HW9: Branch & Bound (Coding)

Objectives:
To practice implementing algorithms using branch and bound.

Due date: Friday, March 9th by 11:00pm.

Materials: starter code (hw9-branch-and-bound.tar.gz)

What to turn in:

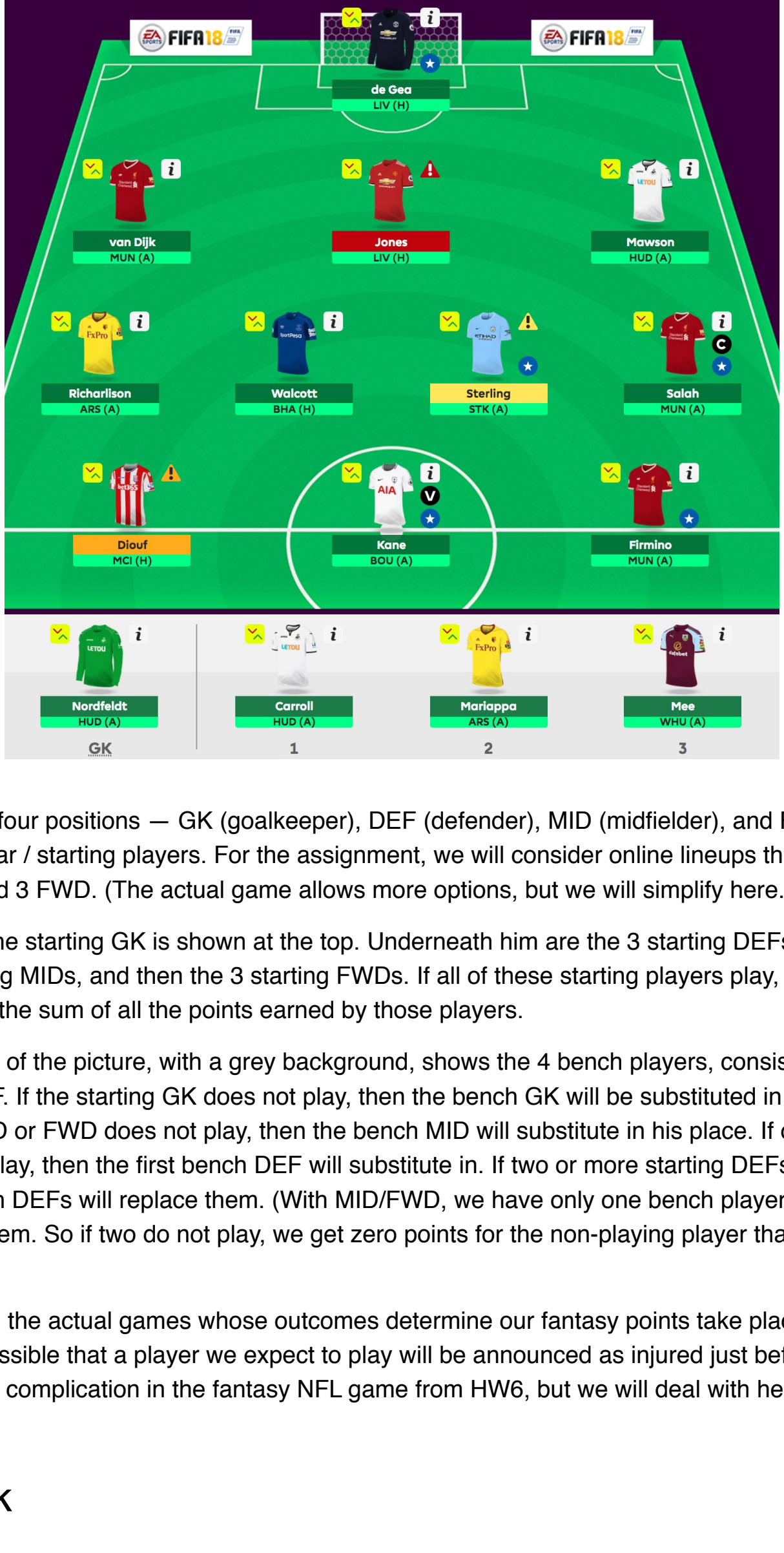
- Your final version of `OptimalLineup.java` and any other Java files that you create or change.
- A PDF file containing your answer for problem 2 (and optionally problem 4).

