6.009 CONFLICT Quiz 1

Spring 2022

Name: **Answers**

Kerberos/Athena Username:

4 questions | 1 hour and 50 minutes

- Please **WAIT** until we tell you to begin.
- This quiz is closed-book, but you may use one 8.5 × 11 sheet of paper (both sides) as a reference.
- You may **NOT** use any electronic devices (including computers, calculators, phones, etc.).
- If you have questions, please **come to us at the front** to ask them.
- Enter all answers in the boxes provided. Work on other pages with QR codes may be taken into account when assigning partial credit. **Please do not write on the QR codes.**
- If you finish the exam more than 10 minutes before the end time, please quietly bring your exam to us at the front of the room. If you finish within 10 minutes of the end time, please remain seated so as not to disturb those who are still finishing their quizzes.
- You may not discuss the details of the quiz with anyone other than course staff until final quiz grades have been assigned and released.

1 Numismatics

Consider a game in which the goal is to move a player to collect coins. As with the game from lab 2, this game is played on a 2-D grid, and on each timestep, the player can move in one of four directions: `"left"`, `"right"`, `"up"`, or `"down"`. In this game, a coin is collected when the player moves to the same spot as a coin.

We will represent the game as a dictionary containing two keys:

- `"board"`, which maps to a list of lists of lists of strings (similar to lab 2) representing the locations of the items in the game
- `"coins"`, which maps to a single integer tracking the number of coins the player has collected so far.

Here is an example game in this representation:

```
game = {
    "board": [
        [[],         [],         [],         ["coin"],   []],
        [["coin"],   ["player"], [],         [],         []],
        [["coin"],   [],         [],         [],         []]
    ],
    "coins": 0
}
```

On the following pages, we will see several pieces of code intended to create a new object called `new_game`, representing the game as updated after moving the player to the left and collecting the coin there. For each, we would like to predict what both the `game` and `new_game` objects will look like after that piece of code is run, starting with `game` defined as above.

For each attempt below, indicate the boards in both `game` and `new_game` by specifying a single letter from the second-to-last sheet of this quiz (pages 23 and 24, which you may remove), or **other** if none of the boards matches, or **exception** if the code will not run to completion but will raise an exception instead.

Also indicate the coins collected by specifying an integer.

1.1 Attempt 1

```
new_game = game
new_game["board"][1][0] = ["player"]
new_game["board"][1][1] = []
new_game["coins"] += 1
```

`game["board"]` (A-J, other, or exception): D

`game["coins"]`: 1

`new_game["board"]` (A-J, other, or exception): D

`new_game["coins"]`: 1

1.2 Attempt 2

```
new_game = {
    "board": game["board"].copy(),
    "coins": game["coins"] + 1
}
new_game["board"][1][0].append("player")
new_game["board"][1][1].remove("player")
```

`game["board"]` (A-J, other, or exception): C

`game["coins"]`: 0

`new_game["board"]` (A-J, other, or exception): C

`new_game["coins"]`: 1

1.3 Attempt 3

```
game_row = [[] for i in range(len(game["board"][0]))]
board = [game_row for i in range(len(game["board"]))]
board[0][3] = ["coin"]
board[2][0] = ["coin"]
board[1][0] = ["player"]

new_game = {
    "board": board,
    "coins": game["coins"] + 1
}
```

`game["board"]` (A-J, other, or exception): A

`game["coins"]`: 0

`new_game["board"]` (A-J, other, or exception): H

`new_game["coins"]`: 1

1.4 Attempt 4

```
board = []
for row in game['board']:
    board.append([cell[:] for cell in row])
board[0][3] = ["coin"]
board[2][0] = ["coin"]
board[1][0] = ["player"]

new_game = {}
new_game["board"] = board
new_game["coins"] = game["coins"] + 1
```

`game["board"]` (A-J, other, or exception): A

`game["coins"]`: 0

`new_game["board"]` (A-J, other, or exception): E

`new_game["coins"]`: 1

Worksheet (intentionally blank)

2 Leaps and Bounds

In this problem, we will consider planning paths through an infinite grid. We will represent each state in this search space as a tuple (r, c) where r is a row index and c is a column index for position in the grid. Because the grid is infinite, r and c can take on any integer values, $-\infty < r < \infty$ and $-\infty < c < \infty$.

A depth-first search in this domain quickly runs into problems. It searches infinitely off in one direction; if the goal point happens to be along that path, it returns a path, but if not, it will loop forever!

One strategy to fix this is to have the search only consider paths whose length (number of states, including the starting state) is less than or equal to some maximum length (call it `max_len`). When implementing this strategy, we would like our search to have the following properties:

(a) It should be guaranteed to return a path if a path of length $l \leq \mathrm{max\_len}$ exists.

