Announcements
Hog Contest Rules

[cs61a.org/proj/hog_contest]
Hog Contest Rules

• Up to two people submit one entry;
  Max of one entry per person
Hog Contest Rules

• Up to two people submit one entry;
  Max of one entry per person
• Your score is the number of entries
  against which you win more than
  50.00001% of the time
Hog Contest Rules

• Up to two people submit one entry; 
  Max of one entry per person

• Your score is the number of entries 
  against which you win more than 
  50.00001% of the time

• One more rule! *Pork Chop*
Hog Contest Rules

• Up to two people submit one entry;
  Max of one entry per person

• Your score is the number of entries
  against which you win more than
  50.00001% of the time

• One more rule! Pork Chop

• All strategies must be deterministic,
  pure functions of the current player
  scores
Hog Contest Rules

• Up to two people submit one entry;
  Max of one entry per person
• Your score is the number of entries
  against which you win more than
  50.00001% of the time
• One more rule! *Pork Chop*
• All strategies must be deterministic,
  pure functions of the current player
  scores
• All winning entries will receive 2
  points of extra credit
Hog Contest Rules

• Up to two people submit one entry; Max of one entry per person
• Your score is the number of entries against which you win more than 50.00001% of the time
• One more rule! *Pork Chop*
• All strategies must be deterministic, pure functions of the current player scores
• All winning entries will receive 2 points of extra credit
• The real prize: honor and glory

[cs61a.org/proj/hog_contest](cs61a.org/proj/hog_contest)
Hog Contest Rules

• Up to two people submit one entry; Max of one entry per person
• Your score is the number of entries against which you win more than 50.000001% of the time
• One more rule! Pork Chop
• All strategies must be deterministic, pure functions of the current player scores
• All winning entries will receive 2 points of extra credit
• The real prize: honor and glory

Fall 2011 Winners
Kaylee Mann
Yan Duan & Ziming Li
Brian Prike & Zhenghao Qian
Parker Schuh & Robert Chatham

cs61a.org/proj/hog_contest
Hog Contest Rules

- Up to two people submit one entry; Max of one entry per person
- Your score is the number of entries against which you win more than 50.00001% of the time
- One more rule! *Pork Chop*
- All strategies must be deterministic, pure functions of the current player scores
- All winning entries will receive 2 points of extra credit
- The real prize: honor and glory

Fall 2011 Winners
Kaylee Mann
Yan Duan & Ziming Li
Brian Prike & Zhenghao Qian
Parker Schuh & Robert Chatham

Fall 2012 Winners
Chenyang Yuan
Joseph Hui

cs61a.org/proj/hog_contest
Hog Contest Rules

- Up to two people submit one entry; Max of one entry per person
- Your score is the number of entries against which you win more than 50.00001% of the time
- One more rule! *Pork Chop*
- All strategies must be deterministic, pure functions of the current player scores
- All winning entries will receive 2 points of extra credit
- The real prize: honor and glory

**Fall 2011 Winners**
Kaylee Mann
Yan Duan & Ziming Li
Brian Prike & Zhenghao Qian
Parker Schuh & Robert Chatham

**Fall 2012 Winners**
Chenyang Yuan
Joseph Hui

**Fall 2013 Winners**
Paul Bramsen
Sam Kumar & Kangsik Lee
Kevin Chen

[cs61a.org/proj/hog_contest](cs61a.org/proj/hog_contest)
Hog Contest Rules

• Up to two people submit one entry; Max of one entry per person
• Your score is the number of entries against which you win more than 50.00001% of the time
• One more rule! *Pork Chop*
• All strategies must be deterministic, pure functions of the current player scores
• All winning entries will receive 2 points of extra credit
• The real prize: honor and glory

Fall 2011 Winners
Kaylee Mann
Yan Duan & Ziming Li
Brian Prike & Zhenghao Qian
Parker Schuh & Robert Chatham

Fall 2012 Winners
Chenyang Yuan
Joseph Hui

Fall 2013 Winners
Paul Bramsen
Sam Kumar & Kangsik Lee
Kevin Chen

Fall 2014 Winners
Alan Tong & Elaine Zhao
Zhenyang Zhang
Adam Robert Villaflor & Joany Gao
Zhen Qin & Dian Chen
Zizheng Tai & Yihe Li

