Tilda Jansson, INDA Group Regular Year 2021

A Study of Quicksort

1. Characteristics and Complexity

Quicksort is a classic, efficient and divide-and-conquer sorting algorithm. The quicksort algorithm was developed by a British computer scientist Tony Hoare in 1959 and it was published in 1961 ('Quicksort', (2021)). To sort an array of elements the algorithm partitions the array into two parts by comparing the elements to a pivot element and placing smaller elements on the left and larger elements on the right, and then recursively sorts the two subarrays. As soon as the deepest recursion level is reached for all partitions and partial partitions, the entire list is sorted.

Using the Master Theorem, it's easy to see that quicksort runs in linearithmic time, *O(n log n)* in the best case, and that's what's expected for random data. The average-case time complexity of Quicksort is also *O(n log n)*. Quicksort achieves optimal performance if we always divide the arrays and subarrays into two partitions of equal size. The two way partitioning function has linear complexity.

The worst case occurs if the pivot element is always the smallest or largest element of the array and subarrays. The (sub)array is then no longer divided into two partitions of approximately equal size. E.g. if we always choose the last element as the pivot element and the input is already sorted or nearly sorted, or sorted in reverse order. The worst-case time complexity is *$O(n^2)$* (Sehgal 2018).

Quicksort is space efficient. E.g. when sorting an array of integers we only need to use the same array those integers are contained within. Quicksort is not stable, i.e. the relative order of equal sort elements is not preserved (Sehgal 2018).

2. Variations of Quicksort

The key process in Quicksort is the partition. Instead of using a two way partition these implementations all use a three way partition. The 3-way partition function partitions the array/subarrays into three parts: one part that contains smaller elements than the pivot, the second part that contains elements equal to the pivot, and the third part contains all elements that are larger than the pivot value. This method is very useful when the input data contains a lot of duplicate values.

```
/**
 * 3-way partitioning an array of ints.
 *
 * @param array  An array of ints to be partitioned.
```

```
  * @param from   Position of the first element in the array/subarray
  * @param to     Position of the last element in the array/subarray
  * @param pivot  index of value to compare with
  *
  * @return breaks   array including index breakpoints
  */
 public int[] threeWayPartition(int[] arr, int from, int to, int pivot){
        int pivotV = arr[pivot];
        int[] breaks = new int[2];
        int mid = from;

        while (mid <= to){
            if(arr[mid] > pivotV){
                swop(arr, mid, to);
                to--;
            }
            else if(arr[mid] < pivotV){
                swop(arr, from, mid);
                from++;
                mid++;
            }
            else{
                mid++;
            }
        }
        breaks[0] = from-1;
        breaks[1] = mid;
        return breaks;
    }
```

While a two-way partition looks like this:

```
    /**
     * Two-way partitioning an array of ints.
     *
     * @param array  An array of ints.
     * @param from   Position of the first element in the array/subarray
     * @param to     Position of the last element in the array/subarray
     *
     * @return counter Position of the pivot element after partitioning
     */

    public int partition(int[] arr, int from, int to)
    {
        int value = arr[to]; //Pivot point
        int counter = from -1;
        for(int i = from; i<to; i++){
            if(arr[i]<= value){
                counter++;
                swop(arr, counter, i);
            }
        }
        swop(arr, counter+1, to);
        return counter+1;
    }
```

QuicksortFixedPivot

This is the most simple version of Quicksort of the four implementations. It uses 3-way partition and the *pivot*, i.e. the element to which the rest of the elements are compared, is always the last element in the array.

QuicksortRandomPivot

The only difference between this implementation and *QuicksortFixedPivot* is that instead of always choosing the last element in the array as the pivot, it picks the pivot using the *median-of-three* method. Then it swaps the chosen pivot with the last element in the array before it calls the *threeWayPartition*-function.

