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 aymptotic complexity of random version of the same problem.
The problem consists in finding the ith smallest element in an array. Unlike the random version, the deterministic version tries to select a good pivot to partition the array with. This way the algorithm has linear time complexity even in the worst case scenario.

2 Design choices

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

- We applied the divide-and-conquer approach as suggested by the slides:
  - **Error case:** In case the number of elements contained in the array is less or equal to 0, the selection index "i" is smaller than 0 or it exceeds the given input array's size "n", the algorithm terminates immediately. Here the algorithm after it being fed said parameters, may result in an undefined computation behaviour.
  - **Base case (# of elements in input array $\leq$ 10):** In this case we perform a naive approach to solve the problem. We use C's built-in qsort() function in order to sort the input array, select and return the ith element.
  - **Recursive case (# of elements in input array $>$ 10):**
    1. We divide the array in groups of 5 elements each, we find the median of each of the subarrays by using the *median5()* function which uses SIMD min/max functions to compute it in a very efficient way. Differently from what has been shown in the slides, we also consider the last group of the input arrays which may have 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.
    3. We perform a **recursive call** to the select-ith algorithm, passing it the medians' array in order to find the median of the medians. At the end of the computation we obtain the median of the medians' value.
    4. We partition the input array, using as pivot the median of medians' value. We partition with a linear time algorithm by moving all the elements smaller than the pivot on the left side of the array, all elements equals to the pivot in the middle and all elements greater than the pivot on the right side of the array.
    5. 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 equal then we have found our element.
- We implemented a *Three-way partition* algorithm, instead of the one proposed in the slides. With such algorithm we divide the array into three different partitions: the elements whose value is less than the pivot's, 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 slides, 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 versions of the algorithm (deterministic and randomized) and a naive one (using qsort to sort the entire array) as a reference to compare the results. We created some tests 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 equal.

Each type of test is executed with different sized arrays, starting from a smaller array of 5 elements to a stress test with 10000 of elements, and it's repeated multiple times to ensure that the algorithm hasn't guessed the selected element only by chance.
The arrays used in testing are generated randomly and the results are then compared with the ones obtained with the naive algorithm.

4 Performance measurements

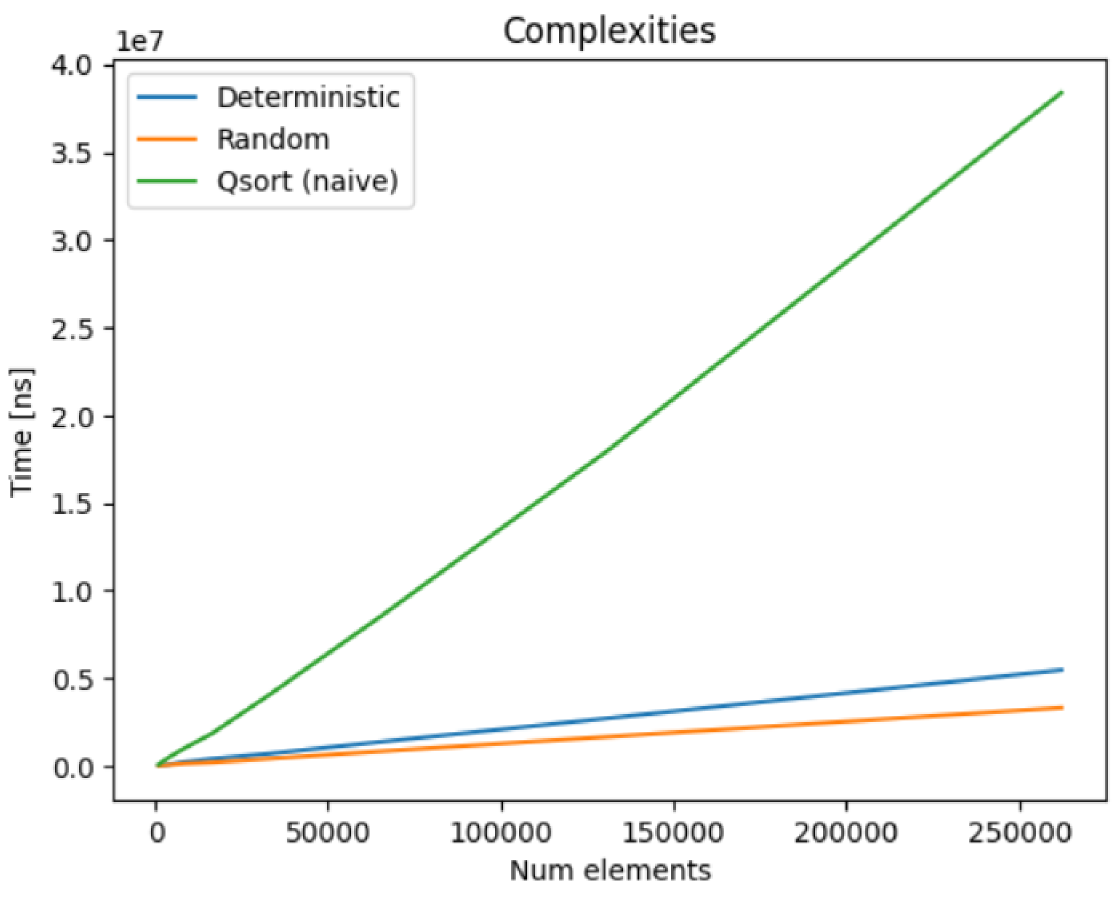
In order to properly measure the performance of the select-ith algorithm we implemented, we choose to use Google's benchmark library. We tested the performance of our implementation of the deterministic select-ith algorithm comparing it to the version that randomly chooses a pivot and to C's built-in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

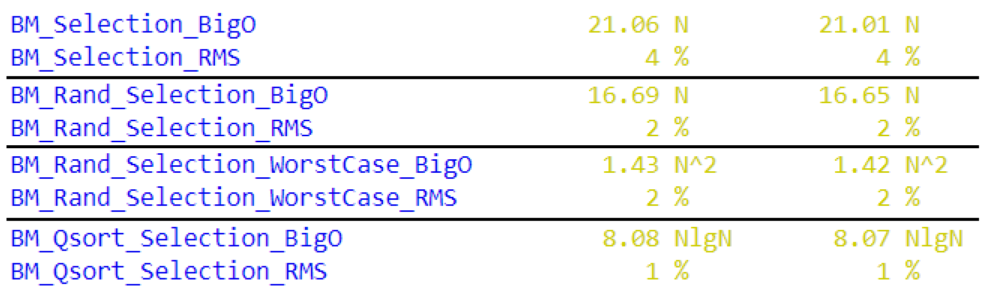
Theoretical expectations:			
	Best	Average	Worst
Deterministic	O(N)	O(N)	O(N)
Random	O(N)	O(N)	O(N^2)
Qsort	O(NlogN)	O(NlogN)	O(N^2)

Experimental results were in line with the theoretical ones reported in the table 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 %
```

We decided to not include the worst case scenario in the plotting since it's scale was completely off and it would have impacted the plot's readability.

5 Conclusion

Our experimental results confirmed what explained during the lectures: the deterministic version is better when considering the worst case scenario, but in the average case, since constant "*b*" is much smaller than constant "*k*", the random version performs better almost every time.