(b) It should be guaranteed to terminate and return `None` if no path of length $l \leq \mathrm{max\_len}$ exists.

(c) It should be guaranteed never to add any paths of length $l > \mathrm{max\_len}$ to the agenda.

On the following pages are several implementations of DFS, all modified in an attempt to accomplish all of the goals above. For each implementation, indicate which of the properties it satisfies by putting some combination of the letters a, b, and c in each box. If a particular implementation does not have any of these properties, write `None` in the box.

For reference, here is an unmodified implementation of a depth-first search:

```
def successors(loc):
    r, c = loc
    return [(r+1, c), (r-1, c), (r, c-1), (r, c+1)]

def dfs(start, goal_point):
   agenda = [[start]]
   while len(agenda) > 0:
      current = agenda.pop(0)
      for child in successors(current[-1]):
         new = current + [child]
         if child == goal_point:
            return new
         agenda = [new] + agenda
   return None
```

Each of the searches below will indicate any lines that have been changed with a comment.

2.1 DFS A

```
def dfs(start, goal_point, max_len):
   agenda = [[start]]
   while len(agenda) > 0:
      current = agenda.pop(0)
      for child in successors(current[-1]):
         new = current + [child]
         if child == goal_point:
            return new
         if len(new) <= max_len:  # CHANGED
             agenda = [new] + agenda  #CHANGED
   return None
```

Properties: a, b, c

2.2 DFS B

```
def dfs(start, goal_point, max_len):
   agenda = [[start]]
   while len(agenda) > 0:
      current = agenda.pop(0)
      for child in successors(current[-1]):
         new = current + [child]
         if child == goal_point and len(new) < max_len:  # CHANGED
            return new
         agenda = [new] + agenda
   return None
```

Properties: None

2.3 DFS C

```
def dfs(start, goal_point, max_len):
   count = 0  # CHANGED
   agenda = [[start]]
   while len(agenda) > 0:
      count += 1  # CHANGED
      if count >= max_len:  # CHANGED
          return None  # CHANGED
      current = agenda.pop(0)
      for child in successors(current[-1]):
         new = current + [child]
         if child == goal_point:
            return new
         agenda = [new] + agenda
   return None
```

Properties: b, c

2.4 DFS D

```
def dfs(start, goal_point max_len):
   agenda = [[start]]
   while len(agenda) > 0:
      biggest = max([len(i) for i in agenda])  # CHANGED
      if biggest > max_len:  # CHANGED
          return None  # CHANGED
      current = agenda.pop(0)
      for child in successors(current[-1]):
         new = current + [child]
         if child == goal_point:
            return new
         agenda = [new] + agenda
   return None
```

Properties: b

2.5 DFS E

```
def dfs(start, goal_point, max_len):
   agenda = [[start]]
   while len(agenda) > 0:
      current = agenda.pop(0)
      if len(current) >= max_len:  # CHANGED
          continue  # CHANGED
      for child in successors(current[-1]):
         new = current + [child]
         if child == goal_point:
            return new
         agenda = [new] + agenda
   return None
```

Properties: a, b, c

Worksheet (intentionally blank)

3 Polynomials

In this problem, we will consider implementing a function called `poly_add(p1, p2)` to perform addition of two polynomials. Our inputs `p1` and `p2` each represent polynomials, but we are not told the details of the actual representation of `p1` and `p2`; rather, we are given descriptions of various helper functions. Regardless of how polynomials are represented internally, the following helper functions are available to us:

- `zero_polynomial()` returns a polynomial representing 0.
- `get_order(poly)` returns the order of the polynomial `poly`. For example, if the input represented $8 + 7x + 4x^3$, this function would return `3`. Assume that the order of the zero polynomial is -1.
- `get_coeff(poly, i)` returns the coefficient associated with the x^i term in the given polynomial. For example, if we called `get_coeff(p, 3)` where `p` represented $8 + 7x + 4x^3$, this function would return `4`; and calling `get_coeff(p, 2)` would return 0.
- `set_coeff(poly, i, val)` *mutates* the given polynomial such that the coefficient associated with x^i is replaced with `val`. For example, calling `set_coeff(p, 2, 9)` where `represented` $8 + 7x + 4x^3$ would result in the input polynomial being mutated to represent $8 + 7x + 9x^2 + 4x^3$.

In the box below, fill in your definition for the `poly_add` function. You may assume that you have access to working versions of all of the helper functions above.

```
def poly_add(p1, p2):
    out = zero_polynomial()
    for pow in range(max(get_order(p1), get_order(p2)) + 1):
        set_coeff(out, pow, get_coeff(p1, pow) + get_coeff(p2, pow))
    return out
```

As we have seen throughout 6.009, the choice of representation is an important one! Careful choice of internal representation can often allow us to write more efficient, more concise, and clearer code; and this problem is no exception! While your code on the previous page should work for *any* internal representation, we'll now try to fill in the details.

