CS 61A Structure and Interpretation of Computer Programs
Summer 2022

Midterm Solutions

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address `<EMAILADDRESS>`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

- ◯ You must choose either this option
- ◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

- ☐ You could select this choice.
- ☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

Preliminaries

(a) What is your full name?

(b) What is your student ID number?

(c) What is your @berkeley.edu email address?

1. (7.0 points) What Would Python Display?

For each of the expressions below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines.

- If an error occurs, write `Error`, but include all output displayed before the error.
- If evaluation would run forever, write `Forever`.
- To display a function value, write `Function`.

The interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`. Assume that you have first started `python3` and executed the below statements.

```
def f(x):
    return not x

def my_pow(x, n):
    print(x, n)
    if f(n):
        return 1
    elif n < 0:
        return 1 // my_pow(x, -n)
    elif n % 2:
        return x * my_pow(x, n - 1)
    return my_pow(x * x, n // 2)


def hero(spider):
    def man(home):
        def marvel(home):
            return None
        print(spider)
        print(marvel)
        return spider - home
    return man

goat = lambda m: lambda n: m - n
bleat = (lambda a, b, c, d: b or a(d)(c))(goat, 5 == 6, 7, 4)
```

(a) **(2.0 pt)** `my_pow(2, -1)`

2 -1 2 1 2 0 0

(b) (1.0 pt) `hero(1, 2, 3)`

```
Error
```

(c) (1.0 pt) `hero(1)(2)(3)`

```
1 Function Error
```

(d) (2.0 pt) `print(1, print(4, goat(5)(4 // 2)))`

```
4 3 1 None
```

(e) (1.0 pt) `bleat`

```
-3
```

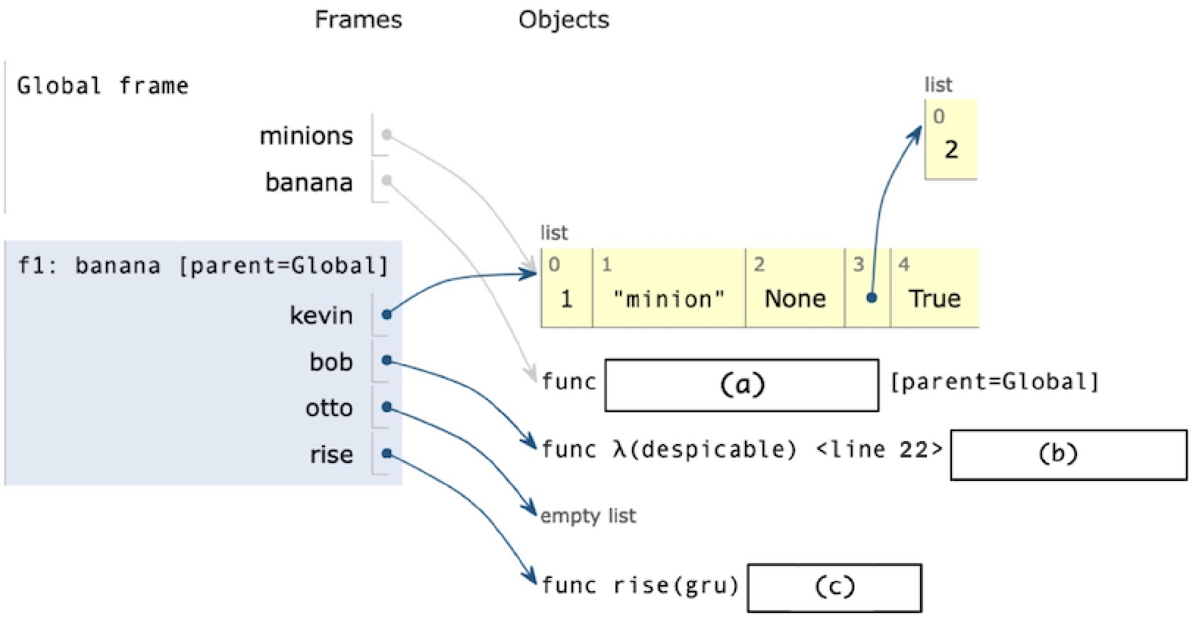
2. (6.0 points) Minions

Answer the following questions to fill in the blanks in the environment diagram, and answer what is printed when the function is run. Line numbers are included for convenience.

```
 1| minions = [1, "minion", None, [2], True]
 2|
 3| def banana(kevin, bob):
 4|     otto = []
 5|
 6|     def rise(gru):
 7|         gru.extend([gru.append([gru[1]])])
 8|         print(gru[2])
 9|         return gru[-1]
10|
11|     # STOP EXECUTION HERE FOR PART I
12|
13|     while kevin.pop():
14|         stuart = kevin.pop(0)
15|         otto.append(bob(stuart))
16|
17|     print(otto)
18|     print(minions)
19|
20|     print(rise(otto))
21|
22| banana(minions, lambda despicable: despicable * 2)
```

