Medians and Order Statistics & Elementary Data Structures

Minh B. Le

University of the Cumberlands

MSCS-532-B01: Algorithms and Data Structures

Dr. Vanessa Cooper

July 29, 2025

Introduction

This paper focuses on two main parts: Selection Algorithms and Elementary Data Structures. We will make an effort to implement algorithms related to these topics in Python, analyze their performances, and discuss their practical applications. All codes referenced in this paper can be found in this GitHub repository: https://github.com/leminh646/Assignment-6.

Selection Algorithms

The first part of the paper aims to implement and analyze selection algorithms deterministically and randomly. The chosen algorithms are Median of Medians (deterministic) and Randomized Quicksort (random). We will first conduct a theoretical analysis, then run a benchmark to compare them empirically. All codes for this part will be stored in the 'selection_algos' folder in the provided GitHub repository.

Performance Analysis

Median of Medians

The deterministic selection algorithm, known as the Median of Medians algorithm, achieves worst-case linear time complexity by carefully choosing a pivot that guarantees balanced partitioning. The core idea is to divide the input into groups of five, find the median of each group, and recursively select the median of these medians as the pivot. This process ensures that the pivot lies within the 30th to 70th percentile of the list, guaranteeing that at least a constant fraction (at least 30%) of elements are discarded in each recursive step. Consequently, the recurrence relation for the runtime is $T(n) \leq T(n/5) + T(7n/10) + cn$, which solves to $O(n)$ (Blum et al., 1973). This deterministic approach avoids the worst-case scenarios associated with poor pivot choices, making it highly reliable, especially in systems where worst-case performance matters, such as in real-time or embedded systems.

Randomized Quicksort

In contrast, the randomized QuickSelect algorithm achieves an expected time complexity of $O(n)$, but not in the worst case. It works by choosing a pivot at random and partitioning the input into elements less than, equal to, and greater than the pivot. If the pivot is poorly chosen, such as being the smallest or largest element, the algorithm may only reduce the problem size by one element per step, leading to a worst-case time of $O(n^2)$. However, on average, the random pivot results in reasonably balanced partitions, leading to the expected recurrence relation $T(n) = T(n/2) + O(n)$, which solves to $O(n)$ (Cormen et al., 2022). The randomness helps prevent consistently poor pivot choices, making it efficient for general-purpose use and competitive on average with other selection methods.

Space complexity

Regarding space complexity, both algorithms operate in $O(n)$ space due to their recursive nature and the temporary sublists created during partitioning. However, the recursive depth differs. The randomized version has an expected depth of $O(logn)$, while the deterministic version, though worst-case optimal in time, can have a deeper recursive structure due to its two recursive calls: one to find the pivot and another to select within a partition. This introduces slightly more overhead in practice. Additionally, the deterministic algorithm involves more work in pivot selection (grouping and sorting mini-sublists), which contributes to larger constant factors, making it slower in practice despite its better asymptotic guarantee (Hoare, 1962; Blum et al., 1973).

Empirical Analysis

'benchmark.py' was created to compare the runtime of deterministic and randomized selection algorithms. The program is designed to test on different input sizes and distributions

(sorted, reverse-sorted, random). A visualization of the results is shown in Figure 1 using matplotlib in Python.

