Growth
Announcements
Measuring Efficiency
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

\[ \text{fib}(5) \]

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

\[
\text{fib}(5) \quad \text{fib}(3)
\]

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

\[
\begin{array}{c}
\text{def } \text{fib}(n):\\
\text{if } n == 0:
\text{return } 0\\n\text{elif } n == 1:
\text{return } 1\\n\text{else:}\\n\text{return fib}(n-2) + \text{fib}(n-1)
\end{array}
\]

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

[Diagram showing the recursive computation of Fibonacci numbers]

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

[Diagram of the recursive computation of the Fibonacci sequence]

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

[Diagram of recursive computation]

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

![Image of the Fibonacci sequence tree](http://en.wikipedia.org/wiki/File:Fibonacci.jpg)
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

\[
def \text{fib}(n):
    \text{if } n == 0:
        \text{return 0}
    \text{elif } n == 1:
        \text{return 1}
    \text{else:}
        \text{return \text{fib}(n-2) + \text{fib}(n-1)}
\]
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

[Diagram of the recursive computation]

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Memoization
Memoization

**Idea:** Remember the results that have been computed before
Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
```
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
```
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
```
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
```
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)
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]
```
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
```
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
**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
**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

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

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

```
fib(3)
  /\  
fib(1) /  
  /\   
fib(0) /  
     0   1
fib(2)
  /\  
fib(0) /  
     0   1
fib(1)
```
Memoized Tree Recursion

Call to fib

fib(5)
  / 
/fib(3) fib(4)
  /   /
/fib(1) fib(2) /fib(0) fib(1)
    /   /
1 fib(0) fib(1) 0 1

0 1

fib(2)
  / 
/fib(0) fib(1)
    /   /
0 fib(0) fib(1) 0 1

1 fib(0) fib(1)
    /   /
0 fib(0) fib(1) 0 1
Memoized Tree Recursion

Call to fib
Found in cache
Memoized Tree Recursion

- Call to `fib`
- Found in cache
- Skipped
Memoized Tree Recursion
Memoized Tree Recursion

- Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

- Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to fib
Found in cache
Skipped
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to fib

Found in cache

Skipped
Memoized Tree Recursion

- Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to \( \text{fib} \)

Found in cache

Skipped
Memoized Tree Recursion

Call to \text{fib} \\
Found in cache \\
Skipped
Memoized Tree Recursion

- **Call to fib**
- **Found in cache**
- **Skipped**
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped

- fib(5)
  - fib(3)
    - fib(1)
      - fib(0)
        - 0
        - 1
    - fib(2)
      - 1
      - fib(1)
        - fib(0)
        - 0
        - 1
  - fib(4)
    - fib(2)
      - fib(1)
        - fib(0)
        - 0
        - 1
      - fib(1)
        - fib(2)
          - 1
          - fib(0)
            - 0
            - 1
          - fib(1)
            - fib(2)
              - 1
              - fib(0)
                - 0
                - 1
        - fib(1)
          - fib(2)
            - 1
            - fib(0)
              - 0
              - 1
    - fib(3)
      - fib(1)
        - fib(2)
          - 1
          - fib(0)
            - 0
            - 1

Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped
Space
The Consumption of Space
The Consumption of Space

Which environment frames do we need to keep during evaluation?
The Consumption of Space

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments
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.
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.
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:
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
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
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)

Interactive Diagram
Fibonacci Space Consumption
Fibonacci Space Consumption

\[ \text{fib}(5) \]
Fibonacci Space Consumption

$\text{fib}(5) \quad \text{fib}(3)$
Fibonacci Space Consumption

\[
\begin{align*}
\text{fib}(5) & \\
\quad & \text{fib}(3) \\
\quad & \text{fib}(4)
\end{align*}
\]
Fibonacci Space Consumption

```
    fib(5)
   /    |
  fib(3) fib(4)
 /    /    |
fib(1) fib(2) fib(1)
 |    |    |
1    fib(0) fib(1)
 |    |
0    1
```
Fibonacci Space Consumption

```
Fibonacci Space Consumption
```

```
<table>
<thead>
<tr>
<th>fib(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(3)</td>
</tr>
<tr>
<td>fib(1)</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>fib(0)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>fib(1)</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>fib(2)</td>
</tr>
<tr>
<td>fib(1)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>fib(0)</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>fib(3)</td>
</tr>
<tr>
<td>fib(2)</td>
</tr>
<tr>
<td>fib(1)</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>fib(1)</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>fib(0)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>fib(1)</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>fib(2)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>fib(1)</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
```
Fibonacci Space Consumption

