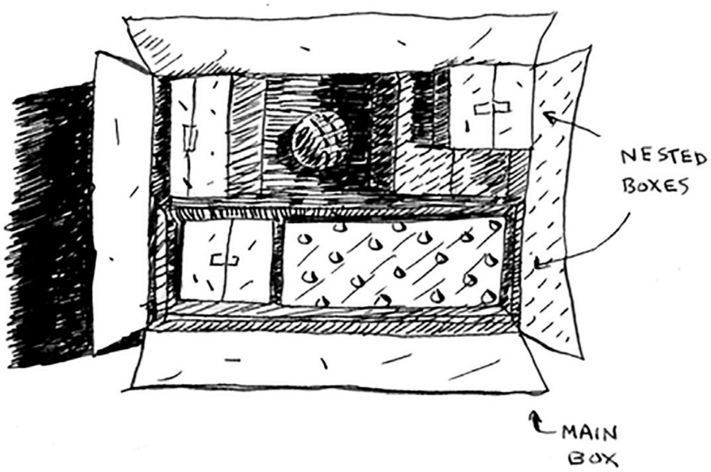
Recursion

You're in your grandma's attic 🏚️ and find this old, locked suitcase 🧳. You ask her about the key, and she points to a huge box 📦 full of smaller boxes 📦📦. "The key's in there somewhere," she says 👵

You open the first box, hoping to find the key 🔑—nope, just **more boxes** 📦📦. So, you dive in, opening box after box, like you're in a never-ending Russian nesting doll situation 😵‍💫. Finally, you hit that last box 📦, and **BOOM** 💥—there's the key! 🎉

Your plan:

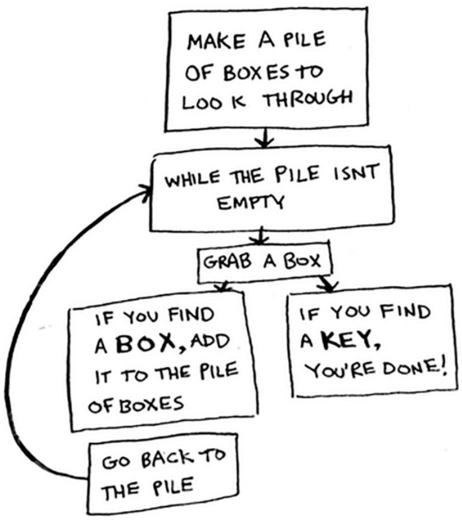
1. **Open the first box** 📦.
2. **Check inside**: Is it the key 🔑? If yes, **yay, you win**! 🎉
3. If not, there might be **more boxes inside** 📦📦📦.
4. **Repeat**: Open one box, check if it has more boxes or the key.
5. Keep opening boxes until you finally find the key 🔑 hiding in one of them!

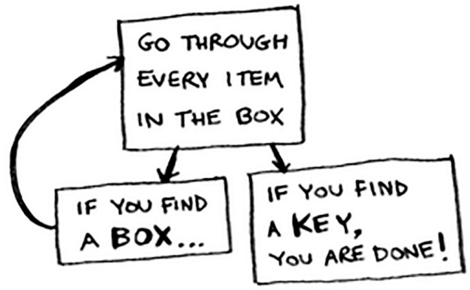


This is like **recursion** 🌀: open a box, check inside, and repeat if there's more boxes. You'll keep going deeper until you get that key!

You're basically Sherlock 🕵️, but with boxes instead of clues!

Actually, you're more like Dr. Strange 🧙, opening portals to different dimensions (boxes) until you find the right one with the key! 🔮🗝️





The first approach uses a while loop. While the pile isn’t empty, grab a box and look through it

The second way uses recursion. Recursion is where a function calls itself.

> 💡 "Loops may achieve a performance gain for your **program** 💻. Recursion may achieve a performance gain for your **programmer** 👨‍💻. Choose which is more important in your situation!"

Recursion can make your code cleaner and easier to understand for the **programmer** 👨‍💻, while loops are often better for **performance** 🚀.

Leigh Caldwell’s quote breaks it down perfectly:

- **Loops**: "Might make your program faster" ⏩.
- **Recursion**: "Might make your life easier" 🛠️.

So, you have to decide what’s more important in your situation: Do you need your code to run as fast as possible 🏃, or do you want it to be easier to write and maintain? ✍️

Base case and recursive case

```
def countdown(i):
    print(i)
    countdown(i - 1)
```

You run it, and uh-oh! 🚨 It keeps going forever!

> 3... 2... 1... 0... -1... -2... 😵

This is an **infinite loop**! (Press Ctrl-C to stop 🛑). Why? Because there's no way for the function to know when to stop! Without a stopping point, it keeps recursing endlessly.

Solution? Add a **base case**!

In any recursive function, you need:

- **Base case** 🏁: When the function **stops** recursing.
- **Recursive case** 🔁: When the function calls itself again.

```
def countdown(i):
    print(i)
    if i <= 0:  # Base case 🛑
        return
    else:  # Recursive case 🔄
        countdown(i - 1)
```

Recursion is a super cool concept because it mirrors the way we break down tasks in real life: doing one piece at a time. But, if you don't tell it when to stop 🛑, it can go on forever, like forgetting when to stop eating pizza 🍕!

Always remember:

- **Base case** = the exit.

