HW6: Dynamic Programming (Coding)

Objectives:
To practice designing and implementing algorithms using dynamic programming.

Due date: Monday, February 19th by 11:00pm.

Materials: starter code (hw6-dynamic-programming.tar.gz)

What to turn in:

- Your final version of `OptimalLineup.java` and any other Java files that you create or change.
- A PDF file containing your answers to the questions from problems 1 and 3 below.

How to submit:
Use the canvas page (https://canvas.uw.edu/courses/1124392/assignments/4091333) to submit your solution.

Overview

In this assignment, you will design and implement a dynamic programming algorithm to compute optimal fantasy football lineups subject to a salary cap constraint. In particular, you must find the subset of players whose total expected points is as large as possible but subject to the constraint that the total price of all the players is no more than $60,000.

So far this is simply a knapsack problem, where the value of an individual player is their expected points (a number given in the input) and the weight of an individual player is their price. However, there are addition constraints that limit the subsets of players you are allowed to use.

Each player has a position, which is one of the set {QB, RB, WR, TE, K, DEF}. The additional constraints are on the number of players you can include from each position. In particular, you must choose only one player at each of the QB, TE, K, and DEF positions; you must choose two players at the RB position; and you must choose three players at the WR position. (As a consequence of these constraints, every valid solution chooses exactly 9 players in total.)

Each player also plays for a particular team. In the final part of the assignment, we will consider an additional constraint that restricts you from choosing certain players that play on the same team if they play particular positions.

Provided Code

Much of the code needed for the program we are making in this assignment is provided for you. In particular, you are already given code for parsing the data files, parsing the command-line arguments, and for keeping track of information about players. You will only need to fill in the bodies of one method that implements the core logic of the algorithm.

You will need to familiarize yourself with three files in the provided code: `OptimalLineup.java`, `Position.java`, and `Player.java`. While the other code will be used by your program, you will not need to use it directly.

`OptimalLineup.``java` contains the entry point for the program. The `main` method parses the command-line arguments and loads the data from the passed-in data file. `OptimalLineup` also contains the one (or two) method bodies that **you will fill in** during the assignment. (Search for "TODO" in the file to find them.)

`Position.``java` is a simple enum class that represents the possible player postions: QB, RB, WR, TE, K, and DEF. The static constant `Position.QB`, for example, is an object representing the QB position, and each of the other positions has a similarly named static constant.

`Player.``java` is a helper class that stores information about a particular player, including their name, position, team, price, and expected points. The following methods will be likely be useful to you:

- `int` **`getPrice`**`()`
  Returns the price of the player.
- `int` **`getPointsExpected`**`()`
  Returns the expected points to be scored by the player.
- `int` **`atPosition`**`(Position p)`
  Returns true iff the player's position matches the given one. You can also call `player.getPosition()` to return the `Position` object representing the player's position, but if you simply want to check whether their position is, say, QB, you can do so more simply by calling `player.atPosition(Position.QB)`.

Problems

1. As with any dynamic programming problem, the key element is to find a set of sub-problems whose solutions, if given to you, would allow you to solve the whole problem.
   a. To find the optimal substructure, the usual approach is to consider how the optimal solution could use the last element. In this case, the elements correspond to players. Assuming that the last player's position is WR, what sub-problem solutions, if given to you, would allow you to solve the problem described in the overview? (Hint: these sub-problems need not have the same set of constraints as the original problem.)
   b. Describe briefly how you would do the same for any other position. (Try to describe this in a way that will work for any of the positions. If you cannot, then describe how to do this for two other positions.)
   c. Suppose that you applied the above approach recursively to each sub-problem mentioned above. Describe the complete set of sub-problems you would need to solve. How many sub-problems are there in total?
2. Implement the dynamic programming algorithm that results from your analysis above in the method `OptimalLineup.findOptimalLineup`. You may do so either iteratively (by filling in the table of sub-problems) or recursively (using memoization, as described in the textbook).

Hint: Whether you implement the dynamic programming algorithm iteratively or recursively, you will need a way to *uniquely* identify each sub-problem. If you implement the method iteratively, then you probably want to map each sub-problem to a unique row index for where the value of the optimal solution will be stored in the table. (The column index, on the other hand, will be the index of the last player that can included in the solution.) If you implement the method recursively, then you will want to map each sub-problem to a unique key that you can use to record the optimal value in the memoization table. Mapping sub-problems to unique indexes or keys could easily be 50% of the code you end up writing.

Your solution must finish within **ten seconds** to get full credit. (Slower solutions that are still correct will receive a small point deduction.)

3. In this problem, we will add an additional constraint, namely, that you cannot have two players from the same team whose positions are QB and WR, QB and K, or K and DEF. The points scored by those players end up being more more highly correlated, so including them both in the lineup increases the standard deviation of points more than less-correlated players, and it is often desirable to keep the standard deviation of points smaller if possible.

    a. Describe how to find the optimal lineup with no highly correlated players using your algorithm from above as a sub-routine. You may need to invoke that algorithm many times, but your solution **must** have the property that, if the optimal solution to the original problem (without this new constraint) does not include any highly correlated players, then you invoke the above algorithm **only once** so that there is no slow down in running time.

    b. **Extra Credit**: Implement the algorithm that results from your analysis above in the method `OptimalLineup.findOptimalLineupWithoutHighCorrelations`. You are already provided with a method `OptimalLineup.getHighCorrelations`, which returns the set of highly correlated players in a given lineup.

Testing

To make testing your solution easier, here are the solutions for the sample data files you are given:

week	expected points
1	125.1
2	125.3
3	127.6

Note that, if you have not yet fixed the bug that was previously on line 33 of `OptimalLineup.java` (changing true to false), then you would see these answers instead:

week	expected points
1	122.6

2	125.1
3	124.9

While your implementation must produce an actual lineup that achieves these expected points, **any lineup** that does so is acceptable. (That is why a particular lineup is not given here.)