(a) (1.5 points) Part I

The following environment diagram shows the execution of the program until, but not including, the `while` loop beginning on line 13.



i. (0.5 pt) Fill in blank (a)

```
banana(kevin, bob)
```

ii. (0.5 pt) Fill in blank (b)

```
[parent=Global]
```

iii. (0.5 pt) Fill in blank (c)

```
[parent=f1]
```

(b) (4.0 points) Part II

Answer the following questions assuming the remaining code has been executed.

i. (1.5 pt) What will be printed to the terminal as a result of executing `print(otto)` on line 17?

```
[2, 'minionminion']
```

ii. (1.0 pt) What will be printed to the terminal as a result of executing `print(minions)` on line 18?

```
[]
```

iii. (1.0 pt) What will be printed to the terminal as a result of executing `print(gru[2])` on line 8?

```
['minionminion']
```

iv. (1.0 pt) What will be printed to the terminal as a result of executing `print(rise(otto))` on line 20?

```
None
```

3. (3.0 points) Bite-Size HOFs

(a) (1.5 points) Inverse Checker

Implement `inverse_checker`, a function that takes in two functions `f` and `g` and returns a function that returns True if `g` is the inverse function of `f` on input `n`. That is, `g` undoes the effect of `f` called on `n`.

```
def inverse_checker(f, g):
    """
    >>> checker0 = inverse_checker(lambda x: x + 1, lambda x: x - 1)
    >>> all([checker0(n) for n in range(100)])
    True
    >>> # `g` is the inverse `f`, but `f` is not the inverse of `g`
    >>> checker1 = inverse_checker(lambda x: x * 2, lambda x: x // 2)
    >>> all([checker1(n) for n in range(100)])
    True
    >>> checker2 = inverse_checker(lambda x: x ** 3, lambda x: x ** -3)
    >>> all([checker2(n) for n in range(1, 100)])
    False
    """
    def checker(n):

        return __________
                  (a)
    return __________
              (b)
```

i. (1.0 pt) Fill in blank (a)

```
return g(f(n)) == n
```

ii. (0.5 pt) Fill in blank (b)

```
checker
```

(b) (1.5 points) Force Truthy

Implement `force_truthy`, a function that takes in a function `f` and returns a function that returns the same thing as `f` when given an argument `n` such that `f(n)` outputs a truthy value, and otherwise returns `True`.

```
def force_truthy(f):
    """
    >>> truthy = force_truthy(lambda x: x // 10)
    >>> all([truthy(n) for n in range(10)])
    True
    >>> truthy(9)
    True
    >>> truthy(10)
    1
    >>> truthy(20)
    2
    """
    def truthy(n):

        return __________
                   (a)
    return __________
               (b)
```

