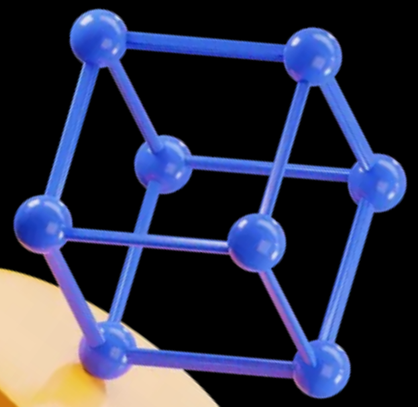
Sohum Sheth, Dean Richardson, Griffin Charlson

HKD MEDIUM SECTION #3

Functions and Recursion

October 28th, 2026



Attendance and Announcements



- 🍕Pizza at the end as always :)
- MOCK INTERVIEWS
  - Nov 2nd-7th
- Happy spooky month

https://calendly.com/hckrdevils/mock-interviews





What is a function?

In mathematics, you are probably familiar with some basic functions, like addition, multiplication, exponentiation, etc...

Each of these math functions take some **inputs** and return an **output**

For example:

1 + 3 = 4



1,3: Inputs
+: Function
4: Output

How can we represent a function in code?

To use a function in code, we need to do two steps

1. Declare the function
    a. This is where we describe what the function will actually do (or, how does it calculate output given any input)

2. Calling the function
    a. This is where we provide some inputs to figure out an output

Example: Addition

Declaring the function:

- Inputs: a and b
- Output: a + b

Python:

```python
def add(a, b):
    return a + b
```

Java:

```java
public static int add(int a, int b) {
    return a + b;
}
```

Example: Addition

Python:

```python
def add(a, b):
    return a + b
```

Java:

```java
public static int add(int a, int b) {
    return a + b;
}
```

After we declare a function one time, we can call it infinite times.

Calling a function:

Python:

```python
print(add(3, 4))
```

Java:

```java
System.out.println(add(3, 4));
```

Prints out 7

What if we change our function slightly?

```python
def double(x):
    return x * 2
```

What does this function do?

It takes an input x, doubles it, and returns it

```python
def double(x):
    return double(x * 2)
```

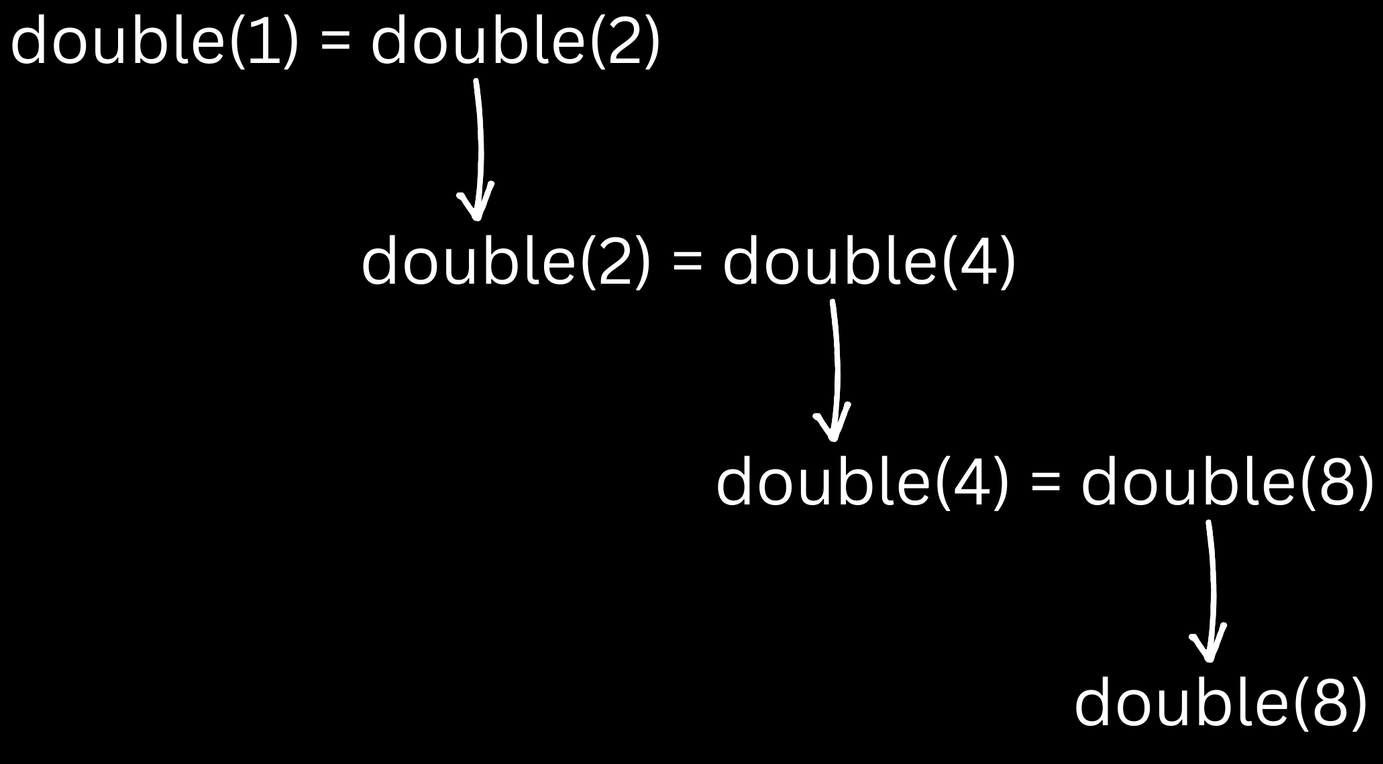
What does this function do?

This function is tricker to understand because it calls itself
This is recursion

Understanding Recursion

```
def double(x):
    return double(x * 2)
```

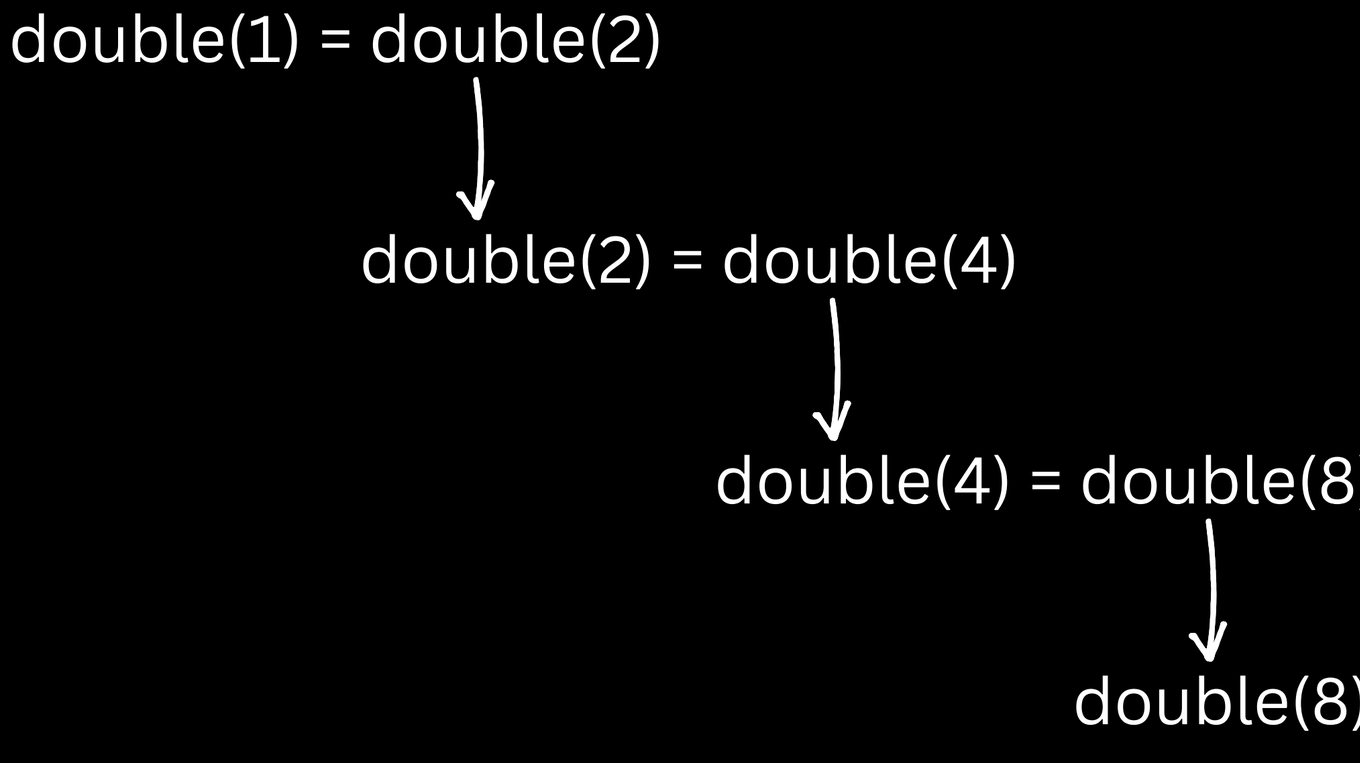
To understand this function, let’s trace the input when we call double(1)



