Computer Laboratory 02

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Essential information

- This assignment is due Tuesday February 11th at noon and will be turned in on gradescope
- The assignment is intended to be done in pairs, although you're allowed to work on your own if you want.
- You can start the assignment before your lab period, although I recommend not getting too far – the lab TAs often have useful hints, and I don't want you wasting too much time struggling with something that the TAs might announce in-class.
- For more rules see the lab rules document on canvas.

2 Introduction

This laboratory is intended to give you further practice with loops, logical conditionals, and functions in python. Additionally, this lab will prominently feature lists. We'll only have just begun working on lists in lecture – you *probably* won't need any new syntax. That said, given how new some of these ideas are, be prepared to need to look up examples of python syntax in zybooks as you work. In particular, you should make sure you know:

- What a python list is
- How to create a list in a loop using the append method.
- How to access values from a list by index.
- How to loop over a list.
- How to get the size of a list.

If you don't know how to do one of these things stop now Chapter 4.1 in the textbook covers many of these, at least briefly. You are also free to discuss these ideas with the students around you, or the lab TAs (so long as your discussion is about the core syntax here, not lab02 itself.)

3 Software environment setup

If you need a reminder on how to set up a project for lab02, we recommend opening the PDF for lab01. For brevity I'll skip the instructions here, and focus on a few core reminders:

- You should create a place on your computer to store files for this class. Many students find having a folder such as Documents/classes/csci1913 or similar useful. Ultimately you are in charge of organizing your own files on your computer. Just know that NOT being deliberate about where you put files can lead to issues when it comes time to find these files for submission
- Each assignment in this class should be it's own pycharm project. If pycharm is showing you lab02 and lab01 files at the same time you may have misconfigured your projects.
- It's always smart to double-check that your project is configured for python3, not python2.
- You are, of course, free to use software other than pycharm, but we cannot promise to help you with any issues that arise due to this decision. – we can only guarantee experience with the pycharm platform.

4 Files

This lab will involve the following files:

- (REQUIRED, YOU WRITE) nim.py – you will program this. This will contain functions that capture key parts of the rules of the game nim.
- (REQUIRED, PROVIDED) nim_tester.py – we will provide this. This file will contain code to help you test if your nim functions meet our expectations.
- (OPTIONAL, PROVIDED) play_nim.py – we will provide this. This file will, when combined with your nim code, let you play a game of nim.

You should begin by downloading the two provided files and moving them into your project folder You should then make a new python file for your solution. Don't forget that you are required to have a comment at the top of any file you author that contains your name, and the name of any lab partners you worked with, as well as docstrings for each function.

5 Nim

Nim[1] is a simple two-player strategy game. The game is played using several "tokens" (sticks, coins, matches, or even pencil marks). These tokens are organized into several rows:

[1] `https://en.wikipedia.org/wiki/Nim`

```
====================
1 #
2 ##
3 ###
4 ####
====================
```

Players then take turns removing tokens. On any turn players can choose a row and remove tokens from it. Players can remove 1 to 3 tokens (neither more less). Once a player has chosen a row and removed tokens from that row, the other player goes. The objective of the game is to make the other player take the last token. This is possible because each turn a player has to take at least one token. Therefore if, at the beginning of a player's turn, there is only one token left, they would be forced to take it, and in doing so, lose.

A quick example game can help make this more clear – try to follow the playthrough here and connect the moves chosen to how the token counts change:

```
====================
1 #
2 ##
3 ###
4 ####
====================
Choose a row, 1 to 4: 3
Take how many tokens, 1 to 3 (no more than available): 2
====================
1 #
2 ##
3 #
4 ####
====================
Choose a row, 1 to 4: 4
Take how many tokens, 1 to 3 (no more than available): 3
====================
1 #
2 ##
3 #
4 #
====================
Choose a row, 1 to 4: 1
Take how many tokens, 1 to 3 (no more than available): 1
====================
1
2 ##
```

```
3 #
4 #
====================
Choose a row, 1 to 4: 2
Take how many tokens, 1 to 3 (no more than available): 1
====================
1
2 #
3 #
4 #
====================
Choose a row, 1 to 4: 2
Take how many tokens, 1 to 3 (no more than available): 1
====================
1
2
3 #
4 #
====================
Choose a row, 1 to 4: 3
Take how many tokens, 1 to 3 (no more than available): 1
====================
1
2
3
4 #
====================
Choose a row, 1 to 4: 4
Take how many tokens, 1 to 3 (no more than available): 1
Game over, the last player lost!
```

