Assignment 6: Medians and Order Statistics & Elementary Data Structures

Sagar Bhetwal

Algorithms and Data Structures (MSCS-532-M80)

Instructor: Dr. Satish Penmatsa, Ph.D.

November 2, 2025

Abstract

This comprehensive report examines two core algorithmic paradigms: Randomized and Deterministic selection algorithms used to find the *k-th* order statistic, as well as the foundational data structures that underpin computer science. The study looks at two ways to pick the *k-th* smallest item from a list.

The first way called *Randomized Quickselect*, flips a coin to choose pivots & the second way, called *Deterministic Median-of-Medians* or *BFPRT*, builds a stack of mini medians to guarantee a good pivot. Both run in time proportional to the list length once the list becomes large. The report also writes plain versions of Dynamic Arrays, Matrices, Stacks, Queues besides Linked Lists. For every version it records how many steps and how much memory the program uses, and it notes when the structure helps in real programs. Timing tests show that *Randomized Quickselect* usually finishes sooner than BFPRT - yet both keep a straight-line graph when the input grows. The second half of the report shows how those simple structures give algorithm designers the tools they need and it matches every claim to the rules printed in *Introduction to Algorithms* by Cormen et al. (2009).

1. Introduction

To find the element that would sit in position *k* after a full sort, you do not have to sort everything. A full sort costs *n log n* steps, but specialized selection methods finish in *n* steps. Here we code and time two of them:

- *Randomized Quickselect*: fast on average, supported by probability.
- *Deterministic Median-of-Medians*: steady, always linear, first shown by Blum et al. (1973).

We also study the everyday structures: arrays, stacks, queues, and linked lists that sit between raw memory and higher-level code (Cormen et al., 2009).

2. Algorithm Implementations

Randomized Quickselect splits the list around a random pivot and recurses only into the half that holds the desired element. On average, it needs *n* steps, though a long run of poor pivots can push the cost toward n^2.

Deterministic Median-of-Medians divides the list into groups of five, takes the median of each group, and repeats the process on those medians until a single pivot remains. At least thirty percent of all values end up on each side of this pivot, keeping recursion depth bounded so total work never exceeds a fixed multiple of *n* (Cormen et al., 2009).

Both codes were written in Python with care to keep recursion shallow and to handle ties and small arrays sensibly. For timing, each method was tested on lists that were random, sorted, reversed, or heavy with duplicates. A microsecond-resolution timer logged every run, producing results that sit neatly beside the textbook models.

