Program Structures and Algorithms
Assignment 6 (Hits as time predictor)
Spring 2024

NAME: Suchita Arvind Dabir
NUID: 002957879
GITHUB LINK: https://github.com/suchitadabir/INFO6205

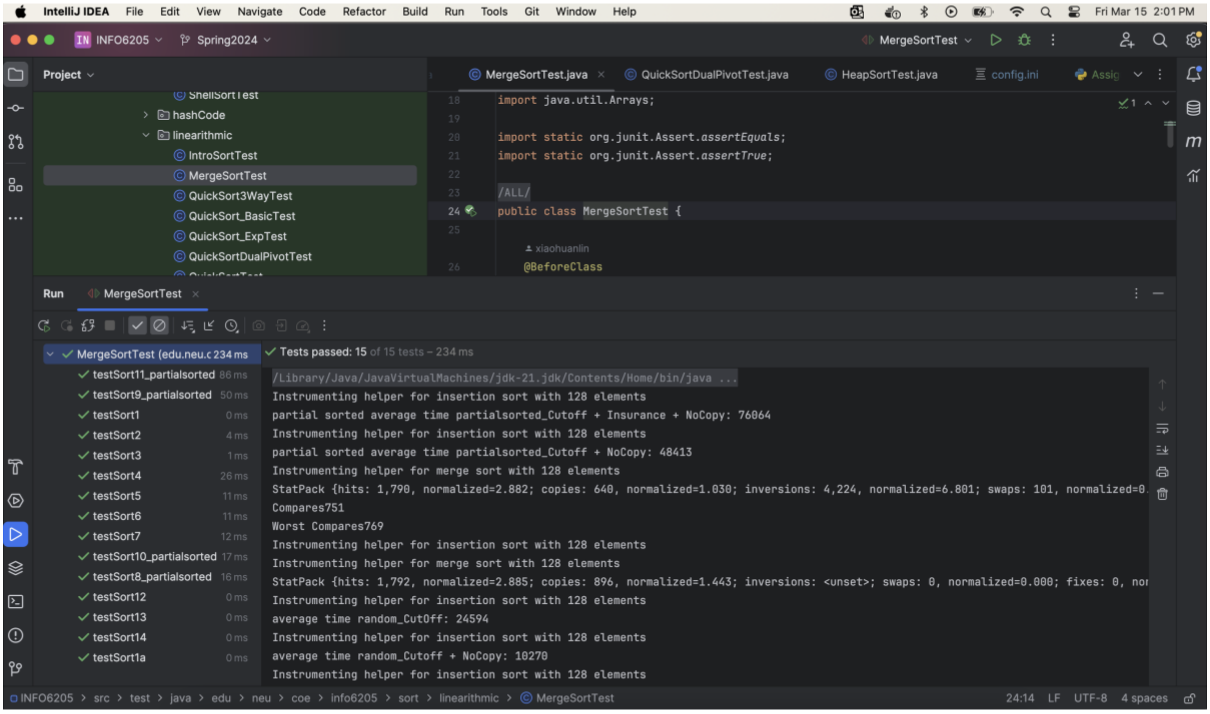
Tasks:

To determine--for sorting algorithms--what is the best predictor of total execution time: comparisons, swaps/copies, hits (array accesses), memory used, or some combination of these. That will mean the graph of the appropriate observation will match the graph of the timings most closely.

Solution:

First, I implement the merge sort with insurance and no-copy optimizations. The screenshots of unit test for all 3 sorting algorithms are attached below:

MergeSortTest:

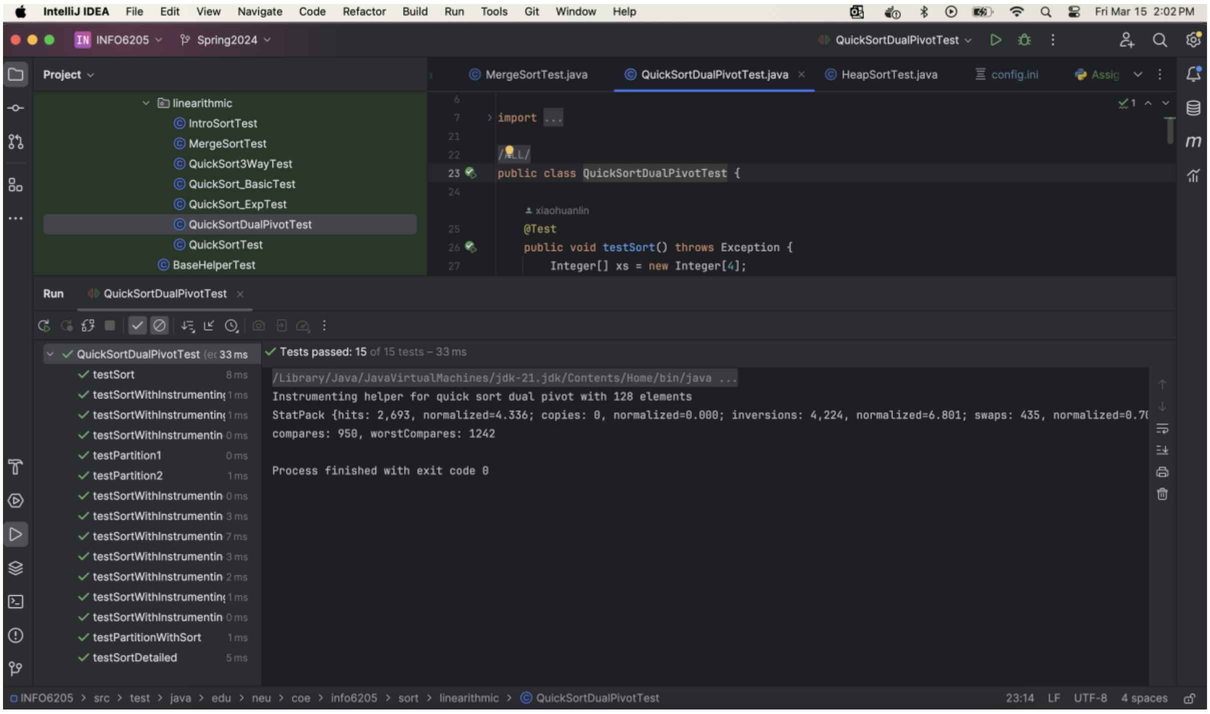


```
import java.util.Arrays;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

/ALL/
public class MergeSortTest {

    xiaohuanlin
    @BeforeClass
```

```
✓ MergeSortTest (edu.neu.c 234 ms
    ✓ testSort11_partialsorted   86 ms
    ✓ testSort9_partialsorted    50 ms
    ✓ testSort1                   0 ms
    ✓ testSort2                   4 ms
    ✓ testSort3                   1 ms
    ✓ testSort4                  26 ms
    ✓ testSort5                  11 ms
    ✓ testSort6                  11 ms
    ✓ testSort7                  12 ms
    ✓ testSort10_partialsorted   17 ms
    ✓ testSort8_partialsorted    16 ms
    ✓ testSort12                  0 ms
    ✓ testSort13                  0 ms
    ✓ testSort14                  0 ms
    ✓ testSort1a                  0 ms
```

```
✓ Tests passed: 15 of 15 tests – 234 ms
/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/java ...
Instrumenting helper for insertion sort with 128 elements
partial sorted average time partialsorted_Cutoff + Insurance + NoCopy: 76064
Instrumenting helper for insertion sort with 128 elements
partial sorted average time partialsorted_Cutoff + NoCopy: 48413
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.
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, nor
Instrumenting helper for insertion sort with 128 elements
average time random_CutOff: 24594
Instrumenting helper for insertion sort with 128 elements
average time random_Cutoff + NoCopy: 10270
Instrumenting helper for insertion sort with 128 elements
```

QuickSortDualPivotTest :

