# Advanced Algorithms and Parallel ProgrammingChallenge1: **“Implement the deterministic algorithm for the Select-ith problem”** Authors: Francesco Ferlin (10717750), Francesco Riccardi (10741078) and Federico Saccani (10700471) Date: 19/10/2024

# Experimental setup

We performed different tests in order to check the correct implementation of the algorithm.  
Different tests check for the correctness of specific edge cases, but they can be broadly grouped into a few categories:

* Few elements (n < 5): since the select implementation, in the first step, groups elements by 5 and discards the remaining, we check that it also works if an array of less than 5 is provided
* Number of elements not multiple of 5, arrays larger than 5: for a similar reason as above
* Tests that take a specific branch of recursion: to test that all the math around size and rank updating works correctly
* Tests with a lot of duplicates: to ensure that the algorithm works correctly even in the presence of duplicates, even if it may not respect the O(n) time complexity
* Huge size and random elements: to perform stress testing

To ensure the correctness, we perform a comparison with a naïve implementation, using the standard library function qsort to retrieve the i-th element (which is higher than i-1 elements) and then we check if the implemented algorithms return the same value. Additionally, we also print a lot of info on the console, in order to be able to manually evaluate whether the results are correct.

Test methods are implemented using macros in order to reduce the clutter and boilerplate.

# Performance measurements

We run 3 separate benchmarks, the first for the deterministic select implementation, the second for the random select implementation and the third for a naïve implementation using qsort.

In all benchmarks, we generate random arrays at different sizes, starting from 4 until 2^18. We make sure there are no duplicate elements in all arrays, as our deterministic select implementation has guaranteed O(n) only in this case.

Additionally, we test by generating a random array for each benchmarking iteration (so by moving the generation inside the loop and stopping+resuming the timer just before and after it). This is because we do not want our measurements to be influenced by a ‘lucky’ or ‘unlucky’ generation of a single random array, which would make it so a specific array size (eg BM\_select/8192) would perform all iterations with said array and end up with a time which is much higher or much lower compared to an expected average run and mess with the complexity calculation.

We also use manual timing as PauseTiming/ResumeTiming seem to have a bit of overhead, even if it doesn’t seem to influence the complexity calculation, but just increase the times a bit.

The benchmarks’ results show the expected theoretical time complexity for both the deterministic and randomized version.  
In particular, for the deterministic version we obtain Linear Time Complexity (~16N).  
For the randomized version, we also obtain Linear Time Complexity (~13N), as we are benchmarking the average case and not the worst one.   
The qsort is there just for comparison (it should have a nlogn complexity in the average case).

We can see the results in the following plotted graphs:

Immagine che contiene testo, linea, diagramma, Diagramma

Descrizione generata automaticamente  
  
Although both the deterministic and the random version are linear time, we can see that the multiplication coefficient “c” is larger for the first one, which is in line with the theoretical result.

# Design choices

The C code closely resembles the pseudo code seen in class, with minor differences which we will highlight here.

## **Deterministic Version**

If the function is called with len < 5, we compute the required rank over a small array by using the qsort algorithm to find it, as the algorithm wouldn’t otherwise work.

The calculation of the median of each group of 5 is implemented using a method which uses at most 6 comparisons, which can be considered constant time. This is because it was unspecified in the slides, and using a sorting-based naïve method would increase both the times and the ‘c’ coefficient in the benchmarks.

The medians of all the groups are stored in the first n/5 positions of the array, avoiding allocating additional memory. This is because memory allocation on the heap has a non-deterministic time complexity, so it’s best to avoid it. The algorithm should still work correctly, as by storing in the first positions, we do not interfere with successive median calculations. This optimization is also done in the [Wikipedia pseudocode of the method](https://en.wikipedia.org/wiki/Median_of_medians#Algorithm).

## **Randomized Version**

We use the standard library function ‘rand()’ to generate random numbers. This function needs to be appropriately seeded beforehand by calling srand(seed), which is not done inside the function in order to follow C conventions.