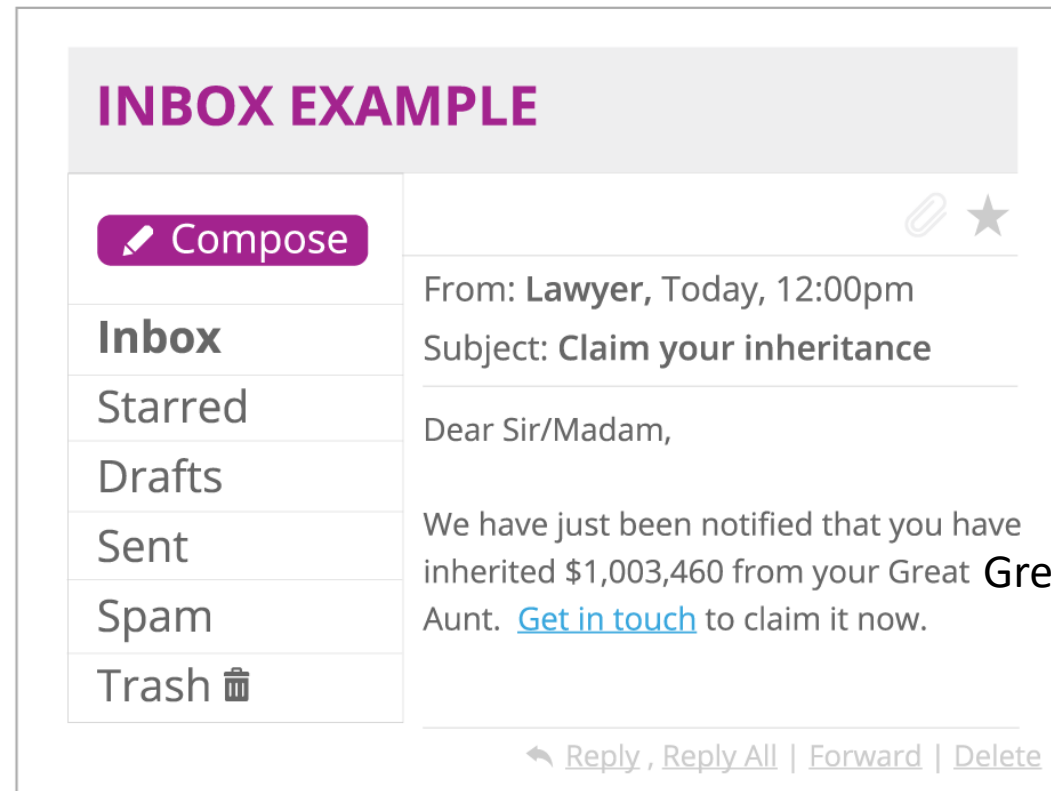


In an alternate world, the average human lifespan is 1000000 years. Then, one day...



t Great Great Great Great Great Great

In an alternate world, the average human lifespan is 10000000 years. Then, one day...

- You receive an email from someone claiming to be your ancestor
- You don't keep entire family records
- How do you know if the claim is legit?
- Assume it's possible for every person in the world to directly contact their parents, but not anyone older

# The Solution

- In order to verify if someone is your ancestor...
  1. Ask your dad to verify if the person is his ancestor *using the same method*
  2. Ask your mom to verify if the person is her ancestor *using the same method*
  3. The person is your ancestor only if either of them say yes
- There is a *repetitive process* that will happen here
  - Dad/Mom will ask his/her parents, who will ask their parents, who will...
- But instead of specifying all the steps, we just tell the next person in line to *do the same thing*
- Expressing a repetitive process or structure in terms of itself is called **recursion**

In “Pseudo-Code”...

```
def is_ancestor(self, person):  
    return (  
        is_ancestor(dad(self), person) or  
        is_ancestor(mom(self), person)  
    )
```

# Wait, But Why?

- You already know some ways to express repetitive processes
  - For-loops and while-loops
- All these ways, including recursion, are shortcuts we use so that the number of steps to explain how to do something is way less than the number of steps needed to actually do it
  - This works because often, a many-step process is simple and “boring” enough that all steps can be described by a *single*, general rule
  - Subtler point: this is the only way if we don’t know exact number of steps in advance
- However, recursion is sometimes *easier*, simpler, and more elegant
  - *Particularly when the process needs to branch off into many parallel parts*
    - Try it: explain solution to ancestor problem using for-loops and while-loops – it’s hard isn’t it?

# Why Is This Valid?

- Everyone asked in this process is trying to solve the same problem:  
determine if a person is their ancestor
- It seems like this is a circular definition

# Smaller Subproblems

- However, each person has a *different*, smaller **subproblem**
  - Your dad/mom has a “smaller” problem because their part of the task involves fewer people
- The actual work done by each person may be the same, but younger people have a “harder” task because they need to depend on more people to finish it

# There's Something Missing...

- Eventually, some person will be asked who obviously knows if the person is their ancestor or not
  - The person is their mom or dad
- In this case, no need for the person to carry out the recursive step



# Base Case

```
def is_ancestor(self, person):  
    if dad(self) == person or mom(self) == person:  
        return True  
    else:  
        return (  
            is_ancestor(dad(self), person) or  
            is_ancestor(mom(self), person)  
        )
```

# Don't Forget to Consider All Base Cases (There Can Be Many)

```
def is_ancestor(self, person):  
    if dad(self) == person or mom(self) == person:  
        return True  
    elif is_dead(self):  
        return False  
    else:  
        return (  
            is_ancestor(dad(self), person) or  
            is_ancestor(mom(self), person)  
        )
```

# Two Features of a Correct Recursive Algorithm

- The same procedure is done on a **smaller subproblem**
  - Typically, the smaller version is *one step away* from the original problem
- The problem eventually becomes small enough to solve in a non-recursive way, expressed as **base cases**
- “Smaller” just means there must be progress towards the base case, in the same way that loops must progress towards the terminating condition
  - Often literally smaller, but not necessarily
  - make problem smaller : loop increment :: base case : terminating condition

# This Feels Like Magic...

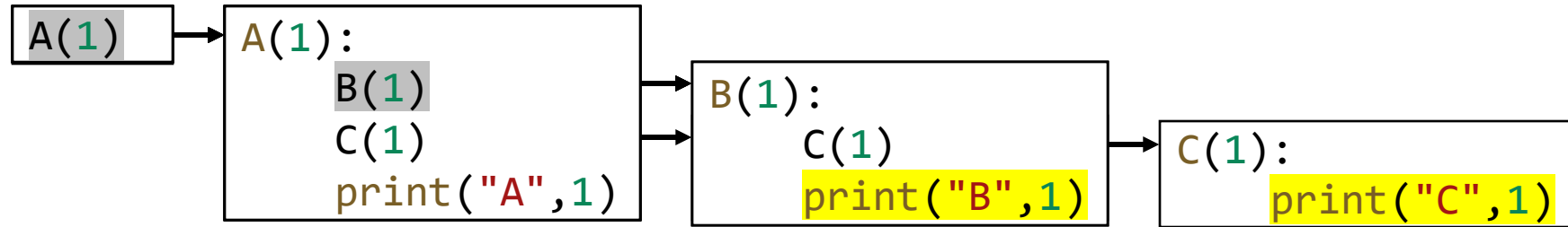
- But recursive functions are just functions
- Computers execute all function calls the same way, regardless of whether it's recursive or not
- We can trace recursive functions the way we trace non-recursive ones

# Recall How Function Calls Get Executed

- Stop where we are in the code
- Jump to the code in our function and run that
- When it's done, return to the point where we left off
- Resume the code as normal

# Review: How Function Calls Get Executed

```
def A(x):  
    B(x)  
    C(x)  
    print("A", x)
```



```
def B(x):  
    C(x)  
    print("B", x)
```

```
def C(x):  
    print("C", x)
```

`A(1)`


**Output:**

C  
B  
C  
A


# Example: Recursive Countdown

```
def count_down(n):  
    if n == 0:  
        print("Blast off!")  
    else:  
        print("Counting down " + str(n))  
        count_down(n - 1)  
  
count_down(4)
```


```
count_down(4):  
    print("Counting down " + str(4))  
    count_down(4 - 1)
```




```
count_down(3):  
    print("Counting down " + str(3))  
    count_down(3 - 1)
```



```
count_down(2):  
    print("Counting down " + str(2))  
    count_down(2 - 1)
```



```
count_down(1):  
    print("Counting down " + str(1))  
    count_down(1 - 1)
```

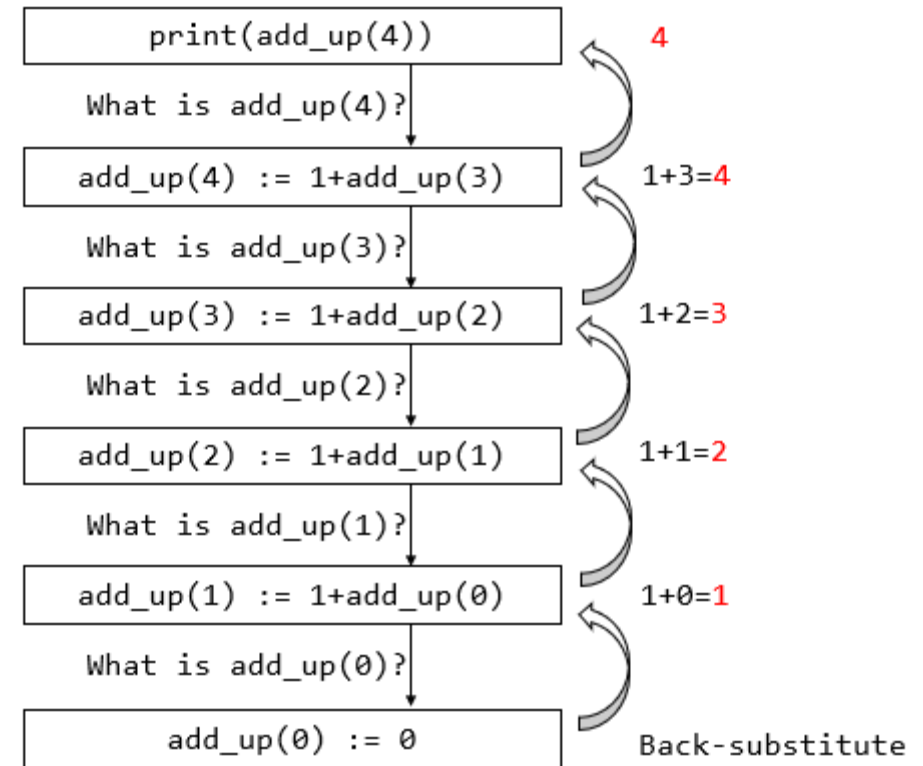


```
count_down(0):  
    print("Blast off!")
```



# Example: With Return Value

```
def add_up(n):  
    if n == 0:  
        return 0  
    else:  
        return 1 + add_up(n - 1)  
  
print(add_up(4))
```



# Example: Multiple Recursive Calls

```
def show(n, lim):  
    if n > lim:  
        return  
    print(n)  
    show(2 * n, lim)  
    show(2 * n + 1, lim)  
  
show(1, 10)
```

```
def show(1, 10):  
    → if 1 > 10:  
        return  
    print(1)  
    show(2*1, 10)  
    show(2*1+1, 10)
```

Output:

show(1, 10)

1

First, we call show(1, 10).

```
def show(1, 10):  
    if 1 > 10:  
        return  
→ print(1)  
   show(2*1, 10)  
   show(2*1+1, 10)
```

Output:  
1

show(**1**, **10**)

1

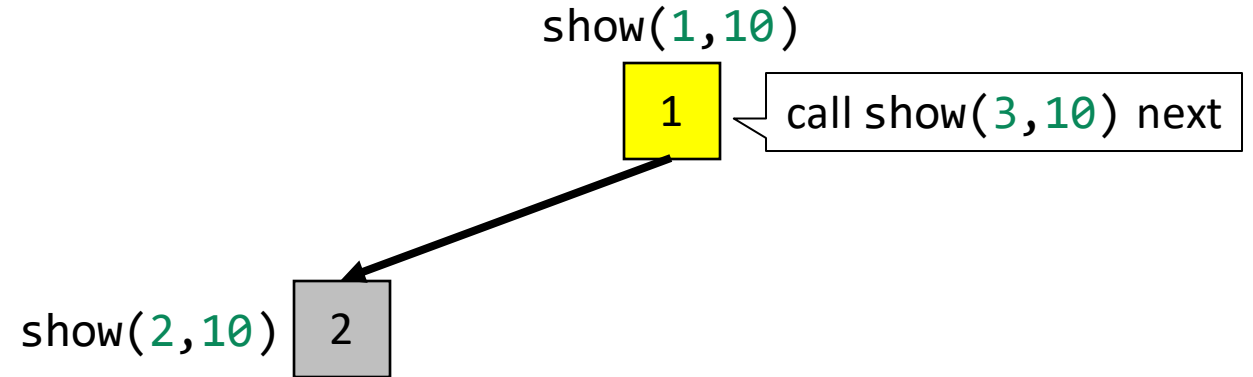
First, we call show(**1**, **10**).

Since  $1 \leq 10$ , we skip the if statement, and print 1.

```
def show(1, 10):  
    if 1 > 10:  
        return  
    print(1)  
    show(2*1, 10)  
    show(2*1+1, 10)
```



Output:  
1



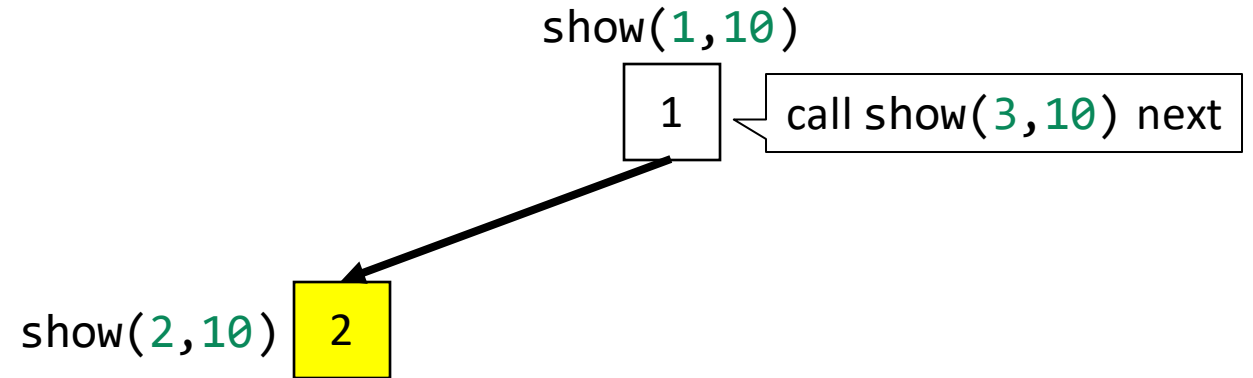
First, we call `show(1, 10)`.  
Since `1 <= 10`, we skip the if statement, and print 1.  
Then, we see that we should call `show(2, 10)` followed by `show(3, 10)`.  
We call `show(2, 10)` first.

```
def show(2, 10):  
    if 2 > 10:  
        return  
    print(2)  
    show(2*2, 10)  
    show(2*2+1, 10)
```



Output:

1  
2



We are now in `show(2, 10)`.

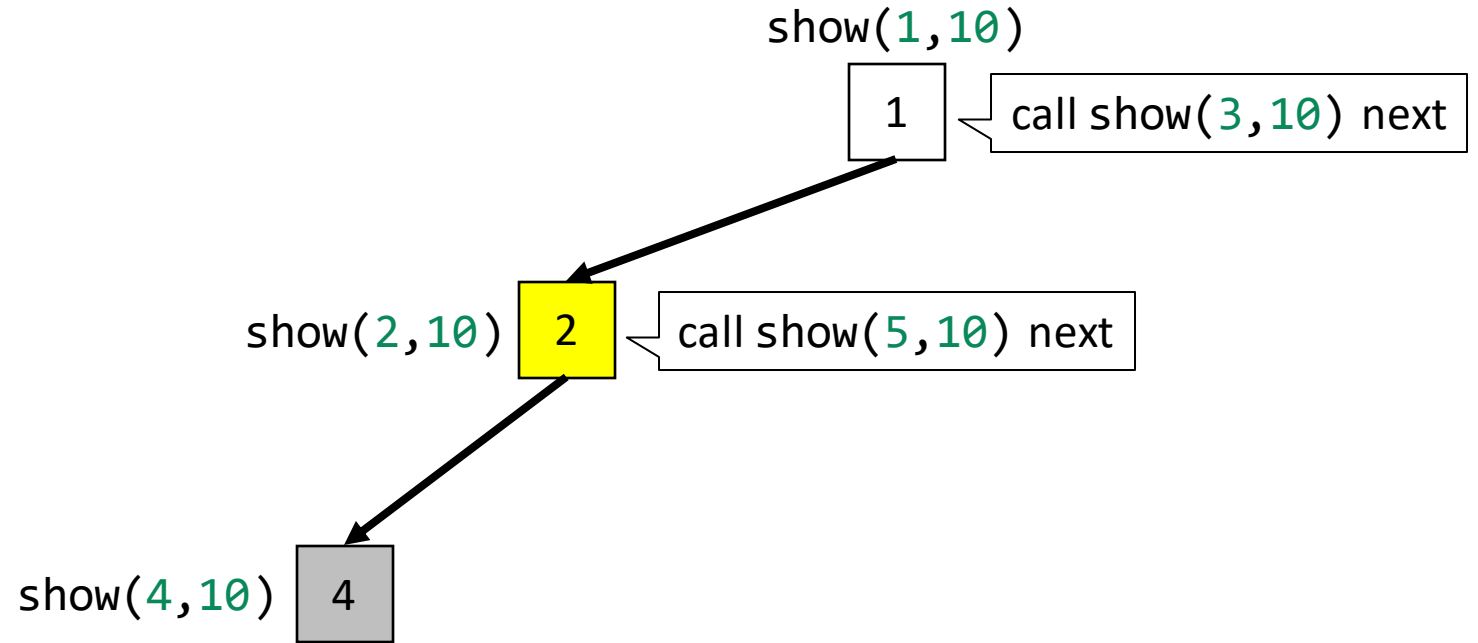
Since `2 <= 10`, we skip the if statement, and print 2.

```
def show(2, 10):  
    if 2 > 10:  
        return  
    print(2)  
    show(2*2, 10)  
    show(2*2+1, 10)
```



Output:

1  
2



We are now in `show(2, 10)`.

Since `2 <= 10`, we skip the if statement, and print 2.

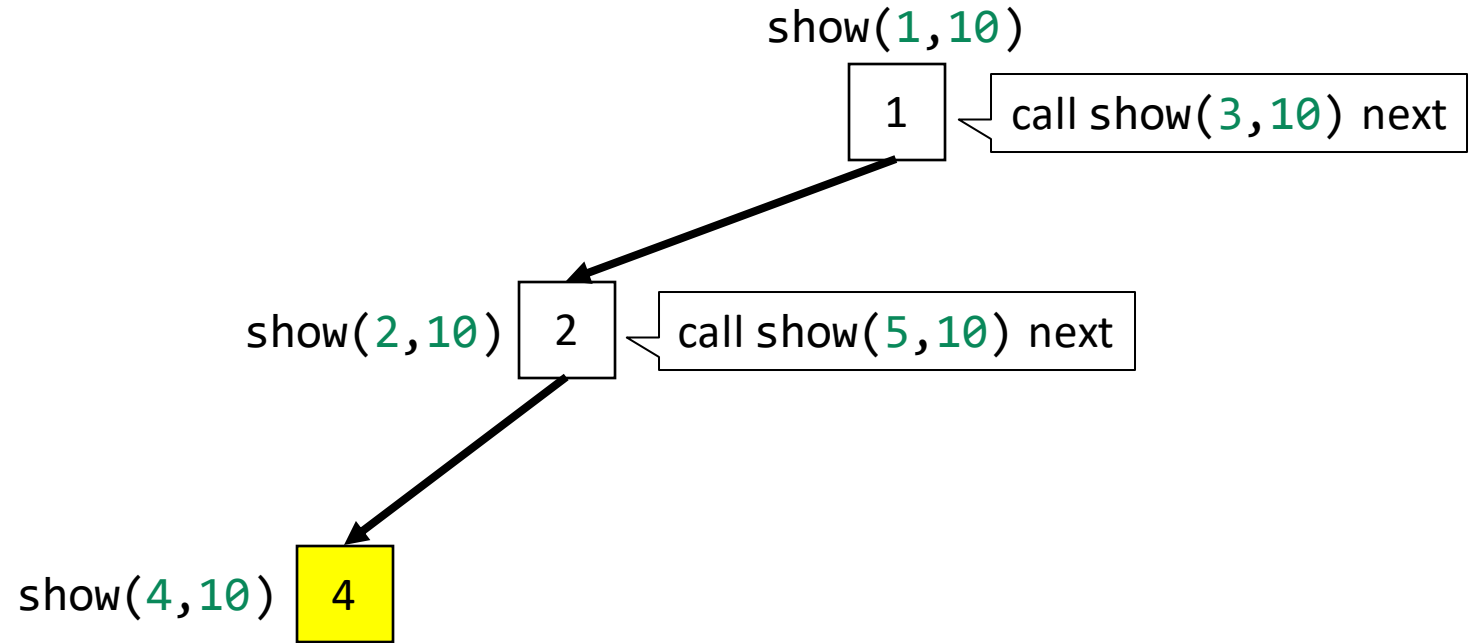
Then, we see that we should call `show(4, 10)` followed by `show(5, 10)`.

We call `show(4, 10)` first.

```
def show(4, 10):  
    if 4 > 10:  
        return  
    print(4)  
    show(2*4, 10)  
    show(2*4+1, 10)
```

Output:

1  
2  
4



We are now in `show(4, 10)`.

Since `4 <= 10`, we skip the if statement, and print 4.

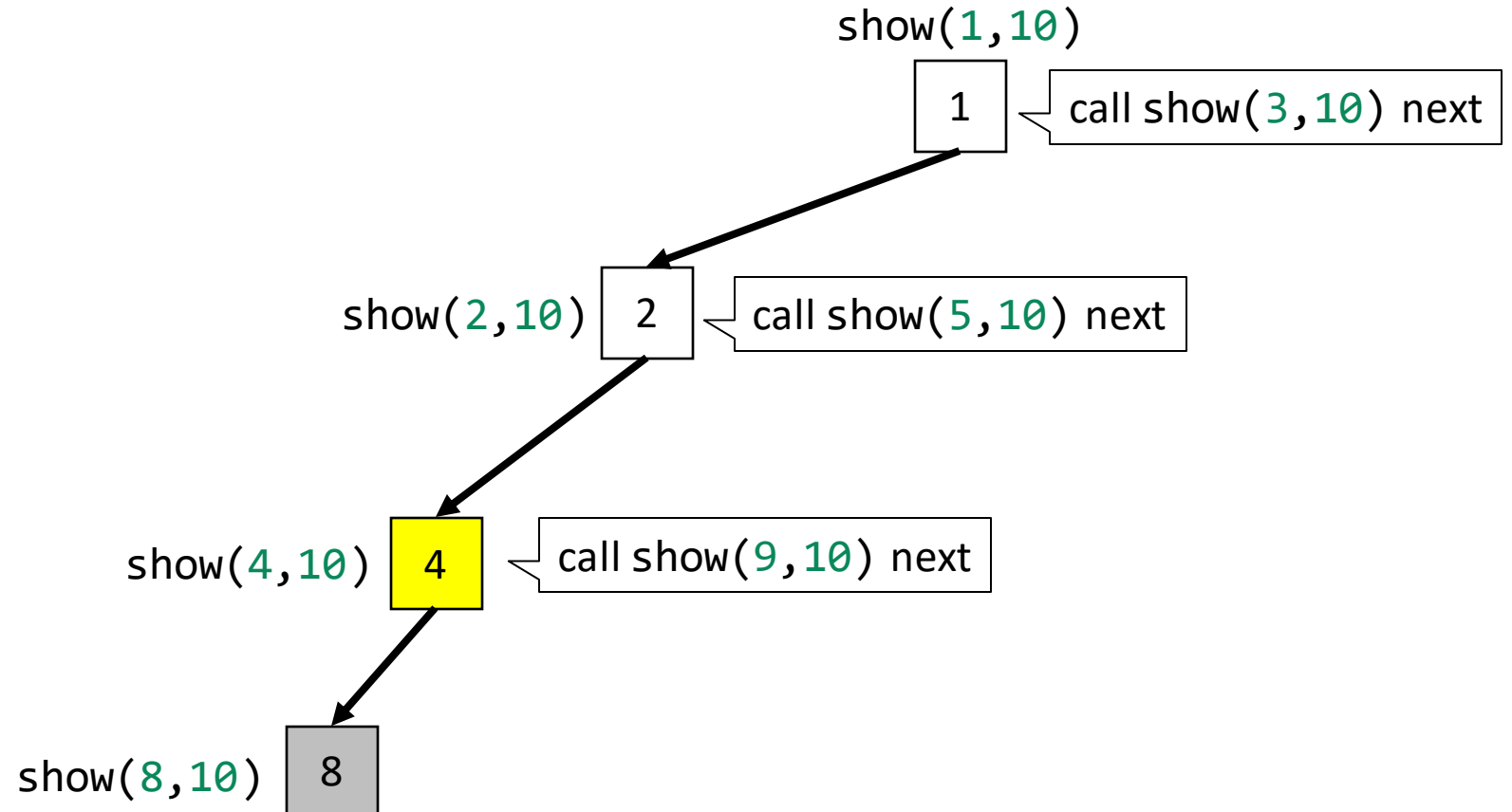


```
def show(4, 10):  
    if 4 > 10:  
        return  
    print(4)  
    show(2*4, 10)  
    show(2*4+1, 10)
```



Output:

1  
2  
4



We are now in `show(4, 10)`.

