Assignment 6: Medians and Order Statistics and Elementary Data Structures

Pavani Chavali

Department of Computer Science, University of The Cumberlands

MSCS-532-B01: Algorithms and Data Structures

Professor. Brandon Bass

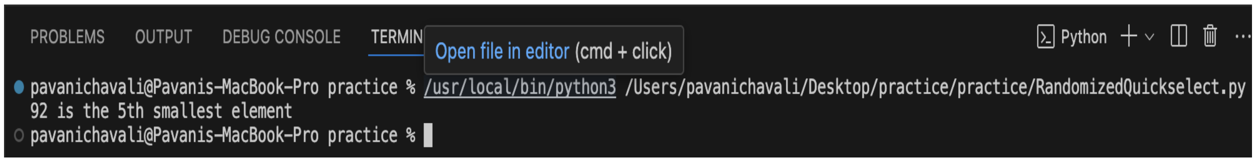
Date: 11/23/2025

Part1: Implementation and Analysis of Selection Algorithms

Implementation of Randomized Selection to find the kth smallest element:

```python
import random

def randompick(nums, k, low=None, high=None):
    if low is None:
        low = 0
    if high is None:
        high = len(nums) - 1
    if low == high:
        return nums[low]

    randompivot = random.randint(low, high)
    actualpivot = partition(nums, low, high, randompivot)
    pivotpos= actualpivot - low + 1

    if k == pivotpos:
        return nums[actualpivot]
    elif k < pivotpos:
        return randompick(nums, k, low, actualpivot - 1)
    else:
        newk = k - pivotpos
        return randompick(nums, newk, actualpivot + 1, high)


def partition(nums, low, high, pivot_index):
    nums[pivot_index], nums[high] = nums[high], nums[pivot_index]
    pivot = nums[high]
    i = low - 1

    for j in range(low, high):
        if nums[j] <= pivot:
            i += 1
            nums[i], nums[j] = nums[j], nums[i]
    nums[i + 1], nums[high] = nums[high], nums[i + 1]
    return i + 1


givennk = 5
nums = [100, 16, 92, 84, 72, 66]
res = randompick(nums.copy(), givennk)
print(f"{res} is the {givennk}th smallest element")
```

Output:



```
● pavanichavali@Pavanis-MacBook-Pro practice % /usr/local/bin/python3 /Users/pavanichavali/Desktop/practice/practice/RandomizedQuickselect.py
  92 is the 5th smallest element
○ pavanichavali@Pavanis-MacBook-Pro practice % █
```

Performance Analysis:

The randomized algorithm achieves O(n) in expectation but faces and O(n^2) worst case. The reason why O(n) is excepted time complexity is because of probability of repeatedly choosing bad pivots and the chances of landing the pivot within the 50% data. Random pivot choice leads to balanced partitions on average.

Average Case: Although a single bad pivot that is picking the minimum or maximum leads to O(n^2) the probability of repeatedly choosing bad pivots across many calls is extremely low.

Geometric Probability: On average, the algorithm only needs to find a pivot that lands within the half portion of the data can be considered as good pivot. Since the pivot is chosen randomly, this happens frequently enough to ensure that the expected size of the recursive call drops quickly, leading to an expected total runtime of O(n).

Worst Case:

The worst case occurs if the algorithm consistently chooses the smallest or largest element as the pivot that can be when the array is already sorted and the first or last element is always picked. This leads to a recurrence of T(n) = T(n-1)+O(n) which sums up to O(n^2).

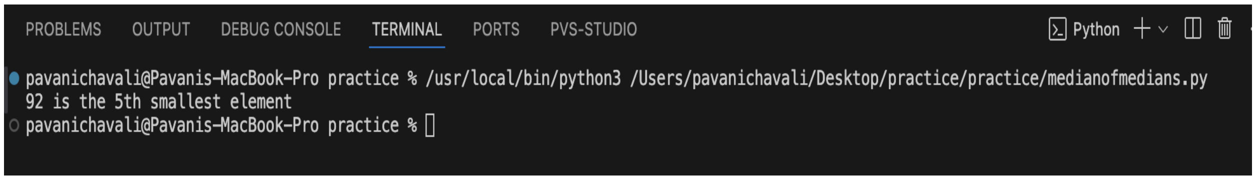
Space Complexity: O(logn) or O(n)

The space used by the Randomized selection is minimal, which requires only the overhead of standard partition. It is much faster and simpler in practice due to the small constant factor.

