CS 61A Structure and Interpretation of Computer Programs
Summer 2021

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

You can complete and submit these questions before the exam starts.

(a) What is your full name?

(b) What is your student ID number?

(c) By writing my name below, I pledge on my honor that I will abide by the rules of this exam and will neither give nor receive assistance. I understand that doing otherwise would be a disservice to my classmates, dishonor me, and could result in me failing the class.

1. Breath of the Environment

(a) The following environment diagram was generated by a program:

cY9uRs3.png

Click here to open the diagram in a new window

In this series of questions, you'll fill in the blanks of the program that follows so that its execution matches the environment diagram.

```
def breath(of, the):
    def wild(daruk):
        of.extend(______(a)______)
        ____(b)_____(____(c)_______)
    return wild


daruk = [0, 1]
zelda = [3, daruk]
breath(zelda, _________(d)_________)(3)
```

i. Which of these could fill in blank (a)? **Select all that apply!**

- [x] `zelda[1]`
- [ ] `daruk`
- [ ] `daruk[1]`
- [x] `of[1]`
- [ ] `of[1:]`
- [ ] `[1, 3]`
- [ ] `zelda`

ii. Which of these could fill in blank (b)?

- ( ) `breath`
- ( ) `(lambda x: zelda[1].extend([daruk]))`
- ( ) `daruk`
- ( ) `wild`
- (●) `the`
- ( ) `zelda`
- ( ) `(lambda x: x)(3)`

iii. Which of these could fill in blank (c)? **Select all that apply!**

- [x] `3`
- [ ] `(lambda x: x)(3)`
- [x] `zelda[0]`
- [ ] `zelda.pop(0)`
- [ ] `zelda.append(3)`
- [x] `of[0]`
- [x] `daruk`

iv. Which of these could fill in blank (d)?

- ( ) `lambda x: None`
- ( ) `[z + 1 for z in daruk]`
- (●) `lambda x: zelda[1].append(x)`
- ( ) `lambda x: zelda[1].extend([daruk])`
- ( ) `the(daruk)`
- ( ) `lambda x: zelda[1] + [3]`

2. LIST-en Up!

(a) Order in the Court!

Write a function `order_order` which takes a non-empty list of 2-argument functions `operators` and returns a new 2-argument function. When called, this returned function should print the result of applying the first function in operators to the two parameters passed in. It should then return another function that applies the second function in `operators` to the parameters, and so on. When the returned function has called the last function in the `operators` list, it should cycle back to the beginning of the list and use the first function again on the next call.

See the doctest for an example.

```
def order_order(operators):
    """
    >>> from operator import add, mul, sub
    >>> ops = [add, mul, sub]
    >>> order = order_order(ops)
    >>> order = order(1, 2) # applies add and returns mul
    3
    >>> order = order(1, 2) # applies mul and returns sub
    2
    >>> order = order(1, 2) # applies sub and cycles back to return add
    -1
    >>> order = order(1, 2) # cycles back to applying add
    3
    >>> order = order(1, 2)
    2
    >>> order = order(1, 2)
    -1
    """
    def apply(x, y):
        print(________________)
        #           (a)
        return order_order(________________)
        #                       (b)
    return ___________________
    #              (c)
```

i. What line of code could go in blank (a)?

operators [0] (x, y)

ii. What line of code could go in blank (b)?

operators[1:] + [operators[0]]

iii. What line of code could go in blank (c)?

apply

(b) Skipping Around

A skip list is defined as a sublist of a list such that each element in the sublist is non adjacent in the original list. For original list `[5, 6, 8, 2]`, the lists `[5, 8]`, `[5, 2]`, `[6, 2]`, `[5]`, `[6]`, `[8]`, `[2]`, `[]` are all skip lists of the original list. The empty list is always a skip list of any list.

Given a list `int_lst` of unique integers, return a list of all unique skip lists of `int_lst` where **each skip list contains integers in strictly increasing order**. The order in which the skip lists are returned does not matter.

```
def list_skipper(int_lst):
    """
    >>> list_skipper([5,6,8,2])
    [[5, 8], [5], [6], [8], [2], []]
    >>> list_skipper([1,2,3,4,5])
    [[1, 3, 5], [1, 3], [1, 4], [1, 5], [1], [2, 4], [2, 5], [2], [3, 5], [3], [4], [5], []]
    >>> list_skipper([])
    [[]]
    """
    if len(int_lst) == 0:
        return ________________
                    #  (a)
    with_first = ____________________
                       #  (b)
    without_first = ____________________
                         #   (c)
    with_first = [ _________ for x in with_first if x == [] or _____________]
                    # (d)                                            (e)
    return with_first + without_first
```

i. What line of code could go in blank (a)?

[[]]

ii. Which of these could fill in blank (b)?

