A Study on Quicksort

Mahadi Ahmed
INDA Group 13
2016

Characteristics and Complexity

Quicksort was first invented by Tony Hoare in 1960. Many people has since studied and refined it since then. Quicksort has gained widespread adoption and fame, it is considered to be one of the top 10 algorithms of all time and has also been used in Unix as the default library sort function.
The quicksort algorithms desirable features are that it is space efficient, data can be sorted in-place and it does not require additional arrays for temporary storage. Also its best and expected performance is $O(nlogn)$. But the quicksort algorithm is also fragile and depending on the implementation of the algorithm may performe as $O(n^2)$ algorithm. The best case scenario is when each pivot choice divides the array in half, consequently input data that is randomised unique elements is very important. The worst case scenario is when each pivot choices divides the array into one part that is n-1 and the other is 1. Problems often manifest when data is partially sorted or fully sorted, data is dominated by repeating elements and when Pivot choice is min/max value of data.

Variations of Quicksort

Four variation of quicksort was implemented.

QuicksortFixedPivot

QuicksortFixedPivot which is the most basic and used a fixed pivot. The choosen pivot is the last element of the array or sub-array. This implementation is fragile and for large problem sizes assumes the best case scenarios are true otherwise it breaks.

```
private int partition(int[] v, int low, int high) {
    this.pivot = v[high]; // last item
    int i = low - 1; // i is lower wall j is higher wall
    for (int j = low; j <= high - 1; j++) {
      if (v[j] <= this.pivot) {
        i += 1;
        int temp = v[i];
        v[i] = v[j];
        v[j] = temp;
      }
    }
    int temp = v[i + 1];
    v[i + 1] = v[high];
    v[high] = temp;

    return i + 1;
  }
```

QuicksortRandomPivot

Instead of always using the last element as the pivot we randomly choose an element where all elements in the array or sub-array is equaly likely to be choosen as pivot. This is done with the expectation that the split of the array will be reasonably well balanced on avarage.

```
private int randPartition(int[] v, int low, int high) {
    int n = low + rand.nextInt(high - low + 1);
    int temp = v[high];
    v[high] = v[n];
    v[n] = temp;
    return partition(v, low, high);
  }
```

QuicksortFixedPivotInsertion

This variation of quicksort has a fixed pivot but a cut-off to insertion sort when the array contains at most k elements.

```
private void insertionSort(int[] v, int low, int high) {
    for (int i = low + 1; i <= high; i++) {
      int j = i;
      while (j > low && v[j - 1] > v[j]) {
        int temp = v[j];
        v[j] = v[j - 1];
        v[j - 1] = temp;
        j -= 1;
      }
    }
  }
```

QuicksortRandomPivotInsertion

This variation of quicksort has a randomly choosen pivot and a cut-off to insertion sort. My cut-off strategy was very simple. During my test i saw significant improvement with large k values ($k > 200$) when the data was already sorted or all the elements in the array was equal. But in all the other cases the large values either was insignificant or made it slower. Thus i settled on the arbitrary value of 6

Methodology

The range of the numbers that could be generated was the same as the problem size.

```
d = new Data(n, n, Data.Order.RANDOM);
```

For every problem size i ran sort() 51 times and timed 50 of them, excluding the first call to sort(). Avarage, max and min time for these 50 sorts was recorded. This was done 3 times for every problem set and then one of them was randomly chosen to be presented as the result. The problem sizes were: 100, 1000, 10000, 100000, 1000000. The set of data used to test the algorithms was: Random, Sorted, Reversed and Equal, where all the values in the equal data set was 1.

Results

All values in the tables are avarage runtimes of 50 sort().

Tables & Graphs

Table 1: Test 1: Random Data

Problem size	InsertionSort	QsortFixedPivot	QsortRandomPivot	QsortFixedPivotInsertion	QsortRandomPivotInsertion	Arrays.sort †
100	0.05 ms	0.019 ms	0.036 ms	0.020 ms	0.054 ms	0.025 ms
1000	0.278 ms	0.152 ms	0.218 ms	0.109 ms	0.251 ms	0.126 ms
10000	17,6 ms	0.984 ms	1.263 ms	0.829 ms	1.237 ms	1.24 ms
100000	1.86 s	9.74 ms	10.92 ms	9.3 ms	9.52 ms	7.09 ms
1000000	over 10 sec	116 ms	124 ms	111.4 ms	111.6 ms	79.9 ms

