The Consumption of Time

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 \( n \).

```python
def factors(n):
    # Greatest integer less than \( n \)
    u = nth_root(n)
    # Test each \( k \) from 1 to \( \sqrt{n} \)
    for every \( k \), \( n/k \) is also a factor!
    # Greatest integer less than \( \sqrt{n} \)
```

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

Fibonacci Space Consumption

```
Assume we have reached this step
```

Has an active environment

Can be reclaimed

Hasn’t yet been created
Order of Growth

A method for bounding the resources used by a function by the "size" of a problem.

\[
m: \text{size of the problem} \\
R(n): \text{measurement of some resource used (time or space)}
\]

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

means that there are positive constants \(k_1\) and \(k_2\) such that

\[
k_1 f(n) \leq R(n) \leq k_2 f(n)
\]

for all \(n\) larger than some minimum \(m\).

Counting Factors

Number of operations required to count the factors of \(n\) using factors_fast is \(\Theta(\sqrt{n})\).

To check the lower bound, we choose \(k_1 = 1:\)

- Statements outside the while: 4 or 5
- Statements within the while (including header): 3 or 4
- while statement iterations: between \(\sqrt{n}\) and \(\sqrt{n-1}\)
- Total number of statements executed: at least \(4 + 3(\sqrt{n})\)

**Assumption:** every statement, such as addition-then-assignment using the += operator, takes some fixed number of operations to execute

To check the upper bound

- Maximum statements executed: \(5 + k_2 \sqrt{n}\)
- Maximum operations required per statement: some \(p\)
- We choose \(k_2 = 5p\)

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 \(\sqrt{n}\)

For every \(k\), \(n/k\) is also a factor!

**def factors(n):**

**Time** | **Space**
---|---
\(\Theta(n)\) | \(\Theta(1)\)

**def factors(n):**

**Time** | **Space**
---|---
\(\Theta(\sqrt{n})\) | \(\Theta(1)\)

Exponentiation

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

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

\[
\text{if } n == 0: \\
\quad \text{return } 1 \\
\text{else:} \\
\quad \text{return } b \times \text{exp(b, n-1)}
\]

**def square(x):**

\[x^2\]

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

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

**Exponentiation**

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

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

\[
\text{if } n == 0: \\
\quad \text{return } 1 \\
\text{else:} \\
\quad \text{return } b \times \text{exp(b, n-1)}
\]

**def square(x):**

\[x^2\]

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

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

Comparing Orders of Growth
### Properties of Orders of Growth

| Constants: Constant terms do not affect the order of growth of a process |
|-----------------------------|------------------|
| \( \Theta(n^3) \) | \( \Theta(2n) \) | \( \Theta(\frac{1}{n}) \) |

| Logarithms: The base of a logarithm does not affect the order of growth of a process |
|-----------------------------|------------------|
| \( \Theta(\log_2 n) \) | \( \Theta(\log_{10} n) \) | \( \Theta(n) \) |

| 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 |
|-----------------------------|------------------|
| \( n \times n \times \log_2 n \times \log_{10} n \times \ln n \) |

### def overlap(a, b):
```python
count = 0
for item in a:
    if item in b:
        count += 1
return count
```

If \( a \) and \( b \) are both length \( n \), then \( \text{overlap} \) takes \( n^2 \) steps.

### Comparing orders of growth (\( n \) is the problem size)

| \( \Theta(n^3) \) | Exponential growth. Recursive \( fib \) takes \( \Theta(2^n) \) steps, where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61803 \).
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Incrementing the problem scales ( R(n) ) by a factor</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \Theta(n^2) )</th>
<th>Quadratic growth. E.g., ( \text{overlap} ) increasing ( n ) increases ( R(n) ) by the problem size ( n ).</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>( \Theta(n) )</th>
<th>Linear growth. E.g., ( \text{slow_factors} ) or ( \text{exp} )</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>( \Theta(\sqrt{n}) )</th>
<th>Square root growth. E.g., ( \text{factors_fast} )</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>( \Theta(\log n) )</th>
<th>Logarithmic growth. E.g., ( \text{exp_fast} ). Doubling the problem only increments ( R(n) ).</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>( \Theta(1) )</th>
<th>Constant. The problem size doesn’t matter</th>
</tr>
</thead>
</table>