Fundamentals of Recursion

Lecture Flow

1. Pre-requisites
2. Problem definitions
3. Basic Concepts
4. Time and Space Complexity of Data Structure
5. Common pitfalls
6. Practice questions and Resources
7. Quote of the day

Pre-requisites

- Understanding iterative program flow
- Functions, local variables and global variables
- Stack data structures
- Willingness to learn

Thinking recursively

Recursion is an important concept in computer science. It is a foundation for many other algorithms and data structures. However, the concept of recursion can be tricky to grasp for many beginners.

Real Life Example

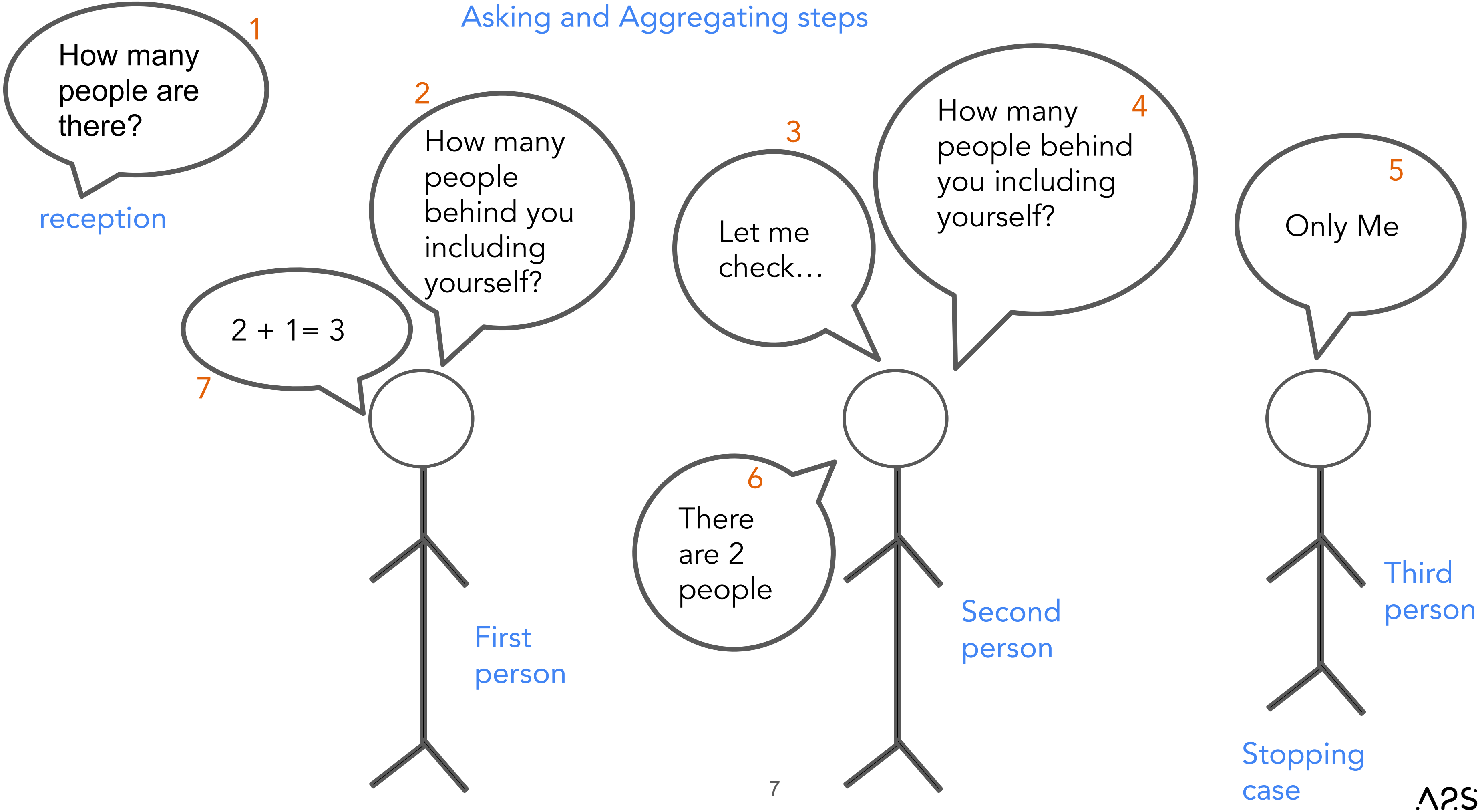
Imagine you are in a long queue and the reception asks you how many people there are in the line. How do you count the number of people in the queue?

Brainstorm

- Let’s assume each person in the queue
  - Can increment numbers by 1
  - Can request other people
- What if I asked the person behind me “how many people there are” and whatever their response is, I can do plus 1 and inform the reception? And
- What if the person I asked, asks the person behind them “how many people there are” and whatever they get, they do plus 1 and inform me?

⋮

Asking and Aggregating steps


1
How many people are there?
reception
2
How many people behind you including yourself?
7
2 + 1= 3
First person
3
Let me check…
4
How many people behind you including yourself?
6
There are 2 people
Second person
5
Only Me
Third person
Stopping case

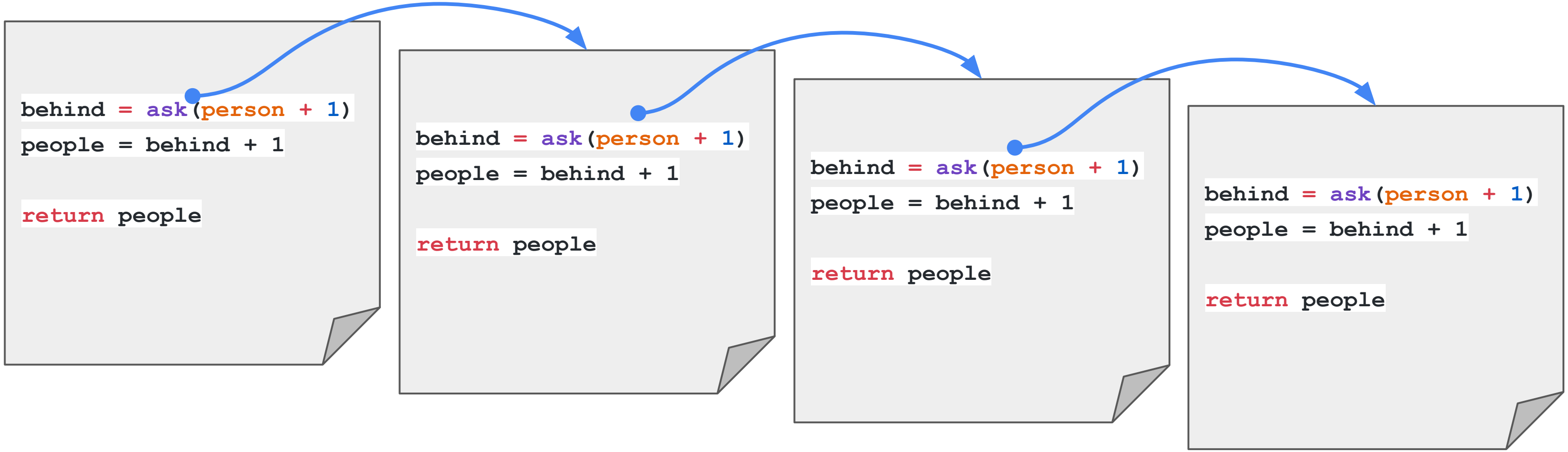
Can we simulate this with code?

- Since we are doing the same thing again and again let's use one function to do the task

Does this work?

```python
def ask(person):

    behind = ask(person + 1)
    people = behind + 1

    return people
```

What happens?



```
behind = ask(person + 1)
people = behind + 1

return people
```

```
behind = ask(person + 1)
people = behind + 1

return people
```

```
behind = ask(person + 1)
people = behind + 1

return people
```

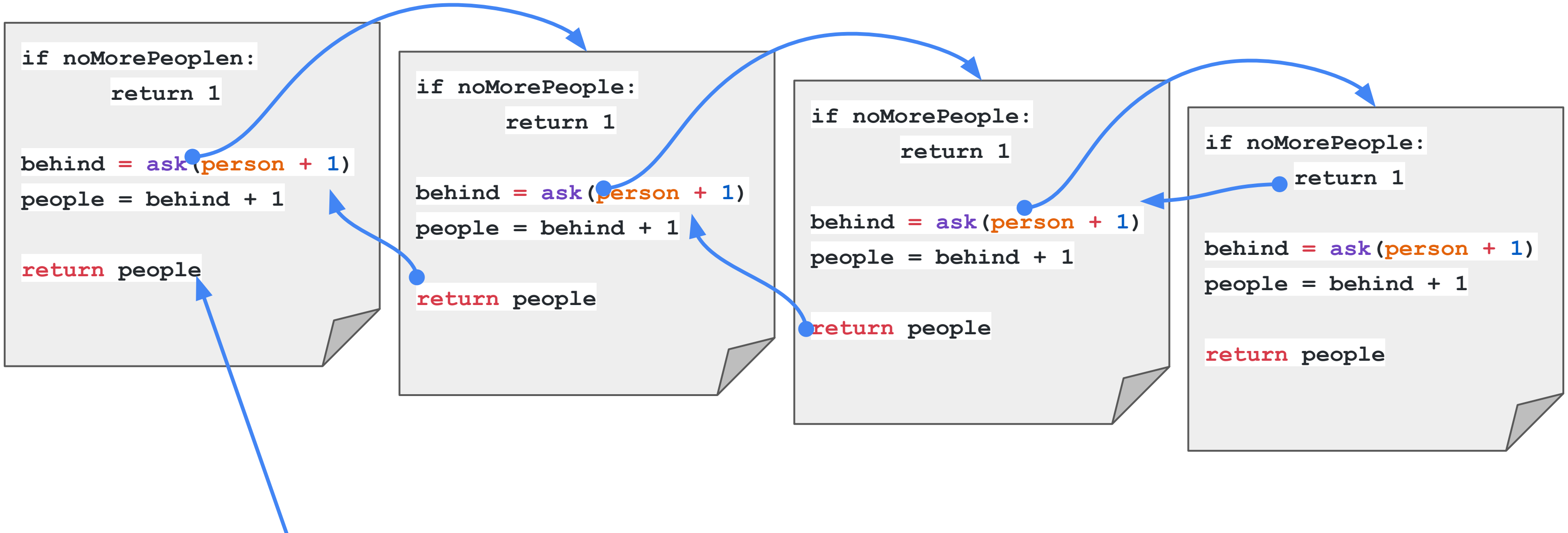
```
behind = ask(person + 1)
people = behind + 1

return people
```

We will never get to return because we are calling the function non stop. What should we do?

