Program Structure and Algorithm

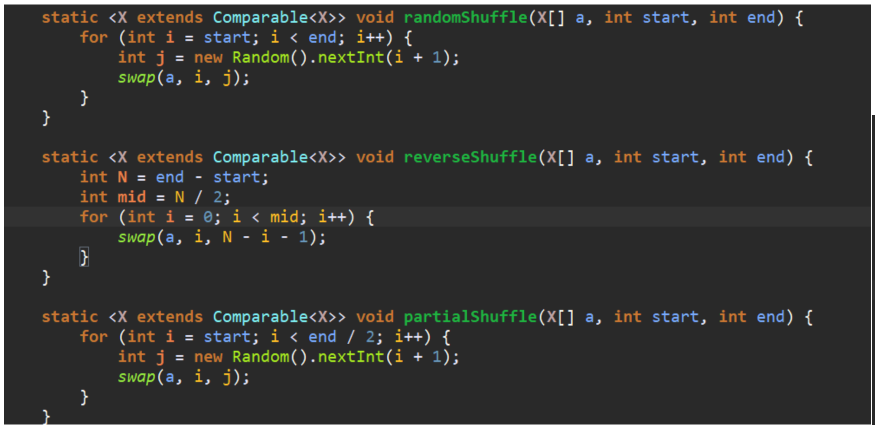
Assignment 3 Benchmark

Shouting Lyu

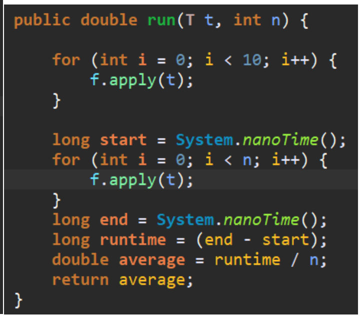
001822094

In general, I completed 3 parts of the assignment: Benchmark, InsertionSort and SelectionSort, and Measuring the running time of two sorting algorithms with random, ordered, reverse-ordered and partial-ordered arrays. In detail, I run 10 times before starting to calculate the running time of each sorting method, and then run 100 times and finally calculate the average time for 7 different values of n: 50, 100, 200, 400, 800, 1600, 3200.

The figures show the 'run()' method and three shuffle method. In the shuffle method, I used Knuth algorithm to guarantee the array is randomly shuffled, and they are in the 'Helper' class. As you can see, I implemented the run method and call the function 10 times before the real experiment, and I used 'nanoTime()' to calculate the average running time.

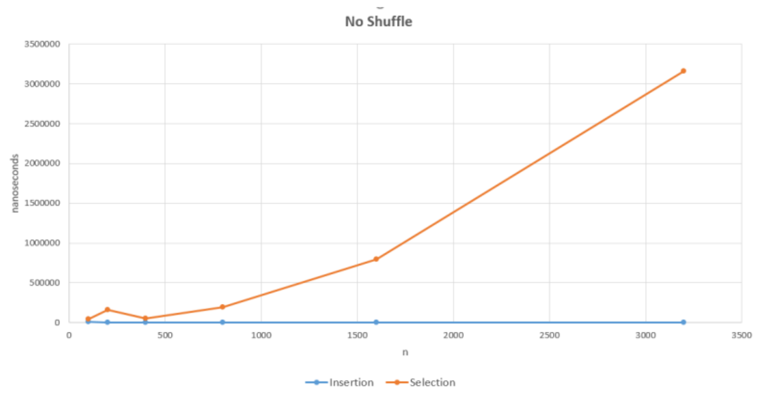


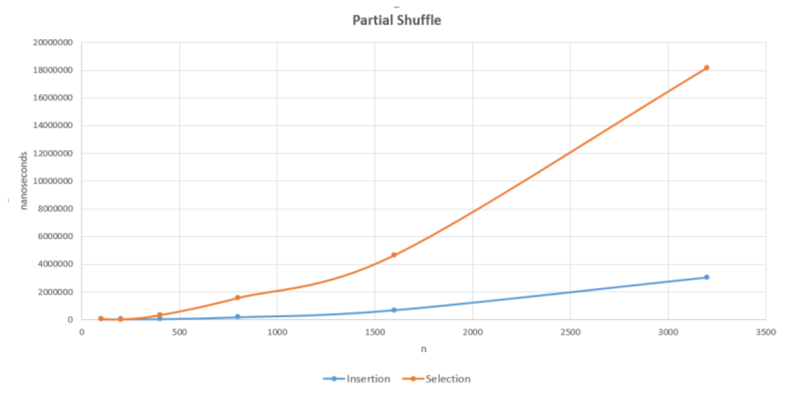
```
    static <X extends Comparable<X>> void randomShuffle(X[] a, int start, int end) {
        for (int i = start; i < end; i++) {
            int j = new Random().nextInt(i + 1);
            swap(a, i, j);
        }
    }

    static <X extends Comparable<X>> void reverseShuffle(X[] a, int start, int end) {
        int N = end - start;
        int mid = N / 2;
        for (int i = 0; i < mid; i++) {
            swap(a, i, N - i - 1);
        }
    }

    static <X extends Comparable<X>> void partialShuffle(X[] a, int start, int end) {
        for (int i = start; i < end / 2; i++) {
            int j = new Random().nextInt(i + 1);
            swap(a, i, j);
        }
    }
```