Assume we have reached this step.
Fibonacci Space Consumption

Assume we have reached this step.
Fibonacci Space Consumption

Assume we have reached this step: $\text{fib}(5) = \text{fib}(4) + \text{fib}(3)$

Has an active environment
Fibonacci Space Consumption

`fib(5)`

`fib(3)`

`fib(1)`

`fib(2)`

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

`fib(4)`

`fib(2)`

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

`fib(3)`

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

Has an active environment
Can be reclaimed

Assume we have reached this step
Fibonacci Space Consumption

Assume we have reached this step

<table>
<thead>
<tr>
<th>fib(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(3)</td>
</tr>
<tr>
<td>fib(1)</td>
</tr>
<tr>
<td>fib(2)</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>fib(0)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>fib(1)</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fib(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(2)</td>
</tr>
<tr>
<td>fib(0)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>fib(1)</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Has an active environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be reclaimed</td>
</tr>
<tr>
<td>Hasn't yet been created</td>
</tr>
</tbody>
</table>
Time
Comparing Implementations
Comparing Implementations

Implementations of the same functional abstraction can require different resources.
Comparing Implementations

Implementations of the same functional abstraction can require different resources

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

```python
def factors(n):
```


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 \( n \)

```python
def factors(n):
    # Slow: Test each k from 1 through n
```

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 \( n \)

```python
def factors(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!
```
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 $n$

```python
def factors(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)```
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 $n$

```python
def factors(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)```
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

```
def factors(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)
```
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 $n$

```python
def factors(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)```
Orders of Growth
Order of Growth
Order of Growth

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

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

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

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

n: size of the problem

R(n): measurement of some resource used (time or space)
Order of Growth

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

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

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

\[ R(n) = \Theta(f(n)) \]
Order of Growth

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

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

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

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

$n$: 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

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$
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 all \( n \) larger than some minimum \( m \)
Order of Growth

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

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

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 all \( n \) larger than some minimum \( m \)
Order of Growth

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

\( n \): 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 all \( n \) larger than some minimum \( m \)
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):
    
    **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!
```
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):
    """Time          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!
```
Order of Growth of Counting Factors

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

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 $n$
    For every $k$, $n/k$ is also a factor!
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
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):
    # Time complexity
    # Space complexity

    # Slow: Test each k from 1 through n
    # Time: \( \Theta(n) \)
    # Space: \( \Theta(1) \)

    # Fast: Test each k from 1 to square root n
    # For every k, n/k is also a factor!
    # Time: \( \Theta(\sqrt{n}) \)
    # Space: \( \Theta(1) \)
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.

def factors(n):

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>$\Theta(\sqrt{n})$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

**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!

Assumption: Integers occupy a fixed amount of 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 $n$

```python
def factors(n):
    # Slow: Test each k from 1 through n
    time_s = \Theta(n)
    space_s = \Theta(1)

    # Fast: Test each k from 1 to square root n
    # For every k, n/k is also a factor!
    time_f = \Theta(\sqrt{n})
    space_f = \Theta(1)
```

(Demo)

Assumption: integers occupy a fixed amount of space
Exponentiation
Exponentiation
Exponentiation

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

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

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

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

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

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

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

Exponentiation

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

```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)
```
## Exponentiation

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

<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>

