Assignment 6 (Hits as time predictor)

1. Intro

In this question, I was asked to figure out--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.

2. Implement

2.1 TimePredictorTest

I wrote a test class named **TimePredictorTest**.class under the package **edu.neu.coe.info6205.sort**

In this case, I was asked to sort randomly generated arrays of between 10,000 and 256,0000 elements (doubling the size each time, at least run 20 times). However, when I used heap sort to sort the array which length is 128,000. It took me about **40 minutes** to calculate its comparisons, swaps, hits. **Therefore, I only sort randomly generated Integer arrays which length between 1,000 and 128,000.**

```
public class TimePredictorTest {

    private final String HeapSort = "HeapSort";

    private final String MergeSort = "MergeSort";

    private final String QuicksortDualPivot = "QuicksortDualPivot";

    @Test
    public void testSortBenchMark() {
        String sortingMethod = QuicksortDualPivot;
        for (int i = 1000; i <= 128000; i *= 2) {
            System.out.println("The length of Array is: " + i);
            int n = i;
            int m = 20;
            Supplier<Integer[]> integersSupplier = getSupplier(n);
            //Calculate the time for sorting
            benchmarkIsInstrumented("false", integersSupplier, n, m, sortingMethod);
            //Count operations
            benchmarkIsInstrumented("true", integersSupplier, n, m, sortingMethod);
        }
    }

    private void benchmarkIsInstrumented(String isInstrumented, Supplier<Integer[]>
integersSupplier, int n, int m, String sortingMethod) {
```

```
        Config config;
        if (sortingMethod.equals(MergeSort))
            config = Config.setupConfig2(isInstrumented, "0", "1", "", "", "false",
"true");
        else
            config = Config.setupConfig(isInstrumented, "0", "1", "", "");
        Helper<Integer> helper = HelperFactory.create(sortingMethod, n, config);
        helper.init(n);
        final PrivateMethodTester privateMethodTester = new PrivateMethodTester(helper);
        StatPack statPack = null;
        if (config.isInstrumented())
            statPack = (StatPack) privateMethodTester.invokePrivate("getStatPack");
        SortWithHelper<Integer> sorter = null;
        switch (sortingMethod) {
            case HeapSort:
                sorter = new HeapSort<>(helper);
                break;
            case MergeSort:
                sorter = new MergeSort<>(helper);
                break;
            case QuicksortDualPivot:
                sorter = new QuickSort_DualPivot<>(helper);
                break;
        }
        benchmark(config, statPack, sorter, sortingMethod, integersSupplier, m);
    }

    private void benchmark(Config config, StatPack statPack, SortWithHelper<Integer>
sorter, String sortingMethod, Supplier<Integer[]> integersSupplier, int m) {
        double avgTime = new Benchmark_Timer<Integer[]>(
                sortingMethod,
                (xs) -> {
                    Integer[] copy = Arrays.copyOf(xs, xs.length);
                    sorter.preProcess(xs);
                    return copy;
                },
                (xs) -> sorter.sort(xs, false), //Copy the array in the preFunction
                sorter::postProcess
        ).runFromSupplier(integersSupplier, m);
        if (!config.isInstrumented()) {
            System.out.println("avgTime in milliseconds: " + avgTime);
        }
        if (config.isInstrumented()) {
            assert statPack != null;
            System.out.println("Compares: " + (int)
statPack.getStatistics(InstrumentedHelper.COMPARES).mean());
```

```
            System.out.println("Swaps: " + (int)
statPack.getStatistics(InstrumentedHelper.SWAPS).mean());
            System.out.println("Hits: " + (int)
statPack.getStatistics(InstrumentedHelper.HITS).mean());
            System.out.println();
        }
    }

    private Supplier<Integer[]> getSupplier(int n) {
        final Random random = new Random();
        return () -> {
            Integer[] result = (Integer[]) Array.newInstance(Integer.class, n);
            for (int i = 0; i < n; i++) result[i] = random.nextInt();
            return result;
        };
    }
}
```