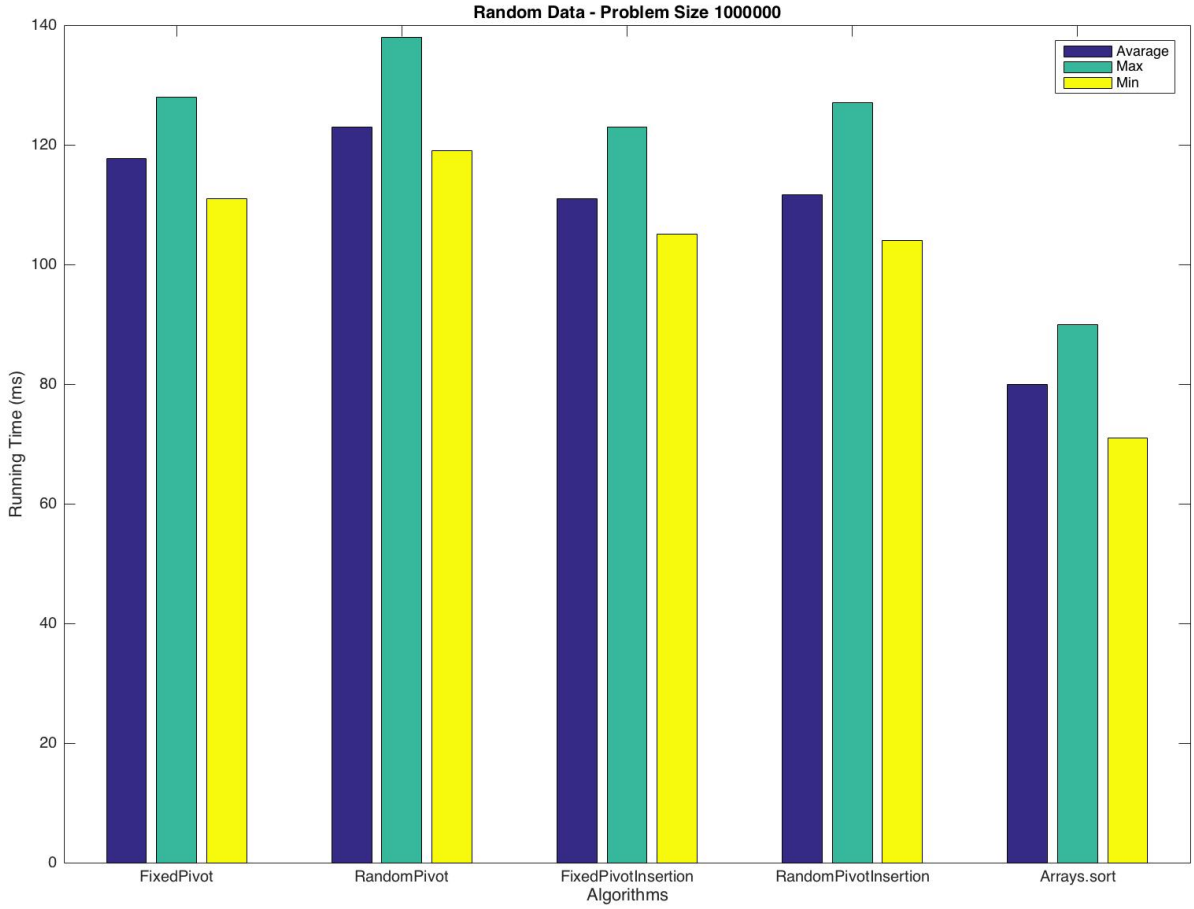


Table 2: Test 2: Sorted Data

Problem size	InsertionSort	QsortFixedPivot	QsortRandomPivot	QsortFixedPivotInsertion	QsortRandomPivotInsertion	Arrays.sort †
100	0.004 ms	0.042 ms	0.039 ms	0.047 ms	0.024 ms	0.005 ms
1000	0.039 ms	0.47 ms	0.193 ms	0.478 ms	0.115 ms	0.053 ms
10000	0.117 ms	StackOverflow	0.94 ms	StackOverflow	0.656 ms	0.175 ms
100000	0.31 ms	-	6,36 ms	-	5.22 ms	0.303 ms
1000000	1.33 ms	-	70.36 ms	-	57.88 ms	2.26 ms

