Final Project Report

General Idea of the Project

The general idea of the project is a Pokémon-style battle game, where the user has to fight random enemies with some basic moves that they are given. The users are also given the option to buy moves as they receive coins when after winning battles and leveling up. The enemies show up at random intervals between 2 to 6 seconds. The enemies would have the same number of moves as the user and would have moves that do damage based on the user's current level (so that the opponent is balanced). The user levels up after each interval of 1000 XP gained. The user would gain XP and coins after winning a battle, based on how much damage the opponent has done.

The moves for the opponent are randomly generated from a pre-set list of names and adjectives and the damage is randomly selected based on the current level that the user is at. For simplicity sake, I don't get the moves for the opponent from the inventory and simply generate random ones based on the user's current level.

I generated a banner for my welcome text from http://patorjk.com/software/taag/#p=testall&h=2&v=2&f=3D-ASCII&t=Welcome%20to%20Ironfortune!.

I implemented a small probability of the user or the opponent missing the move (5%), which is implemented using a Fitness proportionate selection approach (see https://en.wikipedia.org/wiki/Fitness_proportionate_selection). Basically, I generate a number between 0 and 100 and if the number is less than whatever the probability is (in this case 5), then the move would miss, otherwise the move would hit. I use a similar approach for picking the move for the AI opponent. There is a 10% chance that the AI picks a move that is not the highest damaging move.

For the basic commands, I wrote the functions and made a dictionary with the command as the key and the function that the command runs as the value. I used this approach because it would make it easier to associate the command with the function and to validate the commands (if the command enter was correct).

I used the random library for everything that has to do with a random choice or random number generation. I also used the time library for anything that has to do with waiting for a certain number of times, such as waiting for the enemy getting generated or the opponent completing their moves.

I used loops throughout my project. For example, I loop through the list of moves when looking for the ones available to sell.

I used file IO for my move inventory to buy.

I used classes for the Opponent and Move.

I did string manipulation for the commands and interpreting whatever the user would enter. I also used it for interpreting the inventory.

I also formatted (using the f-string method) the strings to make it readable and understandable by the user (this would be considered string formatting more than string manipulation).

I used a lot of functions by breaking everything down into small functions that fulfill a single purpose.

I used sorting to sort the moves inventory and the user's moves.

I used many lists throughout the project.

I used dictionaries for the quote generation and the basic commands.

I decided to take an object-oriented approach, so that I can store all the information about an opponent/move as an object (see **Computational Thinking**).

Computational Thinking

I broke/decomposed the problem down into smaller parts (battle, moves, opponents, etc.) and then broke them down by nouns and verbs. Then, for each noun, I looked at the components I need to keep track of for each of those nouns and made those as a class/objects. The verbs became the functions. I used pattern recognition to recognize what needs to be stored in the objects.

I tried to break down each function so that it only fulfills one thing, and each function has one purpose. I also used pattern recognition for this, to recognize how similar the functions for the program are and to prevent repetition of code, I decided to pass values to some of my functions to tweak the functions accordingly to what I need.

I tried to use abstraction by having the classes complete some of the fundamental functions on the opponents and the moves and simply just calling those functions in the main.py file when needed. For example, if the user buys a move, in the main.py file, I simply call the buy() function from the Opponent.py file, which does all the computations. All the main.py file does is call the buy function when the user buys a move. Also, by having the functions each complete a single thing, I tried to incorporate abstraction, so that the functions simply call the other functions, while having the implementation details hidden.

Algorithm Complexities

Most of my algorithms have a worst-case time complexity of **O(1)** or **O(n)**. Here is the breakdown of the worst-case time complexities of my main functions:

- play() in main.py file: O(1) complexity. This handles the appearance of enemies and the starting of battles.
- battle(opp) in main.py file: O(n) complexity where n is the number of moves/rounds a battle would last until one of the opponents win. This handles the battles between the opponents.
- opponent_move(opp) in main.py file: O(n) complexity, where n is the length of the Moves list for the opponent. This performs the computations for the AI choosing a move.
- quit() in main.py: O(1) complexity. This quits the game.
- buy() in main.py: O(n) complexity, where n is the length of the inventory. This performs the front-end computations for the user buying a move.
- sell() in main.py: O(n) complexity, where n is the length of the inventory. This performs the front-end computations for the user selling a move.
- generate_opponent(lvl, num_moves) in inventory_management.py: O(nlogn) complexity where n is the num_moves (or the number of moves in the main_user's moves list) since it gets sorted

with merge sort. Without the sorting algorithm, it's an O(n) complexity. This function generates a random opponent for the user.

