Problem Set 3: Space Cows Transportation

Greedy algorithm
Brute force algorithm
Dynamic Programming

Introduction

A colony of Aucks (super-intelligent alien bioengineers) has landed on Earth and has created new species of farm animals! The Aucks are performing their experiments on Earth, and plan on transporting the mutant animals back to their home planet of Aurock. In this problem set, you will implement algorithms to figure out how the aliens should shuttle their experimental animals back across space.

Getting Started

Please keep `ps3_partition.py`, `ps3_cow_data.txt`, and `ps3_cow_data_2.txt` in the same folder as `ps3a.py` and `ps3b.py`.

Part A: Transporting Cows Across Space

The aliens have succeeded in breeding cows that jump over the moon! Now they want to take home their mutant cows. The aliens want to take all chosen cows back, but their spaceship has a weight limit and they want to minimize the number of trips they have to take across the universe. Somehow, the aliens have developed breeding technology to make cows with only integer weights.

The data for the cows to be transported is stored in **ps3_cow_data.txt,** and another set of cows *for another separate transport* are in **ps3_cow_data_2.txt** (There are two files for you to read from and test individually). All of your code for Part A should go into **ps3a.py**.

Problem A.1: Loading Cow Data

First we need to load the cow data from the data file **ps3_cow_data.txt**.
The file **ps3_cow_data_2.txt** is given as another file that you can read and test from (We also encourage you to write your own cow list files for testing!), but for now, just work with **ps3_cow_data.txt**

You can expect the data to be formatted in pairs of `x,y` on each line, where `x` is the name of the cow and `y` is a number indicating how much the cow weighs in tons. Here are the first few lines of **ps3_cow_data.txt**:

```
Maggie,3
Herman,7
Betsy,9
…
```

You can assume that all the cows have unique names.

Implement the function `load_cows(filename)` in ps3a.py. It should take in the name of the data text file as a string, read in its contents, and return a dictionary that maps cow names to their weights.

Hint: If you don't remember how to read lines from a file, check out the online python documentation, which has a chapter on Input and Output that includes file I/O here: https://docs.python.org/3/tutorial/inputoutput.html

Some functions that may be helpful: `str.split` `open` `file.readline` `file.close`

Problem A.2: Greedy Cow Transport

One way of transporting cows is to always pick the heaviest cow *that will fit* onto the spaceship first. This is an example of a greedy algorithm. So if there's only 2 tons of free space on your spaceship, with one cow that's 3 tons and another that's 1 ton, the 1 ton cow will still get put onto the spaceship.

Implement a greedy algorithm for transporting the cows back across space in `greedy_cow_transport`. The result should be a list of lists, where each inner list represents a trip and contains the names of cows taken on that trip.

Note:

- **Make sure not to mutate the dictionary of cows that is passed in!**

Assumptions:

- The order of the list of trips does not matter. That is, [[1,2],[3,4]] and [[3,4],[1,2]] are considered equivalent lists of trips.
- All the cows are between 0 and 10 tons in weight
- All the cows have unique names
- If multiple cows weigh the same amount, break ties arbitrarily

- The spaceship has a cargo weight limit (in tons), which is passed into the function as a parameter

Example:
Suppose the spaceship has a weight limit of 10 tons and the set of cows to transport is {“Jesse”: 6, “Maybel”: 3, “Callie”: 2, “Maggie”: 5}.

The greedy algorithm will first pick Jesse as the heaviest cow for the first trip. There is still space for 4 tons on the trip. Since Maggie will not fit on this trip, the greedy algorithm picks Maybel, the heaviest cow that will still fit. Now there is only 1 ton of space left, and none of the cows can fit in that space, so the first trip is [“Jesse”, “Maybel”].

For the second trip, the greedy algorithm first picks Maggie as the heaviest remaining cow, and then picks Callie as the last cow. Since they will both fit, this makes the second trip [“Maggie”, “Callie”].

The final result then is [[“Jesse”, “Maybel”], [“Maggie”, “Callie”]].

Problem A.3: Brute Force Cow Transport

Another way to transport the cows is to look at every possible combination of trips and pick the best one. This is an example of a brute force algorithm.

Implement a brute force algorithm to find the minimum number of trips needed to take all the cows across the universe in `brute_force_cow_transport`. The result should be a list of lists, where each inner list represents a trip and contains the names of cows taken on that trip.

Notes:

- **Make sure not to mutate the dictionary of cows!**
- In order to enumerate all possible combinations of trips, you will want to work with set partitions. We have provided you with a helper function called `get_partitions` that generates all the set partitions for a set of cows. More details on this function are provided below.

Assumptions:

- Again, you can assume that order doesn't matter. [[1,2],[3,4]] and [[3,4],[1,2]] are considered equivalent lists of trips, and [[1,2],[3,4]] and [[2,1],[3,4]] are considered the same partitions of [1,2,3,4].
- You can assume that all the cows are between 0 and 10 tons in weight
- All the cows have unique names
- If multiple cows weigh the same amount, break ties arbitrarily

- The spaceship has a cargo weight limit (in tons), which is passed into the function as a parameter

Helper function `get_partitions`:

To generate all the possibilities for the brute force method, you will want to work with set partitions (http://en.wikipedia.org/wiki/Set_partition). For instance, all the possible 2-partitions of the list [1,2,3,4] are [[1,2],[3,4]], [[1,3],[2,4]], [[2,3],[1,4]], [[1],[2,3,4]], [[2],[1,3,4]], [[3],[1,2,4]], [[4],[1,2,3]].

