**Implementation and Analysis of Selection Algorithms**

Asrith Krishna Vejandla

Student ID: 005033913

MSCS-532 Algorithms and Data Structures

Assignment 6

Professor Satish Penmatsa

February 9th, 2025

**Selection Algorithms**

**Implementation of Selection Algorithms**

Selection algorithms are crucial in computer science, particularly when finding the kth smallest element in an unsorted array. Two prominent approaches to this problem are the randomized Quick Select algorithm and the deterministic Median of Medians algorithm. Both algorithms are modifications of the classic Quicksort partition scheme but employ different strategies for pivot selection to achieve their respective time complexities.

The QuickSelect algorithm, implemented here as randomized\_select, is a probabilistic approach to randomly selecting a pivot element for partitioning. This randomization helps avoid the worst-case scenario of consistently selecting poor pivots.

A screenshot of a computer code

AI-generated content may be incorrect.

On the other hand, the Median of Medians algorithm employs a more sophisticated deterministic approach by recursively selecting a good enough pivot by carefully grouping elements and finding their medians.

**A screenshot of a computer code

AI-generated content may be incorrect.**

**Time Complexity Analysis**

The randomized Quick Select algorithm achieves an expected time complexity of O(n). This expected linear time complexity stems from the random pivot selection, which provides a high probability of achieving a reasonably balanced partition. In each recursive call, the algorithm processes a fraction of the input size, and through probabilistic analysis, we can show that the expected number of comparisons sums to linear time. However, in the worst case (which is rare due to randomization), the time complexity could degrade to O(n²) if extremely unfortunate pivot selections occur consistently.

Conversely, the Median of Medians algorithm guarantees O(n) worst-case time complexity through its deterministic pivot selection strategy. This guarantee is achieved by ensuring that the selected pivot always eliminates a constant fraction of the elements. The algorithm divides the input into groups of five elements, finds their medians, and recursively selects the median of these medians as the pivot. This process ensures that at least 30% of the elements are less than the selected pivot and at least 30% are greater, guaranteeing balanced partitions.

The recurrence relation for the Median of Medians algorithm can be expressed as T(n) ≤ T(n/5) + T(7n/10) + cn, where the first term represents the recursive call to find the median of medians, the second term represents the recursive selection in the reduced array. “cn” represents the linear work done at each level. Through the Master Theorem and substitution method, this recurrence solves to O(n).

**Space Complexity**

The space complexity considerations for both algorithms reveal interesting trade-offs. Due to the recursion stack depth, the randomized QuickSelect algorithm has a space complexity of O(log n) in the average case. This logarithmic space requirement is achieved because only one recursive call is active at any time, and the expected balance of partitions leads to logarithmic depth.

While maintaining the same O(log n) recursive stack depth, the Median of Medians algorithm requires additional space for storing the medians of sub-lists. The algorithm creates sub-lists of size 5 and stores their medians, requiring O(n/5) additional space. However, this extra space can be optimized through in-place implementations, though at the cost of code complexity and constant factors in the time complexity.

**Implementation Overhead Considerations**

While both algorithms achieve linear time complexity (either in expectation or worst-case), they differ significantly in their practical performance due to hidden constants and implementation overheads. The randomized Quick Select typically performs better in practice due to its simplicity and lower constant factors. While theoretically superior in worst-case scenarios, the Median of Medians algorithm incurs higher overhead from its complex pivot selection process, including creating and sorting sub-lists and the recursive median finding.

The code implementation provided demonstrates these trade-offs, with the randomized version generally executing faster on average inputs despite lacking the worst-case guarantees of the deterministic version. This practical difference highlights the important distinction between theoretical complexity analysis and real**-**world performance characteristics.

**Empirical Analysis**

The randomized selection algorithm (Quick Select) demonstrates interesting behavior across different input distributions. For smaller input sizes (up to about 5000 elements), all input types show similar performance with execution times under 0.001 seconds, suggesting that the randomization effectively mitigates any adverse effects from input distribution at these sizes. However, as the input size grows, we observe divergent behavior patterns. The algorithm performs most efficiently on sorted data, maintaining a near-linear growth in execution time. This is somewhat surprising given that sorted arrays typically cause issues for quicksort-based algorithms, but the randomization effectively prevents the worst-case scenario.

Random and duplicate data sets show moderate performance with execution times scaling roughly linearly, though with some fluctuations. The most interesting observation is the behavior with reverse-sorted data, which shows inconsistent performance, especially at larger input sizes (15000-20000 elements). This volatility in execution time for reverse-sorted data suggests that the randomization doesn't always avoid poor pivot selections in this case.

In contrast, the Median of Medians algorithm shows remarkably consistent behavior across all input distributions. The execution time grows linearly with input size for all data types, with almost identical performance for random and sorted data and slightly higher but still linear growth for duplicate data. This empirical observation strongly supports the theoretical O(n) worst-case time complexity. The algorithm's deterministic nature is evident in the smooth, predictable growth of execution time regardless of input distribution.

Comparing the two algorithms directly, we can see that the Randomized Select generally performs faster for smaller inputs, with execution times about half those of Median of Medians for inputs up to 10000 elements. However, the Median of Medians algorithm shows superior predictability in its performance. While its execution times are consistently higher (about 0.01 seconds vs 0.006 seconds for 20000 elements), the growth is steady and reliable.

These empirical results align well with theoretical expectations. The randomized algorithm's average-case O(n) complexity is reflected in its generally good performance, but its susceptibility to poor random choices is visible in the variability of its execution times, particularly for reverse-sorted data. The Median of Medians algorithm's guaranteed O(n) worst-case complexity is demonstrated by its consistent linear growth across all input types, though with a higher constant factor as evidenced by the steeper slope of its timing curves.

This analysis suggests that the choice between these algorithms might depend on specific requirements. Median of Medians would be the better choice if predictable performance is crucial despite its higher constant factors. However, if average-case performance is more important and some variation in execution time is acceptable, the Randomized Select algorithm generally offers better raw performance, particularly for smaller inputs or well-distributed data.

**A graph with colorful lines and numbers

AI-generated content may be incorrect.A graph with lines and numbers

AI-generated content may be incorrect.**

**References**

1. Quick Select Algorithm - <https://www.geeksforgeeks.org/quickselect-algorithm/>
2. Median of Medians - <https://gist.github.com/boulethao/a15809963d326a5ad43f255fbffbf9ff>