Assignment 1- Randomized Algorithms

Marta González i Sentís

1 Introduction

The aim of this project is to compare different algorithms that compute the median of an input array of unordered keys, which we will restrict to belong to the set of $\mathbb{N}$.
The project will study 3 different algorithms, which we will denote by Rmedian, Qselect and Msort. The first two have been described in class and the last one, Msort, is the one consisting of sorting the input and going after position $\lceil \frac{n}{2} \rceil$. As discussed in class the complexity of Rmedian and Qselect is $O(n)$ in expectation while Msort is $O(n\lg n)$.
The codes have been implemented using C++ and some functions of standard libraries. In the next section we briefly explain the implementation and its characteristics. In section 3 there is a description of the type of inputs considered and section 4 contains the results for those inputs. Finally, the last section is dedicated to conclusions.

2 Implementation of the algorithms

Unless otherwise indicated, all the functions have been written by me. Now, let us shortly describe the 3 algorithms studied:

2.1 Rmedian

Rmedian is a randomized algorithm to compute the median of an array. As it is randomized, it can fail with a small probability when two conditions are not fulfilled. In order to detect these cases, the algorithm returns -1 when one of the two necessary conditions is not true. That is the reason why we restrict our array to have natural entries. If all integers were taken, the median value could be -1, and this indicator value wouldn't be useful any longer. Rmedian is implemented in file *Rmedian.cc.*

2.2 Qselect

Qselect is based on Quicksort and hence it uses the same partition function as Quicksort. In the code, the partition function has been randomized, so that the pivot is chosen randomly at each iteration. The code is given in the file *Qselect.cc.*

2.3 Msort

Msort simply sorts the array and returns position $\lceil \frac{n}{2} \rceil$. The sort algorithm is given by the library *algorithm* in C++. The implementation can be found in *Msort.cc.*

3 Types of input

As already mentioned, the input consists of arrays of natural numbers. More precisely, all the algorithms receive the value n followed by a (100xn) matrix consisting on 100 instances

of a natural array of size n.
In fact, Qselect and Msort can receive an integer array as input and in the case of Rmedian, only the "warning value" -1 should be changed for a value outside the range of possible entries (such as $\frac{1}{2}$).
The value of n has been changed in order to see how the three algorithms perform with respect to the size of the input. In particular the value of n has been considered up to 5x10^5.
Moreover, several types of input have been considered. First, we have taken random inputs so that we can know the order of the running time in expectation. And secondly some ad-hoc inputs have been taken into account in order to find good or bad behaviours of the algorithms. Such ad-hoc entries are constant arrays, ordered arrays in ascending and descending order and arrays with a low range of possible values.
Input can be easily generated using file *genInput.cc*.

4 Results

4.1 Random input

We begin testing the three algorithms with arrays of n random entries in the range 0-99. The entries are generated with the function rand() from the library *stdlib.h* using a different seed at each time.
The following table shows the average running time for each algorithm. For a fixed n, there is indicated the best running time with the color green and the worst with red (this convention will be used in the whole report).

n	Rmedian	Qselect	Msort
5E+01	3.34E-05 (0)	8.78E-06	2.33E-05
5E+02	1.10E-04 (0)	4.40E-05	1.26E-04
5E+03	4.95E-04 (0)	1.98E-04	1.10E-03
5E+04	3.30E-03 (0)	2.01E-03	1.30E-02
5E+05	2.53E-02 (0)	3.97E-02	1.55E-01
5E+06	2.09E-01 (0)	2.17E+00	1.81E+00

Table 1: Average running times (seconds) with respect to the input size n.
(0) in Rmedian indicates that 0 failed outputs were obtained

It can be seen that for small entries Qselect has a better performance while Rmedian wins for large values. Furthermore when enlarging n by a factor of 10 in Rmedian and Qselect, we get that the running time is also enlarged by a factor of 10 which is consistent with the fact that these two algorithms are linear on n in expectation ($O(n)$). Rmedian has a better performance in large arrays as it just sorts a "small" subset (the subset has size $\Omega(n^{\frac{3}{4}})$) of the whole array. Finally, Msort has the worst running time, as expected, due to the fact that is $O(n\text{lg}n)$ in average. Qselect is only slower in the case n = 5x10^6. That might happen because the sorting algorithm of Msort comes from a standard library which might be optimized in front of Qselect which uses the standard randomized partition.
We make a comparison plot of empirical running times for the 3 algorithms for n between 5x10^1 and 5x10^5.