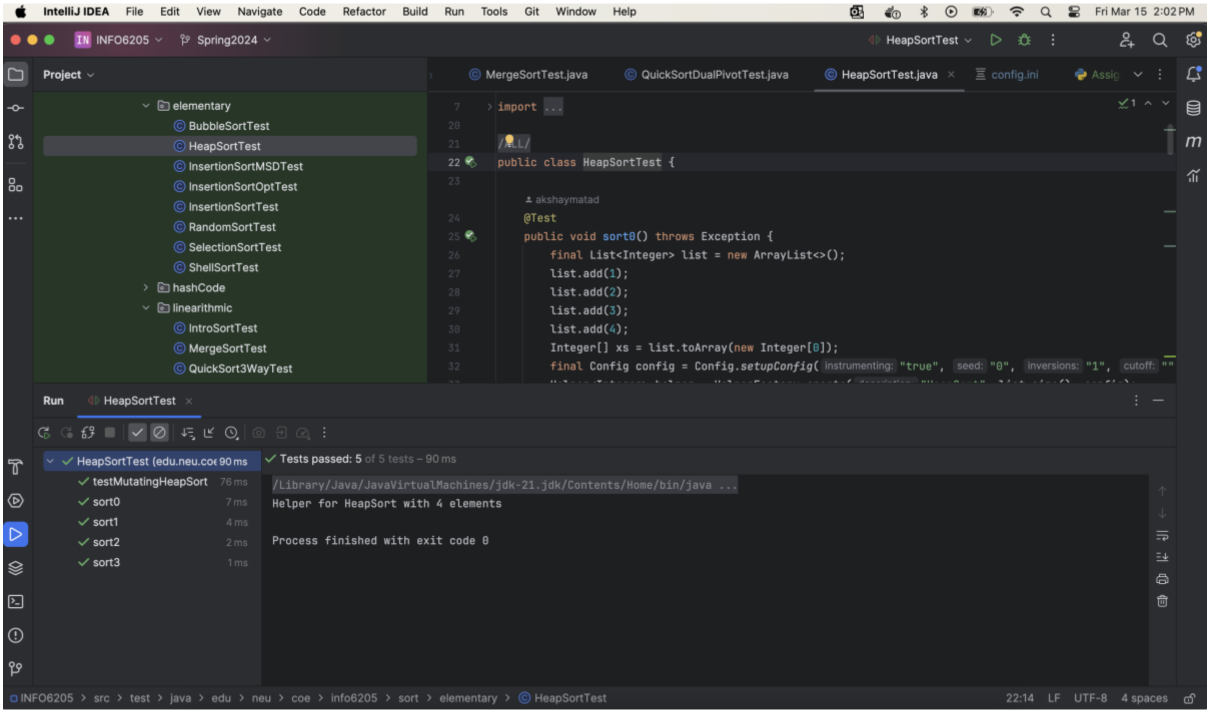


```java
import ...

/ALL/
public class QuickSortDualPivotTest {

    xiaohuanlin
    @Test
    public void testSort() throws Exception {
        Integer[] xs = new Integer[4];
```

```
QuickSortDualPivotTest (ec 33 ms
    testSort                       8 ms
    testSortWithInstrumenting      1 ms
    testSortWithInstrumenting      1 ms
    testSortWithInstrumentin       0 ms
    testPartition1                 0 ms
    testPartition2                 1 ms
    testSortWithInstrumentin       0 ms
    testSortWithInstrumentin       3 ms
    testSortWithInstrumentin       7 ms
    testSortWithInstrumentin       3 ms
    testSortWithInstrumentin       2 ms
    testSortWithInstrumenting      1 ms
    testSortWithInstrumentin       0 ms
    testPartitionWithSort          1 ms
    testSortDetailed               5 ms
```

```
Tests passed: 15 of 15 tests – 33 ms

/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/java ...
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=0.70
compares: 950, worstCompares: 1242

Process finished with exit code 0
```

HeapSortTest:



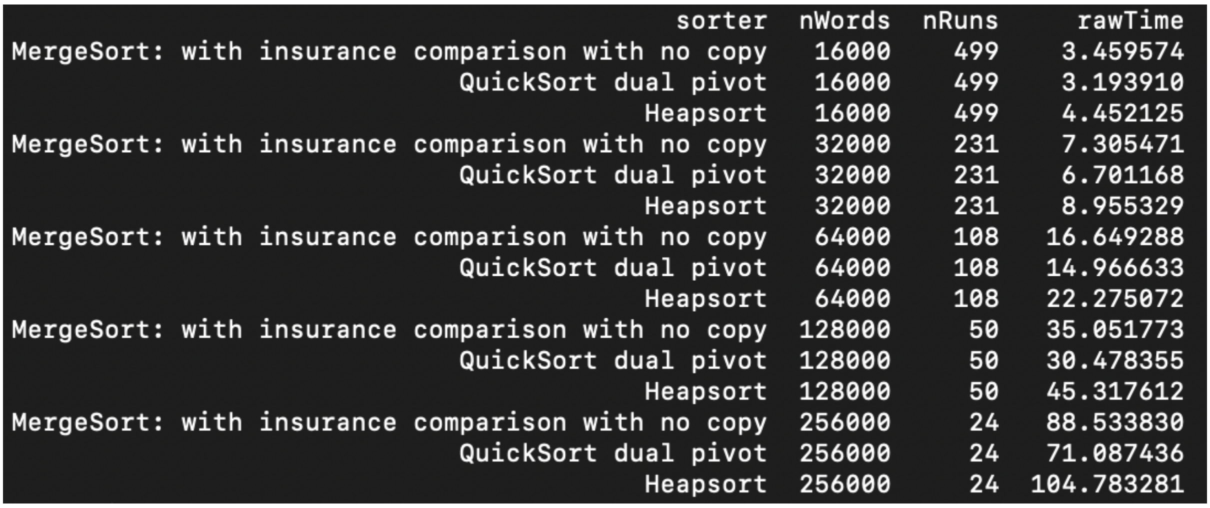
```java
import ...

/ALL/
public class HeapSortTest {

    akshaymatad
    @Test
    public void sort0() throws Exception {
        final List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        Integer[] xs = list.toArray(new Integer[0]);
        final Config config = Config.setupConfig( instrumenting: "true", seed: "0", inversions: "1", cutoff: ""
```

```
HeapSortTest (edu.neu.coe 90 ms
    testMutatingHeapSort    76 ms
    sort0                    7 ms
    sort1                    4 ms
    sort2                    2 ms
    sort3                    1 ms
```

```
Tests passed: 5 of 5 tests – 90 ms

/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin/java ...
Helper for HeapSort with 4 elements

Process finished with exit code 0
```

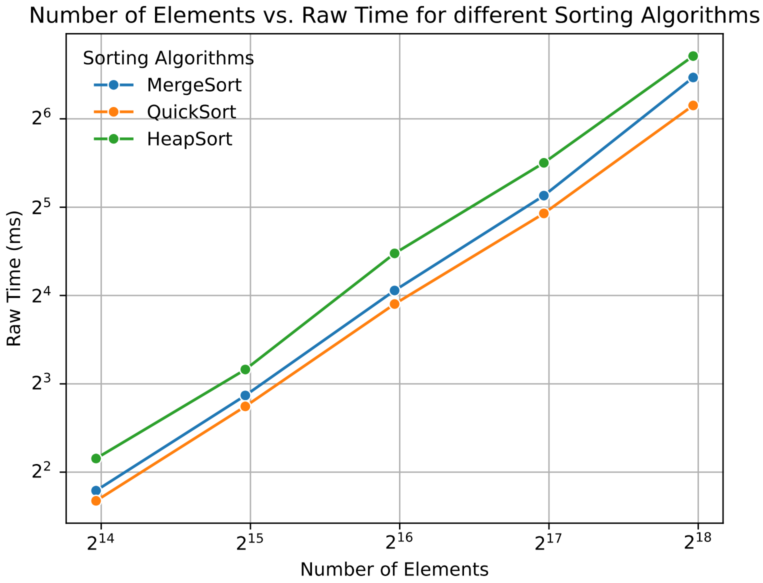
In my analysis, I double the array size starting from 16000 to 256000 and record various statistics with instrumentation enabled and time when it is disabled.

