Project 3

1. Problem Analysis

Given Problem: Pseudo-polynomial Partition.

Given a set consisting of n integers [a1, a2, … an], you want to partition into two parts so that the sum of the two parts is equal. Suppose s = a1 + a2 … + an. The time complexity of your algorithm should be O(ns) or better.

Analysis: We wish to identify if such a partition exists and if so, return true, else return false.

For such 2 partitions to exist, the total array sum should always be an even number. i.e., our first constraint is if the sum of all elements in the array is odd, return false since we can never find 2 equal sum partitions as their total will never equal an odd number*(Reasoning: Even = Odd + Odd OR Even = Even + Even, however Even != Odd + Even)*. Now, if the sum is even, then there might exist 2 partitions each having a total of = sum/2. To form these 2 partitions, we traverse through the given array & for each element, we choose from the following 2 options => 1. Put the element in the first partition OR 2. Put the element in the second partition. Notice, that if there exists a partition for which the total is (sum/2), then the rest of the elements must also equal sum/2. This simplifies our problem =>

If the sum of the given array is odd, return false.

Else, Find a partition in the given array such that,

its total = sum/2 … *(where the sum is the sum of the given array)*

If such a partition is found, return true; Else return false.

The logic for inserting an element into a particular subset: Check if it is possible to include element given_array[n-1] in a particular subset when the sum is greater than the element. Consider for subset2, we check if the element at partition(I, j) can be a part of the subset and that removing this element from the sum results in a valid solution in the previous row. If these conditions are met => include the element in subset2.

NOTE: We assume that all integers in the given array are positive.

2. Pseudocode

```
# fill the partition table in bottom up manner
for row in range(1, total // 2 + 1):
    for col in range(1, n + 1):
        table[row][col] = table[row][col - 1]
        # check if you can include current element or exclude it
        if row >= arr[col - 1]:
            table[row][col] = (table[row][col] or
                               table[row - arr[col - 1]][col - 1])
```

3. Time Complexity

Here, a 2-D matrix/table is created where the number of rows is equal to the number of elements of the given array plus one i.e. (n + 1), and the columns represent possible sums. The time complexity for filling this matrix is O((n+1) * (sum/2)), since our simplified approach checks if a partition has a sum of sum/2. In big O notation, we typically drop constants, so the time complexity can be approximated as O(n * sum). **Hence, the time complexity of this code is O(n * s),** where 'n' is the number of elements in the array and 's' represents the sum of the elements in the input array.

3.2 Data Normalization Notes

The Scaling Constant used here is 350.88. The formula to calculate scaling constant:

Scaling Constant = Average of Experimental Results (in ns) / Average of Theoretical Results.

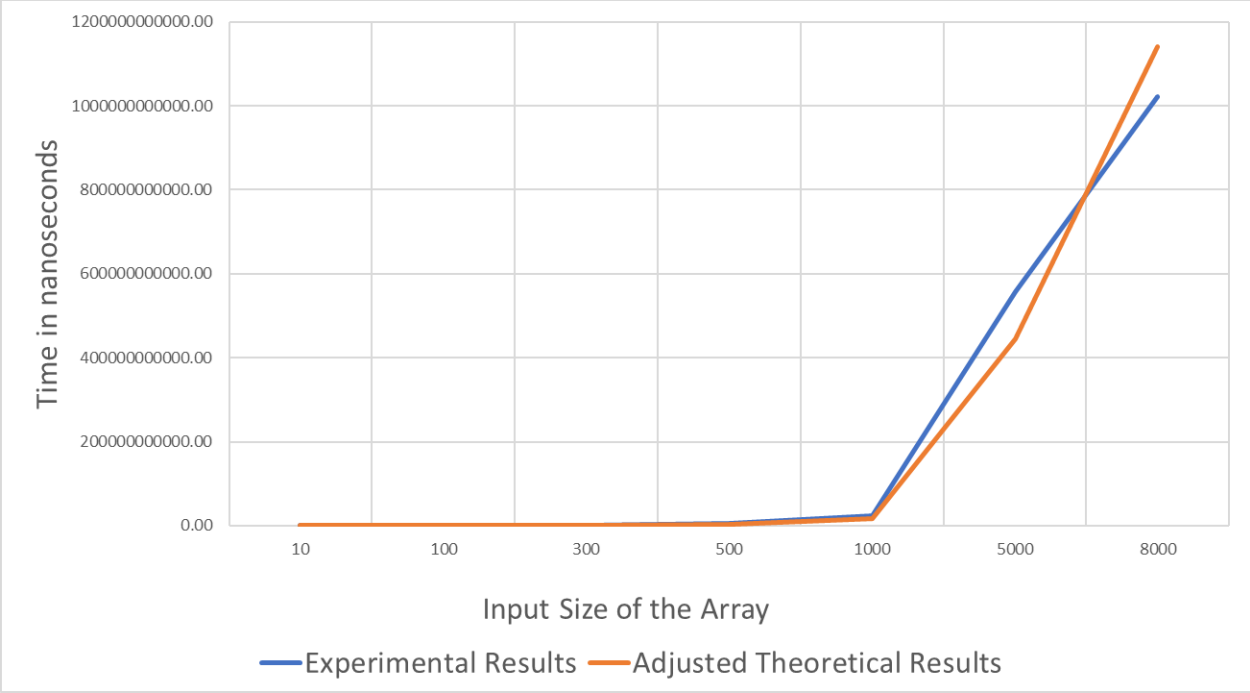
The formula used to calculate the Adjusted Theoretical Result:

*Adjusted Theoretical Result = Scaling Constant * Theoretical Result*

4. Numerical Results

Input Size	Experimental Results (in ns)	Sum	Theoretical Results (Sum * N)	Adjusted Theoretical Results
10	0.00	486	4860	1705287.51
100	249978303.91	5248	524800	184142979.98
300	2006717920.30	15498	4649400	1631391713.24
500	5594440937.04	25040	12520000	4393045177.81
1000	23626113891.60	51620	51620000	18112539303.41
5000	556897758007.00	254242	1271210000	446044964895.24
8000	1022545204877.85	406316	3250528000	1140552424580.51
Average =>	201365026742.21		573882132.5	

5. Graph



5.1 Graph Observations

For smaller values of n, the theoretical time taken is smaller than the time portrayed by experimental results. However, for large values of n, theoretical values are greater than the experimental values as expected. Hence, the time taken for the algorithm will be no longer than the theoretical values which is the upper bound.

6. Conclusions

The time complexity for the following algorithm is **O(n * s),** where 'n' is the number of elements in the array and 's' represents the sum of the elements in the input array**.** Overall, it is observed that the theoretical results are similar to the experimental results.

7. GitHub link

https://github.com/insp7/CSCI_6212/blob/master/project3PseudoPolynomialPartition/code%20-%20pseudo_partition.py