RA Assignment

Gerard Cegarra Dueñas

December 2018

1 Introduction

In this document we explore three different methods for finding the mean of a list of integers. The first method is based on the use of the Quicksort algorithm, the second method is based on the Quickselect algorithm and the third method is based on the RMedian algorithm. The following sections describe some details about the three implementations and the different experiments that have been done in order to compare the performance between the algorithms.

I have implemented all the code used in this assignment in Node.js using only native libraries. I have also implemented an R script to generate the graphs included in this document.

The code can be found at the GitHub public repository `https://github.com/gerardcd/RA_assignment`

2 Implementation

2.1 Quicksort algorithm

The first method for finding the mean of a list of integers is based on Quicksort, and consists on sorting the list and retrieving the middle element (or elements in the case of even-size inputs) once sorted. There are two details about the Quicksort implementation that can notably change the performance of the algorithm when we measure its execution time.

2.1.1 Argument passing to recursive calls

The first implementation detail that can impact on its execution time is the way we pass the list argument to the recursive calls. The simpler way to write the Quicksort algorithm is to generate at each recursive step two arrays, one with the elements smaller than the pivot and another one with the greater ones. Each one of this arrays will be passed to a new recursive call. Then, the signature of the algorithm is as simpler as it could be: $quicksort(L)$, and the partition process is trivial to implement.

However, this implies, at each recursive step, an extra operation of linear time, which doesn't modify the theoretical complexity of the algorithm, but it does modify the real execution time. Hence, I implemented the algorithm signature as $quicksort(L, left, right)$, where $left$ and $right$ are indexes that bound L for each recursive call, and the partition operation is implemented to leave the result of the partition in the same L array. This way, the algorithm takes advantage of the Javascript's pass by reference and the two extra copies of half L are avoided.

2.1.2 Pivot selection

The second implementation detail that makes a big difference on Quicksort's performance is the pivot selection. It doesn't matter how we select the pivot if we consider, for a given size n, that the

input of the algorithm is equally likely to be any one of the $n!$ possible inputs of size n. However, it may have a big impact when the inputs of the algorithm are premeditatedly chosen to provoke the worst case execution time. For example, a possible pivot selection is to choose the first element as the pivot. In this case, any sorted list will make the algorithm to run in $\theta(n^2)$ time.

Randomizing the pivot selection makes it impossible to generate such an input that provokes the worst case execution time. However, as we really want to provoke this worst case behaviour for measurement purposes in the experiments, we will use the first element pivot selection. Then, we will be able to easily generate this worst case execution time with sorted arrays.

2.2 Quickselect algorithm

The second method to find the mean is based on the Quickselect algorithm. It consist on using Quickselect to find the middle element of the list, which is by definition its mean.

The two considerations explained in the above section about the Quicksort algorithm also apply to the Quickselect algorithm. The recursive calls and the partition operation have been implemented in the same way.

2.2.1 Input size parity

There is a third consideration to be done in the quick select implementation about the parity of the size n of the input list L. If n is odd, then $n = 2k + 1$ and the mean of L is the $k + 1$ smallest element of L, which can by found in a single run of the Quickselect algorithm. If n is even, then $n = 2k$ and the mean of L is $m = (a + b)/2$, where a and b are the $k - 1$ and k smallest elements of L respectively. An straight forward way of finding a and b is to run two times the Quickselect algoritm, one for each value. This strategy has a worst case cost of $\Omega(2n^2)$, which is theoreticaly the same as one run of the Quickselect algorithm ($\Omega(n^2)$) but the double in real execution time. Hence, the parity of n should be taken into account in the complexity experiments.

However, I could came up with a better strategy to find the median of L with Quickselect when n is even. First we run Quickselect to find b. As we are passing L by reference to the Quickselect algorithm, we know that, after running the algorithm, any element smaller than b is in the $k - 1$ first positions of L, so a is there as well. Then, we just need to find $a = min\{x_i | x_i \in L, i < k\}$, and this can be done in $\Omega(k - 1) = \Omega(n/2 - 1) = \Omega(n/2)$ time.

