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

<table>
<thead>
<tr>
<th>Time (number of divisions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
</tr>
<tr>
<td>Greatest integer less than $\sqrt{n}$</td>
</tr>
</tbody>
</table>
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.

(Demo)
Assume we have reached this step.

Fibonacci Space Consumption
Fibonacci Space Consumption

Assume we have reached this step

```
fib(5)
   fib(3)
      fib(1)  fib(2)
        1  fib(0)  fib(1)
        0   1
```

```
fib(4)
   fib(2)
     fib(0)  fib(1)
       0  1
```

```
fib(3)
   fib(1)
     fib(0)  fib(1)
       0  1
```

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

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

\( n: \) size of the problem

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

\[ \text{R}(n) = \Theta(f(n)) \]

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

\[ k_1 \cdot f(n) \leq \text{R}(n) \leq k_2 \cdot f(n) \]

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 \( \sqrt{n} - 1 \) and \( \sqrt{n} \)
- Total number of statements executed: at least \( 4 + 3(\sqrt{n} - 1) \)

To check the upper bound
- Maximum statements executed: \( 5 + 4\sqrt{n} \)
- Maximum operations required per statement: some \( p \)
- We choose \( k_2 = 5p \) and \( m = 25 \)

```python
def factors_fast(n):
    sqrt_n = sqrt(n)
    k, total = 1, 0
    while k < sqrt_n:
        if divides(k, n):
            total += 2
        k += 1
    if k * k == n:
        total += 1
    return total
```

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

```python
def factors(n):
    # Space
    # Assumption: integers occupy a fixed amount of space

    # 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!
    Time | Space
    ----------------
    $\Theta(n)$ | $\Theta(1)$
    $\Theta(\sqrt{n})$ | $\Theta(1)$
```

Assumption: integers occupy a fixed amount of space.
Exponentiation
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)
```

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

\[
\begin{align*}
b^n &= \begin{cases} 
1 & \text{if } n = 0 \\
(b \frac{1}{2} n)^2 & \text{if } n \text{ is even} \\
b \cdot b^{n-1} & \text{if } n \text{ is odd}
\end{cases}
\end{align*}
\]
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**2

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)
```

<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
Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process
\[ \Theta(n) \quad \Theta(500 \cdot n) \quad \Theta\left(\frac{1}{500} \cdot n\right) \]

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

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
\[
(\text{Outer: length of } a) + (\text{Inner: length of } b) = n \times \left(\frac{1}{500} \cdot n\right) 
\]

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

If a and b are both length \( n \), then overlap takes \( \Theta(n^2) \) steps

Lower-order terms: The fastest-growing part of the computation dominates the total
\[ \Theta(n^2) \quad \Theta(n^2 + n) \quad \Theta(n^2 + 500 \cdot n + \log_2 n + 1000) \]
Comparing orders of growth (n is the problem size)

- $\Theta(b^n)$: Exponential growth. Recursive $\text{fib}$ takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828$. Incrementing the problem scales $R(n)$ by a factor.
- $\Theta(n^2)$: Quadratic growth. E.g., $\text{overlap}$. Incrementing $n$ increases $R(n)$ by the problem size $n$.
- $\Theta(n)$: Linear growth. E.g., slow $\text{factors}$ or $\text{exp}$.
- $\Theta(\sqrt{n})$: Square root growth. E.g., $\text{factors\_fast}$.
- $\Theta(\log n)$: Logarithmic growth. E.g., $\text{exp\_fast}$.
- $\Theta(1)$: Constant. The problem size doesn't matter.