i. (1.0 pt) Fill in blank (a)

```
f(n) or True
```

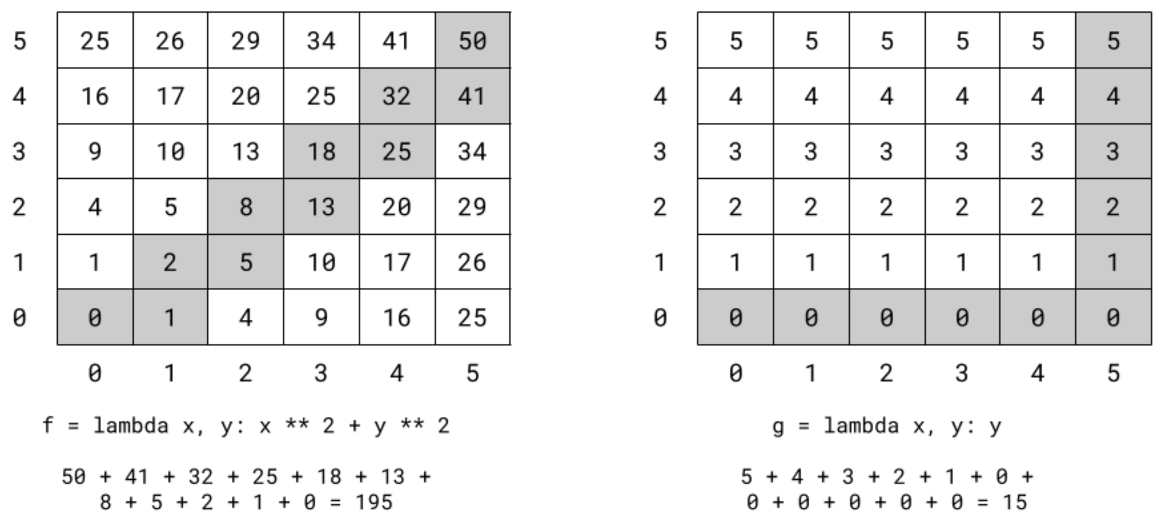
ii. (0.5 pt) Fill in blank (b)

```
truthy
```

4. (6.0 points) Least Resistance

Fill in the definition of the function `least_resistance`, which takes in three parameters, `m`, `n`, and `f`. `m` and `n` are integers which specify a coordinates position on a grid, and `f` is a two-argument function that takes in coordinates and returns a number. Your goal is to find the path of "least resistance" from the position `(m, n)` to the position `(0, 0)` on the grid, relative to `f`, which defines the resistance of each square, and return the total resistance met along that path.

A path is a series of consecutive steps from a coordinate position on the grid to `(0, 0)`, where at each step you may either take one step down, or one step to the left. The total resistance of a path is defined as the sum of `f` called on each coordinate position visited. For example, the below graphic visualizes the paths and of least resistance, and total resistance met, for the first two doctests.



Note: In the skeleton, you are provided a line that uses `float('inf')`. This will return the Python equivalent of infinity. That is, for any number `n`, `float('inf') > n` will be `True`, no matter the value of `n`.

```
def least_resistance(m, n, f):
    """
    >>> f = lambda x, y: x ** 2 + y ** 2
    >>> least_resistance(5, 5, f)
    195
    >>> g = lambda x, y: y
    >>> least_resistance(5, 5, g)
    15
    """
    if __________:
          (a)
        return __________
                  (b)
    elif __________:
            (c)
        return float('inf')
    else:
        r1 = least_resistance(______________________________)
                                           (d)
        r2 = least_resistance(______________________________)
                                           (e)
        return __________(r1, r2) + __________
                  (f)                   (g)
```

(a) **(1.0 pt)** Fill in blank (a)

```
m == 0 and n == 0
```

(b) **(0.5 pt)** Fill in blank (b)

```
f(0, 0) # or f(m, n)
```

(c) **(1.0 pt)** Fill in blank (c)

```
m < 0 or n < 0
```

(d) **(1.0 pt)** Fill in blank (d)

```
m - 1, n, f # can swap blanks (d) and (e)
```

(e) **(1.0 pt)** Fill in blank (e)

```
m, n - 1, f
```

(f) (0.5 pt) Fill in blank (f)

```
min
```

(g) (1.0 pt) Fill in blank (g)

```
f(m, n)
```

5. (7.0 points) Conditional Curry

Implement `cond_curry`, a function that takes in two functions, `f` and `cond`. `f` is a function that takes in two arguments, and `cond` is a predicate function that will take in a single argument and return either `True` or `False`. `cond_curry` returns a curried version of `f` that only "accepts" an argument `x` if calling `cond(x)` would return True. Otherwise, `x` is not accepted as an argument. Once the curried function has accepted two arguments, it will behave exactly as `f` would when called on those arguments.

```
def cond_curry(f, cond):
    """
    >>> from operator import add
    >>> curried = cond_curry(add, is_prime) # assume `is_prime` is implemented
    >>> curried(11)(13) # 11 + 13 = 24
    24
    >>> curried(10)(11)(12)(13) # 10 and 12 are not prime, and so are ignored
    24
    >>> curried(7)(4)(4)(4)(4)(4)(4)(4)(4)(7)
    14
    """
    __________:
        (a)
        __________:
            (b)
            __________:
                (c)
                return __________
                          (d)
            return __________
                      (e)
        __________:
            (f)
            return __________
                      (g)
        return __________
                  (h)
    return __________
              (i)
```