Understanding Recursion

```
def double(x):
    return double(x * 2)
```

This is an infinite function that keeps doubling the input



Understanding Recursion

```
def double(x):
    return double(x * 2)
```

This is an infinite function that keeps doubling the input

Every single recursive function will call itself infinitely, leading to this:

```
RecursionError: maximum recursion depth exceeded
```

How do we fix this?

We need to add a condition that ends the infinite recursive loop

```
def double(x):
    return double(x * 2)
```



```
def double(x):
    if x > 1000:
        return x
    return double(x * 2)
```

Now, instead of endlessly doubling, this function will take an input x, and double it until it exceeds 1000. It will then return that value

Terminology

```
def double(x):
    if x > 1000:
        return x
    return double(x * 2)
```

Base Case: The condition that ends the recursive loop

Recursive Case: The condition that continues the recursive loop (always calls itself)

Why Recursion

With recursion, we can write simpler solutions to complex problems. This is because instead of solving an entire problem, we solve one repeatable step.

All we have to define is

- What are we repeatedly doing (Recursive Step)
- When do we stop (Base Case)

Let’s try!

Problem

326. Power of Three

Solved

Easy | Topics | Companies

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

An integer `n` is a power of three, if there exists an integer `x` such that `n == 3^x`.

Example 1:

> **Input:** n = 27
> **Output:** true
> **Explanation:** $27 = 3^3$

Example 2:

> **Input:** n = 0
> **Output:** false
> **Explanation:** There is no x where $3^x = 0$.

Note that recursion isn't necessary for this problem, but you should absolutely practice using it!

Solution

Example 1:

> **Input:** n = 27
> **Output:** true
> **Explanation:** 27 = 3^3

If we look at this test case, we can learn what the problem is actually asking

27 is a valid power of three, because it can be expressed as 3^3

If we extrapolate this, we can define a power of three as any number that can be represented as 3^x (where x is some positive integer)

Solution

So, how can we easily figure out if a number can be expressed as 3^x ?

Let’s look at 27 again:

$$27 = 3^3$$

If we keep dividing both sides by 3, we get:

$9 = 3^2$

$3 = 3^1$

$1 = 3^0$

Solution

So, any number that can be expressed as: $n = 3^x$ (meaning it is a power of 3)

will always divide down to

3^0, or 1

To turn this into a recursive solution, we need to identify our recursive case and base case

Solution

To turn this into a recursive solution, we need to identify our recursive case and base case

Base Case: If any of these happen, we can exit

- Number is negative
  - Powers of 3 must be positive, so we return false
- Number cannot be divided by 3
  - Powers of 3 can always be divided down to 1, meaning we return false
- Number is 1
  - This means we have successfully divided down to 1, so we return true

Recursive Case: Divide by 3 and call the function

Solution

Java

```java
public boolean isPowerOfThree(int n) {
    if (n <= 0) {
        return false;       // powers of 3 are always positive
    }
    if (n == 1) {
        return true;        // 3^0 = 1
    }
    if (n % 3 != 0) {
        return false;       // can't be divided evenly by 3
    }
    return isPowerOfThree(n / 3);
}
```

Python

```python
def isPowerOfThree(n):
    if n <= 0:
        return False
    if n == 1:
        return True
    if n % 3 != 0:
        return False
    return isPowerOfThree(n // 3)
```

red: base cases

blue: recursive cases

Tracing the Solution

```python
def isPowerOfThree(n):
    if n <= 0:
        return False
    if n == 1:
        return True
    if n % 3 != 0:
        return False
    return isPowerOfThree(n // 3)
```

It is always useful to trace a recursive function by using a recursion tree





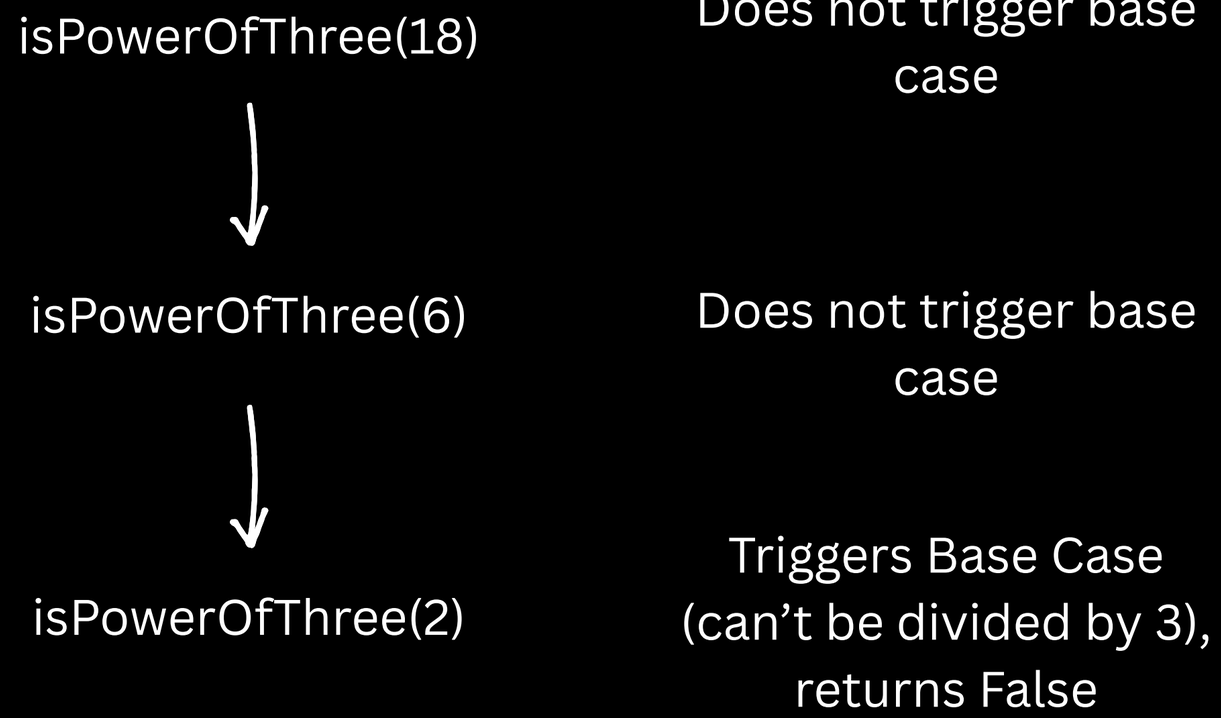


isPowerOfThree(27)	Does not trigger base case
isPowerOfThree(9)	Does not trigger base case
isPowerOfThree(3)	Does not trigger base case
isPowerOfThree(1)	Triggers Base Case, returns True

Tracing the Solution

```python
def isPowerOfThree(n):
    if n <= 0:
        return False
    if n == 1:
        return True
    if n % 3 != 0:
        return False
    return isPowerOfThree(n // 3)
```

It is always useful to trace a recursive function by using a recursion tree



Main Problem

22. Generate Parentheses

Solved

Medium | Topics | Companies

Given `n` pairs of parentheses, write a function to *generate all combinations of well-formed parentheses.*

Example 1:

```
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]
```

Example 2:

```
Input: n = 1
Output: ["()"]
```

Constraints:

- `1 <= n <= 8`

leetcode.com/problems/generate-parentheses/

Test Case

Example 1:

> **Input:** n = 3
> **Output:** ["((()))","(()())","(())()","()(())","()()()"]

n = 3 Means we have to use 3 open parentheses and 3 closed parenthesis

Our goal is forming every single combination of “well formed” parenthesis

“Well Formed”

Example 1:

> **Input:** n = 3
> **Output:** ["((()))","(()())","(())()","()(())","()()()"]

The definition of “well formed” just means that every ‘(’ has a ‘)’ to the right of it.