- inventory() in inventory_management.py: O(n) where n is the number of lines in the move_inventory.txt file (i.e. the number of moves in the inventory). This returns the move inventory.
- generate_inventory() in inventory_management.py: O(nm), where n is the number of adjectives in the moves_adjectives list and m is the number of names in the moves_names list
- merge_sort_moves(moves)+merge(l, r) in inventory_management.py: O(nlogn) where n is the number of moves in the moves list
- generate_quote(reason, name) in quote_generation.py: O(1) complexity
- reset_move() in Move.py: O(1) complexity. This resets the move after a battle to the maximum number of times a move can be used.
- can_be_used() in Move.py: O(1) complexity. This determines if a move can still be used (if there is still some remaining times)
- move_details(_full) in Move.py: O(1) complexity. This returns the move details as per what the program needs it for (which is determined by the _full parameter).
- reset_moves(): O(n) complexity where n is the number of moves in the opponent's moves list. This resets all the moves in the user/opponent's move list
- available_moves() in Opponent.py: O(n) complexity where n is the number of moves in the opponent Move list. This returns the moves in the opponent move list as a string.
- get_hit(move) in Opponent.py: O(1) complexity. This does the computations on the user/enemy for when the opponent hits.
- win() in Opponent.py: O(1) complexity. This performs the computations for when the user wins.
- defeat(opp) in Opponent.py: O(1) complexity. This performs the computations for when the user loses.
- buy_move(move) in Opponent.py: O(1) complexity. This performs the computations for the user buying a move.
- sell_move(move) in Opponent.py: O(n) complexity, where n is the number of moves in the user's moves list. This performs the computations for the user selling a move. It's O(n) because of the remove that is performed in the function.
- available_to_sell() in Opponent.py: O(n) complexity, where n is the number of moves in the user's moves list. This determines the moves that are available to sell/ that are sellable.

Tradeoffs and Issues

- A tradeoff for my inventory storage is that it could've been done with serialization instead of interpreting the lines of the files and instantiating the moves with each component of the line, but that could've opened the door to more input validation issues and thus thought directly reading from the line would be better.
- Another tradeoff is that I could've used a dictionary for the moves, so that the user can type the key name and so it would be easier/more understandable to use a move instead of having to look at a number that doesn't mean much, but implementing it as a list made it simpler to sort and perform operations on.

- Another tradeoff is that I pass the index for a few of my functions instead of passing the actual move, which would mean that I am accessing the list twice. I did this to simplify finding the move on which the operation needs to be done on (removing, accessing…).
- A tradeoff with my available_to_sell() function is that I loop through all my moves to find the ones that can be sold, instead of looping through only the moves that are of 0 buying price. The reason I chose to do this is because currently, the user's moves are sorted by damage points. If ever there is a move that gets given to the user for free with high point damage, that move would be further down in the list. Instead of resorting the list by buying price and then looking at the few that are at 0 buying price, I simply loop through the whole list to reduce the number of operations I perform on the list and prevent from having to sort it again.
- Something else is that I use merge sort for sorting any list of moves
- I didn't spend enough time on my inventory reading and thus do not have much input validation on the file input, but I spent that time to refine my project and the separate features.
- One major issue was that I completely had to remove the concept of buying items, since I wanted to spend more time with input validation and completely refining the main features
- Another issue is that there is a minor bug that I know about, which is that when the AI is playing, if the user enter commands during the time.sleep()s, the commands will run one after another whenever it is the user's turn. The reason I had to keep this in is that many of the potential fixes did not work and/or required a lot of extra code to get it to work. I think that the time.sleep()s are important, because they give a real feel to the game and the AI picking a move, and also doesn't overwhelm the user with a lot of data, which they could easily miss.
- One issue is that I don't save game state, with which moving ahead and buying moves becomes very difficult and useless, because you can't move ahead too much only in one run.
- Due to time constraints, I could not implement a proper way to regenerate the user's health. For now, I have them wait for 5 seconds while their full health regenerates.
- For now, the opponent's moves are randomly generated, meaning that they aren't moves from the move_inventory. This was done for simplicity, so that the opponent would be balanced as per the level that the user is at and the opponent generation would not get more computationally expensive (it prevents from having to check each move and if its at the appropriate level).
- The main_user in the main.py file is a global variable. It could've been used as a local variable that gets passed through the function parameters, but this would've required me to return the main_user in the functions that complete operations on the user to have those changes reflected on the actual user and would've used up more memory, instead of just having one global variable that gets used up throughout the code.
- The user's moves could've been represented as a dictionary to reduce the computation time for removing and/or picking a move, but having it as a list makes it computationally and logically simpler to sort and makes it easier to perform other operations on the moves list.
- The user's initial level is stored as a global variable. It could've been stored as a local variable that gets passed around to the functions, but using it as a global variable allowed me to reduce memory use (similar to having main_user as a global variable).

Potential Extensions/Improvements

- The main extension that I see for my program is adding a GUI component. Basically, the game would play on a graphical interface and there would be buttons to select the moves and there would be random images generated for the opponents. The health bar would be presented on top of each opponent to represent their current health. There would also be a progress bar for XP and the number of coins would also be seen during battle.
  There would be a representation of the inventory to buy moves and each move would be represented as a button, where the user would click which move they want to buy.
- Another extension that I have thought of is implementing items. The items would increase how much the user heals, amplify their damage, make the opponent skip their turn, etc.
- I was thinking of potentially adding a mana component, with which the user would have limited amount of mana to limit how they use their moves, and which moves they use. It would also prevent the user from simply spamming the maximum damage move.
- Saving game state, so the user's progress would get saved and the moves they bought, along with their game stats
- Have the user's health regenerate gradually as time moves on, instead of giving it one shot after a certain time interval.