Following table represents the time to sort the array (rawTime) of length (nWords) using stated sorting algorithm (sorter) for given number of runs (nRuns). These values are recorded with instrumentation disabled.



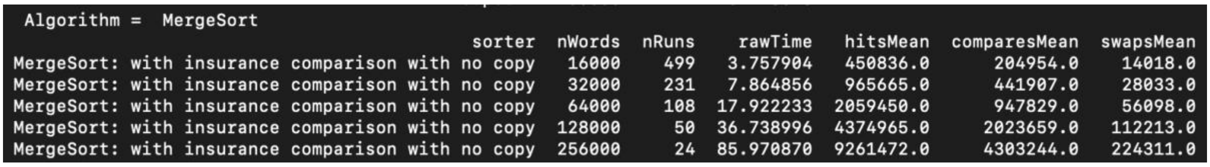
```
                                           sorter  nWords  nRuns     rawTime
MergeSort: with insurance comparison with no copy   16000    499    3.459574
                             QuickSort dual pivot   16000    499    3.193910
                                         Heapsort   16000    499    4.452125
MergeSort: with insurance comparison with no copy   32000    231    7.305471
                             QuickSort dual pivot   32000    231    6.701168
                                         Heapsort   32000    231    8.955329
MergeSort: with insurance comparison with no copy   64000    108   16.649288
                             QuickSort dual pivot   64000    108   14.966633
                                         Heapsort   64000    108   22.275072
MergeSort: with insurance comparison with no copy  128000     50   35.051773
                             QuickSort dual pivot  128000     50   30.478355
                                         Heapsort  128000     50   45.317612
MergeSort: with insurance comparison with no copy  256000     24   88.533830
                             QuickSort dual pivot  256000     24   71.087436
                                         Heapsort  256000     24  104.783281
```

This data is then plotted using the log/log scale where X-axis represents the number of elements and Y-axis shows the time taken to sort those many number of elements. I use log to the base 2 since we are using doubling method. As shown below, time taken to sort the elements increases linearly on the log scale as the number of input elements double. This proves the Linearithmic (n log n) time-complexity of all 3 sorting algorithms.

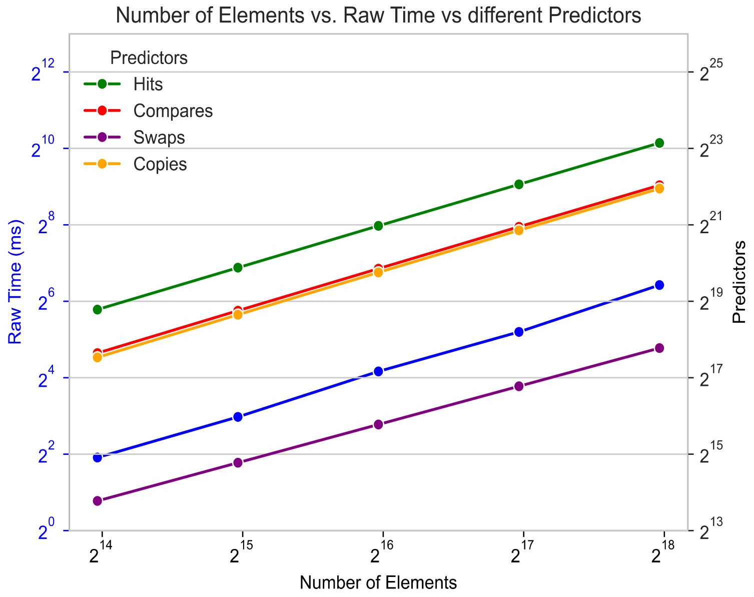


Next, from the collected data, I analyze the different predictors as well as sort time against the number of input elements. Here, I keep the sorting algorithm constant and analyze the relationship between different predictor values vs. sorting time across different number of input sizes. I simplify this process by plotting the time to sort the elements on left Y-axis, 3 predictor values on right Y-axis and number of input elements on X-axis.