[cs61a.org/proj/hog_contest](cs61a.org/proj/hog_contest)
Hog Contest Rules

• Up to two people submit one entry; Max of one entry per person
• Your score is the number of entries against which you win more than 50.00001% of the time
• One more rule! Pork Chop
• All strategies must be deterministic, pure functions of the current player scores
• All winning entries will receive 2 points of extra credit
• The real prize: honor and glory

Fall 2011 Winners
Kaylee Mann
Yan Duan & Ziming Li
Brian Prike & Zhenghao Qian
Parker Schuh & Robert Chatham

Fall 2012 Winners
Chenyang Yuan
Joseph Hui

Fall 2013 Winners
Paul Bramsen
Sam Kumar & Kangsik Lee
Kevin Chen

Fall 2014 Winners
Alan Tong & Elaine Zhao
Zhenyang Zhang
Adam Robert Villaflor & Joany Gao
Zhen Qin & Dian Chen
Zizheng Tai & Yihe Li

Spring 2015 Winners
Sinho Chewi & Alexander Nguyen Tran
Zhaoxi Li
Stella Tao and Yao Ge

(cs61a.org/proj/hog_contest)
Hog Contest Rules

• Up to two people submit one entry; Max of one entry per person
• Your score is the number of entries against which you win more than 50.00001% of the time
• One more rule! Pork Chop
• All strategies must be deterministic, pure functions of the current player scores
• All winning entries will receive 2 points of extra credit
• The real prize: honor and glory

Fall 2011 Winners
Kaylee Mann
Yan Duan & Ziming Li
Brian Prike & Zhenghao Qian
Parker Schuh & Robert Chatham

Fall 2012 Winners
Chenyang Yuan
Joseph Hui

Fall 2013 Winners
Paul Bramsen
Sam Kumar & Kangsik Lee
Kevin Chen

Fall 2014 Winners
Alan Tong & Elaine Zhao
Zhenyang Zhang
Adam Robert Villaflor & Joany Gao
Zhen Qin & Dian Chen
Zizheng Tai & Yihe Li

Spring 2015 Winners
Sinho Chewi & Alexander Nguyen Tran
Zhaoxi Li
Stella Tao and Yao Ge

Fall 2015 Winners...

[Link to Hog Contest website: cs61a.org/proj/hog_contest]
Hog Contest Rules

• Up to two people submit one entry; Max of one entry per person
• Your score is the number of entries against which you win more than 50.00001% of the time
• One more rule! Pork Chop
• All strategies must be deterministic, pure functions of the current player scores
• All winning entries will receive 2 points of extra credit
• The real prize: honor and glory

Fall 2011 Winners
Kaylee Mann
Yan Duan & Ziming Li
Brian Prike & Zhenghao Qian
Parker Schuh & Robert Chatham

Fall 2012 Winners
Chenyang Yuan
Joseph Hui

Fall 2013 Winners
Paul Bramsen
Sam Kumar & Kangsik Lee
Kevin Chen

Fall 2014 Winners
Alan Tong & Elaine Zhao
Zhenyang Zhang
Adam Robert Villaflor & Joany Gao
Zhen Qin & Dian Chen
Zizheng Tai & Yihe Li

Spring 2015 Winners
Sinho Chewi & Alexander Nguyen Tran
Zhaoxi Li
Stella Tao and Yao Ge

Fall 2015 Winners...

cs61a.org/proj/hog_contest
Order of Recursive Calls
The Cascade Function

(Demo)

Interactive Diagram
The Cascade Function

```python
1 def cascade(n):
2     if n < 10:
3         print(n)
4     else:
5         print(n)
6         cascade(n//10)
7         print(n)
8     cascade(123)
```

(Demo)

```
Global frame
```

```
cascade
```

```
f1: cascade [parent=Global]
```

```
f2: cascade [parent=Global]
```

```
f3: cascade [parent=Global]
```

Interactive Diagram
The Cascade Function

```python
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
print(n)
cascade(123)
```

Program output:

```
123
12
1
12
```

(Demo)

