Project 1 Readme Team Jim

Version 1 9/11/24

<table>
<tr><td>1</td><td>Team Name: Jim</td></tr>
<tr><td>2</td><td>Team members names and netids:
● Ignacio Sadurni (isadurni)
● Michael Sorenson (msorenso)
● Juan Pablo Rubero (jrubero)</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: Bin-Packing Problems - The “Knapsack” Problem: An equivalent of DumbSAT for a solver that returns a coin combination that works and has the least number of coins.</td></tr>
<tr><td>4</td><td>Overall success of the project:
The project was a success. All of the codes were able to correctly display the coin combination that worked with the least amount of coins. The advanced program, the dynamic programming code, was able to successfully run all test cases, even the larger test cases in under 0.04 seconds. Furthermore, we were able to successfully plot the data and output of each of the three codes, and were able to show that the knapsack problem gets exponentially slower with larger test cases.</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: Around 15 hours</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/isadurni/theory-project1</td></tr>
<tr><td>7</td><td>List of included files (if you have many files of a certain type, such as test files of different sizes, list just the folder): (Add more rows as necessary). Add more rows as necessary.
<table>
<tr><th>File/folder Name</th><th>File Contents and Use</th></tr>
<tr><td colspan="2">Code Files</td></tr>
<tr><td>bruteforce_jim.py</td><td>Python code for “dumb” knapsack problem solver</td></tr>
<tr><td>pruning_jim.py</td><td>Python code for “less dumb” knapsack problem solver</td></tr>
<tr><td>dynamicprogramming_jim.py</td><td>Python code for “smart” knapsack problem solver</td></tr>
<tr><td colspan="2">Test Files</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td>testdatagenerator_jim.py</td><td>Python program to generate customized random test data</td></tr>
<tr><td>Test Data
-> data_small1_jim.txt
-> data_small2_jim.txt
…
-> data_large5_jim.txt</td><td>Folder with 15 .txt files for all test data, 5 files for small, medium, and large sized cases</td></tr>
<tr><td colspan="2">Output Files</td></tr>
<tr><td>output_jim.pdf</td><td>File containing details regarding inputs as well as output for all test data</td></tr>
<tr><td colspan="2">Plots (as needed)</td></tr>
<tr><td>plots_jim.pdf</td><td>File containing screenshots of plots and analysis of data, comparing number of coins and execution time for the different algorithms</td></tr>
<tr><td>plots_excell_jim.xlsx</td><td>Excel file with table and plots.</td></tr>
<tr><td colspan="2">Readme</td></tr>
<tr><td>readme_jim.pdf</td><td>Readme file</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries:
● Python
○ standard library</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):
● Brute Force
○ List
○ Generator
● Brute Force with Pruning
○ List
○ Generator
● Dynamic Programming
○ List
○ Set
○ Table (matrix)</td></tr>
</table>

10	General operation of code (for each subproject): In all of the three codes, we parsed the stdin to get the target and coin jar from the input. We then passed these values into a function to calculate the solution. **Brute Force:** For the brute force function, we used a recursive function to generate every possible subset of coins from our jar. Our base case checked to see if the index was equal to the length of the jar. If it was, it then checked to see if the sum of the subset was equal to the target, and if true, it yielded the subset. If not it calls the function twice, once with adding the value at the index and once without. In our main function, we then looked for the subset that had the least amount of coins. **Pruning:** Our pruning algorithm is written the exact same as the brute force algorithm, except for in the recursive function we added an if statement after the base case that checks to see if the sum of the subset is greater than the target. If so, we return from the function so that it does not waste time doing unnecessary calculations. **Dynamic Programming:** The dynamic programming function initializes an empty table that is the size of the target. It starts at index 0 and looks forward, adding each coin to the list until it reaches the target. The program also makes sure the coin being added is still available in the jar. It then just returns the list at the last index. This function is a lot faster and more efficient than the first two.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. At first, we tested the codes with a generic [1,5,10,25] coin combination and a target that we knew should work. Once we confirmed that our programs correctly generated the output for that input, we tested the codes again with a coin combination and a target that we knew would not get a solution. Once we confirmed that as well, we moved on to creating the test cases. We created a python code that generated 15 test cases (5 small, 5 medium, 5 large) and the test cases generated random coin combinations with different amounts of coins within a specific range (depending on test case size) and target values. From these test cases we were able to determine the overall correctness of each code, and see how much it took for each program to run successfully.
12	How you managed the code development: We all three met up and combined our knowledge and previous programming experiences to develop the three different programs. We used python3 in Visual Studio Code.
13	Detailed discussion of results: The brute force and pruning algorithm had very similar results. As plotted, we see an exponential trendline where as the number of coins in the jar increases, the time taken to find the smallest combination of coins increases. Between these two algorithms, there wasn't any difference for the smaller test case, all performing efficiently. For the medium sized test cases it was evident that for larger numbers of coins, the pruning algorithm was more efficient, but still regarded inefficient because for the large test cases, they were both unable to complete the execution of the program because it would take too

	much time. These two algorithms proved the inefficiency of the knapsack problem and how for larger data test cases, the time increases 2^N. Our dynamic programming algorithm proved to be extremely efficient, barely noticing a change in runtime for the different sized test cases. The trendline increases very slightly, but we can infer based on our results of this project that for larger sized test cases, the runtime would increase exponentially as well. More detailed discussion of results and analysis are in the GitHub repository at Plots/plots_jim.pdf
14	How team was organized: We met up at the library during the afternoons and worked together in all stages of the project. We had a meeting for investigation/research, a meeting to develop and polish the code, then we individually ran the test cases in our own time, and then we met again one last time to create the plotted data and final submission work.
15	What you might do differently if you did the project again: If we did the project again, we would create the test data more smartly. It would have been good to have an even distribution for the number of coins throughout the test data. The way we programmed the generator so that the small, medium, and larger cases had a random number of coins within a range caused there to be gaps in the x axis of our plots and there were cases which had the same number of coins. Also we would create much larger test cases to test the efficiency of our dynamic programming algorithm and see how the execution time grows.
16	Any additional material: We do not have any additional material to add. Please feel free to reach out to us if you have any questions about our answers.