- ()
- (())
- ((()))
- ()()
- (())()

Examples of Unwell-Formed Parenthesis:

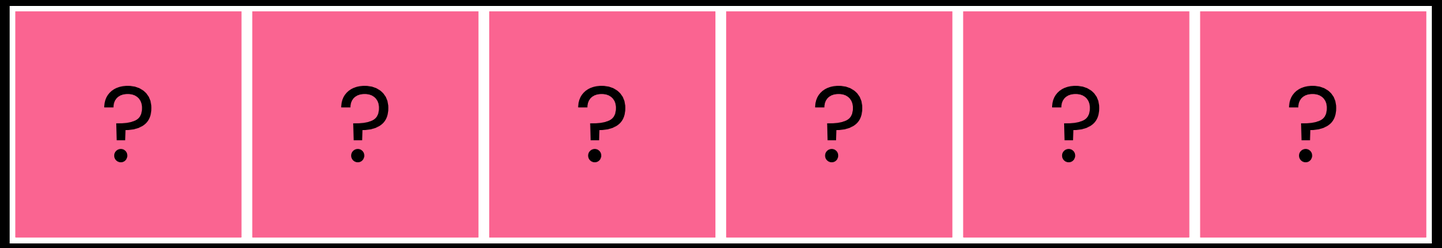
-)(
-))((
-)))(((
-)()(
- ())(

Test Case

Example 1:

```
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]
```

With n=3 open and n=3 close parenthesis, we have 6 total slots



So the solution isn't **just** that you need to have three pairs of parenthesis!
They have to be valid as well!

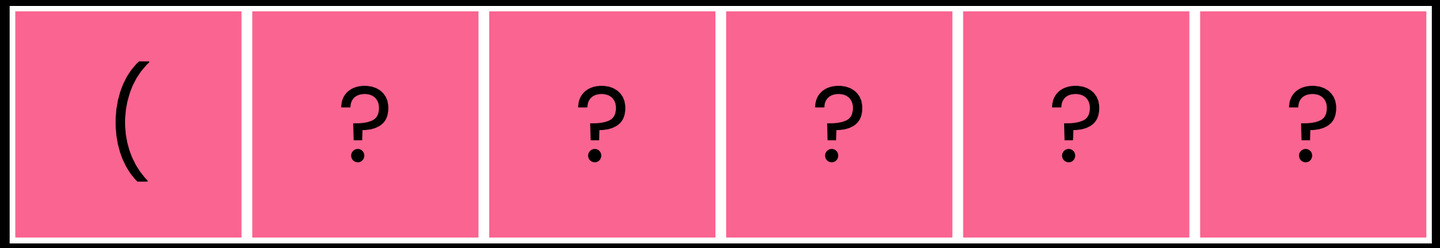
But now you have to find out how to recursively generate
pairs of parenthesis based on the rules of valid parenthesis.

Test Case

Example 1:

> **Input:** n = 3
> **Output:** ["((()))","(()())","(())()","()(())","()()()"]

With n=3 open and n=3 close parenthesis, we have 6 total slots



Let’s go through a valid parenthesis example.
The only valid first step is a ‘(’

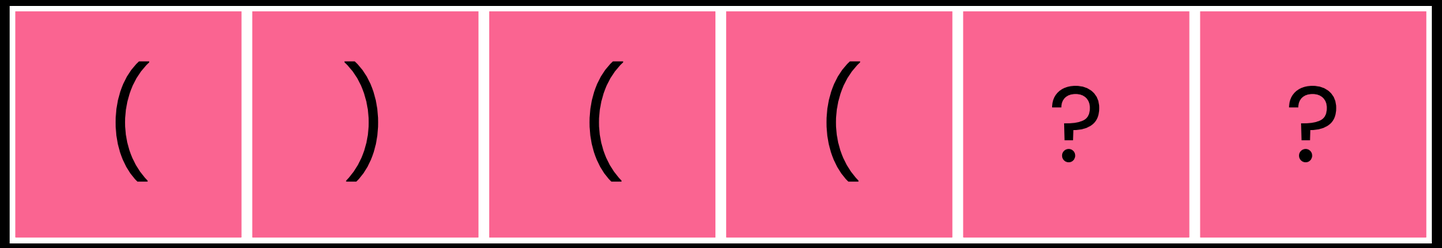
Try not to “hard code” this value,
and have your function generate this yourself

Test Case

Example 1:

> **Input:** n = 3
> **Output:** ["((()))","(()())","(())()","()(())","()()()"]

With n=3 open and n=3 close parenthesis, we have 6 total slots



What if the parenthesis was in this formation?

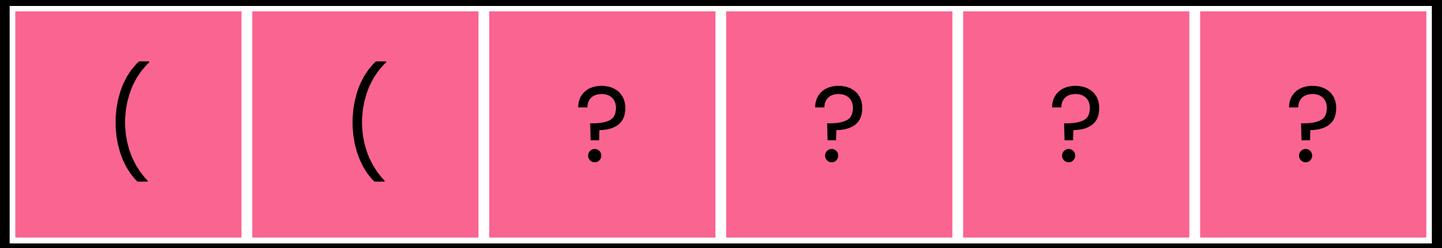
Since there are 3 open and 1 close parenthesis, there are no more open parenthesis left to fill. **So, the next parenthesis must be a close parenthesis.**

Test Case

Example 1:

> **Input:** n = 3
> **Output:** ["((()))","(()())","(())()","()(())","()()()"]

With n=3 open and n=3 close parenthesis, we have 6 total slots



What if the parenthesis was in this formation?

Since there are only 2 open and no close parenthesis, there are open parenthesis to be closed and more open parenthesis to be inserted.
So, the next parenthesis can be either a ‘(’ or a ‘)’.

There are multiple ways to implement recursive functions, such as having recursive "helper functions" in addition to your non-recursive main function.

HINT #1

Some recursive functions have a helper function that actually does the recursing.

```java
void helper_function(ArrayList<String> list, ... ){
        //base case
                list.add( ... );
        //recursive case
                //run the helper_function again
}
ArrayList<String> generate_parenthesis( ... ){
        ArrayList<String> list = new ArrayList<String>();
        helper_function(list, ...);
        return list;
}
```

Java

Python

```python
def helperFunction(list, ... ):
        //base case
                list.append( ... )
                return;
        //recursive case
                //call helperFunction() again

def generateParenthesis(self, n: int) -> list[str]:
        list = []
        helperFunction(list, ... )
        return list
```

(This may be a little hint as to how to do the problem)

Go for it!!!!