2.2 Merge Sort

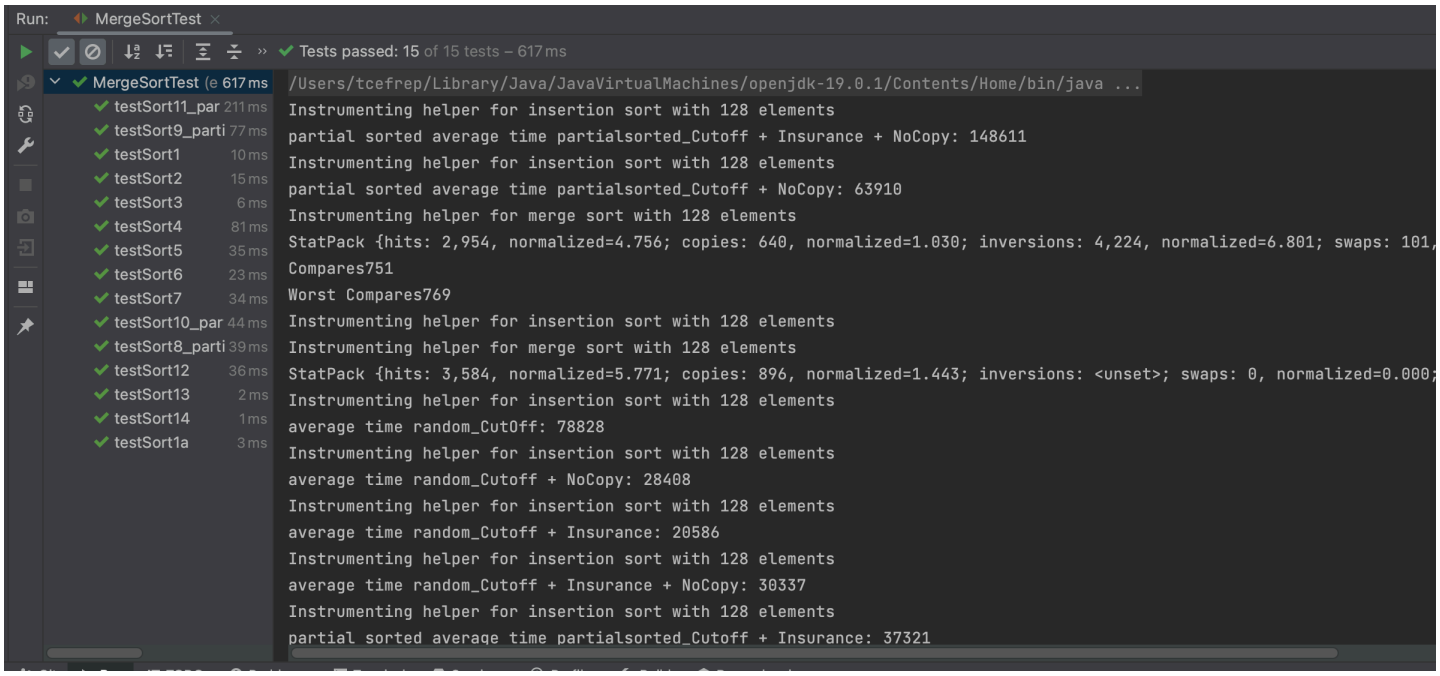
```
    private void sort(X[] a, X[] aux, int from, int to) {
        final Helper<X> helper = getHelper();
        Config config = helper.getConfig();
        boolean insurance = config.getBoolean(MERGESORT, INSURANCE);
        boolean noCopy = config.getBoolean(MERGESORT, NOCOPY);
        if (to <= from + helper.cutoff()) {
            insertionSort.sort(a, from, to);
            return;
        }
        final int n = to - from;
        int mid = from + (n) / 2;

        if (insurance) {
            sort(a, aux, from, mid);
            sort(a, aux, mid, to);
            if (helper.less(a[mid - 1], a[mid])) return;
            merge(a, aux, from, mid, to);
        }
        helper.incrementHits(2 * n);
        if (noCopy) {
            sort(aux, a, from, mid);
            sort(aux, a, mid, to);
        } else {
            sort(a, aux, from, mid);
            sort(a, aux, mid, to);
```

```
            System.arraycopy(a, from, aux, from, n);
        }
        merge(aux, a, from, mid, to);
        // END
    }
```

I also finish the code in Merge Sort. Insurance can make it more adaptive, nocopy can reduce the processing time.

