Basics of Recursion

The usual suspects

Mathematical recurrence relations are the canonical examples of recursion:

Factorial:  \( n! = \begin{cases} 
  n & \text{if } n = 1 \\
  n \times (n - 1)! & \text{if } n > 1 
\end{cases} \)

Fibonacci:  \( fib(n) = \begin{cases} 
  1 & \text{if } n \leq 2 \\
  fib(n - 1) + fib(n - 2) & \text{if } n > 2 
\end{cases} \)

Ackermann function:  \( A(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0 \\
  A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\
  A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 
\end{cases} \)

In [4]:
```
def fact(n):
    if n <= 1:
        return n
    else:
        return n * fact(n-1)

print('fact:',[fact(n) for n in range(1,10)])
```

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

print('fib:',[fib(n) for n in range(1,10)])
```

```
def ack(m,n):
    return n+1 if m==0 else ack(m-1,1) if n==0 else ack(m-1,ack(m,n-1))

print("ack(3,2) =",ack(3,2))
```

fact: [1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
fib: [1, 1, 2, 3, 5, 8, 13, 21, 34]
ack(3,2) = 29

Recursion vs. Iteration?

Consider the following two procedures for computing the length of the list. Which would you choose? Why?
What happens if you change the 10 to 100000?

Here's a second example: suppose we wanted to access an element of a N-dimensional array represented as nested lists.

```
# here's a 2x4x2 3-dimensional array
array = [ [ [1, 2],
            [3, 4],
            [5, 6],
            [7, 8]]
       ]
print(array[1][2][0])

# return array element given a N-dimensional coordinate and an N-dimensional array
# 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]])

print(nd_get_iterative((1, 2, 0), array), nd_get_recursive((1, 2, 0), array))
```
Recursion: processing trees

An abstract syntax tree (AST) is a convenient of representing an arbitrary expression involving numbers and arithmetic operators. Here's a description of an AST in Backus-Naur Form (BNF):

```
ast ::= integer | string | [operator, ast, ast, ...]
operator ::= "+" | "*" | ...
```

Here are the rules for determining the value of an AST, given an `environment` (a dictionary mapping variable names to values):

- if `ast` is an integer, return its value
- if `ast` is a string, treat as a variable name: return value of variable
- otherwise, return result of performing operation on values of `ast` operands

Please write the procedure `eval_ast(ast, env)`
In [98]:

# 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),
}

def eval_ast(ast, env={}):
    if isinstance(ast, list):
        operator = notable[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

print(eval_ast(3))
print(eval_ast('pi', {'pi': 3.141592653}))
print(eval_ast(['*', 2, 21]))

# evaluate 3 + 47*5/pi - 13
ast = ['-',
    ['+',
        ['/',
            ['*', 47, 5],
            'pi'],
        3],
    13]
print(eval_ast(ast, {'pi': 3.141592653}))

3
3.141592653
42
64.80282326723406

Here's another tree processing example: each tree node is represented by a three element list [color, left, right] where

- "color" is the color of the node, either "red" or "black"
- "left": another tree node or None if no left child
- "right": another tree node or None if no right child

Please write the function is_proper(root), which returns true if every path from root to leaves has the same number of black nodes. A node is a leaf if it has no children.
In [99]:

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]

root1 = ['black', None, None] #true
root2 = ['red', None, None] #true
root3 = ['black', ['black', None, None], None] #true
root4 = ['black', ['black', None, None], 'black',
         ['red', ['black', None, None],
          ['black', None, None]]] #true
root5 = ['black', ['black', None, None], None] #false
         ['red', ['red', None, None],
          ['black', None, None]]]

print(root1,is_proper(root1))
print(root2,is_proper(root2))
print(root3,is_proper(root3))
print(root4,is_proper(root4))
print(root5,is_proper(root5))

[['black', None, None] True
['red', None, None] True
['black', ['black', None, None], None] True
['black', ['black', None, None], ['red', ['black', None, None], ['black', None, None]]] True
['black', ['black', None, None], ['red', ['red', None, None], ['black', N one, None]]] False

Recursive enumeration

Write a procedure sublist_sums_to_N(num_list, N) if zero or more numbers in num_list
sum to N.

First cut: generate all possible sublists, sum them, see if any sum to N
In [100]:

# return list containing all sublists of L
# including empty list and L itself

def all_sublists(L):
    if len(L) == 0:
        return [[]]  # empty list
    first = [L[0]]
    rest = L[1:]
    result = []
    for ll in all_sublists(rest):
        result.append(ll)
        result.append(first + ll)
    return result

# generate all the sublists, check their sums, see if any match

def sublist_sums_to_N(num_list, N):
    return any(sum(sublist) == N
                for sublist in all_sublists(num_list))

print(sublist_sums_to_N([1, 2, 5, 9, 17], 34))  # true
print(sublist_sums_to_N([1, 2, 5, 9, 17], 22))  # true
print(sublist_sums_to_N([1, 2, 5, 9, 17], 9))  # true
print(sublist_sums_to_N([1, 2, 5, 9, 17], 13))  # false
print(sublist_sums_to_N([1, 2, 5, 9, 17], 0))  # true

True
True
True
False
True

Seems wasteful to generate them all, then check them all. Better to check as we go!
In [105]:

```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

print(list(generate_all_sublists([1, 2, 3])))
```

In [106]:

```python
# 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

print(sublist_sums_to_N([1, 2, 5, 9, 17], 34))  # true
print(sublist_sums_to_N([1, 2, 5, 9, 17], 22))  # true
print(sublist_sums_to_N([1, 2, 5, 9, 17], 9))   # true
print(sublist_sums_to_N([1, 2, 5, 9, 17], 13))  # false
print(sublist_sums_to_N([1, 2, 5, 9, 17], 0))   # true
```

Is there a way to combine the check with the generation of sublists?

In [102]:

```python
def sublist_sums_to_N(num_list, N):
    if len(num_list) == 0:
        return N==0
    first = num_list[0]
    rest = num_list[1:]
    # start by using first in the sum, then try without
    return sublist_sums_to_N(rest, N - first) or \
        sublist_sums_to_N(rest, N)

print(sublist_sums_to_N([1, 2, 5, 9, 17], 34))  # true
print(sublist_sums_to_N([1, 2, 5, 9, 17], 22))  # true
print(sublist_sums_to_N([1, 2, 5, 9, 17], 9))   # true
print(sublist_sums_to_N([1, 2, 5, 9, 17], 13))  # false
print(sublist_sums_to_N([1, 2, 5, 9, 17], 0))   # true
```

True
True
True
False
True
### Merge sort

```python
In [104]:
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)

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

inp = [23, 3, 45, 7, 6, 11, 14, 12]
print(inp)
print(sort(inp))
```

```
[23, 3, 45, 7, 6, 11, 14, 12]
About to merge [23] and [3]
About to merge [45] and [7]
About to merge [3, 23] and [7, 45]
About to merge [14] and [12]
About to merge [6, 11] and [12, 14]
About to merge [3, 7, 23, 45] and [6, 11, 12, 14]
[3, 6, 7, 11, 12, 14, 23, 45]
```