22. Generate Parentheses

Go for it!!!!

```java
void helper_function(ArrayList<String> list, ... ){
        //base case
                list.add( ... );
        //recursive case
                //run the helper_function again
}
ArrayList<String> generate_parenthesis( ... ){
        ArrayList<String> list = new ArrayList<String>();
        helper_function(list, ...);
        return list;
}
```

Java

Python

```python
def helperFunction(list, ... ):
        //base case
                list.append( ... )
                return;
        //recursive case
                //call helperFunction() again

def generateParenthesis(self, n: int) -> list[str]:
        list = []
        helperFunction(list, ... )
        return list
```

(This may be a little hint as to how to do the problem)

HINT #2

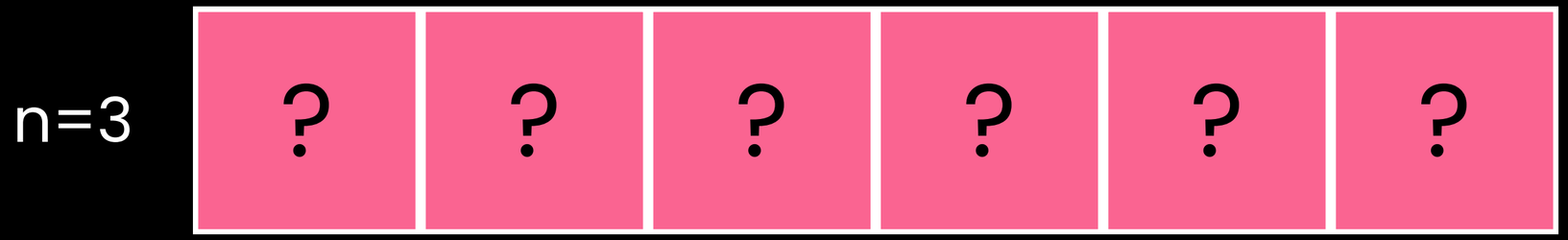
Recursive functions are kind of like lost travellers at a fork in the road... you as a programmer need to give your function **just enough context** to make the purpose work!

If you are lost on how to write your recursive function, ask yourself what variables you need to account for.

Solution

Remember that the point of recursion is to solve a problem by having a recursive case **that works at every step of the way**, and your base case is like your bounds.

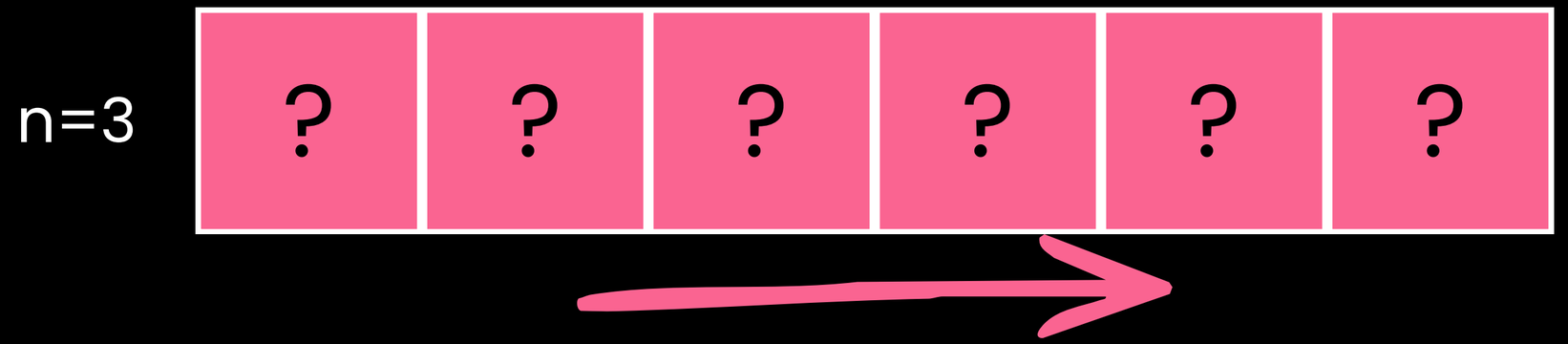
But first, how do we want to go about this problem?
We don’t want to be throwing darts at the dartboard blindfolded—
we want to be logically finding which parenthetical combinations work.



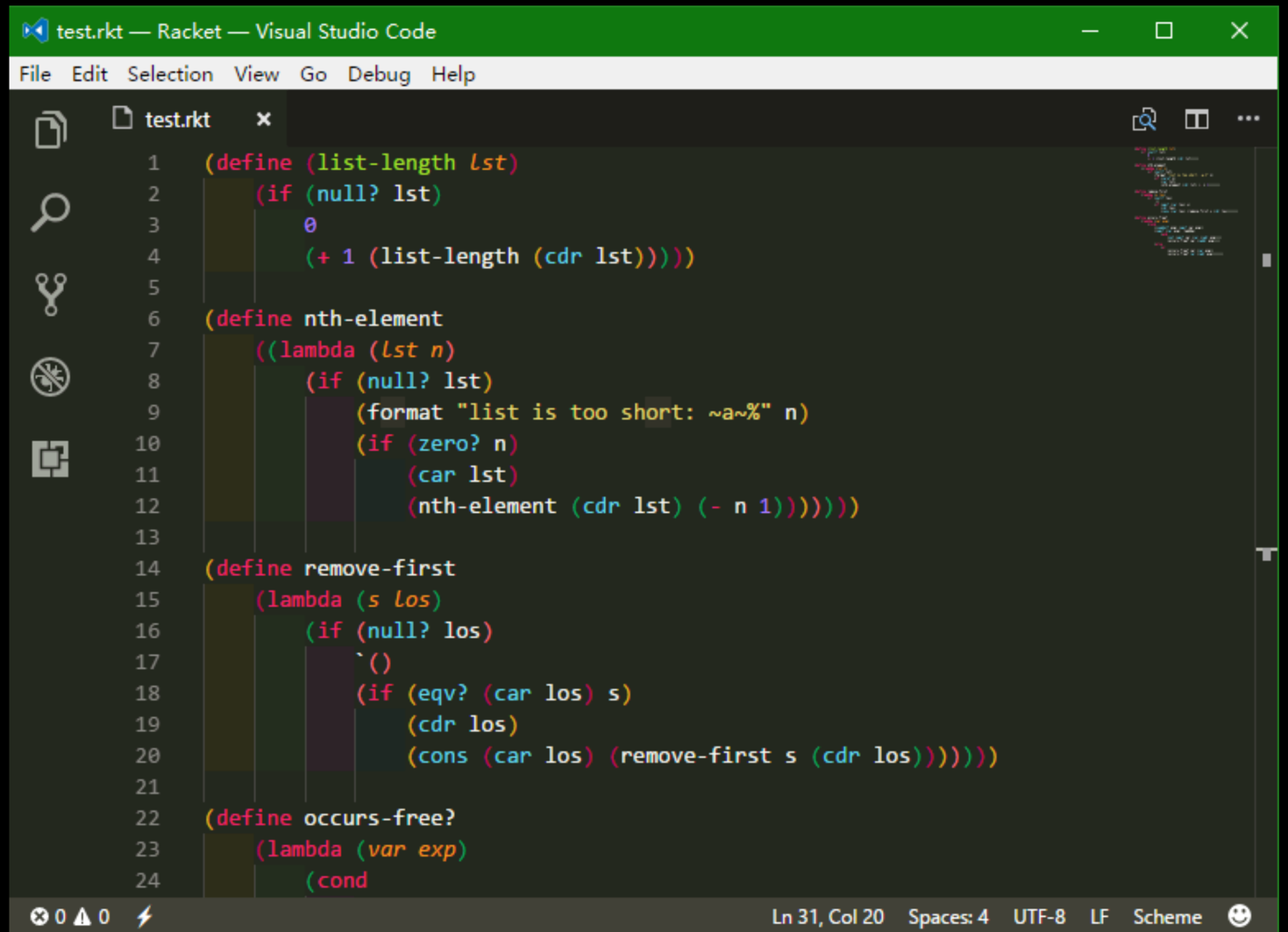
Solution

I would say it’s easiest to start from the left and go to the right.

Just like writing code, you need you keep track of how many parentheses you have so your compiles. It’s only natural that we will follow the same logic for this algorithm.



Maybe you would want to use this algorithm to write Scheme...