(a) (0.5 pt) Fill in blank (a)

○ `if cond(f)`

● `def g(x)`

(b) (1.0 pt) Fill in blank (b)

○ `if cond(f)`

● `def h(y)`

○ `if cond(x)`

○ `if f(x, cond)`

○ `if f(cond, x)`

(c) (0.75 pt) Fill in blank (c)

- ○ `if cond(x)`
- ● `if cond(y)`
- ○ `if cond(f)`
- ○ `def i(z)`

(d) (1.0 pt) Fill in blank (d)

- ● `f(x, y)`
- ○ `f(x, z)`
- ○ `f(y, z)`
- ○ `h`
- ○ `i`

(e) (0.75 pt) Fill in blank (e)

- ○ `f(x, y)`
- ○ `g`
- ● `h`
- ○ `i`

(f) (1.0 pt) Fill in blank (f)

- ● `if cond(x)`
- ○ `if cond(f)`
- ○ `def h(y)`

(g) (0.75 pt) Fill in blank (g)

- ○ `f(x, y)`
- ○ `cond(x)`
- ○ `cond(y)`
- ● `h`
- ○ `g`

(h) (0.75 pt) Fill in blank (h)

- ● `g`
- ○ `h`
- ○ `f(x, cond)`
- ○ `cond(x)`

(i) (0.5 pt) Fill in blank (i)

○ `cond(f)`

● `g`

6. (9.0 points) Blob Sum

(a) (2.0 points) Count Digits

Implement `count_digits`, a function that takes in a number `n` and returns the number of digits `n` contains. You should treat the number 0 as having no digits.

```
def count_digits(n):
    """
    >>> count_digits(0) # 0 has no digits
    0
    >>> count_digits(618)
    3
    >>> count_digits(2022)
    4
    """
    if __________:
          (a)
        return __________
                   (b)
    return __________
               (c)
```

i. (0.5 pt) Fill in blank (a)

```
n == 0
```

ii. (0.5 pt) Fill in blank (b)

```
0
```

iii. (1.0 pt) Fill in blank (c)

```
1 + count_digits(n // 10)
```

(b) (7.0 points) Blob Sum

Implement `blob_sum`, a function that takes in two positive integers, `n` and `k`, and returns `True` if there exists a way to add together the digits of `n` to equal `k`, where every digit of `n` is used exactly once. However, in `blob_sum`, multiple consecutive digits can be considered as a single multi-digit number (a blob), or as multiple individual digits.

Digits are read left-to-right. For example, `123` can `blob_sum` to `15` (`= 12 + 3`) but not `33` (`= 1 + 32`).

You may assume `count_digits` is implemented correctly.

```
def blob_sum(n, k):
    """
    >>> blob_sum(123, 15) # 12 + 3 = 15
    True
    >>> blob_sum(123, 6) # 1 + 2 + 3 = 6
    True
    >>> blob_sum(123, 33) # digits of `n` must be read left-to-right
    False
    >>> blob_sum(123, 24) # 1 + 23 = 24
    True
    >>> blob_sum(123, 12) # every digit of `n` must be used
    False
    >>> blob_sum(123, 35) # every digit of `n` can only be used once
    False
    """
    def helper(n, k, blob):
        if __________:
              (a)
            return __________
                       (b)
        if __________:
              (c)
            return False
        rest, last = __________
                        (d)
        new_blob = __________
                       (e)
        return __________
                   (f)
    return helper(______________________________)
                                 (g)
```

i. (0.5 pt) Fill in blank (a)

```
n == 0
```

ii. (1.0 pt) Fill in blank (b)

```
k == blob
```

iii. (0.5 pt) Fill in blank (c)

```
k <= 0
```

iv. (0.5 pt) Fill in blank (d)

```
n // 10, n % 10
```

v. (2.0 pt) Fill in blank (e)

```
blob + (last * (10 ** count_digits(blob)))
```

vi. (2.0 pt) Fill in blank (f)

```
helper(rest, k - new_blob, 0) or helper(rest, k, new_blob)
```