The median-of-three method selects three random elements from the array, or subarray, and then chooses the median of these three elements as the pivot. The cost for taking three random samples will be *O(1)*. The median-of-three method improves Quicksort in a way so that the worst case is much more unlikely to occur.

QuicksortFixedPivotInsertion

This implementation is basically the same as *QuicksortFixedPivot*, except for the fact that it uses *Insertion sort* for sorting (sub)arrays of certain smaller sizes. This hybrid of Quicksort and Insertion sort has a cut-off to Insertion sort when there are 50 or less elements in the (sub)array, and the (sub)array is not further partitioned.

QuicksortRandomPivotInsertion

This implementation is also a combination of Quicksort and Insertion sort. It picks a random pivot using the *median-of-three* method, and then sorts the array and subarrays using the 3-way partition. It has a cut-off to insertion sort when there are 50 or less elements in the (sub)array, stopping the recursion.

cut-off strategy
For very small arrays, Insertion sort is faster than Quicksort. In order to decide the optimal threshold for switching to Insertion Sort I created a runtime test to measure runtimes for *QuicksortFixedPivotInsertion* and *QuicksortRandomPivotInsertion* using various thresholds for switching to Insertion Sort. The test measures the time needed to sort 1 million elements at different thresholds for switching to Insertion Sort. The test program tests for all thresholds between 0 and 120. To minimize the affect on the result from the JVM warmup, the test program runs 50 times, but only takes the last 20 runs into account when calculating the result.

For each of the 20 runs taken into account, it appends the *best* threshold to two separate arrays, one for *QuicksortFixedPivotInsertion* and one *QuicksortRandomPivotInsertion*. Then it calculates the average *optimal threshold* for each algorithm. After running the program a couple of times I was able to draw the conclusion that an optimal cut-off to Insertion Sort would be around 50.

3. Methodology
The performances of the sorting algorithms were studied by using runtime tests for these algorithms for sorting arrays of integers that were generated randomly, already sorted, sorted in reverse order and equal. The experiments were conducted on a computer with an Dual-Core Intel Core i3 processor with a speed of 1,1 GHz, and 8GB of RAM. To try to control variables in the computing environment as much as possible background tasks were stopped. This was done in order to ensure that the results are more accurate.

All of the test classes are constructed the same way, the only difference is if the arrays to be sorted contain numbers in random, ascending, descending order, or if it’s only equal numbers.

First is a setup of a series of tests using the Data class which generated the integer arrays. The sizes of the arrays ranged from n= 100 to n= 100,000 elements, increasing the problem size with one order of magnitude each time. I also tested to increase the problem size with constant growth by doubling the size for each array (see the charts under *Results*) . The arrays contain numbers between 1 and 100. To study the behavior of the algorithms on arrays of random, ascending, descending or equal elements, each algorithm was used to sort each array/dataset 100 times each. In order to avoid bias the order or runs is mixed up.

An average run-time is calculated for each dataset/array per algorithm. This value is then added to an array which contains the average run-times for each dataset, one array per algorithm. To discard results that are affected by JVM warmup, this process is repeated 70 times, but it is only the last 20 runs that will be used.

For the last 20 runs, the arrays that contain the average run-times for each dataset will get added to new arrays, i.e. we will end up with one array per algorithm containing 20 smaller arrays with the average run-times, for each dataset, for that algorithm. From each of these new arrays, we will pick a random sample, to avoid cherry picking. This sample (i.e. array containing the average run-time for each dataset) gets to represent the result for that algorithm.

