AAPP Challenge Report

Giacomo Orsenigo - Francesco Spangaro - Tosetti Luca

22 October 2024

Academic Year 2024 - 2025

Contents

1 Introduction

The aim of this project is to implement the deterministic version of the select-ith problem, show its asymptotic complexity and compare it with the random version.
The problem consist in finding the i-th smallest element in an array. Unlike the random version, the deterministic one tries to select a good pivot to partition the array. In this way the algorithm has linear time complexity even in the worst case.

2 Design choices

We implemented the algorithm in C, following the guidelines present in the slides. Regarding the design choices we made:

- We applied the divide-and-conquer approach as suggested by the slides:
  - **Error case:** If the number of elements contained in the array is less or equal to 0 or the selection index "i" is smaller than 0, or it exceed the given input array's size ("n") the algorithm terminates immediately since the parameters given to the algorithm may result in an undefined computation behaviour.
  - **Base case (# of elements in input array $\leq$ 10):** In this case we simply perform a naive approach to solve the problem. We use the C language built-in qsort() function in order to sort the input array, select and return the i-th element.
  - **Recursive case (# of elements in input array $>$ 10):**
    1. We divide the array in groups of 5 element each, and we search the median of each of them by using the *median5()* function which uses SIMD min/max functions to find it in a very efficient way.
       Differently from what has been shown in the slides, we also consider the last group of the input array which may have a size smaller than 5. To compute the median of such group since we cannot use the *median5()* function we opted to use the qsort() built-in C function to sort the group and select the middle element.
    2. We arrange the medians in a new array. At this point we have an array of integer containing the medians of each group made of the elements of the original input array.
    3. We perform a **recursive call** to the select-ith algorithm passing him the medians' array to find the median of the medians.
       At the end of the call we obtain the median of the medians' value of the input (whether it is an array of medians coming from another recursive call of the algorithm, or the original input array of the algorithm).
    4. At this point we partition the input array, using as pivot the median of medians' value. We perform the partition with a linear time algorithm, by moving all the elements smaller w.r.t the pivot on the left side of the array, all elements equals to the pivot in the middle and the others on the right side.
    5. Finally we recursively call the select algorithm on the lower or upper half of the input array based on whether the index "i" is respectively lower or greater than the pivot's index "k". If indexes "i" and "k" are equals then we have found our element.
- Another important design choice, was to implement a *Three-way partition* algorithm, instead of the one proposed in the slides. With such algorithm we divide the array to partition into three different parts: the elements whose value is less than the pivot's one, those with a value equal to it, and those with a value greater than it.
  We opted for this implementation since, compared to the one proposed in the slide, it ensures that the median-of-medians computation maintains linear execution time even in a case of many or all coincident elements.

3 Experimental setup

We used Google Colab to build, test and profile the program.
We implemented the two version of the algorithm (deterministic and randomized) and a naive one (using qsort to sort the entire array) as a reference to compare the results and we created some test to validate our implementation.
For both deterministic and random version we have:

- Normal behaviour tests, to test the program in typical cases
- Duplication tests, to test the program with a lot of duplicated elements
- Edge case test, to test the program when all elements of the array are equals

Each type of test is executed with different sizes, starting from a tiny array to a stress test with several thousands of elements, and it's repeated multiple times, to ensure that the algorithm hasn't guessed the selected element only by chance.
Data of each test is generated randomly and the results are then compared with the one obtained with the naive algorithm.