```python
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)
```
Comparing Orders of Growth
Properties of Orders of Growth
Properties of Orders of Growth

**Constants**: Constant terms do not affect the order of growth of a process
Properties of Orders of Growth

**Constants**: Constant terms do not affect the order of growth of a process $\Theta(n)$.
Properties of Orders of Growth

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

\[ \Theta(n) \quad \Theta(500 \cdot n) \]
Properties of Orders of Growth

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

\[ \Theta(n) \quad \Theta(500 \cdot n) \quad \Theta\left(\frac{1}{500} \cdot n\right) \]
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
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

\[ \Theta(\log_2 n) \]
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

\[ \Theta(\log_2 n) \quad \Theta(\log_{10} n) \]
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

\[
\Theta(\log_2 n) \quad \Theta(\log_{10} n) \quad \Theta(\ln n)
\]
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.
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

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

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

*Outer: length of a*
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

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

**Outer:** length of a

**Inner:** length of b
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

\[
\text{Overlap}\def\mathit{overlap}(a, b):
\text{count} = 0
\text{for item in a:}
\quad\text{if item in b:}
\quad\quad\text{count} += 1
\text{return count}
\]

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

\[
(n) \cdot (500 \cdot n) \cdot (\log 2 n) \cdot (\log_{10} n) \cdot (\ln n)
\]

```python
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
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

\[
\text{count} = 0 \\
\text{for} \text{ item in } a: \\
\quad \text{if} \text{ item in } b: \\
\quad \quad \text{count} += 1 \\
\text{return count}
\]