4. Results

Abbreviations for the different quicksort implementations will be used in the tables:

- **Qs V1 :** QuicksortFixedPivot
- **Qs V2 :** QuicksortRandomPivot
- **Qs V3 :** QuicksortFixedPivotInsertion
- **Qs V4 :** QuicksortRandomPivotInsertion

Table 1. Test 1: Random Data

Test 1: Random Data						
Problem Size	**InsertionSort**	**Qs V1**	**Qs V2**	**Qs V3**	**Qs V4**	**Arrays.sort**
100	2 009	1 895	8 679	1 640	8 082	1 315
1 000	197 077	32 951	47 212	31 179	49 358	36 921
10 000	17 216 344	355 579	360 564	347 874	361 889	354 932
100 000	1 702 243 584	3 300 195	3 449 076	3 381 160	3 461 393	3 367 703
1000 000	----------*	33 119 592	34 610 408	34 254 352	34 715 968	35 066 848

*The execution time for InsertionSort is excluded because it took too long to compile. InsertionSort is much less efficient on large lists.

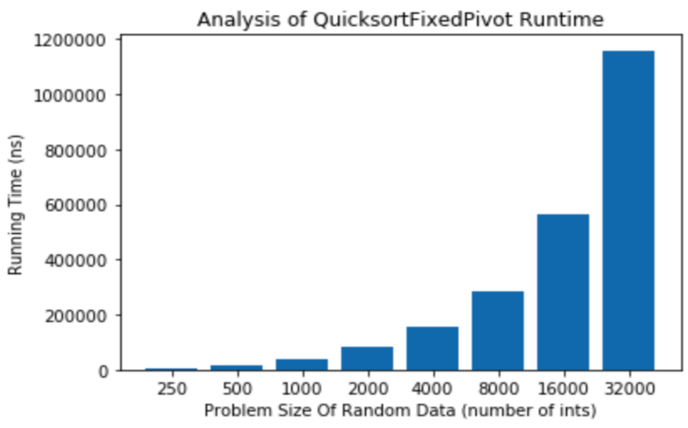

Fig 1. Analysis of QuicksortFixedPivot runtime on random data.

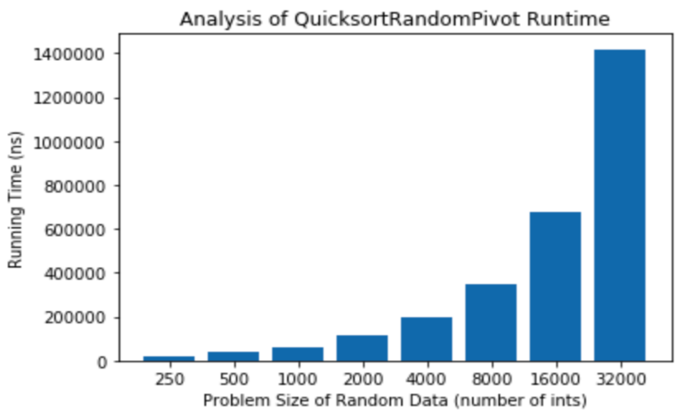

Fig 2. Analysis of QuicksortRandomPivot runtime on random data.

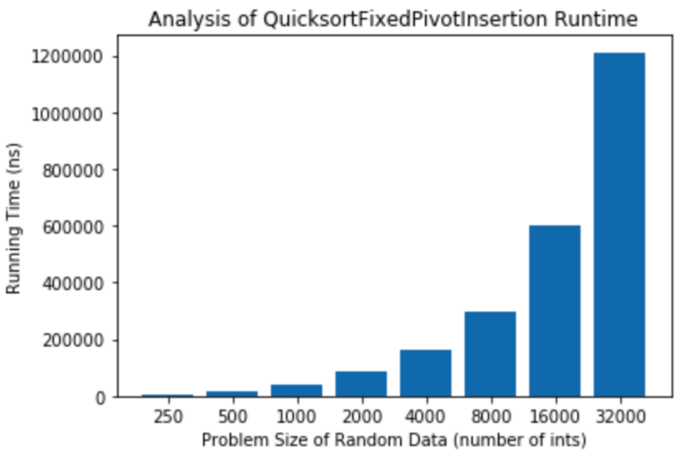

Fig 3. Analysis of QuicksortFixedPivotInsertion runtime on random data.

