Program Structures and Algorithms
Spring 2023(SEC 03)

NAME: Vipul Rajderkar
NUID: 002700991

**Task:**
Determine--for sorting algorithms--what is the best predictor of total execution time: comparisons, swaps/copies, hits (array accesses), or something else.

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.

**Relationship Conclusion:**

Merge Sort: Based on the analysis of the log/log chart and the spreadsheet, it seems that the best predictor of total execution time for merge sort is the number of comparisons performed during the sorting process.

Quick Sort: Number of compare was found the best predictor followed by number of swaps and inversions

Heap Sort: After analysing the log-log plots, it could be inferred that rate of increase in number of compare, swaps and hits was almost same.

**Evidence to support that conclusion and graphical representation:**

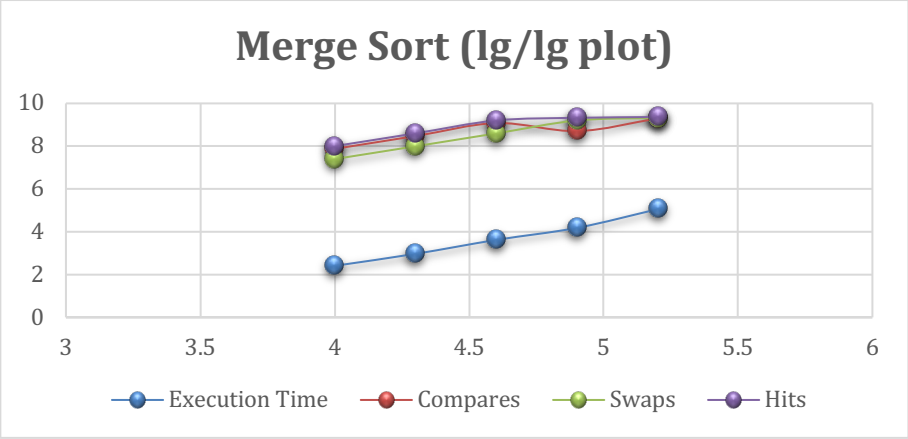
1. **Merge Sort**

It was evident from the results that the execution time of merge sort increases with increase in size of an array. But the rate of increase in execution time is smaller than a linear rate.

| Array Size | Execution Time | Compares | Swaps | Hits |
|---|---|---|---|---|
| 10000 | 256 | 75183576 | 25123981 | 101030388 |
| 20000 | 933 | 299709369 | 99580158 | 399469560 |
| 40000 | 4189 | 1200602204 | 400323781 | 1.604E+09 |
| 80000 | 15393 | 508527950 | 1602898394 | 2.122E+09 |
| 160000 | 115816 | 2018903858 | 2102532030 | 2.316E+09 |

Log scale:

| Array Size | Execution Time | Compares | Swaps | Hits |
|---|---|---|---|---|
| 4 | 2.408239965 | 7.876122978 | 7.40008846 | 8.00445202 |
| 4.301029996 | 2.969881644 | 8.476700319 | 7.99817281 | 8.60148369 |
| 4.602059991 | 3.62211036 | 9.079399136 | 8.60241139 | 9.20513748 |
| 4.903089987 | 4.187323269 | 8.706314828 | 9.20490599 | 9.32671713 |
| 5.204119983 | 5.063768561 | 9.305115638 | 9.32274262 | 9.36477206 |



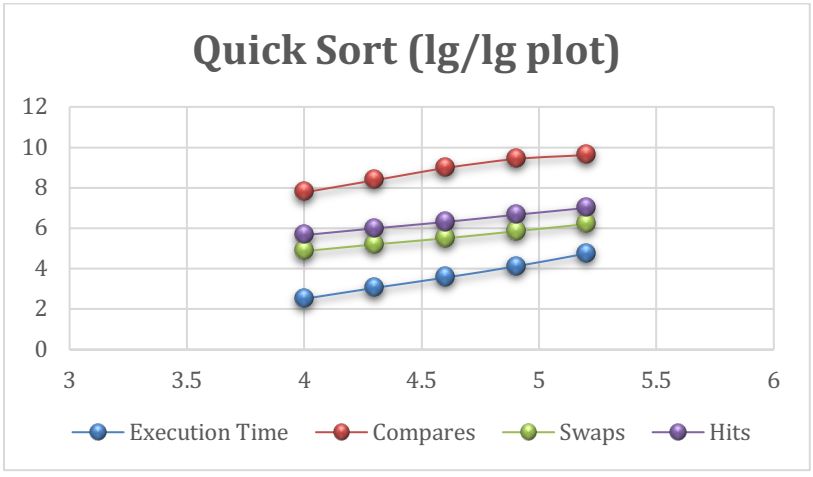

The number of comparisons made during the sorting process appears to be the greatest predictor of the overall execution time for merge sort, according to the analysis of the log/log chart and the spreadsheet. Although not as powerful as the number of comparisons, the number of swaps and inversions also exhibit a moderate correlation with execution time.