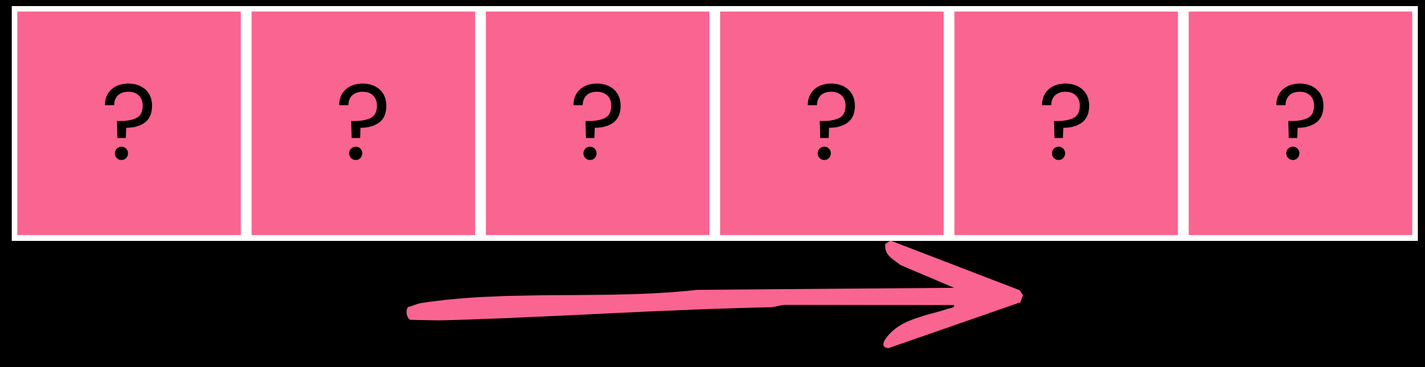


Solution

Let me propose a set of statements and rules for our recursive algorithm.

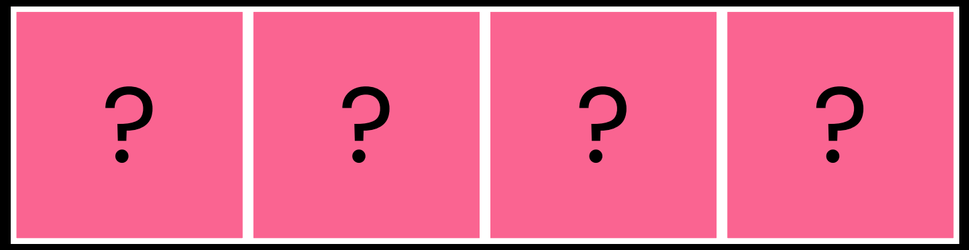
- We only have n opening parentheses
- We only have n closing parentheses
- We can only place a ‘(’ up to however many we have.
- We can only place a ‘)’ if there is an unmatched ‘(’ to the left of it, AND we can only place a ‘)’ up to however many we have.

n=3



Solution

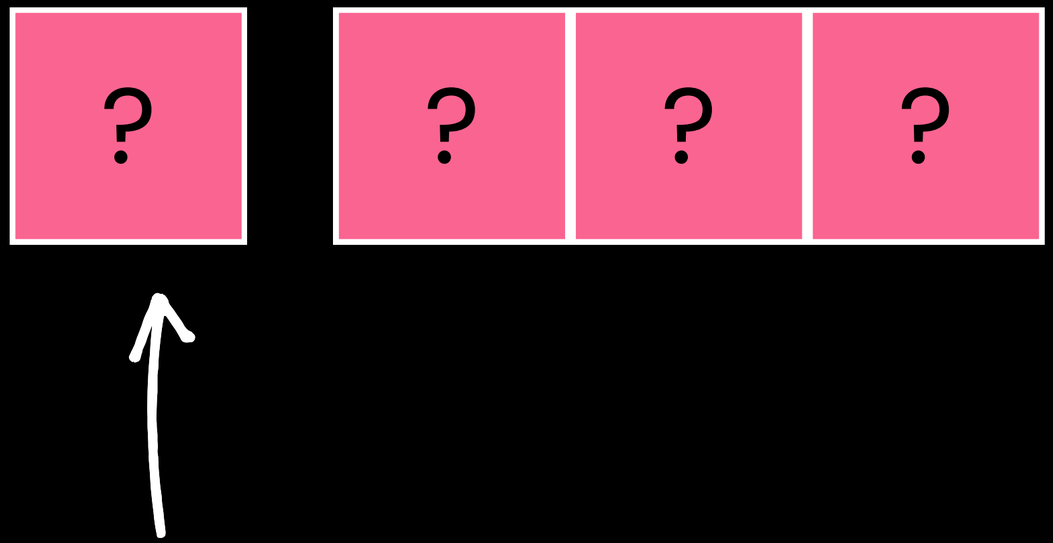
Take the example of n=2, where we have four slots



Our recursive function will fill in just one of these question marks per call according to these rules

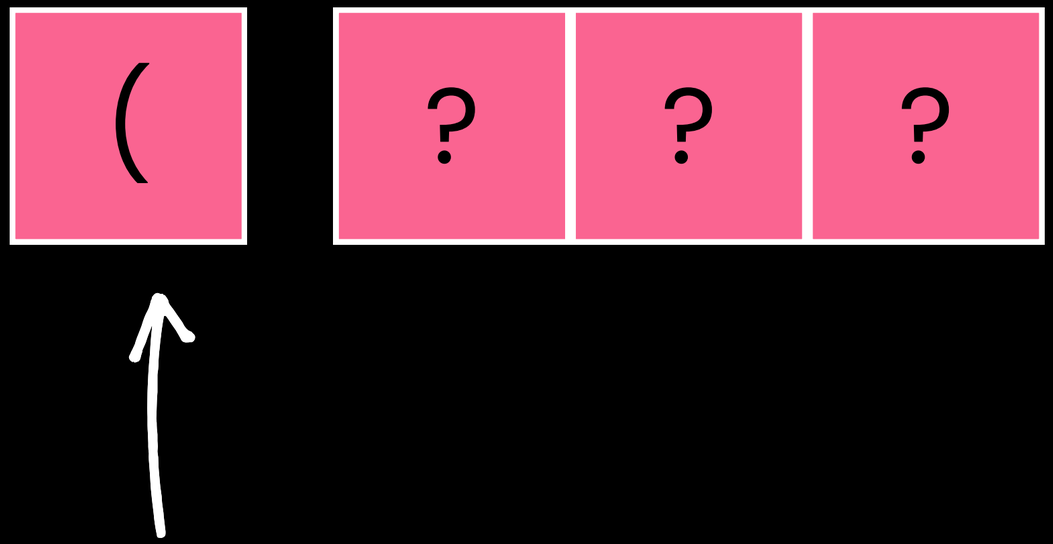
- We have 2 opening parentheses
- We have 2 closing parentheses
- We can only place a ‘(’ up to however many we have.
- We can only place a ‘)’ if there is an unmatched ‘(’ to the left of it

Solution



- We have 2 opening parentheses
- We have 2 closing parentheses
- We can only place a ‘(’ up to however many we have.
- We can only place a ‘)’ if there is an unmatched ‘(’ to the left of it, AND we can only place a ‘)’ up to however many we have.

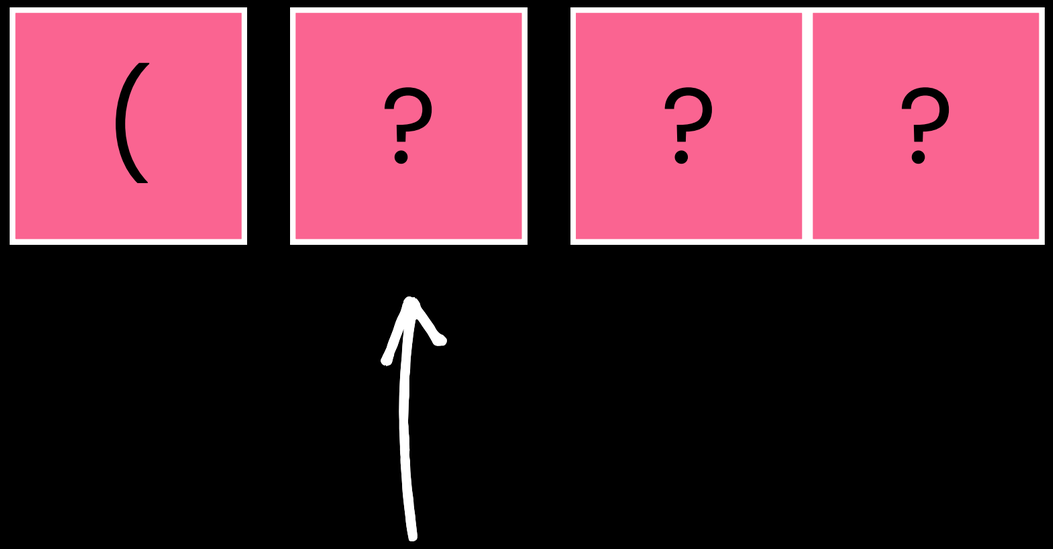
Solution



Rule 3 is valid since 2 > 0.
Rule 4 is not valid, since no ‘(’ before it.

