A Study of Quicksort

Sabrina Chowdhury
INDA Group 5
2018

Characteristics and Complexity

The QuickSort algorithm, developed and published by Tony Hoare in 1961, is a sorting algorithm for placing the elements in an array in order. On average the algorithm takes O(n log n) comparisons to sort *n* items and its worst-case running time is O(n^2). The algorithm works in the way that it recursively divides an array into smaller sub-arrays until there are only sub-arrays filled with one element left. Quicksort then puts them back together into a sorted array.

The Quicksort Operations

1. Pivot : An element in the array is chosen as the pivot element. The choice of the pivot can vary in many different ways and can affect the effectiveness of the algorithm.
2. Partition : Move the pivot until it is in a position in the array that makes all elements on one side of it larger and all elements on the other side smaller than the pivot. Make one subarray for elements smaller than the pivot and one subarray for elements larger than the pivot.
3. Recursively apply the above step to the two sub-arrays.
4. The base case of the recursion are sub-arrays of size zero or one, which are by definition in order.

Variations of Quicksort

Fixed Pivot

- The pivot is always chosen as the middle element. This is because choosing an element near the ends would cause a worst-case scenario for sorted and reversely sorted data.

Fixed Pivot Insertion

- Same as the Fixed Pivot algorithm, however, if the size of the array is less than or equal to a certain number, the insertion sort algorithm is used instead for efficiency.

Random Pivot

- A random pivot is generated in the beginning.

Random Pivot Insertion

- The pivot element is randomly chosen,however, if the size of the array is less than or equal to a certain number, the insertion sort algorithm is used instead for efficiency.

The same partition routine was used for all Quicksort implementations and was inherited in all classes from the QuicksortFixedPivot class. When the array size was smaller than or equal to 16 insertion sort was used. This number was found after testing the algorithm with different input-sizes starting with 100 and then narrowing it down to a number between 10 and 20.

```
public int partition(int arr[], int left, int right)
    {
        //Assign leftmost and rightmost element to placeholders i and j
        int i = left;
        int j = right;

        //Introduce a temporary variable which will be useful when we swap
        //elements in the array
        int tmp;

        //We generate a pivot
        int pivot = getPivot(arr,left,right);

        //While the leftmost element is smaller than the rightmost element
        while (i <= j) {

            //While the leftmost element is smaller than the pivot
            while (arr[i] < pivot)

                //Shift the leftmost value one step
                i++;

            while (arr[j] > pivot)

                j--;

            if (i <= j) {
                tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
                i++;
                j--;
            }

        }
        return i;

    }
```

Methodology

The Test Classes : ensuring the accuracy of the different algorithms

In the main test class I created an array containing the different Quicksort algorithms and another multidimensional array containing several different arrays. These arrays would be fed to the different quicksort algorithms. For each test-case in the test class I fill the first array (at index 0) in the multidimensional array with some elements so that it fits my test-case, e.g. an even array with an even amount of elements. This first array is then copied to the rest of the arrays in the multidimensional array so that all quicksort algorithms have their own array to sort. To test that all quicksort algorithms have sorted their arrays correctly I compare their results with the results of the first array in the multidimensional array - an array that has been sorted by an algorithm that has passed Kattis. This can also be replaced by Java's built in sort algorithm, which will also output a correct result.

TimingExample Class : measuring the runtime for different algorithms

The TimingExample class measures the running time for different algorithms when given different sizes of input. Results affected by JVM warmup are discarded by using each sorting implementation 100 times. As Insertions Sort is included this will take significantly longer than if Insertion Sort was not part of the test, and so this number can be decreased, however, for this report the implementations were run 100 times. Each test was initiated by filling an array with the right amount of the right elements (e.g. 1000 elements in random order or a reversely sorted array of size 1000000). This array was then copied so that five other arrays looked exactly the same. With six identical arrays the six different implementations of the algorithm can now in one go sort their own array and their times can be measured and compared. This method was used for all four tests(equal, random, sorted, reversely sorted).