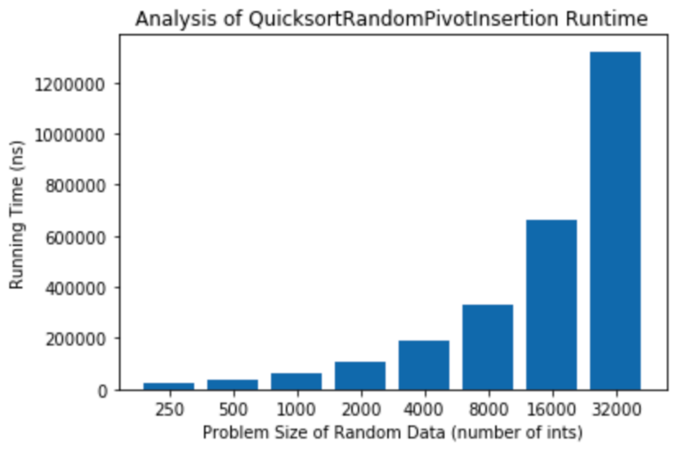

Fig 4. Analysis QuicksortRandomPivotInsertion runtime on random data.

Table 2. Test 2: Sorted Data

Test 2: Sorted Data						
Problem Size	**InsertionSort**	**Qs V1**	**Qs V2**	**Qs V3**	**Qs V4**	**Arrays.sort**
100	122	6 408	6 874	7 947	8 124	392
1 000	773	98 841	29 600	101 903	28 477	771
10 000	8 735	893 568	152 646	915 728	161 064	10 312
100 000	101 134	9 214 879	1 479 635	9 121 544	1 512 424	105 078
1000 000	1 058 760	95 279 424	14 972 896	91 992 192	15 621 872	1 185 728

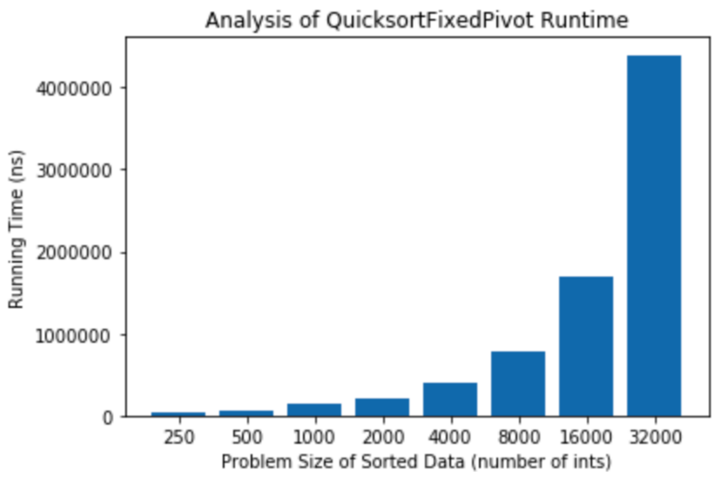

Fig 5. Analysis of QuicksortFixedPivot runtime on sorted data.

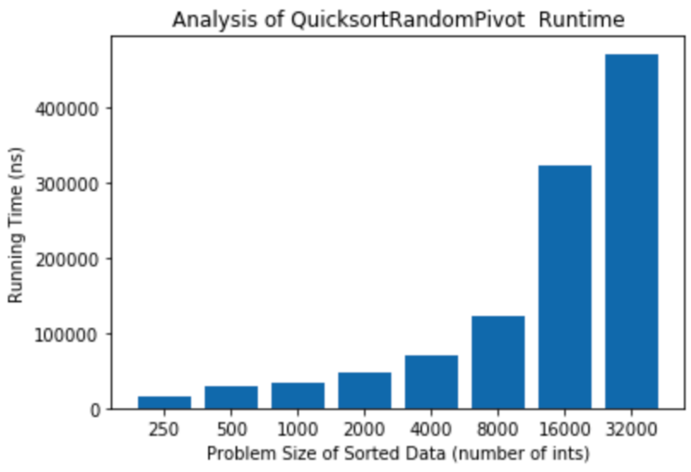

Fig 6. Analysis of QuicksortRandomPivot runtime on sorted data.