Since `4 <= 10`, we skip the if statement, and print 4.

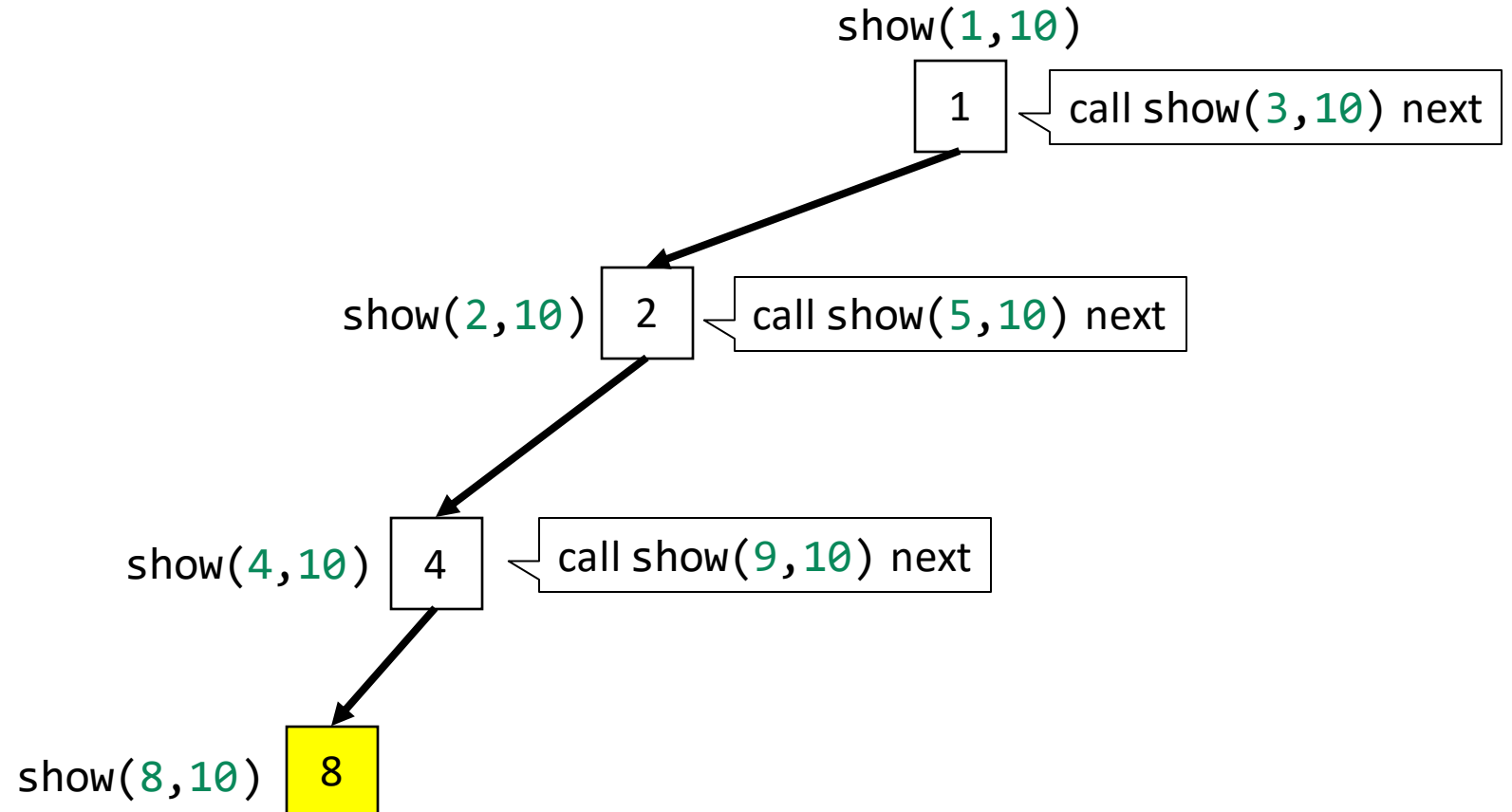
Then, we see that we should call `show(8, 10)` followed by `show(9, 10)`.

We call `show(8, 10)` first.

```
def show(8, 10):  
    if 8 > 10:  
        return  
    print(8)  
    show(2*8, 10)  
    show(2*8+1, 10)
```

Output:

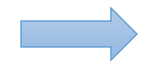
1  
2  
4  
8



We are now in `show(8, 10)`.

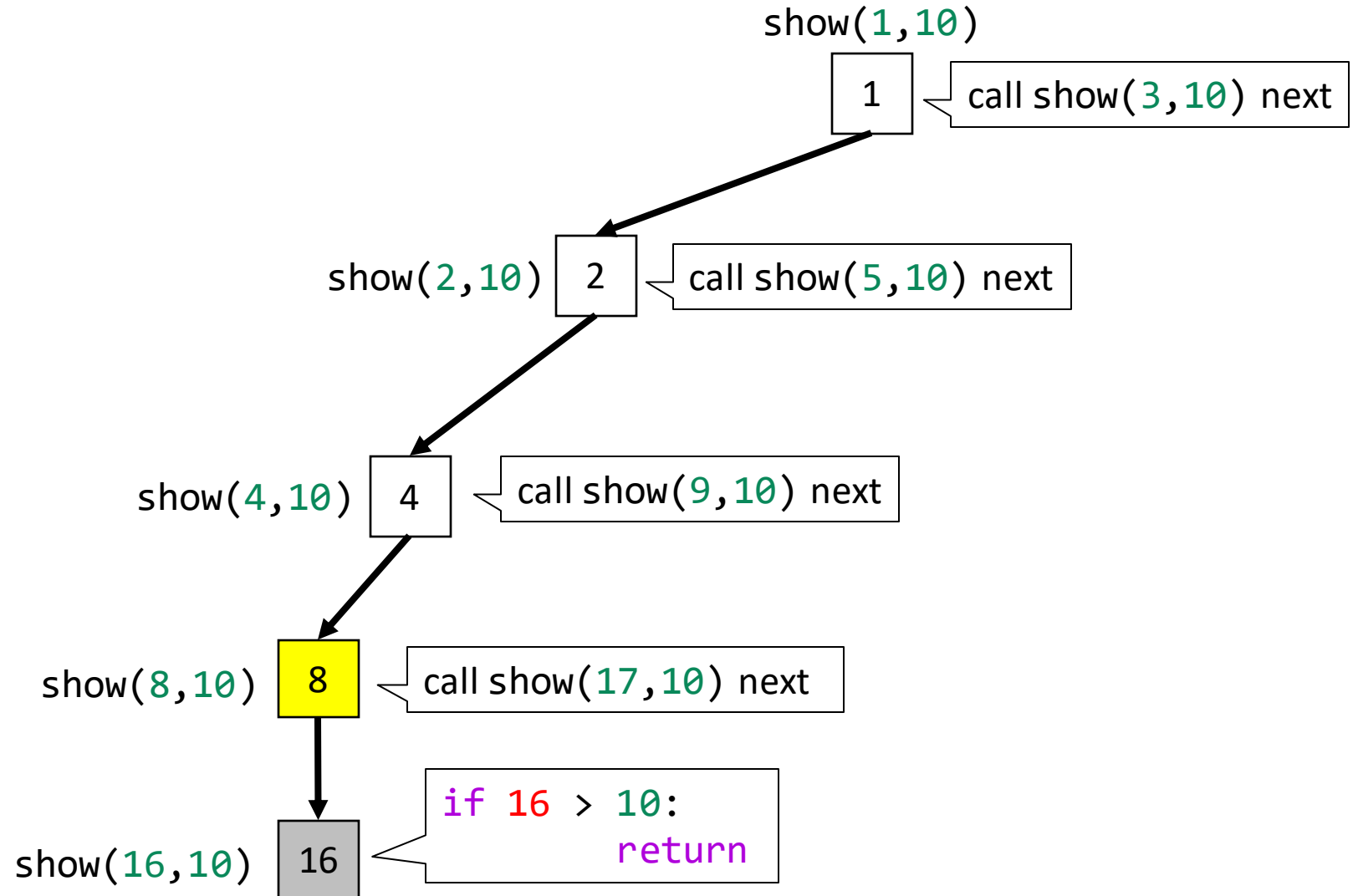
Since `8 <= 10`, we skip the if statement, and print 8.

```
def show(8, 10):  
    if 8 > 10:  
        return  
    print(8)  
    show(2*8, 10)  
    show(2*8+1, 10)
```



Output:

1  
2  
4  
8



We are now in `show(8,10)`.

Since `8 <= 10`, we skip the if statement, and print 8.

Then, we see that we should call `show(16,10)` followed by `show(17,10)`.

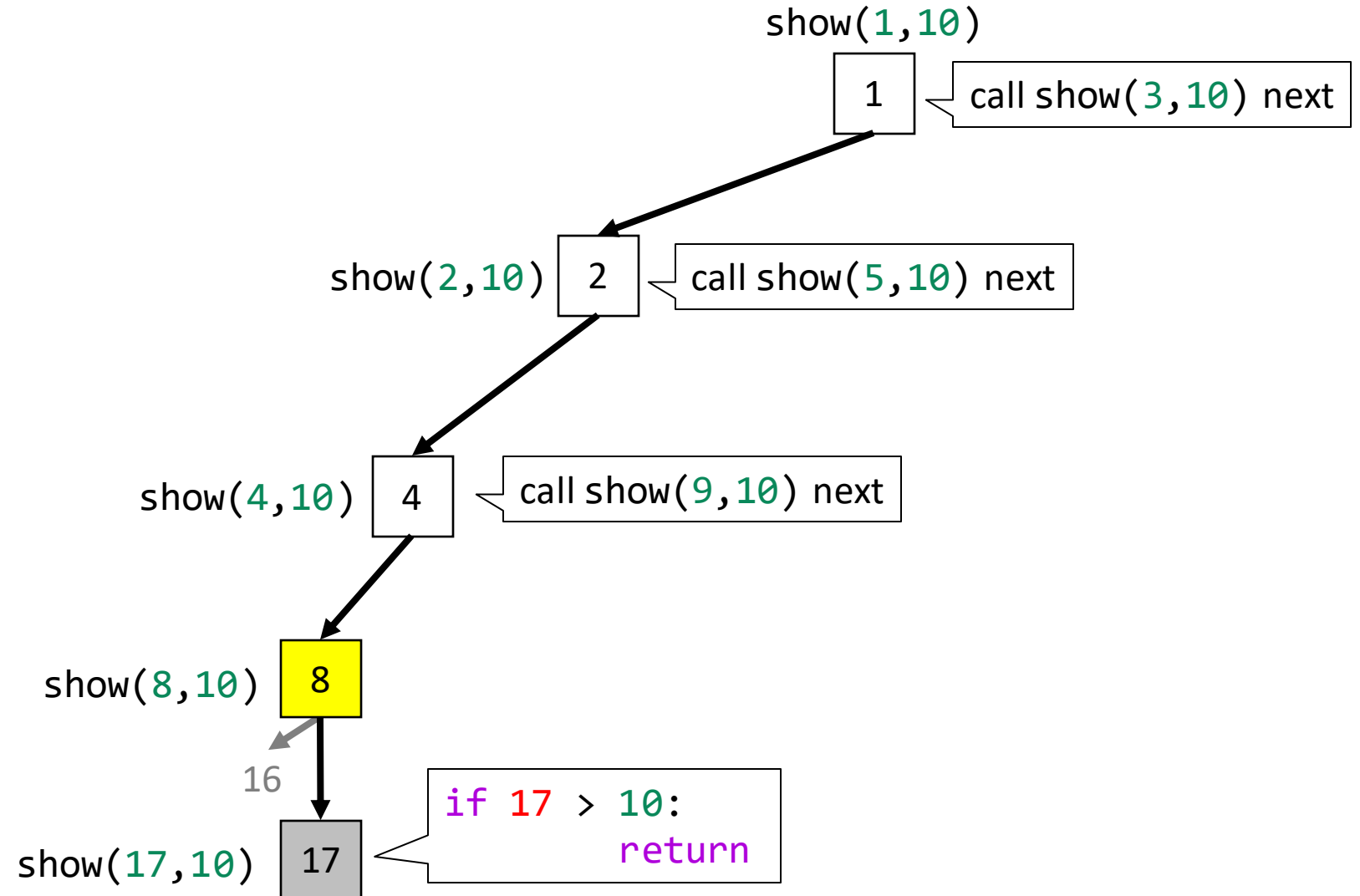
We call `show(16,10)` first, but since `16 > 10`, we return immediately.

```
def show(8, 10):  
    if 8 > 10:  
        return  
    print(8)  
    show(2*8, 10)  
    show(2*8+1, 10)
```

→

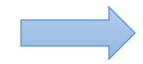
Output:

1  
2  
4  
8



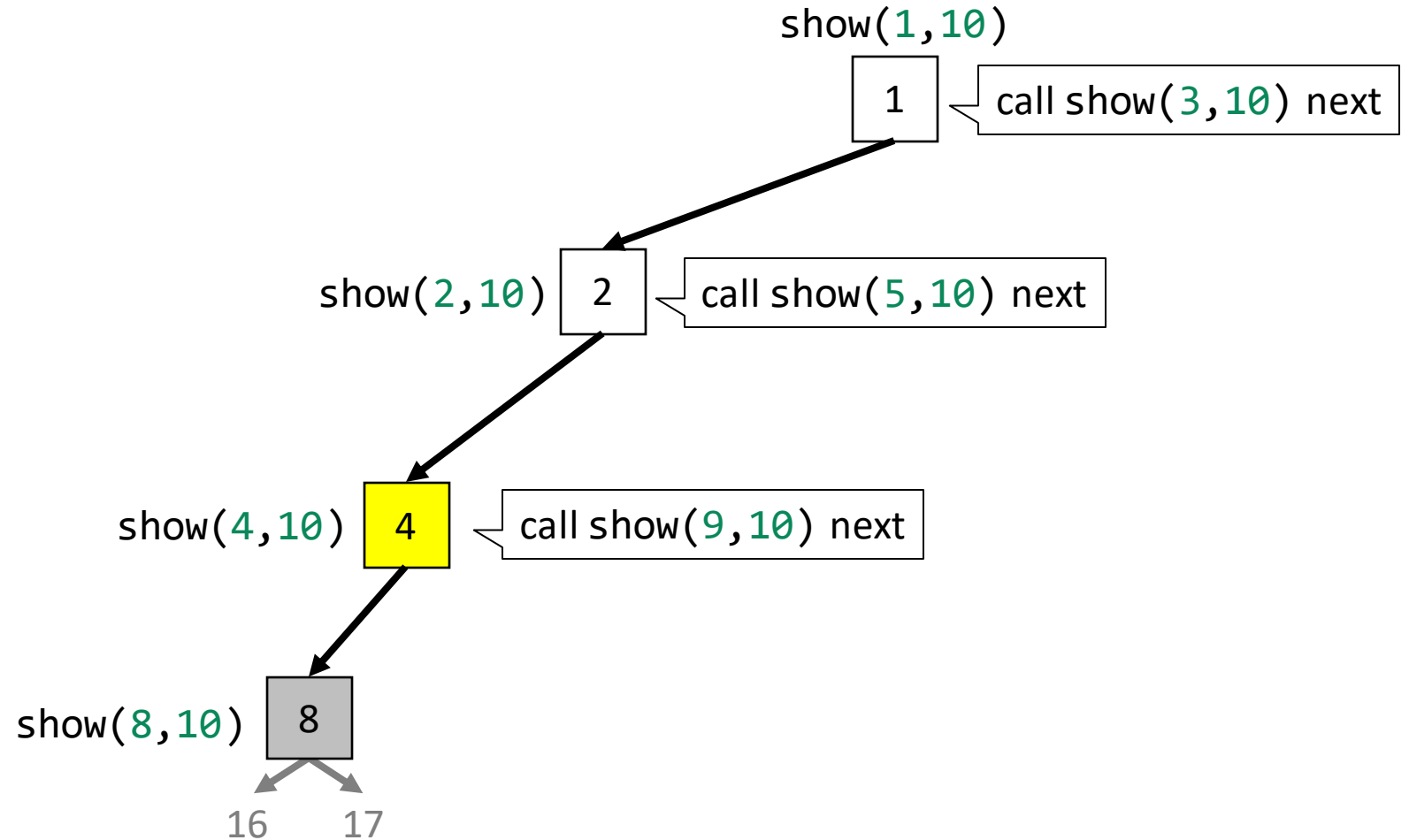
We return to `show(8, 10)` and call `show(17, 10)`.  
Since `17 > 10`, we return immediately.

```
def show(4, 10):  
    if 4 > 10:  
        return  
    print(4)  
    show(2*4, 10)  
    show(2*4+1, 10)
```



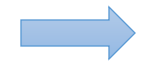
Output:

1  
2  
4  
8



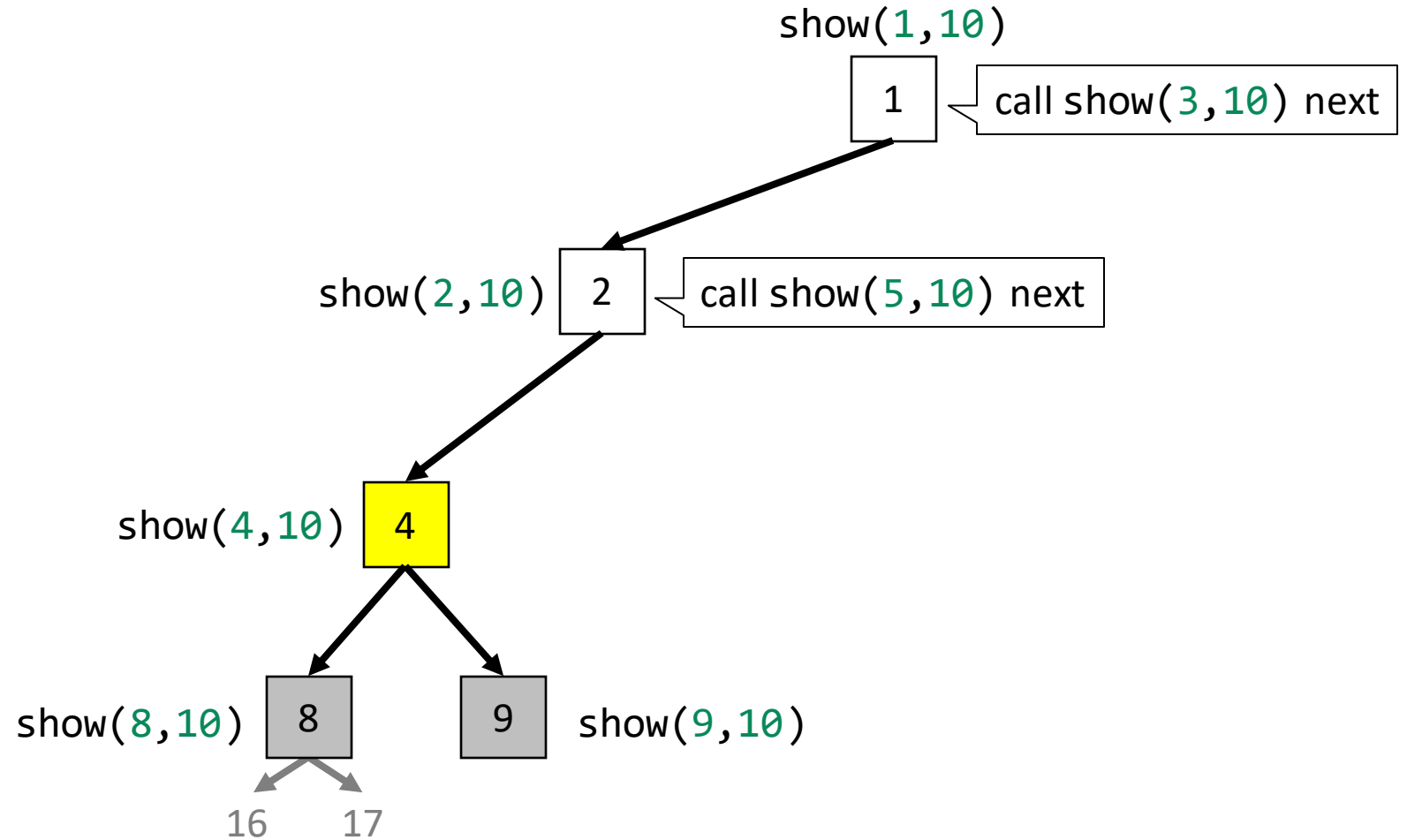
After returning from `show(17, 10)`, we also finish `show(8, 10)`.  
This means we are back in `show(4, 10)`.

```
def show(4, 10):  
    if 4 > 10:  
        return  
    print(4)  
    show(2*4, 10)  
    show(2*4+1, 10)
```



Output:

1  
2  
4  
8

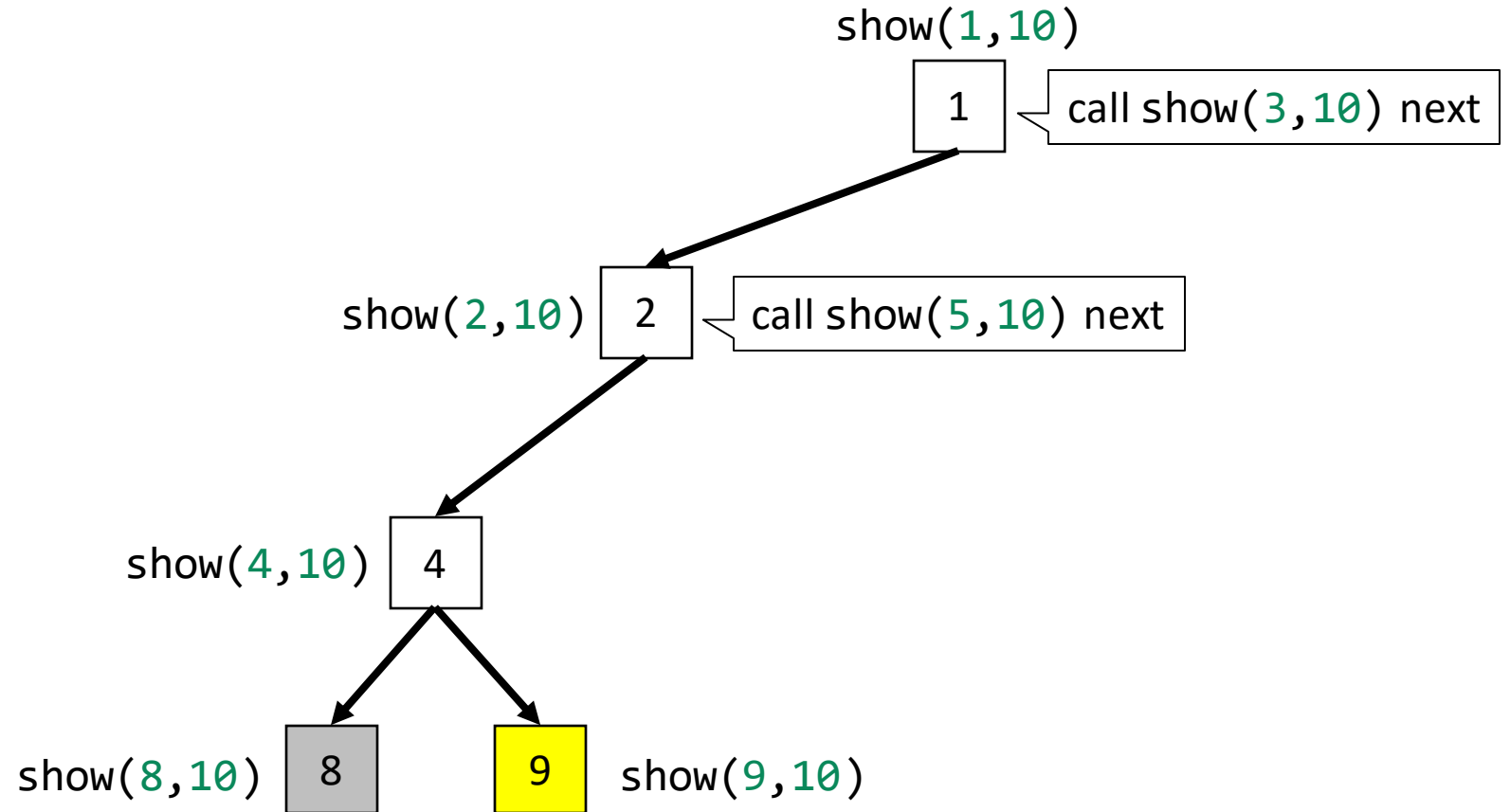


After returning from `show(17, 10)`, we also finish `show(8, 10)`. This means we are back in `show(4, 10)`. We call `show(9, 10)`.

```
def show(9, 10):  
    if 9 > 10:  
        return  
    print(9)  
    show(2*9, 10)  
    show(2*9+1, 10)
```

Output:

1  
2  
4  
8  
9



We are now in `show(9, 10)`.

