Randomized Algorithms - Assignment

Sergio Mosquera Dopico

November 2018

1 Assignment Statement

In the lectures, we have seen two randomized algorithms for finding the median of an input array of unordered keys: QSelect and RMedian. You should do an empirical comparison of the computer time that needs each algorithm to find the median. to find the mean. You also should compare with the algorithm consisting of sorting the input and after going to position $[n/2]$*. For sorting you can use any comparison algorithm.*

2 The Algorithms

As section 1 states, we need to implement two algorithms to find the median of a given set of integers. These two algorithms are Quick-Select and Randomized-Median, we will also need to implement either Quick-Sort or Merge-Sort to achieve the previous implementations (in this case I will implement Merge-Sort).

I will explain the algorithms with its formal statement (or with some pseudocode) and also adding some clarifications to get the main idea and also what we want for our implementation. Then, for each algorithm, I will analyze the expected complexity that we want to achieve.

2.1 Quick-Select

In computer science, quickselect is a selection algorithm to find the k^{th} smallest element in an unordered list. It is related to the quicksort sorting algorithm. Like quicksort, it was developed by Tony Hoare, and thus is also known as Hoare's selection algorithm. Like quicksort, it is efficient in practice and has good average-case performance, but has poor worst-case performance.

We have to take into account for the experiments with this algorithm, is that it has bad performance when executing with data representing the worst-case. We will then see which is the worst case. The worst case appears when we want to select the minimum element and your pivot at every stage happens to be the last element in the list at that stage. We will not need to worry about this because we are looking for the median and also the pivot selection is random, so we cannot test this.

Quickselect uses the same overall approach as quicksort, choosing one element as a pivot and partitioning the data in two based on the pivot, accordingly as less than or greater than the pivot. However, instead of recursing into both sides, as in quicksort, quickselect only recurses into one side – the side with the element it is searching for.

2.2 Randomized-Median

I will explain this algorithm as a set of steps instead of doing in a formal way, because it is easier to understand if we do it like this. It is important to mention that this algorithm is not accurate, but finds the median w.h.p.

At first, we need as input a set S of n elements

1. Pick a set R of $[n^{3/4}]$ elements in S chosen independently and u.a.r. with replacement.
2. Sort the set R.
3. Let d be the $[\frac{1}{2}n^{3/4} - \sqrt{n}]^{th}$ smallest element in the sorted set R.
4. Let u be the $[\frac{1}{2}n^{3/4} + \sqrt{n}]^{th}$ smallest element in the sorted set R.
5. By comparing every element in S to d and u, compute the set $C = \{x \in S : d \leq x \leq u\}$ and the numbers $l_d = |\{x \in S : x < d\}|$ and $l_u = |\{x \in S : x > u\}|$
6. If $l_d > \frac{n}{2}$ or $l_u > \frac{n}{2}$ then FAIL.
7. If $|C| \leq 4n^{3/4}$ then sort C, otherwise FAIL.
8. Output the $[\frac{n}{2} - l_d + 1]^{th}$ element from the ordered C.

2.3 Merge-Sort

In computer science, merge sort (also commonly spelled mergesort) is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the order of equal elements is the same in the input and output. Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945.

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

3 Project implementation

In this section I will explain which technology I have used to implement the project and also where can the project be downloaded.

3.1 Programming language

For this assignment, I have decided to use Java for three main reasons:

1. It is easy to understand. Java uses a C-like syntax and as it is a typed language it is easier to know what do variables represent at every moment.

2. I am very used to Java. I have been using it for almost the 4 years of my degree so It will not be a major issue using it this time for the project although it is not one of my favorite programming languages.

3. Almost every computer runs Java. So it would be easier for anyone wanting to test the project to do it on its own computer.

4 Experiments

We are required to test each algorithm with, at least 100 instances of 50000 integers lists, each integer within a range of at least three digits. In this case I will execute 340 instances (to get more accurate results when plotting) that will be divided as follows:

1. 10 examples will be "special" sets, that is, some cases which will appear with extremely low probability (i.e. an array full of zeros, an array filled using just two numbers different, an ordered array, an inversely ordered array, etc.). The configuration for these instances will be with size 50.000 and digits in range [1,3] for each integer.

