Lauren Hahn (G32735160), Isha Paliwal (G49146952)
Benjamin Marasco (G27637849), and Shaik Sameer (G49843839)

Project #3

1 Problem Statement

The problem our team was assigned was number 7, Pseudo-polynomial partition. Given a set of *n* integers, we were to partition it into two parts so that the sum of the two parts is equal. Our main idea of the solution was to figure out the best way to create the dynamic programming table that runs in a constant amount of time.

2 Theoretical Analysis

Algorithm and Pseudocode:

Our algorithm and pseudocode can be found through this link:
https://github.com/Lhahn01/Project-3-Team7/tree/main

Notation: *DPTable(m, j)* is a Boolean table that figures out if there exists a subset with a sum of *m* that can be constructed using some of the numbers selected from $[a_1, a_2, ..., a_j]$. In our code, this is represented by the *DPTable* being a 2D array. The row values represent *m*, while the columns represent *j*.

Base Case:

- For each *j*, *DPTable(0, j)* is *True* because we can always have a sum of 0 by taking none of the elements within the given set. In our code, this is done through *dpTable[0][y]* being *True*.
- *DPTable(m, 0)* is *False* for all *m* greater than 0 because there are no numbers within the set ($j = 0$) to create a sum greater than 0. In our code, this is achieved through *dpTable[y][0]* being *False*.

Recurrence Relation:

Given a subset of $[a_1, a_2, ..., a_{j+1}]$, we can figure out if we can create a sum *m'* from the numbers selected from that subset. That is by deciding if we should include the a_{j+1} number or not. If we do include a_{j+1}, then that means we have a sum of $m' - a_{j+1}$ that was achieved with the first *j* numbers, which is represented by $DPTable(m' - a_{j+1}, j)$. If we don't include a_{j+1}, then the sum would *m'* using the first *j* numbers, which is represented by *DPTable(m', j)*. Then, *DPTable(m', j + 1)* would equal the max between $DPTable(m' - a_{j+1}, j)$ and *DPTable(m', j)*.

To fill up all of the entries within the *DPTable*, we just follow what was identified in the base case and $DPTable(m', j + 1) = max\{(DPTable(m' - a_{j+1}, j), DPTable(m', j)\}$. Since there are *n* numbers within the given set and potential sums ranging from 0 to *s*, the total number of entries in the table would be *ns*. As each entry would take constant time to compute, the overall time complexity would be ***O(ns)***. In our code, instead of ranging from *0 to s*, it's actually *0 to s/2* because we're only dealing with a subset of numbers that add up to *s/2*. However, when looking at the overall time complexity, it's still ***O(ns)*** as *O(n(s/2))* is the same as *O(ns(½))*, and the constant number doesn't matter. As a result, it's ***O(ns)***. One thing to note is that this algorithm can prove the principle of optimality as a portion of the subset in the two parts partitioned should also have the same sum based on the DPTable we have constructed.

3 Experimental Analysis

3.1 Program Listing

We have implemented the code in Python and Java, which can be found in the GitHub repository. However, the following results are based on the Python code. The size of the set (*n*) is listed below in the

Output Numerical Data section (18, 20, 22, 24, 27). For the actual content of the array, we had random numbers. But, we made sure that the sum (*s)* of the entire array would be 500 to make it consistent for the experiment. To see what the result would be if *s* was not constant and everything was random for a given size, a screenshot is in the GitHub repository as well.

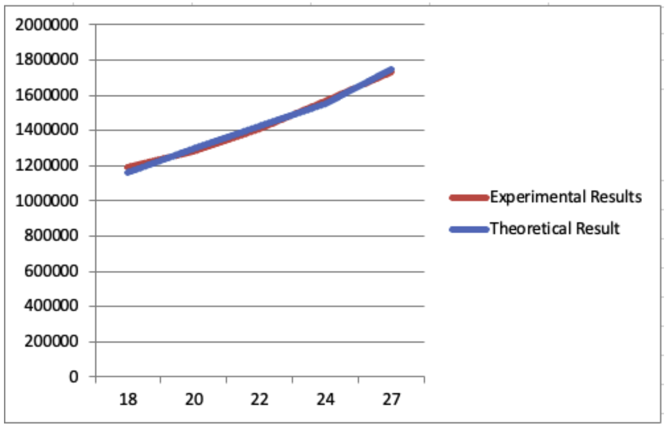
3.2 Data Normalization Notes

The values were normalized by a scaling constant. Using the Excel sheet provided to us, we were able to just use the original scaling constant. That is, we first get the average of each experimental and theoretical value. Then, the average of the experimental results gets divided by the theoretical average result. By doing so, the scaling constant we got was *129.387387*. Our Excel sheet can be found in the GitHub repository.

3.3 Output Numerical Data

n	**Experimental (in *nanoseconds*)**	**Theoretical: *O(ns)***	**Scaling Constant**	**Adjusted Theoretical Results**
18	1193000	9000	129.387387	1164486.49
20	1280000	10000	129.387387	1293873.87
22	1408000	11000	129.387387	1423261.26
24	1565000	12000	129.387387	1552648.65
27	1735000	13500	129.387387	1746729.73

3.4 Graph



3.5 Graphical Observations

The Theoretical Results represent *O(ns)*, where *s* is the sum of all the numbers within a given set. All of the experiments had a consistent sum of 500. And, it was just the number of elements (*n*) that varied like 18 or 20. To get the results, all we had to do was multiply each n-value by 500 and see what the y-values would be. The Experimental Results represent the results we received when we took the time in nanoseconds (the elapsed time). When looking at the plotted graph, we can see that the results of both are pretty much the same as they overlap each other. There isn't really a point where the two results diverge which may indicate that one of the results is growing faster than the other. Hence, it is appropriate to say that the code written for this algorithm is accurate and proves that the time complexity is indeed *O(ns)*.

4 Conclusions

Based on the results we have received from both Experimental and Theoretical Results, we can conclude that our algorithm of partitioning a given set into two parts so that the sum of the two parts is equal has a time complexity of ***O(ns)***.