Does this work?

```
def ask(person):
    if noMorePeople:
        return 1

    behind = ask(person + 1)
    people = behind + 1

    return people
```

What happens?



```
if noMorePeoplen:
      return 1

behind = ask(person + 1)
people = behind + 1

return people
```

```
if noMorePeople:
      return 1

behind = ask(person + 1)
people = behind + 1

return people
```

```
if noMorePeople:
      return 1

behind = ask(person + 1)
people = behind + 1

return people
```

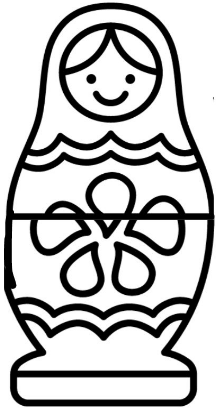
```
if noMorePeople:
      return 1

behind = ask(person + 1)
people = behind + 1

return people
```

Then now we return the number of people to our first caller

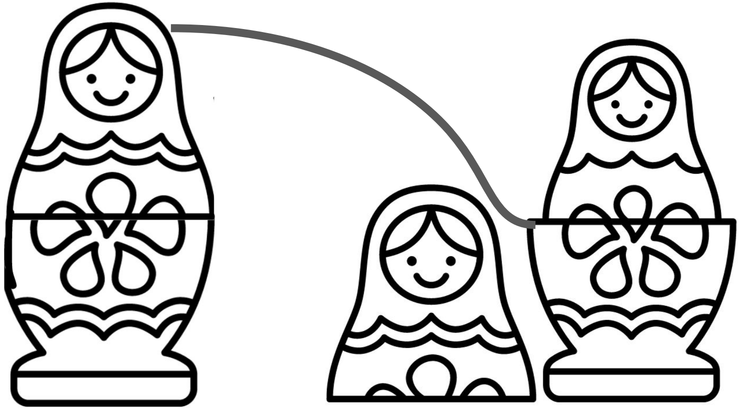
Real Life Example 2

Imagine some one give you a matryoshka and ask you to count the number dolls, how would you solve the problem?