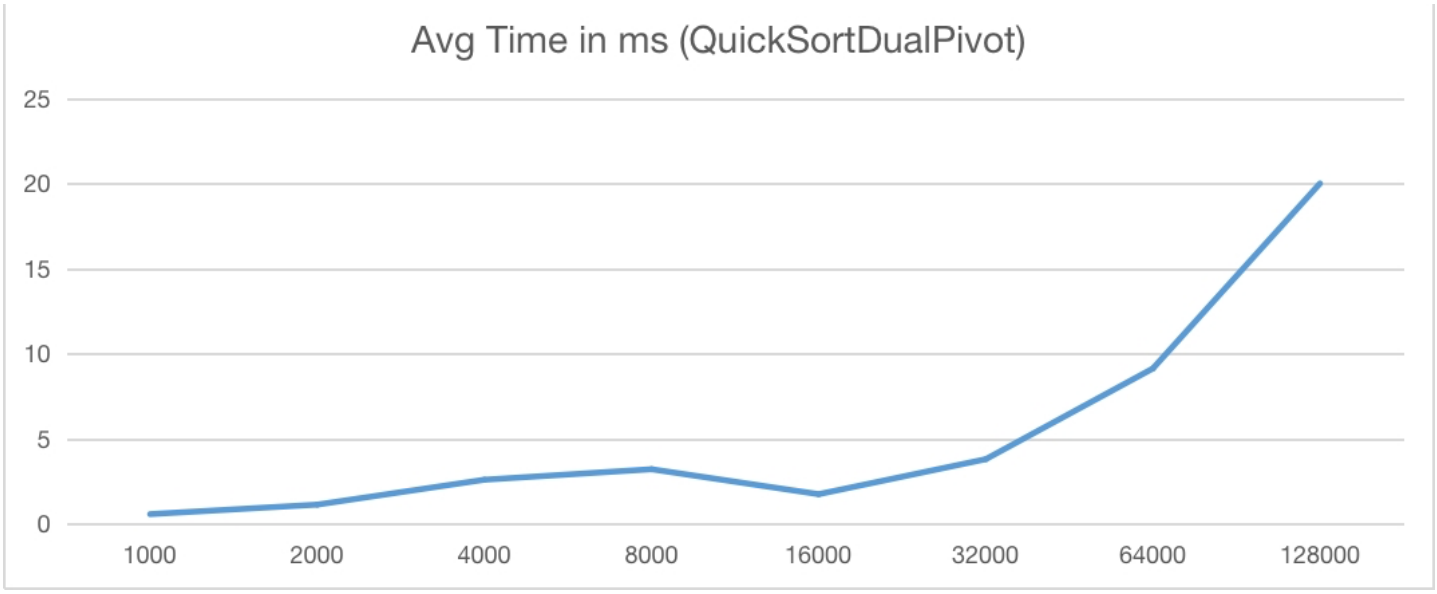
3. Evidence

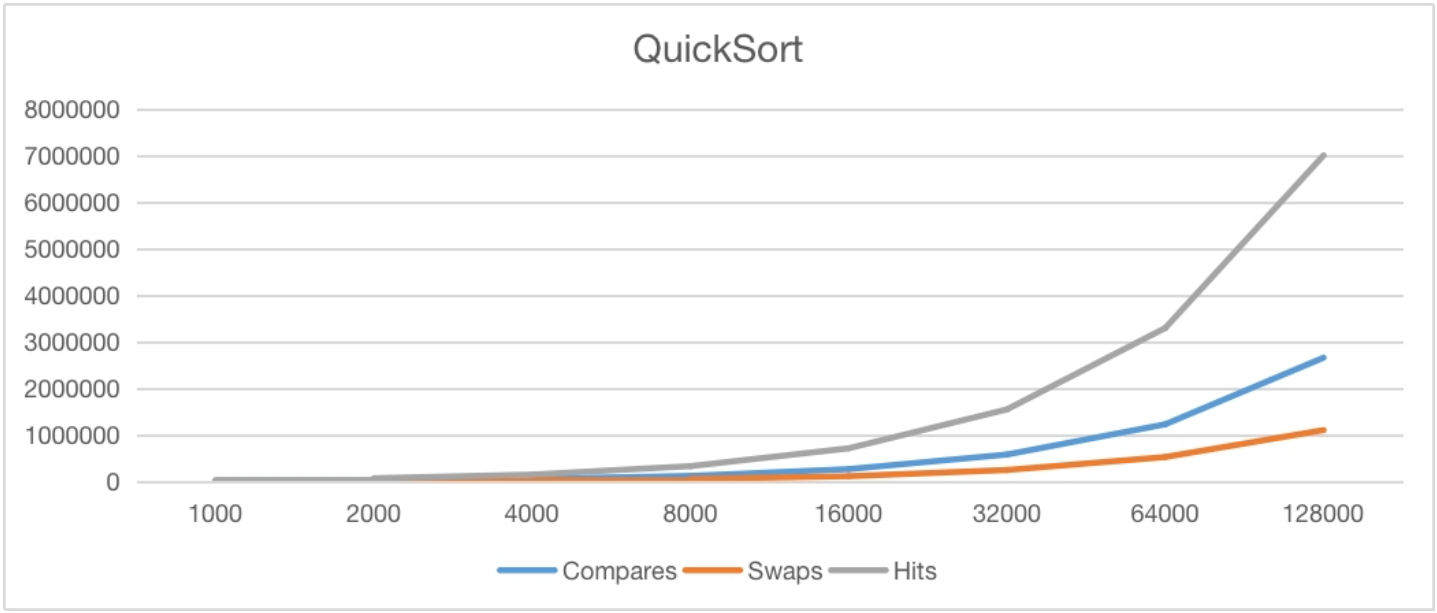


I passed all unit tests of MergeSortTest.

3.1 Quick Sort Dual Pivot

		QuickSortDualPivot		
Array Length	Compares	Swaps	Hits	Avg Time in ms (QuickSortDualPivot)
1000	10988	4869	29907	0.57894585
2000	24624	10873	69973	1.1379395
4000	55001	23924	148348	2.6001604
8000	119754	53566	329424	3.2175187
16000	265795	114056	712911	1.7499147
32000	578939	246675	1547696	3.8063854
64000	1228038	525655	3294275	9.13448135
128000	2657232	1103315	6997859	20.0055104