How to submit:
Use the canvas page (https://canvas.uw.edu/courses/1124392/assignments/4116048) to submit your solution.

Overview

In this assignment, you will write code to implement a branch and bound algorithm for finding optimal fantasy soccer lineups. As in HW6, these are subject to a budget constraint and position contraints. However, the problem becomes substantially more difficult, here, as we are also required to include "bench players" to the lineup that only contribute points if one of the regular players does not play.

Here is an example lineup (my own lineup for this week, which is... not great):



This game has four positions — GK (goalkeeper), DEF (defender), MID (midfielder), and FWD (forward) — and 11 regular / starting players. For the assignment, we will consider online lineups that start 1 GK, 3 DEF, 4 MID, and 3 FWD. (The actual game allows more options, but we will simplify here.)

In the picture, the starting GK is shown at the top. Underneath him are the 3 starting DEFs. Below that are the 4 starting MIDs, and then the 3 starting FWDs. If all of these starting players play, then our points for the game is the sum of all the points earned by those players.

The bottom row of the picture, with a grey background, shows the 4 bench players, consisting of 1 GK, 1 MID, and 2 DEF. If the starting GK does not play, then the bench GK will be substituted in his place. If *any* starting MID or FWD does not play, then the bench MID will substitute in his place. If one starting DEF does not play, then the first bench DEF will substitute in. If two or more starting DEFs do not play, then both bench DEFs will replace them. (With MID/FWD, we have only one bench player that can substitute for them. So if two do not play, we get zero points for the non-playing player that is not replaced.)

Like in the NFL, the actual games whose outcomes determine our fantasy points take places over 2-3 days, so it is possible that a player we expect to play will be announced as injured just before the game. We ignored this complication in the fantasy NFL game from HW6, but we will deal with here for fantasy soccer.

Your Task

You will implement an algorithm that computes the optimal lineup given projected points for each player along with estimated probabilities that each of those players *will play* that week.

As noted above, the introduction of bench players makes this problem substantially more complicated. We were able to solve the problem in HW6 very quickly using dynamic programming because there is limited dependence between the choices available to us when considering two disjoint lists of players, X and Y: in order to know if we can use subset x of X together with subset y of Y, we only need to know the total cost of the players in x and in y (to make sure we don't go over the budget limit) and the number of players from each position in x and in y (to make sure we satisfy the position constraints). In other words, any two subsets x and x' of X that have the same cost and the same set of positions can be combined with the exactly the same subsets y of Y, so can simply keep track of the best one of these subsets of X.

With bench players, there is more dependence between the choices made. In particular, the expected points added by the bench MID depends not only on how many points that player is expected to score but also on the probabilities that each starting MID or FWD will play. Swapping out one of the starting MIDs, for example, with another one expected to score more points could actually decrease the expected points of the total lineup if they reduce the odds that the bench player will be able to play.

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 reading and writing the data files, parsing the command-line arguments, and even computing the lower bounds that you will need for the branch and bound algorithm. You will only need to fill in the bodies of a couple of methods that implement the remainder of the algorithm.

You will mainly need to familiarize yourself with just `OptimalLineup.java`. That is the only file in which you are required to modify the code. However, you will also want to have a brief look at `Player.java`, `Lineup.java`, and `LineupBound.java`, each of which of which you will use from your code.

`Player.``java` is a helper class that stores information about an individual player, including their name, position, price, expected points, and estimated probability to play. This information is read from the input data files by `OptimalLineup`.

`Lineup.``java` is a helper class that stores a list of players representing a complete or partially complete lineup. Note that **the order in which the players appear in the list is significant**: in particular, the second GK, the fourth and fifth DEF, and the fifth MID in the list (if any) are the bench players. `Lineup` provides the following methods that will be useful to you:

- **`Lineup`**`(budget, teamLimit)`
  Creates an empty lineup that will enforce the given budget and team-limit constraints. The latter enforces than no more than `teamLimit` players from the same team are included. The actual game limits to 3 players from one team, but we will mostly ignore that constraint. You turn off this constraint by setting `teamLimit` to 15.