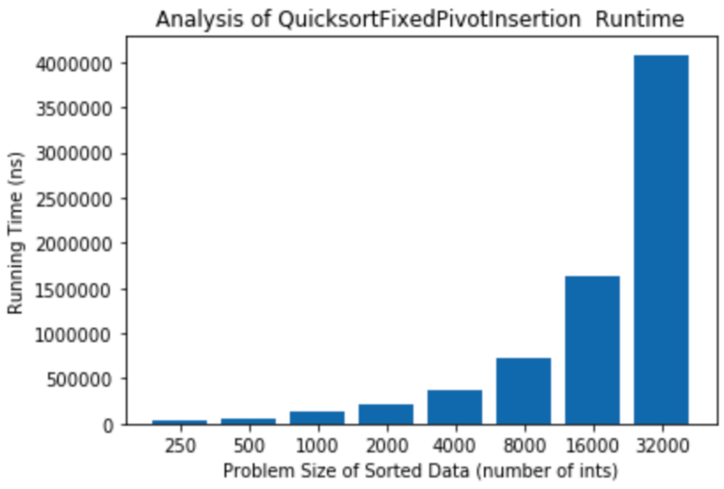

Fig 7. Analysis of QuicksortFixedPivotInsertion runtime on sorted data.

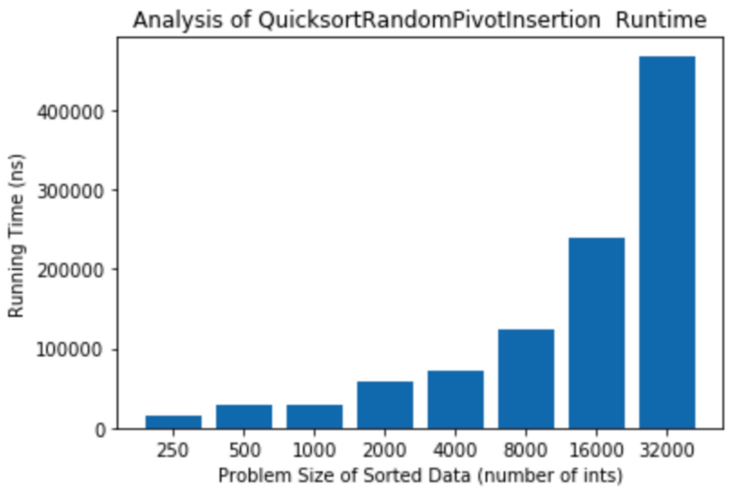

Fig 8. Analysis of QuicksortRandomPivotInsertion runtime on sorted data.

Table 3. Test 3: Reversed Data

Test 3: Reversed Data						
Problem Size	**InsertionSort**	**Qs V1**	**Qs V2**	**Qs V3**	**Qs V4**	**Arrays.sort**
100	3 636	7 704	7 658	6 867	7 971	911
1 000	351 414	112 509	27 977	90 510	42 853	1 780
10 000	32 886 762	1 014 280	142 461	805 716	173 832	18 703
100 000	2 147 483 647	8 878 733	1 307 549	8 222 365	1 594 464	149 126
1000 000	----------*	91 983 560	13 241 767	84 653 168	14 688 919	1 701 533

*The execution time for InsertionSort is excluded because it took too long to compile. InsertionSort is much less efficient on large lists.

