Assignment 5

Recursion, Sorting, and Pathfinding!

SUBMISSION REQUIREMENTS: Submit a single zip file called **assignment5.zip**. It must contain all of your question code and sample data. Information regarding deductions for late submissions, invalid submissions, and grade disputes are available on the cuLearn assignment submission page.

MARKING NOTES: This assignment has **??? marks.** A full marking scheme with notes, deductions, and policies will be made available on the cuLearn assignment submission page shortly.

The Final Assignment

Dramatic, right? For this assignment, we have three (hopefully) straight-forward questions that rely on your ability to perform some abstract problem solving, apply some new concepts, and work with pre-existing code. You are expected to be able to:

- Write a recursive function using a technique we've *briefly* covered in class - caching/memoization
- Implement a simple sorting algorithm, but with a small twist!
- Understand a new algorithm's pseudocode and tweak it for a new purpose

All in all, this assignment is intended to be more representative of the sorts of problems you can expect to see in coding challenges, future assignments, and misc. practice problems.

Testing your Program

- There are some marks allocated to each question for **testing**
- In each file, you should have a **main function** which runs your code with a few different inputs and prints the **expected value** and the **actual value**
- The exact format of the tests isn't too important, and you should have *some* coverage of possible test cases, but we won't expect every case. A few is fine!

Restrictions

- There should be no global variables that are not constants
- You must write all of this code yourself; you may not use previous solutions (even if they are your own!)

Problem 1: Recursive Cached Fibonacci

One of the early problems many people start with in recursion is to generate the **Fibonacci Sequence**. Put your code in the file **question1.py**. The Fibonacci sequence is a sequence of numbers, where the **n'th** number is equal to the sum of the previous two numbers, (n-1) and (n-2). Thus, the first 10 digits are: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.

We see that the Fibonacci sequence is self-referential in its definition, which makes it an excellent candidate for recursion! Except… There is a lot of unnecessary repetition involved in this. To calculate even the 50'th Fibonacci number takes a *very* long time, and might even cause a stack overflow! Let's fix it with **caching**.

What is a cache?
A cache just refers to a dictionary of things that we've *already processed* and don't want to process again. A popular place where you see caches is in your **web browser.** Your web browser will visit a website and try to store as much data as it can offline. The next time you visit the website, it will load faster because it didn't have to download so much the second time!
In our case, we can have a dictionary act as a cache. The keys are the relevant inputs to our function, the value is the final calculated value. This way, if we call the function again, we start by checking if the key is in our dictionary. If it is, just return the value! If it isn't, calculate the value, and toss it into our dictionary for next time. Sometimes you store the cache **globally**, but for our problem, we're passing it **as a parameter** to the function each time. The first time we call the function, we start with an empty cache.

For this problem, implement a function called **cachedfibonacci(int, dict={})** (note, naming just to show types, name the parameters anything you'd like) which:

- The int parameter represents the n'th number of Fibonacci we'd like to return
- The dictionary parameter is defaulted to empty, but overtime fills up with **solved** Fibonacci results
- The function **returns** the final value for the n'th Fibonacci number

Thus you must:

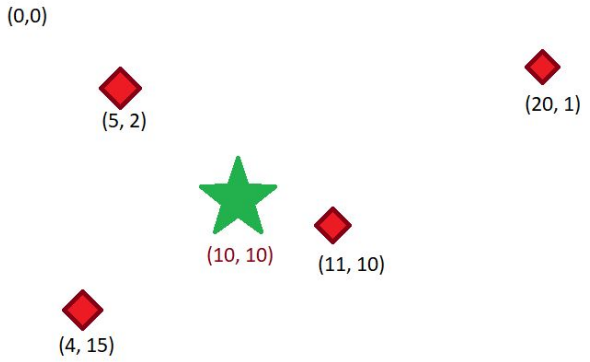
1. Recursively find the n'th Fibonacci number
2. Store each result in the cache
3. If the value you are trying to calculate is already in the cache, use that number and do not repeat the Fibonacci calculation

Test Note: If you are **properly** using the cache, you should be able to calculate the 100'th Fibonacci number (354,224,848,179,261,915,075) near-instantly. If you are not using the cache

correctly, you will likely be unable to compute the number past 50 (likely even *significantly* lower).

Problem 2: Cocktail Sort, and Nearest Enemies

When working with location data (x,y), it's often important to take a list of positions and sort them by how close they are to a specific point. Write your code in a file named **question2.py**



Visualization of different points on a 2D map and coordinates.

For this, we're going to implement **cocktail sort**, a variation of bubble sort that bubbles values up and down in both directions. This sort is mostly for academic purposes, used primarily as a teaching tool for implementing algorithms, but it should still be able to perform well enough for us.

