Project 2

Author Name Eyouel Kibret

1 Problem Statement

We are required to analyze the following program/code sample.

```
ALGORITHM FindMedianOfMedians(nums_list, k):
    N = length(nums_list)

    IF N ≤ 5 THEN
        SORT nums_list using InsertionSort
        RETURN nums_list[k - 1]
    END IF

    // Divide array into groups of 5
    partitions = []
    FOR i FROM 0 TO N-1 INCREMENTING 5 FROM I EACH TIME
        partition = elements of nums_list from index i to i+4
        ADD partition TO partitions
    END FOR

    // Find the median of each group
    medians = []
    FOR each partition IN partitions DO
        SORT partition using InsertionSort
        median = element at position floor(length(group) / 2)
        ADD median TO medians
    END FOR

    // Recursively find the median of all medians
    medianOfMedians = FindMedianOfMedians(medians, (length(medians) + 1) / 2)

    // Partition nums_list around the medianOfMedians
    left = all elements in nums_list less than medianOfMedians
    equal = all elements in nums_list equal to medianOfMedians
    right = all elements in nums_list greater than medianOfMedians

    // Determine which partition contains the k-th smallest element
    IF k ≤ length(left) THEN
        RETURN FindMedianOfMedians(left, k)
    ELSE IF k ≤ length(left) + length(equal) THEN
        RETURN medianOfMedians
    ELSE
        RETURN FindMedianOfMedians(right, k - length(left) - length(equal))
    END IF
END FUNCTION
```

2 Theoretical Analysis

Explain your theoretical estimate in 3-4 sentences.

The deterministic version of the Quick Select (median of medians method) uses divide and conquer strategy to select the median from an unsorted list of numbers. Until the first calling of "findMedianOfMedians" it takes $O(n)$ because sorting takes $O(1)$ because it is sorting a maximum of 5 elements. Finding the median of medians "findMedianOfMedians" method calling, takes $T(n/5)$. Partitioning also takes O(n) because we are working with at most n elements in the list. After pivot selection, it is guaranteed that the pivot is between 30% of the elements in both directions. Therefore, we are working with 70% of the partition at a time which makes it T$(7n/10)$. In conclusion it takes:
$T(n/5)$ to find median of medians, $T(7n/10)$ to recursively search the larger portion if median is not found, and O(n) time for sorting, partitioning and grouping.

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + cn$$

After solving using the method of substitution the time complexity becomes $O(n)$. (Arora, 2022, Section 4.6.2)

3 Experimental Analysis

3.1 Program Listing

(Feel free to include only selected portions if you like. For example, I would like to know which values of "n" you ran the program for.)

```
def findMedianOfMedians(nums_list, k):
    # actual function to find the k-th smallest element using the median of medians algorithm

    N = len(nums_list)
    if N <= 5:
        # if the length of the numbers is less than or equal to 5,
        # just sort it using the insertionSort helper function and return the median

        nums_list = insertionSort(nums_list)
        return nums_list[k-1]

    # Divide the array into groups of 5. There are n/5 groups.
    partitions = [nums_list[i:i + 5] for i in range(0, N, 5)]


    # Sort the small groups using insertion sort. Since each group is
    # of 5 elements, it takes a constant amount of time to sort those
    # groups.

    # Collect all the n/5 medians from the n/5 groups.

    medians = [insertionSort(partition)[len(partition) // 2] for partition in partitions]

    # recursively find the median of the medians
    medianOfMedians = findMedianOfMedians(medians, (len(medians) + 1) // 2)
```

```
    #  Partition the array on the median of medians.
    left, equal, right = partitionArray(nums_list, medianOfMedians)

    # recursively call the function based on the value of k
    # if k is less than or equal to the length of the less than list
    # it implies that the k-th smallest element is in the less than list
    if k <= len(left):
        return findMedianOfMedians(left, k)

    # if k is greater than the length of the less than list
    # and less than or equal to the length of the less than list + length of the equal to list
    # it implies that the k-th smallest element is the median of medians
    elif k <= len(left) + len(equal):
        return medianOfMedians

    # else it implies that the k-th smallest element is in the greater than list
    # so recursively call the function on the greater than list
    else:
        return findMedianOfMedians(right, k - len(left) - len(equal))


if __name__ == "__main__":

    power = 1
    n = 10 ** power
    k = n // 2  # Find the median value

    nums_list = [i for i in range(n)]

    start_time = time.time()
    result = findMedianOfMedians(nums_list, k)
    end_time = time.time()

    execution_time = end_time - start_time
    print(f"Execution time: ", execution_time)
    print(f"The {k}th smallest number is :", result)
```

Github repo: https://github.com/Jahsil/GWU-Algorithms-Fall-2025-Project2.git
I ran the program for values of n by changing the value of power from 1,2,3,4,5,6,7,8. "n" then becomes 10,100,1000,10^4, 10^5, $10^6 10^7 10^8$.

3.2 Data Normalization Notes

Do you normalize the values by some constant? How did you derive that constant?

