Announcements
Time
The Consumption of Time
The Consumption of Time

Implementations of the same functional abstraction can require different amounts of 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?
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 \( n \)
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 \( n \)

\[
\text{def factors}(n):
\]
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 \( n \).

\[
def \text{factors}(n):
    \]
    \[
    \text{Slow: Test each } k \text{ from 1 through } n
    \]
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 $n$
    For every $k$, $n/k$ is also a factor!
```
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):
    # Time (number of divisions)

    **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!
```
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):
    # Time (number of divisions)
    # 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!
```

\[ n \]
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):
    # Time (number of divisions)
    # 
    # 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!
    # Greatest integer less than $\sqrt{n}$
    # $n$
```

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

\[
\text{Time (number of divisions)}
\]

\[
\begin{array}{c|c}
\text{Fast} & \text{Greatest integer less than } \sqrt{n} \\
\hline
\end{array}
\]


(Demo)
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

fib(5)
Fibonacci Space Consumption

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

\[ \text{fib}(3) \]
Fibonacci Space Consumption

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

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

Assume we have reached this step.
Assume we have reached this step.
Fibonacci Space Consumption

Assume we have reached this step

Has an active environment
Fibonacci Space Consumption

假设我们已经达到了这一步：

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

有活跃的环境
可以回收

Assume we have reached this step

Has an active environment
Can be reclaimed
Fibonacci Space Consumption

Assume we have reached this step.

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

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

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

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

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

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

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

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

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

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

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

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

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

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 \( m \)
Counting Factors
Counting Factors

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

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

```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
```
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 \( k_1 = 1 \):

```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
```
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

```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
```
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

```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
```
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 $\sqrt{n}$

```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
```
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 $4 + 3(\sqrt{n} - 1)$

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

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

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

- Statements outside the \texttt{while}: 4 or 5
- Statements within the \texttt{while} (including header): 3 or 4
- \texttt{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 \textit{upper bound}

```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
```
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: $5 + 4\sqrt{n}$

```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
```
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 $p$

```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
```
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 $p$

```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.
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 $n$

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

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

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):

    # 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></th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow:</td>
<td>( \Theta(n) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Fast:</td>
<td>( \Theta(\sqrt{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):
    # 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
```

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

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

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

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

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

Goal: one more multiplication lets us double the problem size

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

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

def square(x):
    return x**2

def 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

```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)
```

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

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

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

```
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)
```
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

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

**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
```
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

```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
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
```

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

\[
\left( n \right) \cdot \left( \frac{500}{1} \cdot n \right) \cdot \left( \frac{1}{500} \cdot n \right) \cdot \left( \log_2 n \right) \cdot \left( \log_{10} n \right) \cdot \left( \ln n \right)
\]

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

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

\[ \Theta(n^2) \]

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

```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

\[ n \cdot (500 \cdot n) \cdot (1500 \cdot n) \cdot (\log_2 n) \cdot (\log_{10} n) \cdot (\ln n) \]

```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

\[ \Theta(n^2) \quad \Theta(n^2 + n) \quad \Theta(n^2 + 500 \cdot n + \log_2 n + 1000) \]
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)

\[ \Theta(b^n) \] Exponential growth. Recursive fib takes

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

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

\[ \Theta\left(\phi^n\right) \text{ steps, where } \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \]

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 \textit{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., \textit{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 $\text{fib}$ 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

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

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

Θ($b^n$) Exponential growth. Recursive fib takes

Θ($\phi^n$) steps, where $\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor

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

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

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

$\Theta(\sqrt{n})$
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)

\( \Theta(b^n) \) 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}

\( \Theta(\log n) \)
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 \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 \( R(n) \).

\[ \Theta(1) \]
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 `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 \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