Results

Table 1: Running time for different Quicksort implementations when given arrays containing random data

Problem Size	Running Time (ns)					
	InsertionSort	Fixed Pivot	Fixed Pivot Insertion	Random Pivot	Random Pivot Insertion	Arrays.sort †
100	338	2207	2180	6211	6806	926
1000	7748	23899	20218	69939	74646	1020
10000	664140	244927	349374	802027	688523	11003
100000	60182313	2678457	2907893	9046946	7190013	101055
1000000	6029881724	28890862	29910071	91907662	98112232	1296268

Table 2 : Running time for different Quicksort implementations when given arrays containing sorted data

Problem Size	Running Time (ns)					
	Insertion Sort	Fixed Pivot	Fixed Pivot Insertion	Random Pivot	Random Pivot Insertion	Arrays.sort †
100	390	3014	3014	8536	7816	3816

1000	1876	23207	19342	65206	75112	12367
10000	15801	216217	237208	645822	707853	36157
100000	142165	2565688	2472004	7074382	6953823	28973
1000000	1532005	32024019	29766731	75054123	85809075	388662

Table 3 : Running time for different Quicksort implementations when given arrays containing reversely sorted data

Problem Size	Running Time (ns)					
	InsertionSort	Fixed Pivot	Fixed Pivot Insertion	Random Pivot	Random Pivot Insertion	Arrays.sort †
100	329	1668	1641	5809	5728	419
1000	14171	19878	20884	62167	67748	13532
10000	1458748	303131	284954	696498	665766	73629
100000	169226900	15247078	8157707	17645059	13607758	2449718
1000000	13772940714	37978575	37270927	79655586	75404072	427819

Table 4 : Running time for different Quicksort implementations when given arrays containing equal data

Problem Size	Running Time (ns)					
	InsertionSort	Fixed Pivot	Fixed Pivot Insertion	Random Pivot	Random Pivot Insertion	Arrays.sort †
100	298	3774	2684	3500	3620	107185
1000	1517	29430	29401	43082	32235	27753
10000	20541	342814	344390	423633	375847	50734
100000	174297	4014899	4781394	4760084	4470001	396259
1000000	5679190	41385088	44198307	46299125	43675864	1266708

Figure 1 : Running time for different Quicksort implementations when given arrays containing equal data