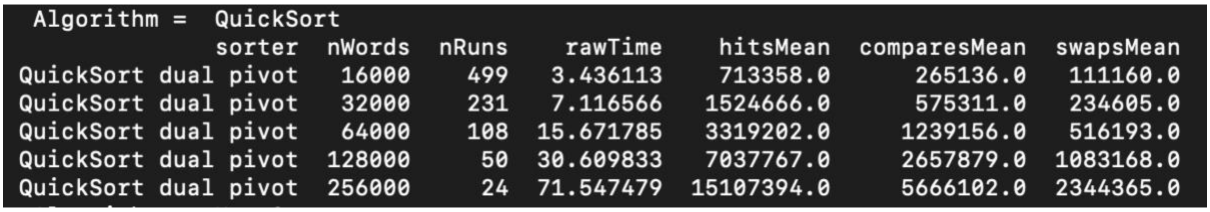
Algorithm = MergeSort: with insurance comparison with no copy:



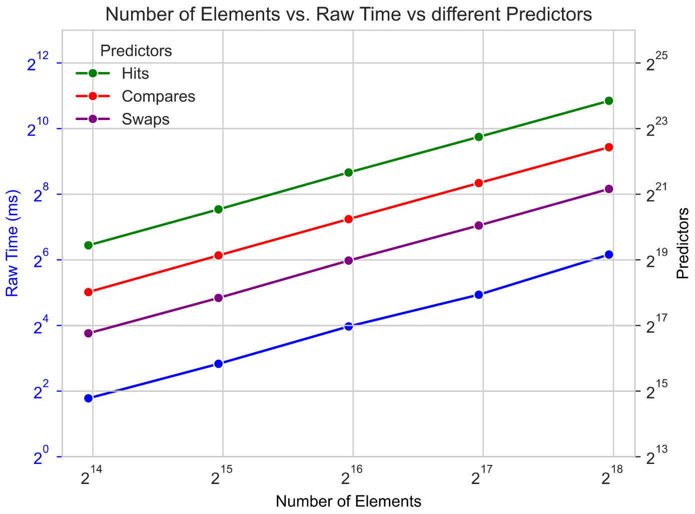
```
 Algorithm =  MergeSort
                                              sorter  nWords  nRuns   rawTime   hitsMean  comparesMean  swapsMean
MergeSort: with insurance comparison with no copy   16000    499   3.757904   450836.0      204954.0    14018.0
MergeSort: with insurance comparison with no copy   32000    231   7.864856   965665.0      441907.0    28033.0
MergeSort: with insurance comparison with no copy   64000    108  17.922233  2059450.0      947829.0    56098.0
MergeSort: with insurance comparison with no copy  128000     50  36.738996  4374965.0     2023659.0   112213.0
MergeSort: with insurance comparison with no copy  256000     24  85.970870  9261472.0     4303244.0   224311.0
```