Write a function, **closest_enemies(tuple(int,int), list(tuple(int,int)))**. This signature is a bit complex, but really, it could look as simple as **def closest_enemies(hero_position, enemy_positions):**

As parameters, the function accepts:

- A tuple, containing two integers, representing the x, y coordinate of our player character
  - Ex. (50, 10), (20, 35)
- A list of tuples, each tuple containing two integers, representing enemy positions
  - Ex. [(10, 20), (55, 10), (23, -5), (0, 200)]

The function should sort in place: This is to say, it should not need to return the list (but it may) as it will modify the input list of enemy positions.

Write a function which uses the cocktail sort (psuedocode on next page) to sort these points by the **shortest Euclidean distance** to the player's position - *not* simply the value. Recall that the distance between two points can be calculated as $dist(a, b) = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$. You

may wish to create a function to do this, but it is not required. **You must write the code yourself:** no libraries, modules, or built-in Python functions to calculate distance/sort.

Problem 2: Cocktail Sort, and Nearest Enemies (cont.)

Pseudocode for Cocktail Sort

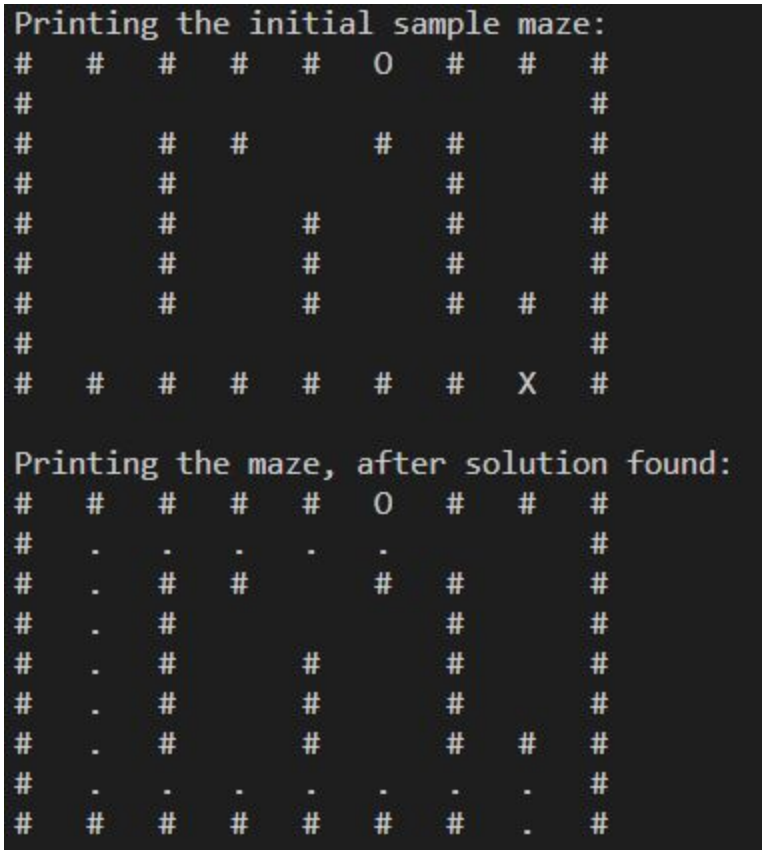
```
cocktailsort(a: list of items):
      Swapped = True
      While Swapped:
            // Shake Up
            For each index i from 0 to list max - 2:
                  If a[i] > a[i+1] then:
                        Swap a[i] and a[i+1]
                        Set swapped to True
                  End if
            End for

            If we didn't swap then end the loop

            // Shake Down
            Set Swapped to False
            For each index i from list max - 2  to 0:
                  If a[i] > a[i+1] then:
                        Swap a[i] and a[i+1]
                        Set swapped to True
                  End if
            End for
      End while
End cocktailsort
```

Problem 3: Searching for an Exit

This problem is to test your ability to make use of new code, solve some more recursive problems, and *start* from pseudocode but specify the algorithm to a specific problem! You will be using a technique referred to as a *depth-first search* (DFS) to look for an exit. A lot of the code has already been provided to you on the cuLearn page to help you with grid coordinates and generating a sample maze. The maze will be sent to the function as a list of lists, so there is no specific way to get the maze data; in your main function tests, you can pull from a file, just hardcode it, or write functions to generate them!



In a file named **question3.py**, include a function called **dfs(list(list(string)), tuple(int, int), list(tuple(int, int)) = []),** or as a more simple example of what it might look like in code: **dfs(maze, position, explored=[])**.

You will need to make use of the functions provided to you to take in some maze and print the maze, with a valid path from the start to end. It does not need to be the shortest path. **Be careful** to review the functions that have already been written for you so you do not do unnecessary work.