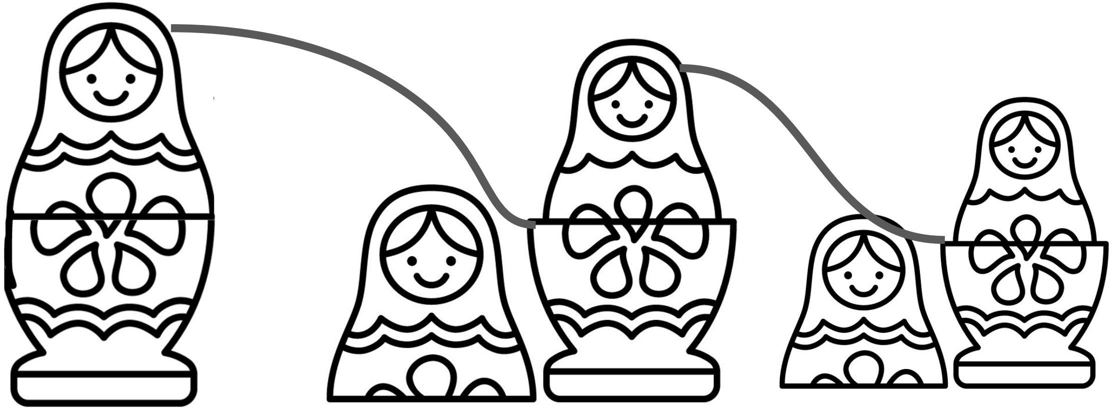
Brainstorm



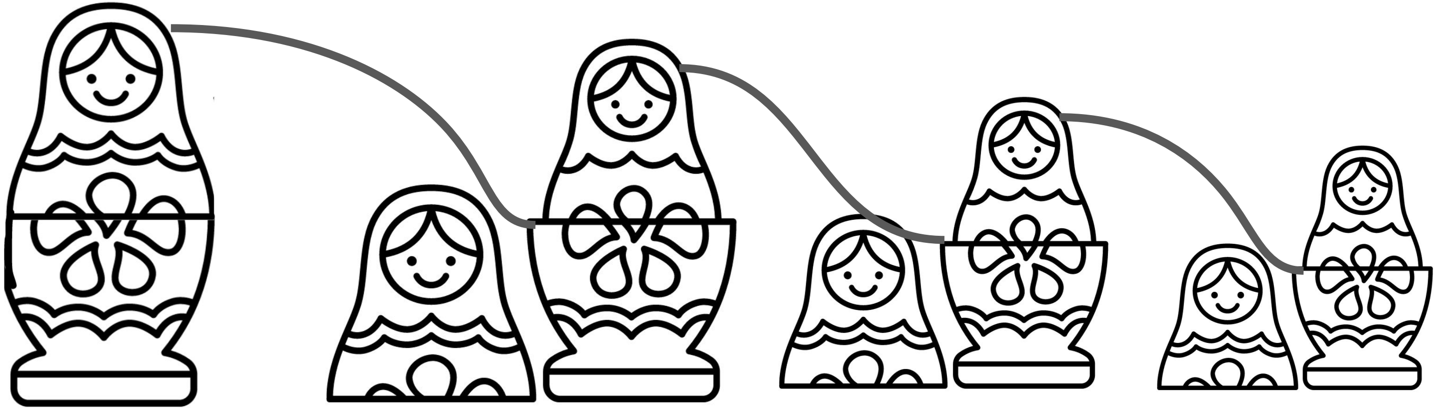
Brainstorm



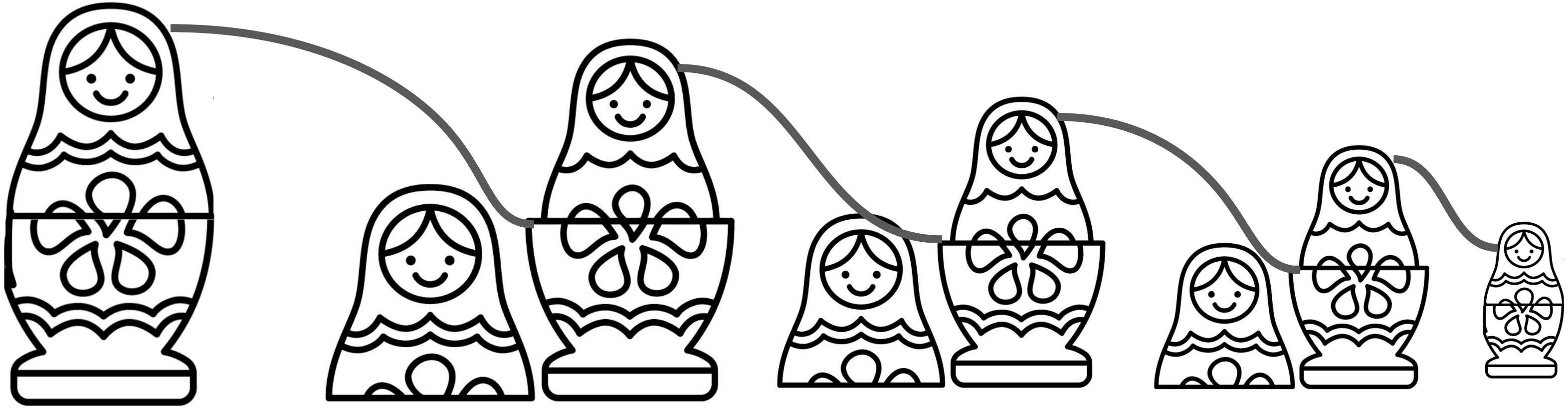
Brainstorm



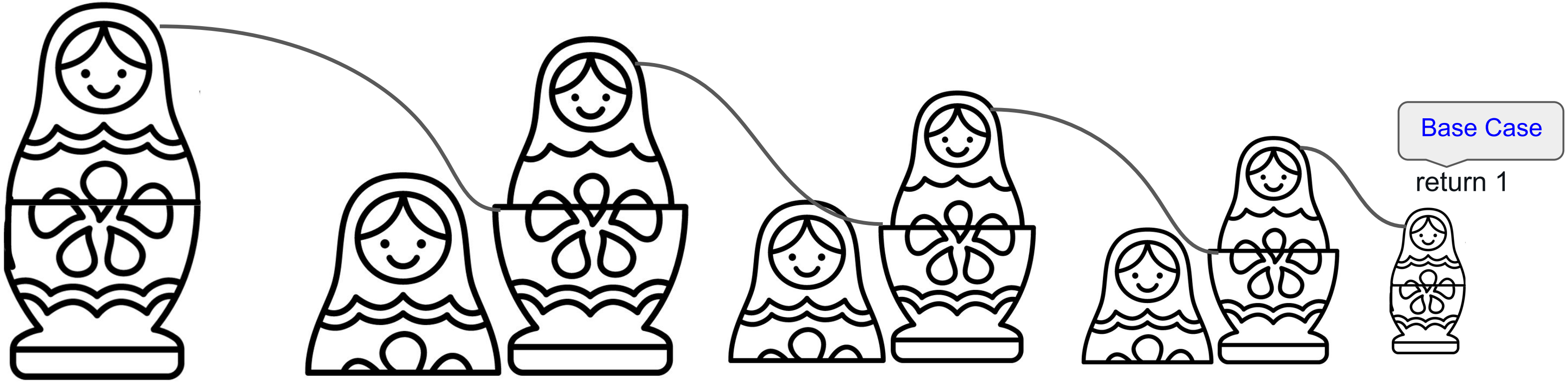
Brainstorm



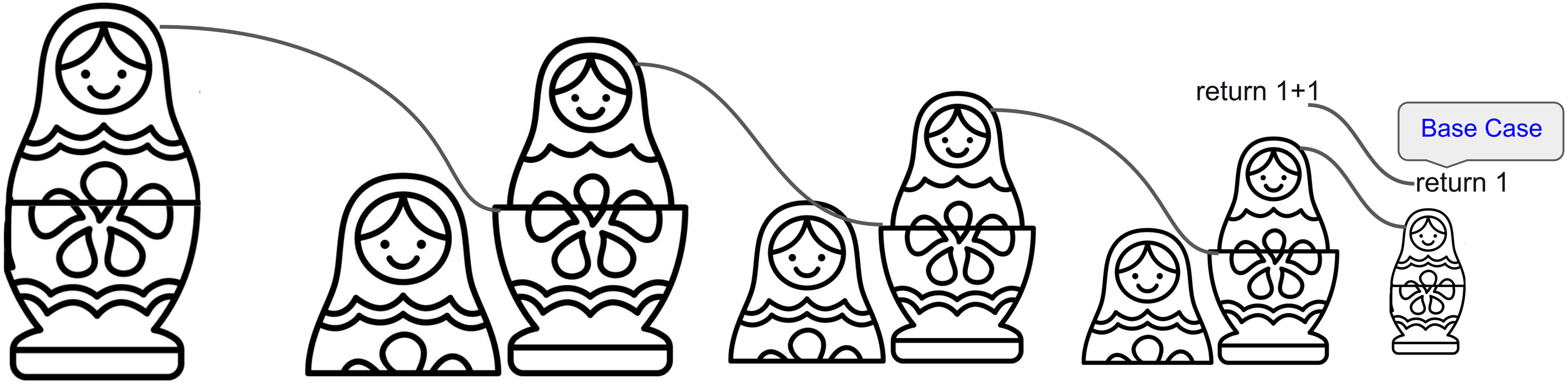
Brainstorm



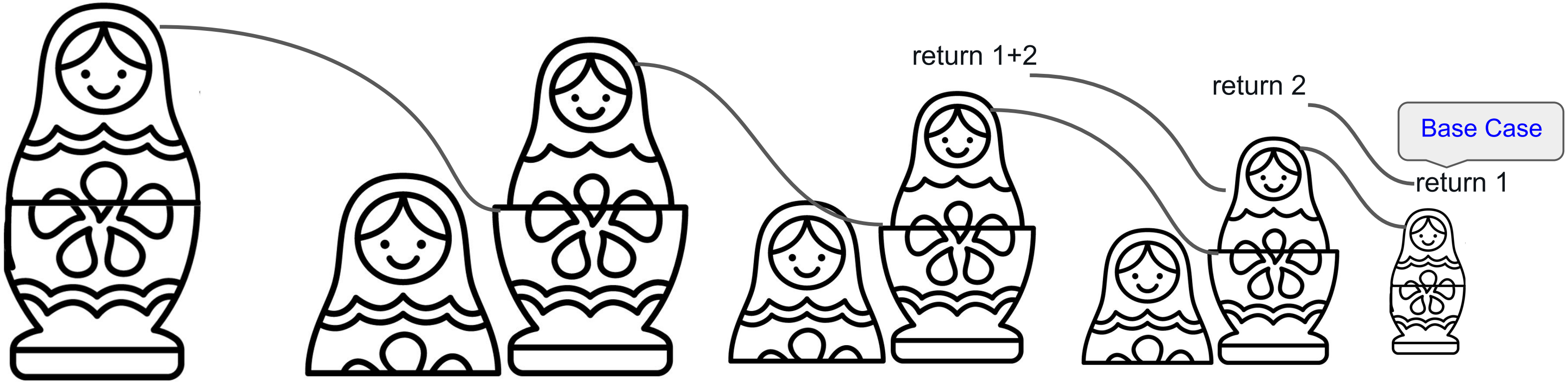
Brainstorm



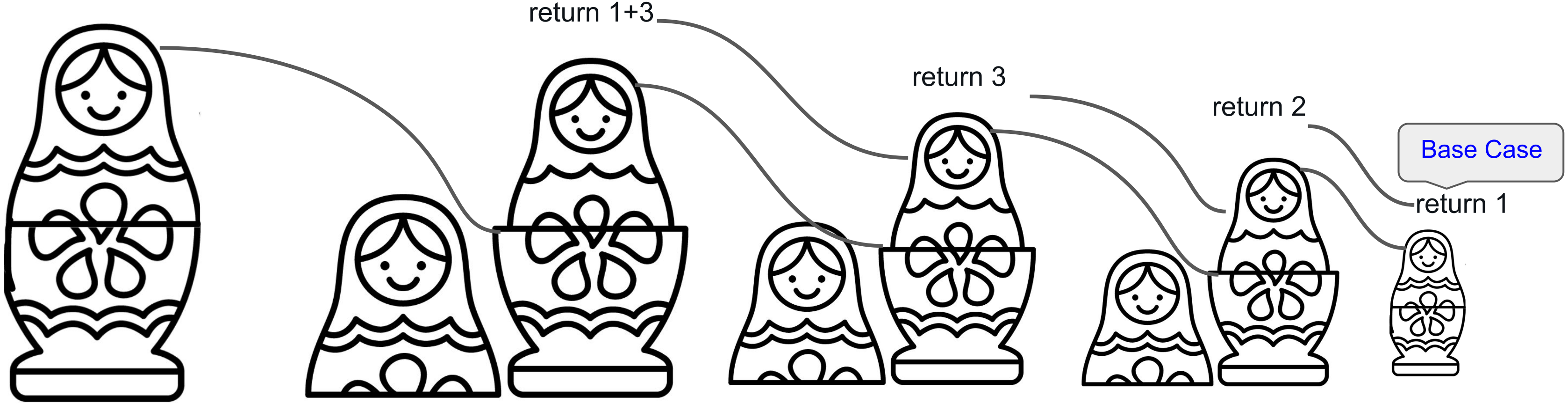
Brainstorm



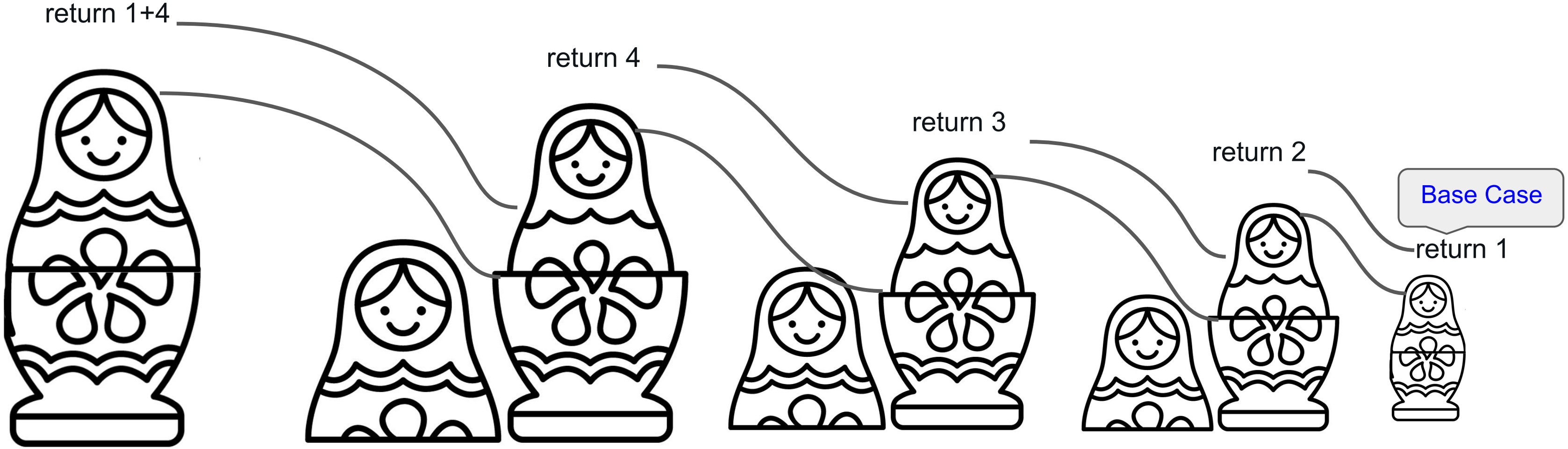
Brainstorm



Brainstorm



Brainstorm



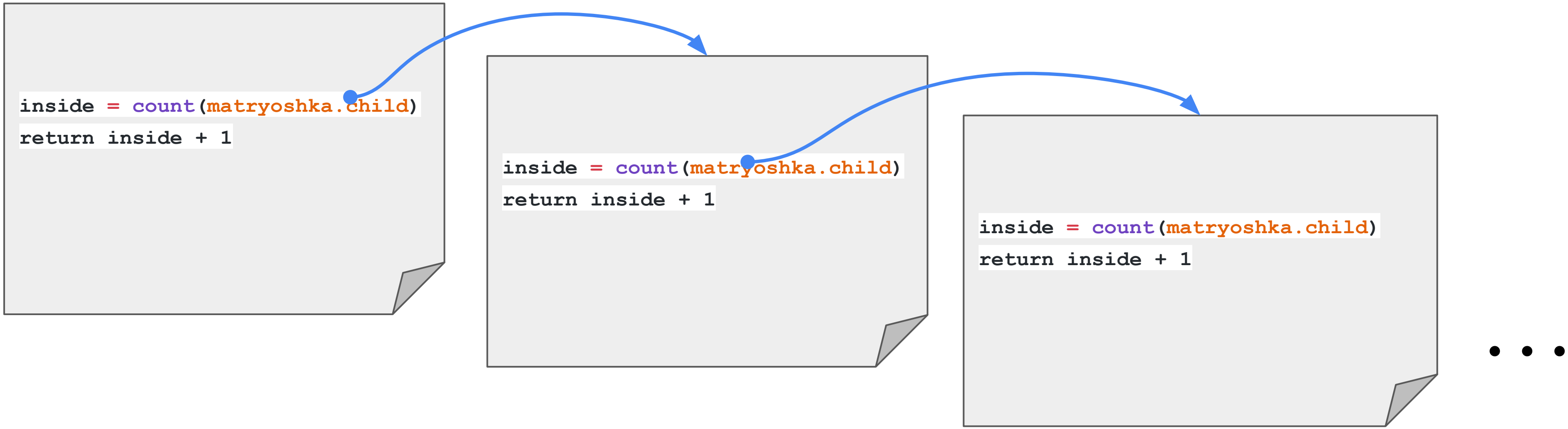
Can we simulate this with code?

- Since we are doing the same thing again and again let's use one function to do the task

Does this work?

```
def count(matryoshka):

    inside = count(matryoshka.child)

    return inside + 1
```

What happens?



```
inside = count(matryoshka.child)
return inside + 1
```

```
inside = count(matryoshka.child)
return inside + 1
```

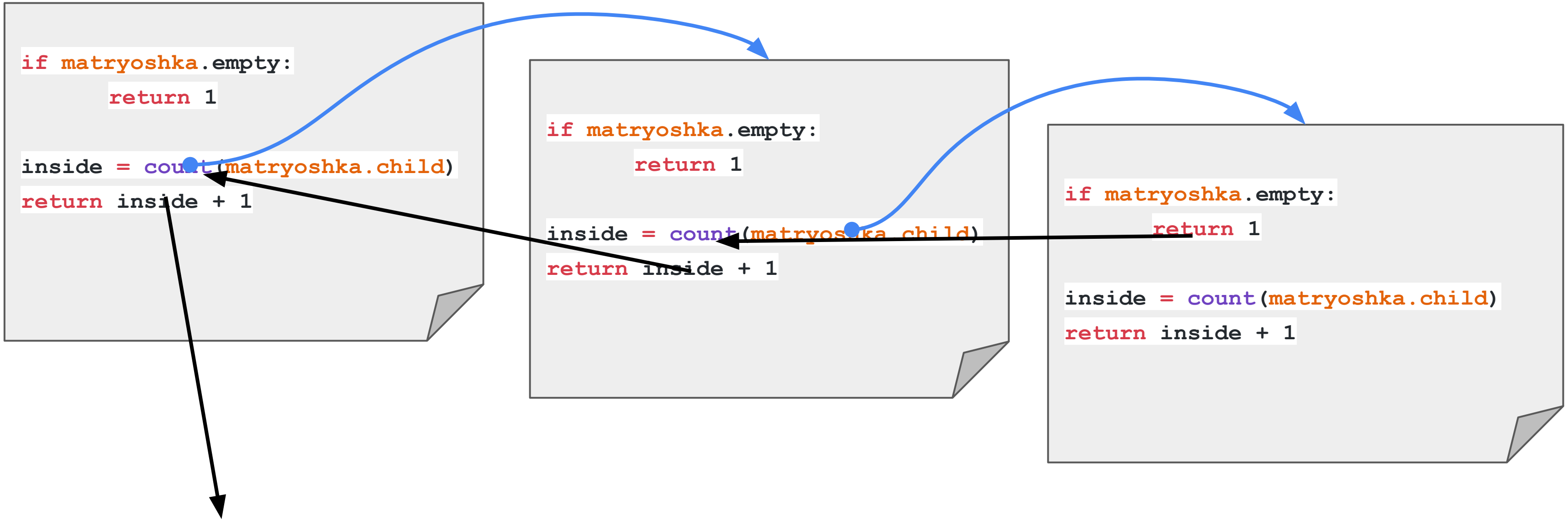
```
inside = count(matryoshka.child)
return inside + 1
```

We will never get to return because we are calling the function non stop. What should we do?

