Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Memoized Tree Recursion

- Call to fib
- Found in cache
- Skipped
The Consumption of Space

Which environment frames do we need to keep during evaluation?
At any moment there is a set of active environments
Values and frames in active environments consume memory
Memory that is used for other values and frames can be recycled

Active environments:
- Environments for any function calls currently being evaluated
- Parent environments of functions named in active environments

Comparing Implementations

Implementations of the same functional abstraction can require different resources

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

def factors(n):
    # Time (number of divisions)
    # Slow: Test each k from 1 through n
    # Fast: Test each k from 1 to square root n
    # For every k, n/k is also a factor!

    Question: How many time does each implementation use division? (Demo)

Order of Growth

A method for bounding the resources used by a function by the “size” of a problem

m: size of the problem
R(n): measurement of some resource used (time or space)

R(n) = Θ(f(n))

means that there are positive constants k₁ and k₂ such that

k₁ f(n) ≤ R(n) ≤ k₂ f(n)

for all n larger than some minimum m
Exponentiation

**Goal:** one more multiplication lets us double the problem size

```python
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)
def square(x):
    return x * x
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)
```

**Comparing Orders of Growth**

### Properties of Orders of Growth

<table>
<thead>
<tr>
<th>Constants: Constant terms do not affect the order of growth of a process</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>( \Theta(100 \cdot n) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logarithms: The base of a logarithm does not affect the order of growth of a process</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta(\log n) )</td>
</tr>
<tr>
<td>( \Theta(\log_2 n) )</td>
</tr>
</tbody>
</table>

<p>| Nesting: When an inner process is repeated for each step  |
| in an outer process, multiply the steps in the outer and |</p>
<table>
<thead>
<tr>
<th>inner processes to find the total number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta(p \cdot n) )</td>
</tr>
</tbody>
</table>

<p>| Lower-order terms: The fastest-growing part of the |</p>
<table>
<thead>
<tr>
<th>computation dominates the total</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta(n^2) )</td>
</tr>
<tr>
<td>( \Theta(n^2 + n) )</td>
</tr>
<tr>
<td>( \Theta(n^2 + 500 \cdot n + \log_2 n + 1000) )</td>
</tr>
</tbody>
</table>

**Comparing orders of growth (n is the problem size)**

<table>
<thead>
<tr>
<th>( \Theta(n^2) )</th>
<th>Exponential growth. Recursive fib takes ( \Theta(n^2) ) steps, where ( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 ) adding the problem scales ( R(n) ) by a factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta(n^2) )</td>
<td>Quadratic growth. E.g., overlap Incrementing ( n ) increases ( R(n) ) by the problem size ( n )</td>
</tr>
<tr>
<td>( \Theta(n) )</td>
<td>Linear growth. E.g., slow factors or exp</td>
</tr>
<tr>
<td>( \Theta(n) )</td>
<td>Square root growth. E.g., factors_fast</td>
</tr>
<tr>
<td>( \Theta(\log n) )</td>
<td>Logarithmic growth. E.g., exp_fast Doubling the problem only increments ( R(n) ).</td>
</tr>
<tr>
<td>( \Theta(1) )</td>
<td>Constant. The problem size doesn't matter</td>
</tr>
</tbody>
</table>