This algorithm has a total cost of $\Omega(n) + \Omega(n/2) = \Omega(3n/2)$ in average and $\Omega(n^2) + \Omega(n/2) = \Omega(n^2 + n/2)$ in the worst case. More refined algorithms might exist, but this one has already a performance really smilar to quickselect with odd inputs (experiment in section 3.2). Then, we can forget about parity at the final complexity experiments without loss of generality.

2.3 RMedian algorithm

The third method is based on the use of the RMedian algorithm, wich is a randomized algorithm that, with high probability, returns the mean of a given list.

For this algorithm we will take in account similar considerations to the ones in the previous sections.

2.4 Input size parity

Like Quickselect, RMedian implementation should also be slightly modified to work with even-sized inputs. This algorithm returns always an element from the list, and as we are working with sets (no repetition) with this algorithm, it would never work for even-sized inputs.

If n is even and $m = (a + b)/2$, then m can't be an element of L, otherwise (as $a \neq b$ because L is a set) $a < m < b$, and then a and b are not correct (they are not the middle elements of L).

However, the modified version of RMedian to work with even-sized inputs would also run in $\Omega(n)$ time. Hence, we can forget about the parity of the inputs when measuring the running time of this algorithm without loss of generality.

2.4.1 R and C sorting

For the two sorting processes of the RMedian algorithm (sort R and C) I have re-used the Quicksort algorithm implementation discussed before (section 2.1). This decision has been taken also for the pivot selection strategy, which is again choosing the first element at each recursive call. In this way, we can induce a worst case execution time to the RMedian algorithm (it sorts C which contains some elements of S in the same order as they are in S) using sorted lists as we do for Quicksort and Quickselect algorithms.

3 Experiments

This section details the different experiments conducted to evaluate and compare the performance of the algorithms.

3.1 Instances generators

I have implemented three different instances generators. Each one of them generates a list of elements with different properties. This data generation is done dynamically at every run of the main program, which will send it to the three algorithms. That's why there is no data inputs in the source code.

The first data generator I have implemented generates random arrays, and will be referred as RAG. This one is intended to measure the Quicksort and Quickselect behaviour in average cases. It receives a length l and a number d of digits as input parameters. It outputs a list $L = (x_0, ..., x_l)$, $|L| = l$, where every $x_i \in L$ acomplishes that $10^{d-1} \leq x_i < 10^d$.

The second data generator implemented, referred as SAG, generates sorted random arrays without repeated elements. This one will be used to measure the worst case behaviour of the three algorithms. It receives the same parameters l and d. It outputs a list L that has the same properties as the the one produced by the random array generator, plus the property that the elements are sorted. To generate this kind of arrays, it calculates the increment $inc = (10^d - 10^{d-1})/l$ and calculates each $x_i \in L$ as $x_i = x_{i-1} + random(1, inc * 2)$, having $x_0 = 10^d$. In this way, the elements in L have a constant density for any interval in $[10^{d-1}, 10^d]$. Note that some elements (expected to be none) might be grater than 10^d.

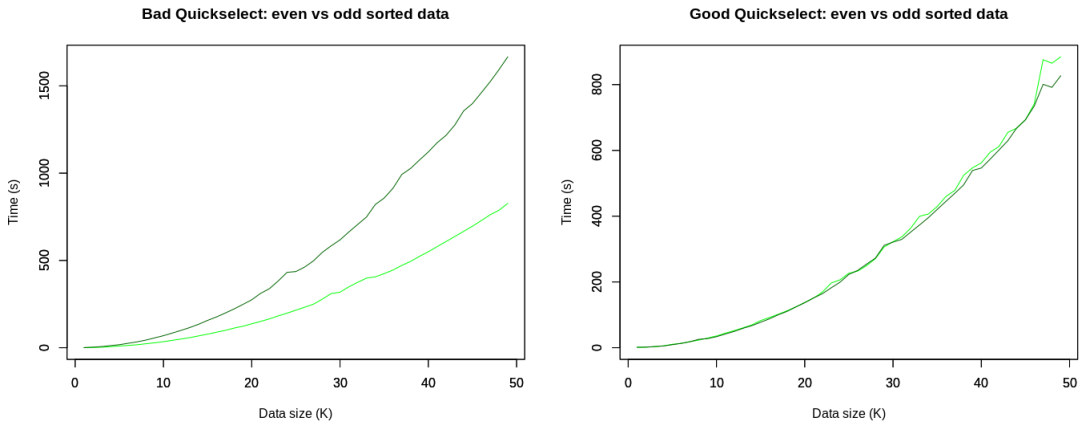
The third data generator (RSG) generates random sets. This one is intended to be used to measure average case behaviours of the RMedian algorithm. It simply gets a list L form the SAG, and returns one of the $l!$ possible permutations choosen u.a.r.

