**Part 1**

**Implementation**

https://github.com/nhemani33090/MSCS532\_Assignment6/blob/main/selection\_algorithms.py

**Performance Analysis**

1. **Time Complexity Analysis**

The efficiency of both algorithms depends on how well they reduce the problem size at each step.

**Median of Medians:** Guaranteed O(n) Time Complexity

* Median of Medians ensures at least 30% of elements are eliminated in each step. Since the problem shrinks quickly and consistently, the total time taken increases linearly with input size, leading to O(n) worst-case complexity.

**Quickselect**: O(n) on Average, O(n2) in the Worst Case

* Quickselect is efficient on average because a random pivot tends to divide the array fairly evenly. This results in a time complexity of O(n) in most cases. However, if Quickselect consistently picks bad pivots (such as always choosing the smallest or largest element), it reduces the array by only one element at a time, leading to a worst-case complexity of O(n2)

1. **Space Complexity Analysis**

**Median of Medians**: The Median of Medians algorithm has a space complexity of O(1) because it operates in-place, meaning it does not require additional storage beyond a few variables for pivot selection and partitioning. However, since it recursively reduces the problem size by at least 30% per step, the recursion depth is at most O(logn), making it memory-efficient.

**QuickSelect:** The Quickselect algorithm also has a space complexity of O(1) because it modifies the input array directly without creating additional copies. However, its recursion depth depends on the quality of the pivot selection. In the best and average cases, where the pivot divides the array fairly, the recursion depth is O(logn). In the worst case, where the pivot is always poorly chosen (e.g., smallest or largest element), the recursion depth can reach O(n), leading to higher memory usage.

Overall, Median of Medians is more predictable in memory usage, while Quickselect can be less efficient in rare cases where recursion depth is high.

**Empirical Analysis**

**Observed Results**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Analysis of Results**

1. **Quickselect is consistently faster than Median of Medians.**

* Across all datasets and input sizes, Quickselect outperforms Median of Medians.
* The difference is more noticeable at larger input sizes (e.g., for 10,000 elements, Quickselect takes ~0.002s, whereas Median of Medians takes ~0.006s).

1. **Median of Medians execution time increases steadily.**

* Since it performs more operations per iteration, the execution time is always slightly higher than Quickselect.
* However, its runtime remains consistent, reinforcing its theoretical guarantee of O(n) worst-case performance.

1. **Sorted and Reverse-Sorted Data Did Not Affect Quickselect Significantly.**

Quickselect does not exhibit worst-case behavior even on sorted data, contradicting theoretical concerns about poor pivot choices.

This suggests that the random pivot selection helped avoid the worst-case scenario, keeping execution time close to its average case O(n) performance.

1. **Scaling with Input Size**

* As input size increases, both algorithms show approximately linear growth, confirming the theoretical expectation of O(n) complexity.
* The growth rate for Quickselect is slightly lower, making it the faster practical choice.

**Relating Results to Theoretical Analysis**

The experimental results align well with theoretical expectations:

* Median of Medians follows its worst-case O(n) guarantee but is slower due to additional calculations for selecting a balanced pivot.
* Quickselect performs as expected in the average case O(n) and does not exhibit worst-case
* O(n2) behavior, reinforcing that poor pivot choices are rare in practice.
* The data confirms that Quickselect is the superior choice for most real-world applications, as it maintains efficiency even with structured inputs.

**Conclusion**

Based on empirical results, Quickselect is recommended over Median of Medians for practical use, as it provides faster execution times across all tested datasets. Median of Medians remains useful for scenarios requiring guaranteed worst-case performance, but in general, its additional overhead makes it less efficient than Quickselect in typical cases.

**Part 2**

**Implementation**

https://github.com/nhemani33090/MSCS532\_Assignment6/blob/main/data\_structures.py

**Performance Analysis**

Each data structure has different time complexities for insertion, deletion, and access operations. The following analysis explains these differences and compares their efficiency.

**Time Complexity Analysis**

**Arrays**

Arrays provide direct access to elements using index-based retrieval, making access operations efficient. However, insertions and deletions can be expensive if elements need to be shifted.

