6.0002 Problem Set 1: Space Cows

Handed out: Monday October 23, 2017
Due: 11:59pm on Wednesday, November 1, 2017

Introduction: Transporting Cows Through Space

A colony of Aucks (super-intelligent alien bioengineers) has landed on Earth and have succeeded in breeding cows that jump over the moon! They want to take all their cows back, but their spaceship has a weight limit, so they won't all fit in one trip. So the aliens need your help to minimize the number of trips they have to take across the universe.

Getting Started

- Do not rename files, change helper functions, or change function/method names
- You will need to keep **`ps1_partition.py`**, **`test_ps1.py`**, **`ps1_cow_data.txt`** in the same folder as **`ps1.py`**.
- Please review the Style Guide on Stellar. We will deduct points during checkoff for violations.

Problem 1: Loading Cow Data (5%)

First we need to load the cow data from the data file **ps1_cow_data.txt**.
You can expect the data to be formatted in pairs of **`x,y`** on each line, where **`x`** is the cow weight in tons and **`y`** is the name of the cow. You can assume that all the cows have unique names, and additionally, you can assume that the weight **`x`** will always be an integer value. Do *not* assume that the cows all have unique weights. Here are the first few lines of **`ps1_cow_data.txt`**:

```
3,Maggie
3,Herman
9,Betsy
```

Implement the function **`load_cows(filename)`** in **`ps1.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 each cow name with its corresponding weight.

For example, the three cows above would be represented as
`{'Maggie': 3, 'Herman': 3, 'Betsy': 9}`

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`

You should now pass the test **`test_load_cows`** in test_ps1.py.

Problem 2: Greedy Cow Transport (20%)

One way of transporting cows is to always pick the heaviest cow remaining that can fit onto the spaceship first. This is an example of a greedy algorithm. So if there are 2 tons of free space on your spaceship, and the two cows left to load weigh 1 ton and 3 tons respectively, the 1 ton cow will get put onto the spaceship.

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

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

Assumptions:

- The order of the trips does not matter. That is, `[[1,2],[3,4]]` and `[[3,4],[1,2]]` are considered equivalent lists of trips.
- 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 `{2: 'Callie', 3: 'Maybel', 5: 'Maggie', 6: 'Jesse'}`.

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 next heaviest cow that can 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 chooses Maggie, the heaviest remaining cow, and then picks Callie, the last cow. Since they will both fit, this makes the second trip `['Maggie', 'Callie']`.

The final result then is `[['Jesse', 'Maybel'], ['Maggie', 'Callie']]`.

You should now pass **`test_greedy_cow_transport`** in test_ps1.py.

Problem 3: Brute Force Cow Transport (20%)

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 need to use set partitions. We have provided you with a helper function called `get_partitions` that generates all the set partitions for a set of cows (or other inputs). 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].
- 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 weight limit of 10 tons and the set of cows to transport is {2:“Callie”, 3:“Maybel”, 5:“Maggie”, 6:“Jesse”}.

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’]].

You should now pass **`test_brute_force_cow_transport`** in test_ps1.py.

Problem 4: Comparing the Cow Transport Algorithms (5%)

Implement `compare_cow_transport_algorithms`. Load the cow data in **ps1_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.
  Note: your answer may be printed in scientific notation -- i.e. you will see 2e-5 seconds instead of .00002 seconds.
- 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.
- Be ready to explain what you find during checkoff.

Problem 5: Dynamic Programming Cow Transport (20%)

The Aucks further master their breeding methods. They are now able to breed cows of specific weights. The Aucks prefer quantity over quality, as such, they want to send as many cows as possible while completely filling the ship.

Implement a dynamic programming algorithm to find that maximum number of cows that add up to the ship weight limit, `dp_make_weight` given a set of integer cow weights. The result should be an integer, representing the maximum number of cows that can fit onto the ship. If no combination of cows will equal the target weight, return `None` The algorithm does not need to return the weights of the cows used, or the weight of the cows altogether (which would be always just be the weight limit).

A solution that does not use dynamic programming will receive zero points for this problem. If your code is taking longer than 5 seconds to run on each test case, you are probably not using dynamic programming.

Notes:

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

Examples:

Suppose the ship can carry 64 tons and the Aucks are able to breed cows with weights 3, 5, 8, and 9 tons.

- Your dynamic programming algorithm should return 20, the maximum number of cows with a weight of 64 tons. In this case, eighteen 3-ton cows and two 5-ton cows.

Suppose the ship can carry 63 tons and the Aucks are able to breed cows with weights 2, 4, 8, and 12 tons.

- Your algorithm should return `None` because no combination of these weights equals 63.

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, which you saw in lecture. Imagine the cows 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?

You should now pass all of the tests in test_ps1.py. If you fail **`test_very_large_input`**, this is a sign that you are not correctly using dynamic programming. Come to office hours for help!

Checkoff: (30%)

Attend an OH between 11/2 and 11/13 to receive your checkoff. Be prepared to explain your code and the algorithms used.

Hand-In Procedure

1. Save

Make sure your code is saved with the name **ps1.py**.

2. Time and Collaboration Info

At the start of ps1.py, 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 1
# Name: Jane Lee
# Collaborators: John Doe
# Time: 8 hours
#
... your code goes here ...
```

3. Sanity checks

After you are done with the problem set, do sanity checks. Run your files and make sure they 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 TAs happy.

4. Submit

Upload all your files to the “Problem Sets” page linked from Stellar. If there is some error uploading, please try the troubleshooting techniques outlined on the site, and as a last resort 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.