6.101 Final Exam

Spring 2024

Name:

Kerberos/Athena Username:

7 questions 3 hours

- Please **WAIT** until we tell you to begin.
- Write your name and kerberos **ONLY** on the front page.
- This exam is closed-book, and 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, but if your work is on another page, please include a clear reference to the relevant page number in the relevant box.
- **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 working.
- You may not discuss details of the exam with anyone other than course staff until exam grades have been assigned and released.

1 Exam Writing

In a scenario straining credulity, you find yourself an instructor designing a final exam for a programming class. The exam should cover all of the important topics from the semester, and it should also be doable in the time allocated for the exam.

You have a bunch of candidate questions and need to pick a subset to include in the exam. Luckily, the TAs have mastered the art of playtesting questions, and they tell you exactly how long the average student will need to complete each one. You have also labeled each question idea with which topics it covers.

Write a function that, given also the total duration available for the exam, returns a set of the names of the questions to include on the exam, such that

- the exam contains at least one question covering each topic represented in the questions list, and
- the total time of the selected questions does not exceed the scheduled duration of the exam.

It is fine to cover all topics in way less time than is scheduled for the exam. In that case, you might even win an award for educational innovation.

Your function should return `None` if it is impossible to design the exam meeting the constraints.

Here is an example bank of questions, where each is represented as a `(name, topics, duration)` tuple:

```
example_questions = [
    ("Mysteries of Cincinnati", frozenset({"recursion", "debugging"}), 5),
    ("The Lost Mountain Goat", frozenset({"classes", "environment diagrams"}), 10),
    ("Functionally Literate", frozenset({"functional programming", "debugging"}), 15),
    ("Soup's On!", frozenset({"debugging", "environment diagrams"}), 20)
]
```

Your co-instructor writes the code below to implement the `design_exam` function. This code is correct, and we will build on it on the following pages.

```
def design_exam(questions, duration):
    all_topics = get_topics(questions)

    def helper(question_list):
        if not question_list:
            return [set()]

        rest = helper(question_list[1:])
        return rest + [({question_list[0]} | i) for i in rest]

    for possible_exam in helper(questions):
        topics = get_topics(possible_exam)
        time = get_total_time(possible_exam)
        if topics == all_topics and time <= duration:
            return {name for name, _, _ in possible_exam}

    return None
```

1.1 Helpers

The code from the facing page is missing two key helper functions: `get_topics` and `get_total_time`. Fill in definitions of these functions below so that the code on the facing page behaves as expected:

```
def get_topics(questions):
    """Return a set containing all topics represented in a given list of questions."""
```

```
def get_total_time(questions):
    """Return a number representing the total time of a given list of questions."""
```

1.2 Lists

How many lists, in total, will be made by `design_exam` when it is called with each of the following input lengths, excluding any lists made by `get_topics` and `get_total_time`?

0 questions: ☐ 1 question: ☐ 2 questions: ☐ 100 questions: ☐

1.3 More Efficient

Alas, your poor co-instructor's code takes a long time to run (particularly when the number of questions is large), and the exam is right around the corner! The main efficiency issue with their code is that it starts by building up all possible exams before even considering whether they are reasonable.

Help them out by writing a new version of the code that satisfies the same goal but without explicitly creating any kind of structure (even a generator) representing all possible exams. Fill in the blanks on the facing page to complete this version of the code. You may make use of the `get_topics` and/or `get_total_time` helper functions from problem 1.1 if you wish.

Their original code is reproduced below for convenience.

```
def design_exam(questions, duration):
    all_topics = get_topics(questions)

    def helper(question_list):
        if not question_list:
            return [set()]

        rest = helper(question_list[1:])
        return rest + [({question_list[0]} | i) for i in rest]

    for possible_exam in helper(questions):
        topics = get_topics(possible_exam)
        time = get_total_time(possible_exam)
        if topics == all_topics and time <= duration:
            return {name for name, _, _ in possible_exam}

    return None
```

```
def design_exam(questions, duration):
    all_topics = get_topics(questions)

    def helper(topics, questions, duration):
        # base cases
        if duration < 0:
            
        if not topics:
            
        if not questions:
            
        # recursive case
        
    return helper(all_topics, questions, duration)
```

