Selection algorithms challenge

Christian Rossi (10736464) - Kirolos Shroubim (10719510) - Antonio Sulfaro (10742266)
October 21, 2024

1 SELECTION PROBLEM

The selection problem involves finding the k-th smallest or largest element in an unsorted array or list of numbers. The goal is to determine the k-th order statistic (i.e., the k-th smallest or largest element) efficiently, without necessarily sorting the entire array.

In this work, we will implement two algorithms for solving the selection problem: the Median of Medians (standard version) and Quickselect (randomized version).

2 EXPERIMENTAL SETUP

To evaluate the performance of both Quickselect and Median of Medians of the algorithms, we implemented them in C. In the file `ith_element.c`, two functions are defined for each version: `getIthElement` for Median of Medians and `getIthElementRand` for Quickselect.

For testing, we utilized the Google Test framework to verify the correctness of both implementations. We designed common test cases for both algorithms, which include arrays with negative numbers, ordered and unordered arrays, and cases that select the smallest, largest, and median elements. Additionally, we included arrays with repeated elements to ensure robustness.

For benchmarking, we employed the Google Benchmark library to measure the time complexity of the algorithms. We generated random arrays of increasing sizes to compare the performance of both versions. The benchmarks take two arguments: the size of the randomly generated array and the index of the element to be selected. All test arrays may contain repeated elements to simulate real-world scenarios.

3 PERFORMANCE MEASURMENT

Since our benchmarks use random array initialization, we run the tests multiple times to ensure accurate results and minimize errors. The benchmarks include two versions: one with unordered arrays and another with pre-sorted arrays.

UNORDERED ARRAYS We first analyze the performance of both algorithms on unordered arrays. Theoretically, we expect Quickselect to be faster on average compared to Median of Medians. Our experimental results are as follows:

Version	**Average complexity found with benchmark**	**Temporal complexity**
Standard	$T(n) \approx 100n$	$\mathcal{O}(n)$
Random	$T(n) \approx 35n$	$\mathcal{O}(n)$

Table 3.1: Ordered array benchmarks

As shown, both algorithms exhibit linear time complexity, but Quickselect has a significantly smaller constant factor, making it faster in practice ($35n$ vs. $100n$).

ORDERED ARRAYS Next, we consider the case of pre-sorted arrays. In this scenario, we expect Quickselect to still outperform Median of Medians. Note that for sorted arrays, the task simplifies to returning the i-th element directly. The results are as follows:

Version	**Average complexity found with benchmark**	**Temporal complexity**
Standard	$T(n) \approx 50n$	$\mathcal{O}(n)$
Random	$T(n) \approx 5500$	$\mathcal{O}(1)$

Table 3.2: Unordered array benchmarks

As expected, both algorithms perform better than in the unordered case, with Quickselect achieving constant time complexity due to the simplicity of directly returning the element.

4 DESIGN CHOICES

Median of Medians is implemented using the `getIthElement` function. This function iteratively calls `select`, which repeatedly selects elements until the correct index is found. The `select` function relies on two helper functions: `partition` and `partition5`. The `partition` function divides the array into two parts, while `partition5` handles sub-arrays of five elements.

In this algorithm, the array is divided into groups of five. The median of each group is found, and then the medians of these groups are recursively used to find a pivot. We chose groups of five because smaller groups provide too little information, while larger groups would require more sorting, reducing efficiency.

For Quickselect, we skip the process of dividing the array into groups and simply select the pivot randomly from the entire array.

The key difference between the two approaches is that the randomized version does not require sorting small groups, making it more efficient on average. However, once the pivot is chosen, both algorithms proceed similarly, as the randomized version also uses the same `partition` function from the standard algorithm.