## 2. Quick Sort

Quick sort typically takes longer to execute the larger the array is. Along with the array size, there are typically more compares, swaps, and inversions. There does not appear to be a clear correlation between the quantity of copies and fixes and array size. The log-log chart shows that quick sort has an O time complexity, meaning that there appears to be an approximate linear relationship between array size and execution time (n log n).

| Array Size | Execution Time | Compares | Swaps | Hits |
|---|---|---|---|---|
| 10000 | 324 | 59137352 | 73122 | 456168 |
| 20000 | 1130 | 237819113 | 158748 | 990319 |
| 40000 | 3645 | 947599391 | 317088 | 2039826 |
| 80000 | 13213 | 2656351070 | 710628 | 4642644 |
| 160000 | 56646 | 4162993911 | 1602866 | 10143923 |

| Array Size | Execution Time | Compares | Swaps | Hits |
|---|---|---|---|---|
| 4 | 2.51054501 | 7.771861874 | 4.86404806 | 5.65912482 |
| 4.301029996 | 3.053078443 | 8.376246755 | 5.20070826 | 5.99577511 |
| 4.602059991 | 3.561697533 | 8.976624773 | 5.50117981 | 6.30959312 |
| 4.903089987 | 4.121001435 | 9.424285472 | 5.85164232 | 6.66676538 |
| 5.204119983 | 4.753169248 | 9.619405776 | 6.20489722 | 7.00620594 |



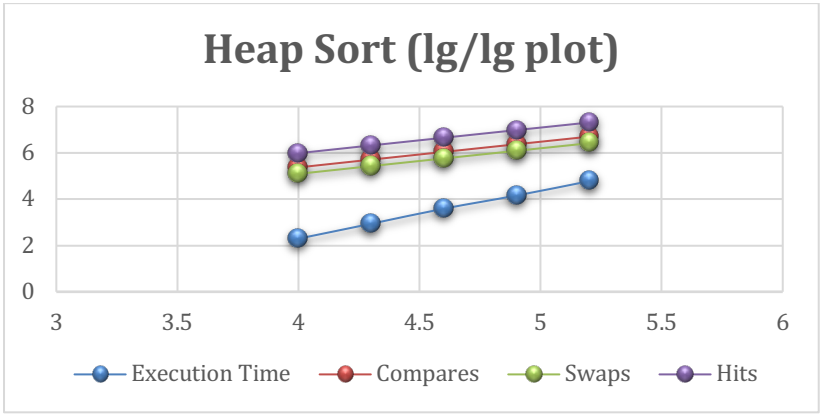
It could be observed that the number of compares has the highest correlation with execution time, followed by the number of swaps and inversions. This suggests that the number of compares is the best predictor of total execution time for quick sort.

### 3. Heap Sort

The total number of compares, swaps, and fixes grows along with the size of the input array.

| Array Size | Execution Time | Compares | Swaps | Hits |
|---|---|---|---|---|
| 10000 | 197 | 235354 | 124125 | 967208 |
| 20000 | 870 | 510633 | 268283 | 2094398 |
| 40000 | 3918 | 1101510 | 576766 | 4510084 |
| 80000 | 14543 | 2362997 | 1233804 | 9661210 |
| 160000 | 60764 | 5046098 | 2627456 | 20602020 |

| Array Size | Execution Time | Compares | Swaps | Hits |
|---|---|---|---|---|
| 4 | 2.294466226 | 5.371721584 | 5.09385926 | 5.98551988 |
| 4.301029996 | 2.939519253 | 5.708108878 | 5.42859315 | 6.32105921 |
| 4.602059991 | 3.593064432 | 6.041988444 | 5.76099965 | 6.65418463 |
| 4.903089987 | 4.162654004 | 6.37346317 | 6.09124617 | 6.98503152 |
| 5.204119983 | 4.783646355 | 6.702955681 | 6.41953545 | 7.3139098 |




Rate of increase in compare, swaps, and hits was found almost the same and hence best predictor could not be decided.