2 Generator Slicing

Throughout the semester, we've seen examples of using *slicing* to make collections from other collections. As a quick reminder, we can slice a list (or tuple or string) `x` using syntax like `x[start:stop:step]`, where `start`, `stop`, and `step` are integers; the result is a new object with the same type as `x`, containing only a subset of values, specifically those starting at index `start` (inclusive) and ending at `stop` (exclusive), counting by `step` each time. For example:

```
>>> x = "abcdefghi"
>>> x[0:3:1]
'abc'
>>> x[1:4:1]
'bcd'
>>> x[1:7:2]
'bdf'
```

We have often seen examples of shorthand that omits one or more of the slicing parameters, but the full form takes all three parameters.

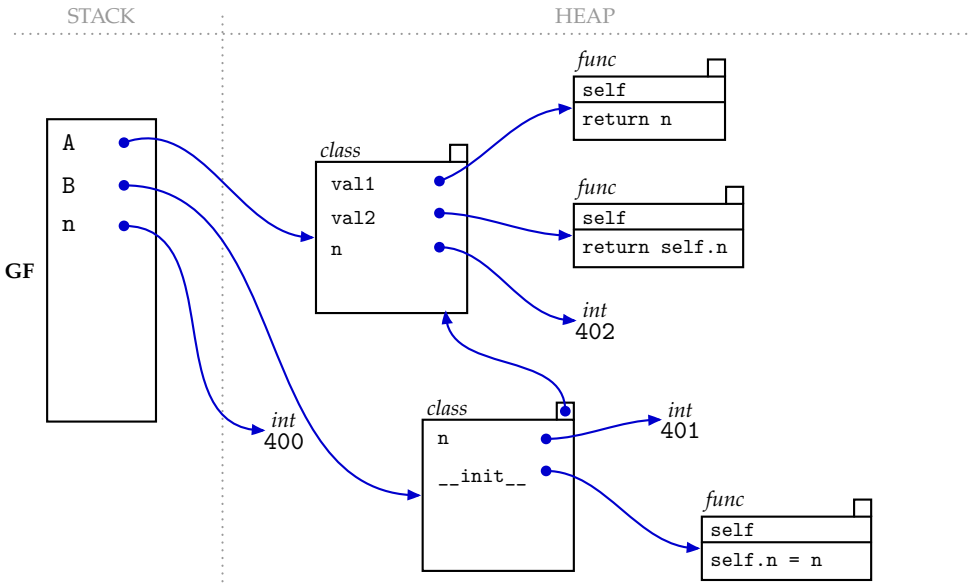
In this problem, we'll write a version of slicing that works for arbitrary iterable objects (i.e., not just for lists and strings and tuples, but for anything we can iterate over, even generators!). On the facing page, fill in the definition of `gen_slice`, which should be a generator that yields elements at the indices specified by the arguments `start`, `stop`, and `step`. For any finite iterable object `x`, we should expect `list(x)[start:stop:step]` to be equal to `list(gen_slice(x, start, stop, step))`.

For full credit, your code should not create any new container objects, and it should work for any iterable inputs (even infinitely long generators); but your code only needs to work for the case where `start` is nonnegative and `step` is positive.

```
def gen_slice(iterable, start, stop, step):
```

3 Environment Diagrams

The following diagram represents the state of a program after some code has been executed, purely in the global frame (no additional frames have been created as of this point in the program's execution). This diagram is almost complete, except that the enclosing frames of the functions are intentionally not shown.



