6.009 Quiz 1: Practice Quiz A

Fall 2021

Name: **Answers**

Kerberos/Athena Username:

5 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.
- 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.

1 Efficiency

Your friend has written a function called `unique`, whose source code is shown below. They explain that this function exhibits *exactly* the behavior they expect, but it is slow for input lists bigger than a few thousand elements. They enlist your help to speed it up.

```
01 |  def unique(input_list):
02 |      output = []
03 |      seen = []
04 |      for item in input_list:
05 |          if item not in set(seen):
06 |              seen = seen + [item]
07 |              output = output + [item]
08 |      return output
```

Part a.

What are the main reasons that the code above runs so slowly on large inputs?

The main issue with the code as written above is `set(seen)` on line 5, which makes a brand new set containing all of the values we've seen so far, each time through the loop.

But just removing that would present another issue, using a list for `seen`, causing slow containment checks. So we will want just to replace `seen` with a set altogether, rather than maintaining a list and then converting to a set.

Lines 6 and 7 are also likely slow on big inputs (creating a new list each time through the loop, rather than appending to a single list).

Part b.

The code on the previous page can be made substantially more efficient (while still computing the same result) by changing a small number of lines. In the boxes below, please indicate up to 5 line numbers that you would want to change, as well as the code that you would replace those lines with, in order to improve this program's efficiency. If you want to change fewer than 5 lines, please leave the remaining boxes blank.

Replace line number 3 with the following code:

```
seen = set()
```

Replace line number 5 with the following code:

```
if item not in seen:
```

Replace line number 6 with the following code:

```
seen.add(item)
```

Replace line number 7 with the following code:

```
output.append(item)
```

Replace line number with the following code:

Worksheet (intentionally blank)

2 Cups Puzzle

Consider the following puzzle:

Standing by the side of a river, you hold a 3-liter cup and a 7-liter cup. The cups do not have markings to allow measuring smaller quantities. You need 2 liters of water. How can you measure out 2 liters of water using only these two cups?

In this particular case, one solution is as follows:

(a) Fill the 3-liter cup from the river

(b) Pour the 3 liters from the 3-liter cup into the 7-liter cup

(c) Fill the 3-liter cup from the river again

(d) Pour the 3 liters from the 3-liter cup into the 7-liter cup (which now contains 6 liters)

(e) Fill the 3-liter cup from the river again

(f) Pour from the 3-liter cup into the 7-liter cup until the latter is full. This requires 1 liter of water, leaving 2 liters in the 3-liter cup.

In this question, you will write a function to solve puzzles of this form. We will generalize things a bit, so that our puzzles will have an arbitrary number of cups, each with an arbitrary (integer) capacity.

We'll start by writing a small helper function called `pour`, which should take the following arguments:

- c_a, representing the capacity of cup a,
- q_a, representing the amount of water currently in cup a,
- c_b, representing the capacity of cup b, and
- q_b, representing the amount of water currently in cup b.

The `pour` function should return a tuple (q'_a, q'_b), representing the quantities in a and b, respectively, after pouring from a to b. Fill in the definition of `pour` below:

```
def pour(ca, qa, cb, qb):
    amount = min(qa, cb - qb)
    return qa - amount, qb + amount
```

Next we'll complete the function `solve_cups_puzzle` to solve this problem generally. `solve_cups_puzzle` takes as arguments a list of cup capacities and a desired quantity. All of the cups start out empty, and, at each point in the puzzle, we can take one of three actions:

- fill a cup from the river (all the way to the top)
- dump a cup's entire contents into the river
- pour from any cup into any other cup (either pouring all of that cup's contents or until the other cup is entirely full, whichever happens first).

The following code has been provided for you:

```
def solve_cups_puzzle(capacities, target):
    """
    >>> solve_cups_puzzle([3, 7], 2)
    [(0, 0), (3, 0), (0, 3), (3, 3), (0, 6), (3, 6), (2, 7)]
    """
    if target > max(capacities):
        return None
    return find_path(*setup_cups_puzzle(capacities, target))
```

This code makes use of two functions: `find_path`, which has been defined for you on the last page of this exam;, and `setup_cups_puzzle`, which we will fill out in the box below. Fill in the necessary pieces below (`start_state`, `goal_test`, and `successors`) so that the function above works as expected.