How about this?

```
def count(matryoshka):
    if matryoshka.empty:
        return 1

    inside = count(matryoshka.child)

    return inside + 1
```

What happens?



```
if matryoshka.empty:
        return 1

inside = count(matryoshka.child)
return inside + 1
```

```
if matryoshka.empty:
        return 1

inside = count(matryoshka.child)
return inside + 1
```

```
if matryoshka.empty:
        return 1

inside = count(matryoshka.child)
return inside + 1
```

Then now we return the number of people to our first caller

Definitions

Recursion: process in which a function calls itself directly

Basic Concept

The basic concept of recursion is that a problem can be solved intuitively and much easily if it is represented in one or more smaller versions.

What was the problem in the previous real life example?

What was the subproblem?

?

Basic components of recursion

- Base case The condition that signals when the function should stop and return the final state.
- Recurrence relation Reduces all cases towards base case. The section where the function calls itself with modified inputs and state
- State An identifier that fully locate which subproblem we are dealing with currently

What were the Base Case, Recursive Relation and The State in the previous example?

State

```
def count(matryoshka):
    if matryoshka.empty:
        return 1

    inside = count(matryoshka.child)

    return inside + 1
```

Base Case

Recurrence Relation

Base case

- are known and simplest cases of the problem
- is a terminating scenario
- initiate the process of returning to the original call function (or problem)

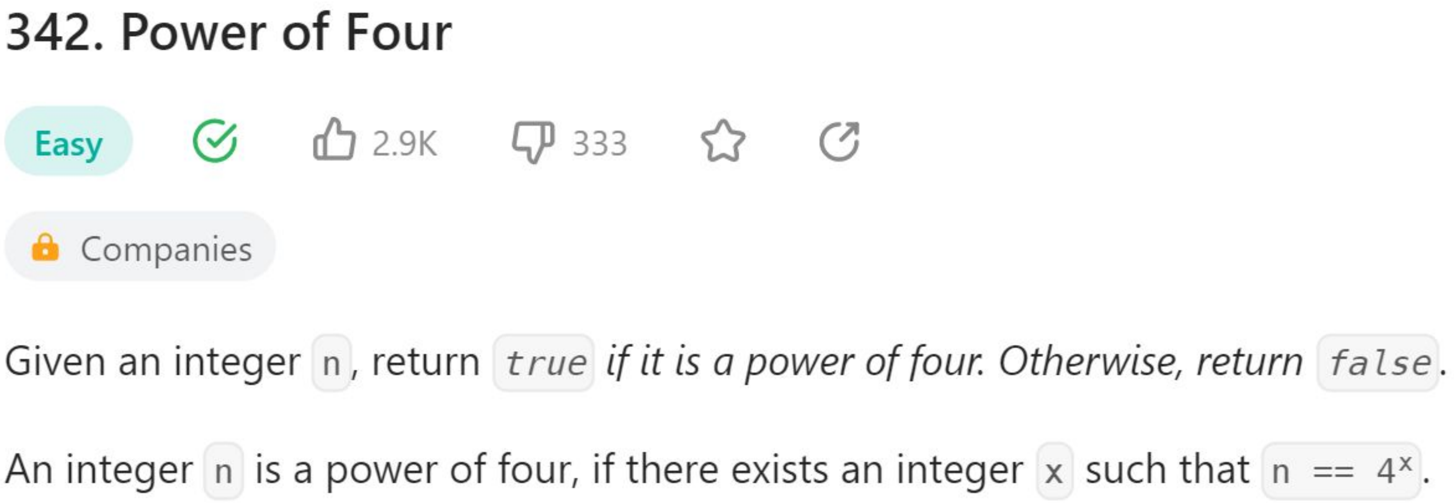
Recurrence Relation

- should be a breakdown of the current problem into smaller problems
- Is a set of cases reduces the current case towards base case
- It aggregates the result from recursive calls and returns the answer we have so far to caller

State

- Identifies the current subproblem completely
- Helps locate which subproblem we are dealing with currently
- Changes in state should lead towards base cases

Sample Question



342. Power of Four

Easy ✓ 👍 2.9K 👎 333 ☆

🔒 Companies

Given an integer `n`, return `true` *if it is a power of four. Otherwise, return* `false`.

An integer `n` is a power of four, if there exists an integer `x` such that $n == 4^x$.

Solution

Q1. What happens if we continue to divide a number which is a power of four by four?

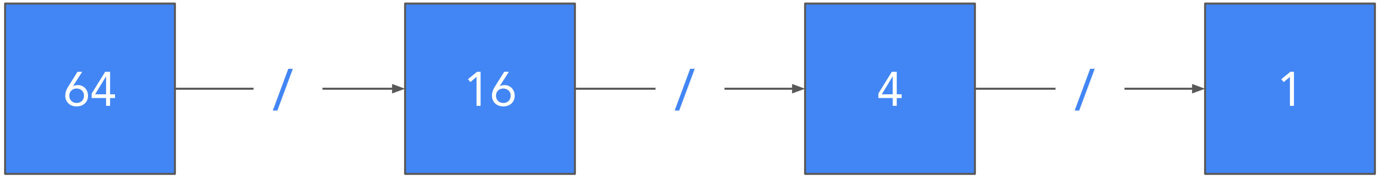
- The final result will be 1

Q2. What if the number is not power of four?

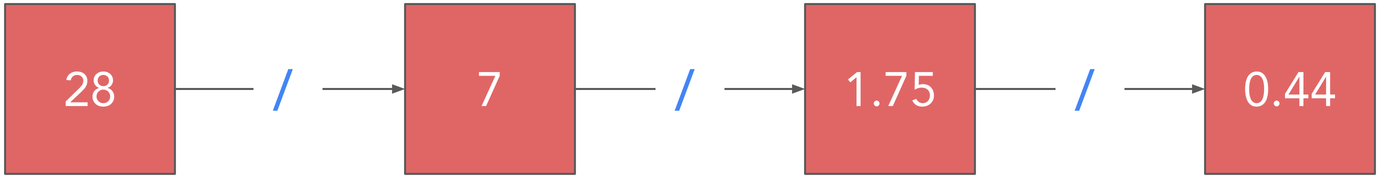
- Their number will be below 1 eventually

From the above questions we can see that we have two base cases and the state is the number that is divided by four every time we have recursive call.

Dividing a power of four by four



Dividing a random number by four



Implementation

```python
def isPowerOfFour(self, n: int) -> bool:
    # basecases
    if n == 1: return True
    if n < 1: return False

    # function calling itself with change in state
    return self.isPowerOfFour(n / 4)
```

Pair Programming

Fibonacci Number

What is the base case and recurrence relation for the above problem?

Base Case

- If n == 1, return 1
- If n == 0, return 0

State

- n

Recurrence Relation

- $f(n) = f(\mathrm{n} - 1) + f(n - 2)$

Implementation

```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    return fib(n-1) + fib(n-2)
```

How does recursion work?

How does the recursion work?

- The system uses some ordering to execute the calls
- The ordering is done based on **Last In First Out (depth first)** order
- It uses stack

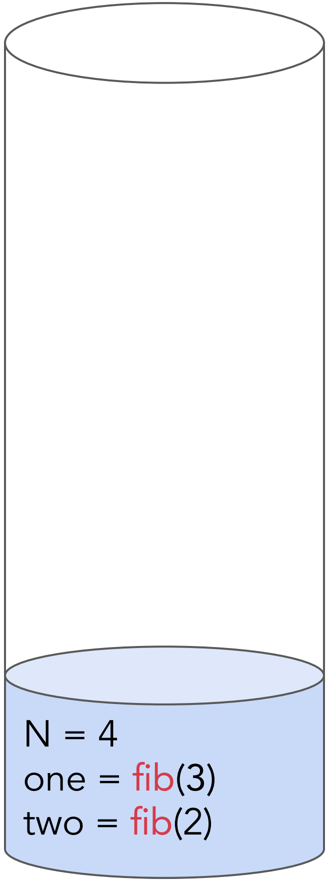
Stack.top -> currently executing

Stack.push -> calling function

Stack.pop -> returning a value

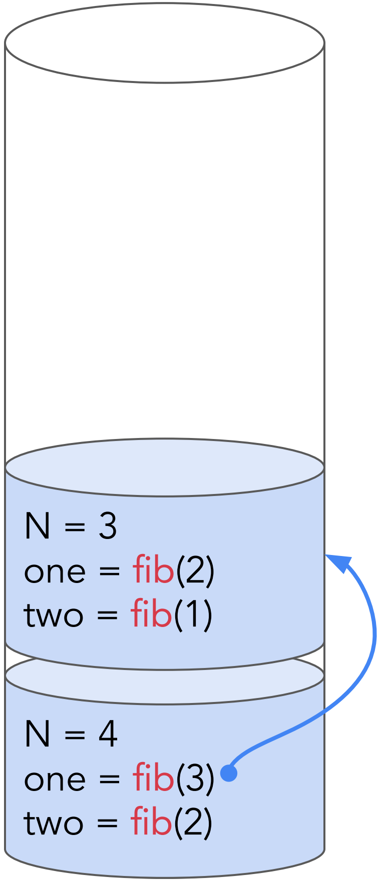
Example Using Call Stack

```
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```

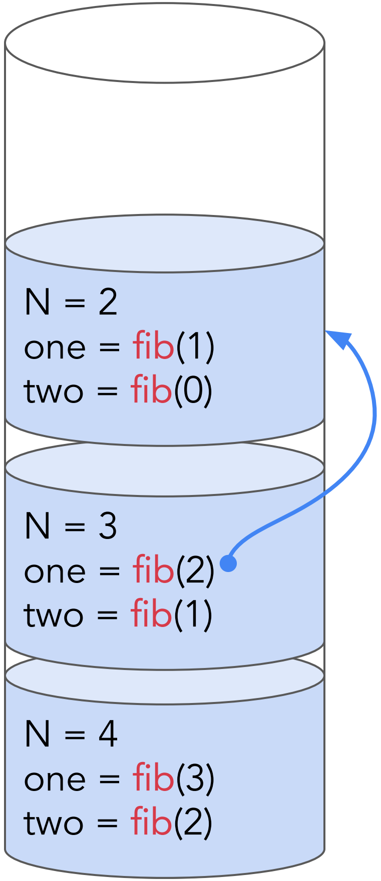


N = 4
one = fib(3)
two = fib(2)

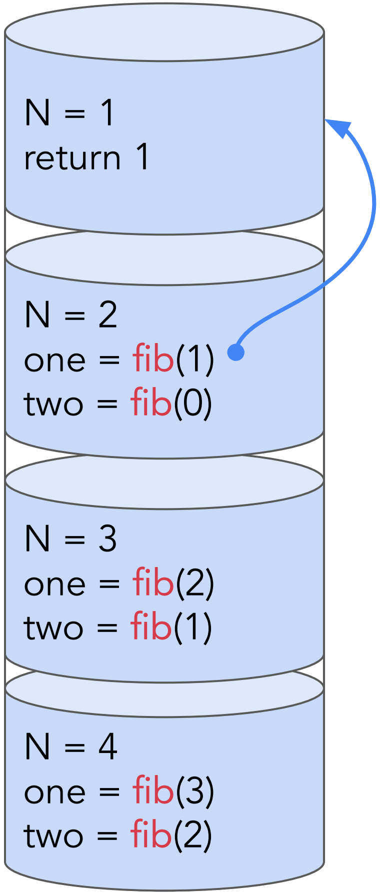
```
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