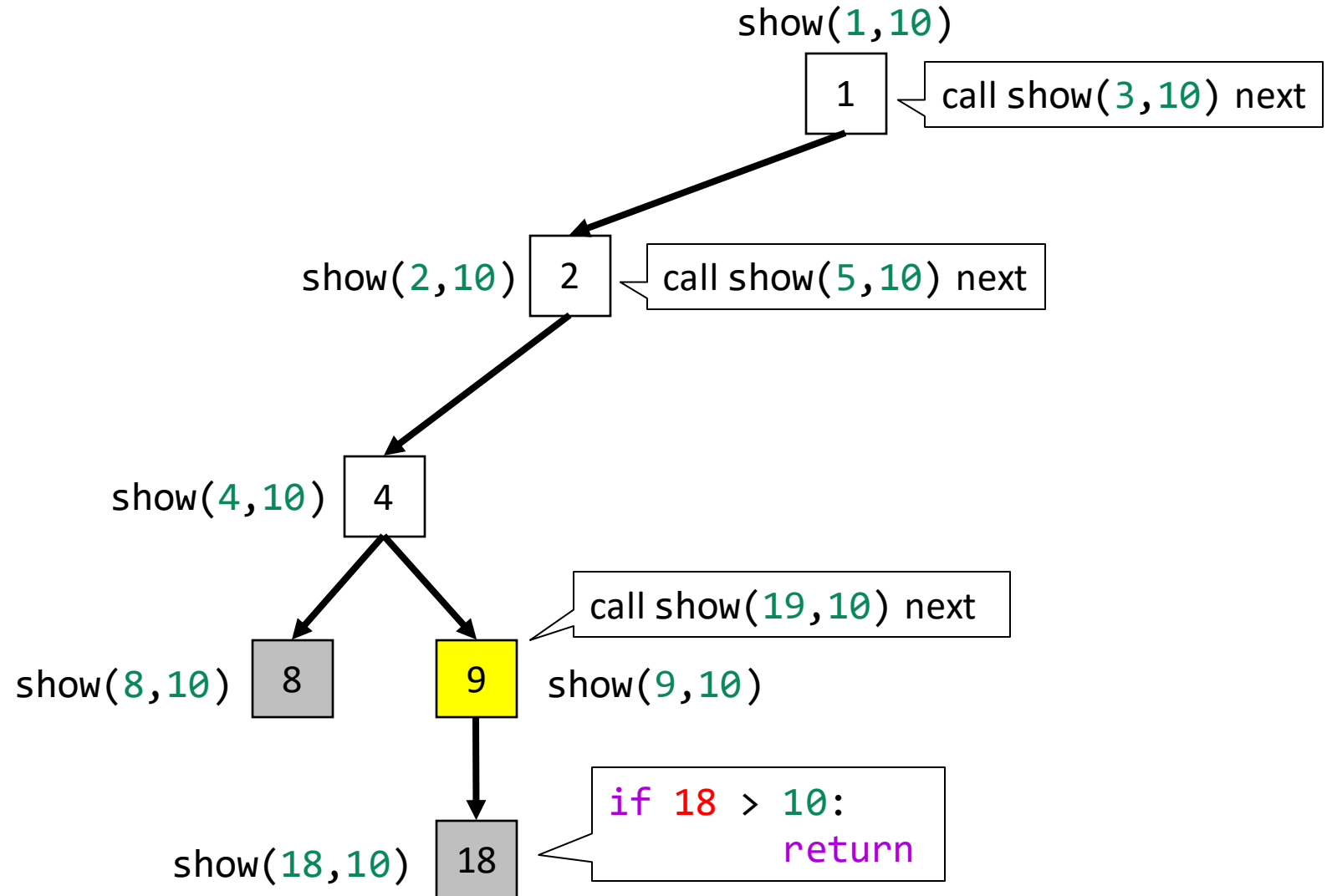
Since  $9 \leq 10$ , we skip the if statement, and print 9.

```
def show(9, 10):  
    if 9 > 10:  
        return  
    print(9)  
    show(2*9, 10)  
    show(2*9+1, 10)
```



Output:

1  
2  
4  
8  
9



We are now in `show(9,10)`.

Since  $9 \leq 10$ , we skip the if statement, and print 9.

Then, we see that we should call `show(18,10)` followed by `show(19,10)`.

We call `show(18,10)` first, but since  $18 > 10$ , we return immediately.

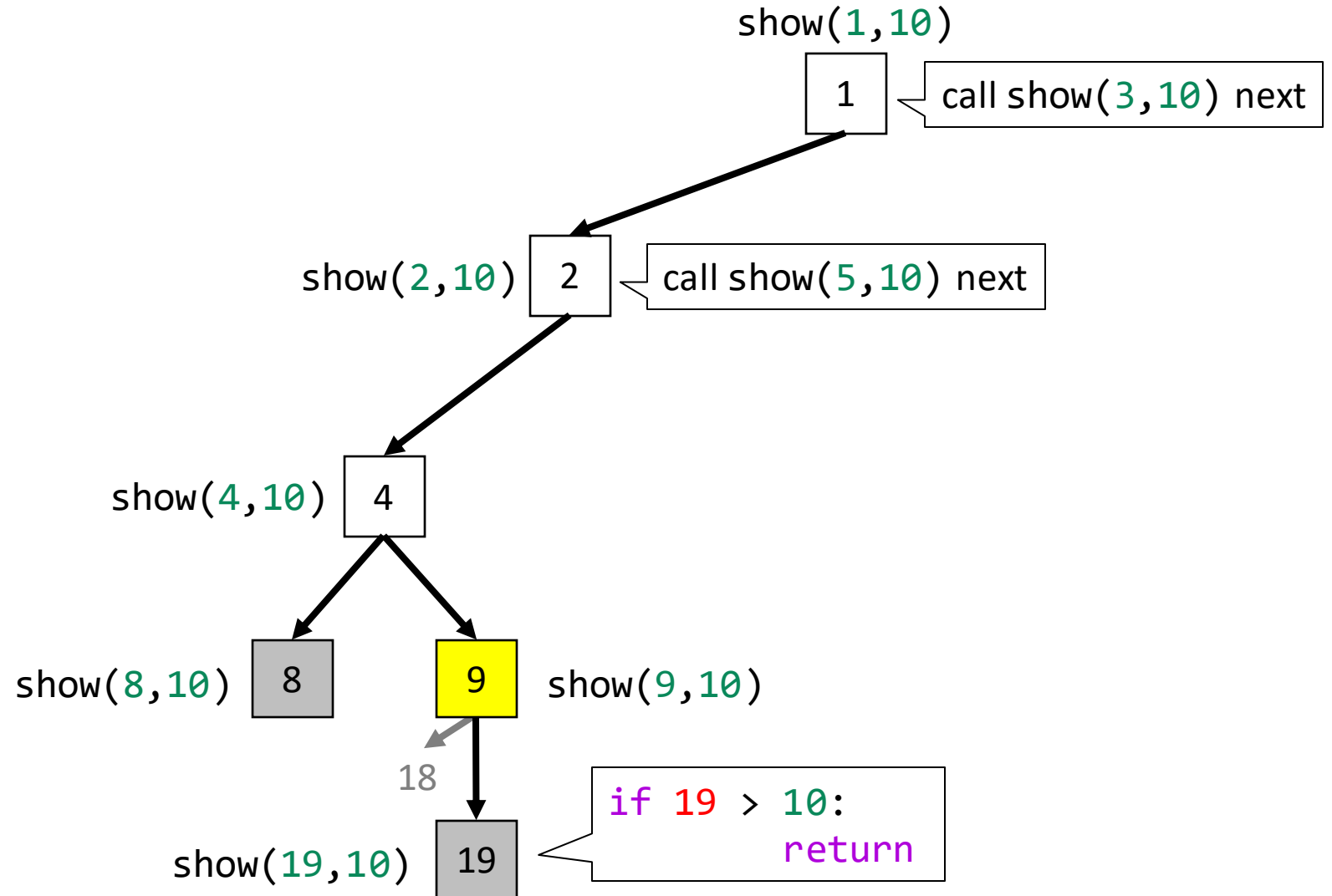


```
def show(9, 10):  
    if 9 > 10:  
        return  
    print(9)  
    show(2*9, 10)  
    show(2*9+1, 10)
```



Output:

1  
2  
4  
8  
9



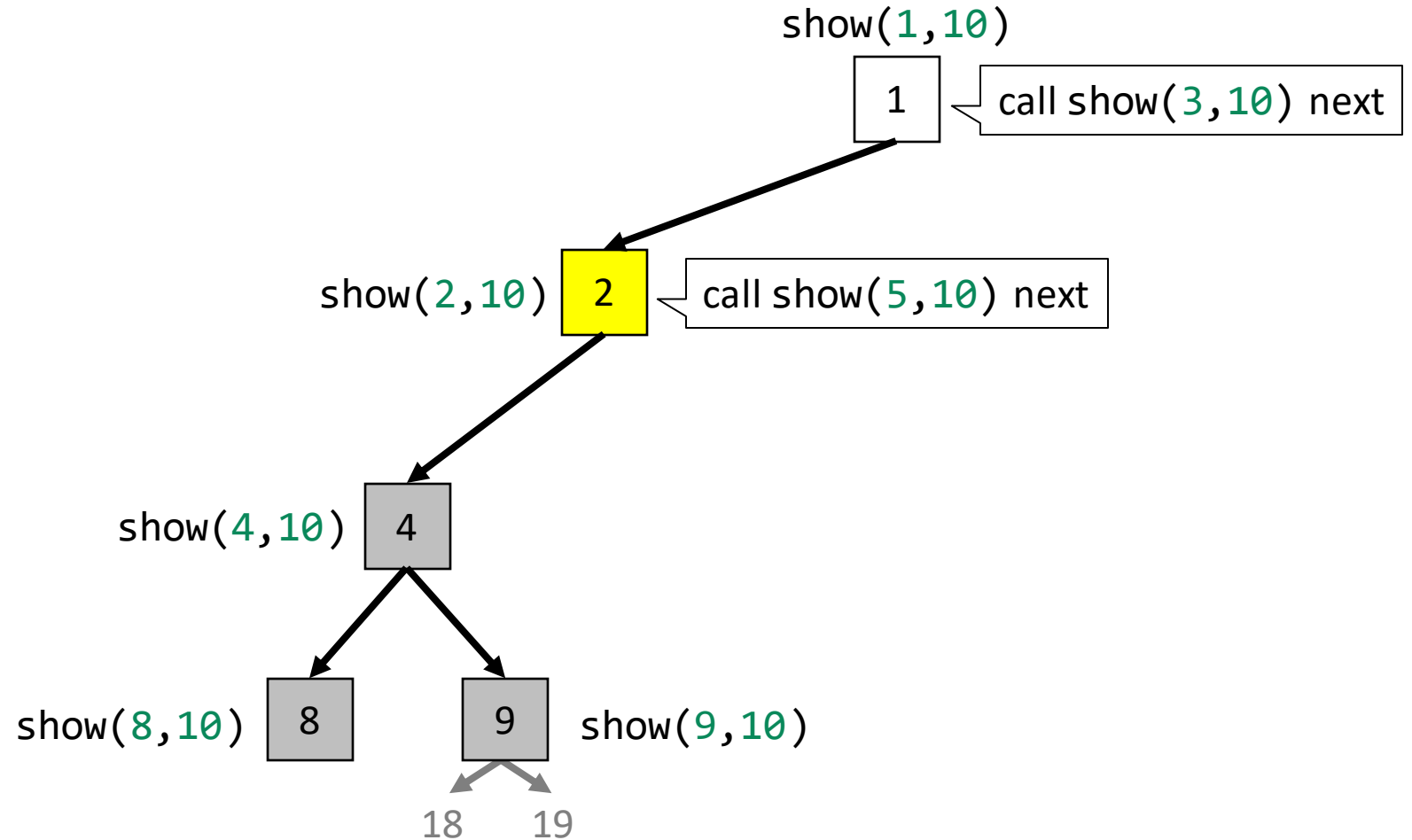
We return to `show(9, 10)` and call `show(19, 10)`.  
Since `19 > 10`, we return immediately.

```
def show(2, 10):  
    if 2 > 10:  
        return  
    print(2)  
    show(2*2, 10)  
    show(2*2+1, 10)
```



Output:

1  
2  
4  
8  
9



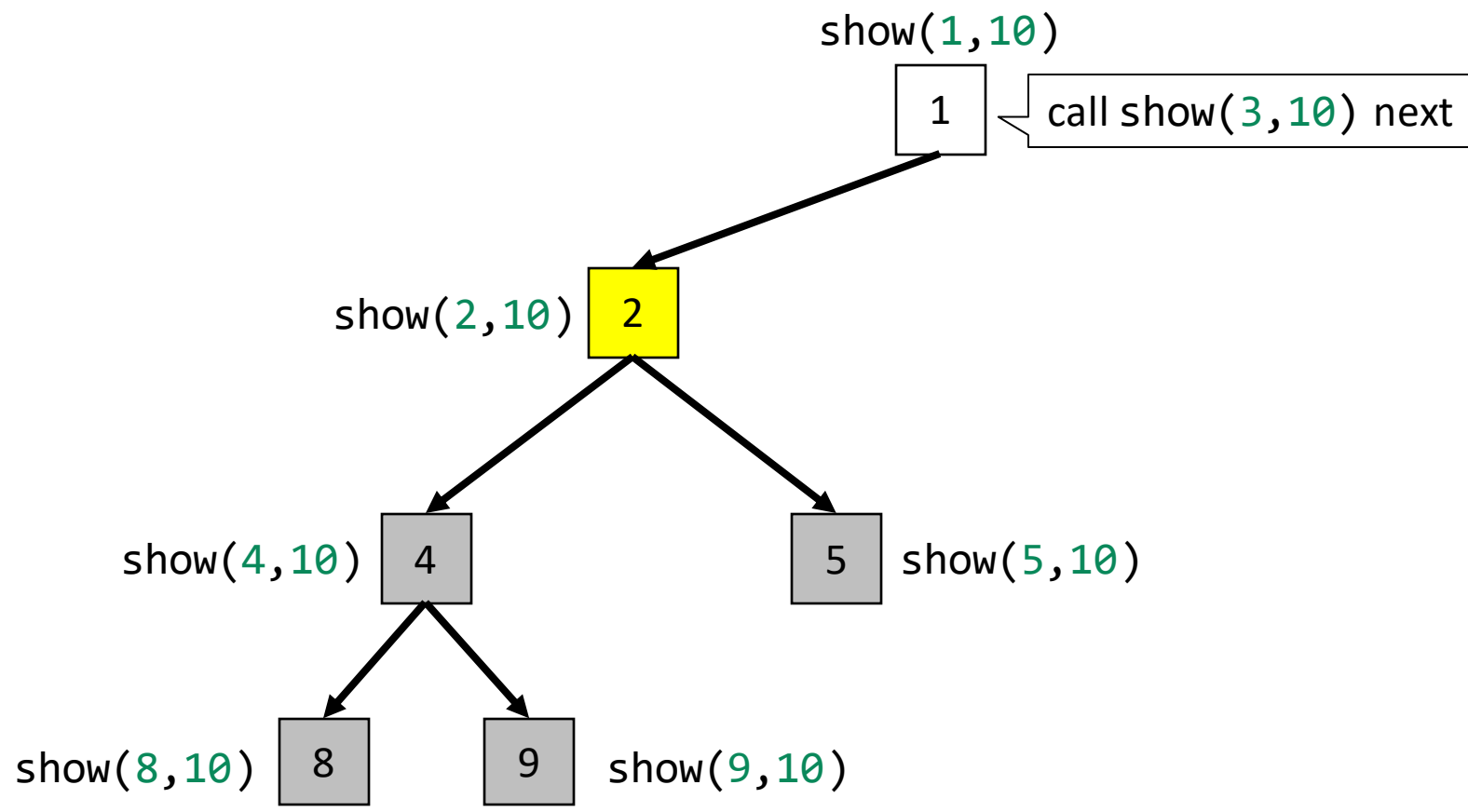
We return to `show(9, 10)`.

Since we are done with `show(9, 10)`, we return to `show(4, 10)`.

But we are also done with `show(4, 10)`, so we return to `show(2, 10)`.

```
def show(2, 10):  
    if 2 > 10:  
        return  
    print(2)  
    show(2*2, 10)  
    show(2*2+1, 10)
```

Output:  
1  
2  
4  
8  
9

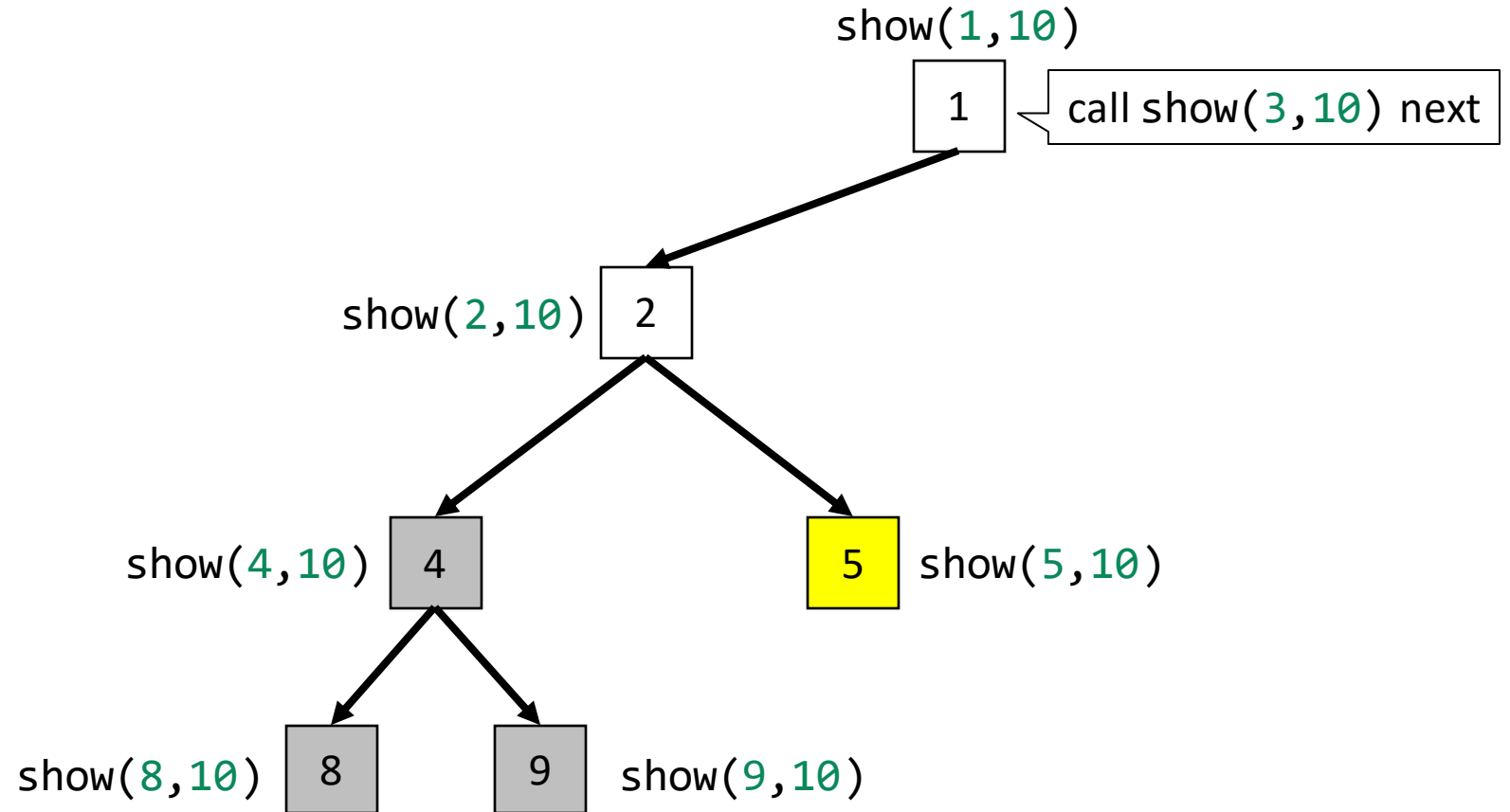


We return to `show(9, 10)`.  
Since we are done with `show(9, 10)`, we return to `show(4, 10)`.  
But we are also done with `show(4, 10)`, so we return to `show(2, 10)`.  
We call `show(5, 10)`.

```
def show(5, 10):  
    if 5 > 10:  
        return  
    print(5)  
    show(2*5, 10)  
    show(2*5+1, 10)
```

Output:

1  
2  
4  
8  
9  
5



We are now in `show(5, 10)`.