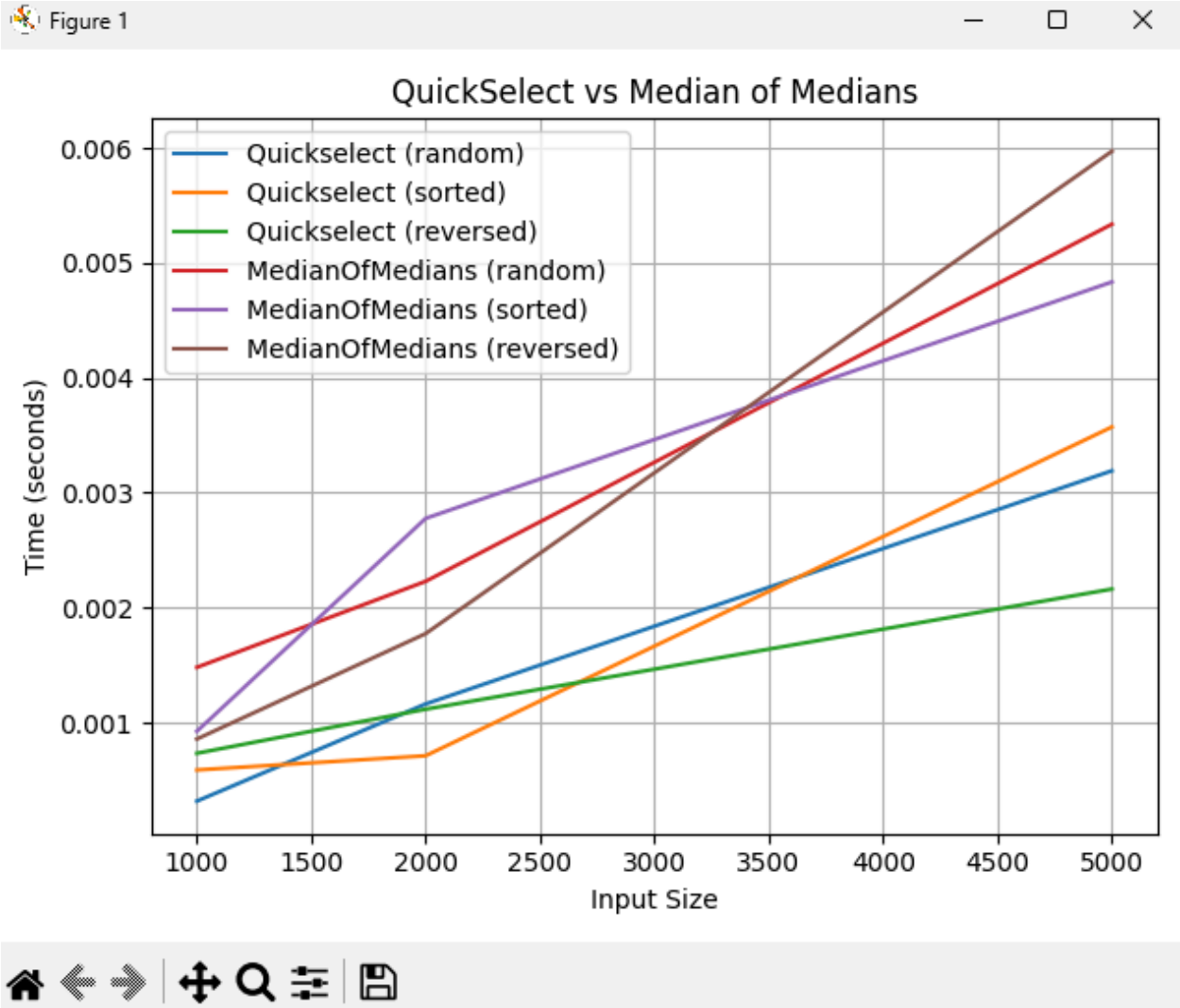

Figure 1: QuickSelect vs Median of Medians in matplotlib

Across varying input sizes and distributions, the Randomized QuickSelect consistently outperforms the deterministic approach in terms of execution time. These observations align closely with the theoretical expectations surrounding both algorithms' time complexity and operational overhead.

For randomly distributed inputs, QuickSelect achieves times of 0.00032s, 0.00116s, and 0.00319s for input sizes of 1000, 2000, and 5000, respectively. Meanwhile, the corresponding times for Median of Medians are consistently higher: 0.00148s, 0.00223s, and 0.00534s. This performance gap is consistent with the theoretical understanding that QuickSelect, despite its $O(n^2)$ worst-case complexity, typically performs in $O(n)$ expected time due to the high probability of balanced partitioning when using a random pivot (Cormen et al., 2022). In contrast, while the deterministic Median of Medians also operates in $O(n)$ worst-case time (Blum et al., 1973), its larger constant factors (due to grouping, sorting, and recursive median selection) result in slower performance in practice.

