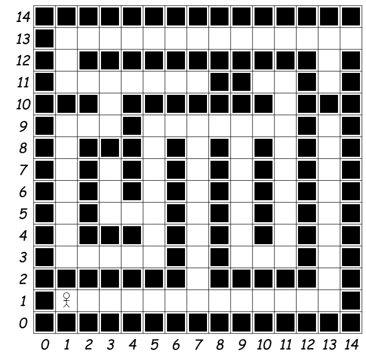
Lecture #20: Recursive Processes, Memoization, Tree Structures

Example: Escape from a Maze

- Consider a rectangular maze consisting of an array of squares some of which are occupied by large blocks of concrete:



- Given the size of the maze and locations of the blocks, prisoner, and exit, how does the prisoner escape?

Maze Program (Incorrect)

```
def solve_maze(row0, col0, maze):
    """Assume that MAZE is a rectangular 2D array (list of lists) where
    maze[r][c] is true iff there is a concrete block occupying
    column c of row r.  ROW0 and COL0 are the initial row and column
    of the prisoner.  Returns true iff there is a path of empty
    squares that are horizontally or vertically adjacent to each other
    starting with (ROW0, COL0) and ending outside the maze."""
    if row0 not in range(len(maze)) or col0 not in range(len(maze[row])):
        return True
    elif maze[row0][col0]:  # In wall
        return False
    else:
        return solve_maze(row0+1, col0, maze) or solve_maze(row0-1, col0, maze) \
            or solve_maze(row0, col0+1, maze) or solve_maze(row0, col0-1, maze) \
    # What's wrong?
```


Maze Program (Corrected)

To fix the problem, remember where we've been:

```
def solve_maze(row0, col0, maze):
    """Assume that MAZE is a rectangular 2D array (list of lists) where
    maze[r][c] is true iff there is a concrete block occupying
    column c of row r.  ROW0 and COL0 are the initial row and column
    of the prisoner.  Returns true iff there is a path of empty
    squares that are horizontally or vertically adjacent to each other
    starting with (ROW0, COL0) and ending outside the maze."""
    visited = set()   # Set of visited cells
    W, H = len(maze[0]), len(maze)
    def escapep(r, c):
        """True iff is a path of empty, unvisited cells from (R, C) out of maze."""
        if r not in range(H) or c not in range(W):
            return True
        elif maze[r][c] or (r, c) in visited:
            return False
        else:
            visited.add((r,c))
            return escapep(r+1, c) or escapep(r-1, c) \
                or escapep(r, c+1) or escapep(r, c-1)
    return escapep(row0, col0)
```


Example: Making Change

```
def count_change(amount, denoms = (50, 25, 10, 5, 1)):
    """The number of ways to change AMOUNT cents given the
    denominations of coins and bills in DENOMS.
    >>> # 9 cents = 1 nickel and 4 pennies, or 9 pennies
    >>> count_change(9)
    2
    >>> # 12 cents = 1 dime and 2 pennies, 2 nickels and 2 pennies,
    >>> # 1 nickel and 7 pennies, or 12 pennies
    >>> count_change(12)
    4
    """
    if amount == 0:        return 1
    elif len(coins) == 0:  return 0
```


Avoiding Redundant Computation

- In the (tree-recursive) maze example, a naive search could take us in circles, resulting in infinite time.
- Hence the visited set in the escapep function.
- This set is intended to catch redundant computation, in which re-processing certain arguments cannot produce anything new.
- We can apply this idea to cases of finite but redundant computation.
- For example, in count_change, 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 sub-problem as when we choose to use no half-dollars and two quarters.
- Saw an approach in Lecture #16: memoization.

Memoizing

- Idea is to 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 = {}
    # Local definition hides outer one so we can cut-and-paste
    # from the unmemoized (red) solution.
    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 solution goes here verbatim
    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 portion of coins that we use, which is always a slice that runs to the end.

```
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 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 not coins: return 0
    elif amount >= coins[0]:
        return count_change(amount, coins[1:]) \
             + count_change(amount-coins[0], coins)
    else:
        return count_change(amount, coins[1:])
return count_change(amount,coins)
```


Result of Tracing

- Consider count_change(57) (returns only):

```
full_count_change(57, ()) -> 0
full_count_change(56, ()) -> 0
...
full_count_change(1, ()) -> 0
full_count_change(0, (1,)) -> 1
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, and work backwards.

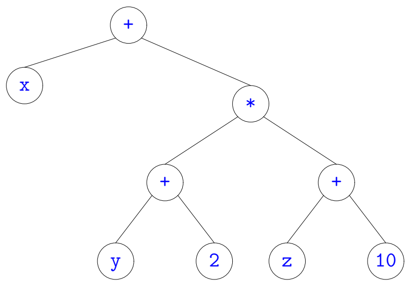
```
def count_change(amount, coins = (50, 25, 10, 5, 1)):
    memo_table = [ [-1] * (len(coins)+1) for i in range(amount+1) ]
    def count_change(amount, coins):
        return memo_table[amount][len(coins)]
    def full_count_change(amount, coins):
        # How often is this called?
        ... # (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)
```


New Topic: Tree-Structured Data

- 1 Linear-recursive and tail-recursive functions make a single recursive call in the function body. Tree-recursive functions can make more.
- Linear recursive data structures (think rlists) have single embedded recursive references to data of the same type, and usually correspond to linear- or tail-recursive programs.
- To model some things, we need mulitple recursive references in objects.
- In the absence of circularity (paths from an object eventually leading back to it), such objects form data structures called *trees*:
  - The objects themselves are called *nodes* or *vertices*.
  - Tree objects that have no (non-null) pointers to other tree objects are called *leaves*.
  - Those that do have such pointers are called *inner nodes*, and the objects they point to are *children* (or *subtrees* or (uncommonly) *branches*).
  - A collection of disjoint trees is called a *forest*.

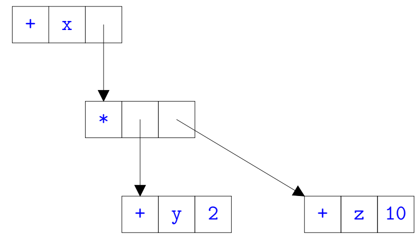
Example: Expressions

- An expression (in Python or other languages) typically has a recursive structure. It is either
  - A literal (like 5) or symbol (like x)—a leaf—or
  - A compound expression consisting of an operator and zero or more operands, each of which is itself an expression.
- For example, the expression x + (y+2)*(z+10) can be thought of as a tree (what happened to the parentheses?):



Expressions as Tuples or Lists

- We can represent the abstract structure of the last slide with Python objects we've already seen:




```
("+", "x", ("*", ("+", "y", "2"), ("+", "z", "10")))
```


Class Representation

- ...or we can introduce a Python class:

```
class ExprTree:
    def __init__(self, operator):
        self.__operator = operator

    @property
    def operator(self):
        return self.__operator

    @property
    def left(self):
        raise NotImplementedError

    @property
    def right(self):
        raise NotImplementedError
```

```
class Leaf(ExprTree):
    pass

class Inner(ExprTree):
    def __init__(self, operator,
                 left, right):
        ExprTree.__init__(self, operator)
        self.__left = left;
        self.__right = right
    @property
    def left(self):
        return self.__left
    @property
    def right(self):
        return self.__right
```

```
Inner("+", Leaf("x"),
      Inner("*", Inner("+", Leaf("y"), Leaf("2")),
            Inner("+", Leaf("z"), Leaf("10"))))
```