Project 1 Readme Team Garcias

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: Garcias</td></tr>
<tr><td>2</td><td>Team members names and netids
- Juan Chaia Gomez: jchaiago
- Michael Egan: megan3
- Rene Alzina: ralzina</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects:
● Knapsack
○ Brute Force
○ Backtracking
○ Best Case</td></tr>
<tr><td>4</td><td>Overall success of the project:
The project was successful since we were able to develop the 3 different solutions for the knapsack binpacking problem. We also successfully generated input files and test cases, and all of our generated graphs and csv files were consistent with what we expected. We were also able to work as a team and learn about how to collaborate when developing code.</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 12</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/ralzina/Project1-TOC</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.

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)

<table>
<tr><td>File/folder Name</td><td>File Contents and Use</td></tr>
<tr><td colspan="2">Code Files</td></tr>
<tr><td>Src</td><td>Knapsack_garcias.py</td></tr>
<tr><td>Src/helpers</td><td>Knapsack_helper_garcias.py</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td colspan="2">Test Files</td></tr>
<tr><td>Tests</td><td>Check_bestcase_knapsack_binpacking_tests.cnf
Check_knapsack_binpacking_tests.cnf</td></tr>
<tr><td>Src</td><td>Test_knapsack_garcias.py</td></tr>
<tr><td colspan="2">Output Files</td></tr>
<tr><td>Results</td><td>output_best_case_knapsack_results_garcias.csv
output_brute_force_knapsack_results_garcias.csv
output_btracking_knapsack_results_garcias.csv</td></tr>
<tr><td colspan="2">Plots (as needed)</td></tr>
<tr><td>Results</td><td>plot_best_case_knapsack_results_garcias.png
plot_btracking_knapsack_results_garcias.png
plot_brute_force_knapsack_results_garcias.png</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries:
Language: Python
Libraries:
● Collections for defaultdict
● Matplotlib for plotting
● Pytest for testing
● Every other library was part of the template</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):

We used a dictionary to map coin values to the amount of coins present. We also have a used dictionary that maps coin values to the amount of coins of that value that you use to reach the target. Both Brute Force and Backtracking used these data structures. However, Best Case added an extra dictionary to track the best case combination of coins.</td></tr>
<tr><td>10</td><td>General operation of code (for each subproject)
● Brute force:
○ This recursive function tries using one of each coin recursively to try all combinations. In each recursive call, it subtracts the used coin to the target. If the target ever reaches exactly 0, there is a solution, but if you try all combinations and it never reaches exactly 0, there is no solution.

● Backtracking:
○ This recursive function tries using one of each coin recursively. In each recursive call, it subtracts the used coin to the target. If the target ever</td></tr>
</table>

	reaches exactly 0, there is a solution, but if the current target is less than 0 then it stops checking. This pruning allows for a better time efficiency. ● Best case: ○ This recursive function is very similar to backtracking, but now we add an extra variable to keep track of the best case. Every time you pick a coin to get closer to the target, you check if it's the best case yet. If it is, then this best case variable becomes the current combination of coins. If not, then you keep trying other combinations. At the end, even if you didn't reach the final target, you will still get the best case which is the combination of coins that gets closest to the target. However, if you do reach exactly the target, you simply return the combination of coins that adds up to the target. In order to run the program: ● To run the program, you will position yourself in the root folder and run ```**uv run main.py**``` which will run the code with the input cnf file, take the time of each case, and generate a graph. ● To run tests, position yourself in the root folder and run ```**uv run pytest**``` which will run the test_knapsack_garcias.py script. This sets up the knapsack class and runs all test cases from the cnf file inside the tests folder. ● There's no script to run the test case as they can be run directly from the root folder with uv run pytest. ● All subprojects will run when you either run the program with uv run main.py or test it with uv run pytest
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. ● We decided to add multiple test cases to test the behavior of our program in different situations. ● There were some that tested when no coins were given and the target was 0, or when no coins were given and the target was greater than 0. ● Other test cases had the target be too big even if you used all the given coins. ● We also added test cases where there was a combination of coins that gave the exact target. ● The test cases were hand crafted by understanding the natural behavior of the bin packing problem and the expected outcome. ● Finally, we added another set of test cases specifically for the best case subproblem because this one doesn't just care if the problem is solvable or non-solvable. This actually cares about the combination of coins that gets you the closest to the target value, so we created a different set of tests which is inside the tests folder. ● We couldn't place our tests script inside a module_tests folder because pytest gave us an error since it couldn't import the modules from src because src was