For full credit, your function should handle an arbitrary number of cups (and it should handle the case where there are multiple cups of the same size).

```
def setup_cups_puzzle(capacities, target):

    start_state = tuple(0 for _ in capacities)

    def goal_test(state):
        return target in state
```

```
# continued from previous page

    def successors(state):
        results = []
        for j1 in range(len(capacities)):
            # fill from river
            results.append(state[:j1] + (capacities[j1],) + state[j1+1:])

            # dump into river
            results.append(state[:j1] + (0,) + state[j1+1:])

            for j2 in range(j1+1, len(capacities)):
                # pour from j1 into j2
                q1, c1 = state[j1], capacities[j1]
                q2, c2 = state[j2], capacities[j2]
                new1, new2 = pour(c1, q1, c2, q2)
                results.append(state[:j1] + (new1,) + state[j1+1:j2] + (new2,) + state[j2+1:])

                # pour from j2 into j1
                new2, new1 = pour(c2, q2, c1, q1)
                results.append(state[:j1] + (new1,) + state[j1+1:j2] + (new2,) + state[j2+1:])

        return results


    return successors, start_state, goal_test  # last line of setup_cups_puzzle
```

3 Wordplay

Anna Graham is interested in a program that can determine whether two words are anagrams (that is, whether they contain exactly the same letters but in shuffled order). Anna has asked some classmates for help and provided each of them with the following set of test cases:

```
def test_anagram_0():
    assert is_anagram('cat', 'act')

def test_anagram_1():
    assert is_anagram('trimmed', 'midterm')

def test_anagram_2():
    assert is_anagram('deductions', 'discounted')

def test_anagram_3():
    assert not is_anagram('tar', 'rats')

def test_anagram_4():
    assert not is_anagram('stops', 'pots')

def test_anagram_5():
    assert not is_anagram('dropped', 'propped')

def test_anagram_6():
    assert not is_anagram('top', 'pots')

def test_anagram_7():
    assert not is_anagram('stops', 'top')
```

For each of the implementations on the following page, indicate which test cases that implementation would **pass** by circling the corresponding numbers. For example, if a function passes only tests 0 and 3, you should circle those numbers but leave the rest uncircled.

Code Sample A:

```
def is_anagram(w1, w2):
    count1 = {}
    for letter in w1:
        count1[letter] = count1.get(letter, 0) + 1

    count2 = {}
    for letter in w2:
        count2[letter] = count2.get(letter, 0) + 1

    return count1 == count2
```

Passing Tests:

(0) (1) (2) (3) (4) (5) (6) (7)

Code Sample B:

```
def is_anagram(w1, w2):
    if len(w1) == len(w2):
        return set(w1) == set(w2)
    return False
```

Passing Tests:

(0) (1) (2) (3) (4) 5 (6) (7)

Code Sample C:

```
def is_anagram(w1, w2):
    for letter in w1:
        if letter not in w2:
            return False
    return True
```

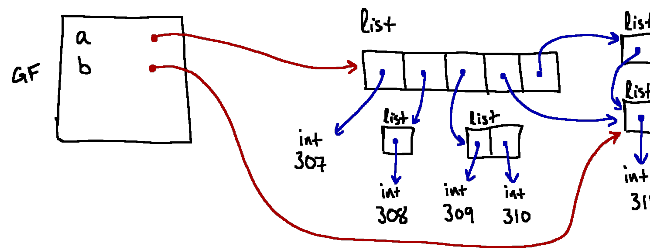
Passing Tests:

(0) (1) (2) 3 4 5 6 (7)

4 Lists

Part a.