- **`canAdd`**`(player)`
  Determines whether the given player can be added without violating the budget, position-limit, or team-limit constraints.

- **`add`**`(player)`
  Adds the given player at the **end** of the list.

- **`getTotalPrice`**`()`
  Returns the sum of the prices of all the players in the lineup.

- **`getExpectedPoints`**`()`
  Returns the expected points from all of the players included the lineup, **including any bench players**. For example, it includes the expected points from the fifth MID added (the bench MID) times the probability that some other MID of FWD will not play.

- **`clone`**`()`
  Returns a fresh copy of the lineup. This is useful if you want to save a copy of a lineup that might be modified in the future.

`LineupBound.``java` is a helper class that computes a lower bound on the points that could be achieved by any completion of a given lineup. (It does so by dynamic programming, using a similar approach to HW6 but modified to account for the probabilities that players will actually play.) It has only two public methods:

- **`LineupBound`**`(partialLineup, minPlayProb, numNeededGK, numNeededDEF, numNeededMID, numNeededFWD)`
  Prepares the class to compute an upper bound on the points that can be achieved by adding the given number of players at each of the positions to the given lineup, while staying within the budget limit. (Always set `minPlayProb` to 0 for this assignment.)

- **`maxPoints`**`(players)`
  Computes an upper bound on the points that can be achieved by adding the required number of players at each position (as specified in the constructor) by choosing from those players passed in and staying within the budget limit.

`OptimalLineup.``java` is the entry point for the program, and it contains the methods that you will modify in the problems below. The most important static method is the following:

- **`static`** `void` **`main`**`(String[] args)`
  Reads the player information from the passed in data file (the first argument). It then creates an instance of `OptimalLineup` and calls its `compute` method to find the optimal lineup, which is then printed.

Unlike in some previous assignments, most of the work is not done in static methods. This is because our branch and bound algorithm needs to keep track of the best lineup found, and it does so in the private fields of the class. Here are the important fields to know:

- `int` **`budget`**
  Records the budget limit for lineups (as passed in to the `OptimalLineup` constructor).

- `List<Player>` **`allPlayers`**
  Records the list of available players (as passed in to the `OptimalLineup` constructor).

- `List<Player>` **`attackers`**
  Records a list of just the FWD/MID players, sorted in decreasing order of expected points.

- `float` **`bestPoints`**
  Records the best point total of any lineup considered so far (or -infinity if none has yet been found).

- `Lineup` **`bestLineup`**
  Records the lineup that achieved the point total of `bestPoints` (or null if `bestPoints` is still -infinity).

- `float[]` **`bestPointsDefense`**
  An array storing the maximum points achievable for any budget. You will help compute this in problem 1.

- `Player[][]` **`bestPlayersDefense`**
  An array storing the players that achieved the maximum point total in the corresponding entry of `bestPointsDefense`. You will help compute this in problem 1.

Here are the important methods of `OptimalLineup`:

- `Lineup` **`compute()`**
  Returns the optimal lineup using the list of players and budget passed to the constructor. It does so by first calling `computeBestDefenseByPrice` (below) to find the optimal choice of defensive players (GK and DEF) for *every possible budget*. It then calls `tryAttackingLineups` to search through possible lineups, but that method only needs to try the choice of attacking players (FWD and MID) because it already knows the best way to add defensive players for any given budget.

- `void` **`computeBestDefenseByPrice()`**
  Fills in the `bestPointsDefense` and `bestPlayersDefense` arrays described above by a *brute force search* of all possibilities. (You will implement part of this problem 1.)

- `int` **`tryAttackingLineups`**`(lineup, start)`
  Performs a *branch and bound* search to find the optimal way **extend** the given lineup to a complete one using only players in the `allPlayers` list starting from index `start`. The search is only over the choice of attacking players, however, since the optimal choice of defenders is already known for the budget remaining after all the attacking players are chosen. (You will implement this method in problem 3.)