- ○ `list_skipper(int_lst)`
- ○ `list_skipper(int_lst[1:])`
- ● `list_skipper(int_lst[2:])`
- ○ `list_skipper(int_lst[0])`
- ○ `list_skipper(int_lst[:-1])`

iii. What line of code could go in blank (c)?

list_skipper(int_lst[1:])

iv. Which of these could fill in blank (d)?

- ○ `x`
- ○ `x[0]`
- ○ `[x[0]] + int_lst`
- ● `[int_lst[0]] + x`
- ○ `int_lst.append(x)`
- ○ `x.append(int_lst)`

v. Which of these could fill in blank (e)?

- ○ `x[0] < int_lst[0]`
- ○ `x < int_lst[0]`
- ● `x[0] > int_lst[0]`
- ○ `x > int_lst[0]`
- ○ `x[0] == int_lst[1]`
- ○ `len(x) < len(int_lst)`

3. Growing Up

(a) Examine the `mystery1` function below which takes in a single positive integer parameter `n`:

```
def mystery1(n):
    i = 1
    total = 0
    while i <= n:
        if len([j for j in range(2, n + 1) if i % j == 0]) > 1:
            total = total + 1
        i = i + 1
    return total
```

i. Select the option indicating the *worst-case* order of growth of the runtime of `mystery1`.

- ○ Constant: $O(1)$
- ○ Logarithmic: $O(log_2 n)$
- ○ Linear: $O(n)$
- ● Quadratic: $O(n^2)$
- ○ Exponential: $O(2^n)$

(b) Assume that there exists a `mystery2` function which takes in a single positive integer parameter `n`. The below graph compares `n` to the *worst-case* time it takes to evaluate `mystery2(n)`.

ETKI5xE.png

You may assume that `mystery2(n)`'s runtime follows a similar trend for all values of `n`.

i. Select the option best matching the *worst-case* order of growth of the runtime of `mystery2`.

- ○ Constant: $O(1)$
- ○ Logarithmic: $O(log_2 n)$
- ● Linear: $O(n)$
- ○ Quadratic: $O(n^2)$
- ○ Exponential: $O(2^n)$

4. Maximum Exponen-tree-ation

(a) For a list of integers `values`, define the **maximum exponentiation** of `values` to be the largest single number that can be achieved by repeatedly combining pairs of consecutive elements from `values` through exponentiation. Two possible exponentiations for the list `[3, 1, 2, 3]` are shown below:

```
(3 ** 1) ** (2 ** 3) = 6561
(3 ** (1 ** 2)) ** 3 = 27
```

Of all the exponentiations of `[3, 1, 2, 3]`, the maximum exponentiation is `6561`, which can be calculated through the below steps:

```
[3, 1, 2, 3]
[3 ** 1, 2, 3]
[3, 2, 3]
[3, 2 ** 3]
[3, 8]
[3 ** 8]
6561
```

Implement `exp_tree`, which accepts a non-empty list of integers `values` and returns a tree whose label is the maximum exponentiation of `values`. Given that the maximum exponentiation is `b ** n` for some two integers `b` and `n`, the two branches of this tree should be the `exp_tree`s of the sublists of `values` used to exponentiate to `b` and `n`, respectively. You may assume that `values` has a unique exponentiation tree.

For example, below is a drawing of the `exp_tree` for `values = [3, 1, 2, 3]`:

6ObE5ae.png

The function should assume the standard CS61A Tree ADT definition, which is already loaded into code.cs61a.org, and is viewable here: code.cs61a.org/tree_adt

Note: Exponentials in Python are expressed as `base ** exponent`.

```
def exp_tree(values):
    """
    Returns the exponential tree that can be made from VALUES
    with the greatest possible root label

    >>> print_tree(exp_tree([5]))
    5
    >>> print_tree(exp_tree([3, 2]))
    9
      3
      2
    >>> print_tree(exp_tree([2, 3, 2]))
    512
      2
      9
        3
        2
    >>> lst = [3, 1, 2, 3]
```

```
>>> print_tree(exp_tree(lst))
6561
  3
    3
    1
  8
    2
    3
"""
if __________:
    #  (a)
   return __________
            # (b)
else:
    def tree_at_split(i):
        base = exp_tree(__________)
                         #  (c)
        exponent = exp_tree(__________)
                            #  (d)
        return tree(__________, [base, exponent])
                    #  (e)
    trees = [tree_at_split(i) for i in range(1, len(values))]
    return max(trees, key=__________)
                          #  (f)
```

i. Which of these could fill in blank (a)?

- ○ `len(values) == 0`
- ● `len(values) == 1`
- ○ `values`
- ○ `len(values) == 2`
- ○ `values[0] < values[1]`
- ○ `len(values) <= 2`
- ○ `len(values) <= 3`

ii. Which of these could fill in blank (b)?

- ○ `0`
- ○ `tree(0)`
- ○ `values[0]`
- ● `tree(values[0])`
- ○ `values[1]`
- ○ `tree(values[1])`
- ○ `values[0] ** values[1]`
- ○ `tree(values[0] ** values[1])`
- ○ `values[1] ** values[0]`
- ○ `tree(values[1] ** values[0])`

iii. What line of code could go in blank (c)?

values[:i]

iv. What line of code could go in blank (d)?

values[i:]

v. What line of code could go in blank (e)?

label(base) ** label(exponent)

vi. What line of code could go in blank (f)?

label

5. You've Seen it All

(a) Wait 'til You See What's in Store

Implement `memory_store`, a function that will return two functions: `add_val` and `times_seen`. When called on a number `var1`, `times_seen` will return the number of times `add_val` has been called on that particular value `var1`.

```
def memory_store():
    """
    >>> add_val, times_seen = memory_store()
    >>> add_val(4)
    >>> for _ in range(3):
    ...     add_val(3)
    >>> times_seen(3)
    3
    >>> times_seen(4)
    1
    >>> times_seen(2)
    0
    >>> add_val(3)
    >>> times_seen(3)
    4
    """
    memory = {}
    def add_val(var1):
        if var1 in memory:
            ----------------
            #      (a)
        else:
            ------------------
            #      (b)
    def times_seen(var1):
        return _________________
        #             (c)
    return add_val, times_seen
```

i. Which of these could fill in blank (a)?

- ○ `memory[var1] = 0`
- ○ `memory[var1] = var1`
- ○ `memory.append(var1)`
- ● `memory[var1] = memory[var1] + 1`
- ○ `return memory[var1]`
- ○ `return memory[0]`
- ○ `return memory.get(var1, 0)`

ii. What line of code could go in blank (b)?

memory[var1] = 1

iii. What line of code could go in blank (c)?

memory.get(var1, 0)

(b) XKSeenD

Definition: A repeatable function is a function that returns another repeatable function, which also returns a repeatable function, and so on infinitely.

Implement the function `add_seen_d`, which takes in an integer `d` and returns a repeatable function `inner`.

When `inner` is called on an integer `innervar`, it prints the sum of all integers that have been passed to `inner` at least `d` times so far among the repeated calls.

You will need to implement the helper function `make_seen_function`, which returns a seen function. A seen function is a function that takes in a value and returns the number of times the number has been seen already.

You may not use lists, dictionaries, or the function defined in 4.1 in this part.

```
initial_seen = lambda x: 0
def add_seen_d(d, seen_fn=initial_seen, total=0):
    """
    >>> a = add_seen_d(2)
    >>> a = a(3)
    0
    >>> a = a(4)
    0
    >>> a = a(3) # 3 seen for the 2nd time, thus added
    3
    >>> a = a(3) # 3 seen for the 3rd time, and won't be added again
    3
    >>> a = a(4) # 4 seen for the 2nd time, thus added
    7
    >>> a = a(5) # 5 is seen for the 1st time, so not added
    7
    """
    def inner(innervar):
        new_seen = __________________________
                           #  (a)
        if new_seen(innervar) == d:
            print(__________________________)
                       #  (b)
            return __________________________
                         #  (c)
        else:
            print(__________________________)
                        #  (d)
            return add_seen_d(d, new_seen, total)
    return inner

def make_seen_function(f, msvar):
    def seen(seenvar):
        if msvar == seenvar:
            return __________________________
                      #  (e)
        return __________________________
                  #  (f)
    return seen
```

i. Which of these could fill in blank (a)?

- ○ `seen_fn`
- ○ `lambda x: seen_fn(x) + 1`
- ○ `make_seen_function(seen_fn, d)`
- ● `make_seen_function(seen_fn, innervar)`
- ○ `make_seen_function(lambda x: seen_fn(x) + 1, d)`
- ○ `make_seen_function(lambda x: seen_fn(x) + 1, innervar)`

ii. What line of code could go in blank (b)?

(You may not use lists, dictionaries, or the function defined in 4.1 in this part)

total + innervar

iii. What line of code could go in blank (c)?

(You may not use lists, dictionaries, or the function defined in 4.1 in this part)

add_seen_d(d, new_seen, total + innervar)

iv. What line of code could go in blank (d)?

(You may not use lists, dictionaries, or the function defined in 4.1 in this part)

total

v. Which of these could fill in blank (e)? **Select all that apply!**

- ☐ `f(msvar)`
- ☐ `f(seenvar)`
- ■ `1 + f(msvar)`
- ■ `1 + f(seenvar)`
- ☐ `f(msvar) + f(seenvar)`
- ☐ `f(seenvar) - msvar`

vi. What line of code could go in blank (f)?

(You may not use lists, dictionaries, or the function defined in 4.1 in this part)

f(seenvar)

No more questions.