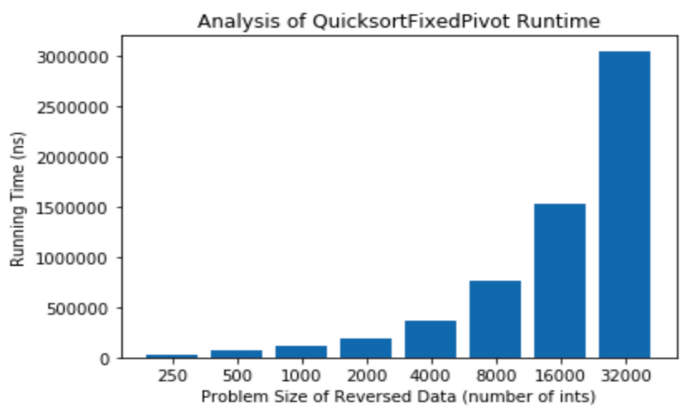

Fig 9. Analysis of QuicksortFixedPivotInsertion runtime on reversed data.

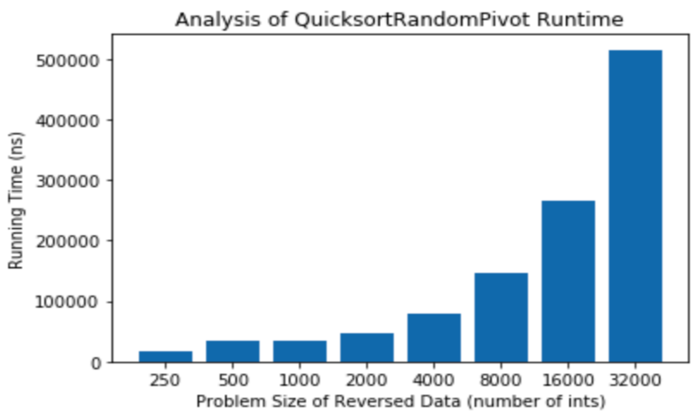

Fig 10. Analysis of QuicksortRandomPivot runtime on reversed data.

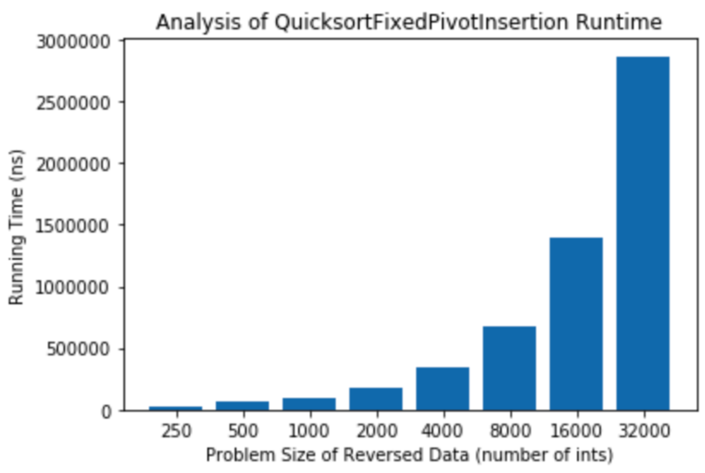

Fig 11. Analysis of QuicksortFixedPivotInsertion runtime on reversed data.

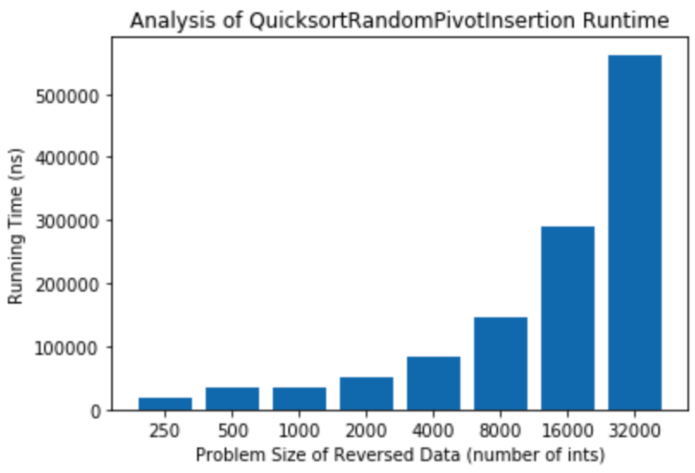

Fig 12. Analysis of QuicksortRandomPivotInsertion runtime on reversed data.

Table 4. Test 4: Equal Data

Test 4: Equal Data						
Problem Size	**InsertionSort**	**Qs V1**	**Qs V2**	**Qs V3**	**Qs V4**	**Arrays.sort**
100	108	119	257	169	253	363
1 000	894	782	1 356	1 319	1 296	734
10 000	7 070	9 119	12 891	10 029	13 343	6 561
100 000	125 611	112 764	157 985	108 760	173 628	89 294
1000 000	894 845	1 119 961	1 552 443	1 149 196	1 575 898	884 544

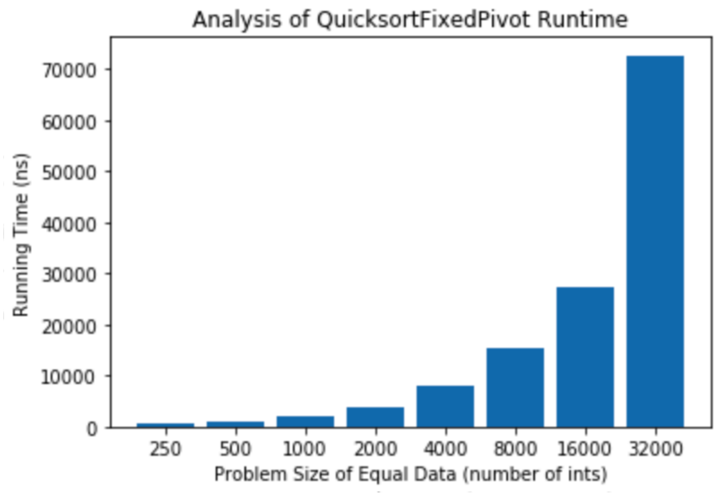

Fig 13. Analysis of QuicksortFixedPivot runtime on equal data.

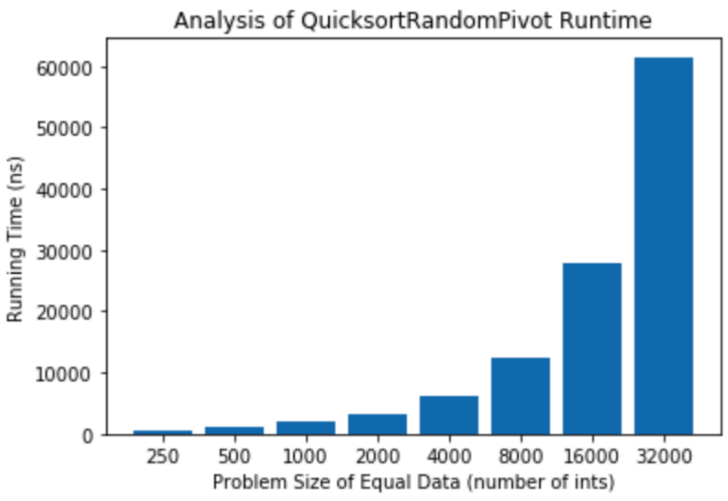

Fig 14. Analysis of QuicksortRandomPivot runtime on equal data.