Problems

1. Implement the missing part of `computeBestDefenseByPrice` in `OptimalLineup.java`. (Look for the "`TODO`" comment in that method.)

   This method already includes code to compute the best choice of GKs for every possible budget. You will do the same thing for DEFs.

   Specifically, write code that tries every possible choice of five defenders. For each choice, try adding all five defenders into a new `Lineup` object (use `teamLimit` of 3 here). If all five can be added, get the total price and expected points by calling the appropriate methods of the `Lineup` object. If the total price is `p`, check whether the expected points for this lineup is higher than the

value currently in `bestPointsDEF[p]`. If it is, replace that value with the expected points of this lineup and store the players of this lineup in `bestPlayersDEF[p]` as well. (In order to do the latter, you need to put the players into an array, not a list.)

Note that you do not need to consider different orders of those defenders. Just add the five in the same order they appear in the list of defenders. (It is possible to prove that it is always best to put the higher scoring players earlier in the lineup, which is how they are ordered in the player list.)

Your code should also count the total number of valid lineups that you built in this process (i.e., ones in which all five DEFs can be added.) Record this number in `countDEF`. The code provided already in `computeBestDefenseByPrice` will print this out. For the file `data/wk5-2017.csv`, you should see a count of around 31M and, for the file `data/wk6-2017`, a count of around 29M.

The next line after your code, in addition to other work, will print out some of the values from the array that you computed. If you implemented it correctly, you should see the following values printed for wk5 and wk6, respectively:

```
*** 216  5.20
*** 217  9.74
*** 218 12.11
*** 219 13.10
*** 220 13.90
*** 221 14.17
*** 223 14.53
*** 224 14.94
*** 225 14.98
*** 229 15.23
*** 230 15.24
*** 231 15.81
*** 232 16.22
*** 233 16.26
*** 235 16.85
*** 240 17.03
*** 253 17.05
```

```
*** 216 10.08
*** 217 13.03
*** 218 14.53
*** 232 14.54
*** 238 14.85
*** 240 15.01
*** 257 15.01
```

The rest of the provided code in `computeBestDefenseByPrice` combines the information in `bestPointsDEF` and in `bestPointsGK` (computed by the provided code) into the array `bestPointsDefense`, mentioned above. This stores the maximum combined points possible from the GK and DEF positions, for each budget, and the players that achieve those points. You will need this information in problem 3 below.

2. Next, you will implement the branch and bound search over attacking players in the method `tryAttackingLineups` described above. Before you do so, answer these questions:

a. The nodes of the search tree will be calls to the `tryAttackingLineups` method. For a call with inputs `lineup` and `start`, what is the set of solutions to consider (the "S" from the lecture slides) per the description of the method above?

b. We will make one recursive call for each choice of next player in the lineup. What is the largest branching factor of any node in this tree?

c. Suppose that we try adding each of the players, from index `i = start` to the end of the `attackers` list, to the end of `lineup` and making a recursive call with the modified lineup and a starting index of `i+1`. Explain why that perfectly splits the solution space between the child nodes. I.e., explain why every potential lineup that we must consider is considered by *exactly one* of these recursive calls.

3. Implement the method `tryAttackingLineups` in `OptimalLineup.java`.

   In problem 2, we described the manner in which to perform the branching. Note that it is possible to use a single `lineup` object for all of these recursive calls. Simply add each new player before making a recursive call and then remove it after the call returns.

   The leaf nodes in the search tree are calls where all 8 attacking players have been filled in. At this point, you can look up the maximum points possible from defensive players for the remaining budget in the array `bestPointsDefense`. (This value will be -infinity if there there is no choice of players available for that budget.) If the points from this lineup plus the maximum possible from defensive players is better than what is currently stored in `bestPoints`, change its value to the expected points of this lineup. Then make a copy of the lineup, add in the defensive players that achieve that number of points, and store it in `bestLineup`.

   So far, we have only described a brute force search. To make it branch and bound, we must add a bounding check. You can get an upper bound on the points achievable by filling in the remainder of the attacking players by using the `LineupBound` class described above. (When you create it, set numNeededGK = 0 and numNeededDEF = 0. We only want it to consider the attacking players.) Call the `maxPoints` method with the sub-list of players still available to be added. The result will be an upper bound for the points of all the attacking players, including those already in the lineup.