It might be worth playing a quick game of nim (on paper) with your lab partner before moving on While complete knowledge of the nim game is not **required** to work on this lab – it will help A LOT.

6 Formal Requirements

While it might be fun to require you to implement a whole nim game from scratch, that would be a bit long for lab02. Therefore we have given you a basic framework (in the provided files – see the description of those earlier in the document!) What you need to provide are 5 core functions that, together, encode the major rules of the nim game. While these functions work together and build-up-to a comprehensive program, each can be understood on it's

own and developed independently. Some of these functions may even be useful outside of this specific application.

The five required functions are:

1. `create_game_state(size, token_max)`
2. `is_valid_move(game_state, row, takes)`
3. `update(game_state, row, takes)`
4. `draw_game_state(game_state)`
5. `is_over(game_state)`

Before describing these functions – let's define an idea that will be used by all of the following functions:

6.1 Game State

All of these functions use the term game state. In computer-game theory, "game state" refers to the information stored in the computer about the game. In our case, a "game state" is the number of tokens in each row (along with the number of rows). We can store this using a python list where each index of the list represents a row, and the value at that index represents the number of tokens in that row. For instance the game state:

```
====================
1 ####
2 ###
3
4 #
====================
```

would be represented with the list `[4,3,0,1]`

Most of the following functions either *create* these lists, or take them as parameters.

6.2 create_game_state(size, token_max)

This function takes two parameters: size and token_max. It should return a newly created list of integers representing the number of tokens in each row. The size parameter sets the length of the newly created list, and the token_max parameter controls the values. In particular, the first row (position 0 in the list) should have 1 token, the second should have 2 tokens and so-forth, up until we hit token_max – after which each position should have token_max tokens.

Example: if you call `create_game_state(5, 3)` you should get the list `[1,2,3,3,3]` as a return value.

Notes:

- You can assume that both size and token_max are positive integers.
- if a size value of 0 is given, you should return an empty list.

- You **cannot** assume that size > token_max.
- I recommend looking at the examples in the tester file to make sure you fully understand this function.

6.3 is_valid_move(game_state, row, takes)

This function takes three parameters, game_state is a list of numbers (how many tokens are in each row) row is a **string** representing which row the user has requested to take tokens from, and takes is a **string** representing how many tokens the user wishes to take. The function should return a boolean value – true to indicate that user has chosen a valid move, or false to indicate that the move would be invalid. **Ultimately, a primary purpose of this function is validating the user's raw input** as such, make sure to code defensively and plan for all kinds of weird input Strings (since you cannot predict what a human might type!)

Notes and Details:

- The row number is expected to be input 1-indexed (so "1" means the first row, not 0)
- This function must not modify the game_state parameter – the given list should have the same values before and after the function.
- Row and takes are strings, not integers. You CANNOT assume that they will only have integers in them or be in any particularly valid form. You can assume the string is non-empty (has at least one letter in it)
- A string that contains anything OTHER than digits (1,2,3,4,5,6,7,8,9 or 0) is invalid.
  - The string method `isdigit` can be used to ask a string if it only has digits (and no other letters).
  - If you are unclear how to use this method feel free to use google to learn more: "python string isdigit"
- If the row number is out-of-range then the row is invalid. The valid row numbers are between 1 and the length of the game_state list (inclusive)
- To be valid the takes number must be between 1 and 3 (inclusive)
- To be valid, the takes number must be less than or equal the number of tokens in the selected row (I.E. if a row only has 2 tokens, it is not valid to take 3 tokens)

Hints:

- the row and takes parameters are strings Exactly as input by the user – this will require some care and some deliberate type conversion. Thinking carefully about this complexity is a core part of what makes this function hard. If you are deliberate, however, about remembering when variables contain strings or integers, this is quite doable.
- rows will be entered by the user 1-indexed (meaning "1" represents the first row) this is different from how rows are handled in python – where the token_count list is 0-indexed (0 represents the first row.) Your code will need to account for this.

