COMP 1405B
Fall 2019 – Practice Problems #9

Objectives

- Practice writing recursive functions

Remember to test your solutions to verify their correctness and discuss your solutions with both other students and TAs in the course. Verifying your solutions with TAs will be a good way of ensuring that you are on a path to success in the course.

Problem 1 (Multiplying - *)

Write a recursive function called **multiply(int,int)** which calculates the value of a*b without using multiplication (only use addition/subtraction operators for calculations). To do this, think of how multiplication is really computed in the most basic sense and identify the base/recursive cases. Test out the function with various numbers.

Problem 2 (Reversing String - **)

Write a recursive function that takes a string and returns the reverse of that string. For example, reverse(“string”) = “gnirts” and reverse(“hello there”) = “ereht olleh”. You can use the string methods and/or slicing operator to extract sections/characters from the string and work toward a base case.

Problem 3 (Merging Lists - **)

Write a recursive function that takes 2 sorted lists as arguments and returns a single, merged sorted list as a result. For example, if the two input lists are [0, 2, 4, 6, 8] and [1, 3, 5, 7, 9], the returned result should be [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. If you are looking for more of a challenge, extend this to implement a merge sort function.

Problem 4 (Nesting Dictionaries - ****)

Problem 7 in problem set #7 dealt with nesting dictionaries to more efficiently store and search a set of words. The structure used in that problem is a natural fit for recursion. If you haven't already, implement a solution to that problem that uses recursion.

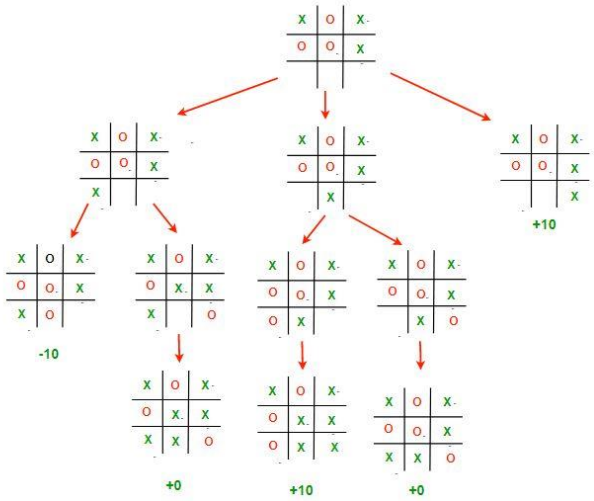
Problem 5 (Factorials and Caching - **)

Create a Python file called **factorial.py** that has a single function called **cachedfactorial(int, dict)**. The cachedfactorial function will accept an integer and a dictionary as arguments. The function must recursively calculate the factorial value. Additionally, this function must use/update the dictionary-based cache to save any intermediate calculation results and terminate early if a required value can be found in the cache. For example, upon completion of cachedfactorial(5), which computes the value of 5! (5*4*3*2*1), the cache should have stored the values of 5!, 4!, 3!, 2!, and 1!. If you try to compute the value of 7! (i.e., 7*6*5*4*3*2*1) afterward, you should not need to recursively compute the 5*4*3*2*1 part again. Instead, you should be able to use the value of 5! stored in the cache to stop early.

Problem 6 (Tic-Tac-Toe Progressive Problem Part 3 - *****)

If you completed Part 2 of this problem, you should have a list-based tic-tac-toe game that can be played by human players. In this part, we will apply a well-known AI algorithm called minimax to create an AI player that cannot be beaten. Before we can do so, we will have to understand a little bit about how games are represented when applying these types of algorithms.

