Problem Set 9: Schedule Optimization

Handed out: Lecture 18
Due: 11:59pm, Lecture 20.

Introduction

At an institute of higher education that shall be 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 wants to do, the program returns a list of subjects that maximizes the amount of value.

The goal of this problem set is to implement optimization algorithms.

You should submit 2 files for this problem set: your code in **`ps9.py`**, and a write-up in pdf format called **`ps9_writeup.pdf`**.

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.

Getting Started

Download and save

Download the files in the zip folder containing all the files you need, including:

- `ps9.py`, the skeleton you'll fill in for Problems 1-5.
- `ps9_test.py`, some code to help you test your solution.
- `subjects.txt`, the list of subjects with value and work information.
- `shortened_subjects.txt`, a shortened version of `subjects.txt`.

Problem 1: Building A Subject Dictionary

The first step is to implement `loadSubjects` in `ps9.py`, which loads a list of subjects from a file. Each line of the file contains a string of the form "name,value,work", where the name is a string, the value is an integer indicating how much a student learns by taking the subject, and the work is an integer indicating the number of hours a student must spend to pass the subject. `loadSubjects` should return a dictionary mapping subject names to tuples, where the tuples are pairs of (learning value, work hours) integers.

```
def loadSubjects(filename):
    """
    Returns a dictionary mapping subject name to (value, work),
```

```
    where the name is a string and the value and work are integers.
    The subject information is read from the file named by the string
    filename. Each line of the file contains a string of the form
    "name,value,work".

    returns: dictionary mapping subject name to (value, work)
    """
    # The following sample code reads lines from the specified file
    # and prints each one.
    inputFile = open(filename)
    for line in inputFile:
        print line

    # TODO: Instead of printing each line, modify the above to parse
    # the name, value, and work of each subject and create a
    # dictionary mapping the name to the (value, work).
```

You can think of the dictionary returned by `loadSubjects` as a representation of the full subject catalog. In subsequent problems, we will use dictionaries of the same structure to represent subject selections (subsets of the full subject catalog).

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

```
>>> subjects = loadSubjects(SUBJECT_FILENAME)
>>> printSubjects(subjects)
Course  Value   Work
======  ====    =====
10.00   1       20
10.01   2       20
10.02   1       16
10.03   6       19
10.04   3       13
10.15   8       13
[... TRUNCATED ...]
9.16    9       10
9.17    9       11
9.18    7       16
9.19    9       9

Total Value:    1804
Total Work:     3442
```

You may want to consider running your code on the smaller `shortened_subjects.txt` files so that you have a simpler, more manageable set of subjects on which to test your functions.

Problem 2: Subject Selection By Greedy Optimization

The Institute hired you to implement the program. They asked you to implement a greedy method (`greedyAdvisor`) to formulate a list of subjects that satisfies each student's constraint (the amount of work student is willing to do).

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 two argument — each of which is a (value, work) tuple — and returns a boolean indicating whether the first argument is "better" than the second. Here, the definition of "better" can be altered by passing in different comparators.

Implement the three comparators in `ps9.py`:

- `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

Each comparator takes in two `(value, work)` tuples and returns `True` if the first tuple is better than the second tuple, and `False` otherwise. Each comparator has its own definition of "better." Be sure to read and satisfy the specifications of each comparator.

For instance, if we're given the following subject dictionary, `smallCatalog`, and a maximum of 15 hours of work:

```
# name     value  work
{'6.00':   (16,   8),
'1.00':    (7,    7),
'6.01':    (5,    3),
'15.01':   (9,    6)}
```

If we were to use `greedyAdvisor` with the value comparator (look below to see what the function's arguments are):

```
>>> greedyAdvisor(smallCatalog, 15, cmpValue)
```

your function will use the comparator and select 6.00 first, then 15.01, and return the following dictionary:

```
{'6.00': (16, 8), '15.01': (9, 6)}
```

The other subjects are not included because they would bring the total workload above the `maxWork` limit.

If we were to use the work comparator:

```
>>> greedyAdvisor(smallCatalog, 15, cmpWork)
```

your function will select 6.01 followed by 15.01 and return:

```
{'6.01': (5, 3), '15.01': (9, 6)}
```

If we were to use the ratio comparator, your function would return:

```
>>> greedyAdvisor(smallCatalog, 15, cmpRatio)
```

your function will select 6.00 followed by 6.01 and return:

```
{'6.00': (16, 8), '6.01': (5, 3)}
```

Again, `printSubjects` can be used to neatly format the dictionary returned by `greedyAdvisor`:

```
>>> selected = greedyAdvisor(smallCatalog, 15, cmpRatio)
>>> printSubjects(selected)
Course  Value   Work
======  ====    =====
6.00    16      8
6.01    5       3

Total Value:    21
Total Work:     11
```