vii. (0.5 pt) Fill in blank (g)

```
n, k, 0
```

7. (6.0 points) Even Out

Fill in the definition for the function `even_out`. `even_out` takes in two parameters: `lst`, which is a list containing only the numbers `1` and `0` as elements, and `d`, which is a non-negative integer. `even_out` mutates `lst` such that exactly `d` instances of `0` occur between each instance of `1`. It also returns two values: the number of zeros it had to add to accomplish this, and the number of zeros it had to remove. You may assume that the first and last elements of `lst` will always be `1`.

```
def even_out(lst, d):
    """
    >>> lst = [1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1]
    >>> a, r = even_out(lst, 2)
    >>> lst
    [1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1]
    >>> a
    3
    >>> r
    3
    >>> a, r = even_out(lst, 0)
    >>> lst
    [1, 1, 1, 1, 1]
    >>> a
    0
    >>> r
    8
    """
    i = 0
    count = d
    added, removed = 0, 0
    while i < len(lst):
        if __________:
              (a)
            i += 1
            count = 0
        elif __________:
                (b)
            __________
               (c)
            i += 1
            count += 1
            added += 1
        elif __________:
                (d)
            __________
               (e)
            removed += 1
        else:
            i += 1
            count += 1
    return added, removed
```

(a) (1.0 pt) Fill in blank (a)

`lst[i] == 1 and count == d`

(b) **(1.0 pt)** Fill in blank (b)

```
lst[i] == 1 and count < d
```

(c) **(1.5 pt)** Fill in blank (c)

```
lst.insert(i, 0)
```

(d) **(1.0 pt)** Fill in blank (d)

```
count == d
```

(e) **(1.5 pt)** Fill in blank (e)

```
lst.pop(i)
```

8. (20.0 points) Boba Cafe OOPerations

(a) (9.0 points) Part I: Makin' Drinks

Tyler and Chris want to open a new restaurant, but they disagree on what style of restaurant would be most successful in Berkeley. Tyler believes that a boba restaurant will attract more sales, while Chris argues that a cafe will attract more customers. As a compromise, they've decided to open a combined boba cafe that will sell a variety of different drinks and food.

Tyler and Chris have decided to represent their restaurant's sales with different Food classes in order to track their sales.

```
class Item:

    all_items = {}

    def __init__(self, name, cost, in_stock=True):
        self.name = name
        self.cost = cost
        self.in_stock = in_stock
```

Complete the implementations for the class Boba and the class Coffee. Boba and Coffee are items in the restaurant. Whenever a new item is created, that item's name should be added to the class attribute dictionary `all_items` with a value of 1 in the class `Item`. If the name is already in the dictionary, the value should increase by 1. `Boba` should have a list `all_boba` that stores all the Boba items that have been sold. `Coffee` should have a list `all_coffee` that stores all Coffee items that have been sold.

Boba should have an additional attribute called `topping` that keeps track of what topping that drink had, if any.

Coffee should have an additional attribute called `temp` that defaults to 'hot' and a method called `add_ice` that sets the temp to 'cold' when it is called.

```
class Boba(_____):
          (a)
    all_boba = ________
                  (b)

    def __init__(self, name, cost, in_stock, topping):

        ________________________
                   (c)
        __________ = _________
            (d)          (e)
        ________________________
                   (f)
        if ___________ not in ___________________:
                (g)                     (h)
            _________________[_________] = 1
                    (i)           (j)
        else:
            _________________[_________] += 1
                    (i)           (j)


class Coffee(_____):
            (a)
    all_coffee = ________
                    (b)

    def __init__(self, name, cost, in_stock):

        ________________________
                   (c)
        _________ = 'hot'
            (k)
        ________________________
                   (l)
        if ___________ not in ___________________:
                (g)                     (h)
            _________________[_________] = 1
                    (i)           (j)
        else:
            _________________[_________] += 1
                    (i)           (j)

    def add_ice(__________):
                    (m)
        __________ = 'cold'
            (n)
```

i. (0.5 pt) Fill in blank (a)