   To get a complete bound, we also need to add in the defending players. We know exactly how many points are achievable for each budget from our array `bestPointsDefense`. Unfortunately, we don't know exactly how much money will be available to spend on those players, but we can upper bound it with all of the money not spent on players already in the lineup. We will use the points possible with that budget (which is potentially too large) as an upper bound on the points from the defense.

   Adding those two upper bounds together gives an upper bound on the total points for any completion of this lineup. If that number is not better than the best points found so far (by more than rounding error of 1e-5), then we can skip searching any of those solutions.

   Just as above, we want to get an estimate of how much work was done. In this case, we will count the recursive calls. Return 1 plus the sum of all the numbers returned by any recursive calls (if any). My solution performs about 80K recursive calls for wk5 and 210K recursive calls for wk6. For

a point of comparison, he number of nodes in a brute force search would be more than 600 billion billion.

You can speed up your testing of this method by using alternative data files, `wk5-offense-2017.csv` and `wk6-offense-2017.csv`. These two files remove all but the optimal set of 7 defenders. As a result, the brute force search over defenders will be extremely fast.

If you implemented all of the above correctly correctly, you should see the following results for wk5 and wk6, respectively. The specific players in your solution might be different, but the total expected points should be the same.

```
 GK  Simon Mignolet          LIV  £5.0m  4.67 100%
     Robert Elliot           NEW  £4.0m  3.37 100%
DEF  Ben Davies              TOT  £5.7m  6.39 100%
     Andrew Robertson        LIV  £4.9m  6.19  17%
     Lewis Dunk              BRI  £4.5m  5.35 100%
     Scott Dann              CPL  £4.9m  5.11 100%
     Zanka                   HUD  £4.6m  4.48 100%
MID  Dele Alli               TOT  £9.5m  7.36 100%
     Maxim Choupo-Moting     STO  £5.6m  6.85  90%
     Richarlison             WAT  £6.0m  6.28  90%
     Andre Carrillo          WAT  £5.5m  5.94  75%
     Christian Eriksen       TOT  £9.7m  5.62 100%
FWD  Alvaro Morata           CHE £10.2m  6.76  90%
     Harry Kane              TOT £12.4m  6.49 100%
     Danny Welbeck           ARS  £7.5m  5.27 100%
*** Totals: price £100.0m, expected points 65.4
```

```
 GK  Ederson                 MCI  £5.5m  5.74 100%
     Robert Elliot           NEW  £4.0m  2.86 100%
DEF  Victor Moses            CHE  £6.5m  5.34  92%
     Ben Mee                 BUR  £4.5m  4.87 100%
     Lewis Dunk              BRI  £4.5m  4.84 100%
     Shane Duffy             BRI  £4.5m  4.82 100%
     Mike van der Hoorn      SWA  £4.5m  4.37  56%
MID  Maxim Choupo-Moting     STO  £5.6m  6.17  92%
     Richarlison             WAT  £6.0m  6.11  92%
     David Silva             MCI  £8.3m  5.65 100%
     Kevin De Bruyne         MCI  £9.9m  5.39 100%
     Robbie Brady            BUR  £5.5m  4.94 100%
FWD  Sergio Aguero           MCI £11.5m  7.28  83%
     Wayne Rooney            EVE  £7.5m  5.61 100%
     Romelu Lukaku           MUN £11.7m  5.15 100%
*** Totals: price £100.0m, expected points 60.6
```

4. **Extra Credit**: Find a way to substantially improve the speed of this program (say, by 20% or more). Turn in both the code and a writeup of what changes you made and how much they increased the performance on the two provided input files.

(Note: approximation techniques are on the table but only if they still return optimal solutions for the provided input files.)

5. **Extra Credit**: Implement the method `findOptimalLineup` in `OptimalLineup.java` so that it returns the optimal lineup that satisfies the per-team limit constraint as well. This can be accomplished using the same approach that we added vertex cover constraints on top of knapsack.