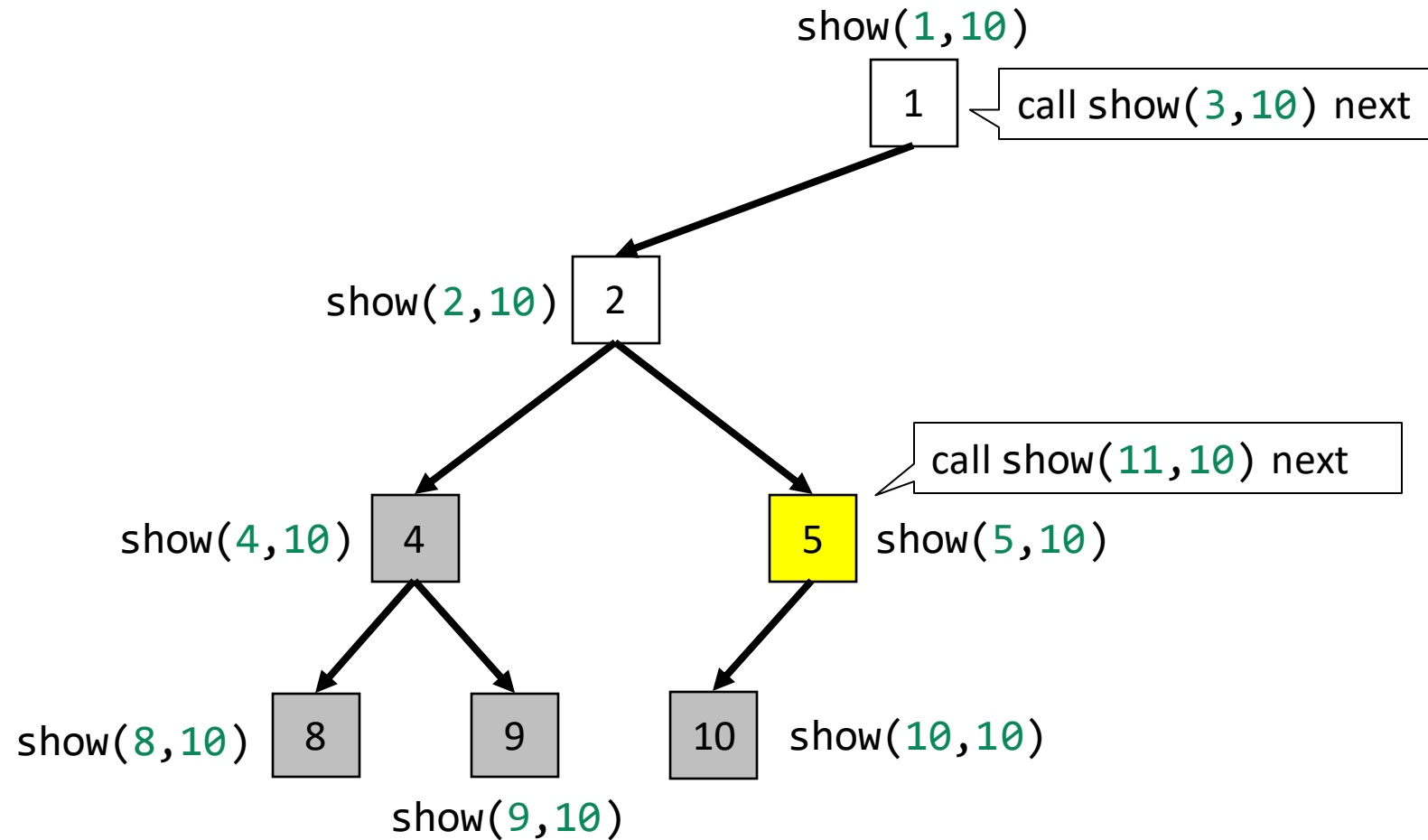
Since `5 <= 10`, we skip the if statement, and print 5.

```
def show(5, 10):
    if 5 > 10:
        return
    print(5)
    show(2*5, 10)
    show(2*5+1, 10)
```



Output:

1  
2  
4  
8  
9  
5

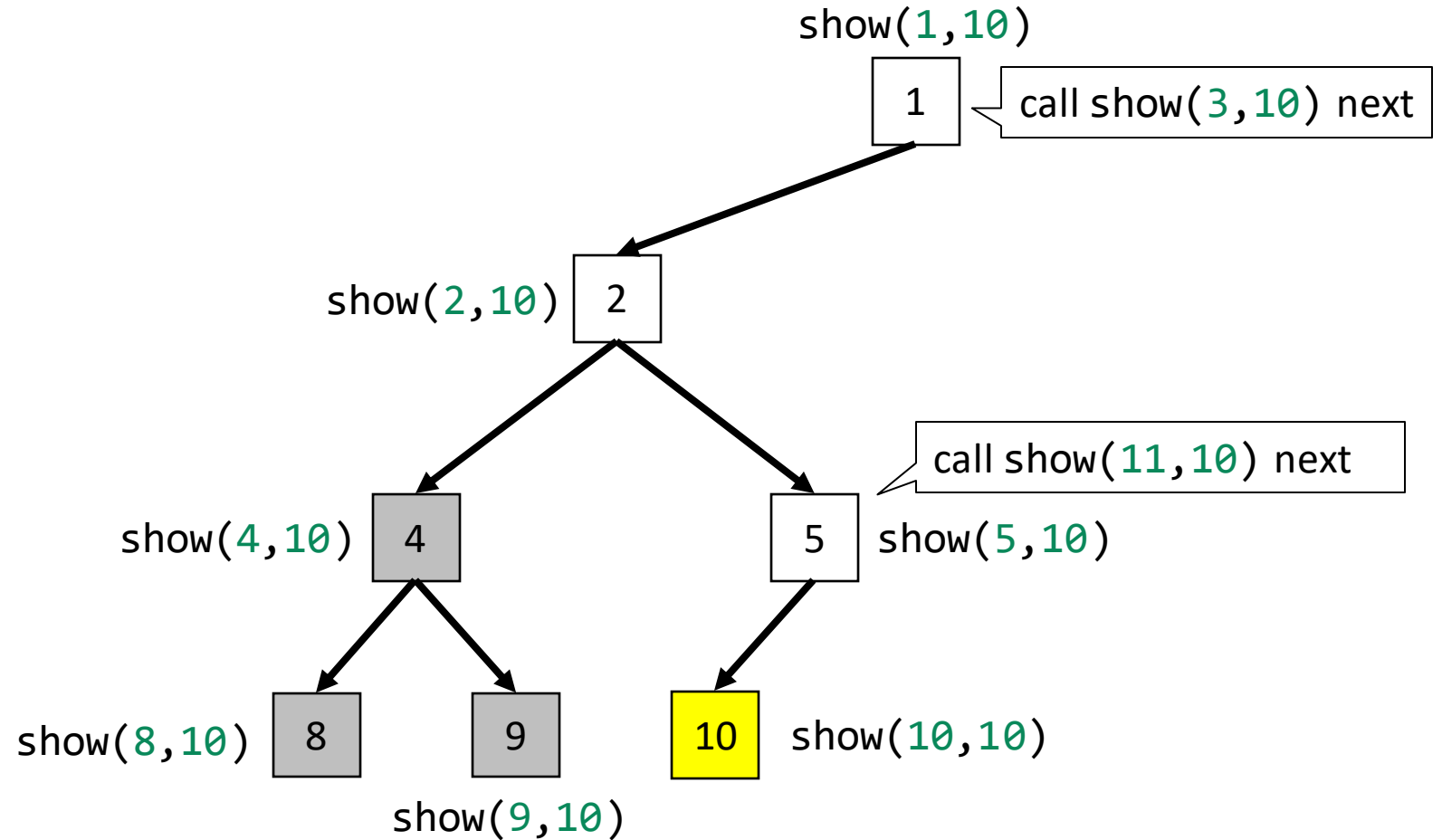


We are now in `show(5, 10)`.  
 Since `5 <= 10`, we skip the if statement, and print 5.  
 Then, we see that we should call `show(10, 10)` followed by `show(11, 10)`.  
 We call `show(10, 10)` first.

```
def show(10, 10):  
    if 10 > 10:  
        return  
    print(10)  
    show(2*10, 10)  
    show(2*10+1, 10)
```

Output:

1  
2  
4  
8  
9  
5  
10



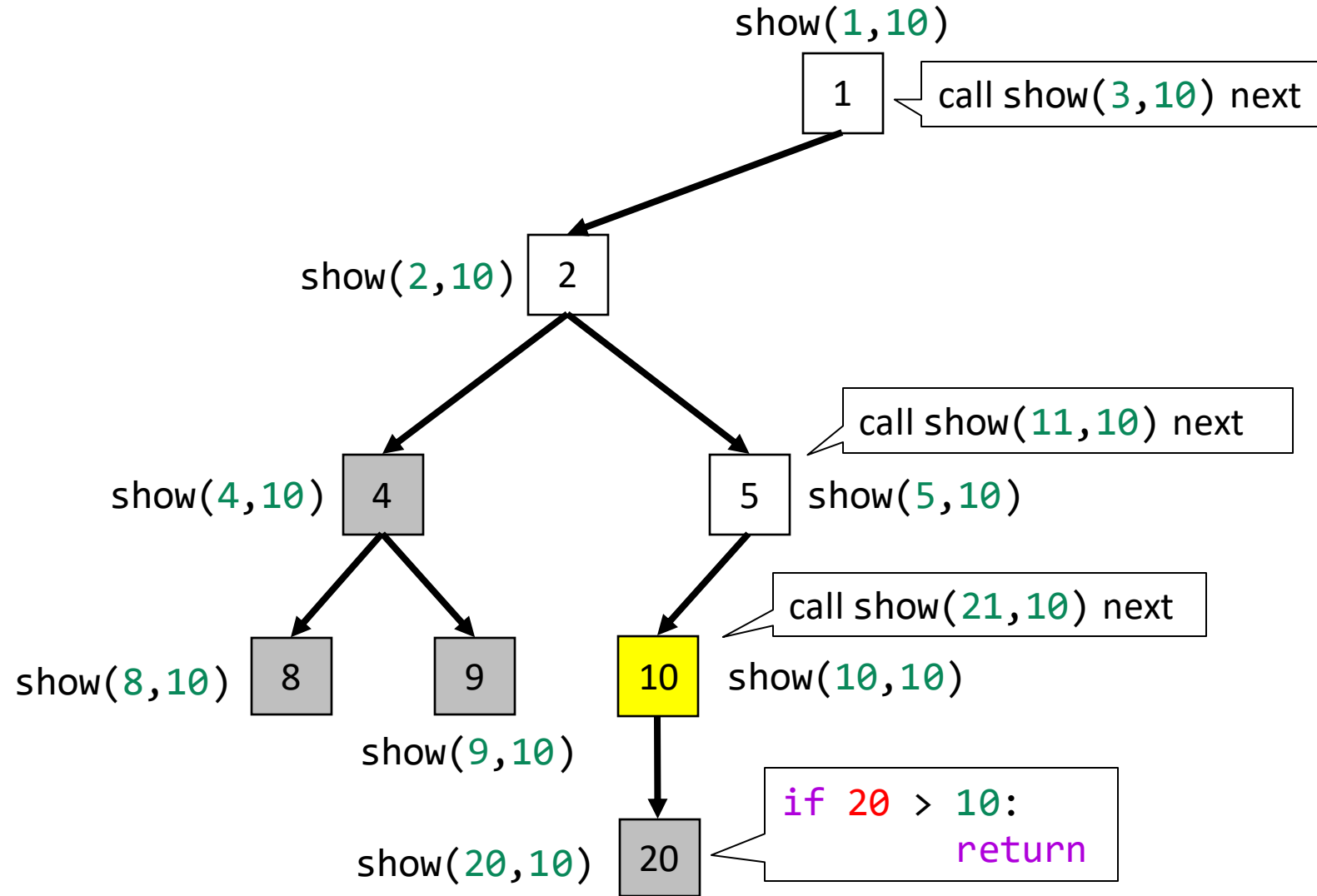
We are now in `show(10, 10)`.

Since `10 <= 10`, we skip the if statement, and print 10.

```
def show(10, 10):
    if 10 > 10:
        return
    print(10)
    show(2*10, 10)
    show(2*10+1, 10)
```

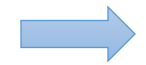
Output:

1  
2  
4  
8  
9  
5  
10



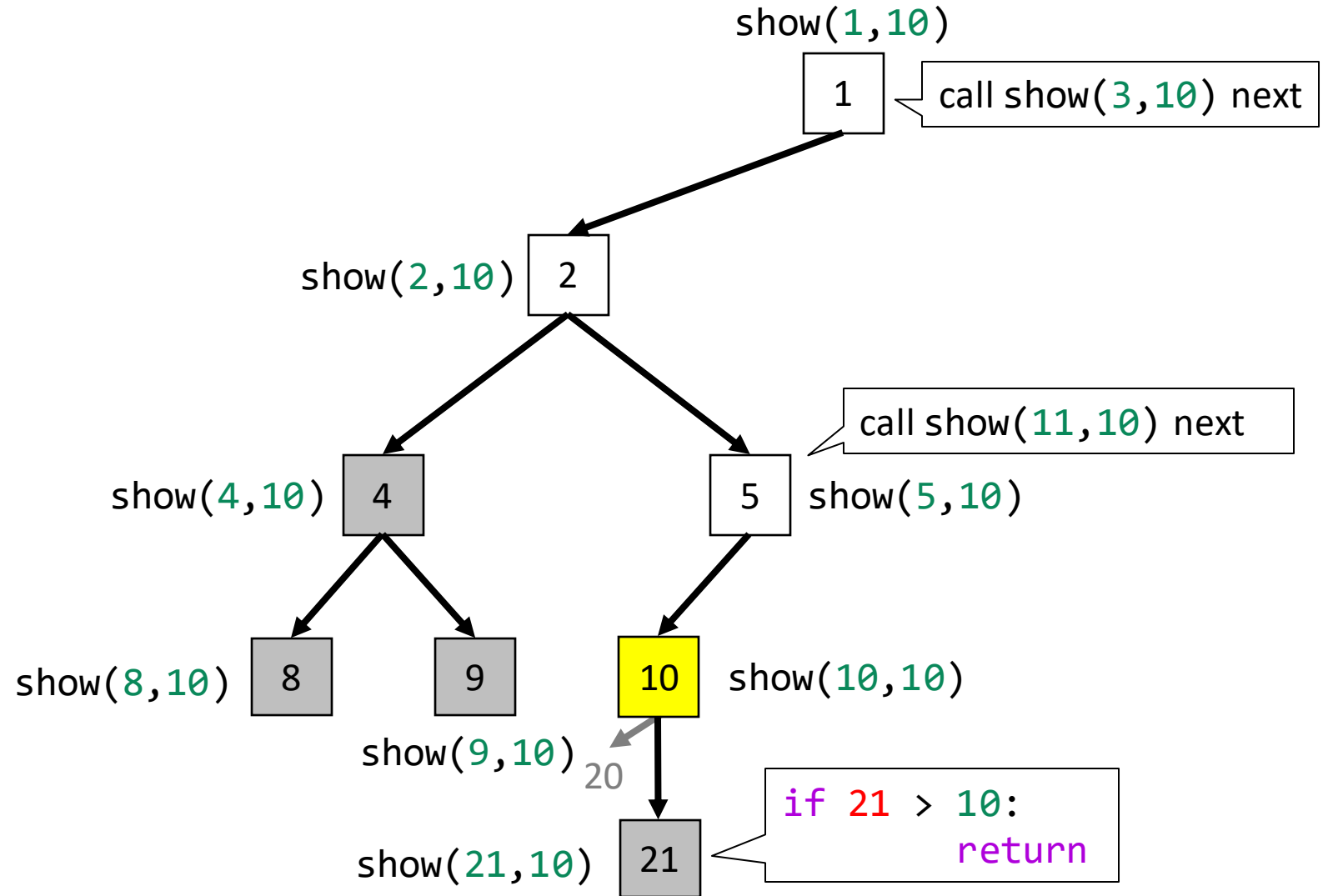
We are now in `show(10, 10)`.  
 Since `10 <= 10`, we skip the if statement, and print 10.  
 Then, we see that we should call `show(20, 10)` followed by `show(21, 10)`.  
 We call `show(20, 10)` first, but since `20 > 10`, we return immediately.

```
def show(10, 10):  
    if 10 > 10:  
        return  
    print(10)  
    show(2*10, 10)  
    show(2*10+1, 10)
```



Output:

1  
2  
4  
8  
9  
5  
10



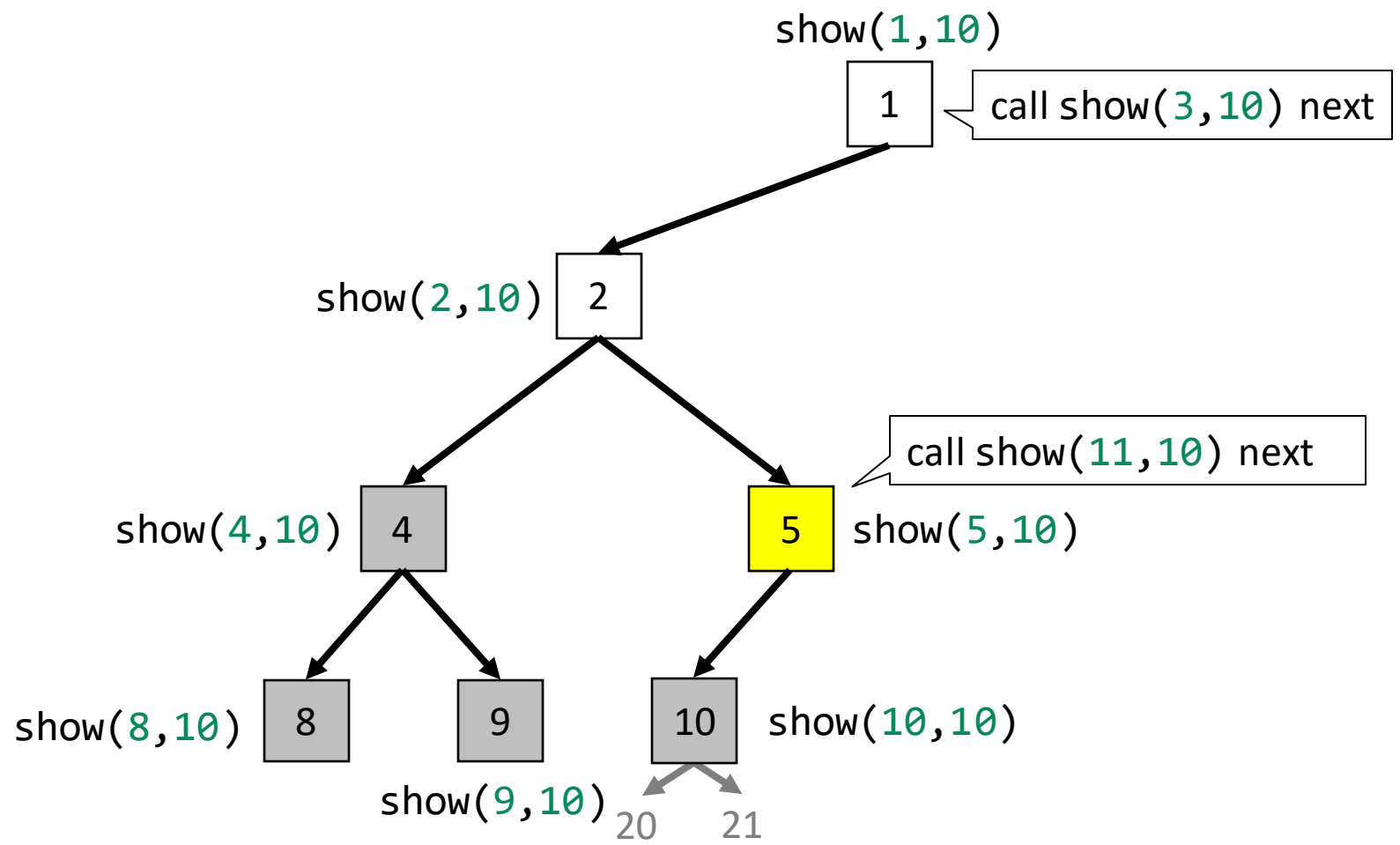
We return to show(10, 10) and call show(21, 10).  
Since 21 > 10, we return immediately.



```
def show(5, 10):  
    if 5 > 10:  
        return  
    print(5)  
    show(2*5, 10)  
    show(2*5+1, 10)
```

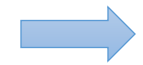
Output:

1  
2  
4  
8  
9  
5  
10



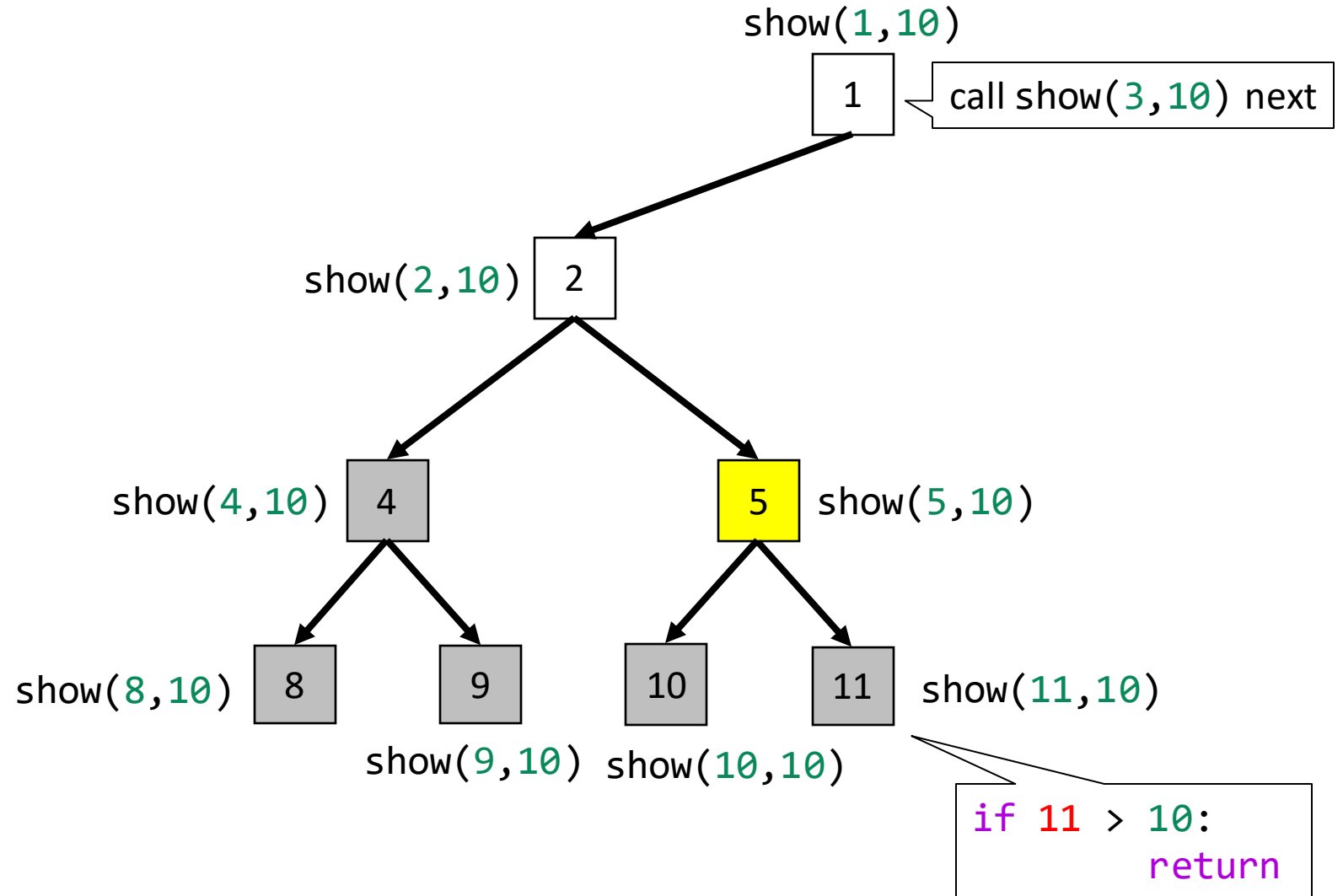
We return to show(10,10).  
Since we are done with show(10,10), we return to show(5,10).

```
def show(5, 10):  
    if 5 > 10:  
        return  
    print(5)  
    show(2*5, 10)  
    show(2*5+1, 10)
```



Output:

