Selection algorithms challenge

Christian Rossi (10736464) - Kirolos Shroubim (10719510) - Antonio Sulfaro (10742266)

October 22, 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 efficiently, without necessarily sorting the entire array.

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

2 EXPERIMENTAL SETUP

For the development and testing of the algorithms we have used Google Colab.

For testing, we utilized the Google Test framework to verify the correctness of both implementations.

For benchmarking, we employed the Google Benchmark library to measure the time complexity of the algorithms.

3 PERFORMANCE MEASURMENT

The benchmarks include two versions: one with unordered arrays and another with pre-sorted arrays.

An important value to check is the Root Mean Square error (RMS) because it provides a measure of variability in performance data. We aimed to maintain it low, but at the same time, we wanted to ensure that we could still benchmark very large arrays effectively.

UNORDERED RANDOM ARRAYS Theoretically, we expect Quickselect to be faster on average compared to Median of Medians. Our experimental results are as follows:

Version	**Average complexity (benchmark)**	**RMS**	**Temporal complexity**
Standard	$T(n) \approx 105n$	$\approx 3\%$	$\mathcal{O}(n)$
Random	$T(n) \approx 33n$	$\approx 7\%$	$\mathcal{O}(n)$

As shown, both algorithms exhibit linear time complexity, but Quickselect has a significantly smaller constant factor, making it faster in practice.

ORDERED RANDOM ARRAYS Since our algorithm doesn't check if the array is already sorted, it cannot simply return the i-th element. The random version, in this case, carries a greater risk of encountering the worst-case scenario, which has a quadratic time complexity. This can occur if it consistently chooses the first or last element as the pivot, so the partitioning would yield highly unbalanced subarrays, requiring many recursive calls to narrow down the selection. The results are as follows:

Version	**Average complexity (benchmark)**	**RMS**	**Temporal complexity**
Standard	$T(n) \approx 62n$	$\approx 1\%$	$\mathcal{O}(n)$
Random	$T(n) \approx 62n$	$\approx 1\%$	$\mathcal{O}(n)$

Nevertheless, as evidenced by our results, Quickselect still outperforms the Median of Medians in practice. This is primarily because it is unlikely that the randomly chosen pivot will consistently be the worst-case option. In this specific case, the results appear to be constant, likely due to the small size of the array, which may not adequately demonstrate the linear complexity. However, we have chosen to include these results in our report to illustrate the significant speed advantage of Quickselect in real-world scenarios, highlighting the effectiveness of randomized algorithms.

4 DESIGN CHOICES

IMPLEMENTATION 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.

TESTING 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.

BENCHMARK We conducted performance benchmarks on arrays of varying sizes to evaluate both algorithms across different scales. Since both algorithms modify the input, the overhead of copying arrays for each test could impact results, but this should be minimal due to the linear nature of both copying and the algorithms' average complexity.

The benchmarks involved generating random arrays of different sizes and selecting an element at a specified index. To provide a more comprehensive evaluation, we also included tests using pre-sorted arrays, as randomly generated arrays are unlikely to be fully sorted. The arrays may contain repeated elements to reflect real-world scenarios.