Tree Recursion

Tree-shaped processes arise whenever executing the body of a function entails making more than one call to that function.

\[
\begin{align*}
\text{n:} & \quad 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots , 35 \\
\text{fib(n):} & \quad 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots , 5,702,887
\end{align*}
\]

```python
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

A Tree-Recursive Process

The computational process of fib evolves into a tree structure
Repetition in Tree-Recursive Computation

This process is highly repetitive; fib is called on the same argument multiple times.

```
   fib(6)
     /   \
   fib(4) fib(5)
   /     /   \  
fib(2) fib(3) fib(3) fib(4)
   \   \   \   \
   1 fib(1) fib(2) fib(2) \
     \   \   \   \
     0 1   fib(1) fib(2)
          \   \   \
          0 1 1 fib(1) fib(2)
               \   \   \
               0 1   0 1
```
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

*Keys are arguments that map to return values*

Same behavior as f, if f is a pure function

**Demo**
Memoized Tree Recursion

Calls to fib with memoization: 35
Calls to fib without memoization: 18,454,929
Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

```python
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n-1):
        prev, curr = curr, prev + curr
    return curr

@memo
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>n steps</td>
<td>3 names</td>
</tr>
<tr>
<td></td>
<td>Independent of problem size</td>
</tr>
<tr>
<td>n steps</td>
<td>n entries</td>
</tr>
<tr>
<td></td>
<td>Scales with problem size</td>
</tr>
</tbody>
</table>
$1 = \$0.50 + \$0.25 + \$0.10 + \$0.10 + \$0.05$

$1 = 1 \text{ half dollar, 1 quarter, 2 dimes, 1 nickel}$

$1 = 2 \text{ quarters, 2 dimes, 30 pennies}$

$1 = 100 \text{ pennies}$

**How many ways are there to change a dollar?**

**How many ways to change $\$0.11$ with nickels & pennies?**

$\$0.11$ can be changed with nickels & pennies by

A. Not using any more nickels; $\$0.11$ with just pennies

B. Using at least one nickel; $\$0.06$ with nickels & pennies
Counting Change Recursively

How many ways are there to change a dollar?

The number of ways to change an amount $a$ using $n$ kinds =

- The number of ways to change $a$ using all but the first kind 
  +
- The number of ways to change $(a - d)$ using all $n$ kinds, 
  where $d$ is the denomination of the first kind of coin.

```python
def count_change(a, kinds=(50, 25, 10, 5, 1)):

  # base cases
  d = kinds[0]
  return count_change(a, kinds[1:]) + count_change(a-d, kinds)
```

Demo
Space Consumption

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of active environments. Values and frames referenced by active environments are kept. Memory used for other values & frames can be reclaimed.

Active environments:

• The environment for the current expression being evaluated
• All environments for expressions that depend upon the value of the current expression
• All environments associated with values referenced by active environments
fib: 

fib(n):
...

if n == 1:
    return 0
if n == 2:
    return 1
return fib(n-2) + fib(n-1)

n: 3

fib

n: 1

fib

fib(n-2)

if n == 1:
    return 0
if n == 2:
    return 1
return fib(n-2) + fib(n-1)

0
Fibonacci Environment Diagram

fib:  

fib(n):
    ...
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)

n: 2
n: 3
n: 1

fib(n-2):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)

fib(n-1):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
Fibonacci Memory Consumption

Assume we have reached this step
Fibonacci Memory Consumption

Assume we have reached this step

Has an active environment
Can be reclaimed
Hasn't yet been created
Active Environments for Returned Functions

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add1 = make_adder(1)
```

Associated with an environment

Therefore, all frames in this environment must be kept