1  
2  
4  
8  
9  
5  
10



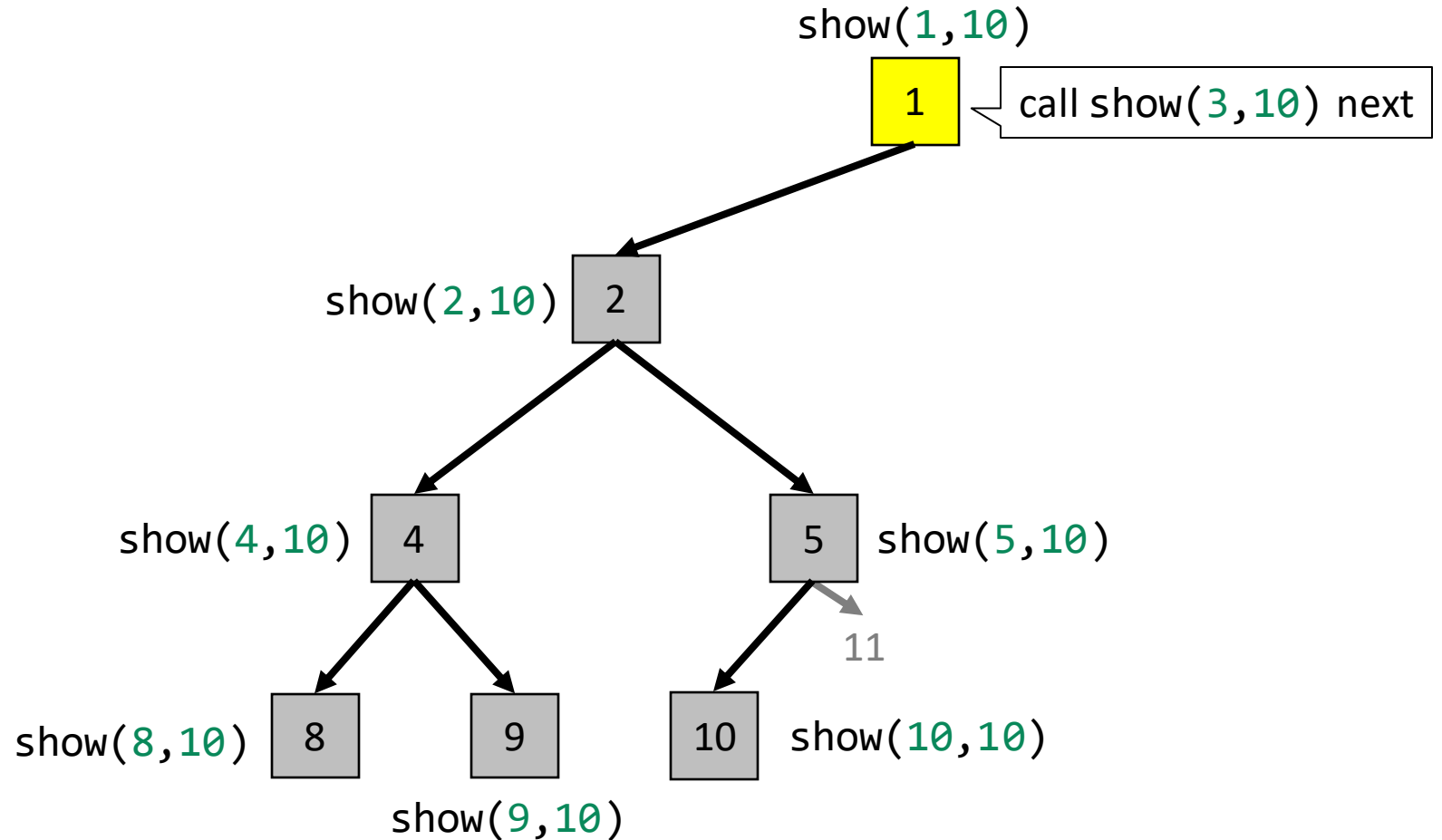
We are back to `show(5,10)`

We call `show(11,10)` next, but since `11 > 10`, we return immediately.

```
def show(1, 10):  
    if 1 > 10:  
        return  
    print(1)  
    show(2*1, 10)  
    show(2*1+1, 10)
```

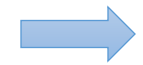
Output:

1  
2  
4  
8  
9  
5  
10



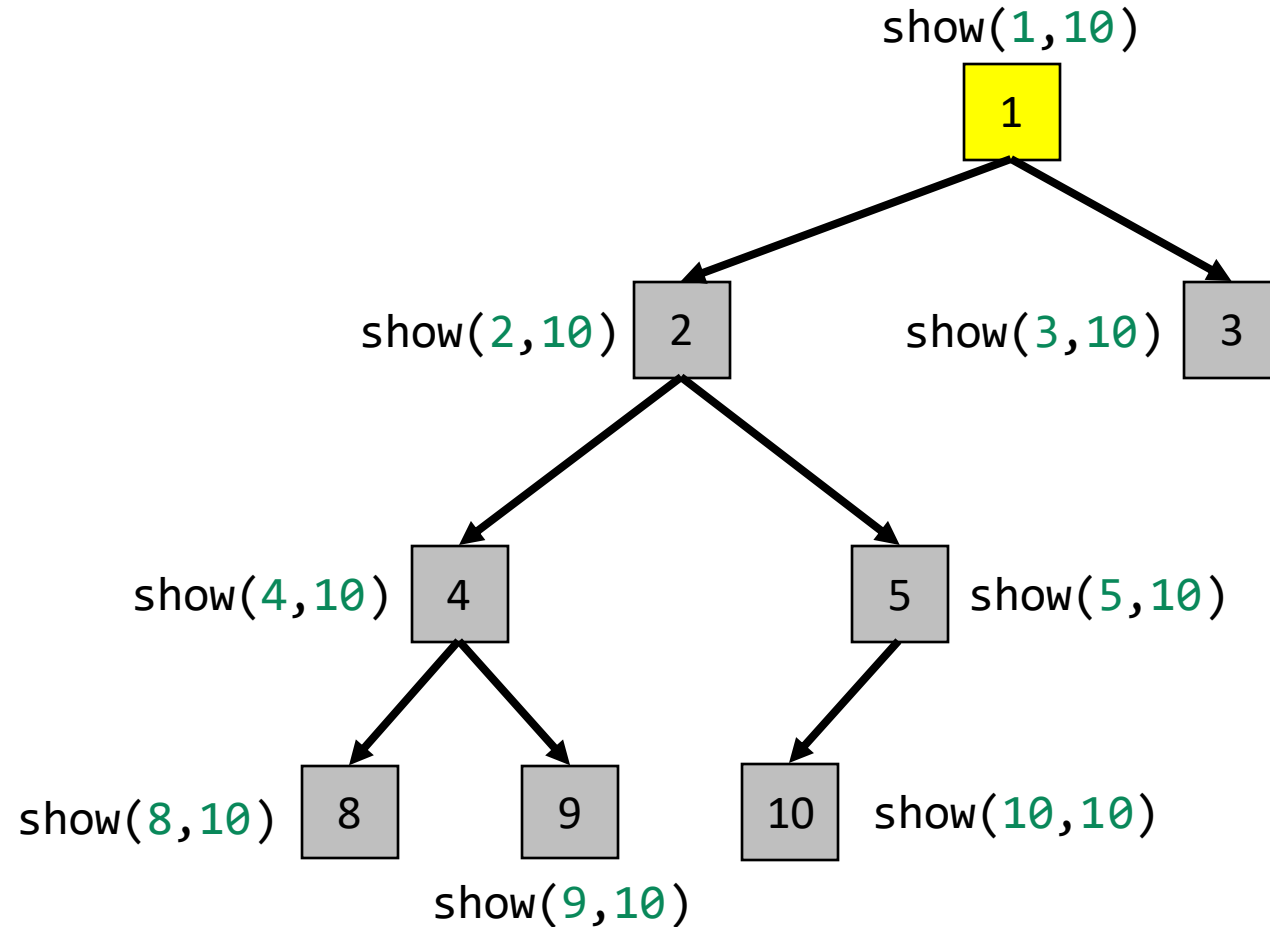
We return to `show(5, 10)`.  
Since we are done with `show(5, 10)`, we return to `show(2, 10)`.  
But we are also done with `show(2, 10)`, so we return to `show(1, 10)`.

```
def show(1, 10):  
    if 1 > 10:  
        return  
    print(1)  
    show(2*1, 10)  
    show(2*1+1, 10)
```



Output:

1  
2  
4  
8  
9  
5  
10



We return to `show(5,10)`.

Since we are done with `show(5,10)`, we return to `show(2,10)`.

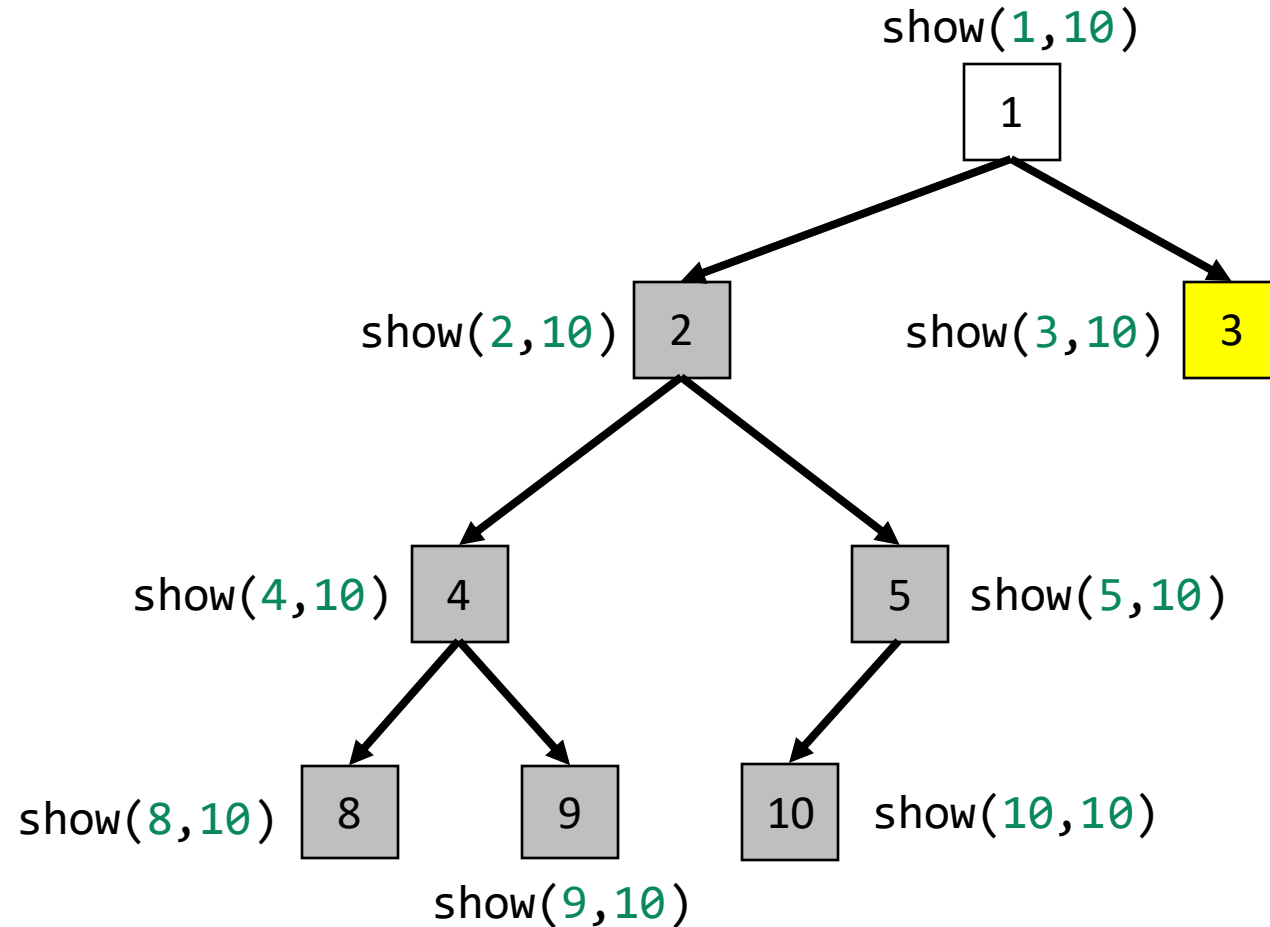
But we are also done with `show(2,10)`, so we return to `show(1,10)`.

We call `show(3,10)` next.

```
def show(3, 10):  
    if 3 > 10:  
        return  
    print(3)  
    show(2*3, 10)  
    show(2*3+1, 10)
```

Output:

1  
2  
4  
8  
9  
5  
10  
3



We are now in `show(3, 10)`.

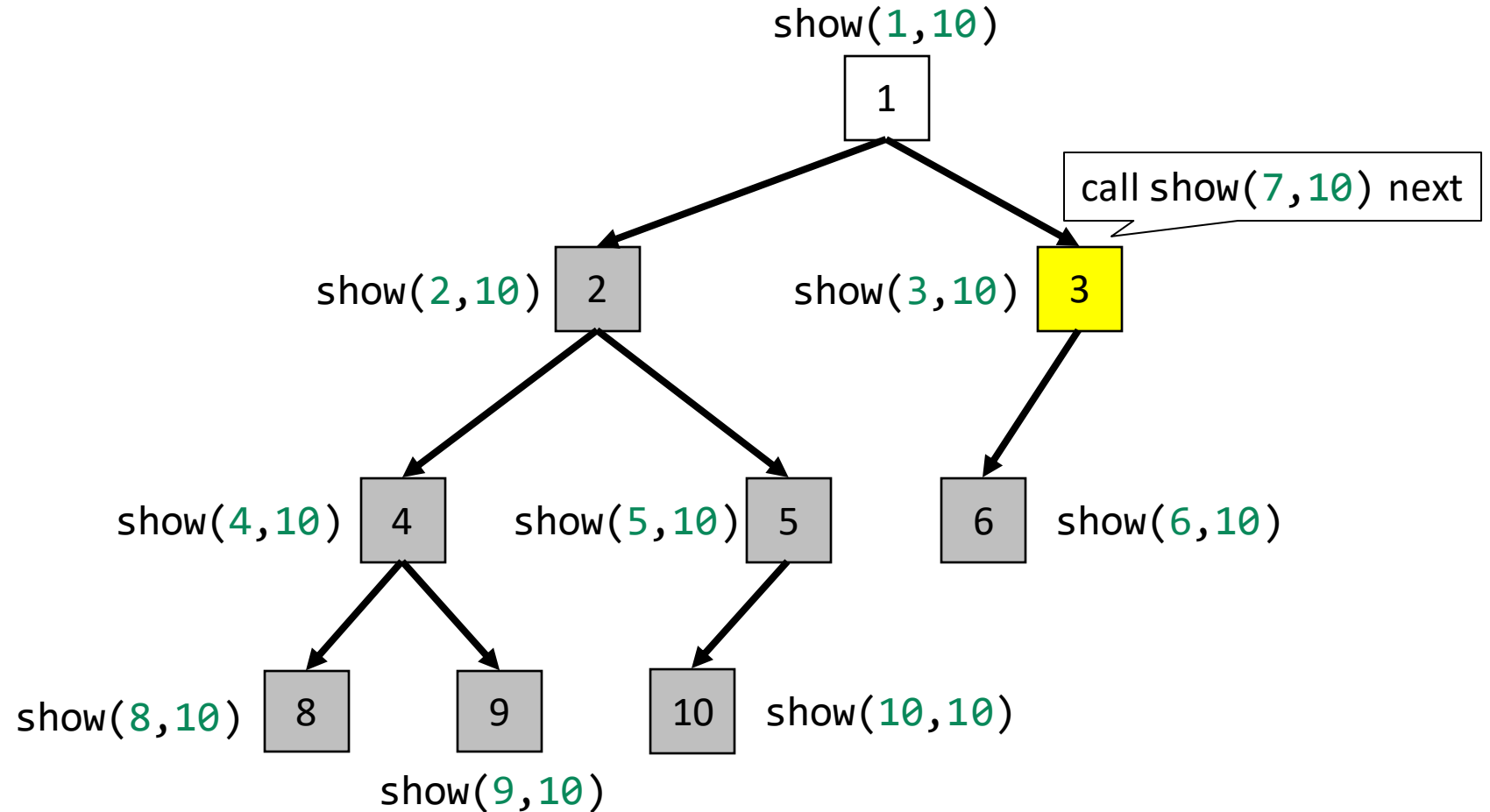
Since `3 <= 10`, we skip the if statement, and print 3.

```
def show(3, 10):  
    if 3 > 10:  
        return  
    print(3)  
    show(2*3, 10)  
    show(2*3+1, 10)
```



Output:

1  
2  
4  
8  
9  
5  
10  
3



We are now in `show(3, 10)`.

Since  $3 \leq 10$ , we skip the if statement, and print 3.

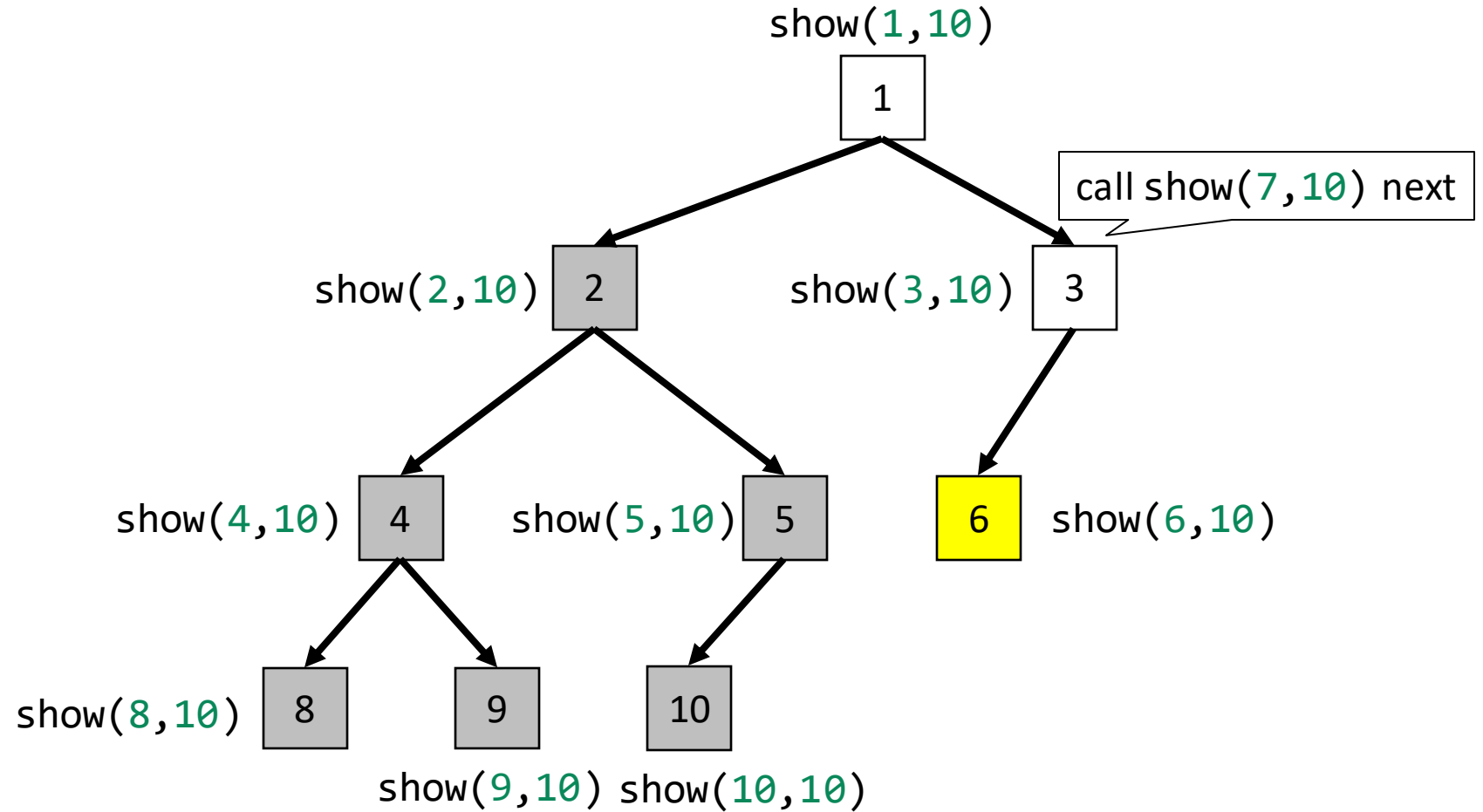
Then, we see that we should call `show(6, 10)` followed by `show(7, 10)`.

We call `show(6, 10)` first.

```
def show(6, 10):  
    if 6 > 10:  
        return  
    print(6)  
    show(2*6, 10)  
    show(2*6+1, 10)
```

Output:

1  
2  
4  
8  
9  
5  
10  
3  
6



We are now in show(6,10).

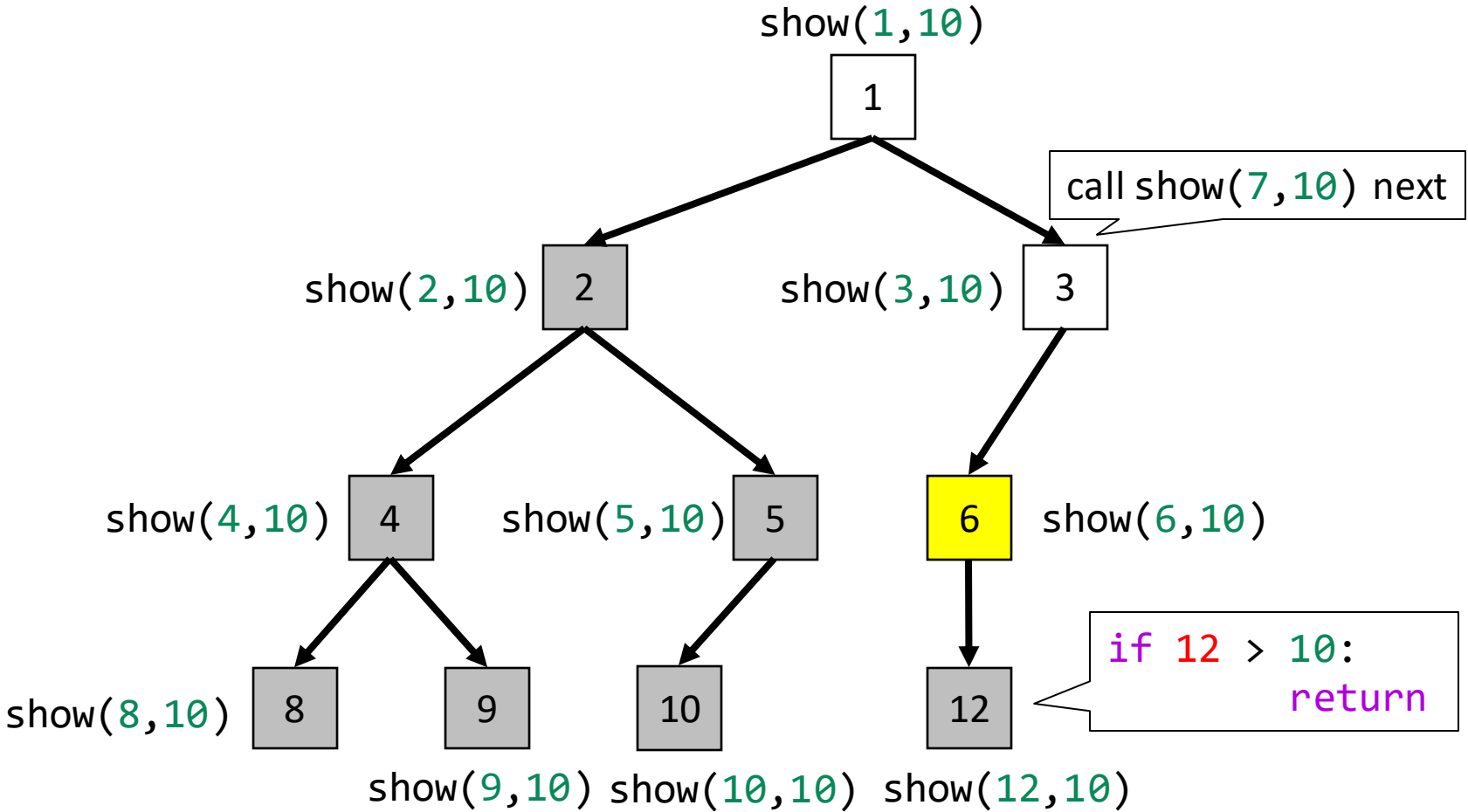
Since  $6 \leq 10$ , we skip the if statement, and print 6.

```
def show(6, 10):
    if 6 > 10:
        return
    print(6)
    show(2*6, 10)
    show(2*6+1, 10)
```



Output:

1  
2  
4  
8  
9  
5  
10  
3  
6



We are now in show(6,10).  
Since 6 <= 10, we skip the if statement, and print 6.  
Then, we see that we should call show(12,10) followed by show(13,10).  
We call show(12,10) first, but since 12 > 10, we return immediately.

