Project Readme Template

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: tset</td></tr>
<tr><td>2</td><td>Team members names and netids: Carlos Basurto, cbasurto</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects:

Bin Packing Problems - The "Knapsack" Problem.

Bruteforce Solver – attempts finding a solution by iterating through every possible combination returning a solution or, if no such solution exists, stating so alongside with the elapsed time to complete the operation. Afterwards, these elapsed times are classified as successes or failures and plotted accordingly.</td></tr>
<tr><td>4</td><td>Overall success of the project:

The project was entirely successful – the knapsack algorithm works, test and failure cases were generated or identified and accordingly processed and plotted, permitting the required analysis.</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete:

Around 5 hours.</td></tr>
<tr><td>6</td><td>Link to github repository:

https://github.com/cbasurtom/knapsack</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</td></tr>
<tr><td>knapsackBruteForce_tset.py</td><td>Main deliverable.

Input:
Test cases (dataTestCases_tset.txt)
Fail cases (checkFailCases_tset.txt)</td></tr>
</table>
</td></tr>
</table>

	Output: Text output (outputKnapsackSolutions_tset.txt) Timed plot (plotsTimedResults_tset.png) Takes the test cases and the fail cases as input and for each attempts to solve the knapsack problem via brute force: attempting all possible combinations of given coins. If successful, it will output the according coin combination. Otherwise, it will print no solution exists. In both cases, it will print and record the elapsed time it took to either find the solution or to attempt all combinations. These elapsed times will then be plotted, classifying them as either successes or failures. The plot is saved as plotsTimedResults_tset.png and all text output is captured in outputKnapsackSolutions_tset.txt. With the current parameters and cases, it takes around 15 minutes to run to completion locally.
knapsackTestCaseGen_tset.py	Knapsack problems generator. Courtesy of fellow student and provided by the professor, it has been modified to generate four types of tests, depending on the total amount to be calculated: Small – 50 - 100 Medium – 101 - 150 Large – 151 - 200 Giant – 201 - 250 Each case will generate an associated 4 coins, values ranging from 1 to 25, with which to attempt to solve the knapsack problem. This script was used to generate dataTestCases_tset.txt.
Test Files	
dataTestCases_tset.txt	Test cases. Generated using knapsackTestCaseGen_tset.py. Includes ten tests per type in the following format: {Test type}, {total}, {4 coins} Values are determined in the generator script.

<table>
<tr><td></td><td>
<table>
<tr><td>checkFailCases_tset.txt</td><td>Test cases guaranteed to fail.

Handpicked cases from all types guaranteed to fail, used to determine the variance of the total time required to go through all possible combinations with an increasing total – invaluable for generating the plot.</td></tr>
<tr><td colspan="2">Output Files</td></tr>
<tr><td>outputKnapsackSolutions_tset.txt</td><td>Text output of knapsackBruteForce_tset.py.

It presents information in the following format for every case:
Solving knapsack for Type: {}, Total: {}, Coins: {}
If solution exists: Solution found: {}
Otherwise, “No Solution”
Time taken for test case: {}</td></tr>
<tr><td colspan="2">Plots (as needed)</td></tr>
<tr><td>plotsTimedResults_tset.png</td><td>Plot depicting elapsed time vs total amount for knapsack problem.

The plot includes two lines, a green one depicting solutions and a red one depicting failures. As the total amount increases, the failure line can be seen increasing exponentially, whilst success lines maintain a very low elapsed time comparatively.</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries:

Python.
Associated libraries: argparse, contextlib, datetime, itertools, matplotlib, random, textwrap, time.</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):

For the entire project, the following data structures proved key:
ArgumentParser, Figure, List, TextIOWrapper, Tuple.
They were utilized for processing arguments, plotting data, storing data, redirecting output, and storing function output, respectively.</td></tr>
<tr><td>10</td><td>General operation of code (for each subproject):

Test Case Generator – After parsing the input, it sets the names of test case types and their associated total amount and coin ranges, prior to assigning a randomization seed and randomly generating the cases for the specified amount, exporting the results to the</td></tr>
</table>

	specified file. Brute Force Solver – After parsing the input, it declares four lists to store the elapsed times and totals of successes and failures, separately. While redirecting output to the specified file, the program will now process every case by calling the time_knapsack() function – first the test cases and then the guaranteed failures – and append the appropriate value to the appropriate list. The time_knapsack() function will first set a start time, then attempt every possible coin combination and if it finds one it will print the successful combination and return success alongside the elapsed time. If it finds no solution, it will return failure alongside the elapsed time. Once all tests have been processed, the lists will then be used to generate the Elapsed Time vs Total Amount for Knapsack Problem plot.
11	What test cases you used/added, why you used them, what did they tell you about the correctness of your code. Test cases – Randomly generated tests allowed me to check the correctness of my knapsack algorithm, as with every success the program would output the combination that would work, which was checked manually. These cases were occasionally manually manipulated to further provide correctness regarding some cases that had multiple, unique, or no solutions – ensuring the algorithm worked in all of these scenarios. Fail cases – Handpicked cases where failure was guaranteed, ensuring that the maximum time would be reached, as every possibility would need to be checked before the time_knapsack() function would return. With these failures, we could plot out the exponential curve of time required for time_knapsack() to check every combination.
12	How you managed the code development: First, I employed and edited the provided knapsack test case generator to obtain a set of problems to work with. Afterwards, I made a simple algorithm to solve the cases and return the elapsed time it took to process, which I would later refine with the itertools library combinations in order to ensure that the knapsack algorithm does check for all possible combinations prior to stating no solution exists. Then, I plotted the elapsed time of the solved or failed cases against the total amount for the given knapsack problem. Realizing that it would be much more useful to understand the time complexity trend with more cases, I then designed a number of cases who were guaranteed to fail, permitting the analysis of NP problems behavior. I finally used the redirect_stdout function from contextlib to redirect and capture the text output as well as save the plot as a png.
13	Detailed discussion of results: Text output – the text output states the case it is considering, whether it was successful or not, and the time required to reach this conclusion. Additionally, if it is successful it will provide the combination of coins that makes it possible. Not only did this permit a verification of correctness of the algorithm, but provide a numerical guide for the user to follow as the length of the answers continuously increases and the elapsed time of

	operations, particularly those that fail, increment exponentially. Plot output – Figure plotting the time required to compute the test cases against the total amount for the given problem, depicting failures in red and successes in green. This figure is likely the most important product of this project, as with a high number of guaranteed failures it was able to capture and depict the exponential curve of required computation time as the number of possibilities to consider also exponentially increases. Overall, I obtained something much more volatile than I expected: seeing the computation time go from 10 seconds to 100 in the increase of around a hundred in the knapsack total amount forced my hand into really reducing the total number. The brutal nature of NP problems manifests itself in its exponential might in the results, deeply felt in my regular need to go for a walk in between test runs.
14	How team was organized: Not available, as there was only one team member who completed the entire project.
15	What you might do differently if you did the project again: I would perhaps consider different knapsack algorithms to compare time efficiency of the different methods, as well as exploring avenues with which to expand the total knapsack number. I am aware that the purpose of this project was to create a “dumb” algorithm that checks every possibility prior to admitting defeat, but it would be an interesting challenge to see if there was any way to optimize it at all. With our current setup alongside randomness there is no simple greedy algorithm that can be applied, but the challenge is hardly unappealing. I suppose that’s why NP problems have the notoriety they do, waiting to be solved.
16	Any additional material: None.