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

- Call to `fib_tree`
- Found in cache

```
fib_tree(35)
Distinct trees with memoization: 25
Distinct trees without memoization: 18,454,929
```
The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

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

A factor \( k \) of \( n \) is a positive integer such that \( n/k \) is also a positive integer.

<table>
<thead>
<tr>
<th>def count_factors(n):</th>
<th>Time (number of divisions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{Slow}: Test each ( k ) from 1 through ( n ).</td>
<td>( n )</td>
</tr>
<tr>
<td>\textbf{Fast}: Test each ( k ) from 1 to square root ( n ). For every ( k ), ( n/k ) is also a factor!</td>
<td>( \lfloor \sqrt{n} \rfloor )</td>
</tr>
</tbody>
</table>

The Consumption of Space

Which environment frames do we need to keep during evaluation?
Each step of evaluation has a set of \textit{active} environments.
Values and frames in active environments consume memory.
Memory used for other values and frames can be recycled.

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

Fibonacci Memory Consumption

Assume we have reached this step

Order of Growth
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)
  \[ R(n) = \Theta(f(n)) \]
  means that there are positive constants \( k_1 \) and \( k_2 \) such that
  \[ k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n) \]
  for sufficiently large values of \( n \).

Counting Factors

Order of growth can still be used, even if we can quantify amounts exactly.

**Problem:** How many factors does a positive integer \( n \) have?

A factor \( k \) of \( n \) is a positive integer such that \( n/k \) is also a positive integer.

**def count_factors(n)**

- **Slow:** Test each \( k \) from 1 to \( n \).
  \[ \Theta(n) \quad \Theta(1) \]
- **Fast:** Test each \( k \) from 1 to \( \sqrt{n} \).
  For every \( k \), \( n/k \) is also a factor!
  \[ \Theta(\sqrt{n}) \quad \Theta(1) \]

Exponentiation

**Goal:** One more multiplication lets us double the problem size.

**def exp(b, n):**

- If \( n = 0 \): return 1
- Else: return \( b \cdot \exp(b, n-1) \)

**def square(x):**

**def fast_exp(b, n):**

- If \( n = 0 \): return 1
- If \( n \) is even: return \( (b^{\log_2(n)})^2 \)
- If \( n \) is odd: return \( b \cdot \fast_exp(b, n-1) \)

**def square(x):**

**def fast_exp(b, n):**

- If \( n = 0 \): return 1
- If \( n \) is even: return \( (b^{1/2})^2 \)
- Else: return \( b \cdot \fast_exp(b, n/2) \)

**def square(x):**

**def fast_exp(b, n):**

- If \( n = 0 \): return 1
- If \( n \) is even: return \( (b^{1/2})^2 \)
- Else: return \( b \cdot \fast_exp(b, n/2) \)
Comparing orders of growth (n is the problem size)

<table>
<thead>
<tr>
<th>Growth Order</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Theta(n^3))</td>
<td>Exponential growth</td>
<td>Recursive fib takes (\Theta(n^3)) steps, where (n = 1 + \sqrt{5}/2 \approx 1.61828) steps, where (n = 1 + \sqrt{5}/2 \approx 1.61828)</td>
</tr>
<tr>
<td>(\Theta(n^2))</td>
<td>Quadratic growth</td>
<td>E.g., operations on all pairs. Incrementing (n) increases (R(n)) by (n).</td>
</tr>
<tr>
<td>(\Theta(n))</td>
<td>Linear growth</td>
<td>Resources scale with the problem.</td>
</tr>
<tr>
<td>(\Theta(\sqrt{n}))</td>
<td>Logarithmic growth</td>
<td>These processes scale well. Doubling the problem only increments (R(n)).</td>
</tr>
<tr>
<td>(\Theta(1))</td>
<td>Constant</td>
<td>The problem size doesn’t matter.</td>
</tr>
</tbody>
</table>