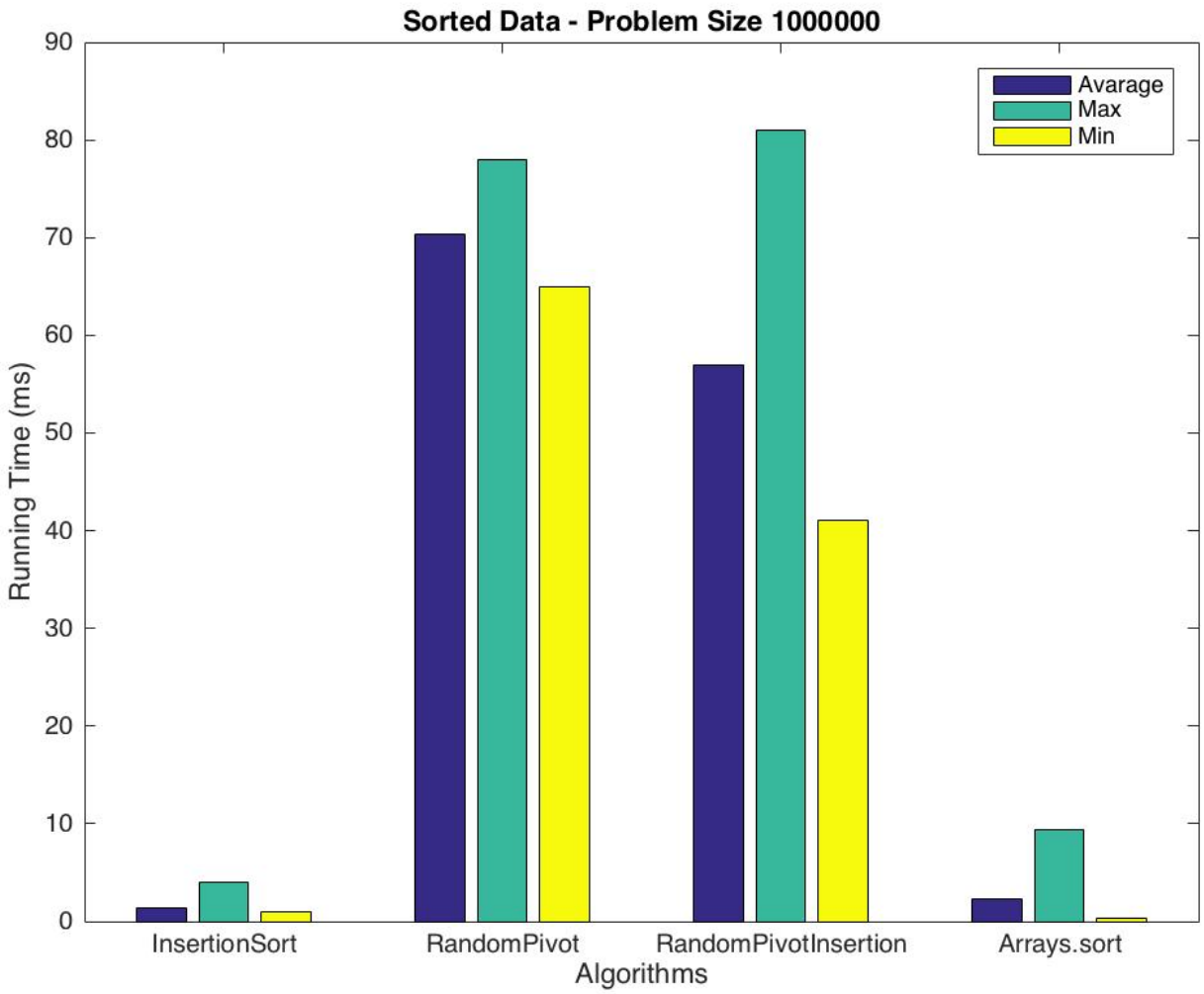


Table 3: Test 3: Reversed Data

Problem size	InsertionSort	QsortFixedPivot	QsortRandomPivot	QsortFixedPivotInsertion	QsortRandomPivotInsertion	Arrays.sort †
100	0.090 ms	0.039 ms	0.037 ms	0.043 ms	0.028 ms	0.011 ms
1000	0.552 ms	0.634 ms	0.202 ms	0.584 ms	0.124 ms	0.076 ms
10000	38.42 ms	32.04 ms	1.15 ms	34.522 ms	0.772 ms	0.317 ms
100000	3.6 sec	StackOverflow	7.06 ms	StackOverflow	5.758 ms	0.508 ms
1000000	over 10 sec	-	74.96 ms	-	58.92 ms	4.09 ms