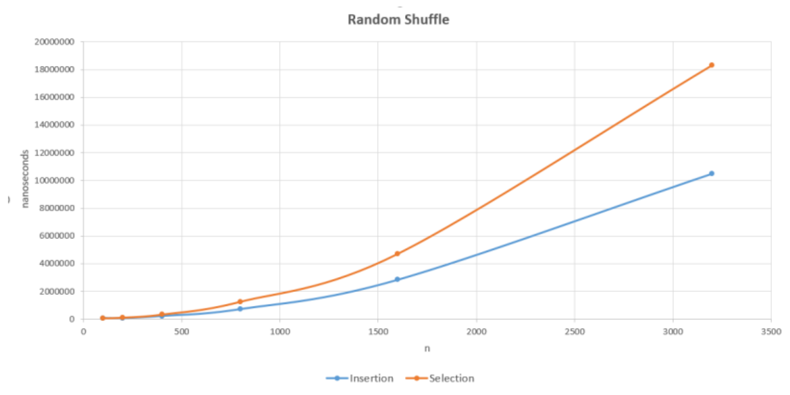


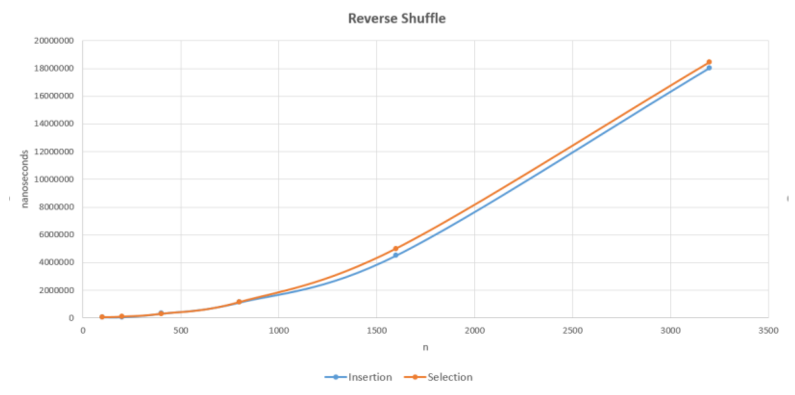
```
public double run(T t, int n) {

    for (int i = 0; i < 10; i++) {
        f.apply(t);
    }

    long start = System.nanoTime();
    for (int i = 0; i < n; i++) {
        f.apply(t);
    }
    long end = System.nanoTime();
    long runtime = (end - start);
    double average = runtime / n;
    return average;
}
```

The other four figures below show the comparison of two sorting algorithms with different inputs. As you can see, the advantages of insertion sorting are obvious with ordered and partial ordered arrays. Then, the insertion sort is still faster than selection sort, though the efficiency is not as perfect as sorting the ordered array. In the worst case, while sorting a reverse-ordered array, the insertion sort is close to the selection sort.

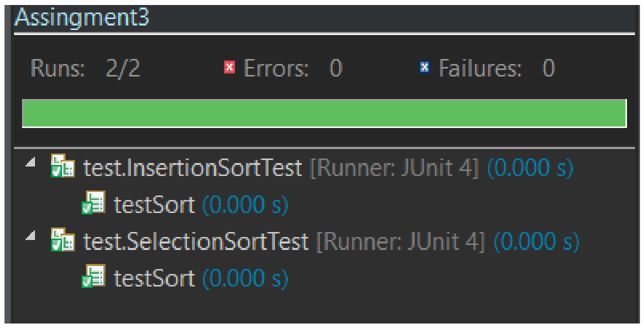

No Shuffle
nanoseconds
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
0
500
1000
1500
2000
2500
3000
3500
n
Insertion
Selection


Partial Shuffle
nanoseconds
20000000
18000000
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
0
500
1000
1500
2000
2500
3000
3500
n
Insertion
Selection


Random Shuffle
nanoseconds
20000000
18000000
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
0
500
1000
1500
2000
2500
3000
3500
n
Insertion
Selection


Reverse Shuffle
nanoseconds
20000000
18000000
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
0
500
1000
1500
2000
2500
3000
3500
n
Insertion
Selection

At last, the following figure shows my passing the unit test.



```
Assingment3
Runs:   2/2          Errors:   0          Failures:   0

test.InsertionSortTest [Runner: JUnit 4] (0.000 s)
    testSort (0.000 s)
test.SelectionSortTest [Runner: JUnit 4] (0.000 s)
    testSort (0.000 s)
```