, or the characteristic of the data being used, we can use number of hits as the best predictor of the runtime
2. When we know the algorithm and the data, we must use above concluded metric
    a. For heap sort, use combination of swaps and compares as the best predictor
    b. For merge sort, use copies as the best predictor
    c. For quick sort, use compares as the best predictor
3. Hit can't always depend on the swaps and compares, hits can also increase because of cache misses. This makes number of hits less reliable to predict the runtime.

4. Hits are operation that are necessary to access the values in the array but don't necessarily affect the sorting process directly, whereas swapping and comparing affects the performance.
    - When an algorithm performs a swap or copy, it is directly changing the contents of the array, which can have a big impact on the overall sorting process
    - when an algorithm performs a comparison, it is determining the relative order of two elements in the array, which is also critical to the sorting process
    - Hits are necessary for accessing the values in the array, but they don't directly change the contents of the array or determine the relative order of the elements
    - Therefore, while hits are still important for the overall efficiency of the algorithm, they may not have as direct an impact on the runtime as swaps/copies or comparisons does

<u>Unit tests:</u>

1. **<u>MergeSortTest.java:</u>**

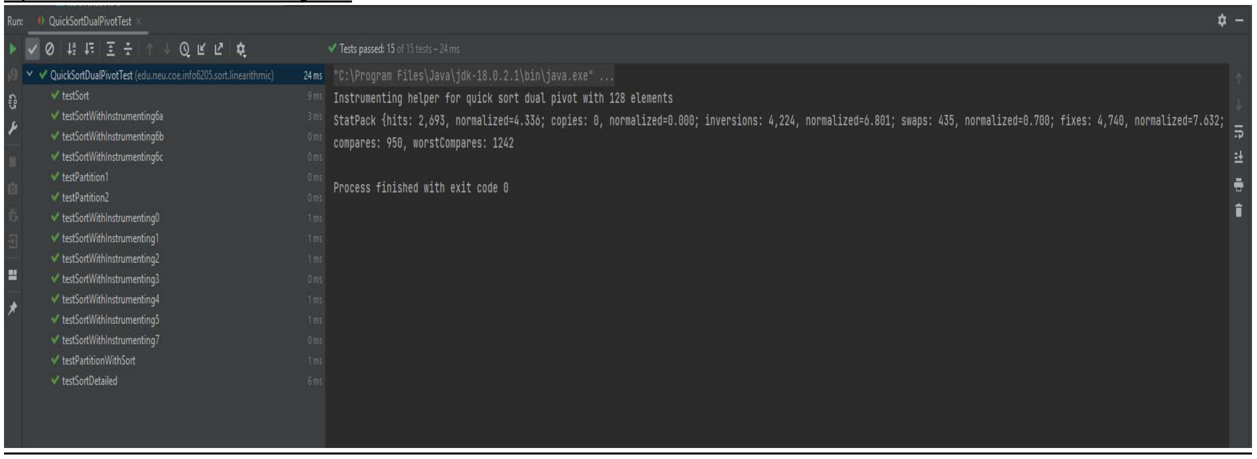


```
Run:  MergeSortTest
Tests passed: 15 of 15 tests – 288 ms

MergeSortTest (edu.neu.coe.info6205.sort.linearithmic)   288 ms
  testSort11_partialsorted   78 ms
  testSort9_partialsorted    51 ms
  testSort1                   2 ms
  testSort2                   7 ms
  testSort3                   2 ms
  testSort4                  48 ms
  testSort5                  15 ms
  testSort6                  12 ms
  testSort7                  13 ms
  testSort10_partialsorted   27 ms
  testSort8_partialsorted    28 ms
  testSort12                  4 ms
  testSort13                  1 ms
  testSort14                  0 ms
  testSort1a                  0 ms
```

```
"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe" ...
Instrumenting helper for insertion sort with 128 elements
partial sorted average time partialsorted_Cutoff + Insurance + NoCopy: 63039
Instrumenting helper for insertion sort with 128 elements
partial sorted average time partialsorted_Cutoff + NoCopy: 48897
Instrumenting helper for merge sort with 128 elements
StatPack {hits: 1,790, normalized=2.882; copies: 640, normalized=1.030; inversions: 4,224, normalized=6.801; swaps: 101, normalized=0.163; fixes: 4,224, normalized=6.801
Compares751
Worst Compares769
Instrumenting helper for insertion sort with 128 elements
Instrumenting helper for merge sort with 128 elements
StatPack {hits: 1,792, normalized=2.885; copies: 896, normalized=1.443; inversions: <unset>; swaps: 0, normalized=0.000; fixes: 0, normalized=0.000; compares: 448, norma
Instrumenting helper for insertion sort with 128 elements
average time random_CutOff: 46187
Instrumenting helper for insertion sort with 128 elements
average time random_Cutoff + NoCopy: 13346
Instrumenting helper for insertion sort with 128 elements
average time random_Cutoff + Insurance: 11187
Instrumenting helper for insertion sort with 128 elements
average time random_Cutoff + Insurance + NoCopy: 11223
Instrumenting helper for insertion sort with 128 elements
```