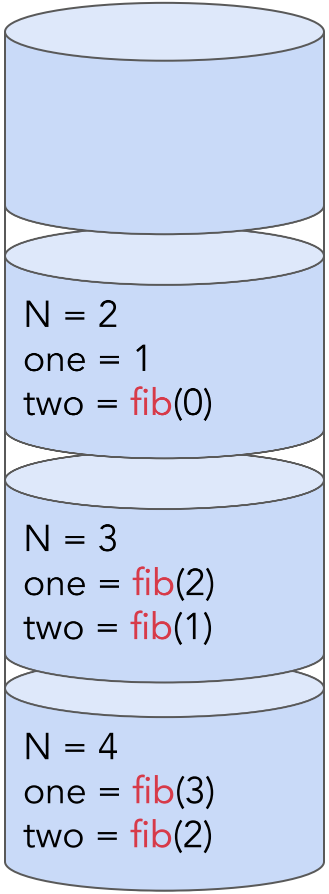
```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



```
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



```
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```

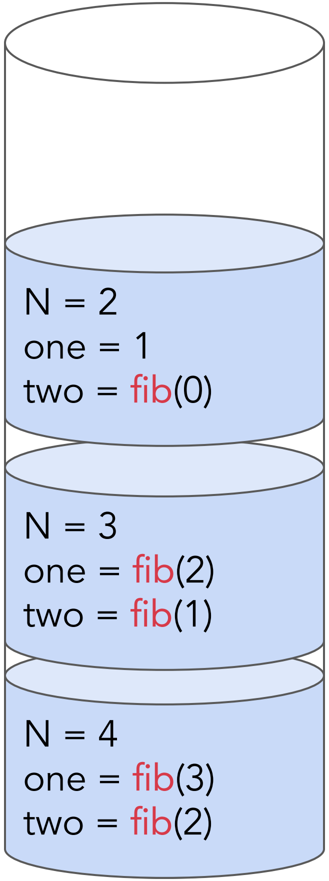


N = 2
one = 1
two = fib(0)

N = 3
one = fib(2)
two = fib(1)

N = 4
one = fib(3)
two = fib(2)

```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```

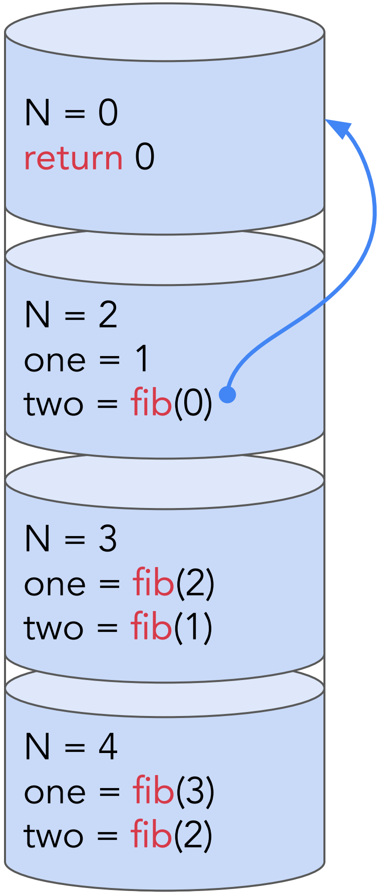


N = 2
one = 1
two = fib(0)

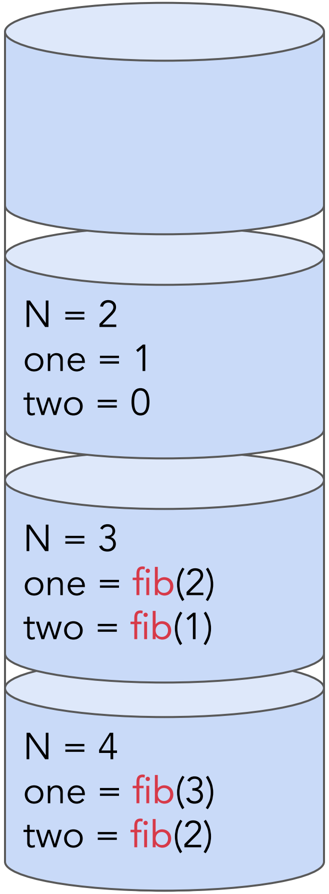
N = 3
one = fib(2)
two = fib(1)

N = 4
one = fib(3)
two = fib(2)

```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



```
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```

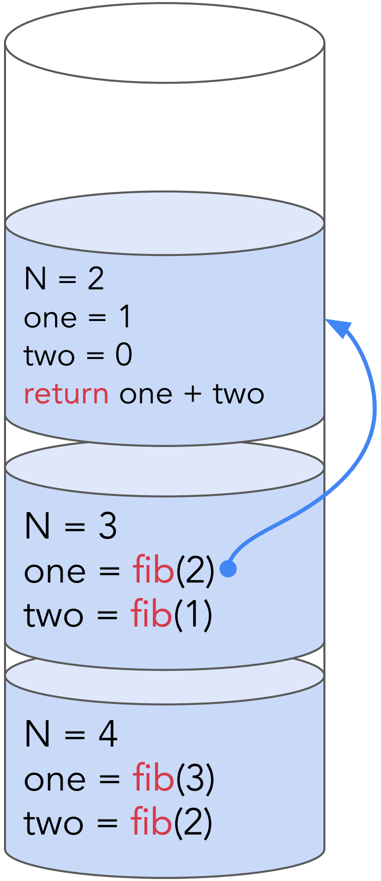


N = 2
one = 1
two = 0

N = 3
one = fib(2)
two = fib(1)

N = 4
one = fib(3)
two = fib(2)

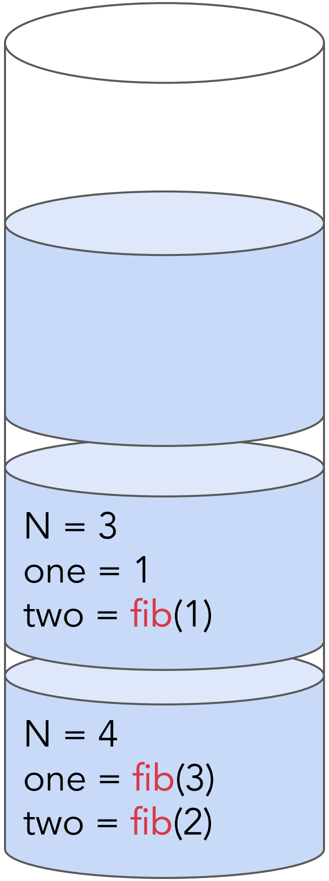
```
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```


N = 2
one = 1
two = 0
return one + two

N = 3
one = fib(2)
two = fib(1)

N = 4
one = fib(3)
two = fib(2)

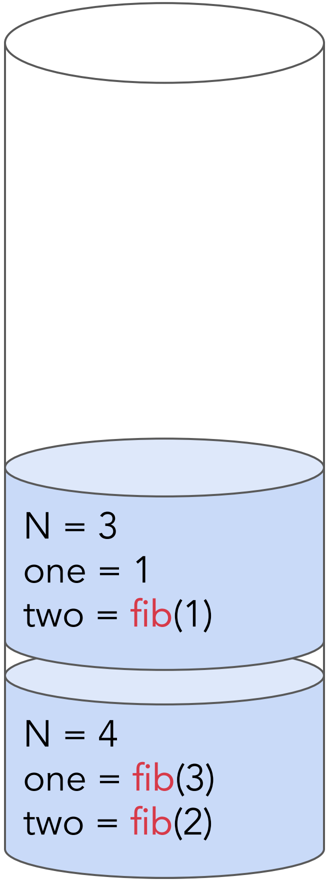
```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



N = 3
one = 1
two = fib(1)

N = 4
one = fib(3)
two = fib(2)

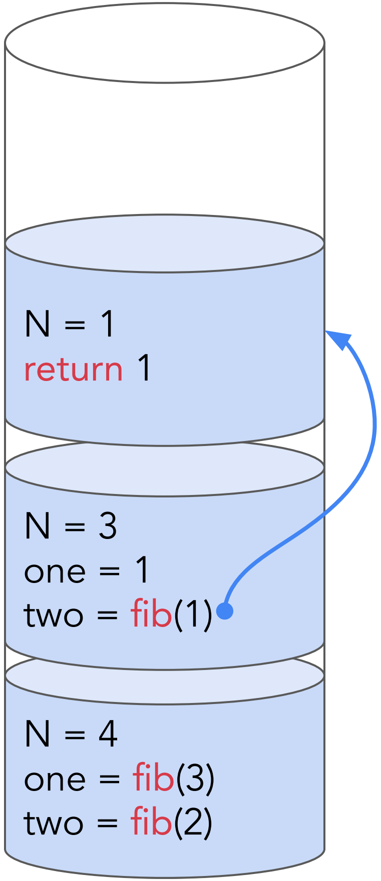
```
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



N = 3
one = 1
two = fib(1)

N = 4
one = fib(3)
two = fib(2)

```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```

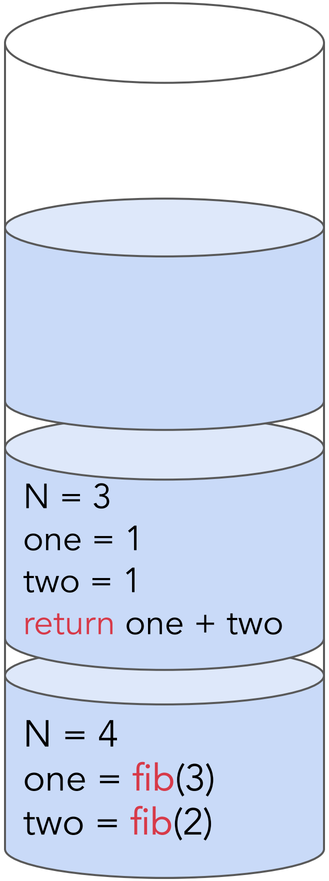


N = 1
return 1

N = 3
one = 1
two = fib(1)

N = 4
one = fib(3)
two = fib(2)