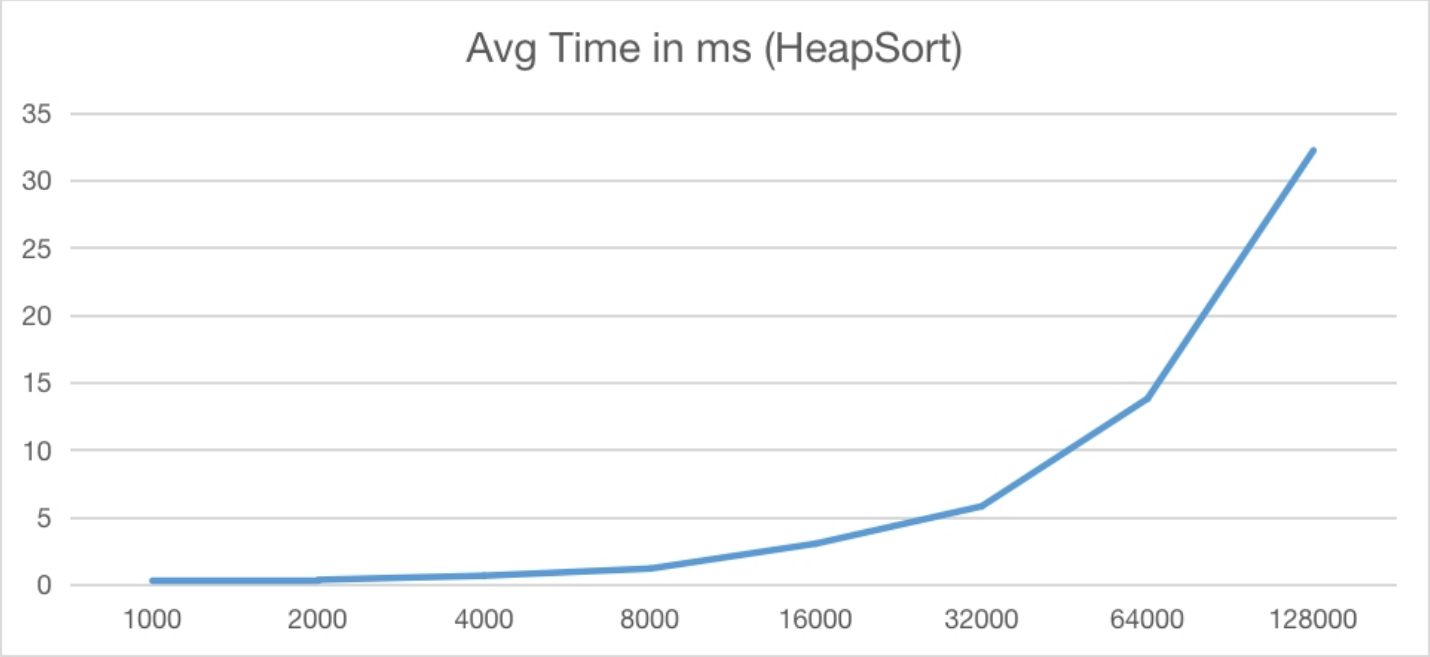


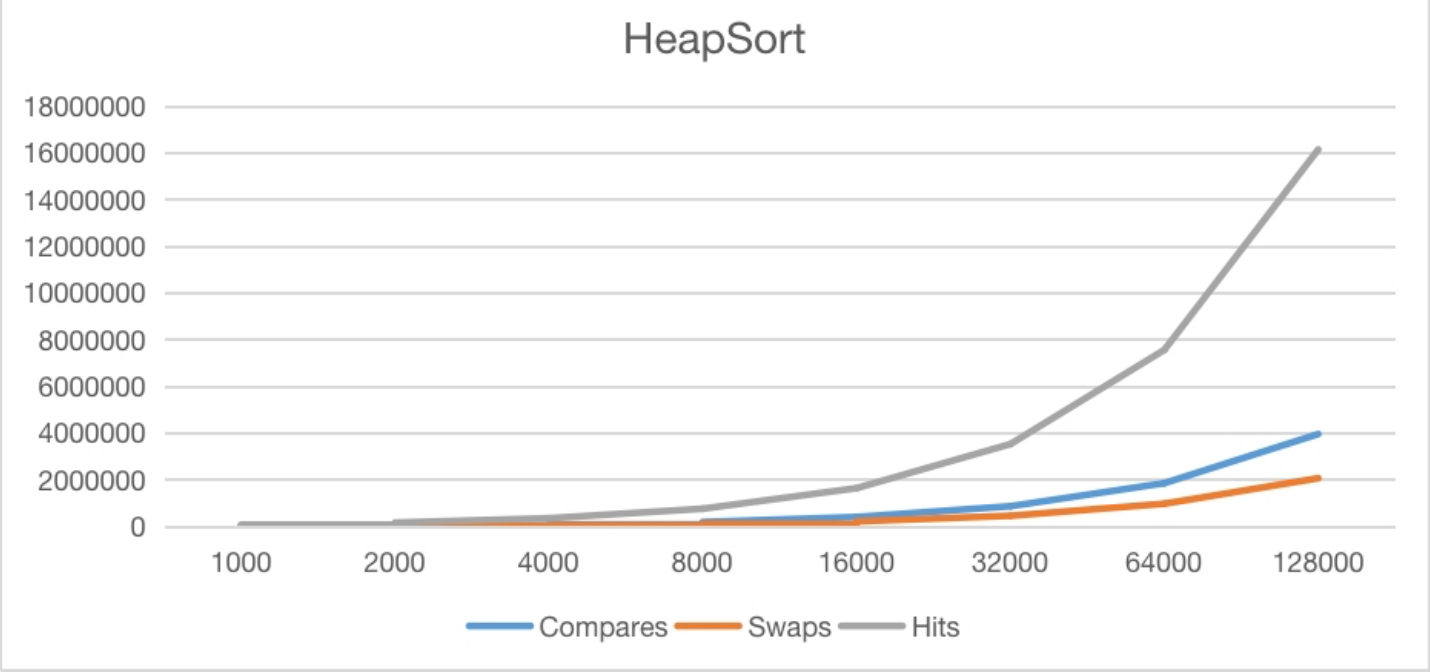


The X-axis is the array length

3.2 Heap Sort

		HeapSort		
Array Length	Compares	Swaps	Hits	Avg Time in ms (HeapSort)
1000	16840	9073	69976	0.29864795
2000	37704	20159	156047	0.36887495
4000	83415	44316	344098	0.67158115
8000	182836	96625	752174	1.19905415
16000	397683	209260	1632409	3.0522668
32000	859288	450497	3520568	5.8211104
64000	1846676	965025	7553455	13.80503755
128000	3949122	2057913	16129898	32.20697085

