6.009 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 Testing and Debugging

For each of the four programs below (which may or may not be correct), consider the behavior described in its docstring, as well as the code itself. Each function is followed by a small piece of test code that prints two variables, inp and out. For each of these variables, indicate the value that will be printed. Additionally, regardless of whether the given program is correct, indicate the value of out that we would expect to be printed by a correct program.

1.1 Program 1

```
def reverse_all(inp_list):
    """
    given a list of lists, return a new list of lists but with all of the inner
    lists reversed, without mutating the given input.
    """
    output = inp_list.copy()
    for x in output:
        x = x[::-1]
    return output

# test code
inp = [[9, 8, 7], [20, 30]]
out = reverse_all(inp)
print(inp)
print(out)
```

Expected value of inp: `[[9, 8, 7], [20, 30]]`

Printed value of inp: `[[9, 8, 7], [20, 30]]`

Expected value of out: `[[7, 8, 9], [30, 20]]`

Printed value of out: `[[9, 8, 7], [20, 30]]`

1.2 Program 2

```
def flip(inp_list):
    """
    given a list, return a new list with its elements in reverse order, without
    mutating the given input.
    """
    out = inp_list.copy()
    ix = 0
    n = len(inp_list) - 1
    while ix <= n:
        out[ix] = out[n - ix]
        ix += 1
    return out

# test code
inp = [1, 2, 3, 4, 5]
out = flip(inp)
print(inp)
print(out)
```

Expected value of inp: `[1, 2, 3, 4, 5]`

Printed value of inp: `[1, 2, 3, 4, 5]`

Expected value of out: `[5, 4, 3, 2, 1]`

Printed value of out: `[5, 4, 3, 4, 5]`

1.3 Program 3

```
def sum_lists(lists):
    """
    given a list of lists, return a new list where each list is replaced by
    the sum of its elements, without mutating the given input.
    """
    output = [0] * len(lists)
    for i in range(len(lists)):
        total = 0
        for i in lists[i]:
            total += i
        output[i] = total
    return output

# test code
inp = [[1, 2], [3, 4, 5], [9, 10]]
out = sum_lists(inp)
print(inp)
print(out)
```

Expected value of inp:	`[[1, 2], [3, 4, 5], [9, 10]]`
Printed value of inp:	`exception`

Expected value of out:	`[3, 12, 19]`
Printed value of out:	`exception`

1.4 Program 4

```
def deepcopy(inp_list):
    """
    given a list containing numbers or other lists of this form (which may
    themselves contain numbers or other lists), return a "deep" copy of that
    list such that no list objects are shared between the two.
    """
    out = []
    for inner in inp_list:
        if isinstance(inner, list):
            out.append(inner[:])
        else:
            out.append(inner)
    return out

# test code
inp = [1, [2, 3], [[7]]]
out = deepcopy(inp)
out[0] = 20
out[1][0] = 30
out[2][0][0] = 40
print(inp)
print(out)
```

Expected value of inp: `[1, [2, 3], [[7]]]`

Printed value of inp: `[1, [2, 3], [[40]]]`

Expected value of out: `[20, [30, 3], [[40]]]`

Printed value of out: `[20, [30, 3], [[40]]]`

2 Path Planning

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 $-\infty < c < \infty$. Furthermore, there are no obstacles, so all locations are valid.

2.1 First Attempt

Intrigued by this problem, Ben Bitdiddle writes the `ben_search` algorithm to find a path between a start state and a goal state in this space:

```
def successors(state):
   (r,c) = state
   return [(r+i, c+j) for (i,j) in [(0,1), (1,0), (0,-1), (-1,0)]]

def ben_search(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
```

Which of the following best describes Ben's search?

BFS, DFS, or neither: $\boxed{\text{DFS}}$

Which of the following best describes the elements in the agenda in Ben's code?

(a) a boolean representing whether the terminal vertex is the goal state,

(b) a single state in the search space,

(c) a tuple containing a single state in the search space,

(d) a tuple containing a path from the start state to some other state,

(e) a list containing a single state in the search space, or

(f) a list containing a path from the start state to some other state

a, b, c, d, e, or f: $\boxed{\text{d}}$

2.2 Behaviors

Consider the following conjectures about Ben's code, and determine if there is any goal state for which the conjecture is true. In all cases, assume that the starting location is $(0, 0)$.