Implement `greedyAdvisor` in `ps9.py`:

```
def greedyAdvisor(subjects, maxWork, comparator):
    """
    Returns a dictionary mapping subject name to (value, work)
    which includes subjects selected by the algorithm, such
    that the total work of subjects in the dictionary is not
    greater than maxWork.  The subjects are chosen using
    a greedy algorithm.  The subjects dictionary should not
    be mutated.

    subjects: dictionary mapping subject name to (value, work)
    maxWork: int >= 0
    comparator: function taking two tuples and returning a bool
    returns: dictionary mapping subject name to (value, work)
    """
    # TODO...
```

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

```
>>> subjects = loadSubjects("shortened_subjects.txt")
>>> print greedyAdvisor(subjects, 7, cmpWork)
```

may output a few different results, such as

```
{'6.00': (10, 1), '6.17': (9, 3), '6.04': (1, 2)}
```

or

```
{'6.00': (10, 1), '6.12': (6, 3), '6.04': (1, 2)}
```

The ordering of your output doesn't matter, *as long as your results match the constraints* – so an output of `{'6.00':(10,1), '6.04': (1,2)}` for the above query wouldn't be right, as it is still possible to add another subject with the constraint of `maxWork = 7`.Finally, be sure to test your greedy algorithm on the full subject data from the `subjects.txt` file.

Problem 3: Subject Selection By Brute Force

As we learned in lecture, 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.

Implement `bruteForceAdvisor` in `ps9.py`:

```
def bruteForceAdvisor(subjects, maxWork):
    """
    Returns a dictionary mapping subject name to (value, work), which
    represents the globally optimal selection of subjects using a brute force
    algorithm.

    subjects: dictionary mapping subject name to (value, work)
    maxWork: int >= 0
    returns: dictionary mapping subject name to (value, work)
    """
    # TODO...
```

For this problem, it may be useful to define and call some helper functions.

Do not worry about efficiency. This is a brute force algorithm after all. In fact, you will find that running `bruteForceAdvisor` on the full subject data may take an extremely long time, or even cause an `OverflowError`, depending on your implementation. You can test your code on `shortened_subjects.txt` and small values of `maxWork`:

```
>>> subjects = loadSubjects("shortened_subjects.txt")
>>> print bruteForceAdvisor(subjects, 3)
{'6.00': (10, 1), '6.04': (1, 2)}
>>> print bruteForceAdvisor(subjects, 4)
{'6.00': (10, 1), '6.17': (9, 3)}
>>> print bruteForceAdvisor(subjects, 5)
{'6.00': (10, 1), '6.18': (10, 4)}
>>> print bruteForceAdvisor(subjects, 6)
{'6.00': (10, 1), '6.18': (10, 4)}
>>> print bruteForceAdvisor(subjects, 7)
{'6.00': (10, 1), '6.12': (6, 3), '6.17': (9, 3)}
```

Notice how the brute force algorithm returns better results than any of the greedy algorithms:

```
>>> subjects = loadSubjects("shortened_subjects.txt")
>>> print greedyAdvisor(subjects, 7, cmpValue)
{'6.00': (10, 1), '6.18': (10, 4), '6.04': (1, 2)}
```

```
>>> print greedyAdvisor(subjects, 7, cmpWork)
{'6.00': (10, 1), '6.17': (9, 3), '6.04': (1, 2)}
>>> print greedyAdvisor(subjects, 5, cmpRatio)
{'6.00': (10, 1), '6.17': (9, 3)}
```

However, a brute force algorithm takes a long time! Later in the course, we will discuss dynamic programming, which is a way we can speed up these algorithms.

Problem 4: How efficient are these algorithms?

Answer the following questions in the file `ps9_writeup.pdf`:

1. What is the algorithmic complexity of `greedyAdvisor?`
2. What is the algorithmic complexity of `bruteForceAdvisor?`
3. Assuming 1 microsecond (1 microsecond = 1×10^-6 seconds) to compute the value of a solution in bruteForceAdvisor, how much time in years (365 days per year) would it take for bruteForceAdvisor to find an optimal solution for the following number of subjects:
    1. 8 subjects?
    2. 16 subjects?
    3. 32 subjects?
    4. 321 subjects (the number of courses in the problem set)?

Hand-In Procedure

1. Save

Save your solutions as **`ps9.py`**. Your writeup should be called **`ps9_writeup.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 9
# Name: Jane Lee
# Collaborators: John Doe
# Time: 3:30
#
... your code goes here ...
```

3. Sanity checks

After you are done with the problem set, do these sanity checks:

- Run the `ps9.py` file, and make sure it can be run without errors.
- Test your `loadSubjects` and make sure it can load the supplied `subjects.txt`.

- Run each function you've implemented and make sure they return what you think they do.

Also, make sure that your writeup contains everything we've asked for.

6.00SC Introduction to Computer Science and Programming
Spring 2011