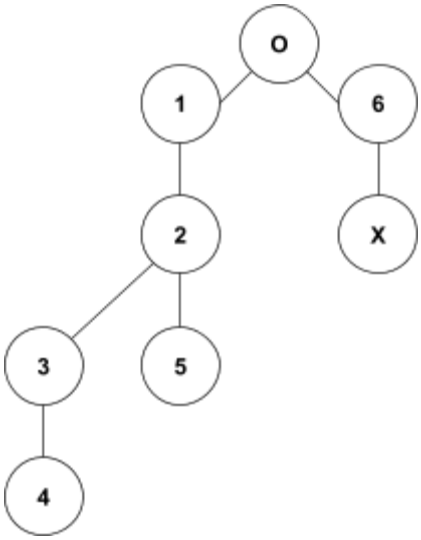
Overview of DFS and provided functions are on the next page

Problem 3: Searching for an Exit (cont.)

Depth-First Search

The DFS is used a lot in computer science as a way to search *trees* or *graphs* for specific nodes and many other purposes, but it was originally developed as a maze solving strategy!

DFS works by following one path as far as it can - let's say it always goes left first. Every time it visits a spot, it adds that position to a list of "explored" nodes so that we know not to visit it again. When it gets to the end (no more places to go) it returns "False" to say it didn't find anything. If it finds what it's looking for, it reports back "True" to say it found it!



In the case of the graph to the right, I've numbered it in the order our DFS would look at the nodes, seeking from O to X. It clearly doesn't take the best path, but it finds the end eventually.

DFS in your Maze

The logic of DFS remains the same, but you will need to work out some extra logic to make it work how we need it to. We can consider our maze a "graph" where each spot is connected to up to four other spots; the spot above, left, right, and below. It's considered "adjacent" if it's one of those four positions, within bounds, and not a wall ("#"). A function has been provided to give you all adjacent spots - don't worry! We're starting from "O" (position supplied in the function) and going to "X".

You will notice in the below pseudocode that it handles looking at everything, but *doesn't* account for actually finding the endpoint and drawing the final solution. You will need to figure this out on your own!

Pseudocode for a Recursive Depth-First Search:

```
dfs(list_of_connected_points, current_point, explored=[])
      Add current_point to explored
      For each point adjacent to current_point:
            If that point is not in the explored list:
                  Call DFS on the list of connected points with the new point
            End if
      End For
```

Overview of provided functions on the next page

Problem 3: Searching for an Exit (cont.)

On cuLearn, you will find a Python file named **maze_helper.py**. **Import** this file into your **question3.py**; you can rename it anything if you'd like. There will be minor deductions for copy-pasting the code into your file rather than importing. Make sure to include the maze_helper.py in your submission.

This file contains a number of functions to help you avoid dealing with spatial issues when writing your algorithm. Carefully review the purpose of these functions, how to use them, and where they might help you.

sample_maze(): This function provides you with one sample maze. Feel free to create more mazes if you'd like.

get_adjacent_positions(maze, position): This function will check each location around the **position** provided and return a **list of tuples** containing the positions of each location that is **not a wall**.

symbol_at(maze, position): This function returns the symbol in the maze at the provided position. While these are simple, these functions have been provided so that you do not need to worry about treating the list as row/col vs x/y.

add_walk_symbol(maze, position): This function will place a dot at the correct position in the maze.

print_maze(maze): Prints the provided maze.

NOTES: These functions are provided so that you do not have to care about the format of the positions. Using this code, you should **never have to access an individual position's x or y coordinates.** There will be deductions for not using the provided functions and relying on the tuple's values (i.e. calling position[0] or position[1]).

Extra

While there are no bonus marks, if you are interested in showing off some cool stuff and trying some more complicated problems, you can build a **maze generator** or show off the maze solving in **turtle** or **pygame** instead! Either one will still receive regular marks (i.e. you will not lose marks for showing the path in 2D) as long as they're still within spec (i.e. the correct maze

format). There will be a cuLearn forum to post if you decided to do some extra, and I might show some off in class! Don't stress over it of course. It's a busy time! Allocate it as you need to :)

Recap

- Your cuLearn submission should be a single file, **assignment5.zip**
- Your zip file should contain **three Python files, question1.py, question2.py, question3.py, maze_helper.py,** and any other data files needed to run your program (if you chose to use files for your maze data).
- Your zip file should contain the sample data used to test your program
- Late submissions will receive a 2.5%/hour deduction up to an 8 hour cut-off period
- Invalid submissions (incorrect name, incorrect function names) will receive a 10% deduction immediately
- As usual, you are expected to submit periodically; as you complete questions, try to submit to cuLearn, just in case of data loss or last-minute submission problems.
- It is your responsibility to verify the submitted files work correctly - redownload and try them again

Updates and Clarifications

- ---