* Access: O(1) - Direct access via index.
* Insertion: O(1) at the end, O(n) if shifting is required.
* Deletion: O(n) due to element shifting.

**Stacks**

Stacks operate using a Last-In, First-Out (LIFO) principle. Pushing and popping elements occur at one end, making these operations highly efficient.

* Push: O(1) - Adds an element to the top.
* Pop: O(1) - Removes the top element.
* Peek: O(1) - Accesses the top element without removal.

**Queues**

Queues follow a First-In, First-Out (FIFO) principle. Enqueue operations add elements to the back, while dequeue operations remove from the front. Depending on the underlying implementation, shifting may be required.

* Enqueue: O(1) - Inserts at the rear.
* Dequeue: O(1) for linked lists, O(n) for arrays (due to shifting).
* Peek: O(1) - Accesses the front element.

**Linked Lists**

Linked lists consist of nodes that store data and pointers to the next node. They allow dynamic memory allocation but require sequential traversal for access.

* Access: O(n) - Must traverse nodes sequentially.
* Insertion: O(1) at the head, O(n) at the end.
* Deletion: O(n) - Requires traversal to find the element.

**Trees**

Trees are hierarchical data structures that allow efficient searching and insertion, especially when balanced.

* Access/Search: O(n) in an unbalanced tree, O(log n) in a balanced tree.
* Insertion: O(1) when adding as a child.
* Deletion: O(n) - May require restructuring.

**Space Complexity Analysis**

Space complexity determines how much additional memory a data structure uses relative to the size of the input. The following analysis compares the space efficiency of arrays, stacks, queues, linked lists, and trees.

**Arrays**

* Space Complexity: O(n) - Requires a fixed amount of contiguous memory equal to the size of the array.
* Memory Usage: Efficient for random access but inefficient when resizing since a larger array must be created, and elements copied.

**Stacks**

* Space Complexity: O(n) - Uses a list where elements are added dynamically.
* Memory Usage: Minimal overhead when implemented using an array. If implemented using a linked list, additional memory is required for pointers.

**Queues**

* Space Complexity: O(n) - Stores elements in a list or linked structure.
* Memory Usage: Using an array, shifting may require additional space for operations. Using a linked list, extra memory is needed for node pointers.

**Linked Lists**

* Space Complexity: O(n) - Requires additional memory for pointers along with stored values.
* Memory Usage: Unlike arrays, memory is allocated dynamically, which can be more efficient when the exact size of data is unknown. However, extra space is required for pointer storage.

**Trees**

* Space Complexity: O(n) - Memory usage grows with the number of nodes.
* Memory Usage: Each node requires storage for its value and child pointers. Balanced trees optimize space better than unstructured tree implementations.

**Trade-offs Between Arrays and Linked Lists for Stacks and Queues**

Stacks and queues can be implemented using both arrays and linked lists. Each approach has advantages and disadvantages based on memory usage, efficiency, and complexity.

* Arrays provide fast access (O(1)) but require shifting for insertions/deletions (O(n)).
* Linked lists allow dynamic allocation and quick insertions/deletions (O(1) at the head).
* Queue operations are more efficient using linked lists since shifting is avoided.

**Practical Applications of Data Structures**

**Arrays**

* Used in databases, caching, and fixed-size storage systems.
* Example: Image processing (2D pixel grids stored in arrays).

**Stacks**

* Used in function calls, undo-redo operations, and expression evaluation.
* Example: Function call stack in programming.

**Queues**

* Used in scheduling tasks, buffering, and breadth-first search (BFS).
* Example: CPU process scheduling.

**Linked Lists**

* Used in dynamic memory management, hash table chaining, and undo operations.
* Example: Managing memory blocks in operating systems.

**Trees**

* Used in hierarchical data storage such as file systems and databases.
* Example: XML/HTML Document Object Model (DOM).

**Conclusion**

Each data structure provides unique advantages based on memory efficiency, speed, and scalability. Arrays excel in fast access, while linked lists support dynamic memory. Stacks and queues are useful for structured data flow, while trees are ideal for hierarchical relationships.