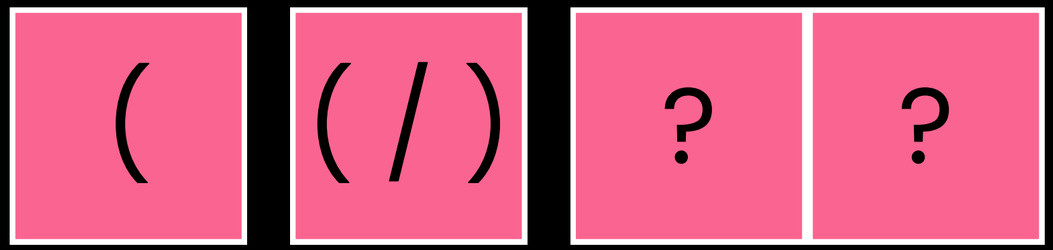
- We have 2 opening parentheses
- We have 2 closing parentheses
- We can only place a ‘(’ up to however many we have.
- We can only place a ‘)’ if there is an unmatched ‘(’ to the left of it, AND we can only place a ‘)’ up to however many we have.

Solution



- We have 1 opening parentheses
- We have 2 closing parentheses
- We can only place a ‘(’ up to however many we have.
- We can only place a ‘)’ if there is an unmatched ‘(’ to the left of it, AND we can only place a ‘)’ up to however many we have.

Solution



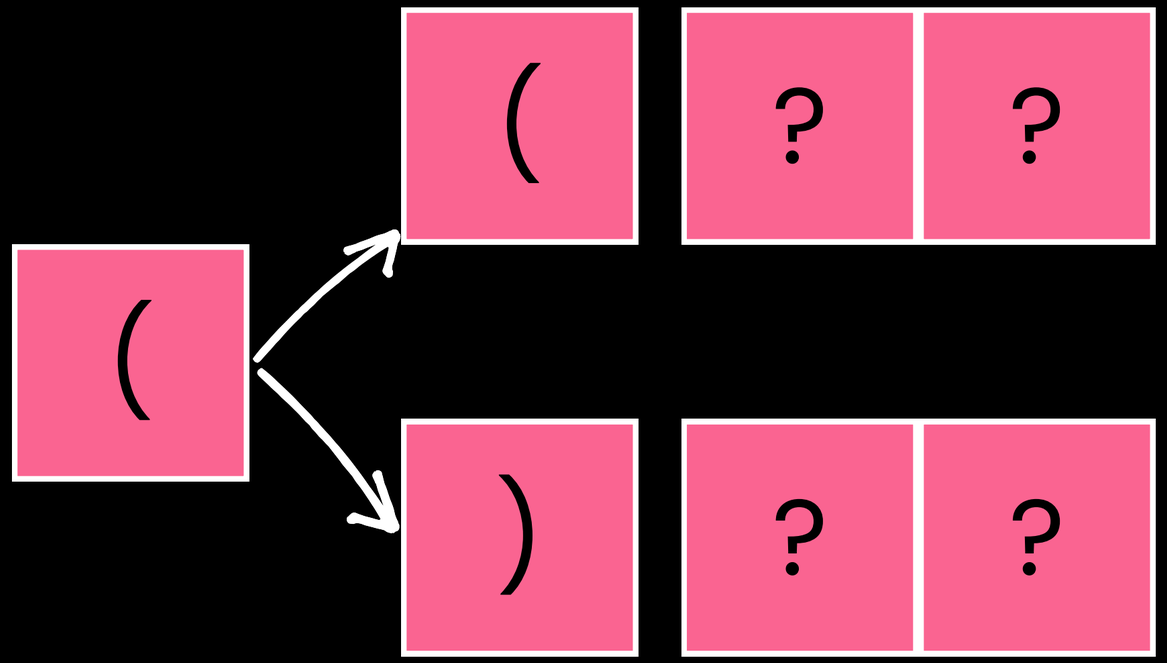
What can go here?

Rule 3 is valid since 2 > 0.
Rule 4 IS ALSO valid, since there is 1 ‘(’ before it.

How do we account for this??

- We have 1 opening parentheses
- We have 2 closing parentheses
- We can only place a ‘(’ up to however many we have.
- We can only place a ‘)’ if there is an unmatched ‘(’ to the left of it, AND we can only place a ‘)’ up to however many we have.

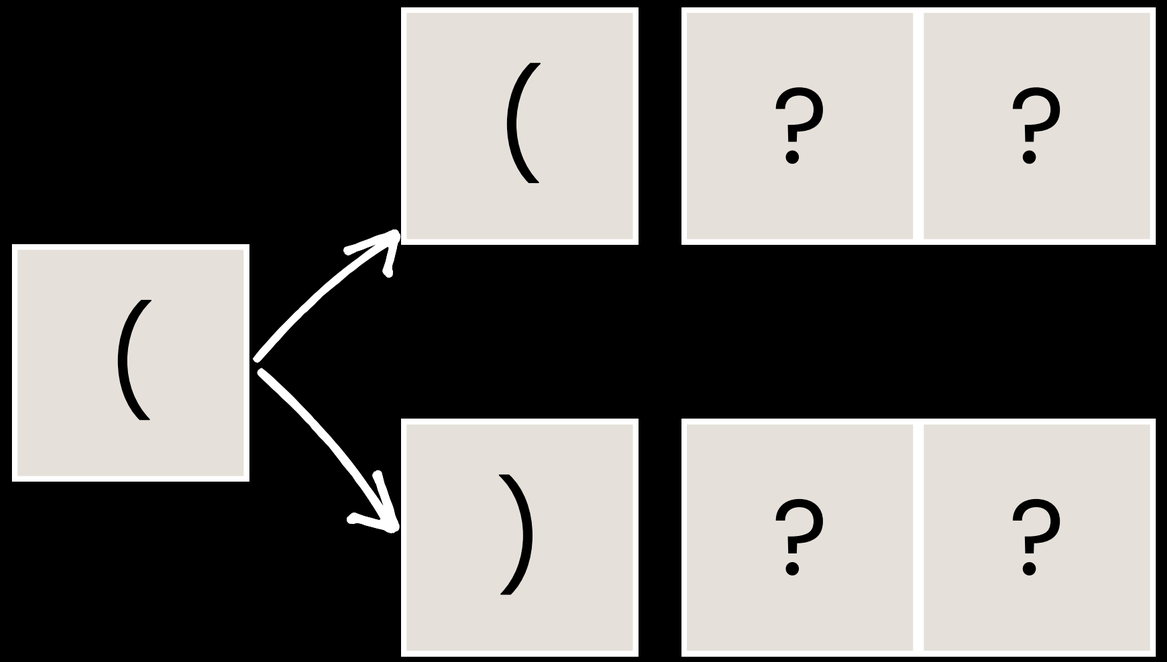
Solution



We need to account for both possibilities, so our recursive tree splits

- We have 1 opening parentheses
- We have 2 closing parentheses
- We can only place a ‘(’ up to however many we have.
- We can only place a ‘)’ if there is an unmatched ‘(’ to the left of it, AND we can only place a ‘)’ up to however many we have.

Solution



But wait, pause, how do I even account for this?!? Don’t I need a for loop, or something?

Not so fast, because the recursive function already accounts for this.

Solution

Instead of having one recursive case like in the warmup, we could instead have two.

What happens then is that if both of the recursive cases are true, then the algorithm explores both possibilities.