Here the choice of internal representation is up to you. You are welcome to use any combination of Python `int`, `float`, `str`, `bool`, `list`, `tuple`, `set`, `frozenset`, and/or `dict` objects; but you are not allowed to use classes (which have not yet been covered in 6.009).

In the box below, briefly describe your choice of representation, including an example representation of $8 + 7x + 4x^3$:

This is not the only possible representation, but we'll use a dictionary mapping powers to the associated coefficients. Any power not in the dictionary will be assumed to have a coefficient of 0.

In this representation, $8 + 7x + 4x^3$ would be represented as `{0:  8, 1:  7, 3:  4}`.

Then, in the box below and the box on the facing page, write code for the `zero_polynomial`, `get_order`, `get_coeff`, and `set_coeff` helper functions using that representation.

```
# your polynomial helper functions here
def zero_polynomial():
    return {}

def get_order(poly):
    return max((k for k in poly if poly[k] != 0), default=-1)
```

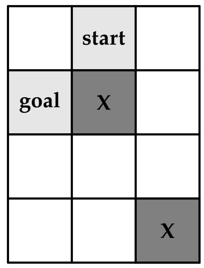
```
# your polynomial helper functions here
def get_coeff(poly, i):
    assert i >= 0 and isinstance(i, int)
    return poly.get(i, 0)

def set_coeff(poly, i, val):
    if val == 0:
        if i in poly:
            del poly[i]
    else:
        poly[i] = val
```

Worksheet (intentionally blank)

4 The Scenic Route

In 6.009 so far, we have spent a fair amount of time considering approaches to finding optimal paths through graphs. Typically, our goal has been finding the *shortest* path from one location to another in a graph. In this problem, we will instead consider the problem of finding the *longest* path connecting two locations in a graph.

Consider a grid representing a map, represented by a 2-D array (list of lists) of strings in Python. A character `'S'` represents a starting location, a character `'G'` represents a goal location, a space (`' '`) represents a location that is safe to travel, and a character `'X'` represents a location that cannot be traversed. For example, consider the following map (where the dark grey "X" cells represent locations that cannot be traversed):

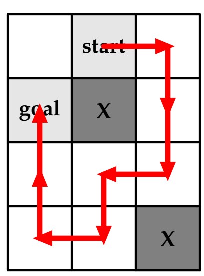


The map above is represented by this list:

```
map1 = [[' ', 'S', ' '], ['G', 'X', ' '], [' ', ' ', ' '], [' ', ' ', 'X']]
```

Our goal is to produce a function `scenic_route`, which takes a grid of the form described above as its input, and which returns the *longest* path from `'S'` to `'G'` that only contains locations that are safe to travel, and that does not contain any location more than once. Assume that you can only move in four directions: up, down, left, or right (diagonal moves are not allowed) and that you cannot move beyond the bounds of the grid. If no path exists, your function should return `None`. You may assume that the start and goal locations are always distinct from each other.

For example, in the map above, the longest such path is shown below, both graphically and as a Python object:




```
[(0, 1), (0, 2), (1, 2), (2, 2), (2, 1), (3, 1), (3, 0), (2, 0), (1, 0)]
```

Answer the questions on the following pages about this problem.

4.1 Iterative Approach

The following code, written in an iterative style, currently returns the *shortest* path from the start to the goal in the given grid (and it does so correctly); but with a few small changes, it can be made to return the longest path instead. On the facing page (page 17), please indicate which lines should be adjusted (and how they should be adjusted) to make these changes.

Make your changes by specifying at most 5 lines that you would like to replace in the code below. For each, specify the line number, and also write at most 4 lines with which you would like to replace that line (if you would like to delete a line, specify its line number and leave the associated box blank). If you would like to make fewer than 5 changes, leave the remaining boxes blank.

Correct code for the helper functions `get_neighbors` and `find_start_and_goal` can be found on the last page of this writeup, which you are free to remove.

```
01 |  def shortest_path(grid):
02 |      start, goal = find_start_and_goal(grid)
03 |      agenda = [[start]]
04 |      seen = {start}
05 |
06 |      while agenda:
07 |          path = agenda.pop(0)
08 |
09 |          for child in get_neighbors(grid, path[-1]):
10 |              if child not in seen:
11 |                  new_path = path + [child]
12 |
13 |                  if new_path[-1] == goal:
14 |                      return new_path
15 |                  else:
16 |                      agenda.append(new_path)
17 |                      seen.add(child)
18 |
19 |      return None
```

Replace line number 04 with:

```
best = None
```

Replace line number 10 with:

```
if child not in path:
```

Replace line number 14 with:

```
best = new_path
```

Replace line number 19 with:

```
return best
```

Replace line number 17 with:

```
```

4.2 Recursive Approach

We could also solve this question using a recursive approach. An outline for such an approach is given on the facing page (page 19). Fill in the various pieces to complete the program. You are welcome to make use of the two helper functions defined on the last page of this exam (`get_neighbors` and `find_start_and_goal`).

