Problem Set 9: Optimizing your Course Load

Handed out: Thursday, April 25, 2013
Due: 11:59pm, Thursday, May 2, 2013

Introduction

At an institute of higher education which shall remain nameless, it used to be the case that a human advisor would help each student formulate a list of subjects that would meet the student's objectives. However, because of financial troubles, the institute has decided to replace human advisors with software. Given the amount of work a student is willing to do, the program returns a list of subjects that maximizes the amount of value. In particular, each course has a certain amount of value to the student, but also a cost - the number of hours it requires a week. Our goal is that the student will tell the software advisor how many hours a week they are willing to spend on courses, and the advisor will choose a list of subjects whose total workload is less than or equal to that, such that the total value of the courses chosen is maximized. But there's one more constraint - due to the aforementioned financial troubles, some of the courses are now "lotteried", and there is a maximum number of lotteried courses a student can take.

We've already seen how to implement greedy and brute force algorithms, so in this problem set we will look at other ways to implement these algorithms and also how to optimize the brute force solution. In particular, the goal of this problem set is to implement an optimization problem with both an objective function and at least one constraint.

Workload

Please let us know how long you spend on each problem. We want to be careful not to overload you by giving out problems that take longer than we anticipated.

Collaboration

You may work with other students. However, each student should write up and hand in his or her assignment separately. Be sure to indicate with whom you have worked. For further details, please review the collaboration policy as stated in the Course Information.

Getting Started

Download ps9.zip, which contains the skeleton code and data you will use for this problem set.

- ps9.py: the skeleton you'll fill in for Problems 1-5.
- subjects.txt: the list of subjects with value, work, and time information.
- subjects_small.txt: a smaller list of subjects.

This problem set is not intended to be extremely code-intensive, but rather to make you think about your algorithms before coding. Try to plan out your code (pseudocode!) before you sit down and actually write code, and if you get stuck, make sure you can work through the problem on paper first!

Problem 1: Building a Subject List

The first step is to **fill in the implementations for the class `Subject` and function `loadSubjects` in `ps9.py`** . `Subject` is a class to represent a single subject, and `loadSubjects` loads a list of subjects from a file.

For `loadSubjects`, each line of the file contains a string of the form "name,value,work,lottery", where the name is a string, the value is an indicating how much a student learns by taking the subject, the work is an integer indicating the number of hours a student must spend to pass the subject, and the lottery is an integer which equals to 1 if the subject is a lottery subject and 0 if not. `loadSubjects` should return a list of Subject instances.

We've provided a function called `printSubjects` to neatly display the contents of such subject lists. Here is a sample output (truncated):

```
>>> subjects = loadSubjects("subjects.txt")
>>> printSubjects(subjects)
Course  Value   Work    Lottery
======  =====   ====    ====
1.00    6       2       0
1.01    10      9       0
1.02    5       18      1
1.03    7       18      0
1.04    1       12      0
[... TRUNCATED ...]
9.17    9       11      1
9.18    7       16      0
9.19    9       9       1

Number of subjects: 342
Total value: 1919
Total work: 3711
Total number of lottery subjects: 72
```

You might want to test your implementations on some smaller subject lists, so you should use the provided `subjects_small.txt` file for you to test with as you work through this problem set.

Hint: Use the python `open()` function to open files and the `close()` function to close the files. The `open()` function returns an instance of `file`, which can be iterated through in a `for` loop, line by line.

Problem 2: Subject Selection using Greedy Optimization

The institute has hired you to implement the program. They have asked that you implement a greedy method to formulate a list of subjects that satisfies each student's constraint (the amount of work the student is willing to do) while maximizing the value of the selected courses. There is also an additional constraint forced by the institute, which is the number of lotteried subjects the students are alowed to take.

The algorithm should pick the "best" subjects first. The notion of "best" is determined by the use of the comparator. The comparator is a function that takes in two arguments -- each of which is a Subject instance -- and returns -1 indicating whether the first argument is "better" than the second, 1 indicating that the first argument is "worse" than the second, and 0 if the two are equal. Here, the definition of "better" can be altered by passing in different comparators as noted below.

First, you need to **fill in the definitions for these comparators**:

- `cmpValue`, which compares the values of the subjects

- `cmpWork`, which compares the workload of the subjects
- `cmpRatio`, which compares the value/work ratios of the subjects

Hint: Use the built-in `cmp()` function to make your job a little easier. However, be mindful of what the `cmp()` function returns vs. what we want your function to return.

The comparators work to help choose the "best" subject at any step of the algorithm. For instance, if we're given the subject list `subs` and a maximum of 15 hours of work:

```
>>> subs = [Subject('6.00',  16, 8, 0),
            Subject('1.00',  7,  7, 0),
            Subject('6.01',  5,  3, 0),
            Subject('6.034', 9,  6, 0)]
```

We can use `greedyAdvisor` with the value, work, and ratio comparators to get different course loads.