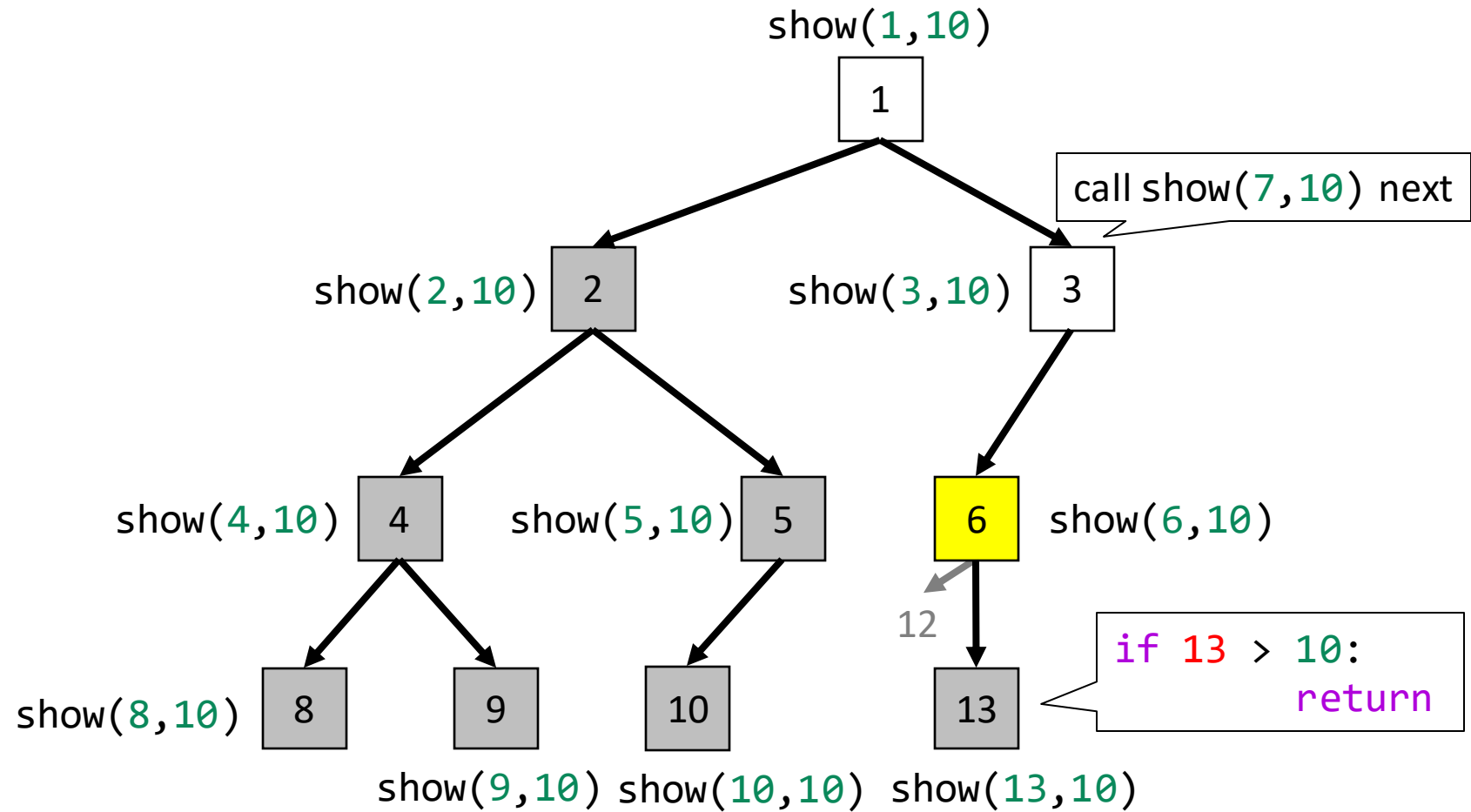


```
def show(6, 10):  
    if 6 > 10:  
        return  
    print(6)  
    show(2*6, 10)  
    show(2*6+1, 10)
```



Output:

1  
2  
4  
8  
9  
5  
10  
3  
6

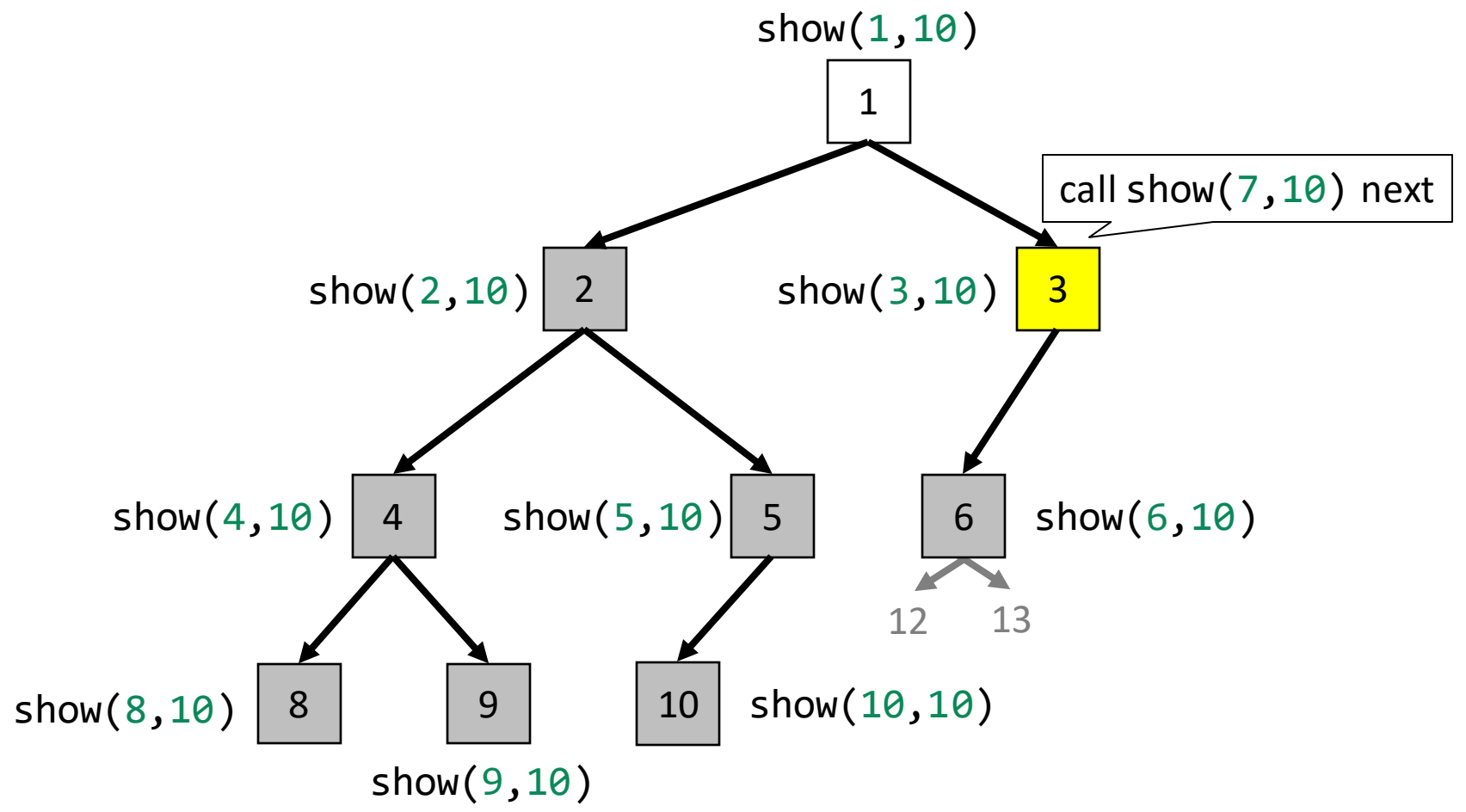


We return to show(6, 10) and call show(13, 10).  
Since 13 > 10, we return immediately.

```
def show(3, 10):  
    if 3 > 10:  
        return  
    print(3)  
    show(2*3, 10)  
    show(2*3+1, 10)
```

Output:

1  
2  
4  
8  
9  
5  
10  
3  
6



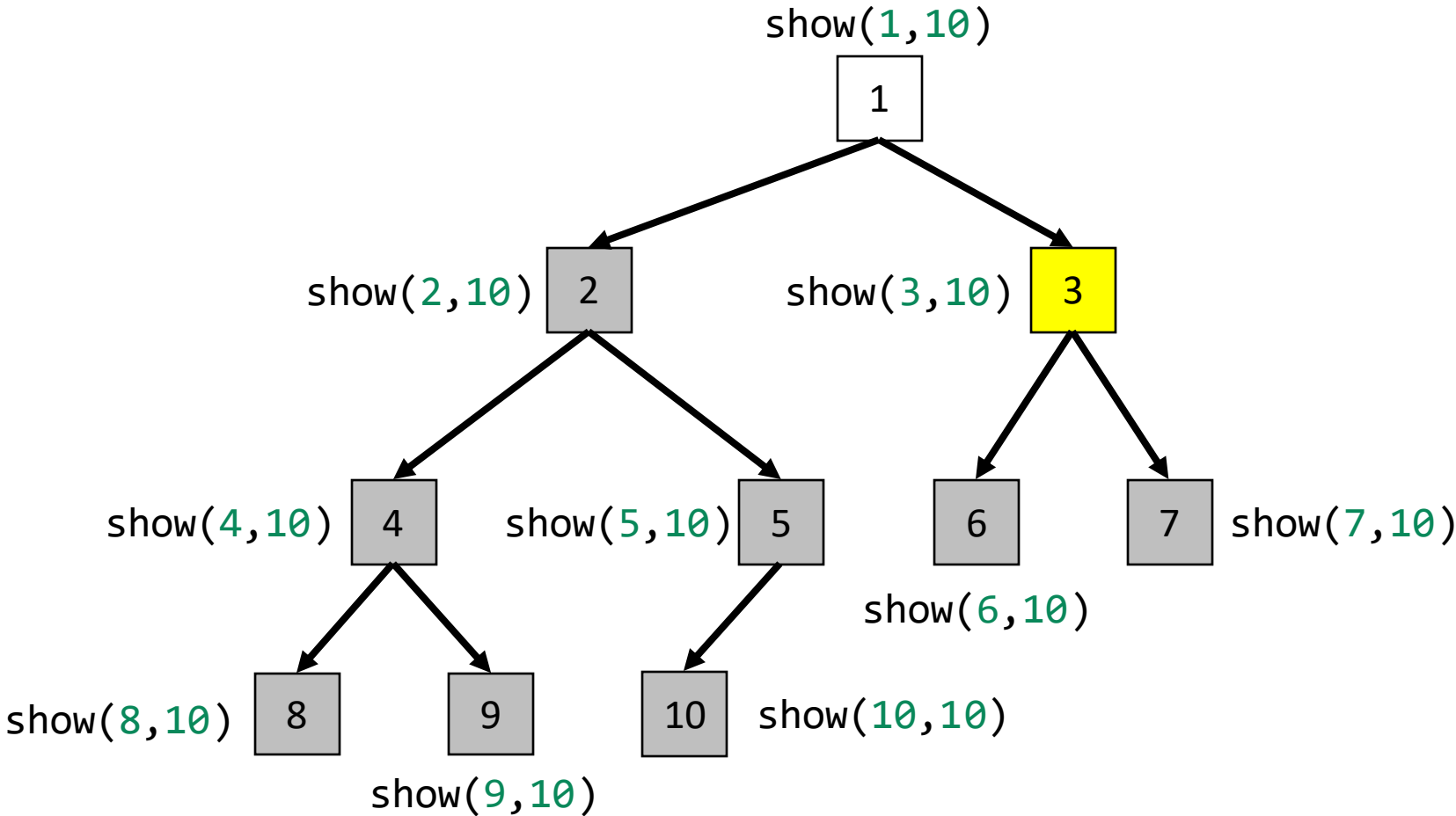
After returning from `show(13,10)`, we also finish `show(6,10)`.  
This means we are back in `show(3,10)`.

```
def show(3, 10):
    if 3 > 10:
        return
    print(3)
    show(2*3, 10)
    show(2*3+1, 10)
```



Output:

```
1
2
4
8
9
5
10
3
6
```

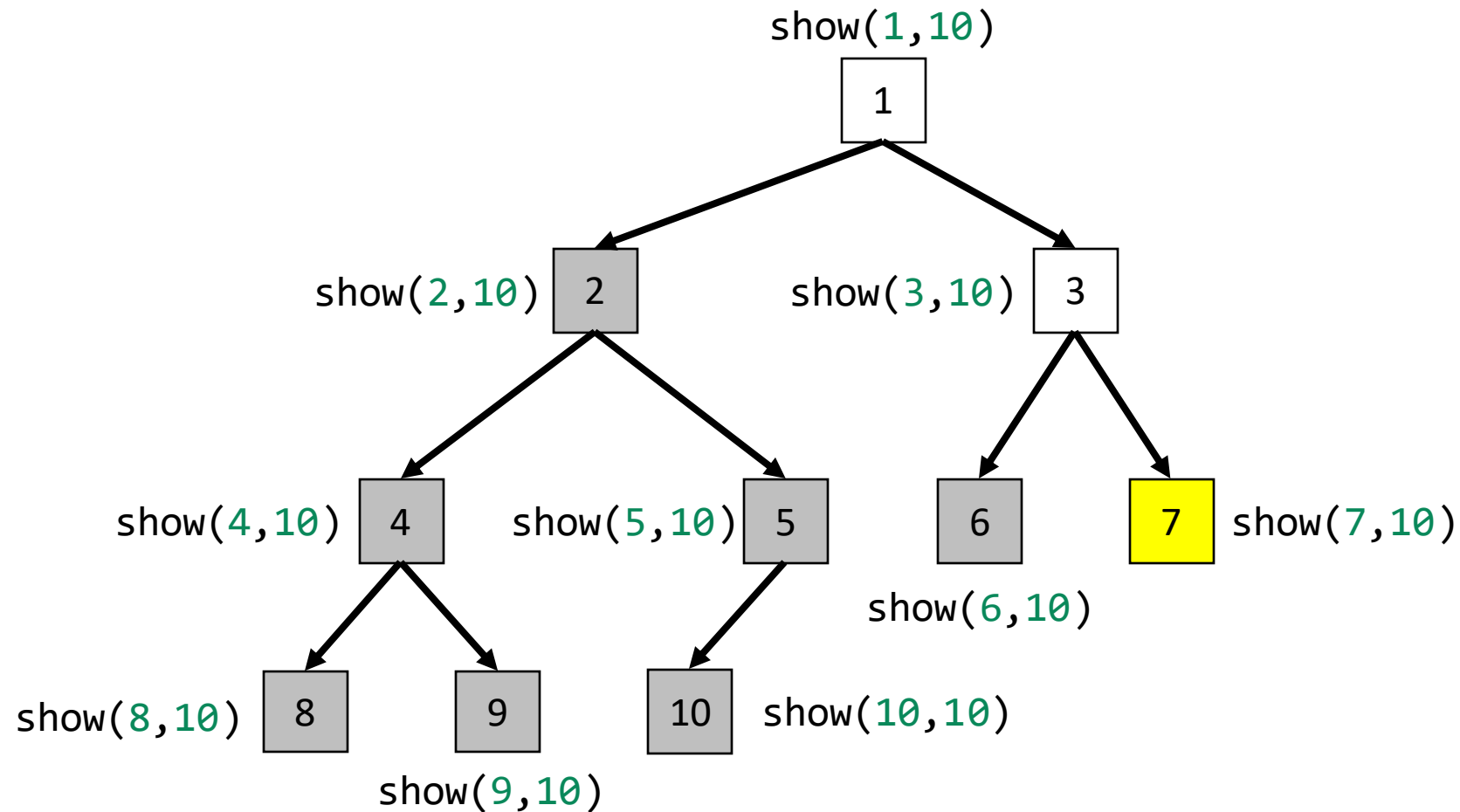


After returning from `show(13, 10)`, we also finish `show(6, 10)`. This means we are back in `show(3, 10)`. We now call `show(7, 10)`.

```
def show(7, 10):  
    if 7 > 10:  
        return  
    print(7)  
    show(2*7, 10)  
    show(2*7+1, 10)
```

Output:

1  
2  
4  
8  
9  
5  
10  
3  
6  
7



We are now in `show(7, 10)`.

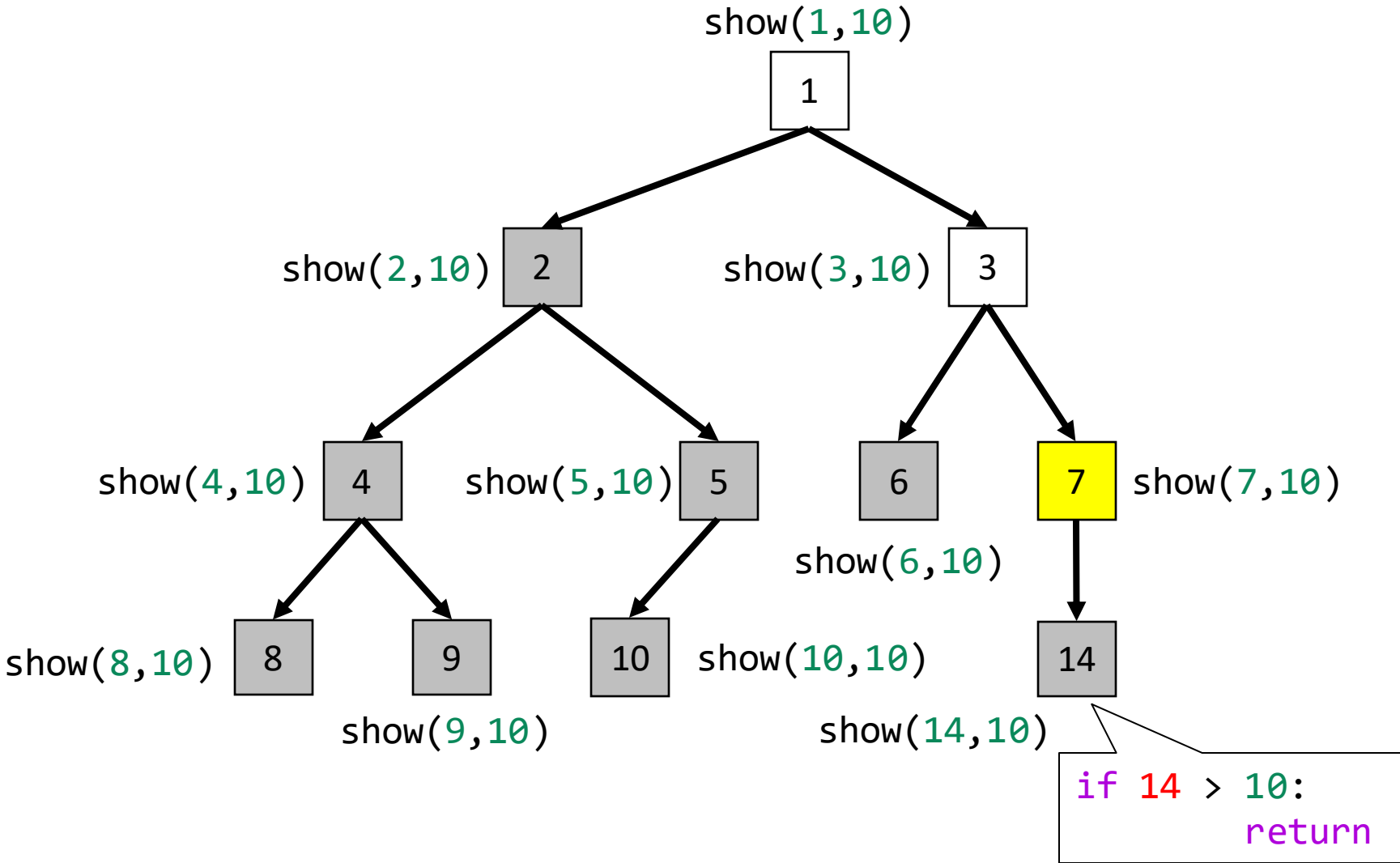
Since `7 <= 10`, we skip the if statement, and print 7.

```
def show(7, 10):
    if 7 > 10:
        return
    print(7)
    show(2*7, 10)
    show(2*7+1, 10)
```



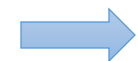
Output:

1  
2  
4  
8  
9  
5  
10  
3  
6  
7



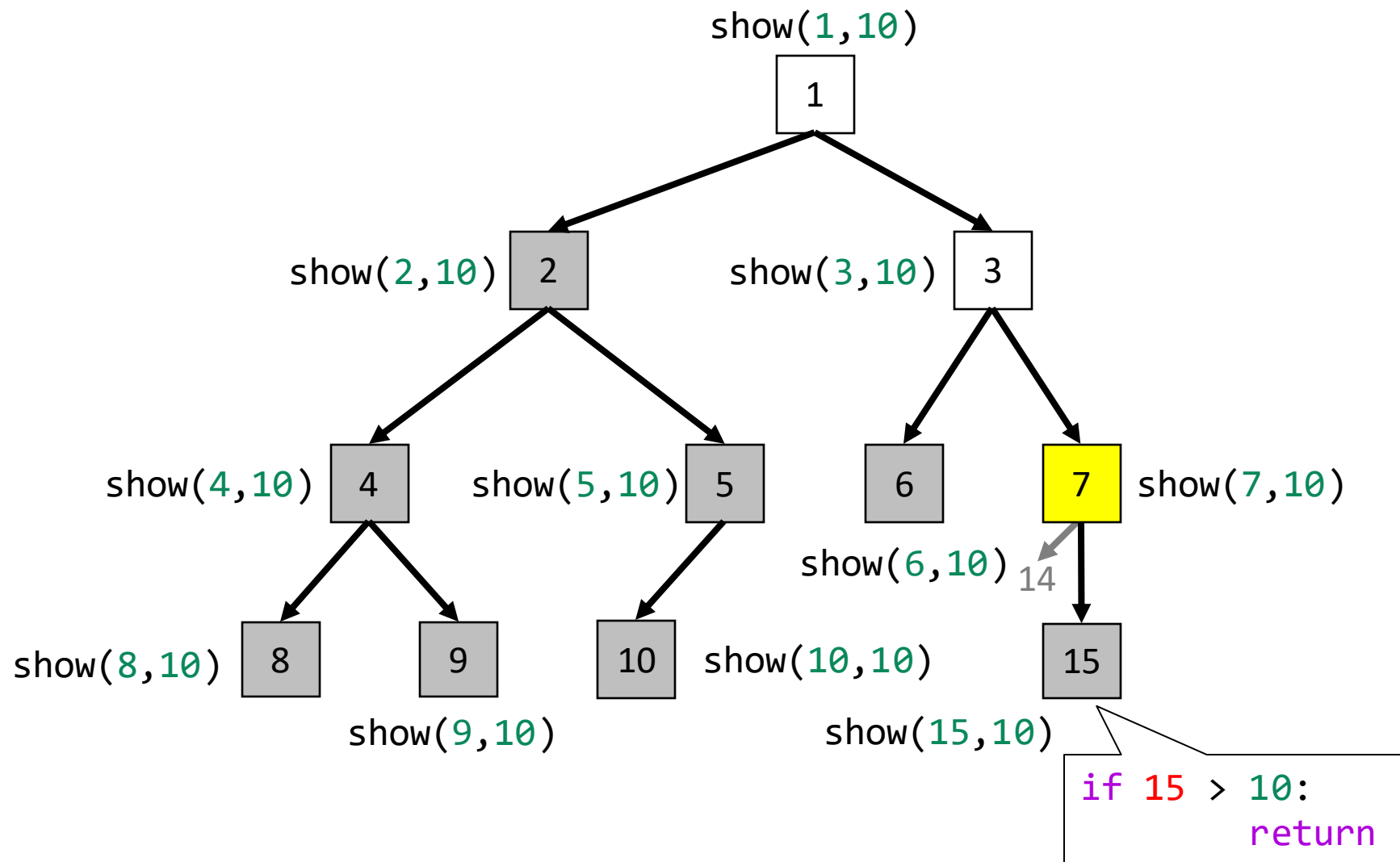
We are now in `show(7, 10)`.  
 Since `7 <= 10`, we skip the if statement, and print 9.  
 Then, we see that we should call `show(14, 10)` followed by `show(15, 10)`.  
 We call `show(14, 10)` first, but since `14 > 10`, we return immediately.

```
def show(7, 10):  
    if 7 > 10:  
        return  
    print(7)  
    show(2*7, 10)  
    show(2*7+1, 10)
```



Output:

1  
2  
4  
8  
9  
5  
10  
3  
6  
7

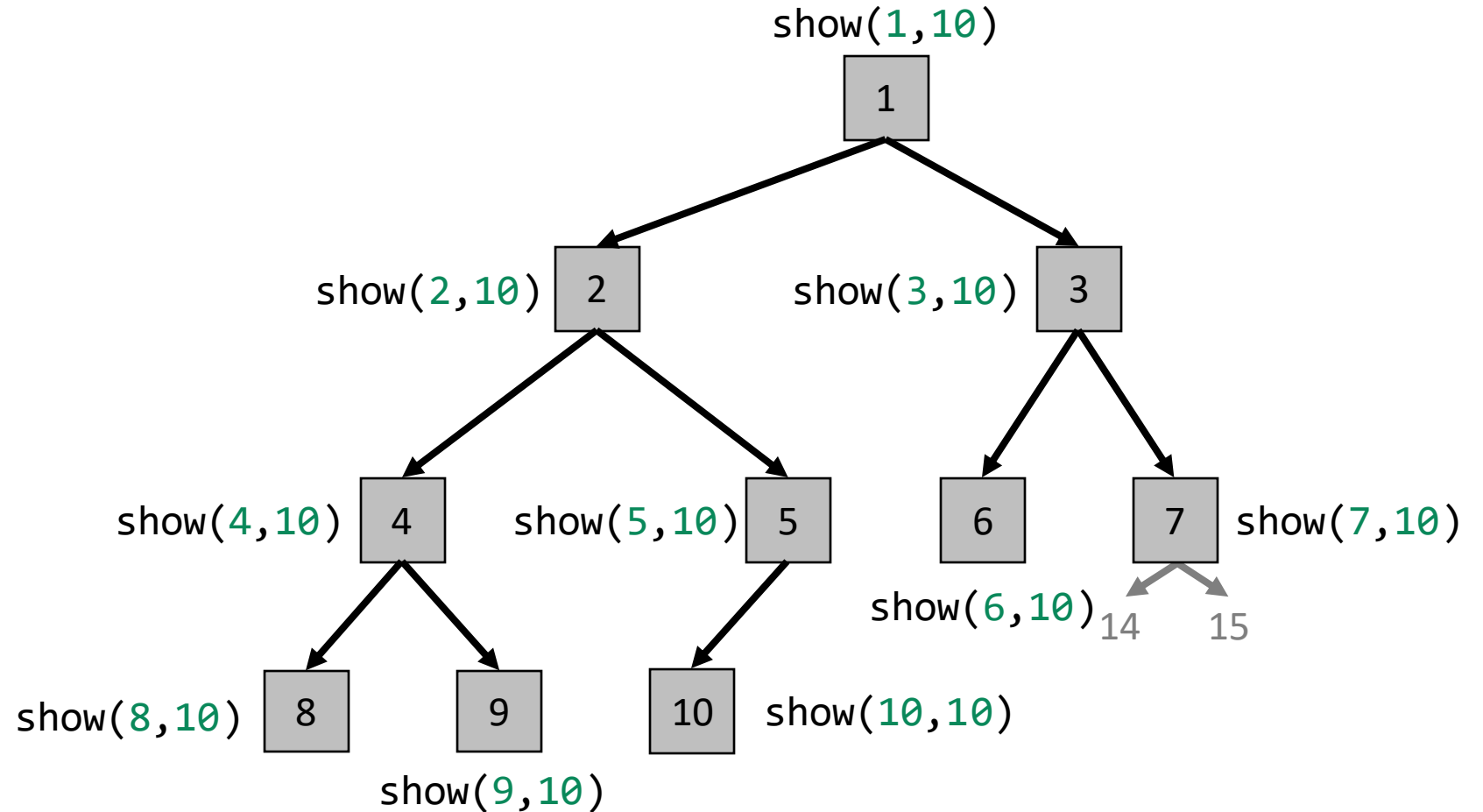


We return to `show(7, 10)` and call `show(15, 10)`.  
Since `15 > 10`, we return immediately.

```
def show(1, 10):  
    if 1 > 10:  
        return  
    print(1)  
    show(2*1, 10)  
    show(2*1+1, 10)
```

Output:

1  
2  
4  
8  
9  
5  
10  
3  
6  
7



We return to `show(7, 10)`.

Since we are done with `show(7, 10)`, we return to `show(3, 10)`.

But we are also done with `show(3, 10)`, so we return to `show(1, 10)`.

We are finally done with `show(1, 10)`.

# How Computers Execute Recursive Function Calls: Like Regular Functions

1. Allocate some memory to execute the function call
  2. Jump to the definition of the function
    - In a recursive function, back to the start of the function body
  3. Execute the function
  4. When finished, deallocate the memory and return to the caller
- For recursive functions, the computer may do 1-4 several times
  - You *never* need to think about recursion this way, except to note that
    - Step 1 implies that infinite recursion can use up memory and cause crashing
    - In a “branching” recursion, one branch is executed completely before the next
      - Helpful to know when printing debug output for recursive solutions



# The Wrong Way to Think About Recursion

- You are a programmer, not a computer
- DON'T do the computer's job
- DON'T try to run the recursion in your head
- DON'T think about all the steps

# How to Write Recursive Solutions

1. Look at the goal: create a function named after what you want to solve
  - Think carefully about the inputs and outputs
2. **Believe** that the function *already exists* and will give you the correct answer on some smaller subproblem(s)
  - This is typically **one step away** from the original problem
3. Using the answers *you already have* for the subproblem(s), build the answer for the original problem
4. Do the easy parts
  - Add the base case
  - Write the initial function call

# Example: Sorting

- A “natural” way:
  - Find the smallest item, put it in front
  - Find the next smallest item, put it in the second position
  - Find the 3<sup>rd</sup> smallest item, put it in the 3<sup>rd</sup> position
  - ...
  - Find the  $i$ th smallest item, put it in position  $i$
- Commonly called **selection sort**
- We can also express this recursively:
  - To sort a list of items, take the smallest item, put it in front, and *sort the remaining items*

# What We Need to Think About

- In the ancestors example, the subproblems already come from the natural recursive structure of the problem
  - “An ancestor is a parent or an ancestor of a parent”
- Here, we need to *invent* the recursive structure
- After finding the smallest element, **we need to perform some extra steps to build the subproblem**
  - If the smallest element is at index  $i$ , we need to form a list containing all the elements except element  $i$ :  $a[:i] + a[i+1:]$

# What We Need to Think About

- Sometimes, after we pass the work to the recursive call and get the answer back, we need to **perform extra steps to turn it into the answer for the original problem**

```
return (  
    is_ancestor(dad(self), person) or  
    is_ancestor(mom(self), person)  
)
```

# What We Need to Think About

- For sorting, once we have the smallest element and the sorted rest of the elements, we need to put them together to get the answer for the original problem

In “Pseudo-Code”...

```
def sort(a):  
    i = index_of_min(a)  
    return [a[i]] + sort(a[:i] + a[i+1:])
```

# With Base Case

```
def sort(a):  
    if len(a) == 1: # <= 1 to handle empty lists too  
        return a  
    else:  
        i = index_of_min(a)  
        return [a[i]] + sort(a[:i] + a[i+1:])
```



# How to Write Recursive Solutions

- DON'T try to run the recursion in your head
- DON'T think about all the steps
- Focus on how to move from *just one step* to the next

# How to Write Recursive Solutions

1. Look at the goal
2. **Believe** we can already do it for some smaller subproblem
3. Using the above, build the answer for the original problem
4. Do the easy parts

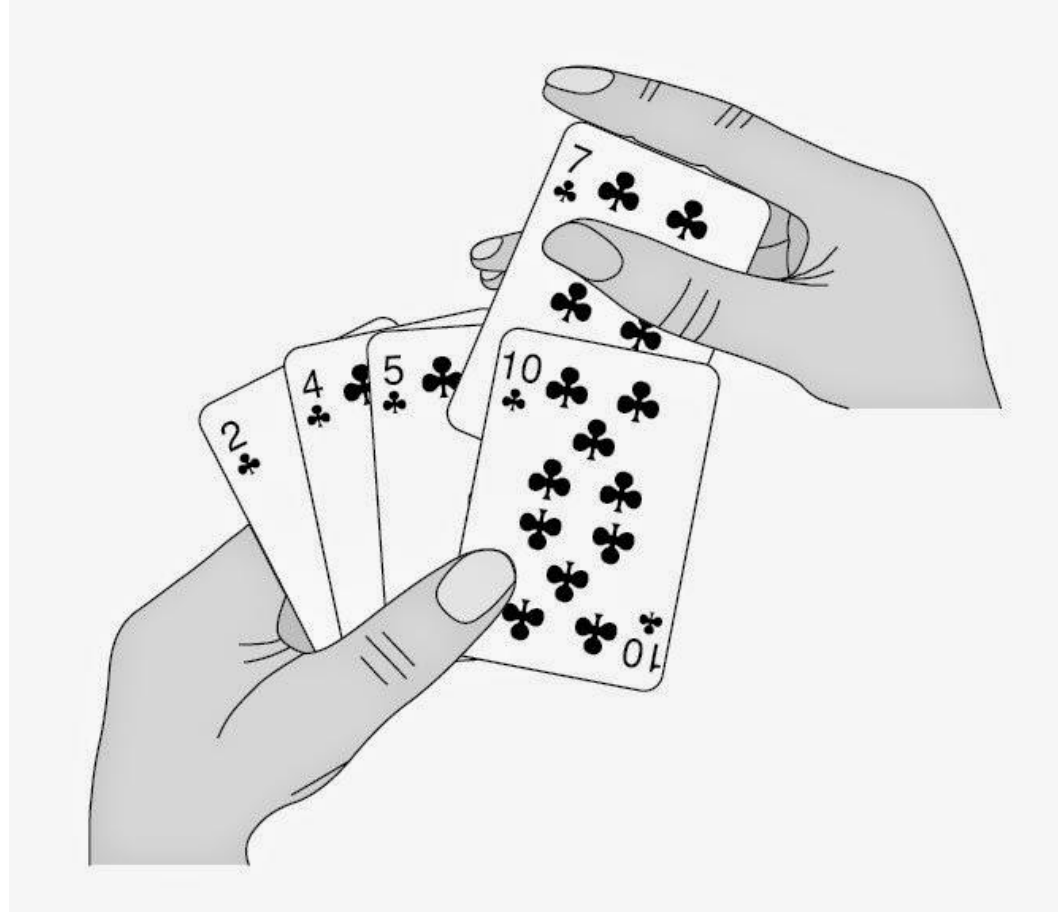
# Selection Sort: How Did We Come Up With It

1. Look at the goal
  - We want to sort
2. Believe we can already do it for some smaller subproblem
  - We can sort a list of size  $n - 1$
3. Using the above, build the answer for the original problem
  - If we want to sort  $n$  items but can only sort  $n - 1$ , then we must take one item out, sort the  $n - 1$  remaining items, and then merge the one element back into the sorted list
  - Take the smallest item out, put it in front
4. Do the easy parts

# Another Way to Do Step 3

1. Look at the goal
    - We want to sort
  2. Believe we can already do it for some smaller subproblem
    - We can sort a list of size  $n - 1$
  3. Using the above, build the answer for the original problem
    - If we want to sort  $n$  items but can only sort  $n - 1$ , then we must take one item out, sort the  $n - 1$  remaining items, and then merge the one element back into the sorted list
    - **Take any item out, and merge it back in by figuring out its correct position**
- Also known as **insertion sort**

# Another Way to Do Step 3



# Note

- You almost never have to write your own sort function
- C++: `sort(a.begin(), a.end())`
- Python: `a.sort()`
- Python (alternative): `sorted(a)`

# Exercises: Write Recursive Functions for the Following

1. Count the number of digits a given positive integer has, by repeatedly dividing by 10
2. Get the minimum of a given list of numbers
  - Pretend Python built-in `min` doesn't work on a list for now
3. Check if a string is a palindrome
  - A palindrome is a string which reads the same forwards or backwards
  - Example: Madam, I'm Adam
  - For simplicity, assume all characters of the input string are uppercase letters

# Counting the Number of Digits

- The number of digits of a number is one more than the number of digits of the number divided by 10 (ignoring the remainder)
- Or, if it's less than 10, it's a single-digit number



# Counting the Number of Digits

```
def num_digits(n):  
    if n < 10:  
        return 1  
    else:  
        return 1 + num_digits(n // 10)
```

# Getting the Minimum

- The minimum in a list is the smaller between the first element and the minimum among the rest
- Or, if there's only one element in the list, it's the minimum

# Getting the Minimum

```
def min_element(v):  
    if len(v) == 1:  
        return v[0]  
    else:  
        return min(v[0], min_element(v[1:]))
```

# Checking if a String is a Palindrome

- A string is a palindrome if its first and last characters match, and the substring without the first and last characters is a palindrome
- Or, if the string is empty or contains only a single character, it's already a palindrome
  - We could treat length 2 as a base case, but notice this requires repeating some logic that's already in the recursive case, and a simpler “zero” base case already covers it
    - Even if the empty string will never be given as input, this is worth doing
  - Convince yourself that the two solutions below are equivalent

# Checking if a String is a Palindrome

```
def is_palin(s):  
    if len(s) <= 1:  
        return True  
    else:  
        return s[0] == s[-1] and is_palin(s[1:-1])  
  
def is_palin(s):  
    if len(s) == 1:  
        return True  
    elif len(s) == 2:  
        return s[0] == s[1]  
    else:  
        return s[0] == s[-1] and is_palin(s[1:-1])
```

# An Actual Problem Where Recursion $>$ Loops

- We have a grid of boxes on a line as shown

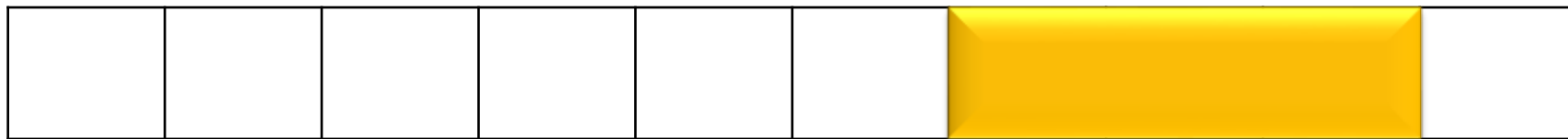


# An Actual Problem Where Recursion > Loops

- We also have two kinds of tiles
- A blue tile can cover any two consecutive boxes on the grid



- A yellow tile can cover any three consecutive boxes on the grid



# An Actual Problem Where Recursion > Loops

- We have unlimited numbers of blue and yellow tiles
- The goal is to cover the whole grid





# An Actual Problem Where Recursion > Loops

## **Allowed**

- Put yellow tiles next to each other
- Put blue tiles next to each other
- Use only blue tiles
- Use only yellow tiles

## **Not Allowed**

- Tiles not aligned to boxes
- Leaving some box uncovered

# An Actual Problem Where Recursion > Loops

- Another way to tile the grid



- Two ways are different if the color of the tile covering a box is different in one way from the other, for at least one box
- How many ways are there to tile a grid of  $n$  boxes?

# Try It Yourself First

- Draw all the possible ways to tile the grid above (10 boxes)
- How many ways are there?
- Any ideas for the algorithm for general  $n$ ?

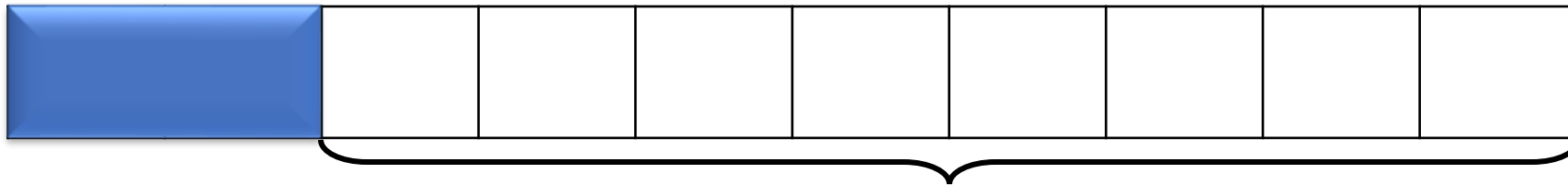
# Towards an Algorithm

- Hard to see the general pattern if the tiles are placed in random order
- Instead, place the pieces from left to right, without leaving any uncovered boxes between placed pieces

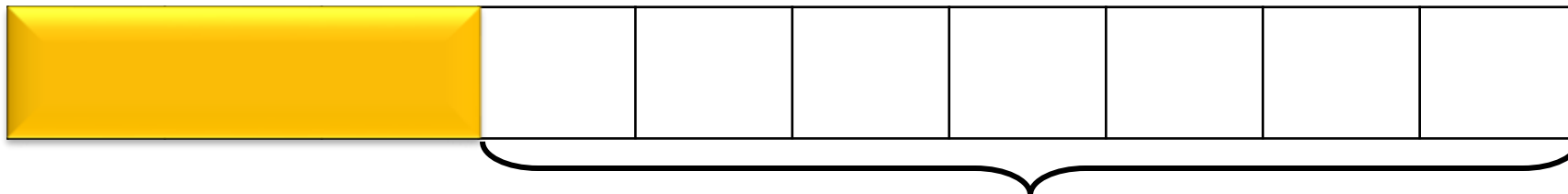


# Towards an Algorithm

- First tile can be blue or yellow
- After placing the first tile, need to continue *tiling rest of the grid*
  - It's the same problem, only smaller!



Same problem with  $n - 2$



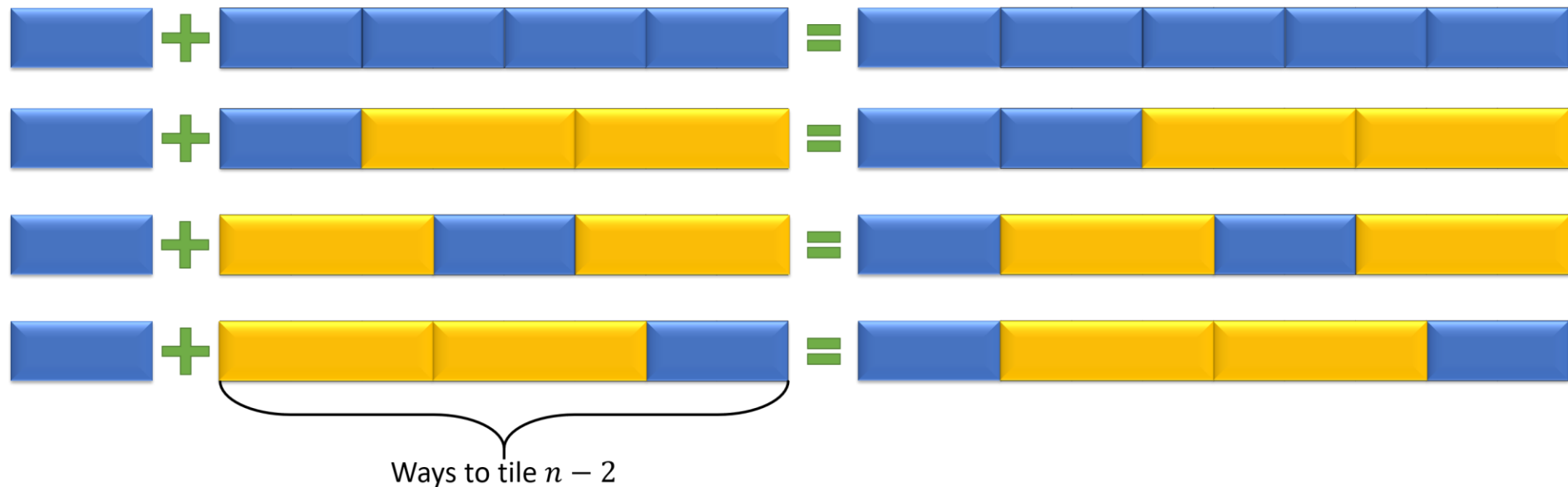
Same problem with  $n - 3$

# Towards an Algorithm

- Need to keep going? Should we think about all the steps?
- If only we already had a magic *blackbox* that counts the ways to tile a smaller grid
- With recursion, we can **believe** we already do!
- Believe we already have `ways(n - 2)` and `ways(n - 3)`
- Now what? How to use those to get `ways(n)`?

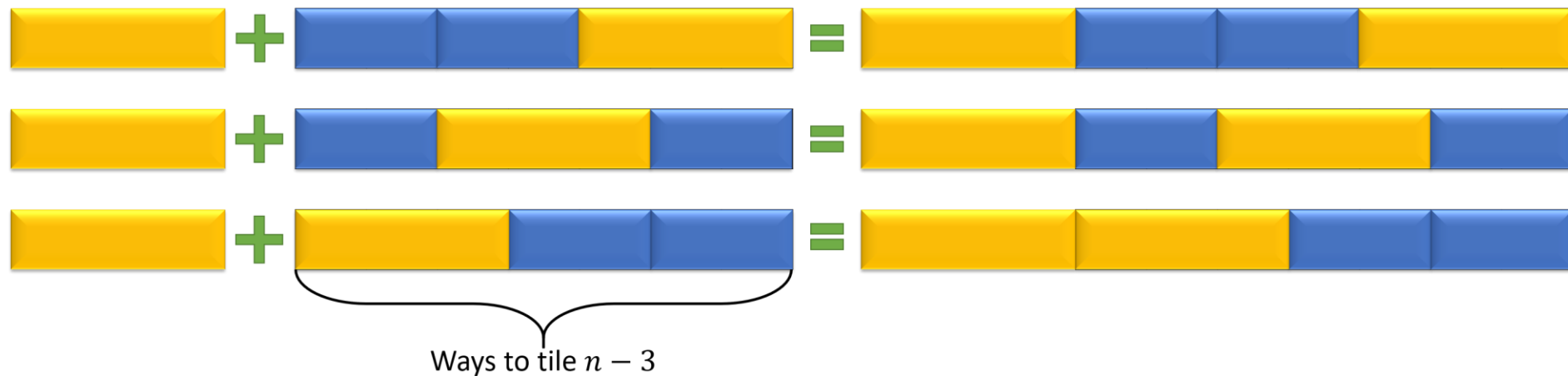
# Towards an Algorithm

- Take some ways to tile a grid with  $n - 2$  boxes and attach a blue tile to the left of all of them, we get ways to tile a grid with  $n$  boxes
- If the ways for  $n - 2$  boxes are all unique, then the ways for  $n$  boxes produced by this process must be all unique



# Towards an Algorithm

- Take some ways to tile a grid with  $n - 3$  boxes and attach a yellow tile to the left of all of them, we get ways to tile a grid with  $n$  boxes
- If the ways for  $n - 3$  boxes are all unique, then the ways for  $n$  boxes produced by this process must be all unique





# Towards an Algorithm

- Any way that begins with a blue tile is different from any way that begins with a yellow tile
- This means that all the ways produced by either of the above two processes must be different from each other
- This means that the number of ways to tile a grid with  $n$  boxes must be *at least the sum* of the number of ways we get from each process

# Towards an Algorithm

- There are no other ways to tile a grid with  $n$  boxes other than starting with either a blue tile or a yellow tile and then tiling the rest of the grid
- Therefore, if we have the complete number of ways to tile a grid with  $n - 2$  boxes for the first process and the complete number of ways to tile a grid with  $n - 3$  boxes for the second process, the two processes combined give all the ways to tile a grid with  $n$  boxes
- The number of ways to tile a grid with  $n$  boxes *is the sum* of the number of ways to tile a grid with  $n - 2$  boxes and of the number of ways to tile a grid with  $n - 3$  boxes

# In Code

```
def ways(n):  
    return ways(n - 2) + ways(n - 3)
```

# What About the Base Case(s)?

- If  $n = 1$ , we cannot even put any tiles, so the answer is 0
- If  $n = 2$ , we cannot start with a yellow tile
  - It is easy to see that there is only one way for this case
- Convince yourself that there is also only one way when  $n = 3$