**Code Snippet:**

Changes in SortBenchmark class:
Main method:

```
Vipul Rajderkar
public static void main(String[] args) throws Exception {
//      Config config = Config.load(SortBenchmark.class);
//      logger.info("SortBenchmark.main: " + config.get("SortBenchmark", "version") + " with word counts: " + Arrays.toString(args));
//      if (args.length == 0) logger.warn("No word counts specified on the command line");
//      SortBenchmark benchmark = new SortBenchmark(config);
//      benchmark.sortIntegersByShellSort(config.getInt("shellsort", "n", 100000));
//      benchmark.sortStrings(Arrays.stream(args).map(Integer::parseInt));
//      benchmark.sortLocalDateTimes(config.getInt("benchmarkdatesorters", "n", 100000), config);

    final Config config = Config.setupConfig( instrumenting: "true",  seed: "0",  inversions: "1",  cutoff: "1",  interimInversions: "");

    for(int i = 10000; i <= 256000; i *= 2) {
        runMergeSortBenchmark(i, config);
        runQuickSortBenchmark(i, config);
        runHeapSortBenchmark(i, config);
    }
    // 10, 20 40 80 160 32 64 128 256
}
```

Merge sort:

```
1 usage  Vipul Rajderkar
public static void runMergeSortBenchmark(int size, Config config) throws Exception {

    int n = size;



    // Merge Sort
    BaseHelper<Integer> helper1 = new InstrumentedHelper<>( description: "test", config);

    MergeSortBasic<Integer> merge = new MergeSortBasic<>(helper1);

    Consumer<Integer[]> randomFunc1 = randArr1 -> merge.sort(randArr1);
    Benchmark_Timer<Integer[]> randomTimer1 = new Benchmark_Timer<>( description: "Sort array of " + n + " elements", randomFunc1);
    Supplier<Integer[]> random1 = () -> {
        Random randI = new Random();
        Integer[] randArr1 = new Integer[n];
        for(int i=0; i<n; i++) {
            int randInt = randI.nextInt(n);
            randArr1[i] = randInt+1;
        }
        return randArr1;
    };
    randomFunc1.accept(random1.get());
    double randTime1 = randomTimer1.run(random1.get(),  m: 1);
    System.out.println("Mergesort Time for array of " + n + " elements is: " + randTime1);

    helper1.postProcess(merge.sort(random1.get()));

    PrivateMethodTester privateMethodTester1 = new PrivateMethodTester(helper1);
    StatPack statPack1 = (StatPack) privateMethodTester1.invokePrivate( name: "getStatPack");

    long mergeCompares = (long) statPack1.getStatistics(InstrumentedHelper.COMPARES).mean();
    long mergeSwaps = (long) statPack1.getStatistics(InstrumentedHelper.SWAPS).mean();
    long mergeHits = (long) statPack1.getStatistics(InstrumentedHelper.HITS).mean();
    System.out.println("Compares = " + mergeCompares);
    System.out.println("Swaps = " + mergeSwaps);
    System.out.println("Hits = " + mergeHits);

}
```

Quick Sort:

```
1 usage  Vipul Rajderkar
public static void runQuickSortBenchmark(int size, Config config) throws Exception {

    int n =size;
    BaseHelper<Integer> helper = new InstrumentedHelper<>( description: "test", config);

    QuickSort<Integer> quick = new QuickSort_DualPivot<>(helper);

    Consumer<Integer[]> randomFunc = randArr -> quick.sort(randArr);
    Benchmark_Timer<Integer[]> randomTimer = new Benchmark_Timer<>( description: "Sort array of " + n + " elements", randomFunc);
    Supplier<Integer[]> random = () -> {
        Random randI = new Random();
        Integer[] randArr = new Integer[n];
        for(int i=0; i<n; i++) {
            int randInt = randI.nextInt(n);
            randArr[i] = randInt+1;
        }
        return randArr;
    };
    randomFunc.accept(random.get());
    double randTime = randomTimer.run(random.get(),  m: 1);
    System.out.println("Quicksort Time for array of " + n + " elements is: " + randTime);

    helper.postProcess(quick.sort(random.get()));
    PrivateMethodTester privateMethodTester = new PrivateMethodTester(helper);
    StatPack statPack = (StatPack) privateMethodTester.invokePrivate( name: "getStatPack");

    long quickCompares = (long) statPack.getStatistics(InstrumentedHelper.COMPARES).mean();
    long quickSwaps = (long) statPack.getStatistics(InstrumentedHelper.SWAPS).mean();
    long quickHits = (long) statPack.getStatistics(InstrumentedHelper.HITS).mean();
    System.out.println("Compares = " + quickCompares);
    System.out.println("Swaps = " + quickSwaps);
    System.out.println("Hits = " + quickHits);

}
```