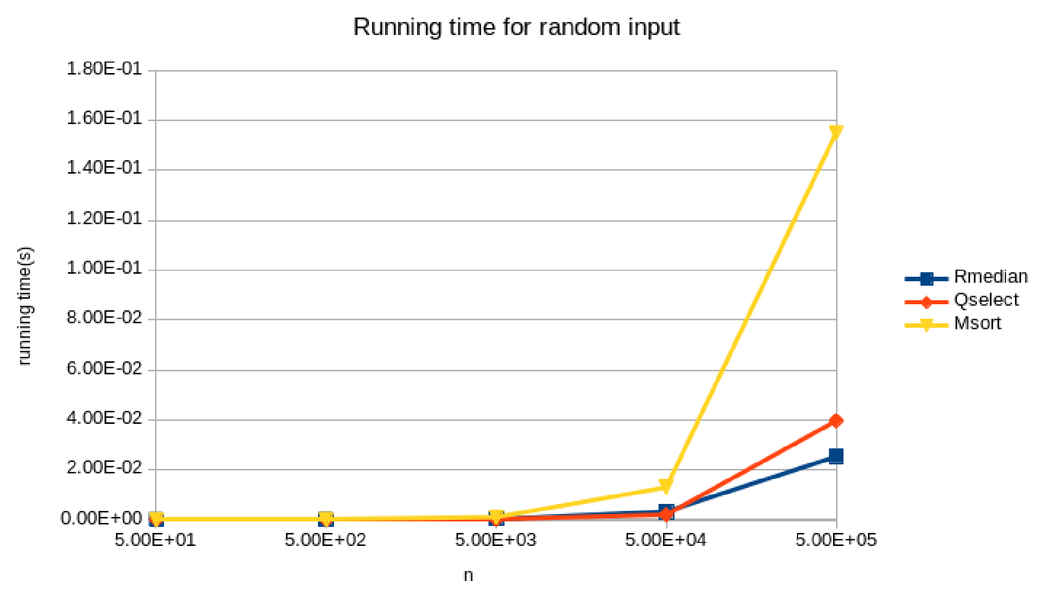

Figure 1: Empirical running times for different values of n

Again the running time of Msort is significantly larger than the ones of Qselect and Rmedian.

4.2 Ad-hoc input

- **Constant array**

  First consider the case where the array is constant, i.e. all the entries are equal. In this case, the vector C defined in Rmedian will contain all the entries of the array, so $|C| = n$. Hence the condition $|C| > 4n^{\frac{3}{4}}$ will be equivalent to $n > 4n^{\frac{3}{4}}$ and this will be true for all $n \geq 257$. Hence the algorithm will output fail always, except for the cases $n < 257$ where it basically will sort all the array and take the middle element (the algorithm in this case is the same as Msort).