Note that `scenic_route_helper`, which is defined within `scenic_route`, should return the longest path between `start` and `goal` in the given grid. `scenic_route_helper` should call itself recursively and should make use of the result of any recursive calls to compute the overall answer

```
def scenic_route(grid):
    in_path = set()

    def scenic_route_helper(start, goal):
        if start == goal:  # base case
            return [start]

        possible = []
        in_path.add(start)

        for n in get_neighbors(grid, start):
            if n in in_path:
                continue
            test = scenic_route_helper(n, goal)
            if test is None:
                continue
            possible.append([start] + test)

        # code after the loop but within the helper function's body
        in_path.remove(start)
        return max(possible, key=len, default=None)

    s, g = find_start_and_goal(grid)
    return scenic_route_helper(s, g)
```

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Options for "Numismatics"

Option A:

```
# this is identical to the original board
[[[],         [],         [],         ["coin"],   []],
 [["coin"],   ["player"], [],         [],         []],
 [["coin"],   [],         [],         [],         []]]
```

Option B:

```
[[[],         [],         [],         ["coin"],   []],
 [["coin"],   [],         [],         [],         []],
 [["coin"],   [],         [],         [],         []]]
```

Option C:

```
[[[],                 [],         [],         ["coin"],   []],
 [["coin", "player"], [],         [],         [],         []],
 [["coin"],           [],         [],         [],         []]]
```

Option D:

```
[[[],         [],         [],         ["coin"],   []],
 [["player"], [],         [],         [],         []],
 [["coin"],   [],         [],         [],         []]]
```

Option E:

```
[[[],         [],         [],         ["coin"],   []],
 [["player"], ["player"], [],         [],         []],
 [["coin"],   [],         [],         [],         []]]
```

Option F:

```
[[[],                 [],         [],         ["coin"],   []],
 [["coin", "player"], ["player"], [],         [],         []],
 [["coin"],           [],         [],         [],         []]]
```

Option G:

```
[[["coin", "player"], ["player"], [],         [],         []],
 [["coin", "player"], ["player"], [],         [],         []],
 [["coin", "player"], ["player"], [],         [],         []]]
```

Option H:

```
[[["player"], [], [], ["coin"], []],
 [["player"], [], [], ["coin"], []],
 [["player"], [], [], ["coin"], []]]
```

Option I:

```
[[["player"], ["player"], ["player"], ["player"], ["player"]],
 [["player"], ["player"], ["player"], ["player"], ["player"]],
 [["player"], ["player"], ["player"], ["player"], ["player"]]]
```

Option J:

```
[[["player"],         ["player"], ["player"], ["coin", "player"], ["player"]],
 [["coin", "player"], ["player"], ["player"], ["player"],         ["player"]],
 [["coin", "player"], ["player"], ["player"], ["player"],         ["player"]]]
```

Helper Functions for `scenic_route`

Code for the helper functions used in question 4 can be found below. These functions are correctly implemented and should not be changed.

```
def find_start_and_goal(grid):
    """
    Given a grid (as described in problem 4), return a (start, goal) tuple,
    where start is a (row, column) tuple representing the location of the cell
    containing 'S' and goal is a (row, column) tuple representing the location
    of the cell containing 'G'
    """
    start = None
    goal = None
    for r in range(len(grid)):
        for c in range(len(grid[r])):
            if grid[r][c] == 'S':
                start = (r, c)
            elif grid[r][c] == 'G':
                goal = (r, c)
    return start, goal


def get_neighbors(grid, location):
    """
    Given a grid (as described in problem 4) and a (row, column) tuple
    representing a location in that grid, return a list of all valid (row,
    column) tuples that can be reached in a single move from the given location
    according to the rules outlined in problem 4
    """
    row, col = location
    out = []
    possible = [(row+1, col), (row-1, col), (row, col+1), (row, col-1)]
    return [
        (nr, nc)
        for nr, nc in possible
        if 0 <= nr < len(grid) and 0 <= nc < len(grid[nr])
        and grid[nr][nc] != 'X'
    ]
```

Worksheet (intentionally blank)