	an external folder. The tests only worked when we placed our test script inside the root directory so that it could access src. The test script is called test_knapsack_garcias.py
12	How you managed the code development ● The way we managed the code development was by looking at the helper function examples for the other types of problems as a guide to creating our own knapsack helper function. ● We then connected it to our knapsack class with the implementation of the methods for brute force, backtracking, and best case. ● By analyzing the cnf template, we made both the input file in the input folder and the file with test cases in the tests folder. The input file has cases of increasing complexity to graph the time it takes to solve each case. The tests simply ensure that the functions work in different scenarios. After running the program, it writes the results into the results folder. This folder has both the csv files with the results of all input cases, and also a plot of the time complexity as we increase the magnitude of the problem. ○ Cnf format ■ To make our cnf format clearer for the cnf files in our input and tests folder, We will explain how it's formatted. <instance> <target> <solvable> p knap <n_unique_coins> <coin_value_i> <coin_amount_i> Where n_unique_coins refers to the amount of unique coins, and coin_value_i as well as coin_amount_i refer to the value, amount pairs for coin i, where i goes from coin 1 to coin n. <solvable> always has an S if the problem is solvable and an N if it's not solvable. This was helpful for our test cases to check the output of our program. However, for our input, we put a ? which basically means we are not trying to check or compare, we just cared about running the program.
13	Detailed discussion of results: ● The approximate time complexity for the program is O(2^m) where m is the total number of coins present since you're basically generating all possible subsets of all the coins present and looking for a subset whose numbers' sum is equal to the target. ● All plots showed that regardless of how efficient your program is, the time complexity is still exponential.

	○ Brute force took the largest amount of time since it tries all combinations even if the combination is no longer valid. ○ Best case was next because it always tries returning an answer even if no combination would add up to target. This means it must do a couple more operations than backtracking to keep track of the best case. ○ Backtracking is the fastest because it prunes and can stop trying coins if it determines the combination is no longer valid which speeds up the process. Also, since it's not best case, it doesn't try to find the best answer, it only cares if it's good or not so it can stop checking early and doesn't have any extra logic.
14	How team was organized Rene Alzina: ● Set up github by implementing a tests script, test cases, input, input parsing, helper functions, plotting, csv file generation, and the knapsack function. ● Implemented the brute force solution. Juan Chaia Gomez: ● Implemented the knapsack function with the best case solution. Michael Egan: ● Implemented the knapsack function with the backtracking solution.
15	What you might do differently if you did the project again Our team was organized by having each one of the three of us implement one of the sub problems for the knapsack problem. What could be better is to not have to create all the helper functions, input files, and test cases since that is not directly related to understanding problems with exponential complexity. All of that was simply to set up the project. It also involved abstract classes and advanced object-oriented programming that can be hard to work with. We met in the beginning to discuss how we were going to do the project and decide the problem and sub problems. After that, we each worked on our own to implement the sub problems. If we did the project again, we would start earlier and ensure we have read all the requirements. This project was hard to understand as instructions were scattered and there were many documents explaining what we needed to implement.
16	Any additional material: ● No test cases were provided for knapsack, so we had to develop our own. ● No helper function was provided for knapsack, so we had to develop our own. ● No knapsack script was provided, so we had to develop our own. ● We had to add a parsing function in dmaics_parser.py to parse our knapsack cnf file. ● We had to create our knapsack cnf input file. ● We updated constants to account for our input and test files, and made sure everything was saved properly into the results folder.