The minimax algorithm (and many other AI algorithms) will work using a ‘game tree’. To understand a game tree, consider an empty tic-tac-toe board. The person making the first move has 9 possible choices of where to go. Depending on where they make their move, the game could be in 9 possible states after the first move is made. The other player then has 8 places to choose from. Once they make their move, each of those 9 possible states could lead to 8 different following states (making 72 possible states after 2 moves). This process can continue up to 9 times, until the game is in an ‘end state’ (somebody has won or the game board is full). A segment of this process can be viewed in the picture below, from this webpage:



The goal of the minimax algorithm is to search through this game tree and find the move that minimizes the possible loss in the worst-case scenario. The algorithm works under the assumption that the opposing player is trying to maximize the loss (hence the mini/max). To do so, the algorithm will search every possible path on this game tree and remember which one(s) generated the best result (in this case, the path(s) that minimizes the chance of losing the game). The AI player using this algorithm will then play the move that leads down the best path.

To search through a game tree, we can apply a recursive implementation of an algorithm called depth first search (DFS). The minimax algorithm we implement will use the same approach, but with added steps to remember the most favorable paths. A general DFS algorithm can be applied to a game tree like this (think about how this would operate, it might be worth drawing an example similar to the picture above):

```
dfs(gameboard):
    if gameboard represents a finished game:
        return
    for every possible move x in this gameboard (empty spaces):
        Make move x       #this changes the state of the gameboard
        dfs(gameboard)    #performs the search in the new gameboard (going deeper)
        Undo the move x #that is, set the gameboard back to what it was
```

You should already have a way of determining if the tic-tac-toe game is over, which is a necessary base case to stop the search process. One further thing you will need before implementing the minimax algorithm will be a method for measuring the ‘value’ of the game board. For this problem, you can create a function called boardvalue that accepts the game board list and returns 1 if the AI player has won, -1 if the human player has won, and 0 if it is a tie or the game is ongoing. The AI player, then, will be trying to maximize the value of the board, while assuming the human player is trying to do the

opposite. Once you have this function, you can start implementing a minimax function using the 'pseudocode' below, which takes advantage of Python's ability to return multiple values.

Input: board – the 2D list representing the tic-tac-toe board
Input: aiplayer – a Boolean representing whether the current player is the AI or not
Output: bestmove – a 2-item list containing the row/column for the best move the player can make in the given board
Output: bestscore – the best score the player can achieve if it makes this move

```
minimax(board, aiplayer):
  if the game is over:
    return None (no move to make, game is over) and boardvalue(board)
  if aiplayer:
    bestmove = None
    bestscore = -Infinity (or any value less than the lowest possible board value)
  else:
    bestmove = None
    bestscore = Infinity (or any value greater than the highest possible board value)

  for every open space in the board (i.e., every possible move):
    Add the player's symbol to the space
    move, score = minimax(board, not aiplayer) #that is, the opposite player
    Remove the player's symbol from the space

  if aiplayer and score > bestscore:
    bestmove = the space we played in (the row and column)
    bestscore = score
  if not aiplayer and score < bestscore:
    bestmove = the place we played in (the row and column)
    bestscore = score
  return bestmove, bestscore
```

Once you have the function implemented, you can create a getaimove() function that calls minimax(gameboard, True) and plays the best move that is returned.

This is a tricky implementation that will likely be hard to find errors in if you make them. If it isn't working properly, you can try printing out information for each board state to get an idea of where it is going wrong (what moves are being made, what gameboards they produce, how the score is measured). It may also be helpful to start by applying it on a

board where the game is almost over. If the AI can find a winning move over the last 3 moves in the game, it will likely generalize to an empty game board. Finally, you may be able to change the design of your game to make the implementation a bit easier. For example, force the AI to always be the “X” player and the human to be the “O” player. If you are still stuck, you can consider reading more about the minimax algorithm and viewing other pseudocode/implementations (Wikipedia, GeeksForGeeks). If you do get the implementation to work, the same approach can be used for similar games (connect four, chess, etc.). The more possible moves there are, the more computationally intensive the process becomes. Think of how large the tree would grow in chess, where there are many more possible moves each turn. For this reason, you often will have to ‘limit the depth’ that your search will go to and implement a ‘heuristic’ that estimates the value of a partially completed game. There are some useful terms in that sentence if you feel like learning more.