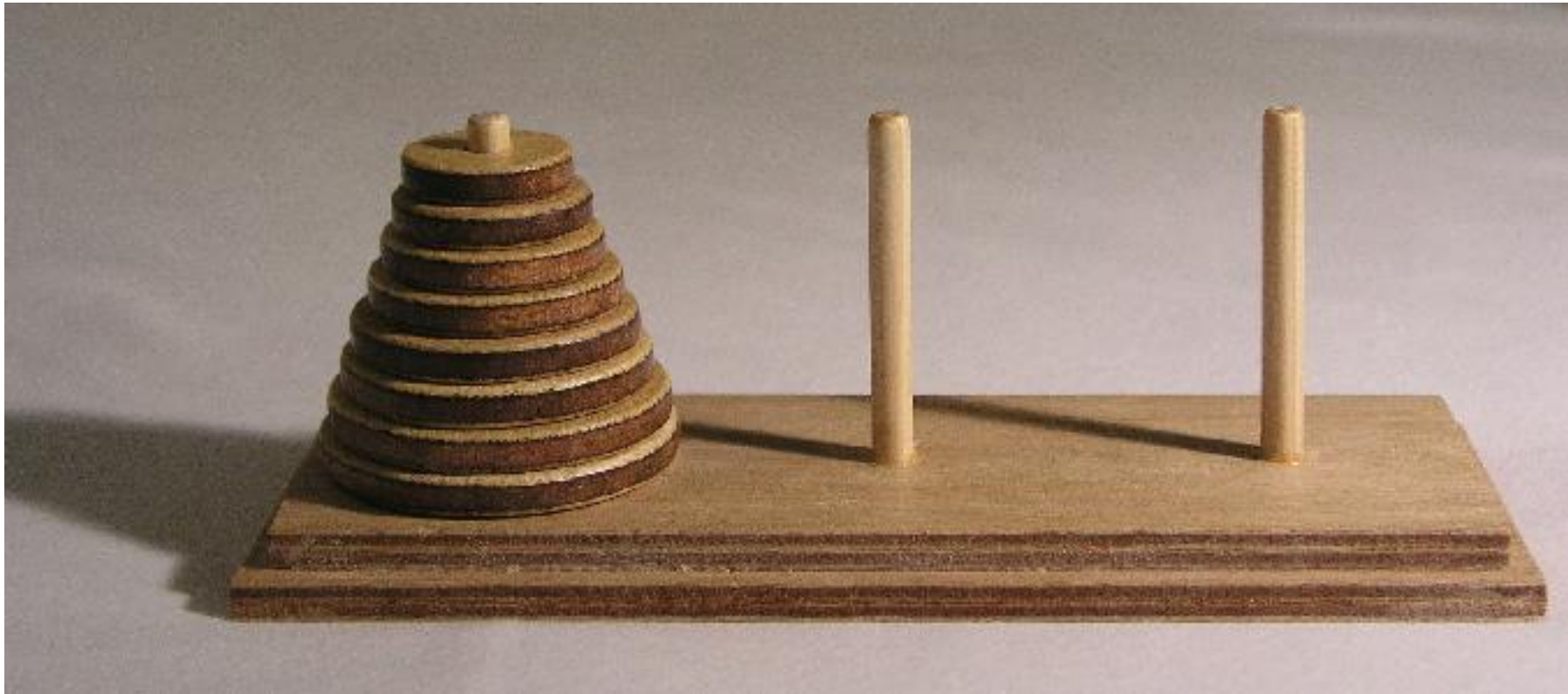
# The Complete Code

```
def ways(n):  
    if n == 1:  
        return 0  
    elif n == 2 or n == 3:  
        return 1  
    else:  
        return ways(n - 2) + ways(n - 3)
```

# Discuss

1. Why is it important to imagine placing the tiles in order from left to right?
  - Placing the tiles in a systematic order is important so that the subproblems look like the original problem; otherwise, recursion cannot work
  - If the first tile is forced to be at the leftmost position, the remaining problem is just like tiling an empty grid if we ignore the already occupied boxes
  - On the other hand, if the first tile is placed randomly, ignoring the already occupied boxes does not lead to “same but smaller” subproblems
2. Is it correct to imagine placing them from right to left instead?
  - Right to left also works
  - There are no other reasonable orderings

# Tower of Hanoi



# Tower of Hanoi

- Three rods and  $n$  disks, all of different sizes, initially stacked in increasing order of size on the first rod
- Move all disks from the first rod to the last rod, obeying the following rules:
  - Only one disk may be moved at a time
  - A move must take the top disk from one stack put it on top of another stack, or on an empty rod
  - No larger disk may be placed on top of a smaller disk
- Spend a few minutes playing to get a feel for the rules:  
<https://www.mathsisfun.com/games/towerofhanoi.html>

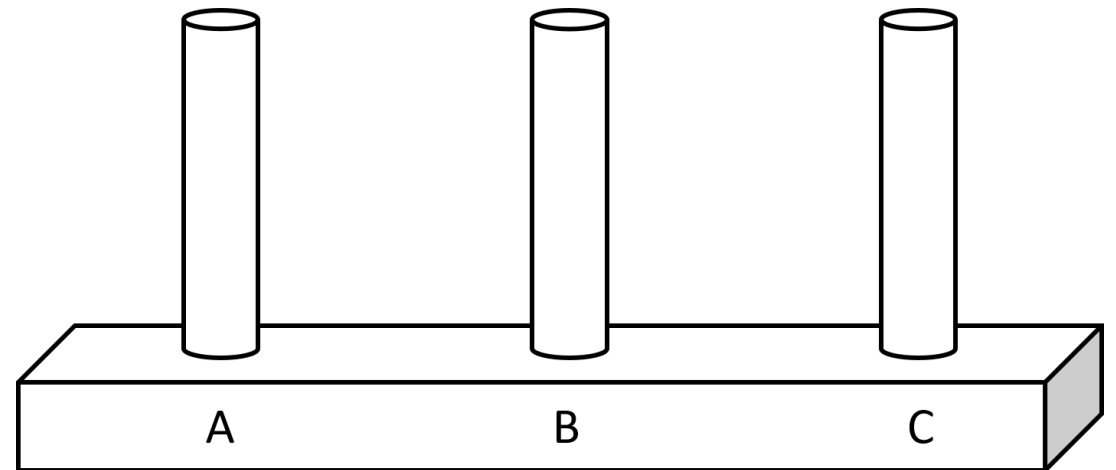


# A Tower of Hanoi Solver Program: The Input

- The number of disks  $n$  given as input
  - Otherwise, might as well solve manually and just hardcode the solution
- Initial placement of disks is always the same, so no need to include in the input

# A Tower of Hanoi Solver Program: The Output

- For convenience, label the rods from left to right A, B, and C
- Sentences of the form  
Move from {x} to {y}
  - {x} and {y} are A, B, or C
  - Understood to mean “take topmost disk from {x} and put it on top of the stack at rod {y},” since this is the only valid kind of move



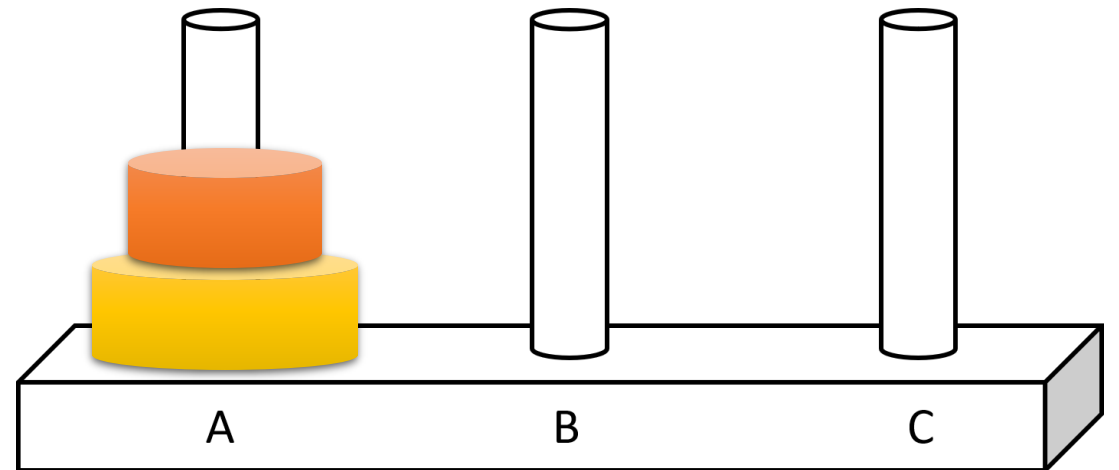
# Quick Check

- What is the output for  $n = 2$ ?

Move from A to B

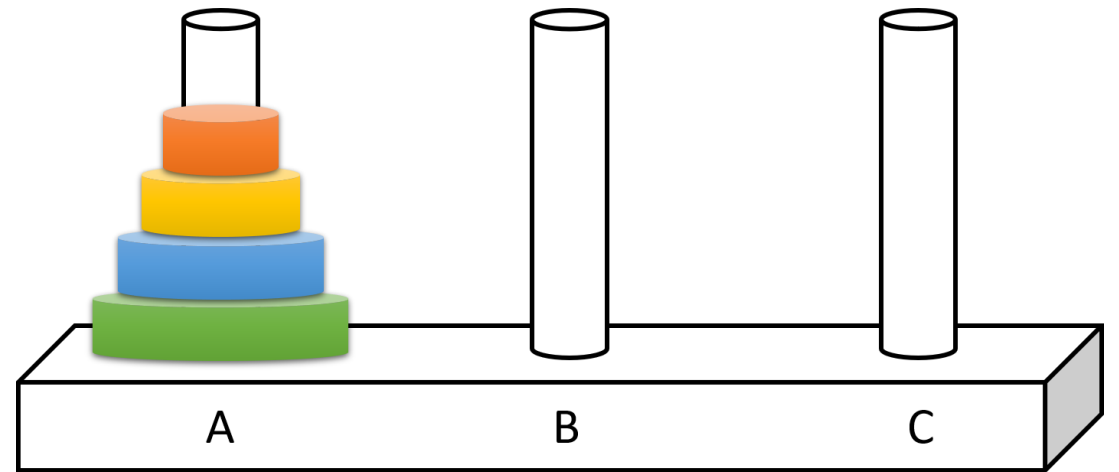
Move from A to C

Move from B to C



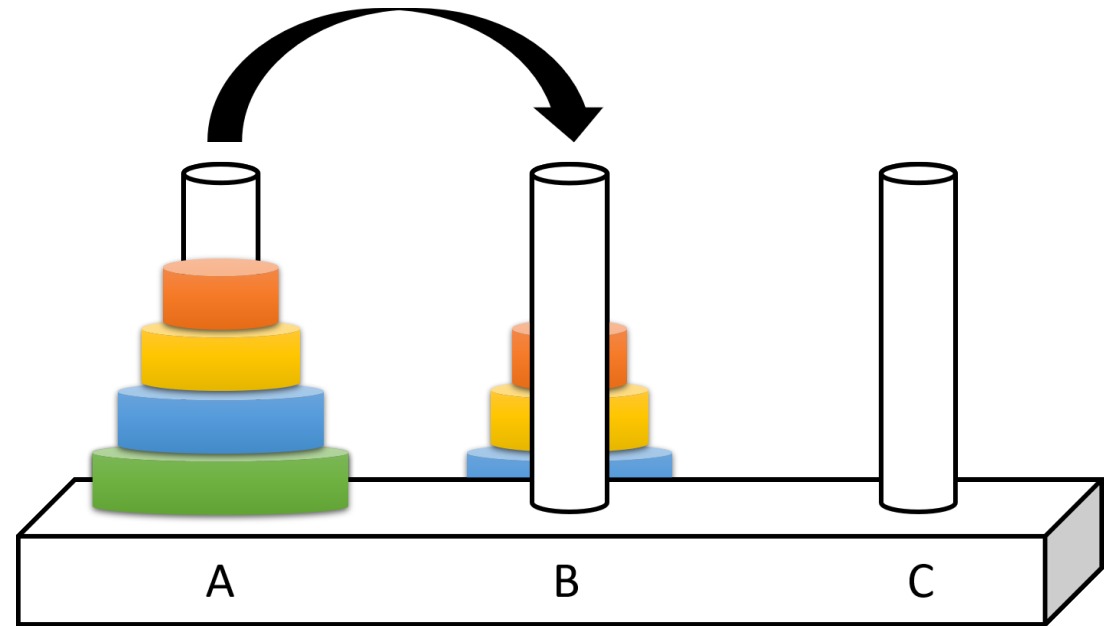
# Towards an Algorithm

- Examine the case  $n = 4$
- Thinking about all the moves is confusing; focus on just one step
- Starting from the smallest disk, no good way to decide which rod to move to
- Instead, turns out, *working backwards* from the largest disk is more helpful



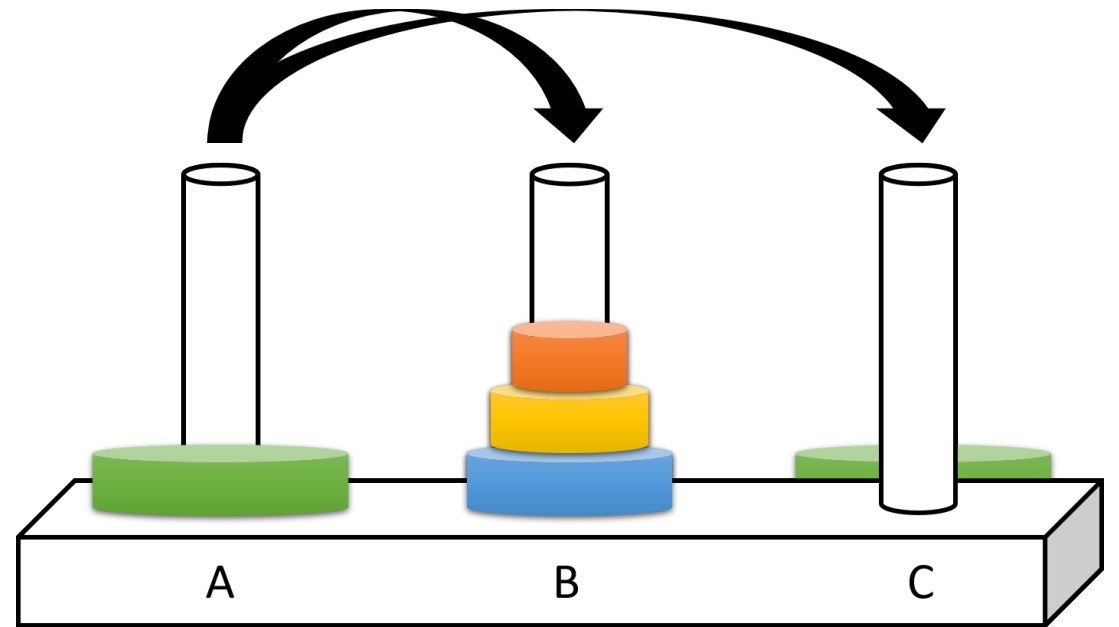
# Towards an Algorithm

- The largest disk must ultimately end up in C
- To move the largest disk from A to C, all the other disks must be out of the way – in B
- Without thinking about all the details, pretend there's some magic blackbox that takes all the smaller disks from A to B



# Towards an Algorithm

- Now we can move the largest disk from A to C
- Pretending we have another magic blackbox to move all the disks from B to C, we're done
- So easy!



# But We Cheated

- We used a magic blackbox
- How do we build such a blackbox in the first place?
- Notice that this task is similar to the original task: move a stack of disks from one rod to another
- So, we can use recursion!

# Something Like This

```
def move_disks(n):  
    move_disks(n = n - 1)  
    print('Move from A to C')
```



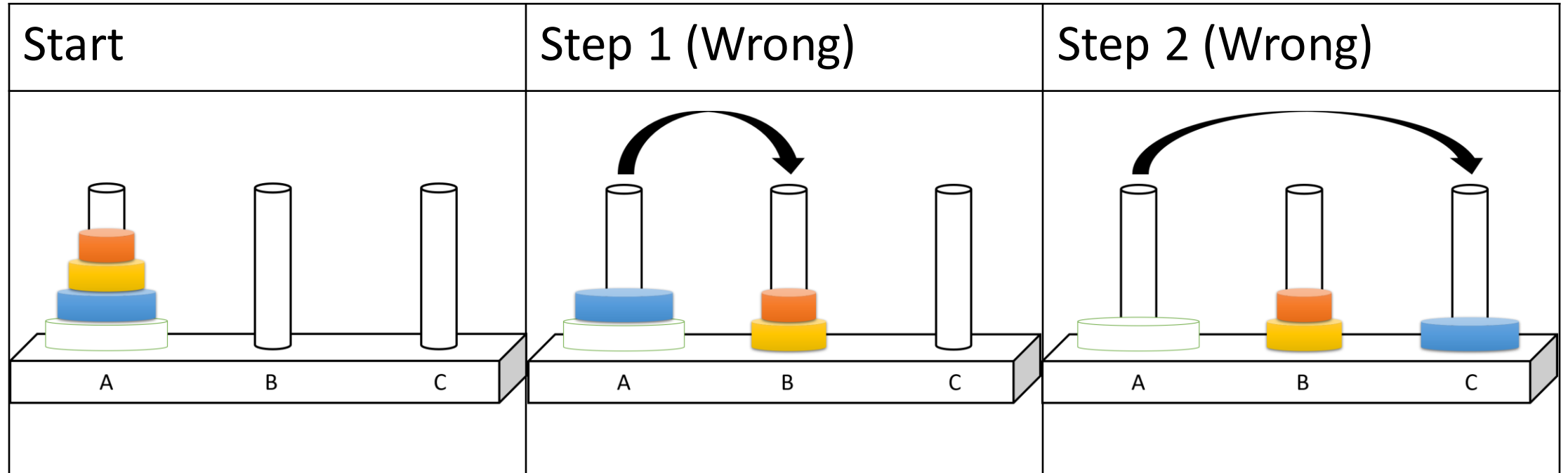
# Except It's Wrong

- The subproblem needs to be slightly different: instead of moving to the rightmost rod, move to the center rod
- Need to *introduce additional parameters* to the recursive function

# Fixing It

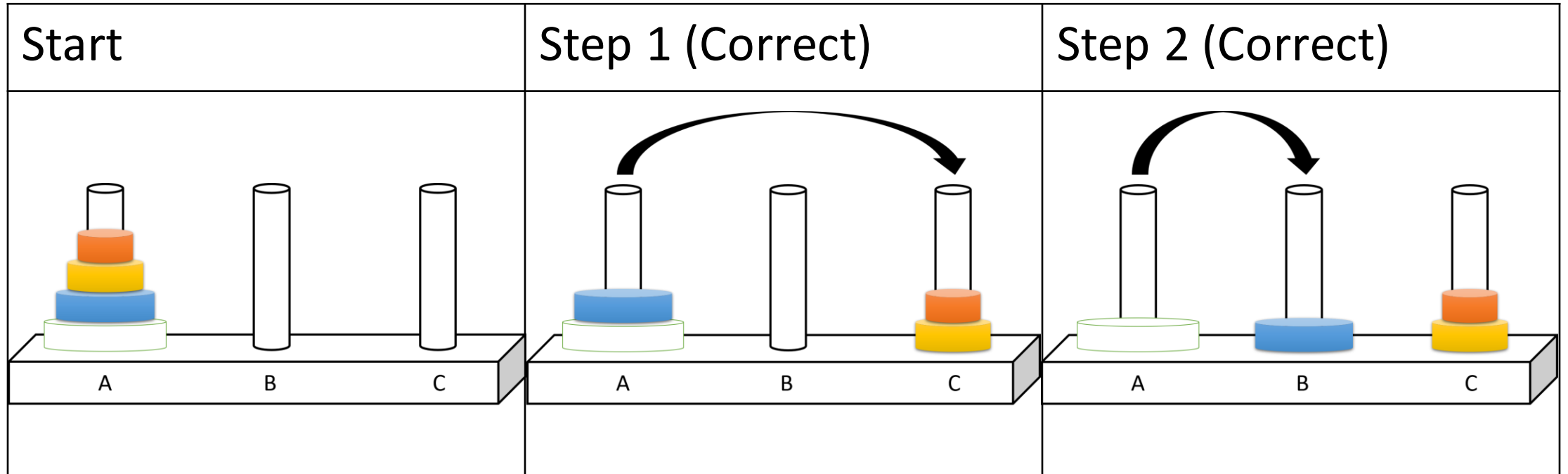
```
def move_disks(n, goal rod):  
    move_disks(n = n - 1, goal rod = 'B')  
    print(f'Move from A to {goal rod}')  
  
print(move_disks(n = int(input()), goal rod = 'C'))
```

# Except It's Wrong



- When  $n = 3$  and the goal rod is B, the two smallest disks need to move out of the way, not into rod B again, but into rod C

# Fixing It



- When  $n = 3$  and the goal rod is B, the two smallest disks need to move out of the way, not into rod B again, but into rod C

# Fixing It

- Instead of always moving “blocking disks” to the middle rod, move them to the rod which is not the goal rod

```
def move_disks(n, goal_rod):  
    move_disks(n = n - 1, goal_rod = 'B' if goal_rod == 'C' else 'C')  
    print(f'Move from A to {goal_rod}')  
  
print(move_disks(n = int(input()), goal_rod = 'C'))
```

# Almost Done

- Now, all the smaller disks are in the middle rod, and the largest disk is in the rightmost
- Now we need to move the smaller disks from the middle rod to the rightmost rod
- How?
- This looks like the original problem again, so recursion again!

# Except It's Wrong

- This time, the rod we start from is not necessarily the leftmost rod
- Suggests we need to introduce another parameter

# Fixing It

```
def move_disks(n, start_rod, goal_rod):  
    move_disks(n = n - 1, )  
    print(f'Move from {start_rod} to {goal_rod}')  
    move_disks(n = n - 1, )  
  
print(move_disks(n = int(input()), start_rod = 'A', goal_rod = 'C'))
```



# Think Carefully What Values to Pass to Recursive Calls

- When moving blocking disks out of the way, starting rod is the same as the starting rod of the whole stack

```
def move_disks(n, start_rod, goal_rod):  
    move_disks(n = n - 1, start_rod = start_rod, )  
    print(f'Move from {start_rod} to {goal_rod}')  
    move_disks(n = n - 1, )
```

# Think Carefully What Values to Pass to Recursive Calls

- Blocking disks must be moved to the rod which is neither the start rod nor the goal rod
  - Not always B or C: for example, when moving a stack from B to C
- Easy way to avoid complicated conditionals: loop through all of ABC

```
def move_disks(n, start_rod, goal_rod):  
    for rod in 'ABC':  
        if rod not in [start_rod, goal_rod]:  
            move_blockers_to = rod  
            move_disks(n = n - 1, start_rod = start_rod, goal_rod = move_blockers_to)  
            print(f'Move from {start_rod} to {goal_rod}')  
            move_disks(n = n - 1, )
```

# Think Carefully What Values to Pass to Recursive Calls

- After moving the largest disk, blocking disks must be taken from where we left them in the first recursive call, and moved to where we wanted to move the entire stack

```
def move_disks(n, start_rod, goal_rod):  
    for rod in 'ABC':  
        if rod not in [start_rod, goal_rod]:  
            move_blockers_to = rod  
    move_disks(n = n - 1, start_rod = start_rod, goal_rod = move_blockers_to)  
    print(f'Move from {start_rod} to {goal_rod}')  
    move_disks(n = n - 1, start_rod = move_blockers_to, goal_rod = goal_rod)
```

# Does It Work If You Run It?

## Did We Forget Something?

- Right, the base case!
- If  $n = 1$ , no recursive calls needed, just move that one disk

```
def move_disks(n, start_rod, goal_rod):  
    if n == 1:  
        print(f'Move from {start_rod} to {goal_rod}')  
    else:  
        ...
```

# Simplifying the Base Case

- Notice repeated `print(f'Move from {start_rod} to {goal_rod}')`
- Even easier base case:  $n = 0$ , do nothing

```
def move_disks(n, start_rod, goal_rod):  
    if n == 0:  
        return  
    # can also "if n > 0:" and erase previous two lines  
    else:  
        ...
```

# Simplifying the Base Case

- If you're not convinced this works, try tracing the code for  $n = 1$
- $n = 0$  is not really a meaningful input to the original problem
- But we're allowed to invent this case as long as we handle it properly
- And we invent such cases for convenience
- This is typical for lots of recursive functions: a “zero case” makes things simpler even if it is never going to be a supplied input
- Don't worry about it if it's confusing – just choose base cases that make sense to you

# Exercise

- We can get rid of the for-loop in the code above by keeping track of what the “temporary” rod is at all times – in other words, by introducing another parameter
- Complete the definition below

```
def move_disks(n, start_rod, temp_rod, goal_rod):  
    ...
```

# Solution

```
def move_disks(n, start_rod, temp_rod, goal_rod):  
    if n > 0:  
        move_disks(n - 1, start_rod, goal_rod, temp_rod)  
        print(f'Move from {start_rod} to {goal_rod}')  
        move_disks(n - 1, temp_rod, start_rod, goal_rod)  
  
print(move_disks(int(input()), 'A', 'B', 'C'))
```



# Practice Problems

- <https://progvar.fun/problemsets/recursion>