61A Lecture 18

Wednesday, October 16
Announcements

• Homework 6 is due Tuesday 10/22 @ 11:59pm
• Project 3 is due Thursday 10/24 @ 11:59pm
• Midterm 2 is on Monday 10/28 7pm–9pm
• Hog strategy contest winners will be announced on Wednesday 10/16 in lecture
Memoization
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: 35
Distinct trees without memoization: 18,454,929
Time
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.

```python
def count_factors(n):
    pass
```

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

(Demo)
The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames in active environments consume memory.

Memory 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

(Demo)
Fibonacci Memory Consumption

Assume we have reached this step
Fibonacci Memory Consumption

Has an active environment
Can be reclaimed
Hasn't yet been created

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

\[ n: \text{size of the problem} \]

\[ R(n): \text{Measurement of some resource used (time or space)} \]

\[ R(n) = \Theta(f(n)) \]

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 \( n \).
Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

```
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></th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib_iter(n)</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>fib(n)</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>
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):

<table>
<thead>
<tr>
<th>Slow: Test each $k$ from 1 to $n$.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
</tr>
<tr>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fast: Test each $k$ from 1 to square root $n$. For every $k$, $n/k$ is also a factor!</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
</tr>
<tr>
<td>$\Theta(\sqrt{n})$</td>
</tr>
</tbody>
</table>
Exponentiation
Exponentiation

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

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
\cdot b \cdot b^{n-1} & \text{otherwise}
\end{cases}
\]

```python
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)

def square(x):
    return x**2

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

(Demo)
**Exponentiation**

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

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)

def square(x):
    return x*x

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
</tbody>
</table>
Comparing Orders of Growth
Comparing orders of growth (n is the problem size)

\[ \Theta(n^6) \quad \rightarrow \quad \Theta(n^2) \quad \rightarrow \quad \Theta(n) \quad \rightarrow \quad \Theta(\sqrt{n}) \quad \rightarrow \quad \Theta(\log n) \quad \rightarrow \quad \Theta(b^n) \]

- **Exponential growth!** Recursive fib takes \( \Theta(\phi^n) \) steps, where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)
- Incrementing the problem scales \( R(n) \) by a factor.
- **Quadratic growth.** E.g., operations on all pairs.
  - Incrementing \( n \) increases \( R(n) \) by the problem size \( n \).
- **Linear growth.** Resources scale with the problem.
- **Logarithmic growth.** These processes scale well.
  - Doubling the problem only increments \( R(n) \).
- **Constant.** The problem size doesn't matter.