**Algorithms**

For part A of my coursework, the only feedback I received was that my pseudocode seemed too much like actual code, and I was advised to use more natural language. I tend to not write pseudocode per se, finding it more beneficial to write out my ideas completely in plain English. I shall try to strike a balance here and provide my explanation for various parts of my code, as well as some pseudocode examples of specific parts.

**Seeding:**

As is the case with many of the algorithms used in my version of the tennis tournament system, my seeding algorithm will be split up between various functions and files. However, the most important elements of seeding will be contained in seasonSpecific.py.

Basic outline:

Seeding will be governed by a matrix contained in a csv file, containing information about how far a player has progressed in any tournament of a season, as outlined below (each gender kept separately):

Torn1 Torn2 Torn3 Torn4

MP03 1 1 1 1

MP04 1 1 1 1

MP05 1 1 1 1

As the season progresses, a list of current winners will be compared with a list of all players.

If a player wins a round, the relevant matrix position will be incremented by 1. So, in the above example, if MP03 won a first round match in the first tournament, and reached the third round in the third tournament but lost the third-round match, their seeding in the next season would be based on the following:

Torn1 Torn2 Torn3 Torn4

MP03 2 1 3 1

MP04 1 1 1 1

MP05 1 1 1 1

As the first season will have no seeding, the seeding algorithm will only be concerned with the collection of data. In subsequent seasons, the algorithm will have to both collect and set seeding information.:

IF first season:

FOR EACH tournament:

IF winner of match in round:

Increment matrix position by one in Original matrix file

ELSE IF NOT first season:

MAKE COPY of Original matrix

RESET Original to default

FOR EACH tournament:

IF winner of match in round:

Increment Original matrix position by one

FOR EACH tournament:

FOR FIRST round:

COMPARE all players with COPY matrix: (i.e previous season’s matrix)

IF number in COPY matrix for player > current round:

Player = seed.

FOR SUBSEQUENT ROUNDS

Ordered List of winners, first half = seeds

Divide list into seeded winners and unseeded winners

Following this, all seeded players can only play other seeded players in each round, bar the final. This is achieved by comparing a list of currently seeded players, with a list of all players, and essentially separating the players into two separate lists:

IF NOT first season

FOR all players (according to gender)

If a player is not in the list of seeded players

Player = non-seed

ELSE IF first

Continue as normal. i.e. pairing players at random and assigning a random score.

During ranking, the seed matrix will once again be used to determine whether difficulty points will be used to multiply the overall rank points. If a player did not achieve at least the same position in the previous tournament in the previous year, the difficulty multiplier removed This should be easily achieved by the following:

IF relevant integer in player COPY seed matrix < number of current round in tournament

difficultyPoints = 1

ELSE

difficultyPoints = relevant multiplier

Setting difficulty = zero would result in all rank points being zero, as anything multiplied by zero is equal to zero. Setting difficulty points = one has the advantage of simply removing the difficulty multiplier, as anything multiplied by one is its self.

**Seasons and Tournaments:**

I view the addition of seasons in part B as an addition of an outer loop; an extra layer of iteration. I seek to keep seasons as contained as tournaments currently are, and to this end have devised a system whereby sets of seasons are contained within their own ‘worlds’. These worlds are separate files, generated by the program. Within each new file generated (named by the user), are two ‘state files’.

The file ‘playerStates’ contains all tournament wide and season wide information about players, and houses a number of temporary files used throughout the program to save information needed between tournaments and seasons.

The file ‘states’ contains two files that keep track of the state of the system, recording whether or not a round has been completed (for tournaments), or a tournament has been completed (for seasons), and importantly how many seasons have been played, so that the program knows when to begin using seeding.

At the beginning of each round, a Boolean value is changed in the file called PREVIOUS\_TORNEMENT\_COMPLETE\_CHECK. If this round is successfully completed, this Boolean value is switched back to its original state, and a value incremented, indicating which round has just been played. Thus, if the system is stopped part way through a round, the tournament can restart from this point.