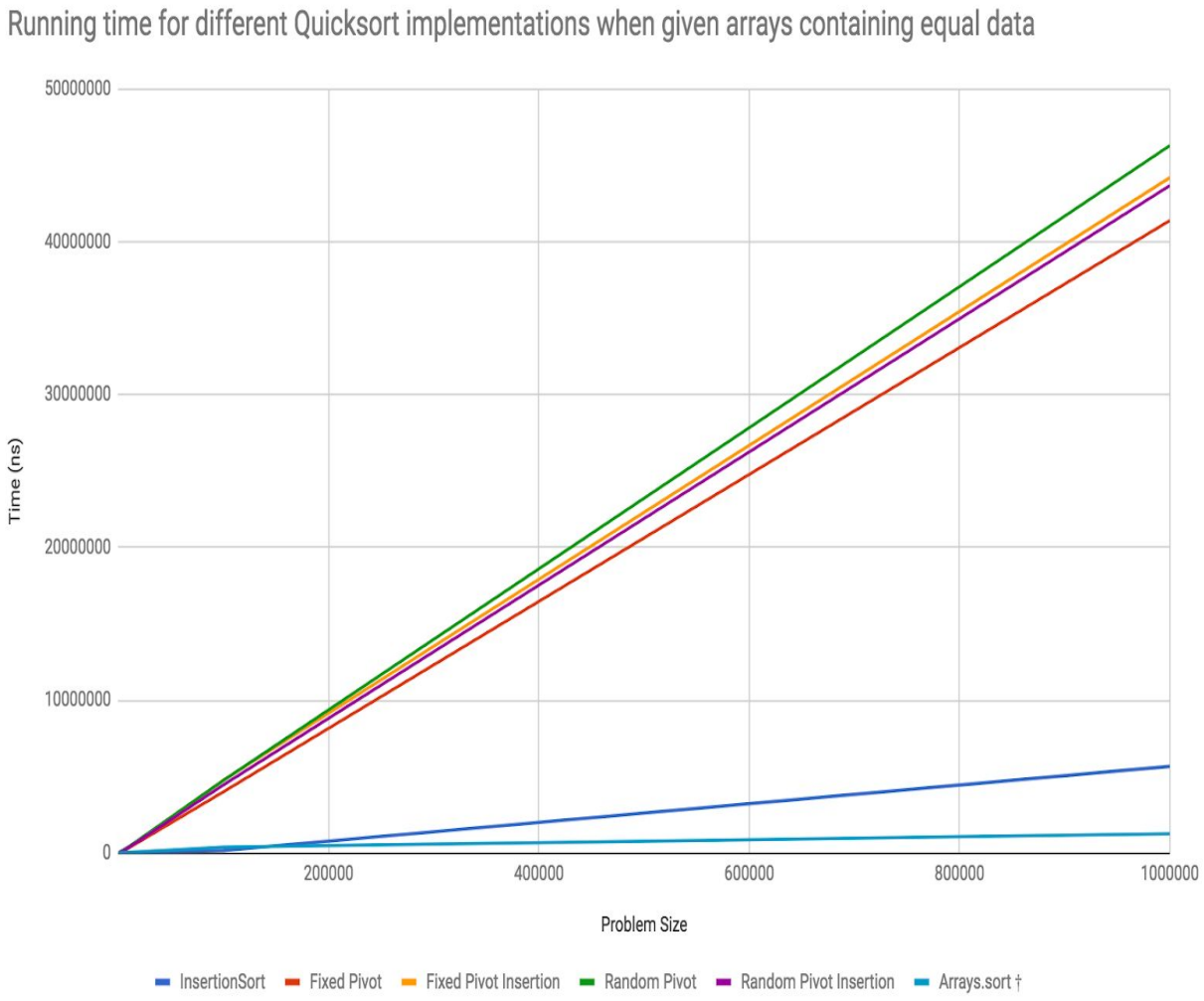


Figure 2 : Running time for different Quicksort implementations when given arrays containing sorted

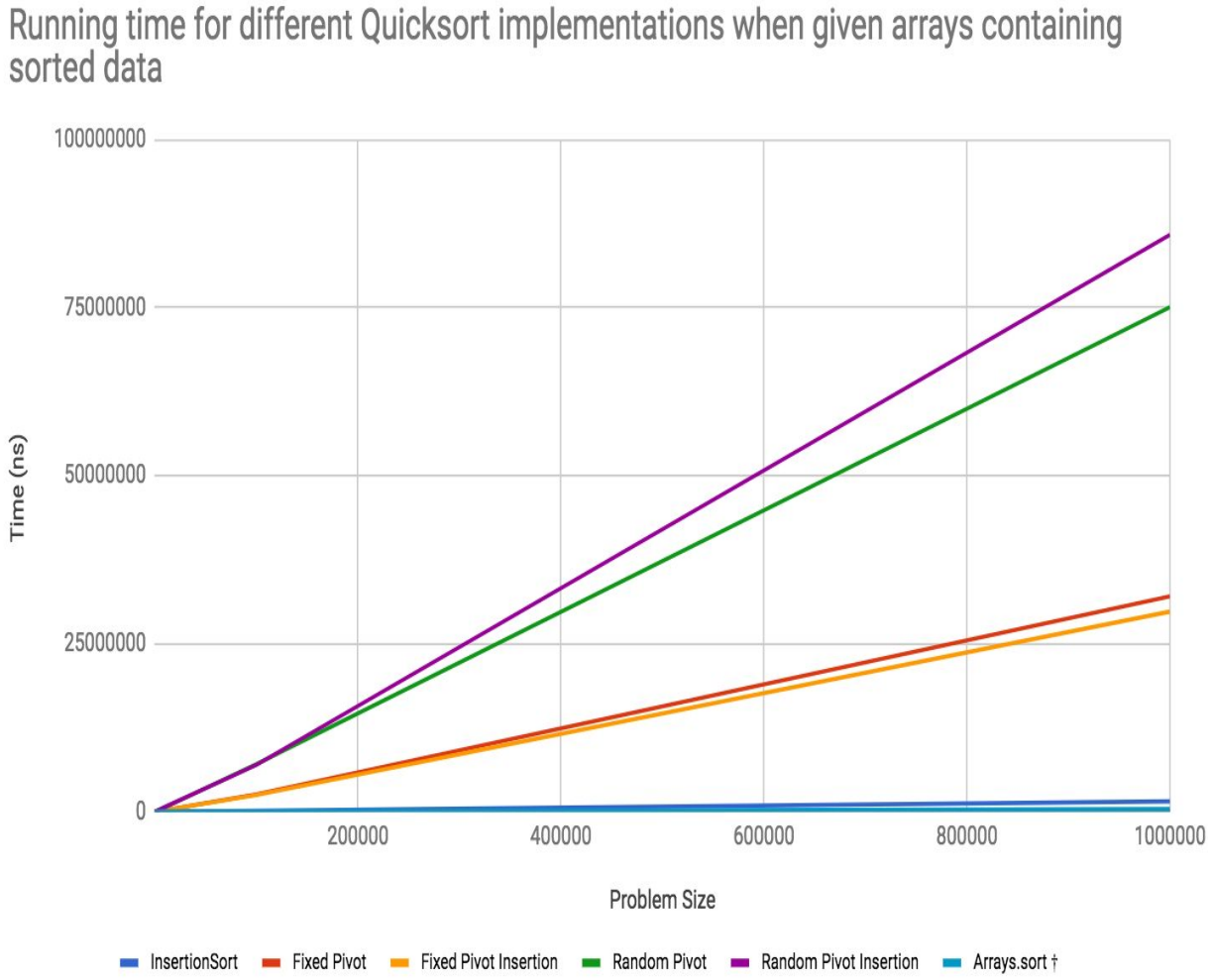


Figure 3 : Running time for different Quicksort implementations when given arrays containing reversely data (insertion sort is not included as it was significantly slower than all the other implementations)

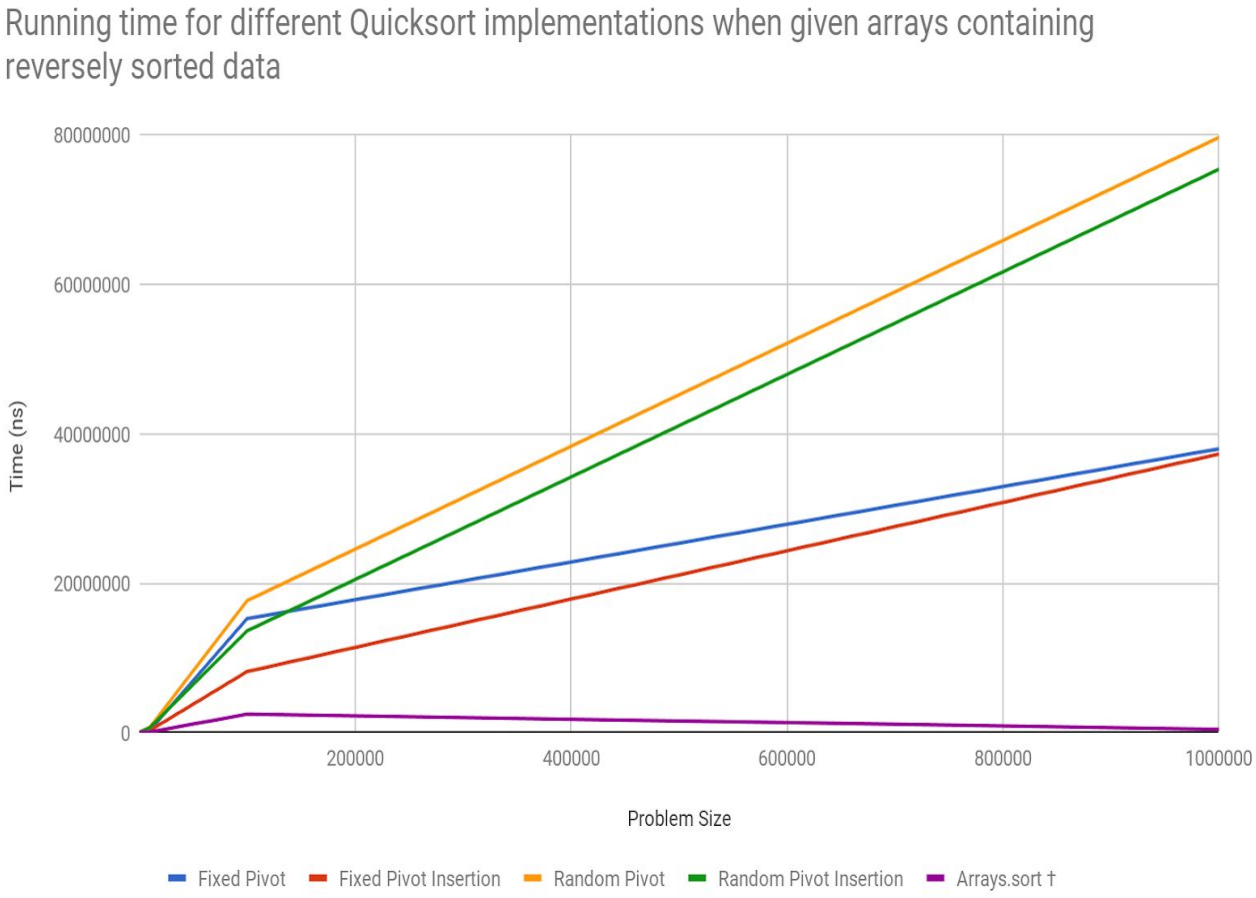
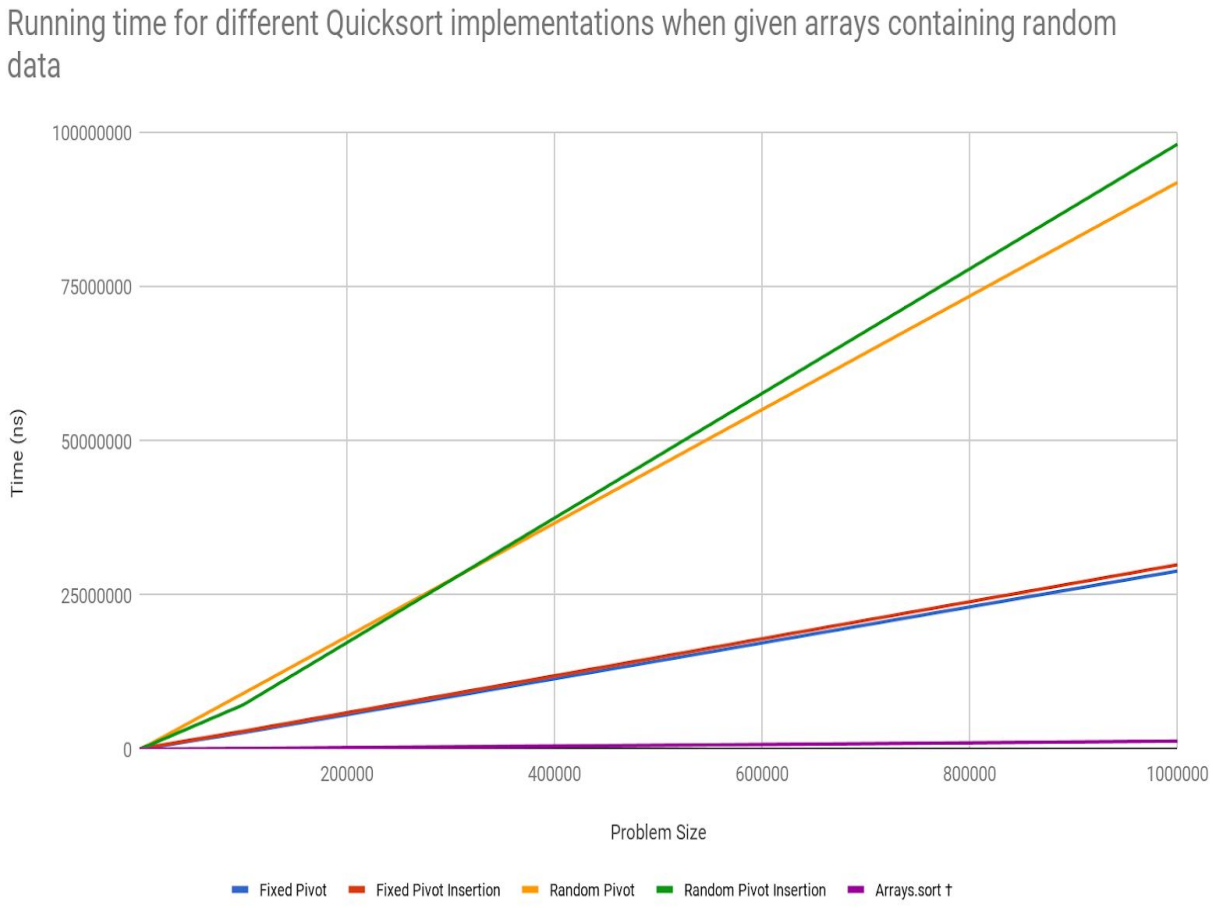


Figure 4 : Running time for different Quicksort implementations when given arrays containing random data (insertion sort is not included as it was significantly slower than all the other implementations)



Discussion

The Fixed Pivot and Fixed Pivot with Insertion Sort implementations were the closest in regards to running times in comparison to Arrays.sort when it came testing the random data and reversely sorted data. Insertion sort was, however, closer to Arrays.sort when it came to sorting arrays containing equal data and sorted data. Based on the data, Insertion Sort is extremely slow compared to the Quicksort implementations on all tests except when it comes to the equal data and sorted data test. In the equal data case,this is because insertion sort will visit n elements giving it a time complexity of O(n) while this scenario will give e.g. the Fixed Pivot implementation a time complexity of O(n^2) - although no swaps will be made, n recursive calls calling for comparison will be made.