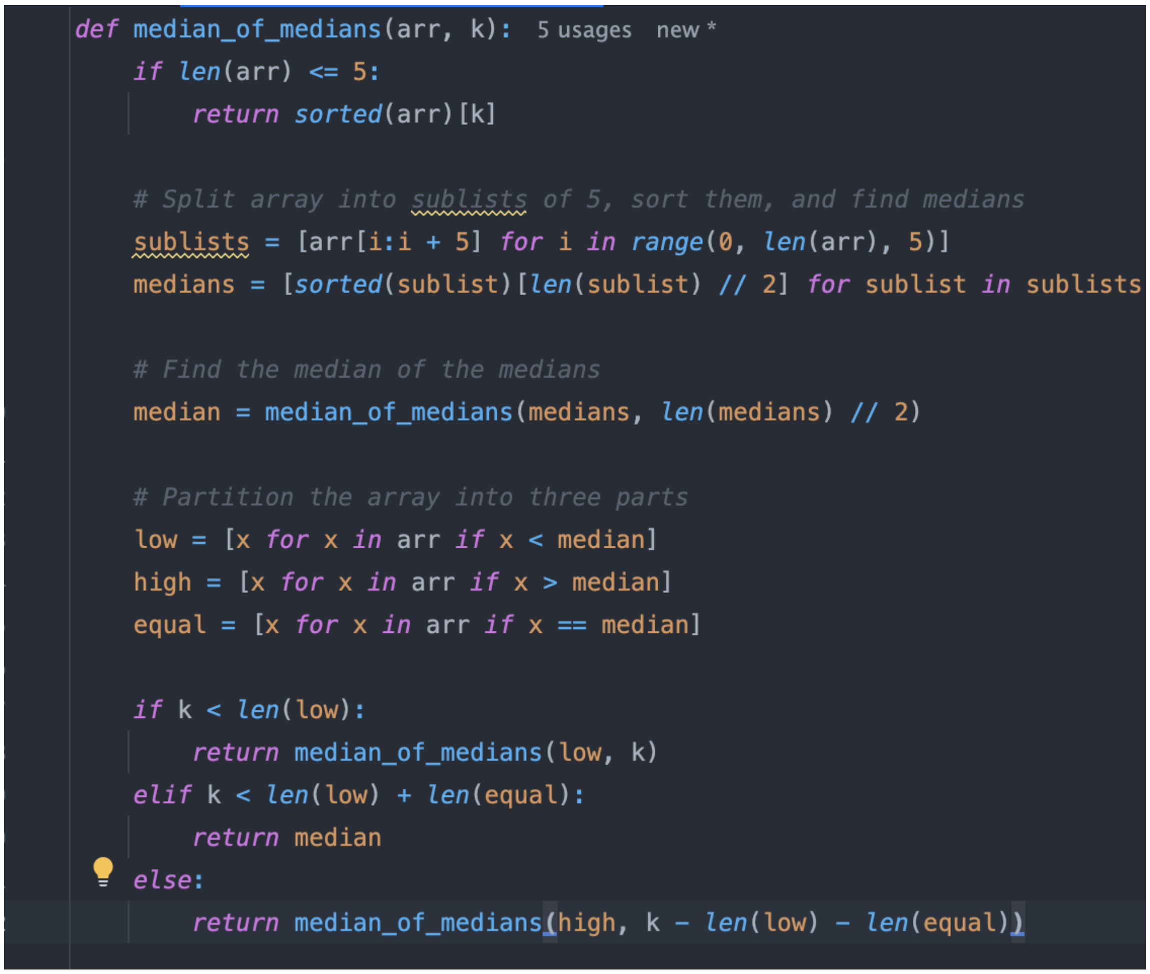
Part 1

Implementation:

a. Deterministic Selection Algorithm (Median of Medians)

Algorithm Steps:

1. Divide the input array into groups of 5 elements.
2. Sort each group and find the median of each group.
3. Recursively apply the same process to find the median of these medians.
4. Partition the array around this median and determine if the k-th smallest element lies on the left or right of the partition.
5. Recursively continue the process on the relevant partition.