  Taking into account the performance of the other 2 algorithms, we see that Msort runs much faster (see table 2). This is due to the fact that at each iteration of Qselect, the randomized partition function always takes the pivot to be the smallest element in the vector, leaving the whole array in one side of the partition. This yields to the worst case runtime of Qselect which is given by the recursion $T(n) = T(n-1) + O(n)$, which is $T(n) = O(n^2)$. Finally, Msort implements the sorting algorithm given by the *algorithm* library which has complexity $O(n\lg n)$ even for the worst case complexity.

n	Qselect	Msort
5.00E+02	3.75E-04	9.02E-05
5.00E+03	3.01E-02	9.75E-04
5.00E+04	2.98E+00	1.18E-02

Table 2: Running times in seconds for constant arrays for Qselect and Rmedian

- **Sorted array**
  In this case, the input is the sorted array in ascending order or the sorted array in descending order. In both cases, Qselect has a very good performance. This is because, the partition function deletes on average half of the array, leading to the recurrence $T(n) = T(\frac{n}{2}) + O(n)$, which is $O(n)$.

n	Rmedian	Qselect	Msort
5E+02	8.11E-05 (0)	1.03E-05	7.58E-05
5E+03	7.90E-05 (0)	6.17E-05	7.84E-04
5E+04	2.44E-03 (0)	5.44E-04	8.76E-03
5E+05	1.83E-02 (0)	5.49E-03	9.88E-02

Table 3: Running times when the input is sorted in ascending order

n	Rmedian	Qselect	Msort
5.00E+02	7.42E-05 (0)	1.90E-05	6.55E-05
5.00E+03	3.80E-04 (0)	1.07E-04	6.40E-04
5.00E+04	2.29E-03 (0)	9.69E-04	7.01E-03
5.00E+05	1.71E-02 (0)	9.55E-03	7.75E-02

Table 4: Running times when the input is sorted in descending order

Again Rmedian has given 0 failed outputs for all values of n.

- **Low range of values in the array**
  Consider the case where the entries of the input array must belong to the set $\{0, 1, 2\}$. The empirical complexity of the algorithms is given by the following table:

n	Rmedian	Qselect	Msort
5.00E+02	7.56E-05 (73)	8.30E-05	1.12E-04
5.00E+03	5.11E-04 (2)	3.57E-03	9.73E-04
5.00E+04	1.97E-03 (100)	3.29E-01	1.27E-02
5.00E+05	1.74E-02 (100)	3.26E-01	1.24E-02

Table 5: Random entries in range $\{0, 1, 2\}$

Here, the number of fails in Rmedian is very large. The reason is that when having such a small range of possible entries, the set C defined in Rmedian is very large. On average, the size of C is $O(n)$, so the condition $|C| > 4n^{\frac{3}{4}}$ is always fulfilled for large n.
In the case of Qselect, the running time is worse than when considering random inputs. This is because the partition function can easily return an unbalanced partition which causes many repetitions to be done in Qselect.

5 Conclusions

In this report we have done an empirical analysis of 3 algorithms that find the median of an input array of natural entries.

First, we have considered random inputs so that we could get an approximation of the expected running time for the 3 algorithms. We have obtained that, for small entries Qselect has the best performance, while Rmedian won for large values of n. Moreover, as seen in class, the probability of fail in the Rmedian algorithm decreases as the input size increases, so it is the best option for large inputs.

Secondly, some ad-hoc inputs have been taken into account in order to force bad or good behaviours. For constant arrays, Rmedian always returns fail and for low range input arrays, the number of fails is also very large. In the case of Qselect, constant arrays lead to the worst case complexity, $O(n^2)$, and gives worse order than in the random case (see table 1 and 3). Thus, Msort, which uses an optimized sorting algorithm yields the best results for constant and lowed ranged input arrays.
In the case of sorted arrays, the best running times arise when considering Qselect.

To sum up, although some ad-hoc inputs yield to fail when using Rmedian, the expected running time for Rmedian is the best one for n sufficiently large and the probability of fail for such inputs is very very small. In fact, for all non ad-hoc inputs, we have obtained 0 failed outputs using Rmedian. For small inputs though ($n < 5e4$), Qselect performs better and it is more reliable due to the fact that the probability of fail of Rmedian for small inputs is not that small. Finally, Msort has been proved to have a worse performance on average which is what we expected as it is $O(n\lg n)$ in average while Rmedian and Qselect are $O(n)$.