Last week
- Reading recursive functions utilized list comprehension
- Tracing recursive functions
  - **dig down, come up**
  - Trace `max_list([4, 2, [4, 7], 5], 8)`

Today
- More recursive functions
- Tracing recursive functions using stacks
- Recursive structures

More recursive examples

- **Factorial function**
  
  \[ \text{Factorial}(n) = n \times \text{Factorial}(n-1) \]

- **Fibonacci function**
  
  \[ \text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2) \]

A recursive function has at least one **base case** and at least one **recursive case**

Another example

A recursive **definition**: Balanced Strings

- **Base case:**
  - A string containing no parentheses is balanced

- **Recursive cases:**
  - `(x)` is balanced if `x` is a balanced string
  - `xy` is balanced if `x` and `y` are balanced strings

How about these functions?

- \[ f(n) = n^2 + n - 1 \]
- \[ f(n) = g(n-1) + 1, \ g(n) = n/2 \]
- \[ f(n) = 5, \ f(n-1) = 4 \]
- \[ f(n) = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \]
- \[ f(n) = f(n/2) + 1, \ f(1) = 1 \]

Recursive programs

- Solution defined in terms of solutions for smaller problems
  
  ```python
  def solve(n):
      ...
      value = solve(n-1) + solve(n/2)
      ...
  ```

- One or more base cases
  
  ```python
  if (n < 10):
      value = 1
  ```

- Some base case is always reached eventually; otherwise it's an infinite recursion
General form of recursion

if (condition to detect a base case):
    {do something without recursion}
else: (general case)
    {do something that involves recursive call(s)}

Recursive programs cont’d

0! = 1 and n! = n.(n-1)!

```python
def factorial(n):
    # pre: n ≥ 0
    # post: returns n!
    if (n==0): return 1
    else: return n * factorial(n-1)
```

Structure of code typically parallels
structure of definition

Fib(0) = 1, Fib(1) = 1, Fib(n) = Fib(n-1) + Fib(n-2)

```python
def fib(n):
    # pre: n ≥ 0
    # post: returns the n-th Fibonacci number
    if (n < 2): return 1
    else: return fib(n-1) + fib(n-2)
```

Structure of code typically parallels
structure of definition

Stacks and tracing calls

- Recall: stack applications in compilers/interpreters
- tracing method calls
- Activation record
  - all information necessary for tracing a method call
  - such as parameters, local variables, return address, etc.
- When method called:
  - activation record is created, initialized, and pushed onto the stack
- When a method finishes:
  - its activation record (that is on top of the stack) is popped from the stack

Tracing program calls

- Recall: stack of activation records
  - When method called:
    - activation record created, initialized, and pushed onto the stack
  - When a method finishes,
    - its activation record is popped

Stack of activation records

Tracing recursive programs

- same mechanism for recursive programs

Stack of activation records
### Tracing Factorial

1. `def f(n):
2. # pre: n≥0
3. # post: returns n!
4. if (n==0): return 1
5. else: return n * f(n-1)

### Tracing Factorial: intuitively

- `f(3)`

### Tracing max_list(), using stack?

1. `def max_list(L):
2. if isinstance(L, list):
3. return max([max_list(x) for x in L])
4. else:
5. # L is an int
6. return L

Trace `max_list([4, 2, [[4, 7], 5], 8])`

### Tracing max_list(), using stack?

Trace `max_list([4, 2, [[4, 7], 5], 8])`

### Tracing Fibonacci

1. `def fib(n):
2. # pre: n ≥ 0
3. # post: returns the n-th Fibonacci number
4. if (n < 2): return 1
5. else: return fib(n-1) + fib(n-2)

Hint: requires 9 pushes

### Why 9?

- Using rewriting
  
  ![Fibonacci recursion diagram]

- \( \text{fib(4)} \)
- \( \text{fib(3)} + \text{fib(2)} \)
- \( \text{fib(2)} + \text{fib(1)} \)
- \( \text{fib(1)} + \text{fib(0)} \)
Recursive vs iterative

- Recursive functions impose a loop
- The loop is implicit and the compiler/interpreter (here, Python) takes care of it
- This comes at a price: time & memory
- The price may be negligible in many cases

- After all, no recursive function is more efficient than its iterative equivalent

Recursive vs iterative cont’d

- Every recursive function can be written iteratively (by explicit loops)
  - may require stacks too
- yet, when the nature of a problem is recursive, writing it iteratively can be
  - time consuming, and
  - less readable

- So, recursion is a very powerful technique for problems that are naturally recursive

More examples

- Merge Sort
- Quick Sort
- Tower of Hanoi
- Balanced Strings
- Traversing Trees
- In general, properties of Recursive Definitions/Structures

Looking for exercises? Implement the above examples without seeing the sample solutions/algorithm.

Merge sort

```python
Msort(A, i, j)
if (i < j)
    p := partition(A)
    S1 := Msort(A, i , (i+j)/2)
    S2 := Msort(A, (i+j)/2, j)
    Merge(S1, S2, i, j)
end
```

Quick sort

```python
Qsort (A, i, j)
if (i < j)
    p := partition(A)
    Qsort (A, i, p-1)
    Qsort (A, p+1, j)
end
```

Tower of Hanoi

```python
Hanoi (n, s, d, aux)
if (n=1)
    "move from " +s+ " to " +d
else
    Hanoi (n-1, s, aux, d)
    "move from " +s+ " to " +d
    Hanoi (n-1, aux, d, s)
end
```
Let's move on to a new topic.
def height(t):
    """Return 1 + length of longest path of t.
    :param t: tree to find height of
    :type t: Tree
    :rtype: int
    >>> t = Tree(13)
    >>> height(t)
    1
    >>> t = descendants_from_list(Tree(13), [0, 1, 3, 5, 7, 9, 11, 13], 3)
    >>> height(t)
    3
    """
    if len(t.children) == 0:
        # t is a leaf
        return 1
    else:
        # t is an internal node
        return 1 + max([height(c) for c in t.children])

def arity(t):
    """Return the maximum branching factor (arity) of Tree t.
    :param t: tree to find the arity of
    :type t: Tree
    :rtype: int
    >>> t = Tree(23)
    >>> arity(t)
    0
    >>> tn2 = Tree(2, [Tree(4), Tree(4.5), Tree(5), Tree(5.75)])
    >>> tn3 = Tree(3, [Tree(6), Tree(7)])
    >>> tn1 = Tree(1, [tn2, tn3])
    >>> arity(tn1)
    4
    """
    if len(t.children) == 0:
        # t is a leaf
        return 0
    else:
        # t is an internal node
        return max([len([n] + [arity(n) for n in t.children]) for c in t.children])

def count(t):
    """Return the number of nodes in Tree t.
    :param t: tree to find number of nodes in
    :type t: Tree
    :rtype: int
    >>> t = Tree(17)
    >>> count(t)
    1
    >>> t4 = descendants_from_list(Tree(17), [0, 2, 4, 6, 8, 10, 11], 4)
    >>> count(t4)
    8
    """
    if len(t.children) == 0:
        # t is a leaf
        return 1
    else:
        # t is an internal node
        return 1 + sum([count(n) for n in t.children])