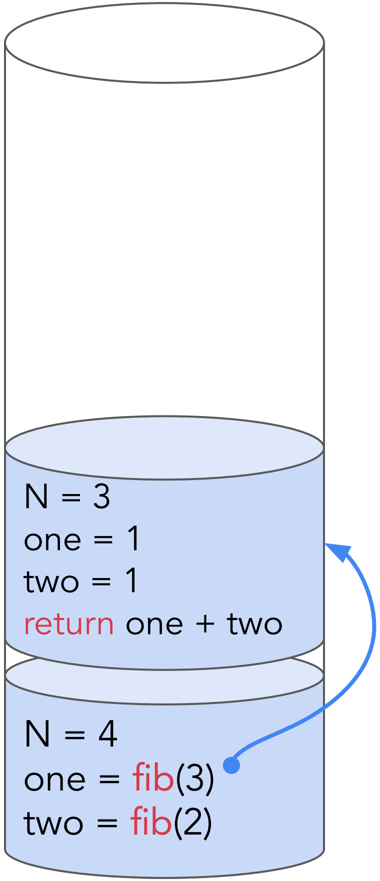
```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



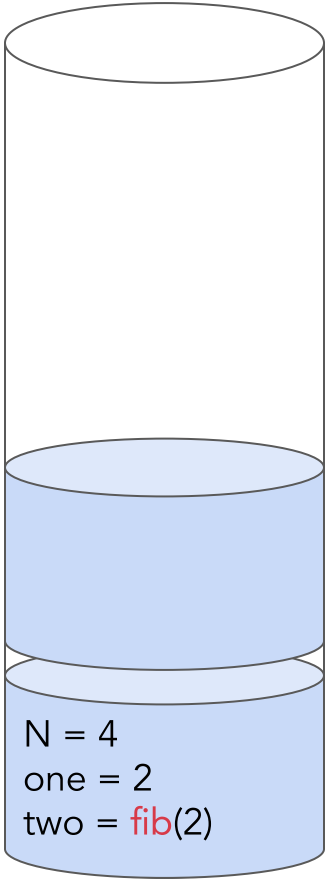
N = 3
one = 1
two = 1
return one + two

N = 4
one = fib(3)
two = fib(2)

```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```

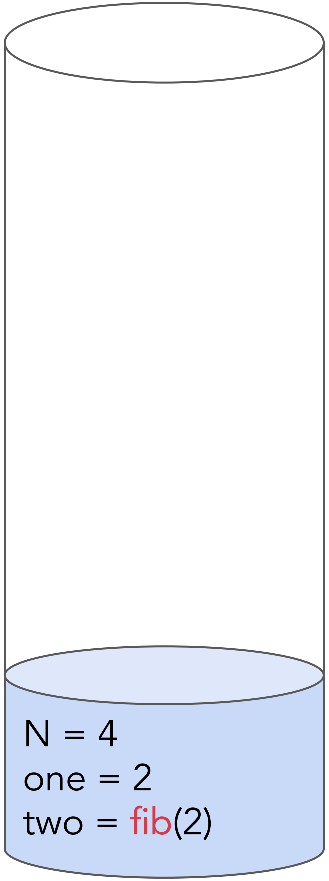


```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



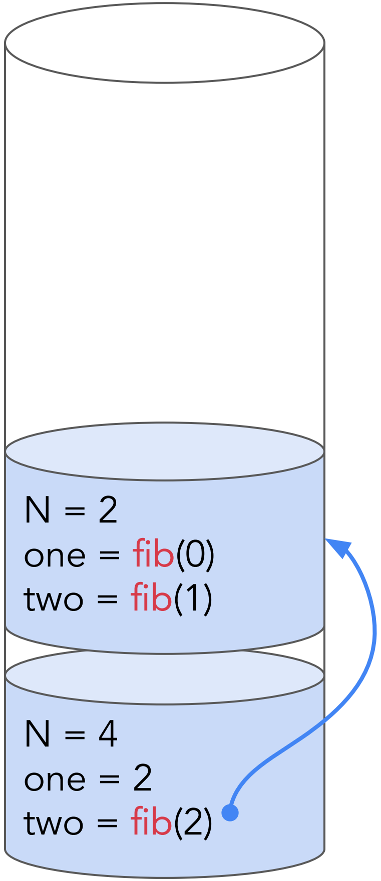
N = 4
one = 2
two = fib(2)

```
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



N = 4
one = 2
two = fib(2)

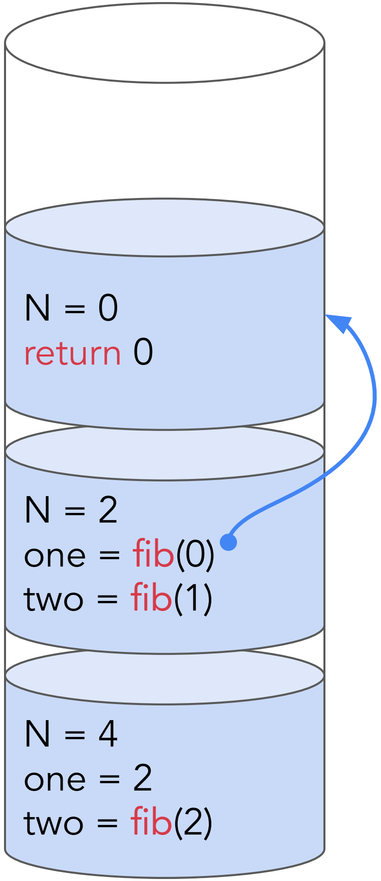
```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



N = 2
one = fib(0)
two = fib(1)

N = 4
one = 2
two = fib(2)

```
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```

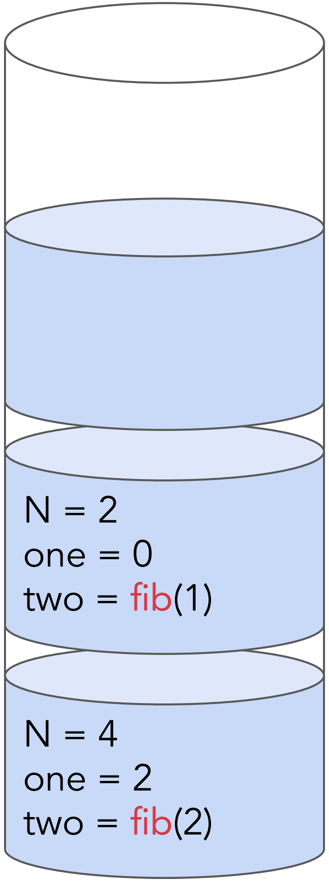


N = 0
return 0

N = 2
one = fib(0)
two = fib(1)

N = 4
one = 2
two = fib(2)

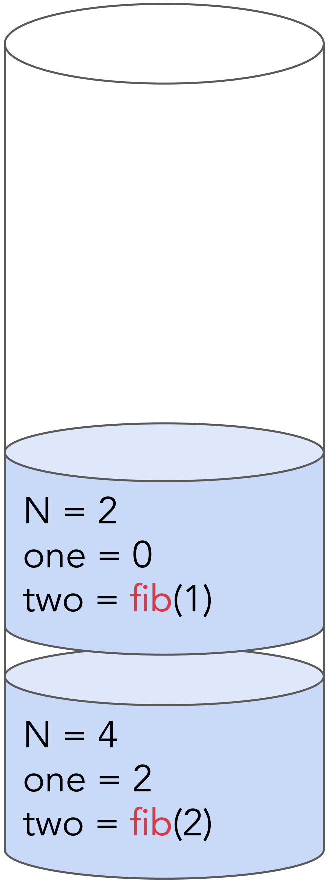
```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



N = 2
one = 0
two = fib(1)

N = 4
one = 2
two = fib(2)

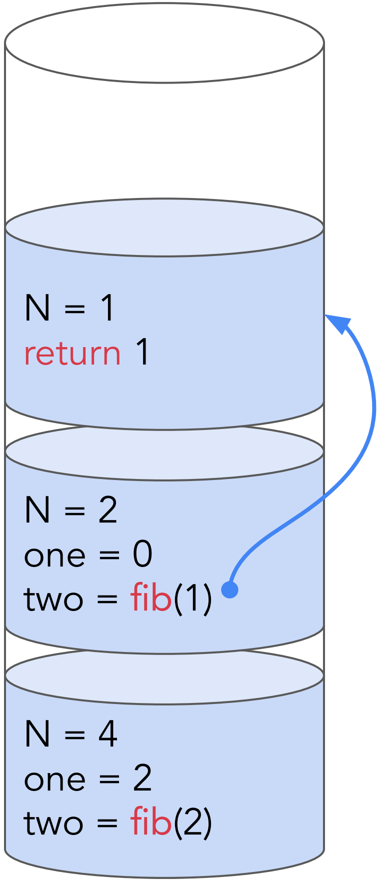
```
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



N = 2
one = 0
two = fib(1)

N = 4
one = 2
two = fib(2)

```
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```

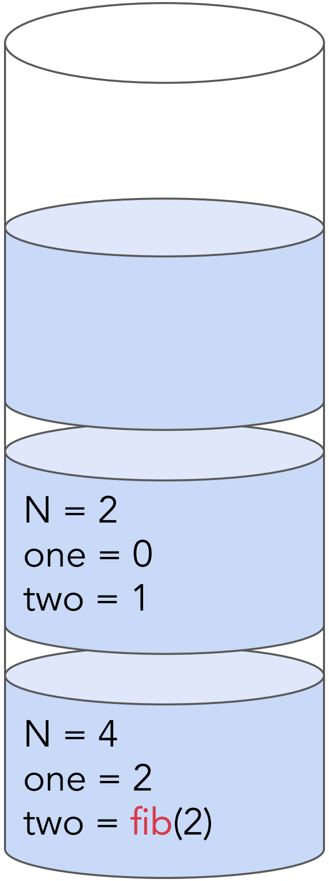


N = 1
return 1

N = 2
one = 0
two = fib(1)

N = 4
one = 2
two = fib(2)

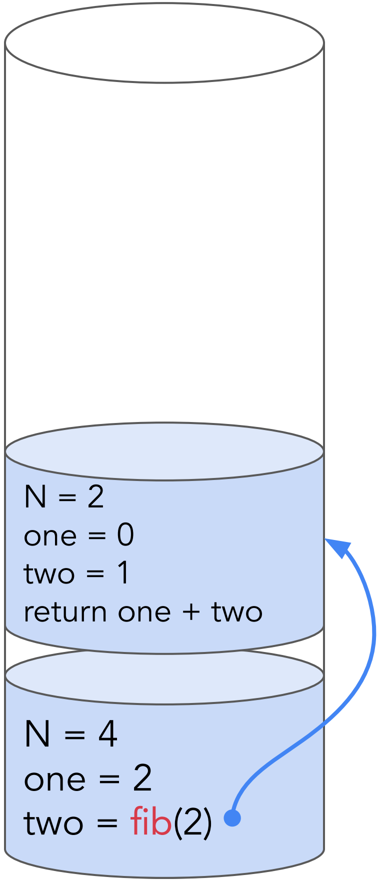
```
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



N = 2
one = 0
two = 1

N = 4
one = 2
two = fib(2)

