# 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.  
The Tests can be divided into 3 categories depending on the number of elements (n) inside the array:

* Few elements: n<=5
* Medium elements: n>5 and n<=50
* High elements: n>50
* Random size and Random elements

To check the correctness, we perform a quick sort retrieving the i-th element (required rank) and then we check if the implemented algorithm returns the same value (the tests use MACROs to perform the checks).  
We consider both cases where there are duplicates and those where there are not.

# Performance measurements

< GRAPH >

< GRAPH >

< GRAPH >

< GRAPH >

< GRAPH >

The benchmarks clearly show the expected theoretical Time Complexity for both the deterministic and randomized version.  
In particular, for the deterministic version we obtain Linear Time Complexity in all the benchmarks.  
For the randomized version, we also obtain Linear Time Complexity despite some rare cases in which the Time Complexity is quadratic due to the unlucky cases in which the pivot corresponds to the min or Max element in the array.  
  
Although the deterministic version is always Linear time, the multiplication coefficient “c” in this case, is larger than the one of the randomized version.  
This result is in line with the expected theoretical result.

# Explanation of design choices

The C code is divided into 4 different functions:

1. int \*our\_select(int arr[], size\_t len, int rank); <--- Deterministic Version
2. int \*rand\_select(int arr[], size\_t len, int rank); <--- Randomized Version
3. size\_t partition(int arr[], size\_t len, int \*pivot);
4. int \*median\_of(int arr[], size\_t len); <--- Only used for the Deterministic Version

## **Deterministic Version**

int \*our\_select(int arr[], size\_t len, int rank) is the main function of the program and it implements the deterministic selection algorithm seen during the lectures.

If the function is called with len<=5 then we compute the required rank over a small array considering constant time by using the qsort algorithm to find it.

If len>5 we proceed with **Step1** dividing all the n elements of the array into groups of 5 elements each.  
We compute the median of each group storing them in the first n/5 positions of the array avoiding to allocate additional spaces (this optimization is also done in the [Wikipedia pseudocode of the method](https://en.wikipedia.org/wiki/Median_of_medians#Algorithm)).  
To calculate the median for each group of 5 elements we rely on the function int \*median\_of(int arr[], size\_t len).  
We call this function foreach of the subgroup of 5 elements: with arr=the start of the subarray of 5 elements and rank=3 (because we want to compute the median).  
The function simply calls recursively our\_select(arr, len, 1 + len / 2) and like said before, perform the median computation with the qsort algorithm (just because we have only 5 elements and we can assume constant time).

After calculating all the medians, we can proceed with **Step2** computing recursively the median of the medians.  
Similarly as before, we compute it by calling median\_of(medians, len\_mul\_5 / 5); with medians=the medians found at Step1 and len=#medians.  
Therefore, Step2 perform the computation recursively relying on int \*our\_select(int arr[], size\_t len, int rank).

After that, **Step3** computes the partition by calling size\_t partition(int arr[], size\_t len, int \*pivot), with pivot=the median of the medians (found at Step2).  
The partition function implements the pseudocode seen during the lectures and modifies the array passed as argument dividing it into 3 parts (1st part: with all the elements <= pivot; 2nd part: pivot; 3rd part with all the elements > pivot).  
  
At the end, we conclude with **Step4**:  
If the required rank = pivot\_index+1 we return the pivot, otherwise we call recursively int \*our\_select(int arr[], size\_t len, int rank) on the left subarray (from 0 to pivot-1) or right subarray (from pivot+1 to len) depending on the required rank.

## **Randomized Version**

The function int \*rand\_select(int arr[], size\_t len, int rank) is used for the randomized version of the algorithm.   
We use the same logic as above with the only difference in the selection of the pivot element.  
This time, the pivot element is randomly chosen and may not be “a good pivot” as in the deterministic version.