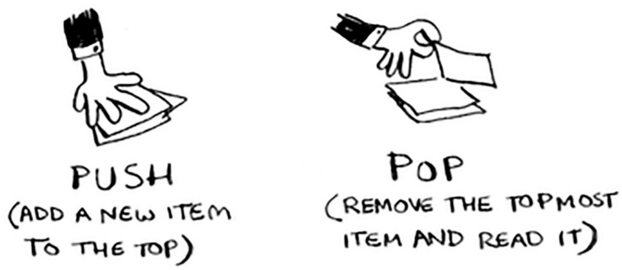
- **Recursive case** = the journey.

The stack

Imagine you're throwing a **BBQ** party 🎉. To keep everything organized, you have a **stack** of sticky notes 📝. This stack is your **to-do list**, but it's not just any to-do list—it's a **stack**, meaning it works in a very specific way:

1. **When you add a task**, it goes to the top of the stack 📋 (like writing "Buy charcoal" and slapping it on the pile).
2. **When you complete a task**, you take the topmost sticky note off 📝👋, read it, and remove it from the pile.

Now, this is exactly how the
call stack works in programming when using recursion (or in general!). Every time a function is called, it's like adding a sticky note to the stack (pushing). When that function finishes, it gets removed from the top (popping).



Let's see the todo list in action.

Let's link it to your BBQ example:

1. **First task**: "Set up the grill" 🔥 — you push this onto your to-do list 📝.
2. **Next task**: "Buy charcoal" 🏪 — this gets pushed on top.
3. **Next task**: "Invite friends" 📞 — added to the top of the stack.

When you start crossing off tasks:

1. You pop the top task first: "Invite friends" 📞— done!
2. Then, you pop the next: "Buy charcoal" 🏪— completed!
3. Finally: "Set up the grill" 🔥— finished!

This is exactly what the call stack does:

- When a function is called, it goes on top of the stack ⬆️.
- When it finishes, it pops off ⬇️.

The call stack

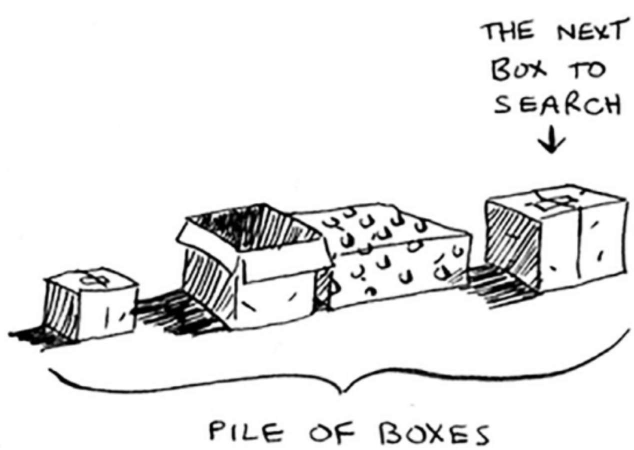
Imagine you're hosting a party 🎉, and each function call is like getting a new guest arriving at your door 🚪.

```
def greet(name):
    print("hello, " + name + "!")
    greet2(name)
    print("getting ready to say bye...")
    bye()
```

when you call `greet("Maggie")`, the computer adds it to the list, calls `greet2("Maggie")` (which goes on top), and keeps going until everything is done. Functions get stacked and unstacked as they're called and finished! 🎉

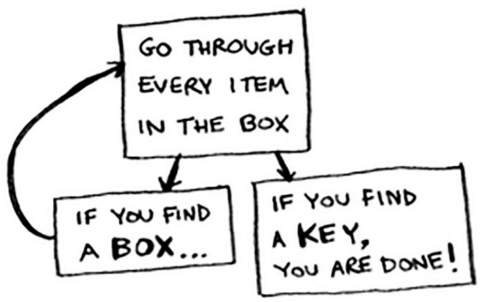
The call stack with recursion

Piles of Boxes and Recursion 📦📦📦

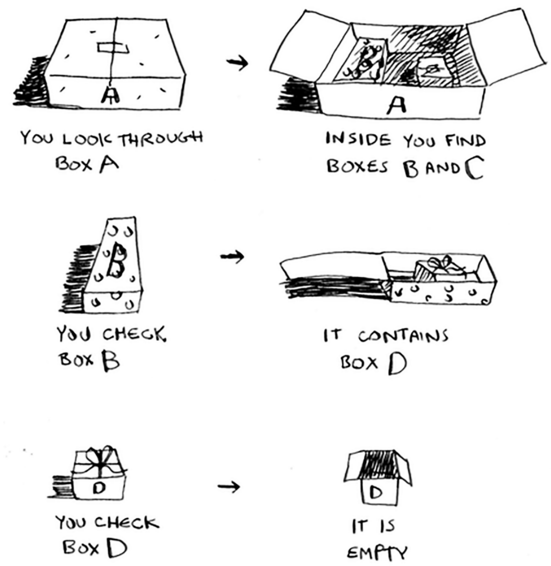


1. **Start with the top box**: This is the first function call. You open it and see what's inside.
2. **Find another box inside**: If the top box contains another box, you place it on top of the pile and open it next.
3. **Repeat**: Keep opening the new boxes you find inside each one, stacking them on top of the pile each time.

4. **Base case**: Eventually, you find a box with no more boxes inside. This is your stopping point. You start removing boxes from the top of the pile.
5. **Unstack**: After finishing with the topmost box, you go back to the box just below it and finish that one, and so on, until all boxes are removed.



The call stack visualized: As functions are called, they're added to the stack.



The call stack unwinding: As functions complete, they're removed from the stack.