Item

ii. **(0.5 pt)** Fill in blanks labeled (b)

```
[]
```

iii. **(1.5 pt)** Fill in blanks labeled (c)

```
super().__init__(name, cost, in_stock)
```

iv. **(0.5 pt)** Fill in blank (d)

```
self.topping
```

v. **(0.5 pt)** Fill in blank (e)

```
topping
```

vi. **(0.75 pt)** Fill in blank (f)

```
Boba.all_boba.append(self)
```

vii. **(0.5 pt)** Fill in blanks labeled (g)

```
self.name
```

viii. **(1.0 pt)** Fill in blanks labeled (h)

```
super().all_items
```

ix. **(0.5 pt)** Fill in blanks labeled (i)

```
super().all_items
```

x. **(0.5 pt)** Fill in blanks labeled (j)

```
self.name
```

xi. (0.5 pt) Fill in blank (k)

```
self.temp
```

xii. (0.75 pt) Fill in blank (l)

```
Coffee.all_coffee.append(self)
```

xiii. (0.5 pt) Fill in blank (m)

```
self
```

xiv. (0.5 pt) Fill in blank (n)

```
self.temp
```

(b) (5.0 points) Part II: Who Wins?

Tyler is feeling a bit competitive still and wants to prove to Chris that the boba part of their restaurant is the more successful part. Write a method called `more_sold` that returns whether more Boba or more Coffee has been sold. Remember Item stores a dictionary with keys as names of drinks and values as the number of that drink sold. If sales are equal, then make sure neither Tyler nor Chris think they are selling more than the other.

```
def more_sold():
        """
        >>> bubble_tea = Boba('Bubble Tea', 4, True, 'Tapioca pearls')
        >>> black_coffee = Coffee('Black', 2, True)
        >>> latte = Coffee('Latte', 4, True)
        >>> Item.more_sold()
        'Coffee'
        """
        boba_names = [__________ for boba in ___________________]
                         (a)                          (b)
        coffee_names = [__________ for coffee in ___________________]
                           (c)                            (d)
        boba_total = [________________ for name in _________________________________]
                            (e)                                   (f)
        coffee_total = [________________ for name in _________________________________]
                              (e)                                   (g)
        if ____________________:
                    (h)
            return 'Boba'
        elif ____________________:
                     (i)
            return 'Coffee'
        else:
            return 'Neither'
```

i. (0.5 pt) Fill in blank (a)

```
boba.name
```

ii. (0.5 pt) Fill in blank (b)

```
Boba.all_boba
```

iii. (0.5 pt) Fill in blank (c)

```
coffee.name
```

iv. (0.5 pt) Fill in blank (d)

```
Coffee.all_coffee
```

v. (0.5 pt) Fill in blanks labeled (e)

```
Item.all_items[name]
```

vi. (0.75 pt) Which of these could fill blanks labeled (f)? Check all that apply.

- [x] `Item.all_items.keys() if name in boba_names`
- [ ] `Item.all_items.values() if name in boba_names`
- [ ] `Item.all_items.items() if name in boba_names`
- [ ] `Item.all_items if name`
- [ ] `Item.all_items if boba_names`
- [x] `Item.all_items if name in boba_names`
- [ ] `Item.all_items[]`
- [x] `boba_names`
- [ ] `boba_names if name in coffee_names`
- [ ] `name`

vii. (0.75 pt) Which of these could fill blanks labeled (g)? Check all that apply.

- [x] `Item.all_items.keys() if name in coffee_names`
- [ ] `Item.all_items.values() if name in coffee_names`
- [ ] `Item.all_items.items() if name in coffee_names`
- [ ] `Item.all_items if name`
- [ ] `Item.all_items if coffee_names`
- [x] `Item.all_items if name in coffee_names`
- [ ] `Item.all_items[]`
- [x] `coffee_names`
- [ ] `coffee_names if name in boba_names`
- [ ] `name`

viii. (0.5 pt) Fill in blank (h)

```
sum(boba_total) > sum(coffee_total) # or len(Boba.all_boba) >
len(Coffee.all_coffee)
```

ix. (0.5 pt) Fill in blank (i)

```
sum(coffee_total) > sum(boba_total) # or len(Coffee.all_coffee) >
len(Boba.all_boba)
```

(c) (6.0 points) Part III: Optimize Cost

This question was removed.

9. (1.0 points) Extra Credit

Here are three questions about lecture. You must get all three correct to earn one point of extra credit.

(a) Songs from which artist were put into Richard's playlist in his Sequences lecture (lecture 8)?

Olivia Rodrigo

(b) Which algorithm was demoed in Laryn's Recursion lecture (lecture 6) as a method of verifying credit card numbers?

Luhn Algorithm

(c) What did Cooper say was Richard's favorite Taylor Swift album in the Objects lecture (lecture 10)?

Red (Taylor's Version)

No more questions.