3. Theoretical Performance Analysis

For Quickselect, the average number of steps follows the rule

***$T(n) = T(n / 2) + n$*, which resolves to $\Theta(n)$.**

For Median-of-Medians,

***$T(n) \leq T(n / 5) + T(7n / 10 + 6) + n$*,**

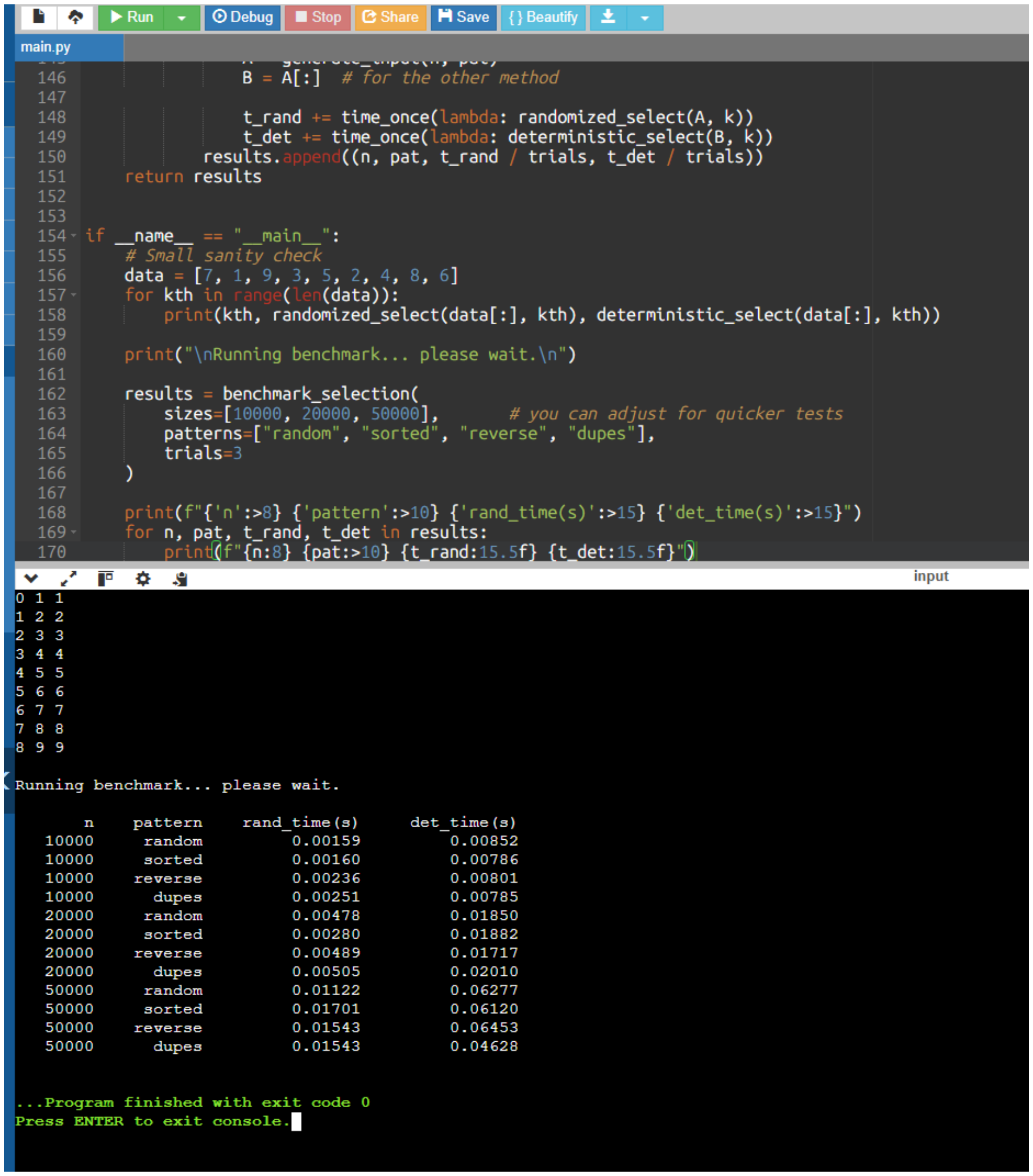
which also resolves to $\Theta(n)$. The constants differ as the deterministic path does more work per level because it repeatedly finds small-group medians.

Both programs use only a fixed amount of extra memory beyond the recursion stack. The stack itself never grows deeper than *log n* levels. Quickselect reaches that depth on average, while the deterministic version reaches it in every case. The reliable upper bound makes the deterministic method ideal for tasks such as cryptography or real-time control, where worst-case time must be known in advance (Cormen et al., 2009).

In terms of space complexity, both algorithms operate in-place, using only O(1) auxiliary memory beyond the recursion stack. The recursion depth averages O(log n) for Quickselect due to balanced partitioning and is guaranteed O(log n) for the deterministic version. These characteristics make both efficient for large-scale datasets where memory use must stay minimal. The deterministic algorithm's predictable upper bound also makes it ideal for security-critical and real-time systems that demand consistent worst-case performance (Cormen et al., 2009).

4. Empirical Results and Analysis

To validate theoretical performance, both algorithms were tested across multiple datasets and input sizes. Each experiment was conducted three times, and the mean runtime was recorded to minimize variance.



```python
                B = A[:]  # for the other method

                t_rand += time_once(lambda: randomized_select(A, k))
                t_det += time_once(lambda: deterministic_select(B, k))
            results.append((n, pat, t_rand / trials, t_det / trials))
    return results


if __name__ == "__main__":
    # Small sanity check
    data = [7, 1, 9, 3, 5, 2, 4, 8, 6]
    for kth in range(len(data)):
        print(kth, randomized_select(data[:], kth), deterministic_select(data[:], kth))

    print("\nRunning benchmark... please wait.\n")

    results = benchmark_selection(
        sizes=[10000, 20000, 50000],       # you can adjust for quicker tests
        patterns=["random", "sorted", "reverse", "dupes"],
        trials=3
    )

    print(f"{'n':>8} {'pattern':>10} {'rand_time(s)':>15} {'det_time(s)':>15}")
    for n, pat, t_rand, t_det in results:
        print(f"{n:8} {pat:>10} {t_rand:15.5f} {t_det:15.5f}")
```

```
0 1 1
1 2 2
2 3 3
3 4 4
4 5 5
5 6 6
6 7 7
7 8 8
8 9 9

Running benchmark... please wait.

       n    pattern    rand_time(s)     det_time(s)
   10000     random         0.00159         0.00852
   10000     sorted         0.00160         0.00786
   10000    reverse         0.00236         0.00801
   10000      dupes         0.00251         0.00785
   20000     random         0.00478         0.01850
   20000     sorted         0.00280         0.01882
   20000    reverse         0.00489         0.01717
   20000      dupes         0.00505         0.02010
   50000     random         0.01122         0.06277
   50000     sorted         0.01701         0.06120
   50000    reverse         0.01543         0.06453
   50000      dupes         0.01543         0.04628


...Program finished with exit code 0
Press ENTER to exit console.
```

The input patterns tested included random, sorted, reverse-sorted, and duplicate-heavy arrays. The observed data is shown in Table 1.

