# DADSA Assignment 2

**A tennis player ranking system.**

Takes scores for a group of tennis tournaments, dubbed "circuits", and efficiently produces useful stats.

## Startup

Requires Python 3. This project was tested using Python 3.6.3.

Please unzip project before running.

### Windows

*Tested with computers provided in UWE Frenchay campus 3Q85*

To run the program:

Double click `RUN.bat`

To reset the previous session data:

Either delete output folder, or double click `CLEAN_PREVIOUS_SESSIONS.bat`

To run the algorithm performance evaluations:

Double click `EVALUATION.bat`

### Linux

First, ensure you are in the source directory:

```
cd src
```

To run the program:

```
python main.py
```

### Usage

On first running the program, all errors and duplicated lines found in the configuration files located under `resources` and `src/config.py` will be brought to the attention of the user to resolve. The program should ignore any potential user errors and proceed, but it is important to keep note of them.

Once all errors have been resolved, the command line interface will be brought up. All commands may be written at the cursor position denoted by `>`. There are a wide range of commands to select

from, given as follows:

**Displays all commands**

```
help
```

**Quits the program.**

```
quit
```

**Starts the next tournament.**

```
start <tournament>
```

**Shows the scoreboard for the given season or tournament.**

```
scoreboard <season> [tournament]
```

**Shows the player with most wins and player with most losses.**

```
stats
```

**Gets number of times a player got a specific score in a tournament or season.**

```
stats score <player> [score] [season] [tournament]
```

**Gets total number of times a player won in a tournament, season, or overall.**

```
stats wins <player> [season] [tournament]
```

**Gets total number of times a player lost in a tournament, season, or overall.**

```
stats losses <player> [season] [tournament]
```

# Justifications

## Tournament ranking

Each time a match is executed, the program will automatically keep the player ranking updated. Players that have lost the tournament may have their ranking points immediately calculated, thus gives us the ability of knowing their tournament position before the tournament being complete.

Tournament ranking simply ranks players by appending them to the front of a doubly linked list upon loss, which gives `O(1)` time.

**Doubly linked list**

- Space complexity: `O(n)`
- Appending front or back (pushing): `O(1)`
- Deleting front or back (popping): `O(1)`
- Insertion at index: `O(n)`
- Deletion at index: `O(n)`
- Search (sequential): `O(n)`

## Season ranking

Season ranking is a little different, due to the collection requiring modifications even after players have been added to it. Season ranking requires the ranking to be maintained as each match passes. A well-programmed sort algorithm will have an average sort time complexity of `O(n log n)`. However self balancing trees tend to have modification time complexities of `O(log n)`. Since we are optimising for sorting on individual element modifications, it would be very expensive to reorganize and verify the entire data structure. Therefore I chose to use an variant of a self balancing tree called an order statistic red-black tree.

**Red-black order statistic tree**

- Space complexity: `O(n)`
- Best case access, search, insertion and deletion: `O(log n)`
- Average case access, search, insertion and deletion: `O(log n)`
- Worst case access and insertion: `O(log n)`
- Worst case search and deletion: `O(n)`
- The red-black self balancing method was chosen as it is guaranteed `O(log n)` lookup and modification time within the tree itself (forgetting node operations).
- Multiple players may have the same score, so each node in the tree must allow for storing multiple players. In this scenario, I have chosen to simply supply a linked list for holding the node values. This is why the search and deletion operations are worst case `O(n)`, unlike normal red-black trees.
- We may be required to find all players at a specified rank, or find the rank of a specified player. Each node is given a size, which can be used for performing these `O(log n)` operations.

## Indexing

We must take into account that third parties, whom are feeding us the data, may not understand nor wish to feed us our direct object references. Instead we will probably be given either player names or other types of identification. These can be used as keys which may then be used for indexing our own player profiles. Hash tables provide a speedy average lookup time of `O(1)` for this operation, so these will be used as a form of compatibility when interfacing with the data provider.

All the player scoring system uses hashing, which means any attempt to find what score a player has uses an average lookup time of `O(1)`.

**Hash table**

- Space complexity: `O(n)`
- Best case search, insertion and deletion: `O(1)`
- Average case search, insertion and deletion: `O(1)`
- Worst case search, insertion and deletion: `O(n)`

## Sorting

When loading data all at once into a list, such as the tournament ranking from file, it is more efficient to use a sorting algorithm over sorting by trees. Some operations in the previous solution have a worst case time complexity of `O(n)`, therefore processing the entire data set will give us an overall worst case time complexity of `O(n^2)`. As stated before, there are sorting algorithms that provide a worst case scenario of `O(n log n)`. Sorting is also able to put the data into an array data structure. Arrays have a guaranteed lookup time of `O(1)`, which is more optimal than the trees `O(log n)`. Hence a sorting algorithm would be better than a tree for less frequently updated data.

This then leaves the question "What sorting algorithm should be used?". There tons of different sorting algorithms, all with their own sets of advantages and disadvantages. Selecting an optimal sorting algorithm is generally down to what is to be expected of the input data. As this assignment is vague on the specifics on what to expect, it's probably best to use a general purpose sorting algorithm that has the best all-round statistics.

Tim sort was the first general purpose sorting algorithm that popped into my head. It is the main sorting algorithm used by Python and Java as it offers excellent benchmarks on sorting real world data. Tim sort achieves this by taking advantage of already sorted sections or "runs" of the provided data set. Though Tim sort is not the only algorithm that does this.

Block sort is a newer sorting algorithm that, like Tim sort, is a combination of merge sort and insertion sort and uses runs. Block sort is almost identical to Tim sort when ran on real world data, but it provides one extra advantage over Tim sort, a space complexity of `O(1)`. For comparison, Tim sort has a space complexity of `O(n)`.

My original design was to use block sort. Though I've found while gradually loading player data from a file, instead of putting it into an array, it could be put into a better data structure that helps with the sorting process. Pipe sort is a sorting algorithm I designed which was inspired by Tim sort. It's similar to Tim sort in the way that it is able to detect runs, both forwards and backwards. Even if the entire data set supplied was in reverse, pipe sort should be able to sort it in `O(n)` time.

Pipe sort differs from Tim sort in two ways. When feeding data into pipe sort, it not only finds runs in the data but it also groups these runs by their relative sizes using the specialised tree data structure specified in the stream solution. Note that this tree is indexed by the size of the runs, not the data itself. The other difference is, insertion sort is now no longer necessary as it is more

optimal to recursively merge sort all similar sized runs. We know all the data is sorted once there is only a single run remaining.

#### Pipe sort

- Space complexity: `O(n)`
- Time complexities:
  - Best: `O(n)`
  - Average: `O(n log n)`
  - Worst: `O(n log n)`

### Code size

All the algorithms defined above require a reasonably large number of lines of code in order to be implemented in a sane manor. The entire project has no excessive portions of duplicated code, nor does it violate PEP-8 (with a row length of 120). It is possible to shrink this project down to less lines of code, but that will require either deleting comments, switching out for lower performing algorithms, or generally decreasing the code quality.

## Other notes

- The library numpy was used in this project, not for ease of use but to emulate a proper C-style array of immutable size.
- I am aware that focusing on algorithms like this being optimal is futile in a language like Python, and that creating these in a lower level language will probably provide a performance gain. This assignment specified for them to be written in this language. If the intention was to get us to create native bindings, then I'd be happy to oblige.
- There are already many great libraries out there for Python that have similar behaviour to what I have written, and reinventing the wheel is usually something that should be avoided in software development. I made the exception here as I assume the assignment was designed to show an understanding of common data structures and algorithms.