Measuring Efficiency

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Memoization

Idea: Remember the results that have been computed before

```
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

Space
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.

Assume we have reached this step:

\[
\begin{align*}
\text{fib}(5) \quad 
\text{fib}(4) \\
\text{fib}(3) \quad 
\text{fib}(2) \\
\text{fib}(1) \quad 
\text{fib}(0) \\
\text{fib}(1) \quad 
\text{fib}(0) \\
\text{fib}(2) \quad 
\text{fib}(0) \\
\text{fib}(0) \quad 
\end{align*}
\]

Fibonacci Space Consumption

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 \( n \).

**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).

**Order of Growth of Counting Factors**

Implementations of the same functional abstraction can require different amounts of time.

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

A factor \( k \) of \( n \) is a positive integer that evenly divides \( n \).

**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)
Exponentiation

Goal: one more multiplication lets us double the problem size

\[
\text{def exp}(b, n):
\begin{align*}
\text{if } n &= 0 \\
& \quad \text{return } 1 \\
\text{else:} & \quad \text{return } b \times \text{exp}(b, n-1)
\end{align*}
\]

\[
\text{def square}(x):
\begin{align*}
& \quad \text{return } x \times x
\end{align*}
\]

\[
\text{def exp_fast}(b, n):
\begin{align*}
\text{if } n &= 0 \\
& \quad \text{return } 1 \\
\text{elif } n \% 2 &= 0: & \quad \text{return square}(\text{exp_fast}(b, n/2)) \\
\text{else:} & \quad \text{return } b \times \text{exp_fast}(b, n-1)
\end{align*}
\]

Comparing Orders of Growth

Comparing orders of growth (n is the problem size)

\[
\begin{align*}
\Theta(n) & \quad \text{Linear growth. E.g., slow factors or exp} \\
\Theta(n^{1/2}) & \quad \text{Square root growth. E.g., factors_fast} \\
\Theta(\log n) & \quad \text{Logarithmic growth. E.g., exp_fast} \\
\Theta(1) & \quad \text{Constant. The problem size doesn’t matter}
\end{align*}
\]

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

Logarithms: The base of a logarithm does not affect the order of growth of a process

Nesting: when an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

Lower-order terms: The fastest-growing part of the computation dominates the total

<table>
<thead>
<tr>
<th>Order of Growth</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Theta(1))</td>
<td>Constant</td>
<td>The problem size doesn’t matter</td>
</tr>
<tr>
<td>(\Theta(n))</td>
<td>Linear</td>
<td>E.g., slow factors or exp</td>
</tr>
<tr>
<td>(\Theta(n^{1/2}))</td>
<td>Square root growth</td>
<td>E.g., factors_fast</td>
</tr>
<tr>
<td>(\Theta(\log n))</td>
<td>Logarithmic</td>
<td>E.g., exp_fast</td>
</tr>
<tr>
<td>(\Theta(n^2))</td>
<td>Quadratic</td>
<td>E.g., overlap</td>
</tr>
<tr>
<td>(\Theta(n^p))</td>
<td>Exponential</td>
<td>Recursive fib takes (\Theta(n^p)) steps, where (p = \frac{1 + \sqrt{5}}{2} \approx 1.61828)</td>
</tr>
<tr>
<td>(\Theta(n!\cdot n\log n))</td>
<td>Faster than (\Theta(n!\cdot n\log n))</td>
<td>Incrementing the problem scales (R(n)) by a factor</td>
</tr>
</tbody>
</table>

Exponential growth. Recursive fib takes \(\Theta(n^p)\) steps, where \(p = \frac{1 + \sqrt{5}}{2} \approx 1.61828\) Incrementing the problem scales \(R(n)\) by a factor

Quadratic growth. E.g., overlap Incrementing \(n\) increases \(R(n)\) by the problem size \(n\)

Linear growth. E.g., slow factors or exp

Square root growth. E.g., factors_fast

Logarithmic growth. E.g., exp_fast Doubling the problem only increments \(R(n)\).

Upper-order terms: The fastest-growing part of the computation dominates the total

Exponentiation

\[
\begin{align*}
\text{def exp}(b, n):
\begin{align*}
\text{if } n &= 0 \\
& \quad \text{return } 1 \\
\text{else:} & \quad \text{return } b \times \text{exp}(b, n-1)
\end{align*}
\]

\[
\text{def square}(x):
\begin{align*}
& \quad \text{return } x \times x
\end{align*}
\]

\[
\text{def exp_fast}(b, n):
\begin{align*}
\text{if } n &= 0 \\
& \quad \text{return } 1 \\
\text{elif } n \% 2 &= 0: & \quad \text{return square}(\text{exp_fast}(b, n/2)) \\
\text{else:} & \quad \text{return } b \times \text{exp_fast}(b, n-1)
\end{align*}
\]

Comparing Orders of Growth

Comparing orders of growth (n is the problem size)

\[
\begin{align*}
\Theta(n) & \quad \text{Linear growth. E.g., slow factors or exp} \\
\Theta(n^{1/2}) & \quad \text{Square root growth. E.g., factors_fast} \\
\Theta(\log n) & \quad \text{Logarithmic growth. E.g., exp_fast} \\
\Theta(1) & \quad \text{Constant. The problem size doesn’t matter}
\end{align*}
\]

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

Logarithms: The base of a logarithm does not affect the order of growth of a process

Nesting: when an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

Lower-order terms: The fastest-growing part of the computation dominates the total

<table>
<thead>
<tr>
<th>Order of Growth</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Theta(1))</td>
<td>Constant</td>
<td>The problem size doesn’t matter</td>
</tr>
<tr>
<td>(\Theta(n))</td>
<td>Linear</td>
<td>E.g., slow factors or exp</td>
</tr>
<tr>
<td>(\Theta(n^{1/2}))</td>
<td>Square root growth</td>
<td>E.g., factors_fast</td>
</tr>
<tr>
<td>(\Theta(\log n))</td>
<td>Logarithmic</td>
<td>E.g., exp_fast</td>
</tr>
<tr>
<td>(\Theta(n^2))</td>
<td>Quadratic</td>
<td>E.g., overlap</td>
</tr>
<tr>
<td>(\Theta(n^p))</td>
<td>Exponential</td>
<td>Recursive fib takes (\Theta(n^p)) steps, where (p = \frac{1 + \sqrt{5}}{2} \approx 1.61828)</td>
</tr>
</tbody>
</table>

Exponential growth. Recursive fib takes \(\Theta(n^p)\) steps, where \(p = \frac{1 + \sqrt{5}}{2} \approx 1.61828\) Incrementing the problem scales \(R(n)\) by a factor

Quadratic growth. E.g., overlap Incrementing \(n\) increases \(R(n)\) by the problem size \(n\)

Linear growth. E.g., slow factors or exp

Square root growth. E.g., factors_fast

Logarithmic growth. E.g., exp_fast Doubling the problem only increments \(R(n)\).

Upper-order terms: The fastest-growing part of the computation dominates the total