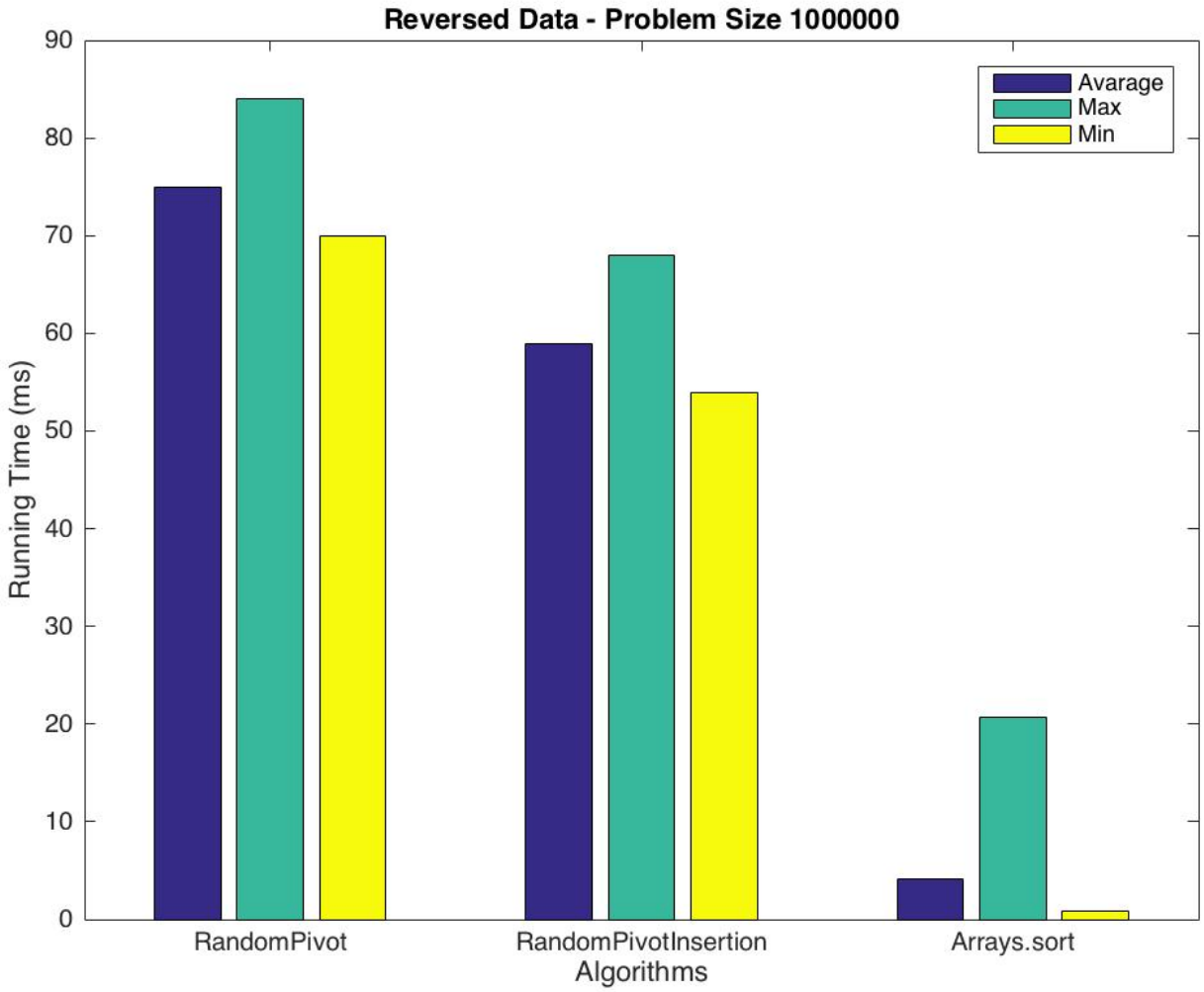
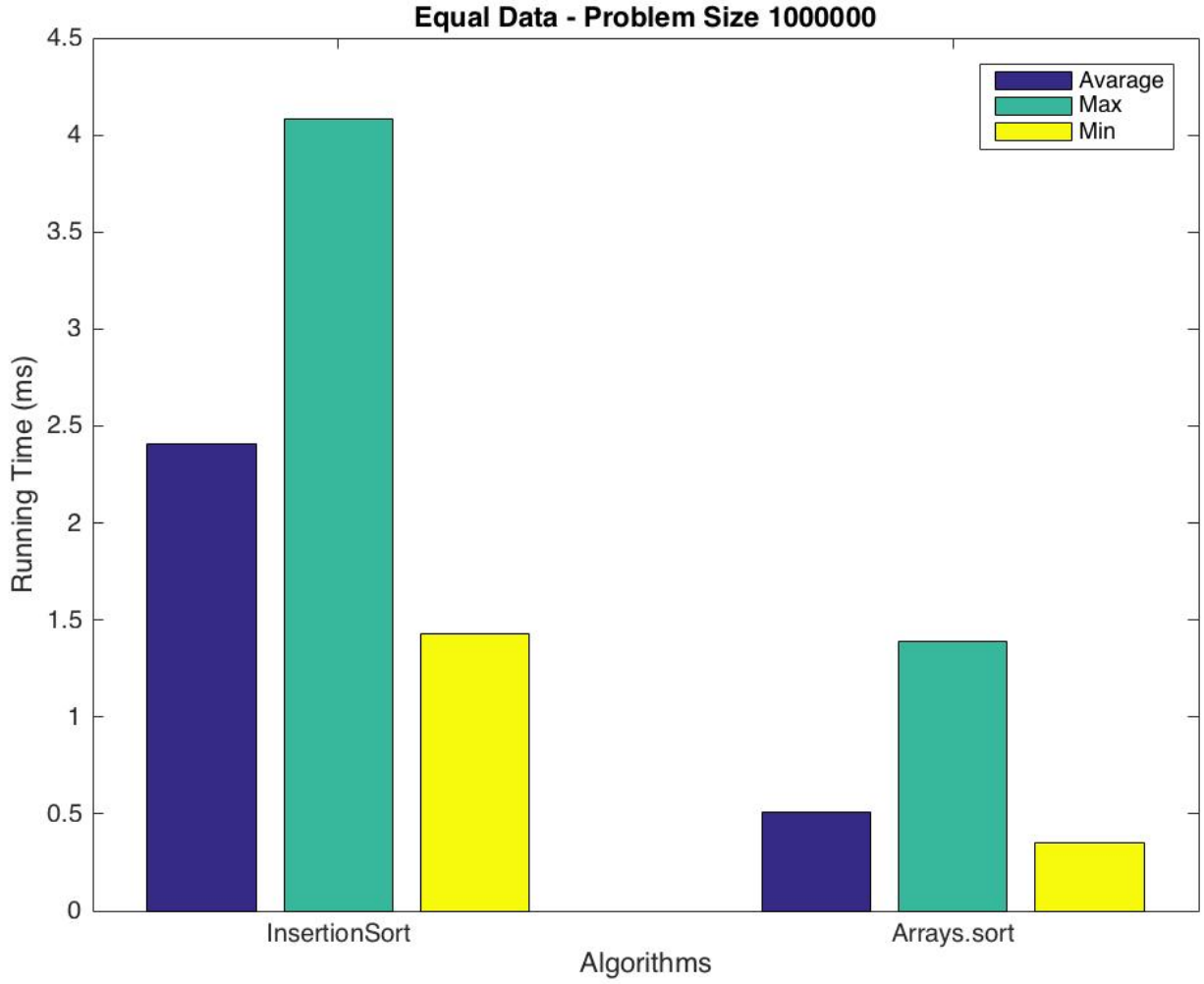