Conjecture 1: Ben's algorithm returns the shortest path to the goal state.

If this conjecture is true for some goal state, enter any such goal state. Otherwise, enter **None**.

goal state or None:

$(0, 1)$, $(1, 0)$, or $(0, -1)$;
or (r, c), where $r < 0$ and $c \in \{-1, 0, 1\}$

Conjecture 2: Ben's algorithm returns a path to the goal that isn't the shortest path.

If this conjecture is true for some goal state, enter any such goal state. Otherwise, enter **None**.

goal state or None:

$(0, 0)$
(or `None` if assuming `start != goal`)

Conjecture 3: Ben's algorithm runs to completion, but returns `None`.

If this conjecture is true for some goal state, enter any such goal state. Otherwise, enter **None**.

goal state or None:

None

Conjecture 4: Ben's algorithm runs indefinitely and does not return a path.

If this conjecture is true for some goal state, enter any such goal state. Otherwise, enter **None**.

goal state or None:

any state not represented above

2.3 Visited

Having sat in on 6.009 lectures 2 and 3, Ben's friend Lem E. Tweakit suggests than Ben can improve his code by keeping track of which states have been visited, so as to avoid visiting the same state more than once. Lem's code is shown below, where changes from Ben's code are marked with comments.

```
def lem_search(start, goal_point):
    agenda = [(start,)]
    visited = {start}   # CHANGED
    while len(agenda) > 0:
        current = agenda.pop(0)
        for child in successors(current[-1]):
            new = current + (child,)
            if child == goal_point:
                return new
            if child not in visited:  # CHANGED
                agenda = [new] + agenda  # CHANGED
                visited.add(child)  # CHANGED
    return None
```

Will Lem's code work substantially better, substantially worse, or about the same as Ben's code if the start state is $(0, 0)$ and the goal state is $(5, 0)$?

Better, Worse, or Same: About the Same

Briefly explain your reasoning.

This approach is good for making sure we never visit the same node twice. So in general, it is good to make use of a visited set. In this case, though, it fails to address the real issue with Ben's code.

The real issue is that Ben's code gets caught in infinite loops for many goal states (and $(5, 0)$ is one such goal state). Since Lem's code keeps encountering new states while searching (still doing DFS), the only paths we prune from the tree are those that work back toward $(0, 0)$.

However, we still end up searching infinitely and using infinite memory in the long run, so the two pieces of code are very nearly identical, functionally speaking.

One could argue that Lem's code is worse because it still gets caught in this same infinite loop, but it also takes time and memory to keep track of the `visited` set.

2.4 Improvements

Ivana de Bugyu (another friend of Ben's) tries running Ben's code on some input, but she finds that it gets caught in an infinite loop! She decides to take on the task of fixing Ben's code. Eventually, she is able to make Ben's code work to the point where she can guarantee that it will return the shortest path between any arbitrary start and goal state. What's even more impressive is that she did this by changing only a single line of Ben's original code! What line did she change, and what single line did she replace it with? Ben's code is reproduced here for convenience, labeled with line numbers.

```
01 |  def ben_search(start, goal_point):
02 |      agenda = [(start,)]
03 |      while len(agenda) > 0:
04 |          current = agenda.pop(0)
05 |          for child in successors(current[-1]):
06 |              new = current + (child,)
07 |              if child == goal_point:
08 |                  return new
09 |              agenda = [new] + agenda
10 |      return None
```

Note that there may be multiple correct answers to this question; any correct answer will suffice.

Line number: 4

Should read: `current = agenda.pop()`

Another possible answer would be to replace line 9 with `agenda.append(new)`

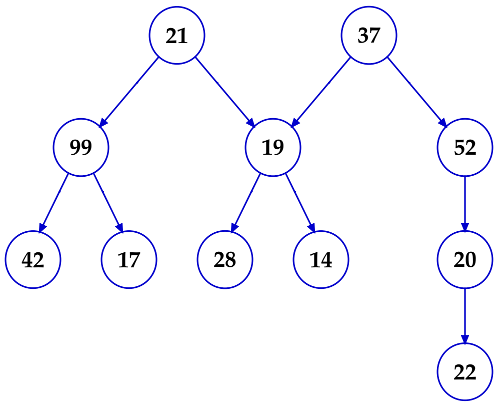
Worksheet (intentionally blank)

3 Cousins

So far in 6.009, we have spent a fair amount of time thinking about *tree* structures. In this problem, we'll explore an interesting relationship between nodes in a similar kind of graph.