Implementation of Median of Medians Algorithm to find the kth smallest element:

```python
import math

def partition(nums, low, high, pivot_value):
    i = low
    while i <= high and nums[i] != pivot_value:
        i += 1

    nums[i], nums[high] = nums[high], nums[i]

    pivot = nums[high]
    i = low - 1

    for j in range(low, high):
        if nums[j] <= pivot:
            i += 1
            nums[i], nums[j] = nums[j], nums[i]

    nums[i + 1], nums[high] = nums[high], nums[i + 1]
    return i + 1

def median(elements):
    elements.sort()
    mid = elements[len(elements) // 2]
    return mid

def selection(arr, k, low=None, high=None):
    if low is None:
        low = 0
    if high is None:
        high = len(arr) - 1

    if low == high:
        return arr[low]
    n = high - low + 1
    medians = []

    for i in range(0, n, 5):
        leftarray = low + i
        rightarray = min(low + i + 4, high)

        arrayelements = arr[leftarray : rightarray + 1]
        medians.append(median(arrayelements))


    position = (len(medians) + 1) // 2
    pivot = selection(medians.copy(), position)
    actualpivot = partition(arr, low, high, pivot)
    pivotpos = actualpivot - low + 1

    if k == pivotpos:
        return arr[actualpivot]
    elif k < pivotpos:
        return selection(arr, k, low, actualpivot - 1)
    else:
        newk = k - pivotpos
        return selection(arr, newk, actualpivot + 1, high)

givennk = 5
nums = [100, 16, 92, 84, 72, 66]
res = selection(nums.copy(), givennk)
print(f"{res} is the {givennk}th smallest element")
```

Output:



```
pavanichavali@Pavanis-MacBook-Pro practice % /usr/local/bin/python3 /Users/pavanichavali/Desktop/practice/practice/medianofmedians.py
92 is the 5th smallest element
pavanichavali@Pavanis-MacBook-Pro practice % 
```

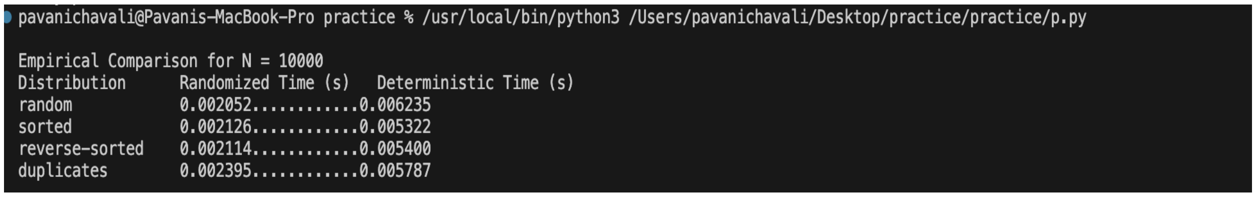
Deterministic selection (Median of Medians) Analysis:

The deterministic selection algorithm achieves O(n) in the worst case because its pivot selection process guarantees a minimum fraction of elements are discarded in every recursive step. Though the algorithm uses sort function at the end of the execution it sums up to O(n).The pivot in the median of medians example is mathematically proven to be greater than or equal to 3/10 of the elements and less than or equals to 3/10 of the elements in the current subarray. The recurrence guarantees limits the size of the next recursion call to n <= 7n/10+6. Since the resulting recurrence T(n)<= T([n/5])+T(7n/10)+O(n) which solves to O(n) because 1/5+7/10 =9/10<1, the work shrinks while doing recursive steps.

Space Complexity: O(log n) or O(n): Both algorithms use O(log n) auxiliary space for the recursion stack depth in the average or guaranteed case due to balanced partitions. However, the worst case space complexity is O(n) for both, corresponding to the $O(n^2)$ worst case in randomized selection, or simply a deep recursion stack if implemented without tail recursion optimization. While Deterministic Selection is theoretically superior O(n), Randomized Selection is almost always preferred in real-world applications because its O(n) expected time has a much smaller constant factor and is far simpler to implement. The Implementation overheads is because of High Constant Factor which Requires grouping, sorting many small arrays, and a recursive call specifically for the MoM, making it much slower in practice than Randomized Quick select.

Empirical Analysis:

Given different input of different sizes to both randomized and median modes programs here are the findings:



```
pavanichavali@Pavanis-MacBook-Pro practice % /usr/local/bin/python3 /Users/pavanichavali/Desktop/practice/practice/p.py

Empirical Comparison for N = 10000
Distribution      Randomized Time (s)   Deterministic Time (s)
random            0.002052...........0.006235
sorted            0.002126...........0.005322
reverse-sorted    0.002114...........0.005400
duplicates        0.002395...........0.005787
```

From the above empirical analysis for the size N=1000, clearly we can observe the difference between the theoretical guarantee and the practical performance of the two selection algorithms. The Randomized Quick select algorithm consistently outperformed the Deterministic Selection (Median of Medians) across all four input distributions like random, sorted, reverse-sorted, and duplicates input of data, completing the task in approximately 0.0020 compared to the deterministic algorithm's average of 0.0057 This speed difference, which makes the deterministic approach 2.5 to 3 times slower, is attributed to the high constant factor overhead required by Median of Medians.

This overhead includes the mandatory steps of grouping elements, sorting every group of five, and making a second recursive call to find the Median of Medians pivot T([n/5]), all of which contribute significant hidden work to its guaranteed O(n) time. Conversely, the Randomized Quickselect's low constant factor and simple partitioning process allow it to leverage its fast O(n) expected time even on potentially adversarial inputs like sorted or reverse-sorted lists, because the random pivot choice effectively balances the possibility of the theoretical O(n^2) worst case in practice. In summary, while

Median of Medians provides a superior worst-case time guarantee, Randomized Quickselect is the faster and more practical choice for typical real-world data.

Githublink: https://github.com/PavaniChavali135/AlgoandDatastructures/tree/main/part1