Project 1 Readme Team cdeleon2

Version 1 9/11/24

A single copy of this template should be filled out and submitted with each project submission, regardless of the number of students on the team. It should have the name readme_"teamname"

Also change the title of this template to "Project x Readme Team xxx"

<table>
<tr><td>1</td><td>Team Name: cdeleon2</td></tr>
<tr><td>2</td><td>Team members names and netids: Cesar De Leon - cdeleon2</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: Bin Packing Problems - The Coin Problem</td></tr>
<tr><td>4</td><td>Overall success of the project: Pretty successful</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 27</td></tr>
<tr><td>6</td><td>Link to github repository:https://github.com/rasecde/knapsack_cdeleon2</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><td>File/folder Name</td><td>File Contents and Use</td></tr>
<tr><td colspan="2">Code Files (src folder)</td></tr>
<tr><td>knapsackSolver_cdeleon2.py</td><td>This is the solver script which implements a dynamic programming algorithm to solve the knapsack/coin problem by determining if a target value can be achieved using specified coin denominations with counts. It reads test cases from a CSV file, computes solutions, and tracks execution time.</td></tr>
<tr><td>plotTimings_cdeleon2.py</td><td>This script reads implements the same code from the solver but instead it uses relevant information (like execution time and total coins), and generates a graph to visualize the relationship between execution time and the number of coins used.</td></tr>
<tr><td colspan="2">Test Files</td></tr>
</table>
</td></tr>
</table>

	data (folder)	This folder contains all the random test cases (in .csv format) generated by the testcase generator, including the testcase generator itself.
	check_knapsackTestCaseGen_cdeleon2.py (Also included in folder)	This testcase generator script was provided by the instructor but I tweaked it in a way that fits within the scope of this project. I started by setting 4 default coin types that are not multiples of each other (3, 7, 11, 13), and added a weight/count to each coin type to fulfill the "bounded" aspect of this problem. I made the small/medium/large testcase ranges based on my own judgement and testing.
	Output Files	
	output	This folder includes all the output from the solver (in a .txt format) after passing through the testcases from the data folder. This also includes the plot (.png format) generated from the plotting script after passing through data_randGraphTest_cdeleon2.csv.
	Plots (as needed)	
	plots_randGraphTest_cdeleon2.png	Like mentioned above, this plot was generated by passing through data_randGraphTest_cdeleon2.csv (50 testcases of each type) to plotTimings_cdeleon2.py, which produced a graph to visualize the relationship between execution time and the number of coins used.
8	Programming languages used, and associated libraries: **Programming Language:** Python **Libraries used:** argparse, random, datetime, textwrap, csv, time, matplotlib.pyplot	
9	Key data structures (for each sub-project): **KnapSack Solver Script:** Dynamic Programming Array, dictionaries, list of dictionaries, list of tuples	

	Plot TImings Script: Arrays (of integers/floats), data structures used in the solver (since it employes the same logic/algorithm)
10	General operation of code (for each subproject): **KnapSack Solver Script:** This script solves the knapsack problem using dynamic programming, where coins can be used a limited number of times to reach a target value. 1. **Reading test cases:** a. Reads in a CSV file containing the total value and coin data for each case generated by the testcase generator b. Each row is parsed into a format like case_type, total_value, [(coin_value, max_count), ...]. 2. **Dynamic Programming setup** a. Initializes a DP array to track which values can be reached b. Creates a list of dictionaries (used_coins) to store the coins used at each step 3. **Dynamic Programming Logic** a. Iterates over each coin and updates the DP array for achievable values b. Tracks the number of times each coin is used for each target value to make sure the algorithm respects the amount of available coins and does not exceed it 4. **Solution Check/Output** a. If the target value is achievable, prints the coin combination used in a format like [[coin_value: coins_used],...] b. If no solution is found, it prints that no solution exists **Plot TImings Script:** This script uses the exact same logic/functions from the knapsack solver (same functions), but only retrieves the values it needs (timings and total number of coins per testcase) and generates a plot to visualize the execution time versus the total number of coins. It also distinguishes between cases where solutions were found and where no solution was found (green and red respectively). 1. **Data Preparation for Plotting** a. Stores the x-axis (number of coins) and y-axis (execution time) values b. Separates data points into two categories: **Green**: Cases with a valid solution. **Red**: Cases without a solution. 2. **Generating Plot** a. Uses matplotlib to generate a scatter plot of execution time vs. number of coins. b. Adds a legend and titles to make necessary distinctions 3. **Displaying or Saving the Plot** a. After generating the plot, simplu uses plot.show() which displays a popup of the graph generated
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code: So all the testcases I used/added were generated by the testcase generator provided by