Interactive Diagram
The Cascade Function

```
1 def cascade(n):
2     if n < 10:
3         print(n)
4     else:
5         print(n)
6         cascade(n//10)
7         print(n)
8  cascade(123)
```

Program output:
```
123
12
1
12
```

(Demo)

Interactive Diagram

- Each cascade frame is from a different call to `cascade`. 
The Cascade Function

1. `def cascade(n):`
2. `if n < 10:`
   3. `print(n)`
   4. `else:`
   5. `print(n)`
   6. `cascade(n//10)`
   7. `print(n)`
8. `cascade(123)`

Program output:

1. 123
2. 12
3. 1
4. 12

(Demo)

- Each cascade frame is from a different call to `cascade`.
- Until the Return value appears, that call has not completed.

Interactive Diagram
The Cascade Function

1. `def cascade(n):`
2. `    if n < 10:
   print(n)
3. `else:
5. `    print(n)
6. `    cascade(n//10)
8. `print(n)

Program output:
123
12
1
12

(Demo)

- Each cascade frame is from a different call to cascade.
- Until the Return value appears, that call has not completed.
- Any statement can appear before or after the recursive call.

Interactive Diagram
The Cascade Function

```python
1 def cascade(n):
2     if n < 10:
3         print(n)
4     else:
5         print(n)
6         cascade(n//10)
7         print(n)
8
cascade(123)
```

Program output:

```
123
12
1
12
```

(Demo)

- Each cascade frame is from a different call to cascade.
- Until the Return value appears, that call has not completed.
- Any statement can appear before or after the recursive call.

Interactive Diagram
The Cascade Function

```python
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
    print(n)
cascade(123)
```

Program output:

123
12
1
12

(Demo)

- Each cascade frame is from a different call to cascade.
- Until the Return value appears, that call has not completed.
- Any statement can appear before or after the recursive call.
The Cascade Function

```python
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
    print(n)
cascade(123)
```

(Demo)

- Each cascade frame is from a different call to cascade.
- Until the Return value appears, that call has not completed.
- Any statement can appear before or after the recursive call.

Program output:

```
123
12
1
12
```
The Cascade Function

```python
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
    print(n)
cascade(123)
```

(Demo)

- Each cascade frame is from a different call to cascade.
- Until the Return value appears, that call has not completed.
- Any statement can appear before or after the recursive call.

Program output:

123
12
1
12

Interactive Diagram
Two Definitions of Cascade

(Demo)
Two Definitions of Cascade

(Demo)

```python
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)

def cascade(n):
    print(n)
    if n >= 10:
        cascade(n//10)
        print(n)
```
Two Definitions of Cascade

(Demo)

```python
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)

def cascade(n):
    print(n)
    if n >= 10:
        cascade(n//10)
        print(n)
```

- If two implementations are equally clear, then shorter is usually better
Two Definitions of Cascade

(Demo)

```python
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n // 10)
        print(n)

def cascade(n):
    print(n)
    if n >= 10:
        cascade(n // 10)
        print(n)
```

- If two implementations are equally clear, then shorter is usually better
- In this case, the longer implementation is more clear (at least to me)
Two Definitions of Cascade

(Demo)

```python
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)
```

```python
def cascade(n):
    print(n)
    if n >= 10:
        cascade(n//10)
        print(n)
```

• If two implementations are equally clear, then shorter is usually better
• In this case, the longer implementation is more clear (at least to me)
• When learning to write recursive functions, put the base cases first
Two Definitions of Cascade

(Demo)

```python
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)
```

```python
def cascade(n):
    print(n)
    if n >= 10:
        cascade(n//10)
        print(n)
```

• If two implementations are equally clear, then shorter is usually better
• In this case, the longer implementation is more clear (at least to me)
• When learning to write recursive functions, put the base cases first
• Both are recursive functions, even though only the first has typical structure
Example: Inverse Cascade
Inverse Cascade

Write a function that prints an inverse cascade:
Inverse Cascade

Write a function that prints an inverse cascade:

1
12
123
1234
123
12
1
1
Inverse Cascade

Write a function that prints an inverse cascade:

```python
def inverse_cascade(n):
grow(n)
    print(n)
shrink(n)
```
Inverse Cascade