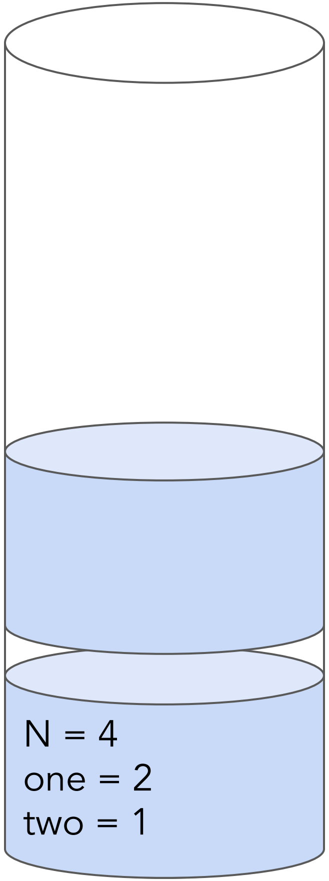
```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



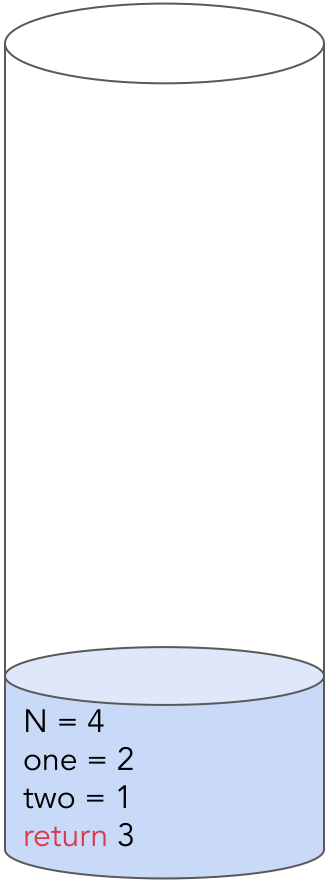
N = 2
one = 0
two = 1
return one + two

N = 4
one = 2
two = fib(2)

```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```



```python
def fib(n: int) -> int:
    if n == 1:
        return 1

    if n == 0:
        return 0

    one = fib(n-1)
    two = fib(n-2)

    return one + two
```

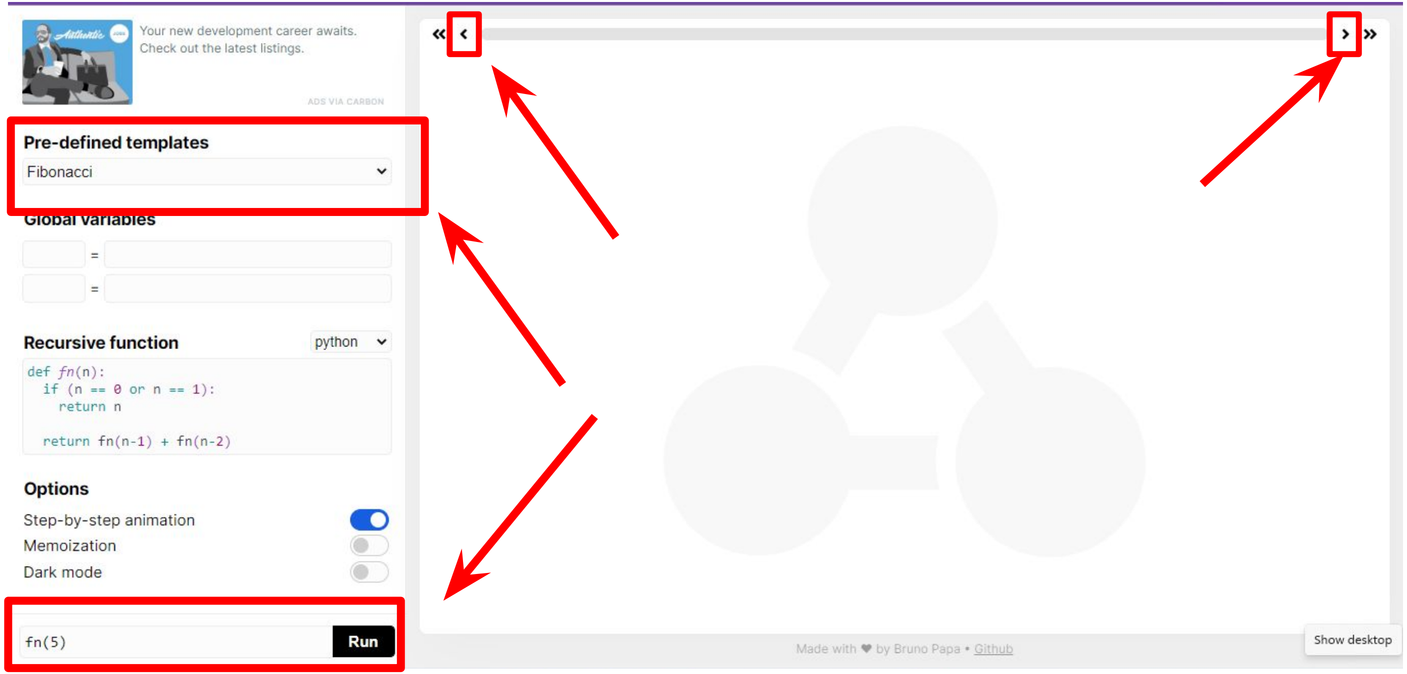


N = 4
one = 2
two = 1
return 3

Using Execution Tree

Step by Step Simulation

Go to this LINK



Pre-defined templates

Fibonacci

Global variables

Recursive function python

```
def fn(n):
  if (n == 0 or n == 1):
    return n

  return fn(n-1) + fn(n-2)
```

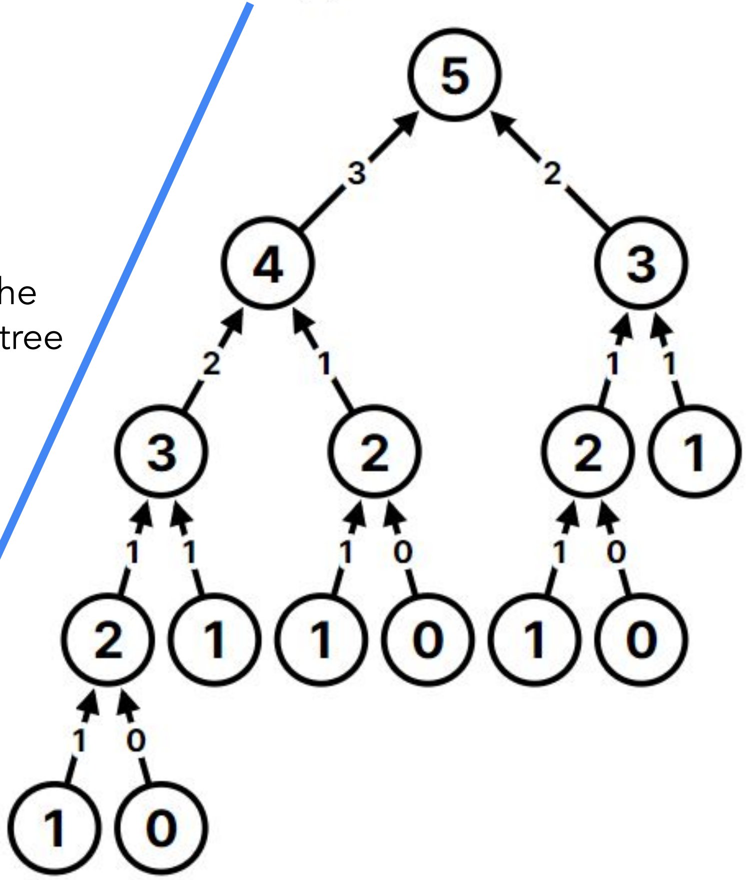
Options

Step-by-step animation ☑
Memoization ☐
Dark mode ☐

fn(5) Run

Made with ❤ by Bruno Papa • Github

fn(5) returns 5

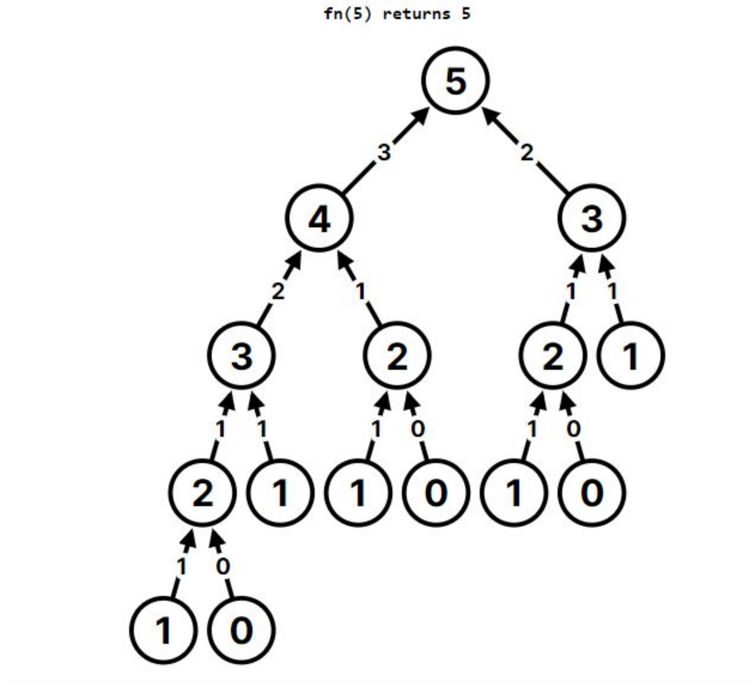


Depth of the execution tree

Num of nodes approximately : $2^{n+1} - 1$

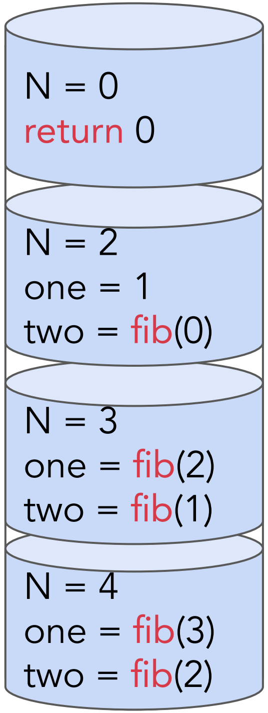
Debugging using the execution tree

- Simulate the code using the execution tree using pen and paper by yourself.
- Simulate your code with recursion simulation sites
- Compare the results


fn(5) returns 5

Debugging using the call stack and print statements

- Simulate the code using the call stack by yourself using pen and paper
- Use print statements to see if your code matches the simulation



N = 0
return 0

N = 2
one = 1
two = fib(0)

N = 3
one = fib(2)
two = fib(1)

N = 4
one = fib(3)
two = fib(2)

```
def recursive_fn(state):
    print(“state: “, state)

    .
    .
    .

    print(“result: “, result)
    return result
```

Complexity Analysis

Time Complexity
(Any suggestion?)