Starting from this state, the following code is then run in the global frame:

```
n = 403

class C(A):
    def __init__(self):
        self.n = self.n
```

After this code is run, consider running the code on the facing page in the order it is specified. Each print statement in the code is followed by a box. For each, consider the value that would be printed by that print statement, and:

- If nothing would be printed because of an infinite loop (or infinite recursion), write `infinite` in the box.
- If nothing would be printed because of an exception other than infinite recursion, write `exception` in the box.
- Otherwise, write the printed value in the box.

```
a = A()
print(a.val1())
```

```
print(a.val2())
```

```
b = B()
print(b.val1())
```

```
print(b.val2())
```

```
c = C()
print(c.val1())
```

```
print(c.val2())
```

```
def foo():
    n = 404
    class D(B):
        pass
    n = 405
    return D()
d = foo()
print(d.val1())
```

```
print(d.val2())
```

```
def bar():
    n = 406
    class E(C):
        def __init__(self):
            self.n = n
    n = 407
    return E()
e = bar()
print(e.val1())
```

```
print(e.val2())
```

4 An Actual Potluck

In a recitation in week 9, we solved a problem that we called the "potluck" problem. But in reality, that wasn't much like a real potluck at all! In that problem, each person picked a single food item from our pantry; but at a real potluck, everybody brings a food item with them, and then they can eat whatever they want! So in this problem, we'll try to make a more true-to-life potluck.

We'll represent the attendees of our potluck as a dictionary mapping each person's name to a list of food items that that person is willing to bring to the potluck, like so:

```
test_people = {
    "alex": ["beans", "cheesecake"],
    "blake": ["tofu", "cheesecake"],
    "cameron": ["ketchup", "beans", "salad"],
    "dana": ["burgers", "salad"],
}
```

Write a function that takes in a dictionary of this form and returns a list of new dictionaries representing all possible combinations of food that can be brought to the party without any two people bringing the same food. Each dictionary should map food items to the person bringing that item to the party, for example:

```
>>> for i in potluck(test_people):
...     print(i)
{'burgers': 'dana', 'ketchup': 'cameron', 'tofu': 'blake', 'beans': 'alex'}
{'burgers': 'dana', 'ketchup': 'cameron', 'tofu': 'blake', 'cheesecake': 'alex'}
{'burgers': 'dana', 'ketchup': 'cameron', 'cheesecake': 'blake', 'beans': 'alex'}
{'burgers': 'dana', 'beans': 'cameron', 'tofu': 'blake', 'cheesecake': 'alex'}
{'burgers': 'dana', 'salad': 'cameron', 'tofu': 'blake', 'beans': 'alex'}
{'burgers': 'dana', 'salad': 'cameron', 'tofu': 'blake', 'cheesecake': 'alex'}
{'burgers': 'dana', 'salad': 'cameron', 'cheesecake': 'blake', 'beans': 'alex'}
{'salad': 'dana', 'ketchup': 'cameron', 'tofu': 'blake', 'beans': 'alex'}
{'salad': 'dana', 'ketchup': 'cameron', 'tofu': 'blake', 'cheesecake': 'alex'}
{'salad': 'dana', 'ketchup': 'cameron', 'cheesecake': 'blake', 'beans': 'alex'}
{'salad': 'dana', 'beans': 'cameron', 'tofu': 'blake', 'cheesecake': 'alex'}
```

For full credit, each dictionary in your answer must contain every person, with every person bringing exactly one food and no two people bringing the same food. You are welcome to define whatever additional helper functions and variables you like.

```
def potluck(people):
```

Worksheet (intentionally blank)

5 Mutable Linked Lists

We first saw linked lists in 6.101 way back in a recitation in week 8. In this problem, we'll work with linked lists again, but with two changes from the versions we saw in recitation: these linked lists will be mutable, and they will specify their elements in a different order.

We will represent a linked list as a nested list of the form `[element, rest]`, where `element` is the element **at the end of the list**, and `rest` is a linked list containing the remaining elements. We'll use `[None]` to represent an empty linked list. According to this representation:

- `[None]` represents an empty linked list;
- `[1, [None]]` represents a linked list containing only `1`; and
- `[9, [8, [7, [None]]]]` represents a linked list containing `7`, `8`, and `9`, **in that order**.

5.1 Length

Below is an implementation for a function that takes a list (of the form described above) as input and returns the length of the linked list it represents:

```
def length(L):
    if L == [None]:
        return 0
    return 1 + length(L[1])
```

It is also possible to write this function iteratively. In the box below, fill in an implementation of `length` that does not make any recursive calls.

```
def length(L):
```

5.2 Deletion

Now let's actually implement some functionality using mutation. We would like to implement deletion on linked lists, as a function called `delete`. This function should take two inputs: a linked list `L` (represented as described in the previous section), and an index whose associated element we should delete.

