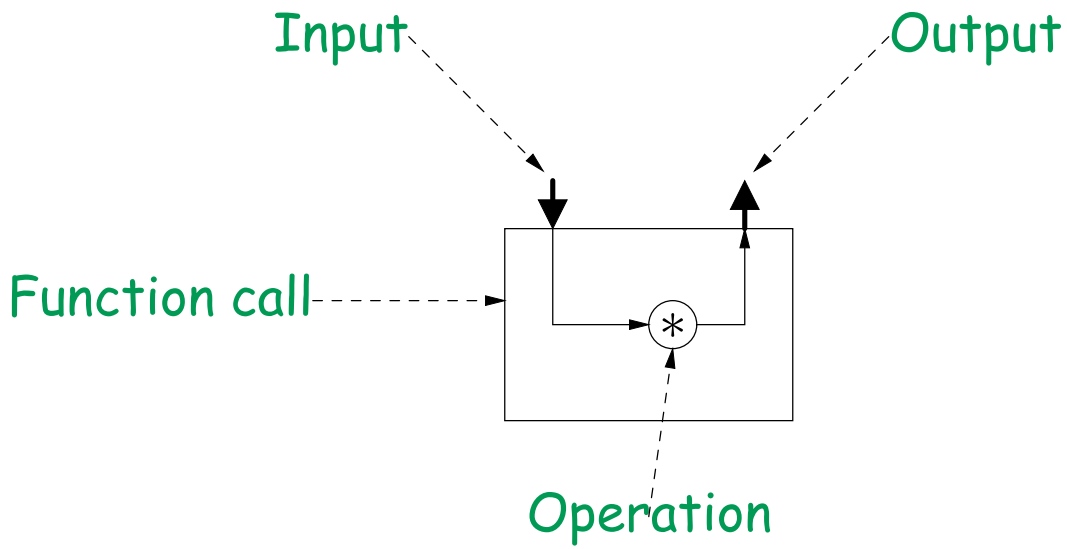
Lecture #18: Recursive Processes, Memoization, Tree Structures

Review: Varieties of Recursive Processes

- We can characterize (potentially) recursive functions according to the patterns in which data flows through them.
- The simplest case is a non-recursive function call, which does something (call it h) to its input data and returns the result:

```
def func0(x):
    return h(x)
```



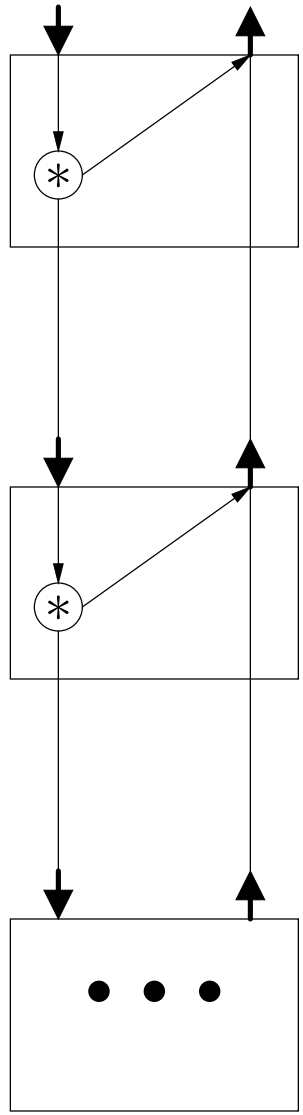
- "Operations" include any processing that does not cause further recursion.
- This is a *leaf call.*

Iterative (Tail-Recursive) Processes

- Tail-recursive processes do no further processing after a recursive call

```
def func1(x):
    if P(x):
        return h1(x)
    else:
        return func1(h2(x))
```

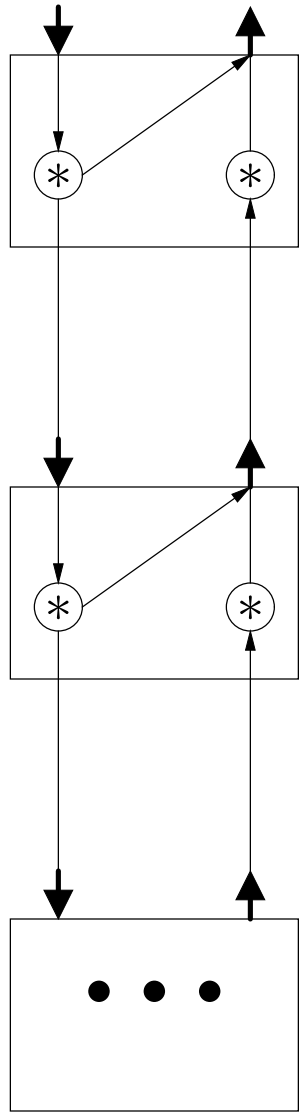
  - Once we make a recursive call, can forget about the caller.
  - Constant space needed for administrative overhead (in principle)
  - Time required (number of operations) proportional to call depth.



Linear Recursions

- Linear recursions do one recursive call and then additional processing

```
def func2(x):
    if P(x):
        return h1(x)
    else:
        return h3((func2(h2(x)))
```

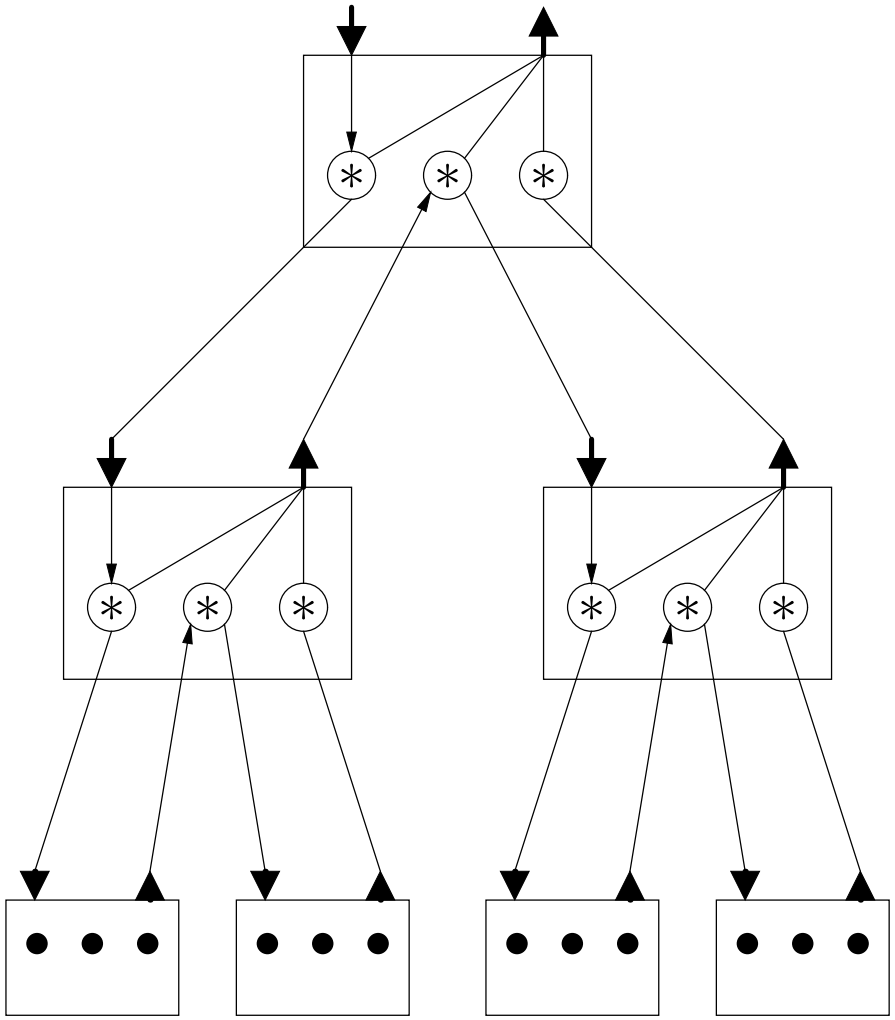


- Must keep track of pending calls, because there is more to do for each.
- Space proportional to depth of calls needed for administrative overhead.
- Time required proportional to call depth.

Tree (General) Recursion

- Tree recursions do more than one recursive call in each function execution.

```
def func3(x):
    if P1(x):
        return h1(x)
    else:
        y = func3(h2(x))
        if P2(x):
            return h3(x, y)
        z = func3(h4(x, y))
        return h5(x, y, z)
```



- Again, must keep track of pending calls (one per level).
- So, space proportional to depth of calls.
- But time required may be *exponential* in call depth.

Avoiding Redundant Computation

- Consider again the classic Fibonacci recursion:

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

- This is tree recursion with a serious speed problem.
- Computation of, say `fib(5)` computes `fib(2)` several times, because both `fib(4)` and `fib(2)` compute it, and both `fib(5)` and `fib(4)` compute `fib(3)`. Computing time grows exponentially.
- The iterative version does not have this problem because it saves the results of the recursive calls (in effect) and ' reuses them.

```
def fib(n):
    if n <= 1: return n
    a, b = 0, 1
    for k in range(2, n+1): a, b = b, a+b
    return b
```

Change Counting

- Here's another version of change-counting, which you did in homework:

```
def count_change(amount, coins = (50, 25, 10, 5, 1))
    """Return the number of ways to make change for AMOUNT, where
    the coin denominations are given by COINS.
    """

    if amount == 0:
        return 1
    elif len(coins) == 0 or amount < 0:
        return 0
    else:
        return count_change(amount-coins[0], coins) + \
               count_change(amount, coins[1:])
```

- Here, we often revisit the same subproblem:
  - E.g., Consider making change for 87 cents.
  - When choose to use one half-dollar piece, we have the same subproblem as when we choose to use no half-dollars and two quarters.

Memoizing

- Extending the iterative Fibonacci idea, let's keep around a table ("memo table") of previously computed values.
- Consult the table before using the full computation.
- Example: `count_change`:

```
def count_change(amount, coins = (50, 25, 10, 5, 1)):
    memo_table = {}
    def count_change(amount, coins):
        if (amount, coins) not in memo_table:
              memo_table[amount,coins]
                 = full_count_change(amount, coins)
        return memo_table[amount,coins]
    def full_count_change(amount, coins):
        original recursive solution goes here verbatim
        when it calls count_change, calls memoized version.
    return count_change(amount,coins)
```

- Question: how could we test for infinite recursion?

Optimizing Memoization

- Used a dictionary to memoize `count_change`, which is highly general, but can be relatively slow.
- More often, we use arrays indexed by integers (lists in Python), but the idea is the same.
- For example, in the `count_change` program, we can index by `amount` and by the *starting index* of the original value of `coins` that we use.

```
def count_change(amount, coins = (50, 25, 10, 5, 1)):
    # memo_table[amt][k] contains the value computed for
    #   count_change(amt, coins[k:])
    memo_table = [ [-1] * (len(coins)+1) for i in range(amount+1) ]
    def count_change(amount, coins):
        if amount < 0: return 0
        elif memo_table[amount][len(coins)] == -1:
              memo_table[amount][len(coins)]
                 = full_count_change(amount, coins)
        return memo_table[amount][len(coins)]
    ...
```

Order of Calls

- Going one step further, we can analyze the order in which our program ends up filling in the table.
- So consider adding some tracing to our memoized `count_change` program:

```
memo_table = {}
def count_change(amount, coins):
    ... full_count_change(amount, coins) ...
    return memo_table[amount,coins]
@trace
def full_count_change(amount, coins):
    if amount == 0:                          return 1
    elif len(coins) == 0 or amount < 0: return 0
    else:
        return count_change(amount, coins[1:]) \
               + count_change(amount-coins[0], coins)
return count_change(amount,coins)
```

Result of Tracing

- Consider `count_change(57)` (returns only):

```
full_count_change(57, ()) -> 0   Need shorter 'coins' arguments
full_count_change(56, ()) -> 0   first.
...
full_count_change(1, ()) -> 0    For same coins, need smaller
full_count_change(0, (1,)) -> 1  amounts first.
full_count_change(1, (1,)) -> 1
...
full_count_change(57, (1,)) -> 1
full_count_change(2, (5, 1)) -> 1
full_count_change(7, (5, 1)) -> 2
...
full_count_change(57, (5, 1)) -> 12
full_count_change(7, (10, 5, 1)) -> 2
full_count_change(17, (10, 5, 1)) -> 6
...
full_count_change(32, (10, 5, 1)) -> 16
full_count_change(7, (25, 10, 5, 1)) -> 2
full_count_change(32, (25, 10, 5, 1)) -> 18
full_count_change(57, (25, 10, 5, 1)) -> 60
full_count_change(7, (50, 25, 10, 5, 1)) -> 2
full_count_change(57, (50, 25, 10, 5, 1)) -> 62
```

Dynamic Programming

- Now rewrite `count_change` to make the order of calls explicit, so that we needn't check to see if a value is memoized.
- Technique is called *dynamic programming* (for some reason).
- We start with the base cases (0 coins) and work backwards.

```
def count_change(amount, coins = (50, 25, 10, 5, 1)):
    memo_table = [ [-1] * (len(coins)+1) for i in range(amount+1) ]
    def count_change(amount, coins):
        if amount < 0: return 0
        else: return memo_table[amount][len(coins)]
    def full_count_change(amount, coins): # How often called?
        ... # (calls count_change for recursive results)

    for a in range(0, amount+1):
        memo_table[a][0] = full_count_change(a, ())
    for k in range(1, len(coins) + 1):
        for a in range(1, amount+1):
            memo_table[a][k] = full_count_change(a, coins[-k:])
    return count_change(amount, coins)
```

Trees As a Class

- Just as linked lists can be repackaged as a class, so can trees.

```
class Tree:
    def __init__(self, label, children=()):
        self.label = label
        for branch in children:
            assert isinstance(branch, Tree)
        self.children = list(children)

    def __repr__(self):
        if self.children:
            children_str = ', ' + repr(self.children)
        else:
            children_str = ''
        return 'Tree({0}{1})'.format(self.label, children_str)

    def is_leaf(self):
        return not self.children

    def __len__(self):
        """The number of vertices in me."""
```