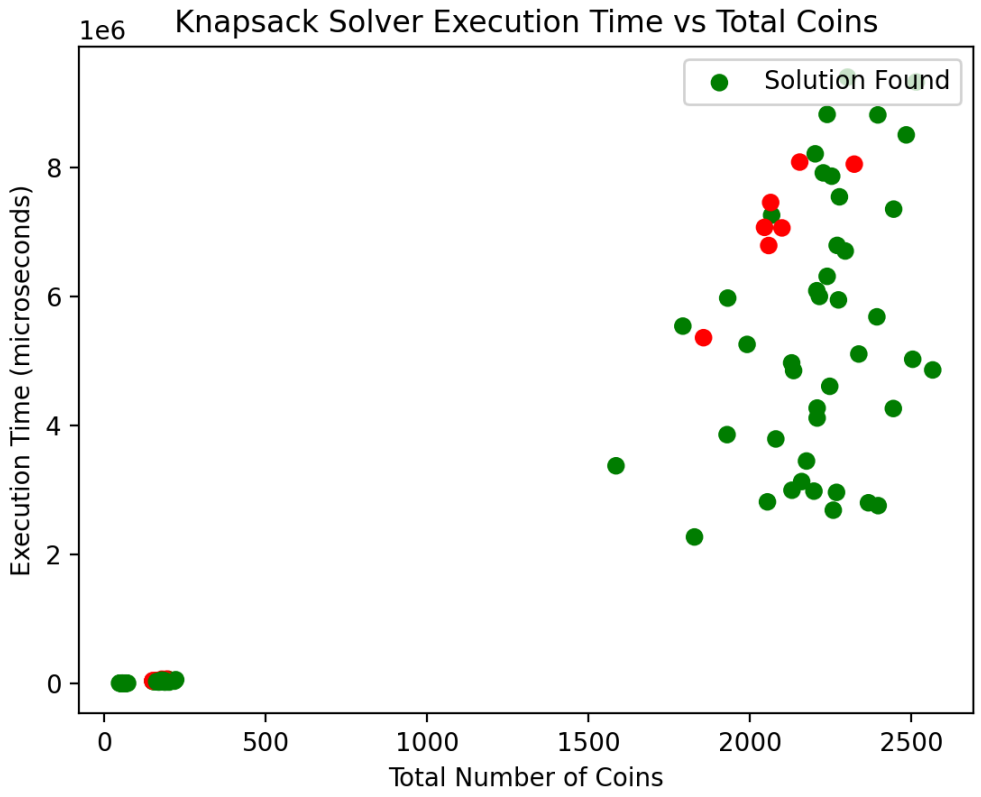
	the professor. However, as I mentioned earlier, I tweaked it in a way that fits within the scope of this project. I started by setting 4 default coin types that are not multiples of each other (3, 7, 11, 13), and added a weight/count to each coin type to fulfill the "bounded" aspect of this problem. The first test case I had smaller case ranges (e.g. small case had a coin weight per coin type of 10-20, medium was 20-30, and large was 50-100). However, I soon realized that when it got to the large test cases, I had unrealistic values which would often result in no solutions being found. The large testcase had a coin weight per coin type of 50-100, while the target value could range from 5000 - 10000. However, this did not make sense, even if I had 100 of each coin (3, 7, 11, 13) as using up all the coins wouldn't equate to the minimum target value. That is why I played around and made the small/medium/large testcase ranges based on my own judgement (with common sense) and testing. Making these adjustments helped me better judge the correctness of my code as I was getting more variability for testcases since there were times when there were no solutions. However, the fact that there was a solution most of the time meant that I was using appropriate/realistic ranges and my code was working as intended (by double checking the output values too).
12	How you managed the code development: I started off this project by taking a look at and understanding the knapsackTestCaseGen.py provided by the professor. After reading through the code, and making necessary changes, I then learned how to generate some testcases and observed what type of output I was receiving. Since I knew that I was reading in csv files, I then decided that my solver should be able to read in CSV files. When I started developing my solver, I started by making sure it could read in csv files properly, extracting the necessary values. Then I moved to the logic, where I determined that a dynamic programming approach would probably be most optimal for this problem. However, I also made sure to improve my logic (for making sure that it respected the amount of available coins) over time as I was having some trouble with the code because it was sometimes using more coins than allowed. Then the main() function for the solver was simply using the helper functions I made above, as well as making use of a timer (based off the DumbSAT example), to retrieve necessary values and formatting them appropriately. After making sure my solver was working properly after running a few testcases I generated, I decided to move on to the plotting part. I initially tried to read in the output .txt files I generated from the solver output, but I realized that it was pretty tricky reading in a .txt file for what I needed to do. That is why I just used the same helper functions and some lines from the main() from my solver in my plotting main(), except I was using matplotlib to plot and removed the print functions I had previously as they were unnecessary. At one point, my solver function was under the assumption that the knapsack problem was unbounded (using unlimited amounts of each coin), but then I had to make some changes to my code after some clarifications sent out by the professor. I also had an outputVerify_cdeleon2.py script that verified the output of the solver by making sure the sum of the coins used matched the target value, but I realized this was redundant since the logic used in the solver already accounted for this.

13

Detailed discussion of results:

Looking through the outputs of the solver, I think the results were pretty good. It did exceptionally well with small and medium-sized test cases, while taking a lot longer with large test cases since the values (amount of coins and target value) were a lot bigger. After double checking the coin combinations for the different test cases, the algorithm was indeed using the appropriate amount of coins, as well as the best combinations (on average) since it tended to use up the larger coins first. From this, I knew that the code was working as expected.

To figure out what the bounding curve and the worst-case time complexity, I generated a test case with 50 different examples of each case type (small, medium, large), to make sure I had enough data to plot. Then I ran it though the plotting script (took a while), and I got the following plot:



It was clear to see that there was not a linear growth, but more of an exponential growth from this plot between the total number of coins and execution time. Based on this observation, my guess for the worst-case time complexity formula would be O(n^2). This makes sense because for each coin, the solver iterates over all possible target values. If we assume n is the total number of coins and the total value is proportional to the input size, the inner loop can iterate up to n times for each coin. This gives an overall time complexity of O(n * n) = O(n^2) in the worst case

14	How team was organized: **This was an individual project**
15	What you might do differently if you did the project again: **Next time I would ask clarification questions on assumptions we can make for whatever problem I'm working on before I go straight to coding. Because I initially assumed that the problem was unbounded (since it wasn't specified in the description), I wasted a lot of time writing an algorithm and code that didn't really solve the knapsack problem efficiently or realistically.**
16	Any additional material: **N/A**