There are two parts

- Number of recursive calls
- The complexity of the function

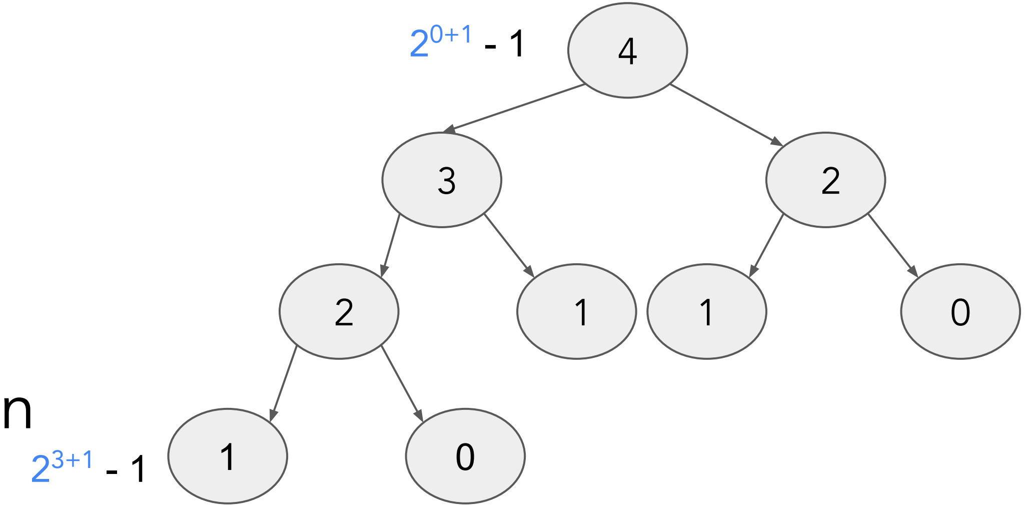
How do we find the number of calls?

- Draw the execution tree
- Execution tree is a tree that describes the flow of execution for the recursive function
- A node represents a call
- The total number of nodes is your final answer

Fib(4)

Facts:

- Two branching on each node
- Height of the tree is the equal to n
- In a full binary tree with n height, we have $2^{n+1} - 1$ nodes



In general

- Let
  - b = number of branches on each node
  - d = depth of the tree
- Total number of nodes = $b^{d+1} - 1$

- After only taking the dominant term

$$(\text{Num of branching})^{\text{depth of the tree}}$$

What is the complexity of the function?

- This is plain time and space complexity analysis of the inner working of the function
- Are we doing any costly operations in the function?

In Summary

$$\underbrace{O(\text{function})}_{\text{Cost of running function}} \bullet \underbrace{O((\text{Num of branching})^{\text{depth of the tree}})}_{\text{Number of recursive calls}}$$

Space Complexity

Before we discuss the space complexity, let's consider these questions

- How does the computer run recursion function?
- How does it know when to return and combine?

It uses call stack

What is the space complexity of the call stack?

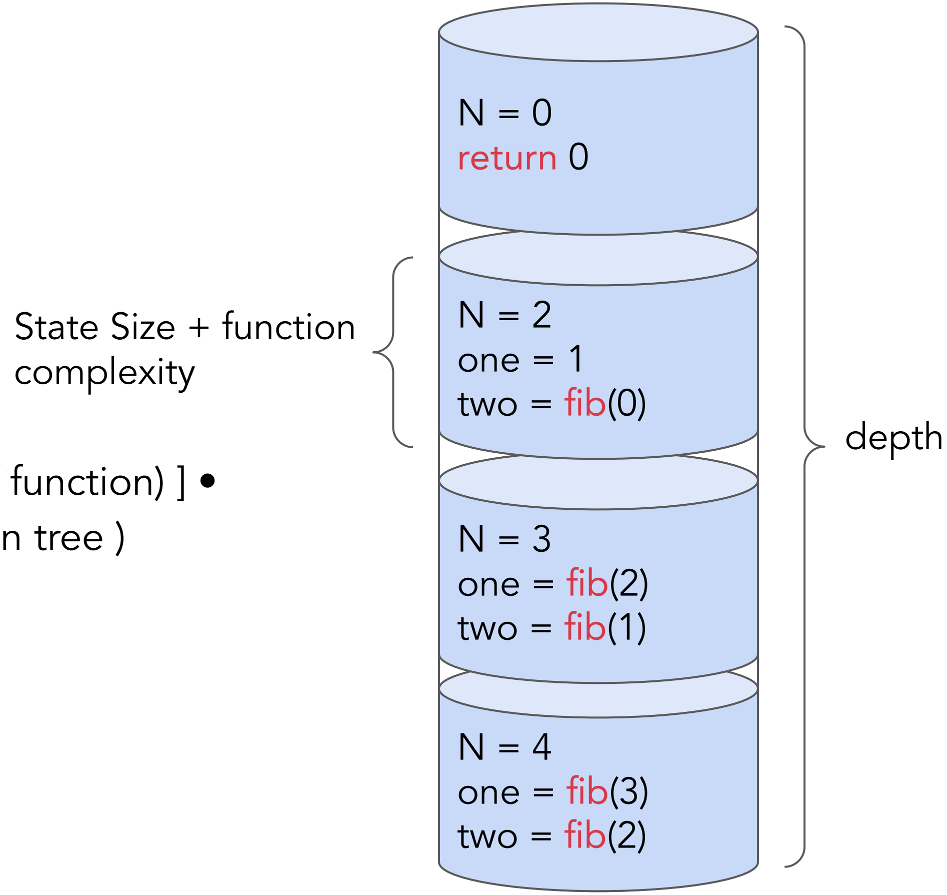
Maximum size of the stack which is maximum depth of the execution tree

Did we forget anything?

What about the states and space complexity of the function?

In Summary

[O(states' size) + O(space complexity of function)] •
O(maximum depth of the execution tree)



Checklist (Ask your instructor, if anything is missing)

- ❑ You have to be very comfortable with writing out the base case and recurrence relation of recursive function
- ❑ You have to be very comfortable with drawing execution tree of a recursion function
- ❑ Understanding the execution order using the execution tree
- ❑ Comfortably analyze the time and space complexity of a recursive function

Recursion versus Iteration

Anything that can be done recursively can be done iteratively

- Iteration is faster and more space-efficient than recursion.
- Recursion has function call overhead
- Iterative programs are easy to debug

So why do we even need recursion?

- If the solution to the given problem is achieved by breaking it down into its subproblems, the recursive approach is a more intuitive and natural way to solve it.

Recognizing in Questions

- Look for patterns in the question that suggest that the same operation is being applied to smaller versions of the same problem.
- Look for base cases or stopping conditions that define when the recursion should end.
- Searching problems with branching conditions

Things to pay attention

Stack Overflow ?

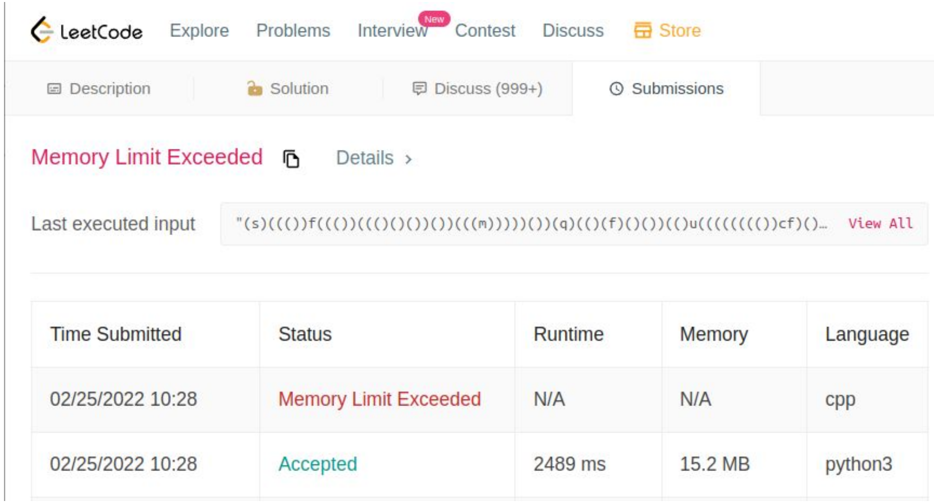
Occurs when the program uses up the memory assigned to it in the call stack

In the case of recursion we will mainly deal with two types:

- Memory Limit Exceeded
- Maximum Recursion Depth Exceeded.

Memory Limit Exceeded

is a type of stack overflow error that occurs when the parameters of our function take too much space.



Memory Limit Exceeded Details ›

Last executed input `"(s)((())f((())((()()())())(((m)))))())(q)(()(f)()())(()u((((((((())cf)()…` View All

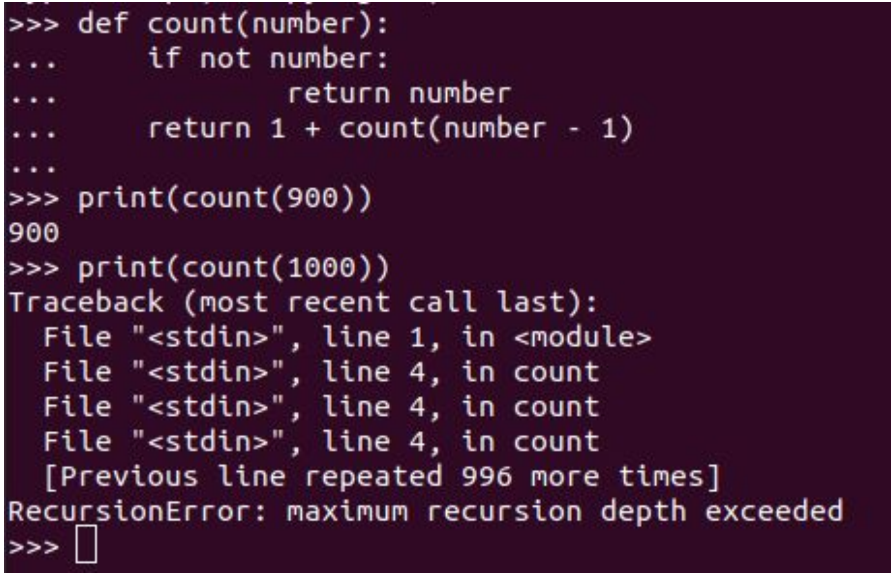
Time Submitted	Status	Runtime	Memory	Language
02/25/2022 10:28	Memory Limit Exceeded	N/A	N/A	cpp
02/25/2022 10:28	Accepted	2489 ms	15.2 MB	python3

Maximum Recursion Depth Exceeded

is a type of stack overflow error that occurs when the number of recursive calls being stored in the call stack is greater than allowed.

Different programming languages have different maximum recursion Depth sizes:

- Python => 1000
- Javascript => 10,000
- Java => around 10,000
- C++ =>No Limit



```
>>> def count(number):
...     if not number:
...             return number
...     return 1 + count(number - 1)
...
>>> print(count(900))
900
>>> print(count(1000))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in count
  File "<stdin>", line 4, in count
  File "<stdin>", line 4, in count
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
>>> 
```

A function with missing base case

```
def fib(n: int) -> int:
    if n == 1:
        return 1
```

What happens when n == 0?

```
    return fib(n-1) +
fib(n-2)
```

Wrong recurrence relation

```python
def numOfPeopleInQueue(n: int) -> int:
    if n == 1:
        return 1

    prev = numOfPeopleInQueue(n + 1) + 1
    return prev
```

Using list as state

```
def divideInBuckets(i, picked: List) -> int:
    if i >= len(arr):
        return 0

    notPick = divideInBuckets(i+1,picked)
    picked.append(arr[i])
    pick = divideInBuckets(i+1,picked)

    return pick + notPick
```

Passed by reference

Practice Questions

Reverse String

Count Good Numbers

Pascal’s Triangle II

Merge Two Sorted List

Decode String

Predict the Winner

Power of three

Power of four

Resources

- Leetcode Recursion I Card
- Leetcode Recursion II Card

Quote

“In order to understand recursion, one must first understand recursion.”

Unknown