- **Outer:** length of a
- **Inner:** length of b

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

\[ \Theta(n^2) \]
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

\[
\times \left(n \times 500 \cdot n \times \left(\log_2 n \times \log_{10} n \times \ln n\right)\right)
\]

```python
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

\[ \Theta(n^2) \quad \Theta(n^2 + n) \]
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

\[ \text{def } \text{overlap}(a, b): \]
\[ \text{count} = 0 \]
\[ \text{for } \text{item in } a: \]
\[ \text{if } \text{item in } b: \]
\[ \text{count} += 1 \]
\[ \text{return } \text{count} \]

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)
Comparing orders of growth ($n$ is the problem size)

$\Theta(b^n)$
Comparing orders of growth (n is the problem size)

\[ \Theta(b^n) \quad \text{Exponential growth. Recursive fib takes} \]

\[ \Theta(\varphi^n) \text{ steps, where } \varphi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \]
Comparing orders of growth (n is the problem size)

\( \Theta(b^n) \)  Exponential growth. Recursive \texttt{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
Comparing orders of growth (n is the problem size)

\[ \Theta(b^n) \]

Exponential growth. Recursive \texttt{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

\[ \Theta(n^2) \]
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., overlap
Comparing orders of growth (n is the problem size)

\( \Theta(b^n) \)  Exponential growth. Recursive \texttt{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., \texttt{overlap}
Incrementing \( n \) increases \( R(n) \) by the problem size \( n \)
Comparing orders of growth ($n$ is the problem size)

$\Theta(b^n)$ 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

$\Theta(n^2)$ Quadratic growth. E.g., overlap

Incrementing $n$ increases $R(n)$ by the problem size $n$

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

\( \Theta(b^n) \)  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

\( \Theta(n^2) \)  Quadratic growth. E.g., overlap

Incrementing \( n \) increases \( R(n) \) by the problem size \( n \)

\( \Theta(n) \)  Linear growth. E.g., slow factors or exp
Comparing orders of growth (n is the problem size)

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

\[ \Theta(n^2) \quad \text{Quadratic growth. E.g., overlap} \]
Incrementing n increases R(n) by the problem size n

\[ \Theta(n) \quad \text{Linear growth. E.g., slow factors or exp} \]

\[ \Theta(\sqrt{n}) \]
Comparing orders of growth (n is the problem size)

\( \Theta(b^n) \)  Exponential growth. Recursive \texttt{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., \texttt{overlap}
Incrementing \( n \) increases \( R(n) \) by the problem size \( n \)

\( \Theta(n) \)  Linear growth. E.g., slow \texttt{factors} or \texttt{exp}

\( \Theta(\sqrt{n}) \)  Square root growth. E.g., \texttt{factors\_fast}
Comparing orders of growth (n is the problem size)

<table>
<thead>
<tr>
<th>Order of Growth</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(b^n)$</td>
<td>Exponential growth. Recursive <code>fib</code> takes</td>
<td>$\Theta(\phi^n)$ steps, where $\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828$</td>
</tr>
<tr>
<td>$\Theta(n^2)$</td>
<td>Quadratic growth. E.g., <code>overlap</code></td>
<td>Incrementing the problem scales $R(n)$ by a factor</td>
</tr>
<tr>
<td>$\Theta(n)$</td>
<td>Linear growth. E.g., slow <code>factors</code> or <code>exp</code></td>
<td>Incrementing $n$ increases $R(n)$ by the problem size $n$</td>
</tr>
<tr>
<td>$\Theta(\sqrt{n})$</td>
<td>Square root growth. E.g., <code>factors_fast</code></td>
<td></td>
</tr>
<tr>
<td>$\Theta(\log n)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparing orders of growth (n is the problem size)

\[ \Theta(b^n) \]  Exponential growth. Recursive \texttt{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., \texttt{overlap}
Incrementing \( n \) increases \( R(n) \) by the problem size \( n \)

\[ \Theta(n) \]  Linear growth. E.g., slow \texttt{factors} or \texttt{exp}

\[ \Theta(\sqrt{n}) \]  Square root growth. E.g., \texttt{factors_fast}

\[ \Theta(\log n) \]  Logarithmic growth. E.g., \texttt{exp_fast}
Comparing orders of growth (n is the problem size)

$\Theta(b^n)$  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

$\Theta(n^2)$  Quadratic growth. E.g., `overlap`
Incrementing $n$ increases $R(n)$ by the problem size $n$

$\Theta(n)$  Linear growth. E.g., slow `factors` or `exp`

$\Theta(\sqrt{n})$  Square root growth. E.g., `factors_fast`

$\Theta(\log n)$  Logarithmic growth. E.g., `exp_fast`
Doubling the problem only increments $R(n)$. 
Comparing orders of growth (n is the problem size)

\( \Theta(b^n) \) 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

\( \Theta(n^2) \) Quadratic growth. E.g., overlap
Incrementing n increases R(n) by the problem size n

\( \Theta(n) \) Linear growth. E.g., slow factors or exp

\( \Theta(\sqrt{n}) \) Square root growth. E.g., factors_fast

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

\( \Theta(1) \)
Comparing orders of growth (n is the problem size)

\[ \Theta(b^n) \] 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

\[ \Theta(n^2) \] Quadratic growth. E.g., overlap
Incrementing n increases R(n) by the problem size n

\[ \Theta(n) \] Linear growth. E.g., slow factors or exp

\[ \Theta(\sqrt{n}) \] Square root growth. E.g., factors_fast

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

\[ \Theta(1) \] Constant. The problem size doesn't matter
Comparing orders of growth (n is the problem size)

\[ \Theta(b^n) \] Exponential growth. Recursive \texttt{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., \texttt{overlap} Incrementing n increases R(n) by the problem size n

\[ \Theta(n) \] Linear growth. E.g., slow factors or \texttt{exp}

\[ \Theta(\sqrt{n}) \] Square root growth. E.g., \texttt{factors_fast}

\[ \Theta(\log n) \] Logarithmic growth. E.g., \texttt{exp_fast} Doubling the problem only increments R(n).

\[ \Theta(1) \] Constant. The problem size doesn't matter
Comparing orders of growth (n is the problem size)

\[ \Theta(b^n) \] Exponential growth. Recursive \texttt{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., \texttt{overlap}
Incrementing \( n \) increases \( R(n) \) by the problem size \( n \)

\[ \Theta(n) \] Linear growth. E.g., slow \texttt{factors} or \texttt{exp}

\[ \Theta(\sqrt{n}) \] Square root growth. E.g., \texttt{factors_fast}

\[ \Theta(\log n) \] Logarithmic growth. E.g., \texttt{exp_fast}
Doubling the problem only increments \( R(n) \).

\[ \Theta(1) \] Constant. The problem size doesn't matter