For this problem, we will represent graphs as a list of 2-element tuples of the form `(1, 2)` indicating that the node with label `1` is a parent of the node with label `2`. A node may have multiple parents and/or multiple children.

For example, the following picture and code represent the same graph:



```
graph1 = [
    (21, 19), (21, 99), (37, 19), (37, 52),
    (99, 42), (99, 17), (19, 28), (19, 14),
    (52, 20), (20, 22)
]
```

We will say that node **A** is a *grandparent* of node **C** if there exists a node **B** such that **A** is a parent of **B** and **B** is a parent of **C**.

We will say that nodes **D** and **E** are *cousins* if they share at least one grandparent but do not share any parents.

Our ultimate goal for this question is to create a function `is_cousin(graph, A, B)` should return `True` if nodes `A` and `B` are cousins, or `False` if `A` and `B` are not cousins.

For example:

```
is_cousin(graph1, 28, 14) => False   # share parent
is_cousin(graph1, 28, 20) => True
is_cousin(graph1, 22, 20) => False
is_cousin(graph1, 14, 42) => True
is_cousin(graph1, 17, 20) => False   # don't share grandparent
```

On the following pages, we will implement the `is_cousin` function.

3.1 Representation

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! Consider the following possible choices of internal representation and choose one to use for solving this problem (your code on the facing page will make use of this function). Circle your choice of `representation` function below.

```
def representation(graph):
    return graph.copy()


def representation(graph):
    return [i[0] for i in graph]


def representation(graph):
    return [i[1] for i in graph]


def representation(graph):
    return [i[::-1] for i in graph]


def representation(graph):
    return set(graph)


def representation(graph):
    return {i[0] for i in graph}


def representation(graph):
    return {i[1] for i in graph}


def representation(graph):
    return {i[::-1] for i in graph}
```

```
def representation(graph):
    out = {}
    for i in graph:
        out[i[0]] = i[1]
    return out


def representation(graph):
    out = {}
    for i in graph:
        out[i[1]] = i[0]
    return out


def representation(graph):
    out = {}
    for i in graph:
        if i[0] not in out:
            out[i[0]] = {i[1]}
        else:
            out[i[0]].add(i[1])
    return out


def representation(graph):
    out = {}
    for i in graph:
        if i[1] not in out:
            out[i[1]] = {i[0]}
        else:
            out[i[1]].add(i[0])
    return out
```

The bottom-right representation is a great choice for this problem. The most common question we need to ask of the data is: what are the parents of some node? This representation makes it easy to answer that question concisely and efficiently.

3.2 Code

Complete the definition of `is_cousin` below. You may assume that the `representation` function in the box below refers to your choice from the previous section.

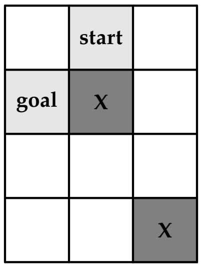
```
def is_cousin(graph, A, B):
    graph = representation(graph)
    # your code below


    def grandparents(node):
        out = set()
        for parent in graph[node]:
            out |= graph[parent]
        return out

    return (graph[A].isdisjoint(graph[B])
            and not grandparents(A).isdisjoint(grandparents(B)))
```

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):

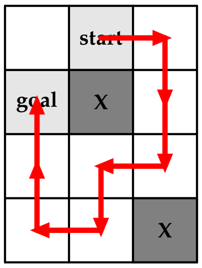


The map above is represented by this list:

```
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 length of 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:



Because this path consists of 8 moves, `scenic_route(map1)` should return `8`.

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 length of 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 length of 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_length(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 len(new_path) - 1
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:

```
if best is None or len(new_path) - 1 > best:
    best = len(new_path) - 1

# but note that the conditional here is not necessary so this could just be:
# best = len(new_path) - 1
```

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 length of 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 0

        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(test + 1)

        # code after the loop but within the helper function's body
        in_path.remove(start)
        return max(possible, 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)

Worksheet (intentionally blank)

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)