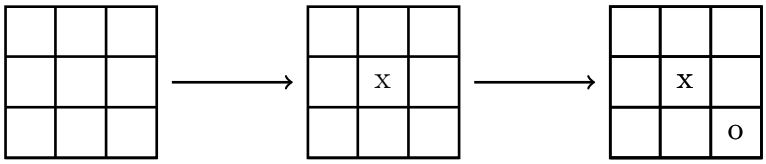
1 Backward Induction: Tictactoe [40 marks]

Consider the two player zero sum game (sequential move) of tic tac toe where,

- Players: {x, o}, player x plays first.
- Actions: $\{0, 1, \ldots, 8\}$, where each action represents the square in which the move will be played as shown below.

0	1	2
3	4	5
6	7	8

- Set of histories: H
  A history $h \in H$ is a sequence of actions taken from the starting state. Note, a history is different from a board position since several histories could lead to the same board position.
  Example: $h = [4, 8]$



- Terminal histories: Z
  The game ends in a win for either player or a draw at any terminal history $z \in Z$.
- Player function: $player(h)$ gives the player who makes the move at a given history $h \in H \setminus Z$.
- Action function: $actions(h)$ gives the valid actions (empty squares) at a given history $h \in H \setminus Z$.
- Utility function of player i: $utility_i(z)$ gives the utility for player i for a given terminal history $z \in Z$.
- Randomized strategy of player i: gives a probability distribution over valid actions i.e., $\Delta actions(h)$ for a given history $h \in H$ where player i plays.

1.1 Task

Find the subgame perfect equilibrium strategy for both players in the tic tac toe game using backward induction covered in class. Note this involves find the strategy for both players at every history $h \in H \setminus Z$.

1.2 Code

- `History` class:
  - `history`: list to keep track of sequence of actions

- `board`: board position (list) corresponding to the history
- `player`: player whose move it is at the given history

There are also several helper functions defined but not implemented. This might be needed when coding `backward_induction`. Feel free to implement this in anyway if needed.

- Global variables `strategy_dict_x` and `strategy_dict_o`: These will be updated in the recursive backward_induction function to keep track of the strategies which are a mapping from histories to probability distribution over actions. Note testing will be done w.r.t these variables (returned by `solve_tictactoe` function which calls `backward_induction`).

  1. These are dictionary with keys as string representation of the history list e.g. if the history list of the history class object is [0, 4, 2, 5], then the key is `"0425"`. Each value is in turn a dictionary with keys as actions `0 to 8 (str "0", "1", ..., "8")` and each value of this dictionary is a float (representing the probability of choosing that action). Example: `{"0452": {"0": 0, "1": 0, "2": 0, "3": 0, "4": 0, "5": 0, "6": 1, "7": 0, "8": 0}}`
  2. Note, the strategy for each history in `strategy_dict_x` and `strategy_dict_o` is probability distribution over actions. But since tictactoe is a Perfect information extensive form game (PIEFG)[1], there always exists an optimal deterministic strategy (SPE). So your strategy will be determistic (0s, 1s). The above data structure just keeps it more general.

- Function `backward_induction`
  Input: `History` class object
  Return: float

  - Implement backward induction for tictactoe.
  - Update the global variables `strategy_dict_x` or `strategy_dict_o` which are a mapping from histories to probability distribution over actions. Do this before you return the best utility.

Refer to code and doc strings for more information. Implement backward_induction function to get the strategies for both players. Also if needed, implement the blank functions part of history class.

1.3 Test

There are no public test cases for this question on gradescope. However, the given code saves your computed `strategy_dict_x` and `strategy_dict_o` in `policy_x.json` and `policy_o.json`. You can run `play_tictactoe.py` as follows and play against your computed policies to test it. This requires `pygame` package.

```
python3 play_tictactoe.py --BotPlayer x --BotStrategyFile policy_x.json
```

or

```
python3 play_tictactoe.py --BotPlayer o --BotStrategyFile policy_o.json
```

[1]This is the type of games we discussed in the class, where players can 'perfectly' see the actual state of the game and there is no part of the game which is invisible. Note that, this may not be a case in every game, e.g., in card games, the hands of the other players or the cards in the deck are unknown.

On gradescope, there will only be private test cases for this question, where the game will be played out starting from several random histories against your strategy for both players.