Once a tournament has been completed, in cannot be re-played until the next season. A function dealing with this relies on SEASON\_COMPLETE\_CHECK. Boolean values are again used for this. Every season all overall rank points and winnings are reset. Statistics, which I will explain later, are not, and represent a player’s progress in their ‘world’.

**Erroneous scores and player withdrawal**

My previous implementation did not allow for errors to be encountered at all. My current submission will deal with this by passing the problem through relevant different functions, representing different aspects of the problem. The method will be spread out, but It will be dealt as follows, using male scores an example:

IF playerA score == player B score OR no maximum score entered OR score is > max

ASK user if a player has withdrawn

IF TRUE

ASK user if withdrawing player is Player A?

IF TRUE:

Player A score = 2

Player B score = 3

ELSE:

Player B score = 2

Player A score = 3

ELSE repeat the step below until satisfactory score met.

User input score, with limitations on maximum, minimum, and no draws possible.

**Score margin algorithm**

Score margins will be recorded in a one dimensional array, which is later compared in parallel, with a list of current winners. This is can be achieved at the point of score processing, by subtracting the losing score from the winning score.

A win by 3 sets for men multiplies the score by 2.5, and a win by 2 sets for men multiples the score by 1.5. These multipliers are both 0.5 less than the corresponding score margin, so a score modifier can be set as:

Score Mod = Win margin – 0.5.

As women only have one multiplier, which is 0.5 higher than the 2 set margin, and as a male win of 2 sets gives a multiplier of 1.5, all multipliers/margins can be dealt with by adding a gender modifier to the equation. In this way, all results can be processed in the same place, with little modification:

IF gender = female

Gender Modifier = 1

ELSE

Gender Modifier = 0

IF Margin = 1

Score Modifier = 1 (i.e. no change in base score)

ELSE

Score Modifier = Gender Modifier + Margin - 0.5

**Statistics**

Statistics will be dealt with in their own file python file. There are eight different features to be displayed: Total Wins, Total Losses, Wins by three sets, two sets, one set, percentage of matches won, most wins by a player, and most losses,

Al eight of these features can be processed by combining four of the others: wins, losses, 1-set wins, and 2-set wins. I store only these four as a result.

Wins and losses, 3-set and 2-set:

As with the seeding matrix mentioned above, occurrences of any of these features increment a point in a stats matrix, attended by a current winners list, a score margin list, and a list of current losers. These are displayed as they are.

Thee set win

A three-set win is process and displayed as follows:

Three set wins = Total wins - (Two set wins + One-set wins )

Percentages

Win percentages are processed as follows:

Percentage = (Total Wins / (Total Wins + Total losses)) \* 100

Most wins and most losses

These are really the same question with my implementation and can be processed with a simple find max algorithm. The following applies for both:

Max = 0

For count in length of array

IF array[count] > Max

Max = array[count]

Return Max.

**Sorting**

Sorting has not been changed from the original submission. As the input size is small, and the sorting algorithm’s role in this program minimal, it wasn’t a priority. Had the possible input been of a larger size, more consideration would have been put towards this aspect.

**Evaluation**

In general, I’m unhappy with many aspects of my system. There is a lot of repeated code, and I have found modifying for seeding quite difficult, and at this time, it does not work. It’s possible it won’t be ready in time for submission, in part due to poor time management, but mainly due to the initial implementation being overly complicated. There is a lot of code, spread over 11 python files, and too many separate repositories.

To improve upon this, a more Object Orientated approach may have been preferable. For some reason I was initially resistant to OOP, but having had to try and deal with the mess I’ve created in this system, I think even having a single Player Class would have helped enormously in keeping things in order.

I had planned to design a modified binary heap to store information. The idea was to insert players into the heap depending on divisibility (modulo % 2) of their array position in addition to the usual practice of inserting lower valued items at the bottom of the heap. This may also have provided more structure to my system.

I think structure has been the main issue. Code repetition is regrettable, but having such a confusing sprawling program has made modification of even fairly simple aspects extremely difficult. This is an issue of maintainability that I will have to learn from. I thought that I’d given myself ample time to complete this project, but the reality is that due to poor structuring, the features I had most looked forward to implementing have had to be left unfinished.