Welcome to Exam Prep 5!

not sponsored :)

→ We'll start at Berkeley time (2:10 pacific time)

→ Worksheet can be found in the Drive folder: links.cs61a.org/exam-prep

→ Music: Django Reinhardt
  → Minor Swing
  → Beyond the Sea
  → Brazil

→ Feel free to ask questions in the meantime!

CS 61A Fall 2020

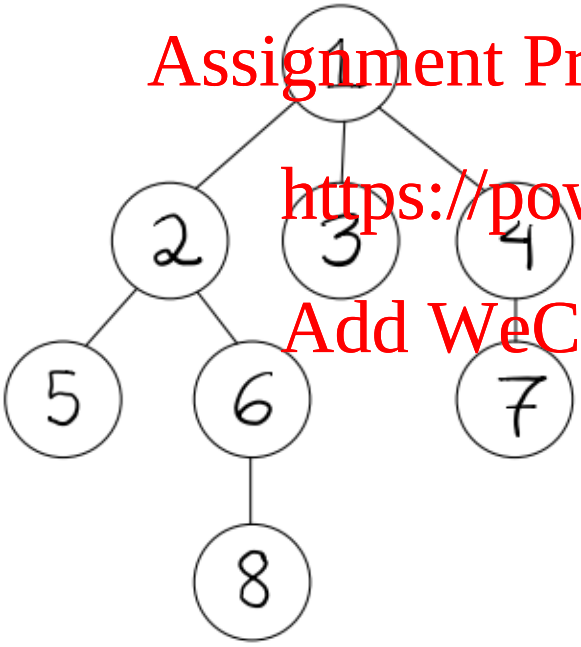
Structure and Interpretation of Computer Programs

Quiz 5

INSTRUCTIONS

- Please review this worksheet before the exam prep session. Coming prepared will help greatly, as the TA will be live solving without allocating much time for individual work.
- Either Sean or Derek will be on video live solving these questions. The other TA will be answering questions in the chat. It is in your best interest to come prepared with **specific** questions.
- This is not graded, and you do not need to turn this in to anyone.

Below is a tree, which will be referred to as `t1` in future questions.



1. **Tree Printer**

(a) Your friend wants to print out all of the values in some trees. Based on your experience in CS 61A, you decide to come up with an unnecessarily complicated solution. You will provide them with a function which takes in a tree and returns a *node-printing function*. When you call a node-printing function, it prints out the label of one node in the tree. Each time you call the function it will print the label of a different node. You may assume that your friend is polite and will not call your function after printing out all of the tree's node labels. You may print the labels in any order, so long as you print the label of each one exactly once.

```
def node_printer(t):
    """
    >>> t1 = tree(1, [tree(2,
    ...                    [tree(5),
    ...                     tree(6, [tree(8)])]),
    ...               tree(3),
    ...               tree(4, [tree(7)])])
    >>> printer = node_printer(t1)
    >>> printer()
    1
    >>> printer()
    4
    >>> printer()
    7
    >>> for _ in range(5):
    ...     printer()
    3
    2
    6
    8
    5
    """
    lamb = _____

    def step():

        nonlocal lamb

        next_node, lamb = lamb()

        print(_____)

        for _____

            _____

    return step
```

Notes

→ lamb is a function
- ↳ It takes no inputs
- ↳ Its first output is (probably) a node
- ↳ Its second output is another function of the same structure

→ on each step() call we should print exactly one thing
- ↳ what do we have access to?
  - lamb: a function, hard to understand
  - t: a tree, but only points to the root — what if we're printing nodes farther down?
  - next_node: Seems promising
    - ↳ So the first return from lamb should be a new, unique node.
    - ↳ `print(label(next_node))`

→ we need to progress through the whole tree
- ↳ How do we decide where to go next?

Can't use t because this would give us repeats; t is the root node
- ↳ we need to consider all of next_node's children; we won't visit next_node again because it only gets printed once

↓ `for b in branches(next_node)`

Now what do we do for each of these branches?
- ↳ We can't recurse immediately, we want to store each node to visit later

Where do we get nodes each time? From lamb.
- ↳ How might we do this if lamb were a list?

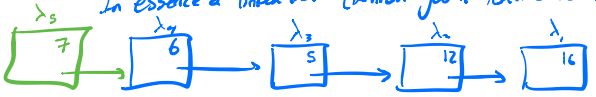
maybe
```
to_visit = [...]
⋮
next_node = to_visit.pop()   # select a new node and remove it from list of nodes to visit in the future

for b in branches(next_node):
    to_visit.append(b)
```

Let's implement this with lambda functions.

Our `next_node = to_visit.pop()` line is replaced with `next_node, lamb = lamb()`

How do we "append" to this lambda function?
- ↳ Store in nested lambda functions; each one holds one value and a reference to the next lambda function.

In essence a linked list (which you'll learn about later)



So you want to create a new lambda function which returns your new value and the previous lambda function
- ↳ Store this info in local variables

```
for b in branches(next_node):
    lamb = (lambda x, f: lambda: (x, f))(b, lamb)
```

Finally, how do we initialize lamb? It should contain the root node of the tree, and its second return value doesn't matter since it will never be called. So one example could be

```
lamb = lambda: (t, None)
```

2. **Layer Generator**

(a) Construct the generator function `layer_gen`, which takes in a tree `t` and returns a *layer iterator* of `t`. A layer iterator returns label values of nodes in the tree in level order; that is, it yields the root node, then each node from the second level, then each node from the third level and so on. *Hint: consider using the* `pop` *method of lists.*

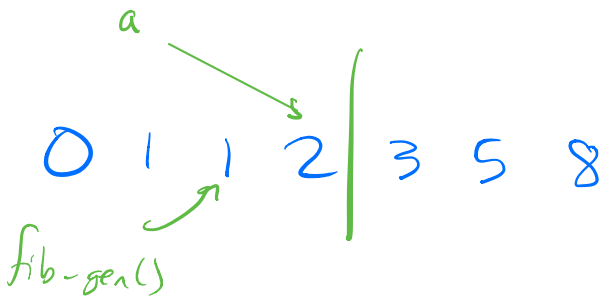
```
def layer_gen(t):
    """
    >> lg = layer_gen(t1)
    >>> for node_label in lg:
    ...     print(nodel_label)
    1
    2
    3
    4
    5
    6
    7
    8
    """
    to_visit = [t]
    while _____:
        node = _____
        _____
        _____
```

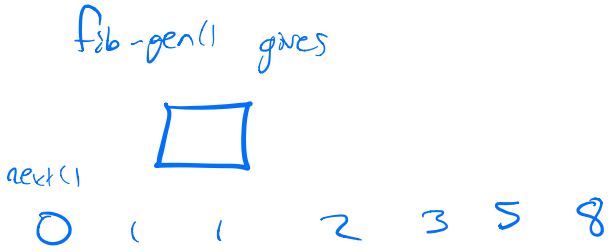
3. **Fibonacci Generator**

(a) Construct the generator function `fib_gen`, which returns elements of the Fibonacci sequence in order.

Hint: The solution doesn't require any lambda functions or crazy workarounds, but instead a clever leap of faith. Consider using the `zip` *function.*

```
def fib_gen():
    """
    >>> fg = fib_gen()
    >>> for _ in range(10):
    ...     print(next(fg))
    0
    1
    1
    2
    3
    5
    8
    13
    21
    34
    """
    yield from _____
    a = _____
    _____
    for _____:
        _____
```

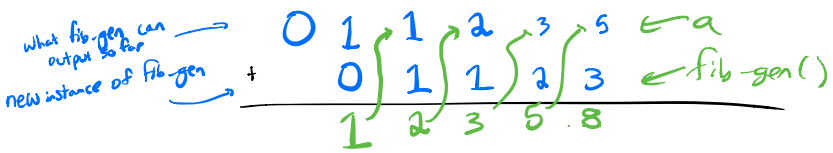




→ What should we yield from to start?
↳ Remember the two "base cases" of Fibonacci: $f(0)=0$, $f(1)=1$

```
yield from [0, 1]
```

→ We don't have enough lines to keep track of prev and curr and to yield values.
↳ So let's recurse!
↓
We only have 0 and 1 so far, but that's enough to create the next value in the sequence.
↓
But we can't reach back to get previous values, so instead let's set two Fibonacci sequences side by side, offset by 1.



So we just need to create two fib generators — one that has stepped forward by 1 — and yield their sum at each step.
To move an iterator forward, we call next.
So

```
a = fib_gen()
next(a)
for x, y in zip(a, fib_gen()):
    yield x + y
```

4. **Partition Generator**

(a) Construct the generator function `partition_gen`, which takes in a number `n` and returns a *n-partition iterator*. An *n-partition iterator* yields partitions of `n`, where a partition of `n` is a list of integers whose sum is `n`. The iterator should only return unique partitions; the order of numbers within a partition and the order in which the partitions are returned does not matter.

```
def partition_gen(n):
    """
    >>> for partition in partition_gen(4):  # note: order doesn't matter
    ...     print(partition)
    [4]
    [3, 1]
    [2, 2]
    [2, 1, 1]
    [1, 1, 1, 1]
    """
    def yield_helper(j, k):
        if _____:
            yield _____
        elif _____:
            for _____ in _____:
                yield _____
            yield _____
    yield _____
```

→ Similar to count_partitions
↳ can give a good hypothesis for what j and k represent

Maybe let j ≜ the current number we're trying to partition
k ≜ the largest number we're allowed to partition with ← could also use the smallest

```
yield from yield_helper(n, n)
```

Why can't we just recurse on yield_partitions?
↳ Idea: Say we have n = 6. Then yield all unique partitions of 5 and put [1] at the end, all unique partitions of 4 and put [2] at the end and so on.

↓

Problem: Doesn't guarantee uniqueness

↳ (2, 1, 1, 1) is a valid partition of 5, so could get (2, 1, 1, 1, 1)
↳ (1, 1, 1, 1) is a valid partition of 4, so could get (1, 1, 1, 1, 2)
} same partition!

BASE CASES
↳ similar to count_partitions
↳ if j == 0, then there is exactly one partition