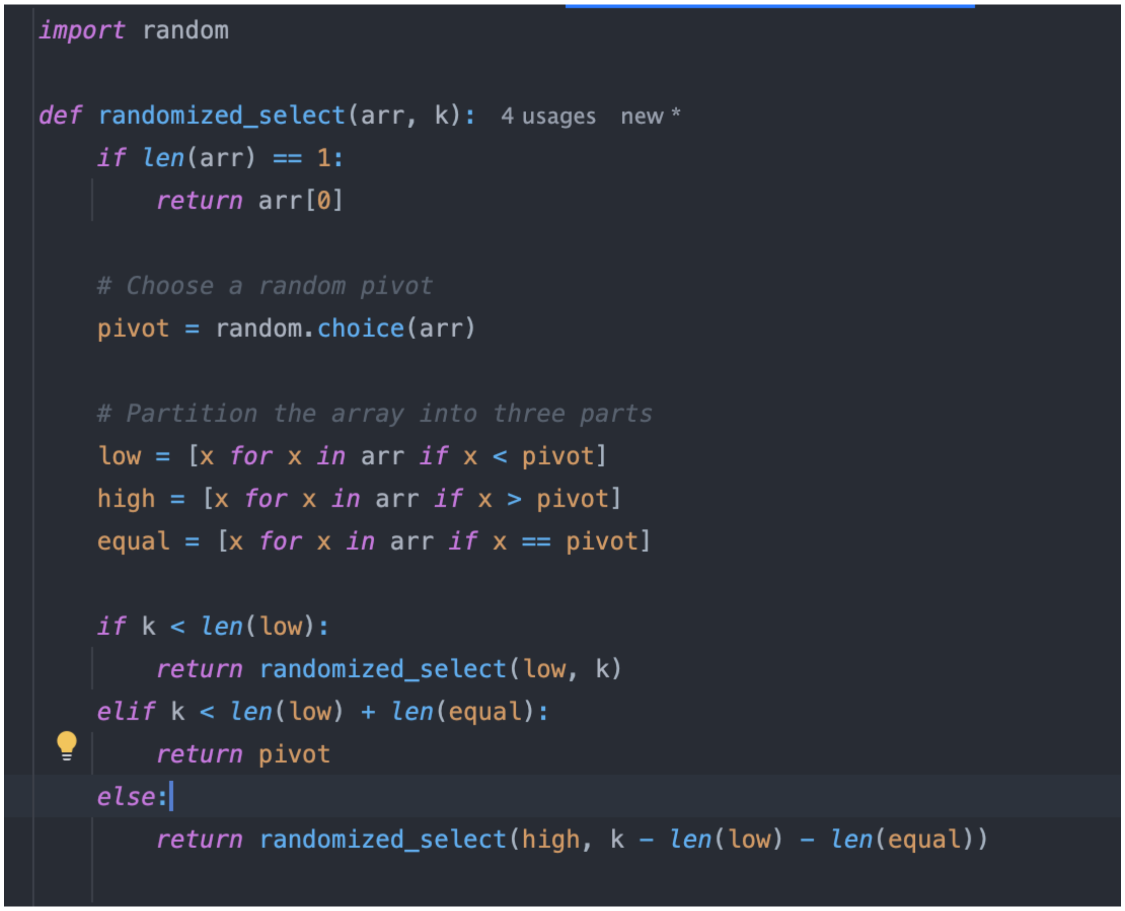


```
def median_of_medians(arr, k):  5 usages  new *
    if len(arr) <= 5:
        return sorted(arr)[k]

    # Split array into sublists of 5, sort them, and find medians
    sublists = [arr[i:i + 5] for i in range(0, len(arr), 5)]
    medians = [sorted(sublist)[len(sublist) // 2] for sublist in sublists

    # Find the median of the medians
    median = median_of_medians(medians, len(medians) // 2)

    # Partition the array into three parts
    low = [x for x in arr if x < median]
    high = [x for x in arr if x > median]
    equal = [x for x in arr if x == median]

    if k < len(low):
        return median_of_medians(low, k)
    elif k < len(low) + len(equal):
        return median
    else:
        return median_of_medians(high, k - len(low) - len(equal))
```

Randomized Selection Algorithm (Randomized Quickselect)

Algorithm Steps:

1. Choose a random pivot from the array.
2. Partition the array based on this pivot.
3. Recursively search in the partition where the k-th smallest element is expected to be.



```python
import random

def randomized_select(arr, k):  4 usages  new *
    if len(arr) == 1:
        return arr[0]

    # Choose a random pivot
    pivot = random.choice(arr)

    # Partition the array into three parts
    low = [x for x in arr if x < pivot]
    high = [x for x in arr if x > pivot]
    equal = [x for x in arr if x == pivot]

    if k < len(low):
        return randomized_select(low, k)
    elif k < len(low) + len(equal):
        return pivot
    else:
        return randomized_select(high, k - len(low) - len(equal))
```

Performance Analysis

Deterministic Algorithm (Median of Medians):

Time complexity

The deterministic selection algorithm achieves $O(n)$ time complexity through a strategic approach that ensures a balanced partitioning of the input array at each recursive step. The process involves the following steps:

1. Dividing the Array into Sub lists:
    - The input array is divided into $\lceil n/5 \rceil$ sublists, each containing at most 5 elements (Cormen et al., 2009).
    - Sorting each sub list of 5 elements takes $O(1)$ time per sublist since the size is constant. Therefore, sorting all sub lists collectively requires $O(n)$ time.
2. Finding Medians of Sub lists:
    - After sorting, the median of each sub list is identified. This step also takes $O(n)$ time since there are $\lceil n/5 \rceil$ medians to find.
3. Recursively Finding the Median of Medians:
    - This recursive call operates on a list of size $\lceil n/5 \rceil$ which leads to the recurrence relation: $\mathrm{T(n)} = \mathrm{T}(\frac{\mathrm{n}}{5}) + \mathrm{T}(\frac{7\mathrm{n}}{10}) + \mathrm{O(n)}$ Where $\frac{n}{5}$ is the median of medians, $T(\frac{7n}{10})$ is for finding the median of medians, where at least $\frac{3n}{10}$ elements are guaranteed to be either less than or greater than the pivot (Blum et al., 1973).
4. Partitioning the Array:
    - The array is partitioned into three parts: elements less than the pivot, equal to the pivot, and greater than the pivot. This partitioning step takes $O(n)$ time.
5. Recursive Selection:

- Depending on the position of k, the algorithm recursively selects the appropriate partition. Due to the balanced partitioning ensured by the median of medians, the problem size reduces by at least a constant fraction in each recursive step.

Using the Master Theorem for divide-and-conquer recurrences, the above recurrence relation resolves to $O(n)$, confirming that the deterministic selection algorithm operates in linear time in the worst case (Cormen et al., 2009).

Space complexity

The depth of recursion is $O(logn)$ because the problem size reduces by a constant fraction (approximately 70%) at each step (Cormen et al., 2009). For additional space or auxiliary space, it is required for storing sub lists and medians, which amounts to $O(n)$.

Overall, the space complexity is $O(n)$, primarily due to the auxiliary space used for sublists and medians, alongside a recursion depth of $O(logn)$.

Additional Overhead

The deterministic algorithm involves more sorting and median calculations at each step, leading to higher constant factors in practice.

Randomized Algorithm (Quickselect):

Time complexity

The algorithm is a probabilistic selection algorithm that, on average, operates in linear time, $O(n)$, but can degrade to quadratic time, $O(n^2)$, in the worst case (Cormen et al., 2009).

The randomized Quickselect algorithm achieves $O(n)$ expected time complexity by selecting a pivot uniformly at random, which probabilistically ensures balanced partitioning on average. The steps involved are:

1. Choosing a Random Pivot:
   - A pivot is selected uniformly at random from the array, which ensures that each element has an equal probability of being chosen.
2. Partitioning the Array:
   - The array is partitioned into three parts: elements less than the pivot, equal to the pivot, and greater than the pivot. This partitioning takes $O(n)$ time.
3. Recursive Selection:
   - Depending on the position of k, the algorithm recursively selects the k-th smallest element from the appropriate partition.

Therefore, we get the recurrence relation for Quickselect is:

$$T(n) = T(\frac{n}{2}) + O(n)$$

This relation solves to $O(n)$ expected time complexity, as the pivot is likely to split the array into reasonably balanced partitions on average (Cormen et al., 2009).