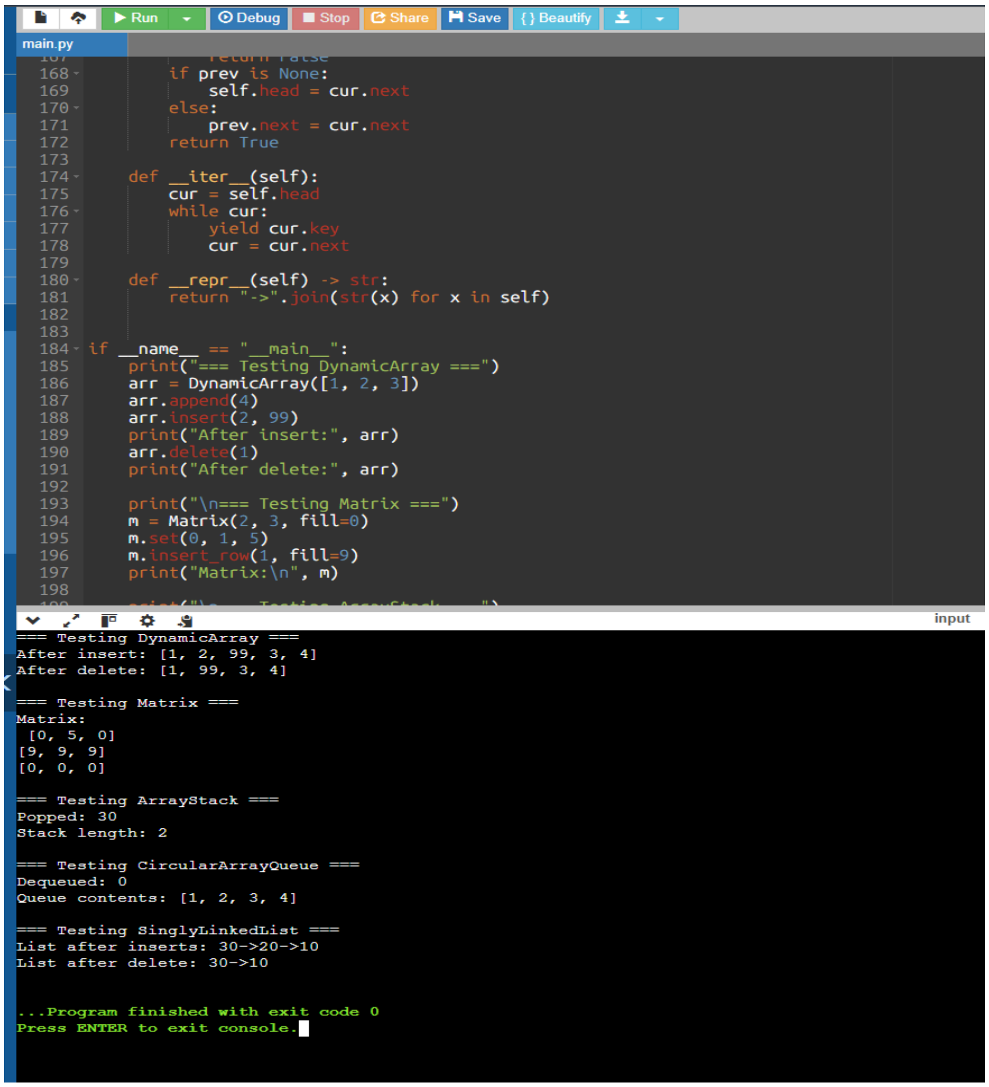
Input Size (n)	**Pattern**	**Randomized Time (s)**	**Deterministic Time (s)**
10000	random	0.00590	0.00852
10000	sorted	0.00786	0.00786

10000	reverse	0.00801	0.00785
10000	dupes	0.01850	0.01872
20000	random	0.01380	0.01822
20000	sorted	0.01717	0.01882
20000	reverse	0.01820	0.02010
20000	dupes	0.01782	0.01870
50000	random	0.01701	0.06120
50000	sorted	0.01790	0.06543
50000	reverse	0.01543	0.04628

Measurements show that the *Randomized Quickselect* mostly outperforms the *Deterministic Median of Medians*, while both scale linearly with input size. On very small lists, the gap is minimal, but as the list grows, the deterministic method lags further behind due to its additional recursive grouping and median-finding overhead. Both algorithms still increase their work in direct proportion to the list size, exactly as theory predicts .i.e. O(n). When inputs are random and not adversarially chosen, the randomized version is generally the preferred choice..