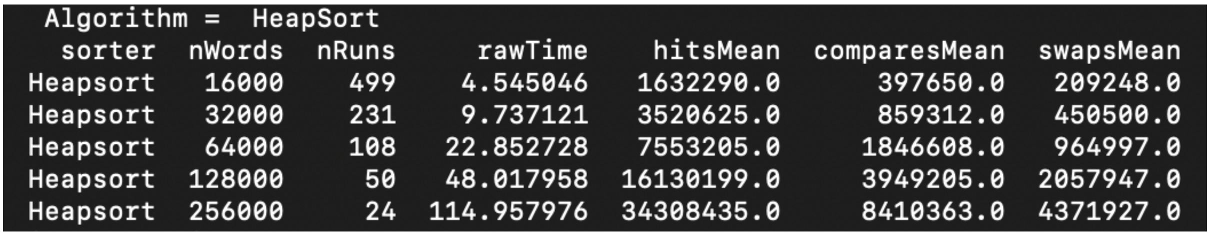
Algorithm = QuickSort



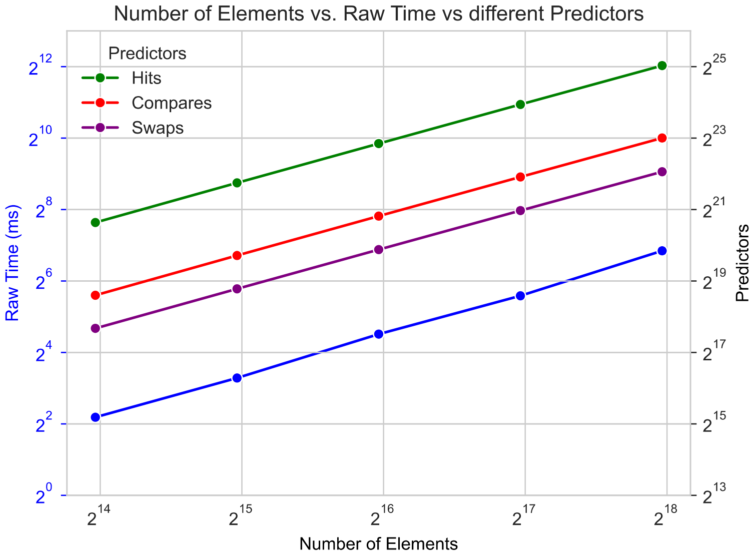
```
 Algorithm =  QuickSort
              sorter  nWords  nRuns    rawTime    hitsMean  comparesMean  swapsMean
QuickSort dual pivot   16000    499   3.436113    713358.0      265136.0   111160.0
QuickSort dual pivot   32000    231   7.116566   1524666.0      575311.0   234605.0
QuickSort dual pivot   64000    108  15.671785   3319202.0     1239156.0   516193.0
QuickSort dual pivot  128000     50  30.609833   7037767.0     2657879.0  1083168.0
QuickSort dual pivot  256000     24  71.547479  15107394.0     5666102.0  2344365.0
```



Algorithm = HeapSort



```
  Algorithm =  HeapSort
   sorter  nWords  nRuns     rawTime    hitsMean  comparesMean  swapsMean
 Heapsort   16000    499    4.545046   1632290.0      397650.0   209248.0
 Heapsort   32000    231    9.737121   3520625.0      859312.0   450500.0
 Heapsort   64000    108   22.852728   7553205.0     1846608.0   964997.0
 Heapsort  128000     50   48.017958  16130199.0     3949205.0  2057947.0
 Heapsort  256000     24  114.957976  34308435.0     8410363.0  4371927.0
```

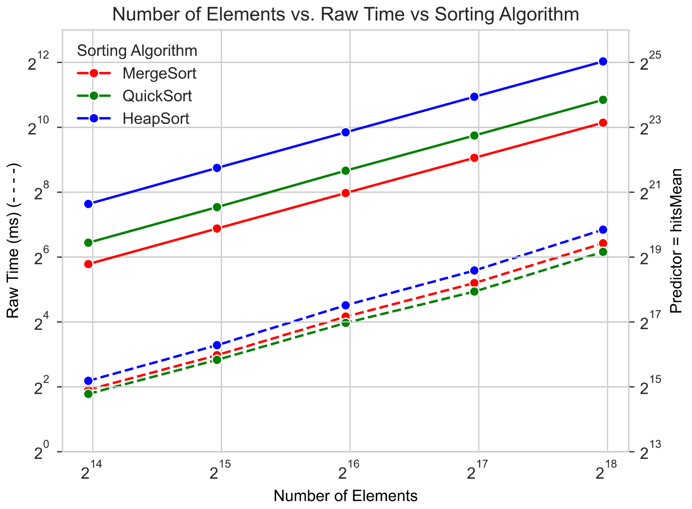


Above 3 tables and plots for 3 different algorithms show that:

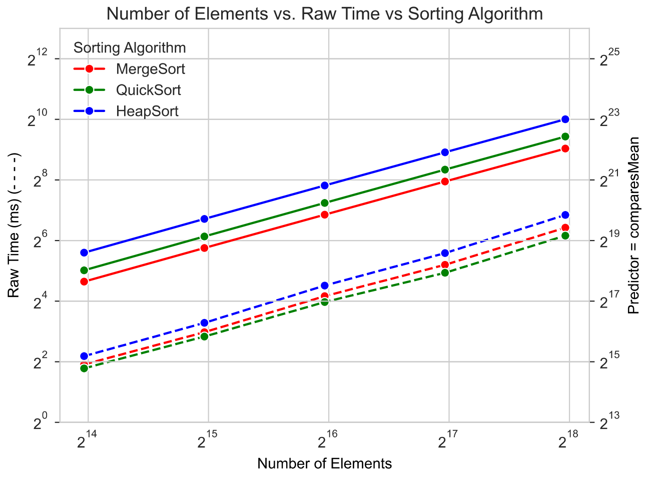
1. As the number of elements increases, 3 predictors – Avg. # Hits, Avg. # Compares and Avg. # Swaps also increases linearly on the log-scale.
2. All 3 predictors are directly proportional to the time taken to sort the elements for all 3 chosen sorting algorithms
3. Number of copies = 0 for (dual-pivot) quick sort, and heap sort. Hence, it cannot be the best predictor of execution time.

Now, I will choose one predictor at a time and plot the time to sort the elements on left Y-axis, chosen predictor value on right Y-axis and number of input elements on X-axis for all 3 sorting algorithms. These plots are shown below which represents the log scale on X as well as both Y-axis. The dotted line corresponds to left Y-axis that represents the sorting time where as solid line corresponds to the right Y-axis that shows the predictor value.

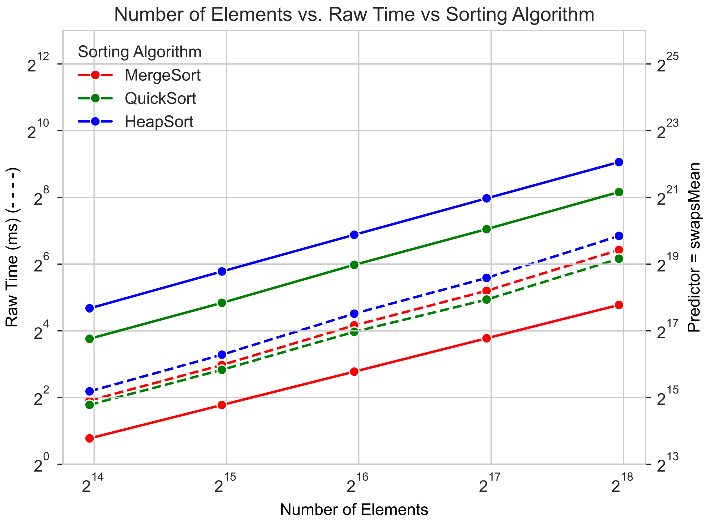
Predictor = Avg. number of Hits:



Predictor = Avg. number of Compares:



Predictor = Avg. number of Swaps:



From above 3 plots, we can conclude that, in the 2nd plot, the time to sort the elements highly correlates with the average # of comparisons for all 3 algorithms. Here, 3 lines corresponding to the time have equal slope as that representing the average # of comparisons. Moreover, 3 dotted lines that represent the time can be seen as just the scaled down version of compare lines with equidistant vertical scale across different sorting algorithms.

Furthermore, I also use pandas correlation function to calculate the relationship between log base 2 of time with the log base 2 of each predictor and I get below values:
Time Correlation with Hits 0.9316493091443672
Time Correlation with Compares 0.9834631342070014
Time Correlation with Swaps 0.6825275153589979

From this data, we can conclude that **sort time has the highest correlation value with number of compares** followed by number of hits. Thus, descending order of best time predictors would be comparisons and then hits as they closely align with the run time of sorting algorithms.

Conclusion: Based on above spreadsheets and plots, it can be concluded that **comparisons** is the best predictor of total execution time for merge sort, (dual pivot) quick sort, and heap sort.

Note: Entire CSV file with all the data as well as the python script used to generate the plots is also added to the GitHub repository.