2. The rest of the elements will be divided in sets of 30 instances, each of them with different sizes starting with 50.000 integers and finishing with 500.000. This is to test the evolution of the elapsed time for each method with increasing size instances. As in the previous case, we will use integers with digits in range [1,3].

These instances will be generated each time we execute the program. This is due to the needed storage to keep all these instances inside a file.

The code is hosted in a GitHub repository[1].

5 Results & Conclusions

After generating instances as explained in section 4 and executed the algorithms explained in section 2, we obtained the following results:

	QUICKSELECT	RANDOMIZED MEDIAN	MERGESORT
50.000	633	273	423
75.000	575	185	551
100.000	754	222	725
125.000	888	279	934
150.000	1302	326	1105
175.000	1377	376	1492
200.000	1525	551	1961
250.000	1893	629	3191
300.000	2595	729	3986
400.000	3701	808	4626
500.000	4893	1039	6023

Figure 1: Total elapsed time for instance size/algorithm

The above table shows the total elapsed time (just the algorithm execution) for each case, that is, the execution of the algorithm for the 30 instances of each size using each of the algorithms. Just to clarify, the special cases were not taken into account for this table because the results where not significant.

As we can see on figure 1, in every case the Randomized Median gets a better time than the other two methods, this does not mean that the second method is strictly better than the other ones, this needs to be clarified. As I mentioned in section 2.2, this algorithm is not always accurate so it finds the median with high probability, that means that although it is the fastest, the result may be wrong.

In the second place we find the Quickselect option that, although from the first instances gets worse results than the traditional way (sort and take $n/2^{th}$ element) with greater instances shows a more efficient behavior. If we plot the results in a graph we obtain the following:

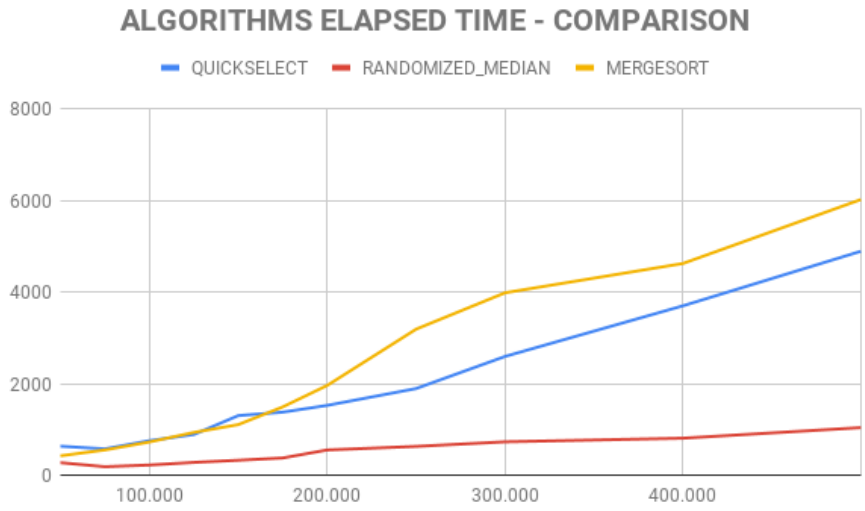

Figure 2: Plot of the algorithms results.

The plot from figure 2 shows an approximation of the complexity of each algorithm, as we can see (we can just have a guess about the following results) the randomized median elapsed time grows very slowly compared to the other two algorithms.

Now, comparing the Mergesort strategy and the Quickselect one, they look similar but we can see that at those points we can see on the graph, mergesort grows faster than quickselect.

This makes sense with the initial hypothesis. We have stated that the expected complexity for these algorithms was:

1. $O(n)$ for Randomized Median. As we can see it has the lowest complexity on the graph and keeps growing much more slower than the other two.

2. $O(n \cdot log(n))$ for Quickselect. It is quite obvious that its complexity is worse than randomized median's one, but also that it is better than mergesort whose expected complexity is $O(n^2)$.

3. $O(n^2)$ for Mergesort and choosing $n/2^{th}$ element. As stated by the previous arguments, its complexity is worse than the other two cases and the growth is approximate to a parable (x^2 graph).

References

[1] **Randomized Algorithms Repository — Github**.
github.com/SergioMD15/Randomized-Algorithms