Write a function that prints an inverse cascade:

```python
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)

def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)
```
Inverse Cascade

Write a function that prints an inverse cascade:

```python
1
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)

1234
123
def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)

grow = lambda n: f_then_g(grow, print, n // 10)
shrink = lambda n: f_then_g(print, shrink, n // 10)
```
Inverse Cascade

Write a function that prints an inverse cascade:

```
def inverse_cascade(n):
grow(n)
print(n)
shrink(n)

def f_then_g(f, g, n):
    if n:
        f(n)
g(n)
```

grow = lambda n: f_then_g(grow, print, n//10)
shrink = lambda n: f_then_g(print, shrink, n//10)
Tree Recursion
Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call
Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

\[ n: \ 0, 1, 2, 3, 4, 5, 6, \ 7, \ 8, \]

Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

\[
\begin{align*}
n & : 0, 1, 2, 3, 4, 5, 6, 7, 8, \\
fib(n) & : 0, 1, 1, 2, 3, 5, 8, 13, 21,
\end{align*}
\]
Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

\[
\begin{align*}
n &: 0, 1, 2, 3, 4, 5, 6, 7, 8, \ldots, 35 \\
\text{fib}(n) &: 0, 1, 1, 2, 3, 5, 8, 13, 21,
\end{align*}
\]

Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

\[
\begin{align*}
  n & : 0, 1, 2, 3, 4, 5, 6, 7, 8, \ldots , 35 \\
  \text{fib}(n) & : 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots , 9,227,465
\end{align*}
\]

Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

\[
\begin{align*}
\text{n:} & \quad 0, 1, 2, 3, 4, 5, 6, 7, 8, \ldots, 35 \\
\text{fib(n):} & \quad 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots, 9,227,465 \\
\end{align*}
\]

```python
def fib(n):
    ...
```

Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call.

\[
\begin{align*}
n: & \quad 0, 1, 2, 3, 4, 5, 6, 7, 8, \ldots, 35 \\
\text{fib}(n): & \quad 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots, 9,227,465
\end{align*}
\]

```python
def fib(n):
    if n == 0:
```

Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

\[
\begin{align*}
\text{n:} & \quad 0, 1, 2, 3, 4, 5, 6, 7, 8, \quad \ldots, \quad 35 \\
\text{fib(n):} & \quad 0, 1, 1, 2, 3, 5, 8, 13, 21, \quad \ldots, \quad 9,227,465
\end{align*}
\]

```python
def fib(n):
    if n == 0:
        return 0
```
Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

\[
\begin{align*}
\text{n: } & 0, 1, 2, 3, 4, 5, 6, 7, 8, \ldots, 35 \\
\text{fib(n): } & 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots, 9,227,465 \\
\end{align*}
\]

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
```


Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

\[
\begin{align*}
n: & \quad 0, 1, 2, 3, 4, 5, 6, 7, 8, \ldots, 35 \\
fib(n): & \quad 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots, 9,227,465
\end{align*}
\]

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
```

Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call.

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

Tree Recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

\[
\begin{align*}
n & : \ 0, \ 1, \ 2, \ 3, \ 4, \ 5, \ 6, \ 7, \ 8, \ \ldots, \ 35 \\
\text{fib}(n) & : \ 0, \ 1, \ 1, \ 2, \ 3, \ 5, \ 8, \ 13, \ 21, \ \ldots, \ 9,227,465
\end{align*}
\]

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

A Tree-Recursive Process

The computational process of fib evolves into a tree structure
A Tree-Recursive Process

The computational process of fib evolves into a tree structure

\[ \text{fib}(5) \]
A Tree-Recursive Process

The computational process of fib evolves into a tree structure

```
fib(5)
  |
  |   fib(3)
```
A Tree-Recursive Process

The computational process of fib evolves into a tree structure

```
    fib(5)
   /    \
  /      \
fib(3)   fib(4)
```
A Tree-Recursive Process

The computational process of fib evolves into a tree structure.
A Tree-Recursive Process

The computational process of fib evolves into a tree structure.
A Tree-Recursive Process

