Lecture 13: Midterm Review

CS 61A - Summer 2024
Raymond Tan & Charlotte Le

Tree Recursion/ADT Trees

General Structure of Tree Recursive Problems

- **Base Case(s)**: usually one or more – when have I found a valid path? an invalid one (i.e. there's no possible way I can end up on a valid path)?
  - count_stair_ways: at the top of the staircase / stepped past the top
    - return 1 represents valid path, return 0 represents invalid path
  - count_partitions: successfully partitioned n fully / exceeded n with parts OR run out of parts to use
  - insect_combinatorics: hit the top-right corner / gone out-of-bounds
- **Recursive Calls**: multiple, often each represents a choice
  - count_stair_ways: take 1 step or take 2 steps
  - count_partitions: use a part of size k or don't use any parts of size k
  - insect_combinatorics: move right or move up
- **Recombination**: some function or operation to construct the answer of your original problem from the answer of your subproblems
  - count_stair_ways, count_partitions, insect_combinatorics: total num of ways → sum recursive calls

General Tips for Trees

- Be familiar with the Tree ADT
    - label is a function that returns the value stored at a node
    - is_leaf is a function that takes a tree and returns whether it is a tree with no branches (a leaf)
    - branches(t) returns a list of branches
        - We can index into this list or iterate over it
- Recursive leap of faith
    - Assume recursive calls work – if they work, what tree do we expect to get back?
- To process tree:
    - Process the label, and loop over branches making recursive calls on each branch
- Make sure to have a base case!
    - Usually when the tree is a leaf

Fall 2020 MT2 Q4A

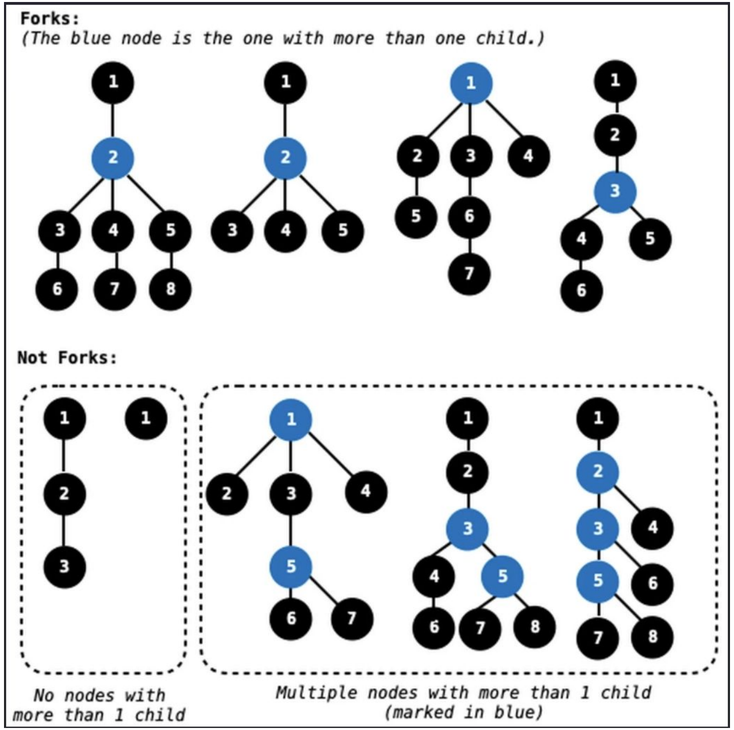
```
def max_path(t):
    """Return the largest sum of labels along any path from the root to a leaf
    of tree t, which has positive numbers as labels.
    >>> a = tree(1, [tree(2), tree(3), tree(4, [tree(5)])])
    >>> max_path(a) # 1 + 4 + 5
    10
    >>> b = tree(6, [a, a, a])
    >>> max_path(b) # 6 + 1 + 4 + 5
    16
    """
    return _________ + max(_________ + _________)
```

Solution

```python
def max_path(t):
    """Return the largest sum of labels along any path from the root
    of tree t, which has positive numbers as labels.
    >>> a = tree(1, [tree(2), tree(3), tree(4, [tree(5)])])
    >>> max_path(a) # 1 + 4 + 5
    10
    >>> b = tree(6, [a, a, a])
    >>> max_path(b) # 6 + 1 + 4 + 5
    16
    """
    return label(t) + max([0] + [max_path(b) for b in branches(t)])
```

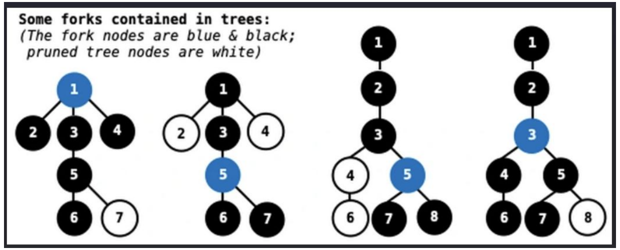
Fall 2020 MT 2 Q4C

A fork is a tree in which exactly one node has more than one child.



Fall 2020 MT 2 Q4C

```
def max_fork(t):
    """Return the largest sum of the labels in any fork contained in tree t,
    which has positive numbers as labels. If t contains no forks, return 0.
    >>> a = tree(1, [tree(2), tree(3), tree(4, [tree(5)])])
    >>> max_fork(a) # 1 + 2 + 3 + 4 + 5
    15
    >>> b = tree(6, [a, a, a])
    >>> max_fork(b) # 6 + (1 + 4 + 5) + (1 + 4 + 5) + (1 + 4 + 5)
    36
    """
    n = len(branches(t))
    if n == 0:
        return 0
    elif n == 1:
        below = _________
        if _________:
            return _________ + below
        else:
            return 0
    else:
        here = sum([_________ for b in branches(t)])
        there = max([_________ for b in branches(t)])
        return label(t) + max(here, there)
```



Solution

```python
def max_fork(t):
    """Return the largest sum of the labels in any fork contained in tree t,
    which has positive numbers as labels. If t contains no forks, return 0.
    >>> a = tree(1, [tree(2), tree(3), tree(4, [tree(5)])])
    >>> max_fork(a) # 1 + 2 + 3 + 4 + 5
    15
    >>> b = tree(6, [a, a, a])
    >>> max_fork(b) # 6 + (1 + 4 + 5) + (1 + 4 + 5) + (1 + 4 + 5)
    36
    """
    n = len(branches(t))
    if n == 0:
        return 0
    elif n == 1:
        below = max_fork(branches(t)[0])
        if below > 0:
            return label(t) + below
        else:
            return 0
    else:
        here = sum([max_path(b) for b in branches(t)])
        there = max([max_fork(b) for b in branches(t)])
        return label(t) + max(here, there)
```

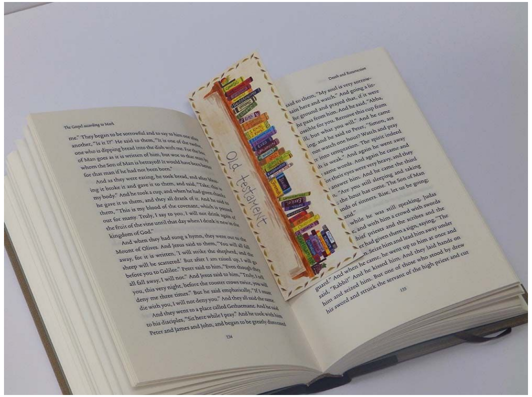
Iterators and Generators

General Tips

- The two main functions when dealing with iterators:
    - iter(iterable): This creates an iterator tracking the underlying iterable
    - next(iterator): Returns the next item of an iterator
        - StopIteration error if there are no more items left
- Generator functions return generator objects
    - Keep this in mind when using recursion in generator functions
- yield from will yield all values of an iterable one at a time
    - You can still call yield directly on an iterable – this will just yield the entire iterable at once

Bookmark Analogy

- Iterators can be thought of as "bookmarks" for a corresponding iterable
  - Pages of a book represent items in an iterable
- Calling `next` on the iterator gives us the next item in the sequence
  - Until the bookmark reaches the very end of the iterable, where calling `next` now returns an error



Spring 2018 Final Q4A

```
def times(f, x):
    """Return a function g(y) that returns the number of f's in f(f(...(f(x)))) == y.
    >>> times(lambda a: a + 2, 0)(10) # 5 times: 0 + 2 + 2 + 2 + 2 + 2 == 10
    5
    >>> times(lambda a: a * a, 2)(256) # 3 times: square(square(square(2))) == 256
    3
    """
    def repeat(z):
        """Yield an infinite sequence of z, f(z), f(f(z)), f(f(f(z))), f(f(f(f(z)))), ...."""
        yield ______________________________________________________________________________
        ____________________________________________________________________________________

    def g(y):
        n = 0
        for w in repeat(____________________________________________________________):
            if ___________________________________________________________________________:
                ______________________________________________________________________
            ________________________________________________________________________
    return g
```

Solution

```python
def times(f, x):
    """Return a function g(y) that returns the number of f's in f(f(...(f(x)))) == y.
    >>> times(lambda a: a + 2, 0)(10) # 5 times: 0 + 2 + 2 + 2 + 2 + 2 == 10
    5
    >>> times(lambda a: a * a, 2)(256) # 3 times: square(square(square(2))) == 256
    3
    """
    def repeat(z):
        """Yield an infinite sequence of z, f(z), f(f(z)), f(f(f(z))), f(f(f(f(z)))), ...."""
        yield z
        yield from repeat(f(z))

    def g(y):
        n = 0
        for w in repeat(x):
            if w == y:
                return n
            n += 1
    return g
```

Break

Lists & Mutability

List Slicing

- List slicing returns a specified "chunk" of a list
- Syntax:

```
lst[i:j:k]
```

`i`: Starting Index
(**inclusive**)

`j`: Ending Index
(**exclusive**)

`k`: Step size
(default set to 1)

List Comprehensions

```
[<expression> for <element> in <sequence>]
[<expression> for <element> in <sequence> if <conditional>]
```

- expression: the expression we want to include in the final list
- element: the variable bound to where we currently are in the sequence
- sequence: the iterable we are basing the list comprehension on
- conditional (optional): only include expression if this conditional is true

Mutability

- The same object can change in value throughout the course of computation
- All names that refer to the same object are affected by a mutation
- Only objects of mutable types can change
  - **Mutable:** lists & dictionaries
  - **Immutable:** strings, tuples, numeric types, etc.

Identity Operators

Identity

<exp0> **is** <exp1>

evaluates to True if both <exp0> and <exp1> evaluate to the same object

Equality

<exp0> **==** <exp1>

evaluates to True if both <exp0> and <exp1> evaluate to equal values

Identical objects are always equal values

List Methods

- `append(el)`
  - Add `el` to the end of the list.
  - Return None.
- `extend(lst)`
  - Extend the list by concatenating it with `lst`.
  - Return None.
- `insert(i, el)`
  - Insert `el` at index `i`. This does not replace any existing elements, but only adds the new element `el`.
  - Return None.
- `remove(el)`
  - Remove the first occurrence of `el` in list. Errors if `el` is not in the list.
  - Return None otherwise.
- `pop(i)`
  - Remove and return the element at index `i`.

Making a Copy vs Mutating

These will create an entirely new list:

- Taking any slice of a list
  - `a[1:3]`
- Writing a list comprehension
- Concatenating lists
  - `a = a + [3, 2]`

These will mutate a list that already exists

- Any of the mutation functions (see previous slide)
- Bracketing on the right side of an assignment statement
  - `a[0] = 3`
  - `a[1:3] = [3, 4, 5]`
  - **not** `a = [3, 4, 5]`
  - `a += [3, 4, 5]` is a special case in which mutation actually does occur

You Should Know:

- How to construct a new list
- How to index elements out of a list
- How to mutate a list by indexing
- How to take a slice of a list
- How to write a list comprehension and what they do
- List mutation operations
  - They'll be on the study guide!
  - You should know what they do
- What operations create a new list, and which ones mutate an existing list
- How to represent lists in environment diagrams

Lists & Mutability in Environment Diagrams

Advice

- Practice with Python Tutor!!!!
- Understand what mutates a list and what creates a copy
- Each line of the environment diagram is a clue!
  - Names show what variables you should have
  - Values show what the expressions should evaluate to eventually
  - Frame names show which function is called
  - Frame numbers show order of program flow

Step-by-Step: Assignment Statements

(1) evaluate the expression on the right of the = sign to get a value/object
- when encountering a name while evaluating, always search the current frame first
- then, search the parent frame, and then that frame's parent frame, etc. (until global frame)

(2) does the name on the left of the = already exist in the *current* frame?

↳ yes
- erase the current binding (either a value or object)
- bind the name to the value/object from (1)

↳ no
- bind the new name to the value/object

notes
- if there are multiple expressions in a statement, evaluate all expressions first from left to right before making any bindings

Step-by-Step: Lambdas

(1) draw the lambda function object with: func & λ & formal parameters & parent frame

notes
- a function's parent frame is the frame in which the function was defined
- lambda expressions (unlike def statements) do not create any new bindings in the environment

Step-by-Step: def Statements

(1) draw the function object with: func & intrinsic name & formal parameters & parent frame
(2) does the intrinsic name of the function already exist in the current frame?
↳ yes
- erase the current bindings
↳ no
- write it in
(3) bind the newly created function object to this name

notes
- a function's parent frame is the frame in which the function was *defined*

Step-by-Step: Call Expressions

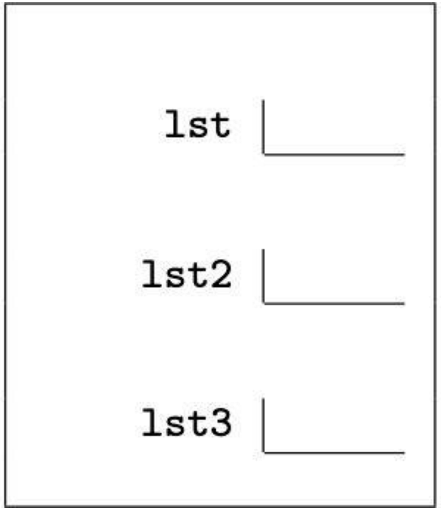
(1) evaluate the operator (should be a function)
(2) evaluate the operands left to right to obtain a value/object for each
(3) open a new frame (necessary for every call expression)
(4) label the new frame with: sequential frame number & intrinsic name & parent frame of function
(5) bind the formal parameters of the function to the arguments whose values/objects you found in (2)
(6) execute the body of the function until a return value is obtained
(7) write down the return value in the frame

notes
- if a function does not have a return value, it implicitly returns None
- do not draw frames for built-in or imported functions e.g., min(...) and add(...)
- with nested call expressions, remember to open frames in the other that they are called

Su19 MT Q3a

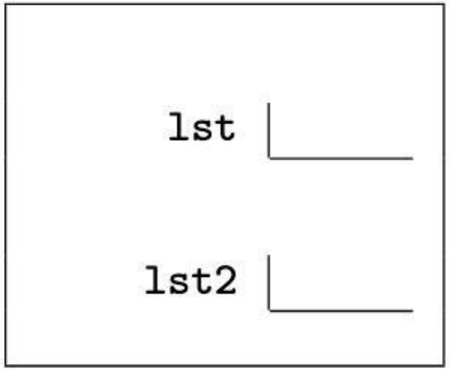
```python
lst = [2, 4, lambda: lst]
lst2 = lst
lst = lst[2:]
lst3 = lst[0]()
```

PythonTutor



[Su19 MT Q3b]

```
lst = [[5], 2, 4, 10]
lst2 = lst[1:3] + lst[:3]
for n in lst2[:2]:
    lst.append(lst2[n])
```



[PythonTutor]

Su19 MT Q3c

```
lst = ['goodbye', 0, None, 8, 'hello', 1]
while lst.pop():
    x = lst.pop()
    if x:
        lst.pop()
    else:
        lst.append('three')
```

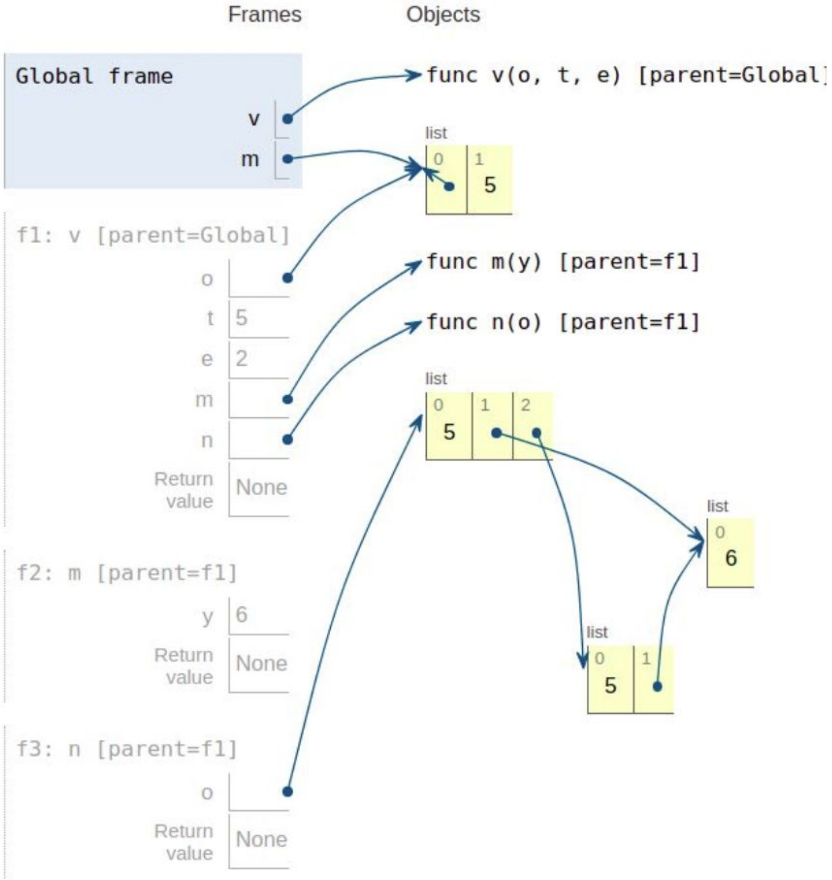
PythonTutor

You may not write any numbers or arithmetic operators (+, -, *, /, //, **) in your solution.

```
def v(o, t, e):
    def m(y):
        _________ #(a)
    def n(o):
        o.append(_____)#(b)
        o.append(_____)#(c)
    m(e)
    n([t])
    e = 2
m = [3, 4]
v(m, 5, 6)
```

Blank (c) choose all that apply

```
o
[o]
list(o)
list([o])
o + []
[o[0], o[1]]
o[:]
```



Miscellaneous

Higher-Order Functions

A function that:

- takes a function as an argument value
- **and/or** returns a function as a return value

```
def composer(func1, func2):
    """Return a function f, such that f(x) = func1(func2(x))."""
    def f(x):
        return func1(func2(x))
    return f
```

Lambda Expressions

- Does not bind to a name
- Body is not evaluated until lambda is called
- Can be used as an operator or an operand

```
def multiply(x, y):
    return x * y
>>> multiply(2, 3)
6
```

```
multiply = lambda x, y : x * y

>>> multiply(2, 3)
6
>>> (lambda x, y : x * y)(2, 3)
6
```

```
negate = lambda f, x : -f(x)
negate(lambda x : x * x, 3)
```

- negate has two parameters f and x and returns -f(x)
- f → lambda x : x * x
- x → 3
- negate returns:
    - - f(x) → - (lambda x : x * x)(3) → - 9

Memoization

- Each time we execute a recursive computation, we record the result of that computation
- That way, if we ever see exactly the same parameters a second time, we can access the result directly, rather than having to execute a new series of recursive calls

Orders of Growth

NOA = Number of Operations

Most efficient

- **Constant** growth
  - Increasing n doesn't affect NOA
- **Logarithmic** growth
  - Doubling n only affects NOA by a constant
- **Linear** growth
  - Incrementing n increases NOA by a constant
- **Quadratic** growth
  - Incrementing n increases NOA by n times a constant
- **Exponential** growth
  - Incrementing n multiples NOA by a constant

Least efficient

Study Guide: cs61a.org/study-guide/orders-of-growth