However, in the worst case—where the pivot consistently results in highly unbalanced partitions (e.g., always the smallest or largest element)—the time complexity degrades to $O(n^2)$. Such scenarios are rare due to the randomness in pivot selection, making the expected linear time complexity a reliable performance measure in practice (Cormen et al., 2009).

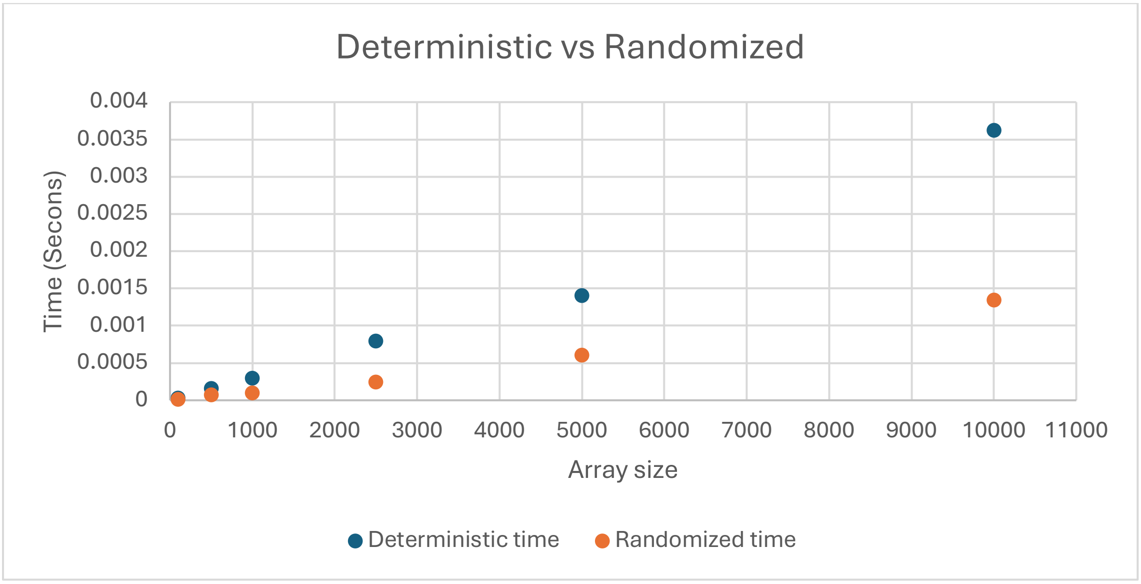
Space Complexity

On average, the recursion depth is $O(log\ n)$ due to the probabilistic nature of pivot selection leading to balanced partitions. In the worst case, the recursion depth can reach $O(n)$ if partitions are consistently unbalanced (Cormen et al., 2009). The algorithm typically performs in-place partitioning, requiring $O(1)$ additional space beyond the recursion stack. Thus, the space complexity is $O(\log n)$ on average and $O(n)$ in the worst case.

Additional Overheads

The randomized algorithm is simpler and faster in practice because it avoids the overhead of ensuring a good pivot but sacrifices worst-case guarantees.

Empirical analysis



Expected Results:

- For smaller input sizes, both algorithms should have comparable performance.
- For larger input sizes, the randomized algorithm should generally be faster due to lower constant overheads, except in cases where the random pivot consistently performs poorly.
- The deterministic algorithm is slower due to the overhead of finding the median of medians, but its performance is consistent, regardless of input distribution.

Part 2: Elementary Data Structures Implementation and Discussion

Arrays and Matrices

- Insertion/Deletion: Inserting or deleting an element at a random index in an array takes $O(n)$ time in the worst case because shifting elements is required to maintain the contiguous memory structure (Cormen et al., 2009). For example, if you insert or delete at the beginning of the array, all n elements must be shifted one position.
- Access: Arrays provide direct access to any element in $O(1)$ time because elements are stored in contiguous memory and can be indexed directly (Weiss, 2013)
- Applications: Arrays are used in situations where fast access to elements is needed and where the size of the data structure is fixed or known in advance (Weiss, 2013). They are used for storing matrices in numerical computations, lookup tables, and static datasets.

