Readme_MJ

Version 1 10/16/24

A copy of this file should be included in the github repository with your project. Change the teamname above to your own

1. Team name: **MJ**
2. Names of all team members: **Max Johnson**
3. Link to github repository: **https://github.com/maxj723/theory-project-one**
4. Which project options were attempted: **Knapsack Problem solved like DumbSAT**
5. Approximately total time spent on project: **>5 hours**
6. The language you used, and a list of libraries you invoked: **Python, matplotlib.pyplot, numpy, scipy.optimize, time, csv, os**
7. How would a TA run your program (did you provide a script to run a test case?):
   Two options:
   - Use `python3 check_knapsackvalid_max.py` to run the validating test cases on the function.
   - Use `python3 knapsackSAT_MJ.py` to run the function on a large dataset and produce a input vs time plot
8. A brief description of the key data structures you used, and how the program functioned.
   To solve the knapsack problem using a dumbsat method, I used recursive backtracking. The recursion function had three inputs–curr(current sum), i(current index), and combo(current coin combination). It starts with base cases–1)if the current sum is equal to the desired value, it will return the current combination. 2)if the current index is greater than the last coin index, it will return 'no combo found.' 3) if the current sum is greater than the desired value, it will also return 'no combo found.'

   From there the "skip" does not add the current coin to the sum and increments the index, going a level deeper in recursively. "Take" adds the current coin value to the sum and adds the coin to the combo, then increments the index and goes a level deeper.

   At the end, the backtracking function is called with a 0 sum, 0 index, and empty combo. This will return a list of valid coins that sum to the desired value or None if no valid combo is found.

9. A discussion as to what test cases you added and why you decided to add them (what did they tell you about the correctness of your code). Where did the data come from? (course website, handcrafted, a data generator, other)
   All of the test cases in the Test Files folder came from a slightly modified version of the test cases generator that was provided in the class files. They were used to test lots of possibilities for the sake of generating the inputs vs time plot. For the validity test, I just put together a couple small cases that would be easy to

determine whether correct. That way, when testing with my own program, I could determine if it worked right.

10. An analysis of the results, such as if timings were called for, which plots showed what? What was the approximate complexity of your program?
The plot_InputsTime_max.jpg file shows a plot of the # of inputs vs execution time. It generally shows that as the number of inputs increases, the execution time increased exponentially. This makes sense because with each addition of an input, every combination of inputs would need to be done again. This shows that the time complexity of the program is 2^n. This is also shown on the plot, where the line of best fit is y = 2^x.

11. A description of how you managed the code development and testing.
Initially, I made a first draft of the program using recursion. It worked, but when I tested it on larger test cases from the provided test case generator, I started hitting max recursion depth. This was a problem because the test cases were too big for what I was trying to do. Then I tried modifying my program, but after lots of iterations, I decided that I would just make some of the test cases smaller. So I modified the generator, and it started working. Afterwards, I just cleaned up the files in the git repository.

12. Did you do any extra programs, or attempted any extra test cases: **N/A**