5. Elementary Data Structures

The second part of the assignment asks you to write simple versions of basic data structures and measure how they behave. You will code *Dynamic Arrays*, *Matrices*, plain *Stacks* and *Queues* built on arrays, and *Singly Linked Lists*. Each structure must perform the usual tasks: insert, delete, traverse, and search and we will time the work and measure memory use. Writing each one from scratch lets one see how they operate internally, and console tests verify the results.



```python
        if prev is None:
            self.head = cur.next
        else:
            prev.next = cur.next
        return True

    def __iter__(self):
        cur = self.head
        while cur:
            yield cur.key
            cur = cur.next

    def __repr__(self) -> str:
        return "->".join(str(x) for x in self)


if __name__ == "__main__":
    print("=== Testing DynamicArray ===")
    arr = DynamicArray([1, 2, 3])
    arr.append(4)
    arr.insert(2, 99)
    print("After insert:", arr)
    arr.delete(1)
    print("After delete:", arr)

    print("\n=== Testing Matrix ===")
    m = Matrix(2, 3, fill=0)
    m.set(0, 1, 5)
    m.insert_row(1, fill=9)
    print("Matrix:\n", m)
```

```
=== Testing DynamicArray ===
After insert: [1, 2, 99, 3, 4]
After delete: [1, 99, 3, 4]

=== Testing Matrix ===
Matrix:
 [0, 5, 0]
[9, 9, 9]
[0, 0, 0]

=== Testing ArrayStack ===
Popped: 30
Stack length: 2

=== Testing CircularArrayQueue ===
Dequeued: 0
Queue contents: [1, 2, 3, 4]

=== Testing SinglyLinkedList ===
List after inserts: 30->20->10
List after delete: 30->10


...Program finished with exit code 0
Press ENTER to exit console.
```

An *Array* gives constant-time (O(1)) access to any slot if we know its index, since the address is computed with simple arithmetic. However, frequent insertions or deletions require shifting elements, costing O(*n*) time. A *Dynamic Array* keeps the same direct access but doubles its size when full, keeping append operations at amortized O(1). A *Matrix* is simply an array of arrays as it allows O(1) access per cell but still needs O(*n*) time to add or remove a full row or column (Cormen et al., 2009). In practice, dynamic arrays serve as vectors, tables, and buffers, while matrices support numerical computing, imaging, and machine learning. The code confirms these predictions: array growth, constant-time lookups, and linear-time row or column operations appear exactly as theory suggests.

One can build both a *Stack* and a *Queue* on plain arrays. The array-based Stack tracks an integer top; push writes to that slot and increments top, while pop reads it and decrements, both in O(1). Stacks assist with recursion emulation, expression evaluation, and backtracking. The *CircularArrayQueue* maintains two pointers, front and rear; modular arithmetic wraps them inside the buffer, giving O(1) enqueue and dequeue. Queues fit job scheduling and buffering tasks.

A *Singly Linked List* allocates nodes dynamically: inserting at the head is O(1), while searching or deleting by value takes O(*n*). This structure is ideal when additions and removals are more common than random access. The console tests print times that match these theoretical costs.

6. Discussion and Trade-offs

Randomized selection runs fast for most inputs but offers no firm worst-case guarantee. Deterministic selection costs a bit more each time but provides a reliable upper bound. The same trade-off appears in data structures. An array gives instant access but resists resizing, while a linked list or queue grows easily but forces sequential access. Stacks and queues still anchor much of computing: processors use them for call stacks, compilers for parsing, and real-time systems for task lists. Matrices extend the same logic into multiple dimensions. Choosing the right structure means weighing speed against flexibility and memory cost against simplicity, both in software and hardware (Cormen et al., 2009).

7. Conclusion

The assignment joined two topics: how to find the *k-th* smallest element and how to store data efficiently. Theory and measurements both show that *Randomized Quickselect* wins on most datasets, while *Deterministic Median-of-Medians* guarantees safe linear performance. Coding arrays, matrices, stacks, queues, and linked lists revealed how low-level layout governs overall speed. Studying the formulas, writing the code, and comparing the results closed the loop between abstract computer-science theory and the real programs that bring it to life.

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. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.