The results from sorted inputs show a similar pattern. QuickSelect performs at 0.00059s, 0.00071s, and 0.00357s for sizes 1000, 2000, and 5000, respectively. Median of Medians takes 0.00093s, 0.00278s, and 0.00484s. Interestingly, QuickSelect remains efficient despite the sorted input, which might seem adversarial in some pivot strategies. This is because its randomized pivot choice avoids the consistent selection of poor pivots, a vulnerability seen in naive QuickSort variants (Hoare, 1962). Median of Medians, although slightly more stable across distributions, does not gain any significant advantage from the sorted structure and still incurs overhead from median calculations.

Similarly, in reverse-sorted inputs, QuickSelect achieves times of 0.00073s, 0.00112s, and 0.00216s, outperforming MoM's times of 0.00086s, 0.00178s, and 0.00597s. This further supports the theoretical notion that QuickSelect's average-case performance is robust across distributions due to its randomized pivot selection, whereas the deterministic algorithm, although worst-case optimal, is not necessarily faster in practice for modest input sizes due to its more complex pivot selection process.

In conclusion, the empirical data reinforce theoretical analysis: the Median of Medians algorithm guarantees linear worst-case time, making it preferable for environments where performance consistency is critical. However, in most practical scenarios with moderate input sizes and typical data distributions, Randomized QuickSelect is faster due to its lower overhead and strong average-case behavior, despite its probabilistic nature and potential for degraded performance in rare worst-case instances.

Elementary Data Structures

The purpose of this part is to explore and implement several basic data structures, including Arrays and Matrices, Stacks and Queues, and Linked Lists. The code and demo for each class can be found in the 'elementary_data_structures' folder of the provided GitHub repository. The rest of the paper will focus on analyzing and comparing those data structures.

Performance Analysis

Dynamic arrays provide constant‑time random access and amortized constant‑time appends, but incur linear time for arbitrary insertions and deletions because elements must be shifted (Cormen et al., 2022). In a list of length n, indexing any element at position i executes in $O(1)$, while inserting or deleting at position i takes $O(n - i)$ on average, which is $\Theta(n)$ in the worst case (Cormen et al., 2022). Nested dynamic arrays used as matrices inherit these costs: accessing or updating the element in row r and column c of an $m \times n$ matrix remains $O(1)$, but inserting an entire row requires shifting $O(m)$ row pointers and inserting a column requires $O(m \times n)$ individual element shifts (Goodrich et al., 2014).

When employing these structures as stacks, array‑backed implementations achieve amortized $O(1)$ for both push and pop at the end of the list, with occasional $O(n)$ spikes during capacity resizing; peeking the top element is always $O(1)$ (Cormen et al., 2022). A singly

linked-list stack, by contrast, attains true worst-case $O(1)$ for push and pop at the head, since each operation merely redirects a few pointers; however, each node requires extra memory for its next-pointer, and the lack of contiguous storage leads to poor cache performance and larger constant factors in practice (Goodrich et al., 2014).

Queue implementations reveal a similar dichotomy. A simple array-based queue that enqueues at the tail in amortized $O(1)$ must dequeue at the head by shifting all remaining elements, yielding $O(n)$ per dequeue (Cormen et al., 2022). A linked-list queue that maintains both head and tail pointers provides true $O(1)$ enqueue and dequeue by adjusting only a couple of pointers on each operation. Yet this approach suffers from the same pointer-overhead and cache-miss disadvantages as the linked-list stack (Goodrich et al., 2014). Circular-buffer techniques can mitigate the array approach's dequeue penalty, restoring $O(1)$ time at both ends without per-node overhead, though at the cost of more complex index arithmetic and fixed maximum capacity (Cormen et al., 2022).