- The value comparator first finds 6.00 and then 6.034. (more value is better)
- The work comparator first finds 6.01 and then finds 6.034. (less work is better)
- The ratio comparator chooses 6.00 and then 6.01. (higher ratio is better)

Below is the example output:

```
>>> printSubjects(GreedyAdvisor(cmpValue).pickSubjects(subs, 15, 1))
Course  Value   Work    Lottery
======  =====   ====    ====
6.00    16      8       0
6.034    9      6       0

Number of subjects: 2
Total value: 25
Total work: 14
Total number of lottery subjects: 0

>>> printSubjects(GreedyAdvisor(cmpWork).pickSubjects(subs, 15, 1))
Course  Value   Work    Lottery
======  =====   ====    ====
6.01    5       3       0
6.034   9       6       0

Number of subjects: 2
Total value: 14
Total work: 9
Total number of lottery subjects: 0

>>> printSubjects(GreedyAdvisor(cmpRatio).pickSubjects(subs, 15, 1))
Course  Value   Work    Lottery
======  =====   ====    ====
6.00    16      8       0
6.01    5       3       0

Number of subjects: 2
Total value: 21
Total work: 11
Total number of lottery subjects: 0
```

Fill in the unimplemented methods in GreedyAdvisor.

Note: For any given input, there is not always a single (i.e., unique) right answer for the list of subjects. For instance, multiple subjects may have the same value and/or work and/or value to work ratio.

Hint: We suggest that you first use the `subjects_small.txt` to test your implementation. In fact, you should be able to manually figure out what the answer should be given a particular value for `maxWork`. Be sure you test on the small case before moving on to the complete subject list! Also, don't forget to first sort the subjects based on the comparator. The `sorted()` function allows you to pass in the comparator you want to use and returns a copy of the input list that is sorted.

Problem 3: Subject Selection using Brute Force

As we've learned in lecture and in problem set 8, a greedy algorithm does not always lead to the globally optimal solution. One approach to finding the globally optimal solution is to use brute force to enumerate all possible solutions and choose the one yielding the best value while satisfying the given constraints.

For this problem, **fill in the implementation of the `BruteForceAdvisor` class**.
The `pickSubjects` method in `BruteForceAdvisor` should return the globally optimal list of subjects such that the list has the most value while not exceeding `maxWork` and `maxLottery`. **You should not use the `getPartitions` algorithm from pset 8.** The algorithm you use for `pickSubjects` should be recursive with pruning of the infeasible branches using backtracking, similar to the examples shown in lecture. We recommend you make a helper function inside the class `BruteForceAdvisor`.

Hint: Don't try to run your brute force algorithm on the entire `subjects.txt` list, because it will take a really long time! Make sure to test it on a smaller selection of subjects. Again, you might want to try creating a small subject list first and solve the problem manually and check your answers before proceeding to larger subject lists.

Problem 4: Subject Selection using Memoization

We saw in lecture that memoization can be used to reduce an exponential-time optimization algorithm to a pseudo-polynomial-time algorithm. This is done using memoization by breaking up the original problem into sub-problems and **memoizing** the solutions to the sub-problems. This results in a speed-up because the solutions to sub-problems can then be used multiple times without being re-computed!

Implement `MemoizingAdvisor` in `ps9.py` . The `pickSubjects` method of `MemoizingAdvisor` should return a subject list which is globally optimal just as brute force, but has a much faster implementation by using this memoization method.

Hint: Can you reduce this problem to another smaller problem we've encountered earlier that can be solved by memoization?

Problem 5: Performance Comparison

We want to measure the performance of the various advisors. **Fill in the `measureTimes` function**, which should read in the list of subjects, and then run each of the advisors for each of a series of subject list sizes as passed in, using a random sample for each subject

list size for a fixed number of times, and measure the average amount of time taken for each subject list size.

For the purpose of this problem, plot the different advisor curves on the same plot to compare their performance by calling `measureTimes` with `subjectSizes` being the list `[10, 20, 30, 40, 50]`, `maxWork` set to 40, `maxLottery` set to 10 and `numRuns` set to 5 (it might take some time to finish all the runs, so make sure you get it to work on smaller samples first before generating the final plot). Make sure you choose a new random sample of subjects for each run. Use `cmpRatio` as the comparator for the greedy advisor. What trend do you observe among the three advisors? How does the time taken to pick subjects grow as the number of subject used increases? Why do you think that is the case? **Include your answer in the write-up along with the plot.**

Do not generate the plot every time your code is run since it takes a while to run. That is, running your ps9.py file should not output anything. We should be able to generate your plot by calling the `measureTimes` function you wrote. Also, make your plot look nice with titles, labels, and a legend. Considering that we're using algorithms that sometimes have logarithmic or exponential growth, you may find it useful to visualize your plot with a log scale on the y-axis.

Hint: You might want to check out `random.sample` function in python.

Hand-In Procedure

1. **Save.**
   All of your code should be in a single file named `ps9.py`. Submit your plot and answer to the last problem in `ps9.pdf`.

2. **Time and collaboration info.**
   At the start of the file, in a comment, write down the number of hours (roughly) you spend on this problem set, and the names of whomever you collaborated with. For example:

```
# Problem Set 9
# Name: Jane Lee
# Collaborators: John Doe
# Time: 4:30
... your code goes here ...
```

3. **Sanity checks.**
   After you are done with the problem set, do these sanity checks:
   - Test your `loadSubjects` and make sure it can load the supplied `subjects.txt`.
   - Run each function and make sure they return what you think they do.
   - Make sure that simply running your code (i.e. Run->Run Module or F5) does not cause any output.