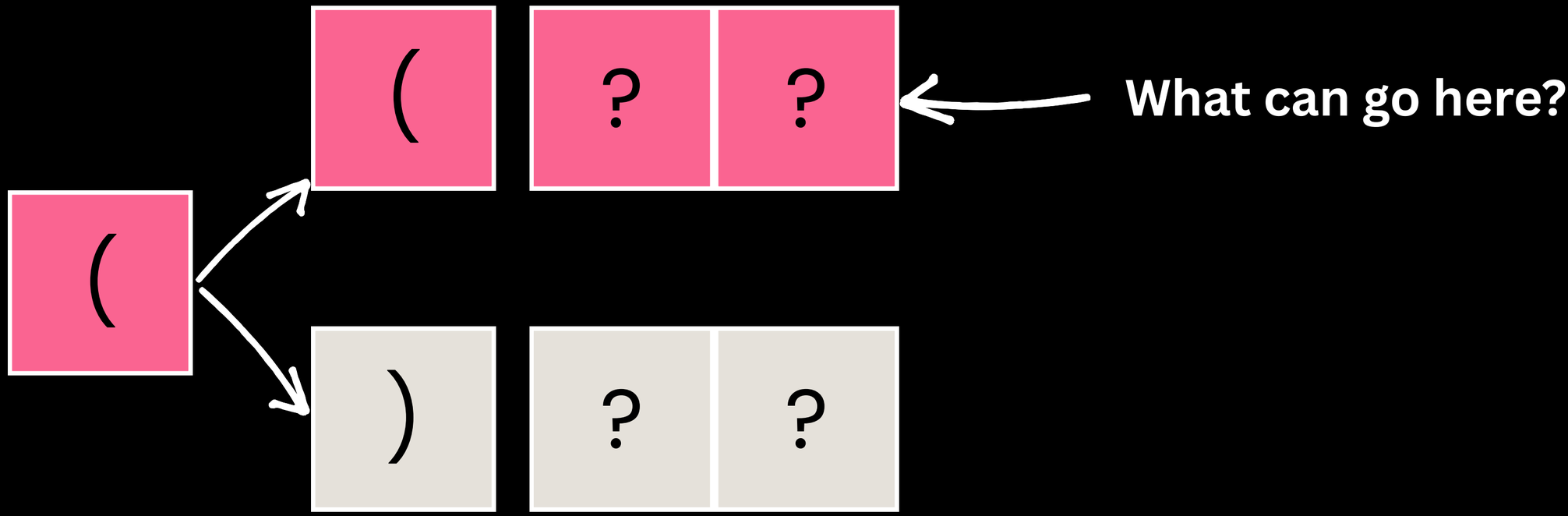
```python
def isPowerOfThree(n):
    if n <= 0:
        return False
    if n == 1:
        return True
    if n % 3 != 0:
        return False
    return isPowerOfThree(n // 3)
```

```
13 void helper_function(ArrayList<String> list, ... ){
12     //base case
11         list.add( ... );
10     if(/* recursive case 1 */):
 9         helper_function(list, current + '(', ... );
 8     if(/* recursive case 2 */):
 7         helper_function(list, current + ')', ... );
 6 }
```

red: base cases

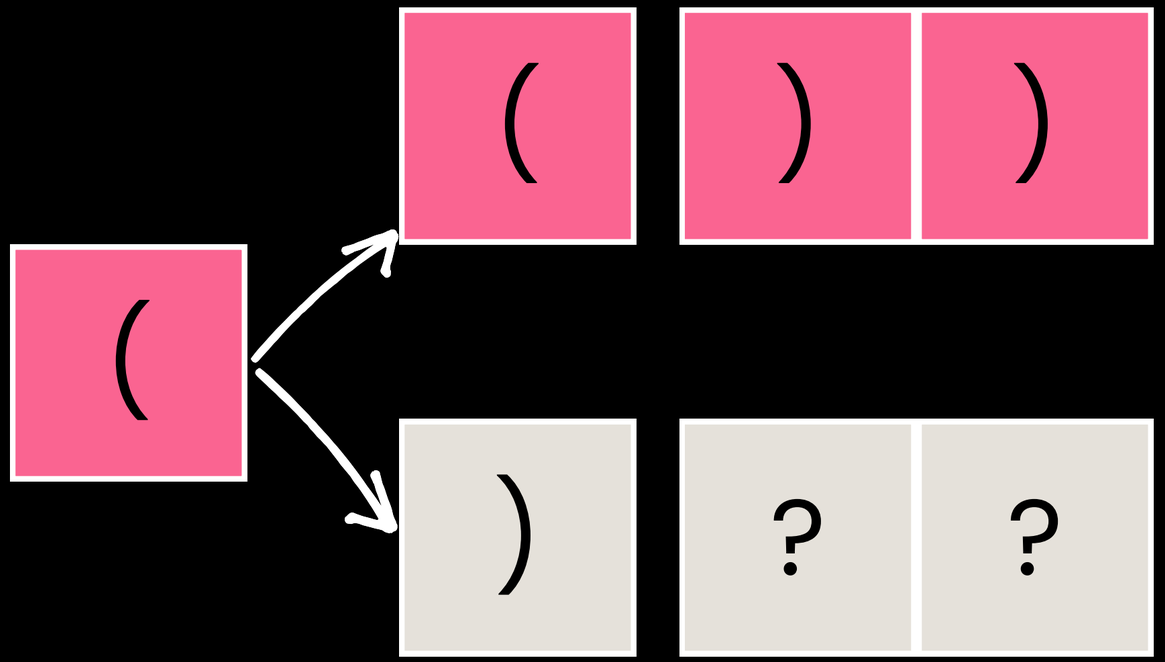
blue: recursive cases

Solution



- We have no more opening parentheses
- We have 2 closing parentheses
- We can only place a ‘(’ up to however many we have.
- We can only place a ‘)’ if there is an unmatched ‘(’ to the left of it, AND we can only place a ‘)’ up to however many we have.

Solution

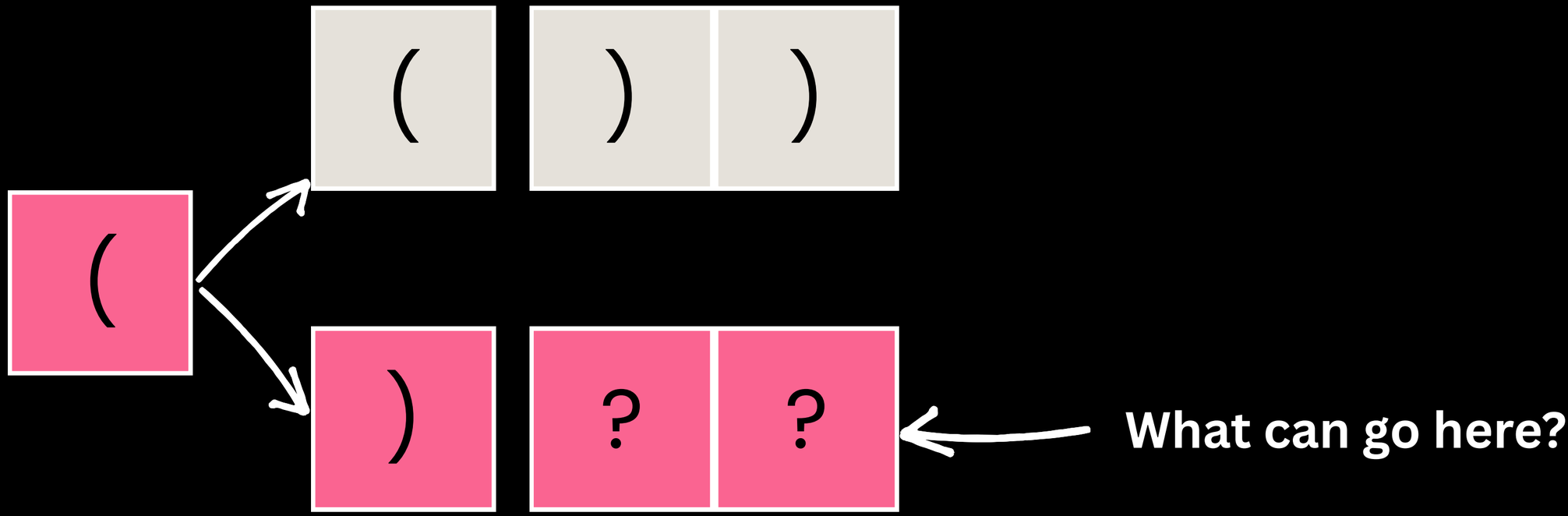


We only have closing parenthesis left, so we use them

(this would take two steps, I'm just simplfying here)