- remember to convert the string inputs to integers before using them as list indexes – indices MUST be integers. the `int(...)`function can be used for this, but only if you have already checked if the string is numeric.

6.4 update(game_state, row, takes)

The update method takes three parameters game_state, row, and takes. While these have the same name as the parameters to the is_valid function, there are important differences. Like before, game_state is a list of how many tokens have in each row in a nim game (similar to one that create_game_state might return). row is an **int** representing which row the user has requested to take tokens from – this will be 0-indexed (not 1-indexed like the user input) and therefore will represent a list index directly. takes is an int representing how many tokens the user wishes to take. The function should return a **newly created** list that represents the board after the listed number of tokens have been taken from the listed row.

Example: if you call `update([5,3,4,1], 2, 3)` it should return `[5,3,1,1]` (index 2 in the array has been reduced by 3, the rest are unchanged.) The input list should remain unchanged.

Notes:

- You do not need to re-check that the inputs are valid in this function. If you look at how the functions are used you will see that this function is only called after validating inputs using the previous one.
- You should assume that row and takes are integers, that row is a valid index for the game_state list, and that takes is between 1 and 3, and not greater than `game_state [i]`. This is to say, you can assume that these parameters indicate a valid move as previously defined.
- Remember that the row is a 0-indexed integer in this problem, not a 1-indexed string like the last problem. You can use it directly like a list index.
- The game_state list input to this function should be unmodified by this function (I.E. it should have the same value before and after this function runs)
- The returned list should be a copy of the input list except for the index indicated by row – which should have fewer tokens as indicated by the takes variable.

Hints:

- This function's requirements may not match your expectations – make sure to read them carefully. For instance, the parameters here are formatted differently than the ones in the previous function.
- Despite the name – the function does not actually "update" a list. Remember – it's job is to create a new list, NOT change the list given to it.
- This function is a bit hard to describe, and therefore it can be a bit hard to understand. Make sure you have a sense of what this is expected to do before you start coding. A great way to do that is to review the tester code provided with this assignment, and to update a few lists by-hand.

6.5 draw_board(game_state)

The draw_board function takes one parameter, game_state is a list of numbers (similar to one that create_game_state might return) You can assume that the game_state list contains only non-negative numbers. The draw_board function has no return value, and should instead use print statements to produce an output directly to the terminal.

The output of the function is a depiction of the state of the game, for instance, if given the list `[5,2,3,1]` The following should be printed:

```
====================
1 #####
2 ##
3 ###
4 #
====================
```

Formally:

- The first line should contain 20 equal-signs.
- for a list length n, the next n lines should be the row number (1-indexed), followed by a single space, and then a number of pound-signs `#` as indicated by the game_state list.
- The final line should contain 20 equal-signs.
- No line starts-with or ends-with any spaces.
- Your output must match letter to letter – including spaces, newlines, capitalization, symbols etc.

Hints:

- A common task for this process is to repeat a string a fixed number of times. There is a simple way to do this in python. If you do not know the simple way to do this, you can also make a simple way to do that with a helper function (a non-required function you define to help you complete your task.)
- You may find it useful to know that you can use `+` to "add" strings together (concatenating them) – this is a common way to build strings in loops. Alternatively, you may find it useful to know that `print(whatever, end="")` will print whatever you want *without printing the newline character* – allowing you to build up a line of output in a loop.
- Make sure you pay special attention to the requirements on this one – we are looking for letter-to-letter equality, it's not enough for it to look correct to a human, it needs to print the exact correct series of letters.
- Remember that PDFs like to place "weird" but pretty version of characters. I STRONGLY recommend against using copy-paste as part of this function – you might accidentally copy a letter that IS NOT what we are expecting.