The computational process of fib evolves into a tree structure
A Tree-Recursive Process

The computational process of fib evolves into a tree structure.
A Tree-Recursive Process

The computational process of fib evolves into a tree structure.
The computational process of fib evolves into a tree structure.
A Tree-Recursive Process

The computational process of fib evolves into a tree structure.
A Tree-Recursive Process

The computational process of fib evolves into a tree structure.

```
<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>
</tbody>
</table>

fib(4)

fib(2)

fib(0) fib(1)
0 1

fib(3)

fib(0) fib(1)
1 fib(2)
1 fib(0) fib(1)
0 1
```
A Tree-Recursive Process

The computational process of fib evolves into a tree structure.
A Tree-Recursive Process

The computational process of $\text{fib}$ evolves into a tree structure
A Tree-Recursive Process

The computational process of fib evolves into a tree structure
A Tree-Recursive Process

The computational process of fib evolves into a tree structure.
A Tree-Recursive Process

The computational process of fib evolves into a tree structure
A Tree-Recursive Process

The computational process of fib evolves into a tree structure.
A Tree-Recursive Process

The computational process of fib evolves into a tree structure.
A Tree-Recursive Process

The computational process of fib evolves into a tree structure
A Tree-Recursive Process

The computational process of fib evolves into a tree structure.
A Tree-Recursive Process

The computational process of fib evolves into a tree structure.
A Tree-Recursive Process

The computational process of fib evolves into a tree structure.
Repetition in Tree-Recursive Computation
Repetition in Tree-Recursive Computation

This process is highly repetitive; fib is called on the same argument multiple times.
Repetition in Tree-Recursive Computation

This process is highly repetitive; fib is called on the same argument multiple times.
Repeticion in Tree-Recursive Computation

This process is highly repetitive; fib is called on the same argument multiple times.

(We will speed up this computation dramatically in a few weeks by remembering results.)
Example: Counting Partitions
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

```python
count_partitions(6, 4)
```
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

```
count_partitions(6, 4)
```

```
2 + 4 = 6
1 + 1 + 4 = 6
3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6
2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6
1 + 1 + 1 + 1 + 1 + 1 = 6
```
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

$count\_partitions(6, 4)$

2 + 4 = 6
1 + 1 + 4 = 6
3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6
2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6
1 + 1 + 1 + 1 + 1 + 1 = 6
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

$$\text{count_partitions}(6, 4)$$

2 + 4 = 6
1 + 1 + 4 = 6
3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6
2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6
1 + 1 + 1 + 1 + 1 + 1 = 6

[Diagram of partitions]
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

count_partitions(6, 4)

2 + 4 = 6
1 + 1 + 4 = 6
3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6
2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6
1 + 1 + 1 + 1 + 1 + 1 = 6
Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

\[
\text{count_partitions}(6, 4)
\]
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

```
count_partitions(6, 4)
```

*Recursive decomposition: finding simpler instances of the problem.*
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

$\text{count\_partitions}(6, 4)$

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

$count\_partitions(6, 4)$

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

\[
\text{count_partitions}(6, 4)
\]

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

\[
\text{count_partitions}(6, 4)
\]

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

\[
\text{count_partitions}(6, 4)
\]

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

```
count_partitions(6, 4)
```

• Recursive decomposition: finding simpler instances of the problem.
• Explore two possibilities:
  • Use at least one 4
  • Don't use any 4
• Solve two simpler problems:
Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

\[
\text{count_partitions}(6, 4)
\]

• Recursive decomposition: finding simpler instances of the problem.
• Explore two possibilities:
  • Use at least one 4
  • Don't use any 4
• Solve two simpler problems:
  • \( \text{count_partitions}(2, 4) \)
Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

\[
\text{count_partitions}(6, 4)
\]

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - \( \text{count_partitions}(2, 4) \)
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

$$\text{count\_partitions}(6, 4)$$

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - $\text{count\_partitions}(2, 4)$
  - $\text{count\_partitions}(6, 3)$
Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

\[
\text{count_partitions}(6, 4)
\]

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - \( \text{count_partitions}(2, 4) \)
  - \( \text{count_partitions}(6, 3) \)
Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

\[
\text{count_partitions}(6, 4)
\]

