6.00 Introduction to Computer Science and Programming
Fall 2008

6.00: Introduction to Computer Science and Programming

Problem Set 8: Dynamic Programming

Handed out: Tuesday, October 21, 2008
Due: Thursday, October 30, 2008

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 a dynamic programming algorithm and learn how to pass functions as arguments.

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 syllabus.

Getting Started

Download and save these files into the same folder.

- ps8.py: the skeleton you'll fill in for Problems 1–5
- subjects.txt: the list of subjects with value and work information.

Problem 1: Building A Subject Dictionary

The first step is to implement `loadSubjects` in `ps8.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 prin
```

```
        # 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).
```

Hint: You may find `split()` and `strip()` methods of strings useful when implementing this. See their documentation at the online Python Library Reference. In particular, you may want to use `split()` to split at commas and `strip()` to remove the line break at the end of each line.

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 writing your own, smaller `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 arguments—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.

We've provided three comparators for you to pass in:

- `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

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. Note: subject names are sorted as strings prior to printing, so that the ordering is unique and guaranteed.

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 `ps8.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. Also, 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.

We have provided an implementation of the brute force algorithm in the function `bruteForceAdvisor` in `ps8.py`. This function returns a dictionary mapping subject name to (value, work), which represents the globally optimal selection of subjects using a brute force algorithm.

```
    def bruteForceTime():
        """
        Runs tests on bruteForceAdvisor and measures the time required to compute
        an answer.
        """
        # TODO...
```

Using the `time` function introduced in Problem Set 6, see how long this takes for different values of `maxWork`. Place your code in the `bruteForceTime` function. What's the largest `maxWork` you can pass in before the brute force algorithm takes an "unreasonable" amount of time? (You can decide what constitutes an unreasonable amount of time.) Write your observations in the comments underneath your `bruteForceTime` code.

Problem 4: Subject Selection By Dynamic Programming

We saw in lecture that dynamic programming can be used to reduce an exponential-time optimization algorithm to a pseudo-polynomial-time algorithm. This is done by breaking up the original problem into sub-problems and *memoizing* the intermediate solutions to the sub-problems. This results in a speed-up because the solutions to sub-problems are used multiple times, and memoization avoids re-computing these solutions.

In this problem, you will use dynamic programming to find the globally optimal subject selection.

Implement `dpAdvisor` in `ps8.py`, which returns a dictionary mapping subject name to (value, work) found by the dynamic programming method.

```
    def dpAdvisor(subjects, maxWork):
        """
        Returns a dictionary mapping subject name to (value, work) that contains a
        set of subjects that provides the maximum value without exceeding maxWork.

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

Hint: Can you reduce this problem to another problem we've encountered which can be solved by dynamic programming?

Here is a sample output, using the subject catalog that was loaded as shown in the example in Problem 1:

```
>>> printSubjects(dpAdvisor(subjects, 30))
Course  Value   Work
======  ====    =====
12.04   7       1
12.09   8       2
```

```
14.02   10      2
15.01   7       1
18.08   10      3
2.03    6       1
22.01   6       2
22.03   10      2
22.05   6       2
22.06   10      3
24.12   6       1
6.00    10      1
7.00    7       1
7.05    8       2
7.06    4       1
7.16    7       1
7.17    10      1
7.18    10      2
8.08    4       1

Total Value:    146
Total Work:     30
```

Problem 5: Performance Comparison

Measure the performance of `dpAdvisor` as you did in Problem 3. How does the performance of the dynamic programming algorithm compare to that of the brute force algorithm? Place your timing code in `dpTime` and your observations in comments below `dpTime`.

```
    def dpTime():
        """
        Runs tests on dpAdvisor and measures the time required to compute an
        answer.
        """
        # TODO...
```

Hand-In Procedure

1. **Save.** All your code should be in a single file called `ps8.py`.
2. **Time and collaboration info.** At the start of the file, in a comment, write down the number of hours (roughly) you spent on this problem set, and the names of whomever you collaborated with. For example:

    ```
    # Problem Set 8
    # Name: Jane Lee
    # Collaborators: John Doe
    # Time: 1:30

    ... your code goes here ...
    ```

3. **Sanity checks.** After you are done with the problem set, do these sanity checks:

    - Run the `ps8.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 implemented in Problems 2 and 4 and make sure they return what you think they do.