To help you with creating partitions**,** we have included a helper function `get_partitions(list)` that takes as input a list and returns a generator that contains all the possible partitions of this list, from 0-partitions to n-partitions, where n is the length of this list.

To use generators, you must iterate over the generator to retrieve the elements; you cannot index into a generator! For instance, the recommended way to call `get_partitions` on a list [1,2,3] is the following:

```
for partition in get_partitions([1,2,3]):
    print(partition)
```

Try out this snippet of code to see what is printed!

Generators are outside the scope of this course, but if you're curious, you can read more about them here: http://wiki.python.org/moin/Generators.

Example:

Suppose the spaceship has a cargo weight limit of 10 tons and the set of cows to transport is {“Jesse”: 6, “Maybel”: 3, “Callie”: 2, “Maggie”: 5}.

The brute force algorithm will first try to fit them on only one trip, [“Jesse”, “Maybel”, “Callie”, “Maggie”]. Since this trip contains 16 tons of cows, it is over the weight limit and does not work.

Then the algorithm will try fitting them on all combinations of two trips. Suppose it first tries [[“Jesse”, “Maggie”], [“Maybel”, “Callie”]]. This solution will be rejected because Jesse and Maggie together are over the weight limit and cannot be on the same trip. The algorithm will continue trying two trip partitions until it finds one that works, such as [[“Jesse”, “Callie”], [“Maybel”, “Maggie”]].

The final result is then [[“Jesse”, “Callie”], [“Maybel”, “Maggie”]].

Note that depending on which cow it tries first, the algorithm may find a different, optimal solution. Another optimal result could be [[‘Jesse’, ‘Maybel’],[‘Callie’, ‘Maggie’]].

Problem A.4: Comparing the Cow Transport Algorithms

Implement `compare_cow_transport_algorithms`. Load the cow data in **ps3_cow_data.txt**, and then run your greedy and brute force cow transport algorithms on the data to find the minimum number of trips found by each algorithm and how long each method takes. Use the default weight limits of 10 for both algorithms.

Note: Make sure you've tested both your greedy and brute force algorithms before you implement this!

Hints:

- You can measure the time a block of code takes to execute using the time.time() function as follows:

```
start = time.time()
## code to be timed
end = time.time()
print end - start
```

  This will print the duration in seconds, as a float.

- Using the given default weight limits of 10 and the given cow data, both algorithms should not take more than a few seconds to run.

Problem A.5: Write-up

1. What were your results from `compare_cow_transport_algorithms`? Which algorithm runs faster? Why?
2. Does the greedy algorithm return the optimal solution? Why/why not?
3. Does the brute force algorithm return the optimal solution? Why/why not?

Part B: Golden Eggs

After the Aucks transport the cows, one of their interns finds flocks of golden geese. Due to budget cuts they are forced to downsize their ships so they can't simply take the geese back, but instead decide to take their golden eggs back. Their ships can only hold a certain amount of weight, and are very small inside. So, because all the eggs are the same size, but have different weights, they want to bring back as few eggs as possible that fill their ship's weight

limit. Golden eggs are all the same size, but may have different densities, thus 1 two-pound egg is better than 2 one-pound eggs.

Problem B.1: Dynamic Programming: Hatching a Plan

The Aucks want to carry as few eggs as possible on their trip as they don't have a lot of space on their ships. They have taken detailed notes on the weights of all the eggs that geese can lay in a given flock and how much weight their ships can hold.

Implement a dynamic programming algorithm to find the minimum number of eggs needed to make a given weight for a certain ship in `dp_make_weight`. The result should be an integer representing the minimum number of eggs from the given flock of geese needed to make the given weight. Your algorithm does not need to return what the weight of the eggs are, just the minimum number of eggs.

Notes:

- If you try using a brute force algorithm on this problem, it will take a substantially long time to generate the correct output if there are a large number of egg weights available.
- You may implement your algorithm using the top-down recursive method or the bottom-up tabulation method. **The `memo` parameter in `dp_make_weight` is optional. You may or may not need to use this parameter depending on your implementation.**

Assumptions:

- All the eggs weights are unique between different geese, but a given goose will always lay the same size egg
- The Aucks can wait around for the geese to lay as many eggs as they need.

Example:

Suppose the first ship can carry 99 pounds and uses the first flock of geese they find, which contains geese that lay eggs of weights 1, 5, 10, and 20 pounds.

Your dynamic programming algorithm should return 10 (the minimum number of egg needed to make 99 pounds is 4 eggs of 20 pounds, 1 egg of 10 pounds, 1 egg of 5 pounds, and 4 eggs of 1 pound).

Hints:

- Dynamic programming involves breaking a larger problem into smaller, simpler subproblems, solving the subproblems, and storing their solutions. What are the subproblems in this case? What values do you want to store?
- This problem is analogous to the knapsack problem. Imagine the eggs are items you are packing. What is the objective function? What is the weight limit in this case? What are the values of each item? What is the weight of each item?

Problem B.2: Write-up

1. Explain why it would be difficult to use a brute force algorithm to solve this problem if there were 30 different egg weights. You do not need to implement a brute force algorithm in order to answer this.
2. If you were to implement a greedy algorithm for finding the minimum number of eggs needed, what would the objective function be? What would the constraints be? What strategy would your greedy algorithm follow to pick which coins to take? You do not need to implement a greedy algorithm in order to answer this.
3. Will a greedy algorithm always return the optimal solution to this problem? Explain why it is optimal or give an example of when it will not return the optimal solution. Again, you do not need to implement a greedy algorithm in order to answer this.
