Programming Assignment

Miguel Alcón Doganoc

December 13, 2018

1 Introduction

The objective of this assignment is to do an empirical comparison of the computing time that the algorithms **QSelect**, **RMedian**, **QuickSort** and **MergeSort** need to find the median of an array of numbers. Being **QuickSort** and **MergeSort** sorting algorithms, finding the median means sorting the array and going to the position $\lceil n/2 \rceil$. We have that:

- Quick select or **QSelect** takes $O(n)$ in expectation.
- Randomized median or **Rmedian** takes $O(n)$ in expectation.
- Merge sort or **MergeSort** takes $O(n \log n)$.
- Quick sort or **QuickSort** takes $O(n \log n)$ in average.

2 Algorithm Implementation

I used *Python 3* to make the implementation of the whole experiment. I have written all the code, but I have inspired by some algorithms on Internet. The implementations of **QSelect** and **Mergesort** where inspired by [1] and [2], respectively, but they have some modifications made by me. My implementation of **QuickSort** is based on the **QSelect** implementation and what I know about the algorithm. I want to mention that the pivot of my **QuickSort** is selected randomly. Finally, my implementation of **RMedian** follows the slides provided in class [3], and it uses my **QuickSort** as the sorting algorithm. All the implementations of the algorithms are in the file `src/algorithms.py`.

3 Instances generation

As it is said in the statement, I had to generate 100 instances of an input consisting of an 50000 integer array, each integer within a range of at least three digits. In order to do that, I wrote a *Python 3* script. This script and the data generated by it is located in `src/algorithms.py` and `data/`, respectively.

Although my **QuickSort** uses a random implementation (pivot selected randomly), I generated some random instances sorted in ascending and descending order (ad hoc instances). I made it in order to check if the algorithms have the same behavior as with the totally random instances.

For the experiment, I generated 100 instances:

- 15 random and sorted in ascending order.
- 15 random and sorted in descending order.
- 70 totally random.

4 Computing time

With the instances generated in last section, I start with the experimentation. First, I wrote a *Python 3* script that defines the experiment. You can find it at `src/experiment.py`. In order to ensure the correct behavior of the algorithms in the experimentation, for each instance executed I check if the resultant median obtained by each algorithm is the same (you can find it in the code as an inner variable of the class *Experiment* named *problems*).

However, with the execution of the totally random instances I obtained the following times (see table 1). As you can see, **QuickSort** and **MergeSort** needs more computing time than **QSelect** and **RMedian**, as I expected because of their complexity. The fastest algorithm is **QSelect**, almost 5 times faster than its following opponent, **RMedian**.

Algorithm	Min	Max	Mean	Total
QuickSort	0.24353900	0.26651900	0.25306539	17.71457700
MergeSort	0.39706300	0.40827900	0.40039161	28.02741300
QSelect	0.01248700	0.03899800	0.02248419	1.57389300
RMedian	0.05955100	0.06731900	0.06281120	4.39678400

Table 1: Computing time of each algorithm using totally random instances.

With the execution of the ad hoc instances, I obtained the following results (see tables 2 and 3). As you can see, there is no significant difference between both results. Furthermore, comparing this times with (see table 1), there is also no significant difference, as I expected. This happens because my **QuickSelect** implementation choose the pivot randomly, so an ad hoc instance that produces a bad behavior on **QuickSort** is impossible to create.

After seeing all these results, I thought that I could improve the **RMedian** algorithm using **QSort** instead of **QuickSort** to select the $(\lfloor n/2 \rfloor - I_d + 1)$-smallest element in sorted C (all these notation is based on [3]). So, I had to check if I was right. I implemented the new algorithm, **RMedian-QSelect**, and I obtained the following results (see table 4).

Algorithm	Min	Max	Mean	Total
QuickSort	0.23792200	0.24938600	0.24414540	3.66218100
MergeSort	0.31680300	0.32050200	0.31914867	4.78723000
QSelect	0.01225900	0.03180900	0.02111927	0.31678900
RMedian	0.06020600	0.06623700	0.06258440	0.93876600

Table 2: Computing time of each algorithm using random instances sorted in ascending order.

Algorithm	Min	Max	Mean	Total
QuickSort	0.23534500	0.25866600	0.24728947	3.70934200
MergeSort	0.32325900	0.33261300	0.32645280	4.89679200
QSelect	0.01449000	0.03068200	0.02064093	0.30961400
RMedian	0.05946800	0.06985100	0.06263613	0.93954200

Table 3: Computing time of each algorithm using totally random instances sorted in descending order.

Instance type	Min	Max	Mean	Total
Totally random	0.03404000	0.04049000	0.03602529	2.52177000
Ascending ad hoc	0.03374800	0.03725300	0.03572167	0.53582500
Descending ad hoc	0.03447100	0.03816200	0.03582367	0.53735500

Table 4: Computing time of RMedian-QSelect algorithm using each type of instances.

As you can see, the computing time that **RMedian-QSelect** needs is almost a half of what **RMedian** needs.

To finish, I plotted the computing time of the algorithms for this experiment (see figures 1 and 2) to show more graphically their differences.