Choosing between arrays and linked lists depends on workload characteristics. In scenarios demanding frequent random access, such as binary search, direct indexing in arrays delivers superior performance thanks to contiguous memory that leverages hardware-level caching (Sedgewick & Wayne, 2011). In contrast, applications requiring frequent insertions or deletions at unknown positions, like maintaining an active playlist or undo-redo history, benefit from linked lists to avoid $\Theta(n)$ shifts (Goodrich et al., 2014). For double-ended operations, specialized structures such as the double-ended queue (deque) strike a balance, offering amortized $O(1)$ operations at both ends via circular buffers and avoiding per-node pointer costs (Sedgewick & Wayne, 2011).

Discussion

In many software systems, arrays and linked lists form the backbone of more complex abstractions. Arrays, either fixed-size or dynamically resized, are widely used for tasks that require efficient random access and predictable iteration patterns. For example, graphical applications often store pixel data or vertex buffers in contiguous arrays to exploit fast indexing and cache locality, ensuring smooth rendering performance (Sedgewick & Wayne, 2011). In contrast, linked lists shine in scenarios where the cost of shifting elements outweighs the overhead of pointer management: operating systems frequently maintain free-memory blocks or I/O buffer chains as linked lists, allowing blocks to be inserted or removed in constant time when processes allocate or release resources (Goodrich et al., 2014).

When choosing between arrays and linked lists, memory usage and access speed often guide the decision. Arrays allocate contiguous memory, which on one hand maximizes data locality and minimizes per-element overhead, but on the other hand risks wasted capacity or expensive resize operations when growth is unpredictable (Cormen et al., 2022). Linked lists avoid these resizing costs by allocating nodes on demand, yet each node must reserve space for a pointer, and noncontiguous storage patterns incur additional cache misses, degrading traversal throughput. Thus, in high-performance numerical computing or time-sensitive loops, such as real-time signal processing, arrays are typically preferred; whereas in memory-fragmented or highly dynamic workloads, like maintaining a playlist that supports frequent middle-insertions, a linked list may offer a cleaner and more efficient solution (Goodrich et al., 2014).

Stacks and queues, though conceptually simple, underpin many real-world algorithms and system architectures. Stacks implemented atop arrays or lists serve as the foundation for function-call management in program runtimes and expression evaluation in compilers; their

strict LIFO discipline maps naturally to last-in, first-out resource usage (Cormen et al., 2022). While an array-based stack delivers amortized constant-time push and pop operations, with occasional $O(n)$ resizing, an equivalent linked-list stack guarantees true $O(1)$ performance at the cost of extra pointer storage and reduced cache benefits (Sedgewick & Wayne, 2011). In practice, language runtimes often opt for array-based stacks because resizing events are infrequent and contiguous frames enable faster memory access.

Queues appear in contexts ranging from breadth-first graph traversal to task scheduling in operating systems. A naïve array-based queue suffers from $O(n)$ dequeues due to shifting, but this can be overcome via circular-buffer techniques, restoring $O(1)$ performance at both ends with fixed capacity (Cormen et al., 2022). Linked-list queues achieve this flexibility inherently by maintaining head and tail pointers, enabling constant-time enqueue and dequeue without preallocating space, albeit with the familiar pointer-induced memory and cache overhead (Goodrich et al., 2014). High-throughput networking applications often prefer circular buffers because they bound memory usage and leverage sequential memory access, whereas event-driven frameworks with unpredictable queue growth might favor linked-list implementations for their simplicity and unbounded flexibility.

References

Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., & Tarjan, R. E. (1973). *Time bounds for selection*. Journal of Computer and System Sciences, 7(4), 448–461. https://doi.org/10.1016/S0022-0000(73)80033-9

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). Random House Publishing Services. https://reader2.yuzu.com/books/9780262367509

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures and Algorithms in Java* (6th ed.). Wiley.

Hoare, C. A. R. (1962). Quicksort. *The Computer Journal, 5*(1), 10–16.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison‑Wesley.