Your function should work by mutating the given list to remove the element at the given index, without making any new lists. We'll also do a little bit of error handling here: if the given index is not a valid index into the list, your code should raise an `IndexError`. You should consider negative indices to be invalid.

Here is a small example showing the intended behavior of this function:

```
>>> test_list = [9, [8, [7, [None]]]]
>>> delete(test_list, 0)
>>> test_list
[9, [8, [None]]]
>>> delete(test_list, 1)
>>> test_list
[8, [None]]
>>> delete(test_list, 20)
IndexError
>>> delete(test_list, -1)
IndexError
```

Fill in the implementation of `delete` in the box on the facing page. Feel free to implement your solution recursively or iteratively, and feel free to use the `length` function if you wish.

```
def delete(L, index):
```

6 Reduction Junction

In this problem, we'll look at a few functions implemented via `reduce` from the LISP lab. As a reminder of `reduce`'s behavior, here is a version designed to work on Python lists:

```
def reduce(func, input, initial):
    out = initial
    for elt in input:
        out = func(out, elt)
    return out
```

6.1 Mystery Reduction 1

```
def mystery1(func, inp):
    return reduce(lambda a, b: a + [func(b)], inp, [])
```

What is the output of `mystery1(lambda x: x > 0, [1, -2, -3, 4, 5])`?

What is the output of `mystery1(lambda x: x > -5, [1, -2, -3, 4, 5])`?

What is the output of `mystery1(lambda x: x < -5, [1, -2, -3, 4, 5])`?

This function could be recreated (without `reduce`) using a single line of Python. Fill in the box below so that `mystery1` and `mystery1a` produce the same output for all valid inputs.

```
def mystery1a(func, inp):
    return
```

6.2 Mystery Reduction 2

```
def mystery2(func, inp):
    return reduce(lambda a, b: a or func(b), inp, False)
```

What is the output of `mystery2(lambda x: x > 0, [1, -2, -3, 4, 5])`?

What is the output of `mystery2(lambda x: x > -5, [1, -2, -3, 4, 5])`?

What is the output of `mystery2(lambda x: x < -5, [1, -2, -3, 4, 5])`?

This function could be recreated (without `reduce`) using a single line of Python. Fill in the box below so that `mystery2` and `mystery2a` produce the same output for all valid inputs.

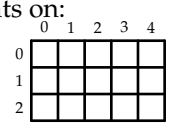
```
def mystery2a(func, inp):
    return
```

Worksheet (intentionally blank)

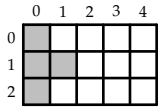
7 This Little Light of Mine

In this problem, we'll consider a version of a puzzle called "Lights Out." The puzzle is described by a 2-d grid of lights, some of which are originally on and some of which may be off. As a player, you have the ability to toggle a light from on to off (or *vice versa*), but with a catch: toggling any light changes not only that light, but also all of the adjacent lights as well. The goal of the game is to turn all of the lights off in as few moves as possible.

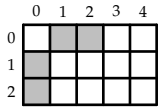
For example, consider the case of a 3-by-5 game where all of the lights are initially on. We can complete the puzzle as follows. We start with all of the lights on:



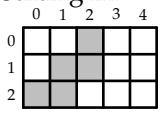
We start by toggling the light at row 1, column 0, which also toggles the lights adjacent to that spot, resulting in the following board:



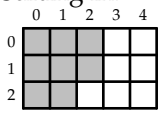
Then we toggle the light at row 0, column 1 (which toggles its adjacent lights as well). Notice that when we toggle a light that was on, it turns off; and when we toggle a light that was off, it turns on. The end result looks like:



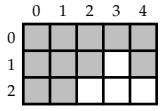
Then we toggle the light at row 1, column 1, resulting in:



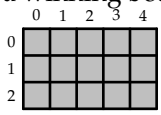
Then we toggle the light at row 0, column 0, resulting in:



Next we toggle the light at row 0, column 4:



And, finally, row 2, column 3, which results in a winning board:



This problem continues on the next page.