- Similarly, remember that PDFs like to use more, or fewer, spaces than actually included/intended. Another good reason to re-type this instead of copy/paste.

6.6 is_over(game_state)

The is_over function takes one parameter, game_state (as above). This function returns a boolean value to indicate if the game is over. If every value in the input list is 0 – the game is over. If any value is not 0 – the game is not over. (You can assume the list is not empty – I.E. it has at least one thing in it.)

7 Grading

Grading on this assignment will be 50% automated tests, and 50% manual checks. The automatic tests are effectively the same as the tests seen in the tester file.

The Manual grading will be 10% code style, and 40% spot-checks / partial credit (about 8 points per function) The spot-checking will focus on

- Bad approaches to the problem. This can take the form of correct solutions, but vastly over-complicated/inefficient, or cases where you coded to pass the tests, but failed to actually solve the problem as stated in the PDF.
- Things that make the code *anything less than a joy to read* – bad variable names, problematic code duplication (especially where this could have been avoided by calling a function). poor use of "formatting" features like newlines, whitespace, etc.
- Partial credit for code that fails tests, depending on severity of test-failure.

For code style, we will use the following style guidelines for this lab. Future labs may have a **more restrictive style guideline** (as will most workplaces).

- Every function should have a docstring (if you are not confident you know what this is, just google it!)
- Every file should have a comment at the top, which should say, at a minimum, your name and your partners name.
- Variable names should be as descriptive as possible (clearly `n` and `i` will be accepted here as common loop variables or math variables, but `c` instead of `count` would be considered a bad variable name choice)
- Reasonable use of whitespace – don't spread the lines of a function out with massive whitespace, don't use 1-space-only for indentation. That sort of thing.
- If there are problems that actively make your code hard to read, we reserve the right to mention them, even if they are not listed here. Ultimately, code is about communication, if your code doesn't make sense it's not correct (no matter how well it runs!)

8 Submission

Before submitting your work:

- Please use the tester code on canvas, and any tests you devise on your own, to test the code and carefully check that your code is bug-free. Buggy code is worth very little in the programming community – especially if the bugs prevent it from running.
- Make sure the file is named `nim.py` (case sensitive)
- Make sure the that you name, and your partner's name, are in a comment at the top of the file
- If you and your partner are submitting one copy of the assignment – make sure whoever submits it marks their partner **in gradescope itself**. Please do this unless you and your partner are turning in different files.

You are only required to submit the `nim.py` file, which will be submitted through gradescope. You can submit as many times as you want, only the final submission will be considered for grading.

If you worked with a lab partner, gradescope has a way to add your partner to your submission after you submit (to show yourself as having worked in a group). We prefer you use this feature if at all possible. Not using it may lead to trouble getting credit for your work. More information on this feature can be found here:
`https://help.gradescope.com/article/m5qz2xsnjy-student-add-group-members` If you worked with a lab partner, please make sure both you and your partner have a copy of the lab files for future reference.

If you finish this lab during the lab time, feel free to notify your lab TAs, and then leave early. If you do not finish this lab during the lab time you are responsible for finishing and submitting this assignment before the deadline.

9 More Fun

There is A LOT of room for improvement in the play_nim.py file. If you finish this lab early, here are a few ideas for more fun you can have: None of these tasks are required, or will be graded, but you can certainly complete them for bragging rights, and share your modified play_nim file on slack.

- Add turn count tracking – know how long you've played!
- Allow players to enter their names, and track whose turn it is
- (after the second) track win/loss for players 1 and 2
- add an AI (start with just random choices, then see if you can do better!)
- Allow users to enter the initial number of tokens, either the length and max, or the whole list!