Table 4: Test 4: Equal Data

Problem size	InsertionSort	QsortFixedPivot	QsortRandomPivot	QsortFixedPivotInsertion	QsortRandomPivotInsertion	Arrays.sort †
100	0.006 ms	0.050 ms	0.057 ms	0.046 ms	0.064 ms	0.003 ms
1000	0.037 ms	0.423 ms	0.485 ms	0.407 ms	0.428 ms	0.020 ms
10000	0.233 ms	StackOverflow	StackOverflow	StackOverflow	StackOverflow	0.118 ms
100000	0.422	-	-	-	-	0.305 ms
1000000	2.41 ms	-	-	-	-	0.512 ms



Discussion

In the Random data set with n=1000000 the avarage run time was generally very close to the best run time. In my test it was approximately a performance difference of 4-6%.
Arrays.sort was in every regard better than my implementations. In terms of performance the QuicksortRandomPivotInsertion was closest to Arrays.sort ("closest" as in how close the earth is to the closest galaxy).
With Sorted data set the two RandomPivot implementations had very large differences in the best and worst times. Over a large enough test this difference would probably even out.
The most surprising thing was how explicit the differences in times with regards to problemsize and data set was. With sorted data the FixedPivot variations couldn't handle n=10000 but the RandomPivot variations handled them well compared to that. Also Arrays.sort handled the equal data set surprisingly well. Insertion sort behaved as i expected with the random data set.