The code below represents an attempt to solve a "Lights Out" puzzle using the `find_path` function we know and love from 6.101's readings and recitations (the code for which has been reproduced on the last page of this exam). The docstrings below accurately describe the intended behavior of each function; and if implemented correctly, the functions described by these docstrings would correctly solve the problem.

```
00 |  def make_board(rows, cols):
01 |      out = []
02 |      for r in range(rows):
03 |          out.append([False] * cols)
04 |      return out
05 |
06 |  def solve_tile_flipping_puzzle(height, width, off_locations=None):
07 |      """
08 |      height and width are integers, specifying the number of rows and columns in
09 |      the game board, respectively, and off_locations is an optional argument, a set
10 |      of (r, c) tuples that should initially be off (all other lights are initially
11 |      on).
12 |
13 |      Returns a tuple of 2-d arrays representing the status of each light (False
14 |      means on, True means off) representing the sequence of game states leading to
15 |      the solution, or None if the puzzle cannot be solved.
16 |      """
17 |
18 |      # make an initial board (tuple of tuples), start with all False (all lights
19 |      # on) and then set some to True based on the off_locations input
20 |      board = make_board(height, width)
21 |      for r, c in off_locations:
22 |          board[r][c] = True
23 |      initial = tuple(board)
24 |
25 |      def flip_single_light(input_board, row, col):
26 |          """Return a new board representing the result of toggling a single light"""
27 |          output_board = list(input_board)
28 |          output_board[row][col] = not output_board[row][col]
29 |          return tuple(output_board)
30 |
31 |      def flip_light_and_neighbors(input_board, row, col):
32 |          """Return a new board resulting from toggling a light and its neighbors"""
33 |          out = input_board
34 |          for dr, dc in [(0,0), (-1,0), (1,0), (0,1), (0,-1)]:
35 |              flip_single_light(out, row+dr, col+dc)
36 |          return out
37 |
38 |      def neighbors(input_board):
39 |          return [flip_light_and_neighbors(input_board, r, c)
40 |                  for r in range(height) for c in range(width)]
41 |
42 |      return find_path(neighbors, initial, lambda state: all(all(r) for r in state))
```

Unfortunately, the code on the facing page contains multiple mistakes. In each box below, briefly describe one such mistake. Each description should include: the relevant line number(s), a brief description of the mistake, and a brief description of the behavior that would be exhibited as a result of that mistake (including a description of the exception that would be raised, if any).

If there are fewer mistakes than boxes, leave the remaining ones blank. If there are more mistakes than boxes, you only need to specify one error per box (but each box should describe a different kind of bug, i.e., multiple instances of the same logical error leading to the same result do not count as distinct mistakes).

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Path-finding Code

```
def find_path(neighbors_function, start_state, goal_test, bfs=True):
    """
    Return a path through a graph from a given starting state to any state that
    satisfies a given goal condition (or None if no such path exists).

    Parameters:
      * neighbors_function(state) is a function which returns a list of legal
        neighbor states
      * start_state is the starting state for the search
      * goal_test(state) is a function which returns True if the given state is
        a goal state for the search, and False otherwise.
      * bfs is a boolean (default True) that indicates whether we should run a
        bfs or dfs

    Returns:
        A path from start_state to a state satisfying goal_test(state) as a
        tuple of states, or None if no path exists.

    Note the state representation must be hashable in order for this function
    to work.
    """
    if goal_test(start_state):
        return (start_state,)

    agenda = [(start_state,)]
    visited = {start_state}

    while agenda:
        this_path = agenda.pop(0 if bfs else -1)
        terminal_state = this_path[-1]

        for neighbor_state in neighbors_function(terminal_state):
            if neighbor_state not in visited:
                new_path = this_path + (neighbor_state,)

                if goal_test(neighbor_state):
                    return new_path

                agenda.append(new_path)
                visited.add(neighbor_state)
```

Worksheet (intentionally blank)