Problem Set 3: Space Cows and Chickens

Handed out: Tuesday, November 10, 2015
Due: 11:59pm, Wednesday, November 18, 2015

Introduction

A colony of Aucks (super-intelligent alien bioengineers) has landed on Earth and have 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

Download **ps3.zip** from the website.

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**. 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**.

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/2/tutorial/inputoutput.html

Some functions that may be helpful:
```
string.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:

- 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:

- Though we're working with lists, you can assume that order doesn't matter -- that is, [[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! An example of a generator that you have probably used before is the number generator returned by the `xrange` function (the `range` function returns a list of numbers as opposed to 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”]].

Problem A.3: 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.

Hint: 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.

Problem A.4: Writeup

Answer the following questions in a PDF file called **ps3.pdf**.

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?
4. Imagine that instead of taking the entire cow, we could also chop the cows up and take fractions of them onto the spaceship. For example, if there was 2 tons of space left in the spaceship and we only had a 5 ton cow left, we could just take 2 tons of that cow and leave the 3 remaining tons for the next trip. Under these conditions, would a greedy algorithm return the optimal solution? Would a brute force algorithm return the optimal solution? Explain.

Part B: Space Chickens and Cows

Before the Aucks can transport the cows, they discover that one of their interns had successfully bred giant mutant chickens that lay golden eggs. They decide they want to take these money-making fowl back to Aurock instead of the cows.

However, the Auck spaceship runs on methane, and the Auck engineers realize that their methane supply has completely run out. One engineer points out that the cows can produce methane to fuel the ship, so as long as they bring the right number of cows on the trip, the cows could produce enough fuel to make it home. Therefore, they cannot just take chickens with them on the one trip.

Furthermore, their spaceship is rather old and in need of repairs. They can only take one trip from Earth to Aurock before the spaceship will break down. Therefore, they may not be able to transport all the cows and chickens back to Aurock.

The data for the mutant cows is stored in **ps3_cow_data.txt**. The data for the mutant chickens is stored in **ps3_chicken_data.txt**. All of your code for Part B should go into **ps3b.py**.

Problem B.1: Loading the Data

We need to write a function to load the chicken data from the data file **ps3_chicken_data.txt**. It should be very similar to the `load_cows` function from Part A.

You can expect the data to be formatted in pairs of `x,y,z` on each line, where `x` is the name of the chicken, `y` is a number indicating how much the chicken weighs in tons, and `z` is the mass of the eggs (in kilograms) that chicken `x` lays. Here are the first few lines of **ps3b_chicken_data.txt**:

```
Berta,6,10
Alphonisa,4,5
Fluffy,9,5
…
```

You can assume that all the chickens have unique names.

Implement the function `load_chickens(filename)` in ps3b.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 chicken names to a tuple of their weights and egg mass.

We have imported your `load_cows` function from Part A at the top of **ps3b.py**. You do not need to reimplement `load_cows`.

Problem B.2: Greedy Animals Transport

The heavier golden eggs a chicken lays, the more valuable it is. Since their spaceship can only make one trip from Earth to Aurock, the Aucks want to transport the chickens that will give them the *highest value*, or the chickens that will lay the heaviest total golden egg mass. Cows do not have value, but they also need to transport cows in order to fuel their trip.

Note:

- **Make sure not to mutate the dictionary of cows or the dictionary of chickens!**

Assumptions:

- All the cows and chickens have unique names

- All the cows and all the chickens are between 0 and 10 tons in weight
- All chickens lay golden eggs that have a mass between 0 and 10 kilograms. They will be put in space hibernation during the journey, so they will not lay any eggs during the trip.
- The value of a chicken is equivalent to its golden egg mass
- Cows do not add any value
- The spaceship has a cargo weight limit (in tons), which is passed into the function as a parameter
- The spaceship's cargo weight is only dependent on the total weight of the cows and chickens it is transporting
- The spaceship can only make one trip from Earth to Aurock so not all cows and chickens may be transported back to Aurock
- The amount of methane needed to fuel the spaceship for the trip home is the weight of the cargo + a fuel constant. The fuel constant is passed into the function as a parameter.
- The amount of methane a cow produces is twice its weight

You will come up with your own greedy algorithm to pick the list of cows and chickens that the Aucks should transport. Remember that the Aucks want to transport the chickens that will give them the highest value.

Hint: Your greedy algorithm should pick a single animal at a time to add to the spaceship. After each pick, the spaceship should be able to still make the journey home. You will continue picking animals until the constraints can no longer be satisfied, or you cannot add any more value to the trip. **It's up to you to determine how to pick the next animal, and when to stop picking animals for the trip.**

Before you begin coding, answer the following questions in the same PDF file **ps3.pdf**:

1. What is the objective function?
2. What are the constraints?
3. What strategy will your greedy algorithm follow? In other words, how will your greedy algorithm pick which animals to take on the one trip? Write your algorithm in psuedocode, and justify your design choice(s). **There is no one correct answer** so you may pick whatever strategy you want as long as it addresses the objective function and constraints you identified in the previous questions.
4. What do you expect your greedy algorithm to return when you run it on the given chicken and cow data if the spaceship can only take one trip? Assume that the spaceship cargo weight limit is 20 tons and the fuel constant is 5. Write out the names of the cows and chickens that your greedy algorithm would return for that one trip.

Implement your greedy algorithm as `greedy_animal_transport` **in ps3b.py**. The result should be a list of names of cows and chickens taken on the one trip from Earth to Aurock. The names may be in any order.

Problem B.3: Writeup

Answer the following questions in the same PDF file **ps3.pdf**.

A. Give an example of when the solution found by your greedy algorithm would not be optimal. You may use the cows and chickens data provided or provide your own data set for this. You may use the default cargo weight limit and fuel constraints given, or you may make up your own.
B. Explain why it would be difficult to use a brute force algorithm if there were 20 different cows and 20 different chickens. **You do not need to implement the brute force algorithm**.

Hand-In Procedure

1. Save

Save your solutions as **ps3a.py** and **ps3b.py**. Save your answers to Part A.4 and answers from Part B.2-3 in **ps3.pdf**

2. Time and Collaboration Info

At the start of each file, in a comment, write down the number of hours (roughly) you spent on the problems in that part, and the names of the people you collaborated with. For example:

```
# Problem Set 3
# Name: Jane Lee
# Collaborators: John Doe
# Time:
#
... your code goes here ...
```

3. Sanity checks

After you are done with the problem set, do sanity checks. Run the code and make sure it can be run without errors.

Make sure that any calls you make to different functions are under if __name__ == '__main__'. This will make your code easier to grade and your graders and TA's happy.

4. Submit

Upload all your files to Stellar. If there is some error uploading, email the file to 6.0002-staff [at] mit.edu.

You may upload new versions of each file until the 11:59pm deadline, but anything uploaded after that will be ignored, unless you still have enough late days left.