2. **<u>QuickSortDualPivotTest.java</u>**



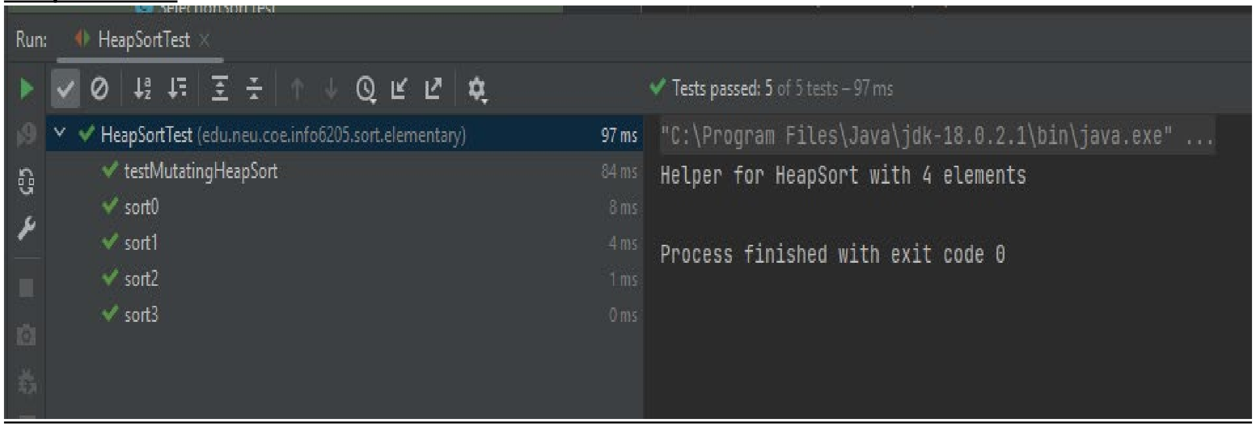
```
Run:  QuickSortDualPivotTest
Tests passed: 15 of 15 tests – 24 ms

QuickSortDualPivotTest (edu.neu.coe.info6205.sort.linearithmic)   24 ms
  testSort                          9 ms
  testSortWithInstrumenting6a       3 ms
  testSortWithInstrumenting6b       0 ms
  testSortWithInstrumenting6c       0 ms
  testPartition1                    0 ms
  testPartition2                    0 ms
  testSortWithInstrumenting0        1 ms
  testSortWithInstrumenting1        1 ms
  testSortWithInstrumenting2        1 ms
  testSortWithInstrumenting3        0 ms
  testSortWithInstrumenting4        1 ms
  testSortWithInstrumenting5        1 ms
  testSortWithInstrumenting7        0 ms
  testPartitionWithSort             1 ms
  testSortDetailed                  6 ms
```

```
"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe" ...
Instrumenting helper for quick sort dual pivot with 128 elements
StatPack {hits: 2,693, normalized=4.336; copies: 0, normalized=0.000; inversions: 4,224, normalized=6.801; swaps: 435, normalized=0.700; fixes: 4,740, normalized=7.632;
compares: 950, worstCompares: 1242

Process finished with exit code 0
```

3. **HeapSortTest:**



Run: HeapSortTest

Tests passed: 5 of 5 tests – 97 ms

```
HeapSortTest (edu.neu.coe.info6205.sort.elementary)   97 ms
    testMutatingHeapSort                              84 ms
    sort0                                              8 ms
    sort1                                              4 ms
    sort2                                              1 ms
    sort3                                              0 ms
```

```
"C:\Program Files\Java\jdk-18.0.2.1\bin\java.exe" ...
Helper for HeapSort with 4 elements


Process finished with exit code 0
```