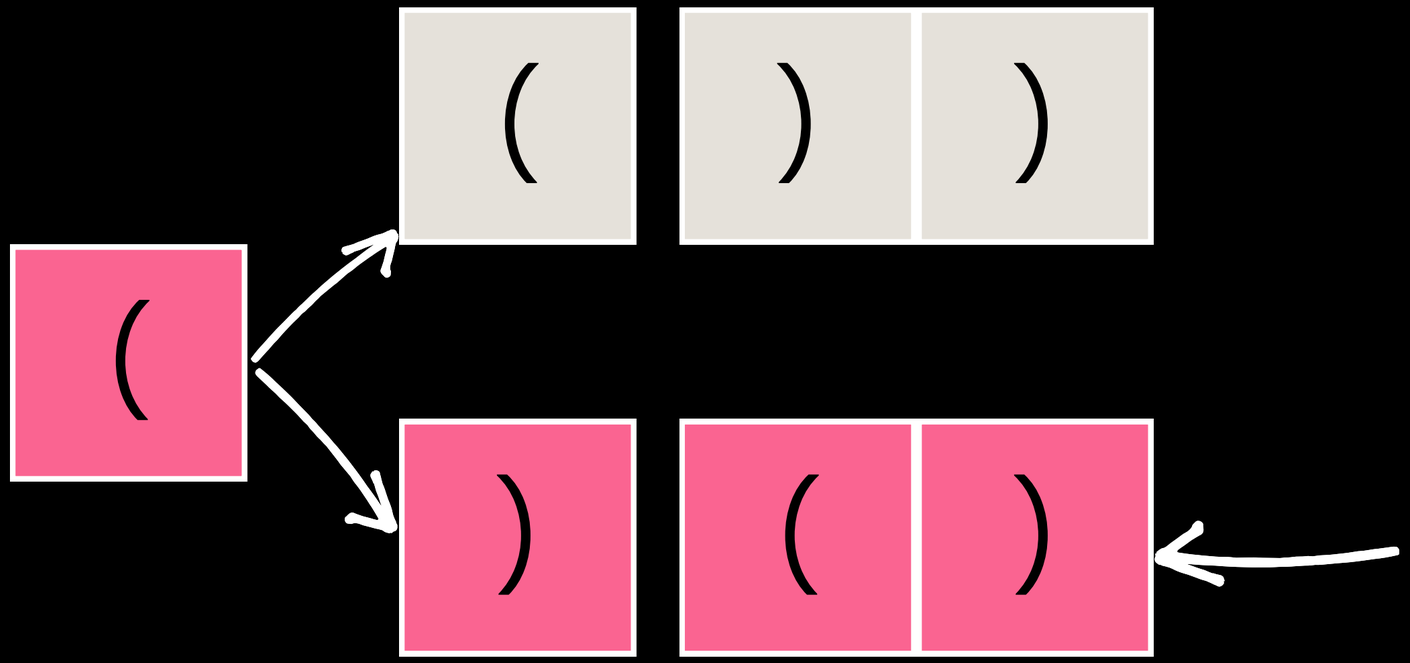
- We have no more opening parentheses
- We have 2 closing parentheses
- We can only place a '(' up to however many we have.
- We can only place a ')' if there is an unmatched '(' to the left of it, AND we can only place a ')' up to however many we have.

Solution



- We have 1 opening parentheses
- We have 1 closing parentheses
- We can only place a ‘(’ up to however many we have.
- We can only place a ‘)’ if there is an unmatched ‘(’ to the left of it, AND we can only place a ‘)’ up to however many we have.

Solution

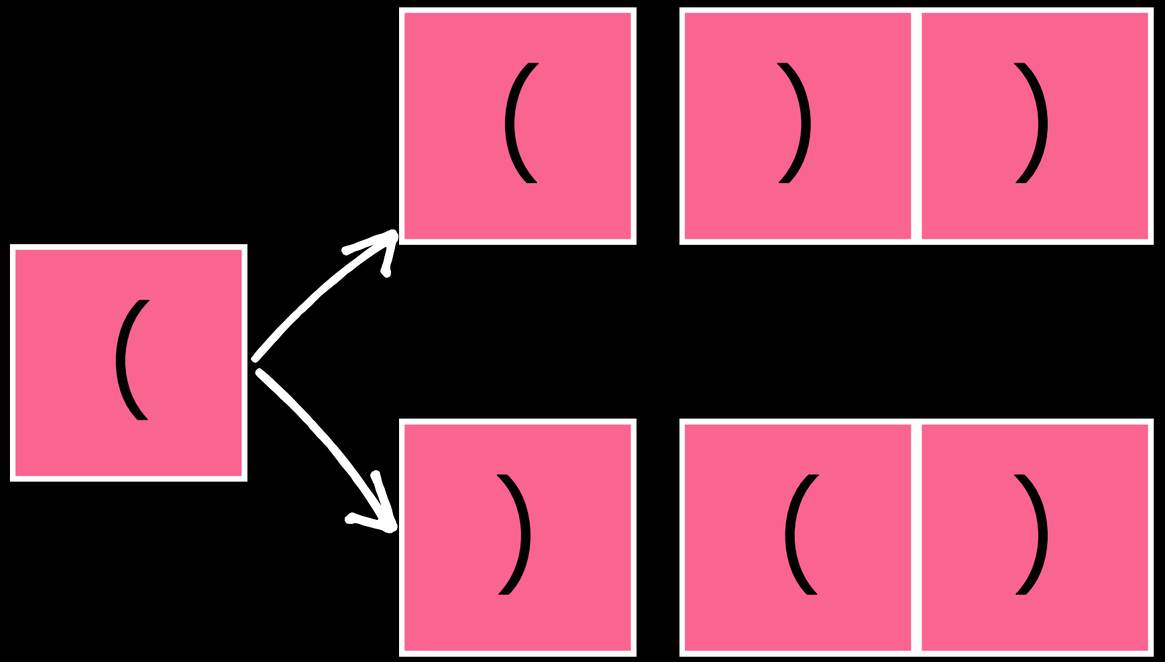


The only combination that does not violate the rules is ( )

(this would take two steps, I'm just simplfying here)

- We have 2 opening parentheses
- We have 2 closing parentheses
- We can only place a '(' up to however many we have.
- We can only place a ')' if there is an unmatched '(' to the left of it, AND we can only place a ')' up to however many we have.

Solution



For n=2, the two solutions are

(( ))
( ) ( )

If you follow it for n=3, you would get all the solutions as well.

- We have 2 opening parentheses
- We have 2 closing parentheses
- We can only place a ‘(’ up to however many we have.
- We can only place a ‘)’ if there is an unmatched ‘(’ to the left of it, AND we can only place a ‘)’ up to however many we have.

Solution

We create a list that keeps track of all valid solutions so far

We create a recursive function that fills each slot with an open, close, or both parenthesis

- It must keep track of how many open/close parenthesis we have used so far, and the slots that have already been filled
  - If we still have open parentheses available, we can use them
  - If we have used more open than close parenthesis, we can use a close parenthesis
- Once all slots have been filled, we add our combination to the list (this is our base case, signalling the end of recursion).

Return the list after all recursion is complete.

Solution - Java

```java
class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> results = new ArrayList<>();
        build("", 0, 0, n, results);
        return results;
    }

    private void build(String current, int openCount, int closeCount,
                       int n, List<String> results) {
        // Done: we've used all n pairs
        if (current.length() == 2 * n) {
            results.add(current);
            return;
        }
        // Rule 1: can add "(" if we haven't used all n
        if (openCount < n) {
            build(current + "(", openCount + 1, closeCount, n, results);
        }
        // Rule 2: can add ")" if there's an open one to close
        if (closeCount < openCount) {
            build(current + ")", openCount, closeCount + 1, n, results);
        }
    }
}
```

red: base cases

blue: recursive cases

Solution - Python

```python
class Solution:
    def generateParenthesis(self, n: int) -> list[str]:
        results = []
        self.build("", 0, 0, n, results)
        return results

    def build(self, current, open_count, close_count, n, results):
        # Done: we've used all n pairs
        if len(current) == 2 * n:
            results.append(current)
            return

        # Rule 1: can add "(" if we haven't used all n
        if open_count < n:
            self.build(current + "(", open_count + 1, close_count, n, results)

        # Rule 2: can add ")" if there's an open one to close
        if close_count < open_count:
            self.build(current + ")", open_count, close_count + 1, n, results)
```

red: base cases

blue: recursive cases

Hacker Devils

Presented by Sohum Sheth, Dean Richardson and Griffin Charlson

THANK YOU

for your time and attention