Yes, I scaled the values by a constant. Since the experimental values are so low usually in the microsecond range and theoretical values much higher which doesn't have a specific unit of measurement like seconds. I used the average ratio to normalize the values of theoretical values. I calculated the scaling factor by taking the sum of the theoretical values and dividing by the sum of the experimental values.

$$\text{Scaling factor(c)} = \sum Theoretical\ Values(i) / \sum Experimental\ Values(i)$$

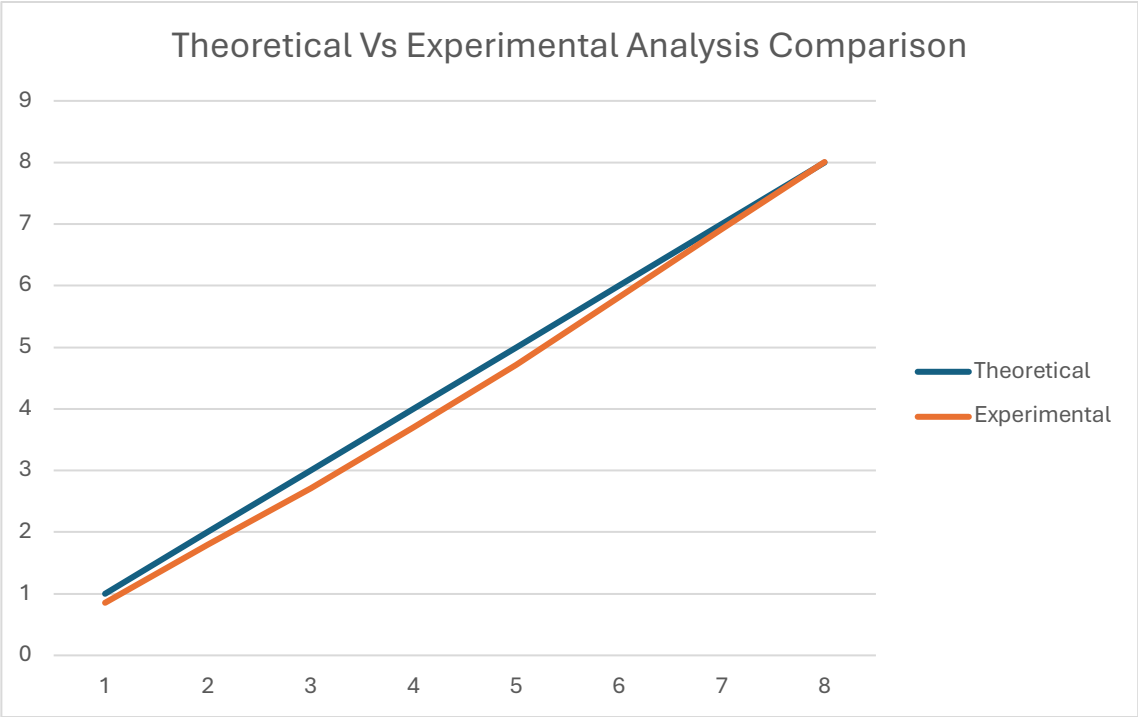
I chose this approach because unlike one-point scaling factor, the scaling factor I chose balances the whole dataset. So, c = 111111110 / 2.04E+02 = 544662.3039. Therefore, multiplying each theoretical value by this constant will result in values that are near each other so comparison will be easy. To show the values clearly, I took a log scale to observe the difference between consecutive values of n. If log scale is not used the graph won't represent a linear relationship so log scale is indeed required.

3.3 Output Numerical Data

n	Experimental	Theoretical
10	7.14E+00	10
100	6.26E+01	100
1000	5.16E+02	1000
10000	5.03E+03	10000
100000	5.21E+04	100000
1000000	6.58E+05	1000000
1.00E+07	8.33E+06	1.00E+07
1.00E+08	1.02E+08	1.00E+08

n(log)	Experimental(log)	Theoretical(log)
1	0.85383	1
2	1.796514	2
3	2.712258	3
4	3.701302	4
5	4.716789	5
6	5.818359	6
7	6.920659	7
8	8.008036	8

3.4 Graph



3.5 Graph Observations

From the above graph I can specify that:

- The theoretical values align well with the experimental values. There is no tremendous change(error) between them.
- Without the scaling factor the experimental data points would have been a flat line and the theoretical values would have been an increasing one. Therefore, a scaling constant of 544662.3039 modified the magnitude of the experimental values so that they can be compared.
- The experimental values are slightly below the theoretical ones for some values of n. This could be to the python interpreter, memory allocation or other reasons.
- When the value of n increases, the experimental values grow in sync with the values of the theoretical showing minimal difference. It implies that it follows the expected linear trend $O(n)$.
- For much higher value of n both lines grow in strong concordance, this implies that the theoretical implementatin $O(n)$ closely aligns with and supports the experimental results. In addition, the growth rate for both graphs follows in a similar way.

4 Conclusions

After careful analysis the theoretical analysis of code became $O(n)$. But to test it, a basic python program was built using built in methods and functions. After writing the python program, when working with smaller values of "n" there was no memory allocation issue. However, when working with higher values of "n" especially from n starting at 10^9, memory allocation was a huge problem so the program couldn't be tested for this value and above. However, for values under 10^9 the program worked well. The python program returned the median and the time taken for each n (by changing the value of power in the code), in seconds. The usage of a scaling factor was necessary because the values of the theoretical values with different values of "n" where large when compared to the experimental results. A scaling factor of 544662.3039 was taken for easy comparison. A log scale needed to be used to see the difference of values between different "n" intervals to compare them. It can be concluded that this algorithm always ensures a good partition and running at linear time makes it preferable rather than sorting a list and finding the median which takes $O(nlogn)$.

Reference

Arora, A. (2022). *Analysis and Design of Algorithms* (3rd ed.). Cognella, Inc.