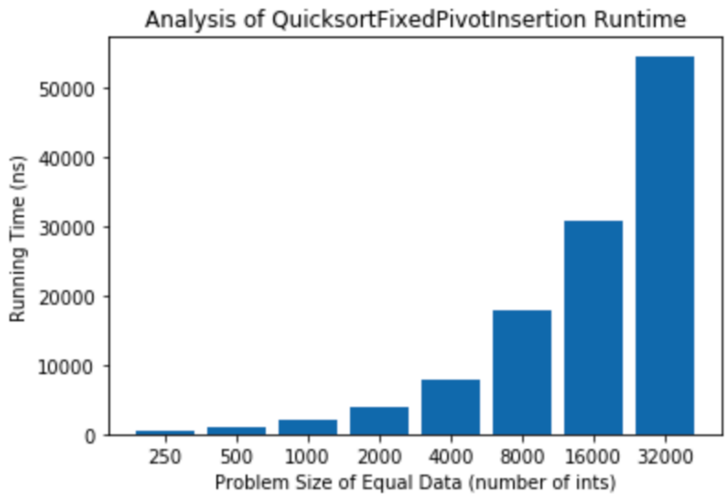

Fig 15. Analysis of QuicksortFixedPivotInsertion runtime on equal data.

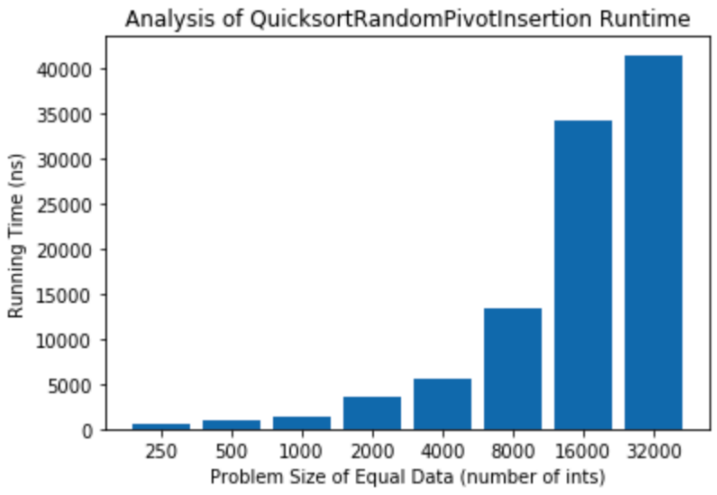

Fig 16. Analysis of QuicksortRandomPivotInsertion runtime on equal data.

5. Discussion

When sorting random data, QuicksortFixedPivot seems to be the fastest, see table 1. It came to no surprise that Insertion sort would be the slowest since it's much less efficient on larger lists. QuicksortRandomPivot and QuicksortRandomPivotInsertion are the slowest out of the four quicksort variants. This is probably due to the fact that generating a random number is expensive.

Looking at the charts, see figs. 1-4, the time required to sort random data is slightly more than doubled if the array's size is doubled. This corresponds to the expected *O(n log n)* time complexity.

For QuicksortFixedPivot and QuicksortRandomPivotInsertion, sorting data in reversed/descending order generally took a little longer than sorting data in sorted/ascending order. Both QuicksortRandomPivot and QuicksortFixedPivotInsertion were slower at sorting sorted data rather than reversed data, see table 2 and 3.

Both QuicksortRandomPivot and QuicksortRandomPivotInsertion are significantly faster for sorted and reversed input data than for random data.

When sorting equal elements, QuicksortFixedPivot is the fastest of the four implementations, and QuicksortFixedPivotInsertion is the second fastest. QuicksortRandomPivot and QuicksortRandomPivotInsertion have very similar sort times, and are a bit slower than the other two. This is probably because generating a random number is expensive.

My Quicksort implementations do not quite come close to that of the JDK, but when sorting random data QuicksortFixedPivot and Arrays.sort have similar performance.

In conclusion, the efficiency of Quicksort ultimately depends on the choice of the pivot.

6. Reference list

Sehgal, K. (2018). *A Quick Explanation of Quick Sort.* Available at https://medium.com/karuna-sehgal/a-quick-explanation-of-quick-sort-7d8e2563629b (Accessed: 17 March 2021).

'Quicksort' (2021). *Wikipedia.* Available at https://en.wikipedia.org/wiki/Quicksort (Accessed: 17 March 2021).