Heap Sort:

```
1 usage  Vipul Rajderkar
public static void runHeapSortBenchmark(int size, Config config) throws Exception {

    int n = size;
    BaseHelper<Integer> helper2 = new InstrumentedHelper<>( description: "test", config);

    HeapSort<Integer> heap = new HeapSort<>(helper2);

    Consumer<Integer[]> randomFunc2 = randArr2 -> heap.sort(randArr2);
    Benchmark_Timer<Integer[]> randomTimer2 = new Benchmark_Timer<>( description: "Sort array of " + n + " elements", randomFunc2);
    Supplier<Integer[]> random2 = () -> {
        Random randI = new Random();
        Integer[] randArr2 = new Integer[n];
        for(int i=0; i<n; i++) {
            int randInt = randI.nextInt(n);
            randArr2[i] = randInt+1;
        }
        return randArr2;
    };
    randomFunc2.accept(random2.get());
    double randTime2 = randomTimer2.run(random2.get(),  m: 1);
    System.out.println("Heapsort Time for array of " + n + " elements is: " + randTime2);

    helper2.postProcess(heap.sort(random2.get()));

    PrivateMethodTester privateMethodTester2 = new PrivateMethodTester(helper2);
    StatPack statPack2 = (StatPack) privateMethodTester2.invokePrivate( name: "getStatPack");

    long heapCompares = (long) statPack2.getStatistics(InstrumentedHelper.COMPARES).mean();
    long heapSwaps = (long) statPack2.getStatistics(InstrumentedHelper.SWAPS).mean();
    long heapHits = (long) statPack2.getStatistics(InstrumentedHelper.HITS).mean();
    System.out.println("Compares = " + heapCompares);
    System.out.println("Swaps = " + heapSwaps);
    System.out.println("Hits = " + heapHits);

}
```

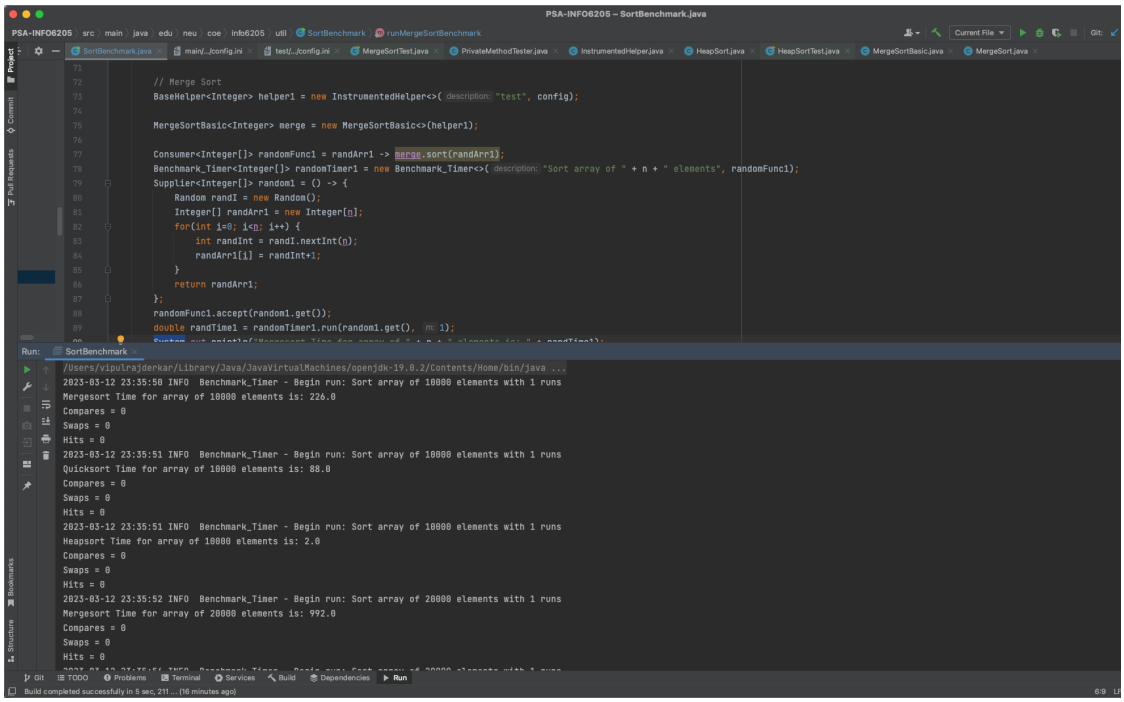
Output:

Both with and without instrumention outputs have been added in the output file.

For entire output logs please refer Hits as a predictor Output.docx file (Added in the repository)

Sample Output:

Without instrumentation:




**With instrumentation:**