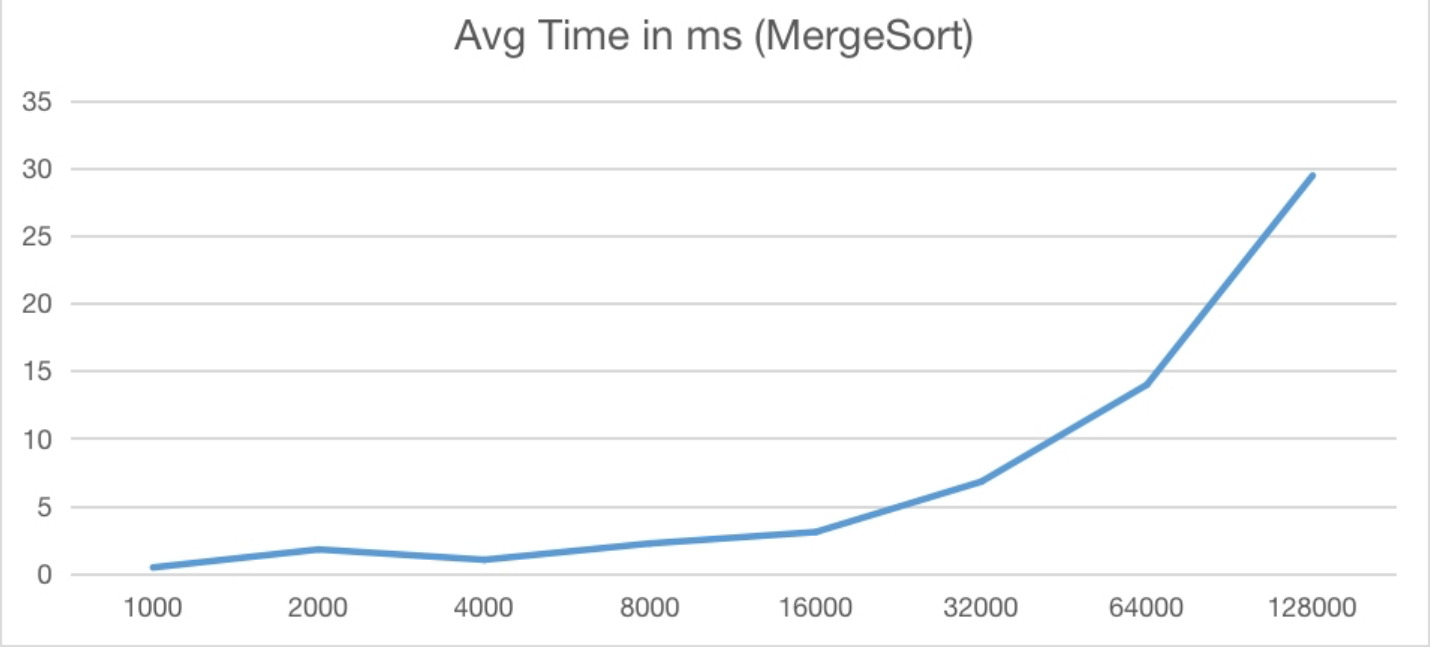


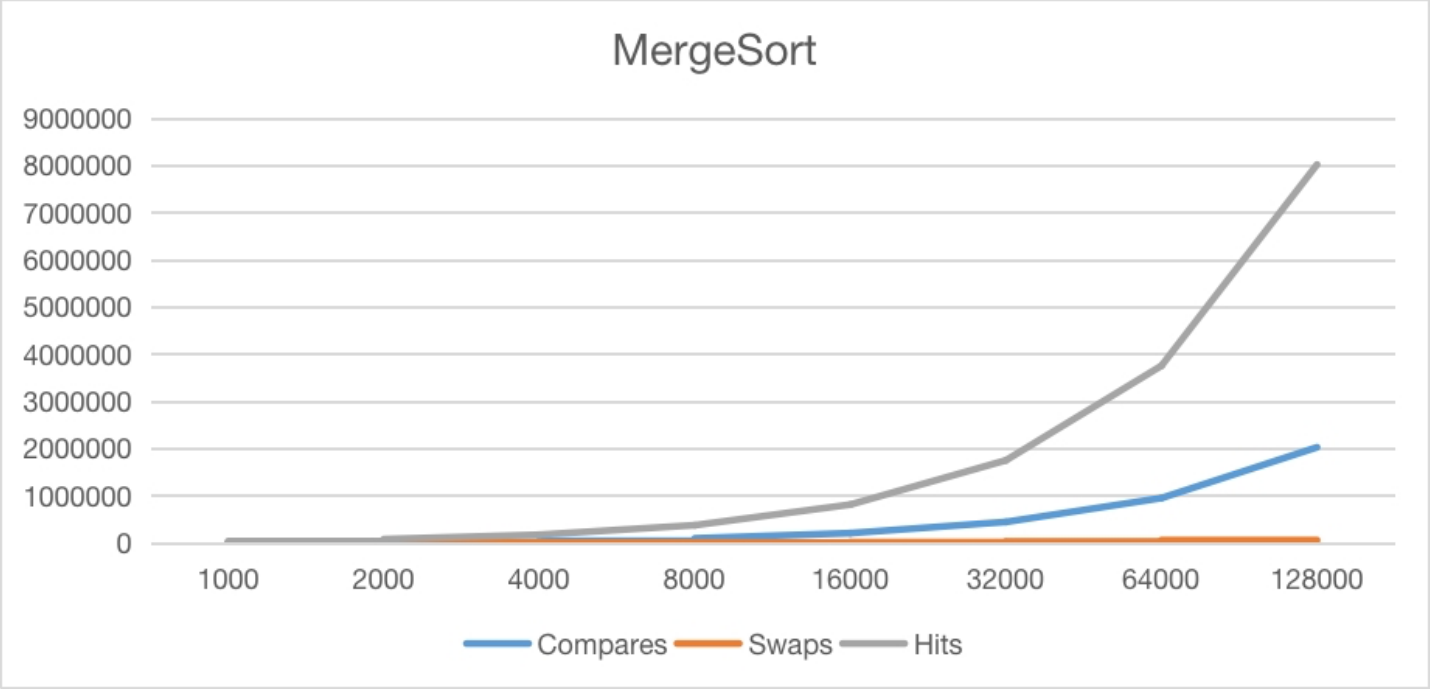


The X-axis is the array length

3.3 Merge Sort

		MergeSort		
Array Length	Compares	Swaps	Hits	Avg Time in ms (MergeSort)
1000	8817	876	34616	0.44978745
2000	19607	1736	77201	1.7821813
4000	43246	3515	170486	1.01753535
8000	94471	6989	372891	2.2252041
16000	204946	14046	809917	3.0678125
32000	441894	28065	1747778	6.7984437
64000	947814	56113	3751523	13.97622915
128000	2023603	112166	8014925	29.4481291





The X-axis is the array length

4. Conclusion

From the line chart of chapter 3, we can observe that the hits and comparisons' rising trend is similar to the time which spent by sorting method.

> In conclusion, **the best predictor of total execution time is hits and comparisons.** In this case, I used random Integer array which means for two elements, we only need to compare them once. However, as the professor mentioned that in class, for other types elements such as String, maybe we need to compare many times for them (Since there are a lot of characters in each string). Therefore, operations

> that take long time (Comparisons) or need to performed multiple times (Hits) can better predict the time of the sorting algorithm.