4 Performance measurements

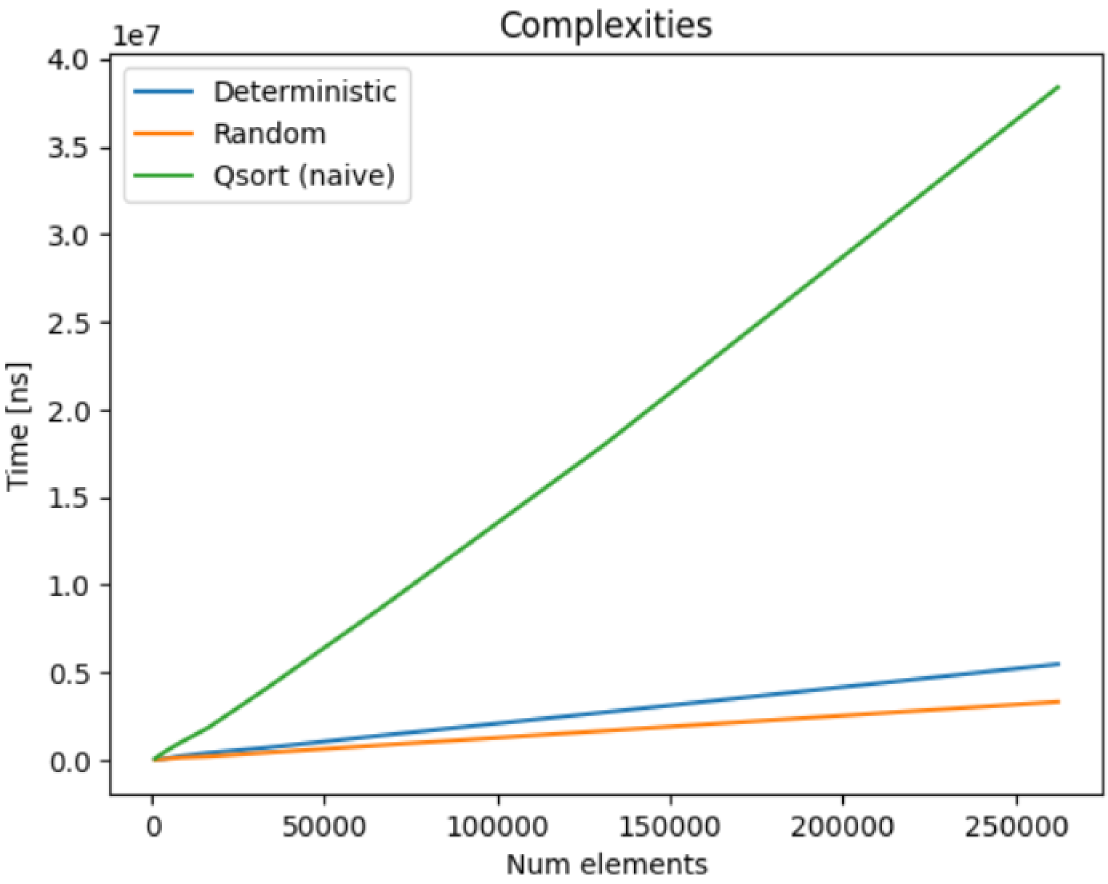
In order to properly measure the performance of the select-ith algorithm we implemented, we choose to use Google's benchmark library. We tested the performance of our implementation of the deterministic select-ith algorithm comparing it to the version that randomly chooses a pivot and to C's built-in qsort implementation.

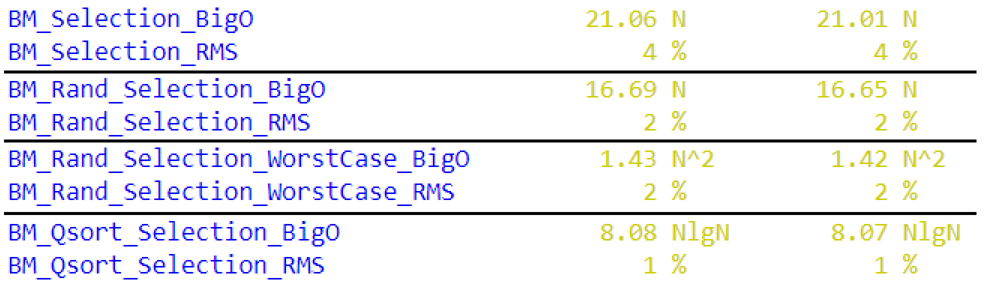
Theoretical expectations:			
	Best	Average	Worst
Deterministic	O(N)	O(N)	O(N)
Random	O(N)	O(N)	O(N^2)
Qsort	O(NlogN)	O(NlogN)	O(N^2)

Experimental results were in line with the theoretical ones written above. To test this, we ran multiple benchmarks for each sorting algorithm, with arrays' sizes ranging from 2^{10} to 2^{18}, each one filled with random values.
To obtain more stable results in the average case, we decided to initialize the array with random values before each iteration of the benchmark. We found that:

- The deterministic version correctly computes in a time **O(kN)**, with k constant.
- The random version correctly computes in a time **O(bN)**, with b constant and $b << k$.
- The qsort correctly computes in a time **O(cNlogN)**, with c constant.

The benchmark's results were then fed to a python script in order to generate a graph of the execution time compared to the number of elements in the arrays.





```
BM_Selection_BigO                       21.06 N           21.01 N
BM_Selection_RMS                            4 %               4 %
BM_Rand_Selection_BigO                  16.69 N           16.65 N
BM_Rand_Selection_RMS                       2 %               2 %
BM_Rand_Selection_WorstCase_BigO         1.43 N^2          1.42 N^2
BM_Rand_Selection_WorstCase_RMS             2 %               2 %
BM_Qsort_Selection_BigO                  8.08 NlgN         8.07 NlgN
BM_Qsort_Selection_RMS                      1 %               1 %
```

5 Conclusion

Our experimental results confirms what explained during our lectures: the deterministic version is better when considering the worst case scenario, but given the average case, since constant "b" is much smaller than constant "k", the random version performs better almost always.