Consider the following environment diagram, captured at some point during the execution of a program:

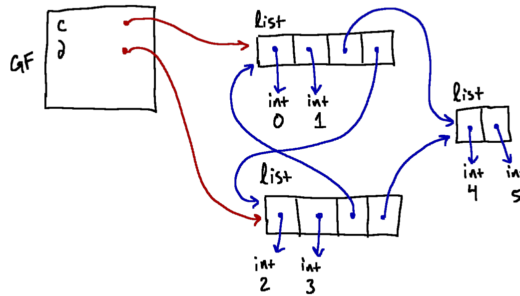


In the box below, write a small piece of code (1-4 lines) that results in the environment diagram above. If no valid Python code could produce this diagram, write `NONE` in the box instead.

```
b = [311]
a = [307, [308], [309, 310], b, [b]]
```

Part b.

Consider the following environment diagram, captured at some point during the execution of a program:



In the box below, write a small piece of code (1-4 lines) that results in the environment diagram above. If no valid Python code could produce this diagram, write `NONE` in the box instead.

```
c = [0, 1, [4, 5]]
d = [2, 3, c, c[-1]]
c.append(d)
```

Part c.

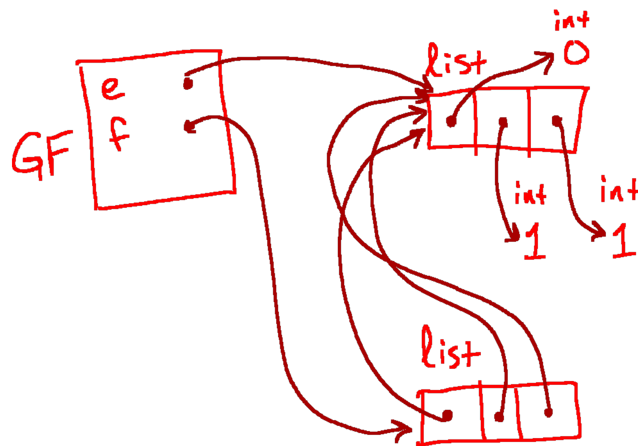
Consider the following piece of code:

```
e = [1, 1, 1]
f = [e, e, e]
f[0][0] = 0
print(e)
print(f)
```

What will be printed to the screen when this program is run?

```
[0, 1, 1]
[[0, 1, 1], [0, 1, 1], [0, 1, 1]]
```

In the space below, draw the environment diagram that would result from running the code above.



5 Flatten Tuple

Consider the following (partially complete) code for "flattening" a tuple:

```
def flatten_tuple(x):
    if not x:
        return x

    ##############################
    # TODO: ADDITIONAL CODE HERE #
    ##############################

    else:
        return flatten_tuple(x[0]) + flatten_tuple(x[1:])
```

This code is intended to take as input a tuple, which may contain any elements, including other tuples (which could contain arbitrary elements, and so on). The goal of this function is to return a new tuple containing all the non-tuple elements from the given input, but with all levels of nesting removed. For example, consider the following results:

```
>>> flatten_tuple((1, 2, 3))
(1, 2, 3)

>>> flatten_tuple((1, (2, 3)))
(1, 2, 3)

>>> flatten_tuple((1, (2, (3,))))
(1, 2, 3)

>>> flatten_tuple((1, (2, (3,), ('cat', 5, 6))))
(1, 2, 3, 'cat', 5, 6)

>>> flatten_tuple((1, (2, (3,), (((),),), ('cat', 5, 6))))
(1, 2, 3, 'cat', 5, 6)
```

On the facing page, write the code that can replace the commented-out region above in order to make this code work as expected. For full credit, try to write only a small amount of code, and make sure that the resulting function can handle tuples whose elements are of arbitrary type (not just integers and strings).

Your code to complete the `flatten_tuple` function:

```
    if not isinstance(x, tuple):
        return (x,)
```

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

find_path Function From Question 2

```
def find_path(successors, start_state, goal_test):
    agenda = [[start_state]]
    visited = {start_state}

    while agenda:
        current_path = agenda.pop(0)
        terminal_state = current_path[-1]

        if goal_test(terminal_state):
            return current_path

        for child in successors(terminal_state):
            if child in visited:
                continue
            new_path = current_path + [child]
            visited.add(child)
            agenda.append(new_path)

    return None
```

Worksheet (intentionally blank)