3.2 Quickselect parity

In this experiment we measure the impact of the Quickselect algorithm optimization for the even-size inputs (described in section 2.2.1). We aim to compare the two different implementations of the Quickselect by their worst case time execution, so the input dataset for this experiment will be generated by the SAG.

The input dataset consists in two big groups. One is formed by even-sized inputs and the other one by odd-sized inputs. Each one consist of inputs of increasing size (from 10^3 to $5*10^4$ in intervals of 10^3) and 5 digits. For each size we find 5 different inputs. These 5 inputs are sent both to the non-optimized and the optimized version of the algorithm. Then the execution time is averaged among the 5 inputs and it represents the execution time for that specific size for each algorithm.

In the following graphs we can see the result:



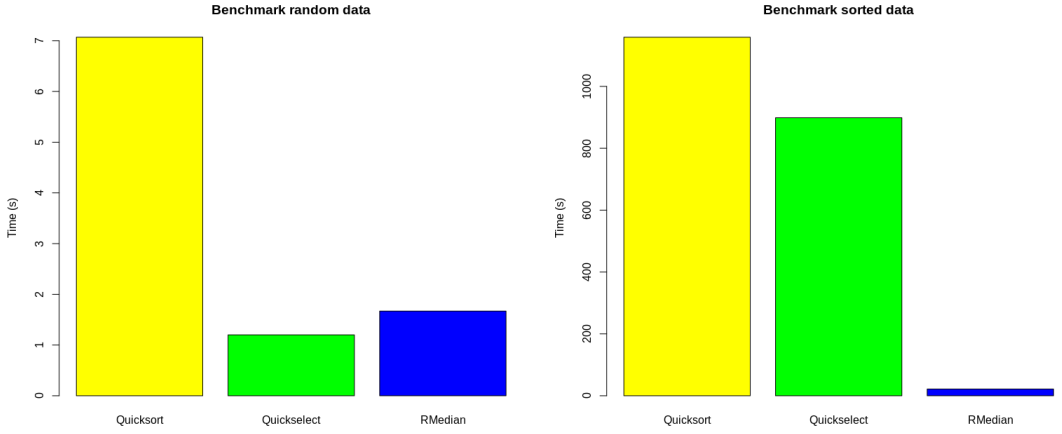
The left hand side graph shows the execution time of the non-optimized algorithms for the even-sized inputs (dark green) and the odd-sized ones (light green). The right hand side graph shows the same data for the optimized algorithm. We can see how the non-optimized Quickselect algorithm takes double time to solve the even-sized inputs than to solve the odd-sized ones, while the optimized version makes no difference. Eventough the worst case complexities of both algorithms are theoretically the same $\Omega(2n^2) = \Omega(n^2 + n/2)$, they perform different in practice.

3.3 Benchmark

In this experiment we use a benchmark input dataset to see the algorithms execution time it the average case and in the worst case. The input dataset has two groups: one for the average case measurment that is generated by the RAG, and another one for the worst case that is generated by the RSG (so then is compatible with the RMedian algorithm). As we have long discussed, we can use only odd-sized inputs withous loss of generality.

Then, each group will consist of 100 inputs with $5 * 10^4$ elements of 5 digits. Every algorithm will find the median of each input and we will average their execution time among the 100 instances.

The following graphs show the results of the experiment:



As we can clearly see, and as it was expected, Quicksort is the slowest algorithm both in worst and average case.

It is interesting that Quickselect works slightly better than RMedian in the average case. Actually, they both have an expected cost of $\Omega(n)$, but here we can see that RMedian has a real greater cost.