2 Alpha beta pruning: Notakto [50 + 10 marks]

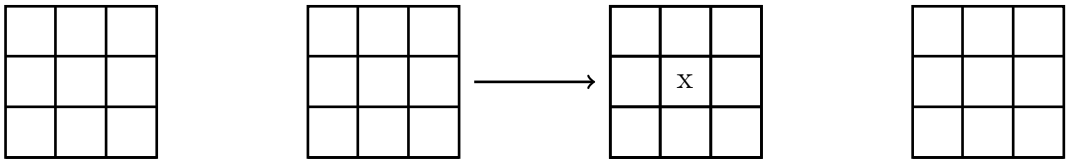
In this question, we consider the two player zero sum game of Notakto.

- Players: {1, 2}, player 1 plays first
- Number of boards: n
- Actions: $\{0, 1, \ldots, 9n-1\}$, where each action represents the square in which the move will be played as shown below (if $n = 2$).

0	1	2
3	4	5
6	7	8

9	10	11
12	13	14
15	16	17

- Histories: H
  A history $h \in H$ is a sequence of actions taken from the starting state. Note, a history is different from a board position since several histories could lead to the same board position. Example: $h = [4]$



- Terminal histories: Z
  The game ends in a win for either player at any terminal history $z \in Z$.
- Player function: $player(h)$ gives the player who makes the move at a given history $h \in H \setminus Z$.
- Action function: $actions(h)$ gives the valid actions (empty squares on **live boards**, note on dead boards a 3 in a row has been achieved) at a given history $h \in H \setminus Z$.
- Utility function of player i: $utility_i(h)$ gives the utility for player i for a given terminal history $z \in Z$.
- Randomized strategy of player i: gives a probability distribution over valid actions i.e., $\Delta actions(h)$ for a given history $h \in H$ where player i plays.

2.1 Task

1. Implement alpha beta pruning covered in class to find the maxmin value of the game for any given history (player 1 is the maximizer and player 2 is the minimizer). Note, in some cases of alpha beta pruning several nodes are pruned by the algorithm. In this case, the better move occurs on the left side of the tree. Alpha beta pruning run where the lowest action

numbers are searched first might take longer time to run. So, after getting valid actions, try to search a stronger action first (center, corners and then the remaining in that order). [50 marks]

2. Implement maxmin rule. But store and use the values of already visited board positions to avoid recursive calls for other histories that reach the same board position. [10 marks]

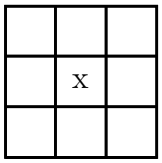
See which is faster for number of boards $n = 2$. For $n = 3$, both methods might take several minutes and maybe even run out of memory. Other variants of alpha beta pruning might be required to efficiently search such large games.

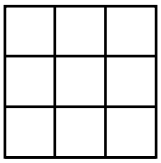
2.2 Code

- `History` class:
  - `num_boards`: int
  - `history`: list to keep track of sequence of actions
  - `boards`: list of lists where each list represents one of the boards.
  - `current_player`: player whose move it is at the given history
  - `active_board_stats`: list to keep track of active boards i.e., `active_board[i] = 1`, if board i is active (no three in a row), otherwise `active_board[i] = 0`.

  There are also several helper functions defined but not implemented. This might be needed when coding `alpha_beta_pruning` and `maxmin` functions. Feel free to implement this in anyway if needed.

- Global variable `board_positions_val`: These will be updated in the recursive maxmin function to keep track values of already visited board positions, to avoid recursive calls for other histories that reach the same board position. These are dictionary with keys as string representation of the boards list. Example for $n = 2$ and the following boards the key will be `"0000x0000000000000"`. Each value of the dictionary is a float denoting the maxmin value of the game at that board position.





- Function `alpha_beta_pruning`
  Implement alpha beta pruning using a move ordering (center, corners and then the remaining in that order). The maxmin value for several histories will be tested for this function. Additionally, all the histories visited via the pruning will also be checked. Note, tracking histories visited is already taken care of in the code via global variable `visited_histories_list`.

- Function `maxmin`
  Implement maxmin rule, but store the values of already visited board positions to avoid recursive calls for other histories with the same board position. Use the given global variable

`board_positions_val_dict` to track the value of visited board positions. This is a dictionary with keys as str version of `self.boards` and value represents the maxmin value. Use the `get_boards_str` function in `History` class to get the key corresponding to `self.boards`. The maxmin value for several histories will be tested for this function.

Refer to code and doc strings for more information.

2.3 Test

There are no public test cases for this question on gradescope, test the implementation for few histories yourself (maybe make use of optimal strategy Notakto). There will only be private test cases for this question, where the maxmin values for several histories will be checked for `maxmin` and for `alpha_beta_pruning` we'll check both the value and the histories visited.