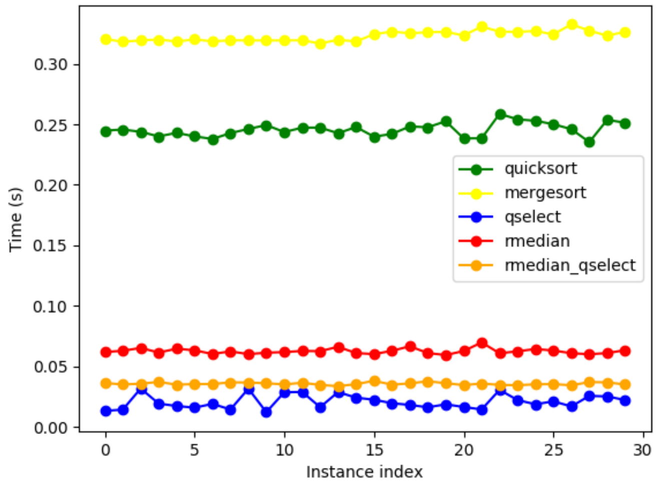

Figure 1: Computing time of the algorithms using the ad hoc instances

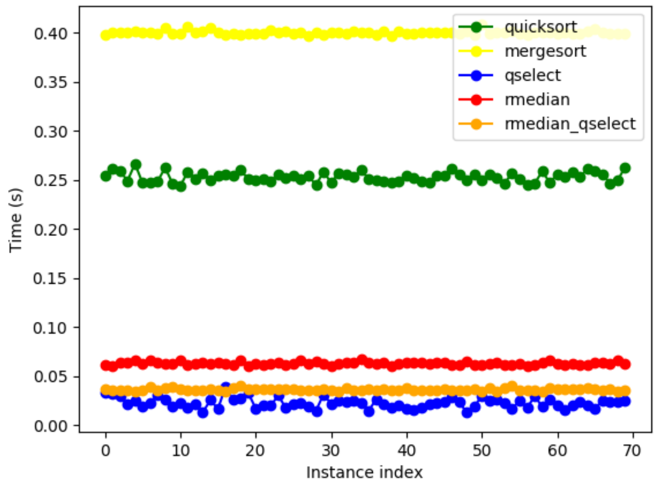

Figure 2: Computing time of the algorithms using the totally random instances

5 Complexity plots

I extended the `src/experiment.py` script to plot the complexity of each algorithm. In order to do that, I created 100 new totally random instances of increasing n (from 1000 to 100000, with steps of 1000) and I run the experiment. Here you have the resultant plot (see figure 3).

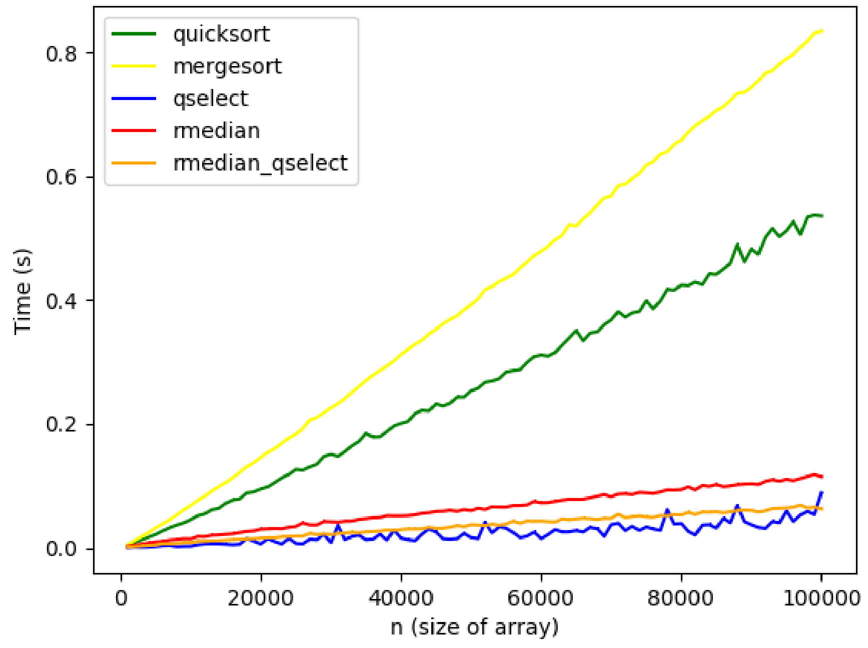

Figure 3: Complexity of the algorithms

6 Conclusions

From all the experiments, I can conclude that **QSelect** is the fastest algorithm to find the median of an array of numbers. It is also more polyvalent, because it can be used to find the i-th element of an unsorted array. Furthermore, if **RMedian** uses **QSelect**, it is obtained a faster algorithm than **RMedian** alone, in some cases even better than **QSelect**. I want to mention that, as you can see in all figures, the worst cases of **QSelect** need more or less the same times than in **RMedian-QSelect**. You can also see that the computing times of **QSelect** and **QuickSort** have more variance (specially when increasing n) than the others. This happens because the selection of the pivot is really important, so its correct or wrong selection flows into good or bad times, and it is random. Because all these things, and because it does not have significant variances, I can say that the more reliable algorithm is **RMedian-QSelect**.

References

[1] Wikipedia. *QuickSelect*. [ONLINE] Available at: https://en.wikipedia.org/wiki/Quickselect

[2] Wikipedia. *MergeSort*. [ONLINE] Available at: https://en.wikipedia.org/wiki/Merge_sort

[3] Josep Diaz. *Concentration of a random variable around its mean*. [ONLINE] Available at: http://www.lsi.upc.edu/~diaz/Concentration.pdf