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - \( \text{count_partitions}(2, 4) \)
  - \( \text{count_partitions}(6, 3) \)
- Tree recursion often involves exploring different choices.
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

\[
\text{count_partitions}(6, 4)
\]

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(2, 4)
  - count_partitions(6, 3)
- Tree recursion often involves exploring different choices.
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

$count\_partitions(6, 4)$

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - $count\_partitions(2, 4)$
  - $count\_partitions(6, 3)$
- Tree recursion often involves exploring different choices.
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

$$\text{count_partitions}(6, 4)$$

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(2, 4)
  - count_partitions(6, 3)
- Tree recursion often involves exploring different choices.
Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - \( \text{count_partitions}(2, 4) \)
  - \( \text{count_partitions}(6, 3) \)
- Tree recursion often involves exploring different choices.
The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(2, 4)
  - count_partitions(6, 3)
- Tree recursion often involves exploring different choices.

def count_partitions(n, m):

Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - \( \text{count_partitions}(2, 4) \)
  - \( \text{count_partitions}(6, 3) \)
- Tree recursion often involves exploring different choices.

```python
def count_partitions(n, m):
    # Implementation goes here
```
Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(2, 4)
  - count_partitions(6, 3)
- Tree recursion often involves exploring different choices.

```python
def count_partitions(n, m):
    if n == m:
        return 1
    elif n < m:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        return count_partitions(n-m, m) + with_m
```


Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - \( \text{count_partitions}(2, 4) \)
  - \( \text{count_partitions}(6, 3) \)
- Tree recursion often involves exploring different choices.

```python
def count_partitions(n, m):
    if n == 0:
        return 1
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(2, 4)
  - count_partitions(6, 3)
- Tree recursion often involves exploring different choices.

```python
def count_partitions(n, m):
    if m > n:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```
Counting Partitions

The number of partitions of a positive integer n, using parts up to size m, is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

• Recursive decomposition: finding simpler instances of the problem.
• Explore two possibilities:
  • Use at least one 4
  • Don't use any 4
• Solve two simpler problems:
  • count_partitions(2, 4)
  • count_partitions(6, 3)
• Tree recursion often involves exploring different choices.

```python
def count_partitions(n, m):
    if m > n:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - $\text{count_partitions}(2, 4)$
  - $\text{count_partitions}(6, 3)$
- Tree recursion often involves exploring different choices.

```python
def count_partitions(n, m):
    if m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```
Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

- Recursive decomposition: finding simpler instances of the problem.

- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4

- Solve two simpler problems:
  - count_partitions(2, 4)
  - count_partitions(6, 3)

- Tree recursion often involves exploring different choices.

```python
def count_partitions(n, m):
    if n == 0:
        return 1
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - $\text{count\_partitions}(2, 4)$
  - $\text{count\_partitions}(6, 3)$
- Tree recursion often involves exploring different choices.

```python
def count_partitions(n, m):
    if n == 0:
        return 1
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```
Counting Partitions

The number of partitions of a positive integer n, using parts up to size m, is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(2, 4)
  - count_partitions(6, 3)
- Tree recursion often involves exploring different choices.

```python
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```
Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - `count_partitions(2, 4)`
  - `count_partitions(6, 3)`
- Tree recursion often involves exploring different choices.

```python
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```
Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - \( \text{count_partitions}(2, 4) \)
  - \( \text{count_partitions}(6, 3) \)
- Tree recursion often involves exploring different choices.

```python
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```
Counting Partitions

The number of partitions of a positive integer \( n \), using parts up to size \( m \), is the number of ways in which \( n \) can be expressed as the sum of positive integer parts up to \( m \) in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - \( \text{count_partitions}(2, 4) \)
  - \( \text{count_partitions}(6, 3) \)
- Tree recursion often involves exploring different choices.

```python
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```
Counting Partitions

The number of partitions of a positive integer $n$, using parts up to size $m$, is the number of ways in which $n$ can be expressed as the sum of positive integer parts up to $m$ in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(2, 4)
  - count_partitions(6, 3)
- Tree recursion often involves exploring different choices.

```python
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

(Demo)