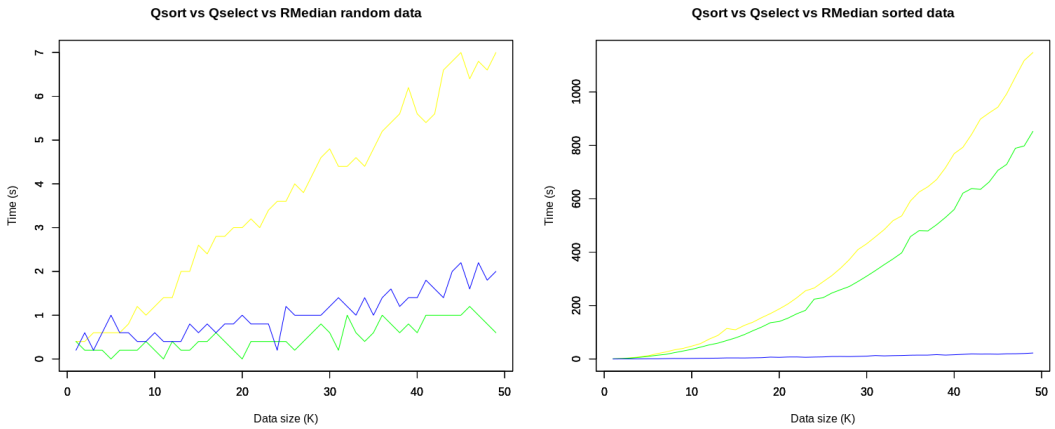
This can be understood if we analyze the algorithms complexity at a greater level of granularity. In the average case, the Quickselect recurrence is $T(n) = T(n/2) + \Omega(n) = 2n - 2 + k = \Omega(2n)$. On the other hand, the RMedian algorithm cost is the cost of find R ($\Omega(n^{3/4})$), sort R ($\Omega(n)$), find C, ld, lu ($\Omega(n)$), and sort C ($\Omega(n)$). So the cost of RMedian is $T(n) = \Omega(3n) + \Omega(n^k) = \Omega(3n)$. We can see something close to this 2/3 relation in the left hand side graph.

In the right side hand graph, we can observe the big difference between Quicksort, Quickselect and RMedian in their worst case. Here, we find the big difference we expected to see between RMedian and the two other algorithms. With sorted inputs, Quicksort and Quickselect run in $\Omega(n^2)$. Recall that RMedian sorts C using Quicksort, and that C has some elements of S in the same order as they are in S. Hence, if S is sorted, C is sorted, and RMedian sorts C in $\Omega(|C|^2)$ time. However, $|C| = n^{3/4}$ and then, sorting C takes time $\Omega((n^{3/4})^2) = \Omega(n^{3/2}) = \Omega(n)$. Then, RMedian keeps the linnear complexity even in its worst case.

3.4 Complexity

In this final experiment we will observe the evolution of the algorithms execution time while increasing the input size. This experiment's input data is a combination of both two first experiments. We will use two groups of inputs, one with random inputs generated by the RSG, and another one with sorted inputs generated by the SAG. Each one will consist of inputs of increasing size (from 10^3 to $5 * 10^4$ in intervals of 10^3) formed by numbers of 5 digits. Again, we will average the result for each size among 5 different inputs.

The following graph shows the results:



These two graphs confirm what we could observe in the last experiment (section 3.3). In the left hand side graph we see how the three algorithms evolve close to linear time (Quicksort increases quicker as its average case time is $\Omega(nlogn)$) and also that Quickselect performs a little bit better than RMedian.

On the other hand, in the right hand side graph, we also confirm that in terms of the worst case, RMedian evolves in the linnear way described before (section 3.3), while Quicksort and Quickselect show a quadratic curve.

4 Reliability

RMedian is the only algorithm that can be considered in terms of reliability, as Quicksort and Quickselect are deterministic algorithms and always output a correct answer. RMedian either outputs a correct answer or fails, so it will never give a wrong answer. The algorithm succeeds with high probability, i.e., $Pr[Success] \geq 1 - 1/(n^{1/4})$. I checked this implementing a result checker that I

used over all experiments involving RMedian. I never found an error in more than 10^4 executions of RMedian. In any case, given the non-probable event that RMedian fails, it can be detected and executed again. In my opinion, all this arguments lead to the conclusion that, although RMedian is less reliable than Quicksort or Quickselect, is still a highly reliable algorithm.