Stack

- **Push/Pop**: Both operations in a stack occur at the end of the structure, making them $O(1)$ in time complexity (Cormen et al., 2009). Since no element shifting is required, adding or removing the most recent element is efficient.
- **Access (Peek)**: Accessing the top element without removing it is also $O(1)$ because it only requires checking the last element in the stack (Weiss, 2013).
- **Applications**: Stacks are used extensively in function call management, such as the call stack in recursive programming and parsing expressions. They are also used in algorithms that require a backtracking mechanism, such as depth-first search (Cormen et al., 2009).

Queues

- **Enqueue/Dequeue**: In a queue, inserting an element and removing an element both occur at opposite ends of the structure, leading to $O(1)$ time complexity for each operation (Cormen et al., 2009). Using Python's deque implementation ensures that both operations are efficient (Knuth, 1998).
- **Access**: Since elements are processed in a first-in-first-out (FIFO) manner, access to the front or rear of the queue is also $O(1)$.
- **Applications**: Queues are widely used in scheduling systems, such as CPU job scheduling, printer queue management, and BFS for tree or graph traversal (Weiss, 2013). They are also crucial in communication protocols and messaging systems where order must be preserved (Cormen et al., 2009).

Linked Lists

- **Insertion/Deletion**: Inserting or deleting a node at the head of a singly linked list is $O(1)$, since only the head pointer needs to be updated (Knuth, 1998). However, searching for a specific element or inserting/deleting at an arbitrary position requires traversal, making these operations $O(n)$ in the worst case.
- Traversal: Traversing a linked list from head to tail takes $O(n)$ time, as there is no direct access to elements (Cormen et al., 2009).
- Applications: Linked lists are suitable for scenarios where dynamic memory allocation is needed, and frequent insertions and deletions occur, such as in dynamic databases, implementing undo functionality in software, and managing dynamic collections (Weiss, 2013).

When implementing stacks and queues, arrays and linked lists offer different trade-offs in terms of memory and performance. Arrays provide $O(1)$ access and are efficient in terms of memory, as elements are stored contiguously, improving cache performance (Cormen et al., 2009). However, arrays have fixed sizes and resizing can be costly, especially for queues where dequeueing from the front requires shifting elements, resulting in $O(n)$ time (Weiss, 2013).

In contrast, linked lists are dynamic, allowing for efficient $O(1)$ insertions and deletions without resizing or shifting (Knuth, 1998). This makes them ideal for queues, where both enqueue and dequeue operations remain efficient. However, linked lists require extra memory for storing pointers and have slower access times compared to arrays, as accessing elements involves traversing the list, which can take $O(n)$ time (Cormen et al., 2009).

In summary, arrays are preferable when the data structure size is known and fast access is critical, while linked lists are better suited for dynamic structures that require frequent modifications (Weiss, 2013).

Discussion on practical application of Arrays, Linked list, Stacks and Queues use

Arrays are ideal for fixed-size data structures requiring fast access, such as lookup tables or matrices. They provide $O(1)$ access and have excellent cache locality, making them efficient for tasks like image processing. However, arrays struggle with frequent insertions and deletions, especially in large datasets, as shifting elements is costly (Weiss, 2013).

Linked lists are better suited for dynamic data structures with frequent insertions and deletions, such as task scheduling or browser history management. Linked lists grow and shrink efficiently without resizing, but they incur extra memory overhead for storing pointers and are slower for random access compared to arrays (Cormen et al., 2009).

Stacks follow a Last-In-First-Out (LIFO) order and are useful in managing function calls and recursive algorithms. They also support undo operations in applications like text editors and calculators. Both array-based and linked-list stacks have $O(1)$ push and pop operations, though arrays provide better memory locality (Knuth, 1998).

Queues are First-In-First-Out (FIFO) data structures, commonly used for task scheduling, data streaming, and breadth-first search (BFS) algorithms. Linked-list queues allow efficient enqueue and dequeue operations without the need for shifting, whereas array-based queues may require circular buffers to avoid costly operations (Weiss, 2013).

Reference

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

Knuth, D. E. (2012). *The Art of Computer Programming Volume 3: Sorting and searching*. Addison-Wesley.

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

Motwani, R., & Raghavan, P. (1995). *Randomized Algorithms*. Cambridge University Press.

Weiss, M. A. (2013). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson.