6.009: Fundamentals of Programming

Lecture 3:
Basics of Recursion

[Slides and code on Calendar page]
Basic Recipe

To solve a problem recursively we identify
• A simple **base case** (or cases) – a terminating scenario that does not use recursion to produce and answer, and
• A set of rules that **reduce** all other cases towards the base case

```python
def fact(n):
    if n <= 1:
        return n
    else:
        return n * fact(n-1)
```

Note the growing stack of calculations to be completed as the recursion proceeds. The size of the stack is the **depth** of the recursion.
Procedure Activation Records

• When Python executes a procedure call it creates an activation record that contains
  – A pointer to the activation record at the time of the call and where to resume the computation when the procedure completes
  – A new environment frame to hold the bindings for the argument values and any variables local to the procedure. This includes a pointer to the parent environment frame to be used to find non-local variables.

• Procedure calls are nested, so we can use a last-in, first-out data structure (called a stack) to hold the activation records.

• Activation records take room! Recursion too deep? You can overflow the stack 😞
Recursion or Iteration?

Consider the following two ways of computing the length of a list:

```python
# iterative
def length(l):
    count = 0
    for element in l:
        count += 1
    return count

# recursive
def rlength(l):
    return 0 if l==[] else 1 + rlength(l[1:])
```

Which would you choose? Why?

Question: do we need recursion? Can’t we always solve the problem iteratively?
Access N-Dimensional Array

```python
def nd_get(coord, array):
    # Example usage:
    coord = (1, 2, 0)
    array = [
        [[1, 2], [3, 4], [5, 6], [7, 8]],
        [[9, 10], [11, 12], [13, 14], [15, 16]]
    ]
    value = nd_get(coord, array)
    print(value)
```

A 2x4x2 3-dimensional array

Array[1][2][0] = 13

Let's write `nd_get(coord, array)`
Recursion or Iteration?

# iterative
def nd_get_iterative(coord, array):
    for c in coord:
        array = array[c]
    return array

# recursive
def nd_get_recursive(coord, array):
    if len(coord) == 1:
        return array[coord[0]]
    else:
        return nd_get_recursive(coord[1:], array[coord[0]])

“tail” recursion
Recursion: processing trees

\[(3 + 2) \ast (2 \ast 4)\]

\[
\text{eval\_ast}(\text{ast}, \text{env})
\]

- If \text{ast} is an integer, return its value
- If \text{ast} is a string, treat as variable name: return \text{env}[\text{ast}]
- Otherwise perform operation on values of the operands

\[
["\ast", ["+", 3, 2], ["\ast", 2, 4]]
\]
eval_ast(ast, env)

# map operator names to appropriate arithmetic function
otable = {
    '+': (lambda x,y: x + y),
    '-': (lambda x,y: x - y),
    '*': (lambda x,y: x * y),
    '/': (lambda x,y: x / y),
}

# return value of ast, variable values found in env dict
def eval_ast(ast, env={}):
    if isinstance(ast, list):
        operator = ottable[ast[0]]
        operands = [eval_ast(operand, env) for operand in ast[1:]]
        v = operands[0]
        for vv in operands[1:]:
            v = operator(v,vv)
        return v
    elif isinstance(ast, str):
        return env[ast]
    else:
        return ast
is_proper(node)

- Return true if every path from node to leaves has same number of black nodes
is_proper(node)

# root is proper if children are proper and
# have the same number of black nodes

def is_proper(root):
    # returns (proper,black_count)
    def helper(root):
        if root is None:
            return (True,0)

        black = 1 if root[0] == 'black' else 0
        lproper,lcount = helper(root[1])
        rproper,rcount = helper(root[2])

        ### add these lines to deal with single children
        if root[1] is None: return (rproper, rcount+black)
        if root[2] is None: return (lproper, lcount+black)
        ###

        if lproper and rproper and lcount==rcount:
            return (True,lcount + black)
        return (False,0)

    return helper(root)[0]
sublist_sums_to_N(num_list, N)

sublist_sums_to_N(num_list, N)
• Return true if zero or more numbers from num_list sum to N

• sublist_sums_to_N([1,2,5,9,17], 34) = True
• sublist_sums_to_N([1,2,5,9,17], 22) = True
• sublist_sums_to_N([1,2,5,9,17], 9) = True
• sublist_sums_to_N([1,2,5,9,17], 13) = False
• sublist_sums_to_N([1,2,5,9,17], 0) = True

• Strategy: recursive enumeration to find trial sublists
Plan A: try them all!

# return list containing all sublists of l
def all_sublists(l):
    if len(l) == 0:
        return [[]]  # empty list

    first = [l[0]]  # as its own list
    rest = l[1:]
    result = []
    for ll in all_sublists(rest):
        result.append(ll)
        result.append(ll + first)
    return result

# generate all sublists, check them all
def sublist_sums_to_N(l,N):
    return any(sum(sublist) == N
                for sublist in all_sublists(l))
Plan B: check one at time

```python
# generate all possible sublists one at a time
def generate_all_sublists(l):
    if len(l) == 0:
        yield []  # empty list
    else:
        first = [l[0]]
        rest = l[1:]
        for ll in generate_all_sublists(rest):
            yield ll
        yield first + ll

# generate all sublists, check them as we go
def sublist_sums_to_N(num_list, N):
    for sublist in generate_all_sublists(num_list):
        if sum(sublist) == N:
            return True
    return False
```

6.009 Fundamentals of Programming
Plan C: don’t build sublists!

# don't actually generate sublists
def sublist_sums_to_N(l,N):
    if len(l) == 0:
        return N == 0
    first = l[0]
    rest = l[1:]
    # try using first element, then try not using it
    return sublist_sums_to_N(rest, N-first) or \
       sublist_sums_to_N(rest, N)
It takes practice to find ways of solving a problem recursively. The hard part is coming up with sub-problems that have the same structure as the original problem.

Wait! $2^{2N} - 1 = (2^N - 1)(2^N + 1)$ is!
Tiling with Trominos

Insight: place a tromino in such a way as to generator four sub-problems: $2^{N-1} \times 2^{N-1}$ boards, each with one missing square
Tiling with Trominos

\[ 2^N \]

\[ 2^N \]

\[ 2^N \]
Tiling with Trominos

$2^N$ $2^N$ $2^N$
Recursion: divide and conquer

Problem: sort the elements of a list

Approach: merge sort
  • If list has only one element, return it
  • Recursively sort 1\textsuperscript{st} half, then 2\textsuperscript{nd} half
  • Merge two sorted lists into one sorted list
Merge Sort

$\log_2 N$ depth

Full binary tree with $N$ leaves has $N-1$ nodes
def sort(L):
    """Returns a new sorted list containing the same elements as L""
    if len(L) < 2:
        return L[:]
    else:
        middle = int(len(L)/2)
        left = sort(L[:middle])
        right = sort(L[middle:]):
        #print('About to merge', left, 'and', right)
        return merge(left, right)
merge(left, right)

def merge(left, right):
    """Assumes left and right are sorted lists. Returns a single new list built, in order, from the elements of left and right."""
    result = []
    i,j = 0, 0
    # while there are elements in both lists
    while i < len(left) and j < len(right):
        # copy smallest element to result
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    # copy over any remaining elements. Only one of the lists # has any elements remaining!
    result.extend(left[i:])
    result.extend(right[j:])
    return result

Don’t use result.append(left.pop(0))