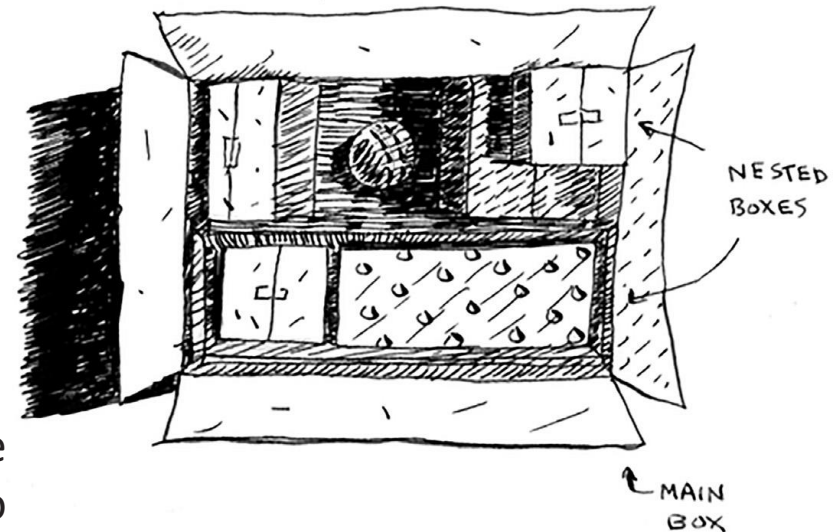


# Recursion



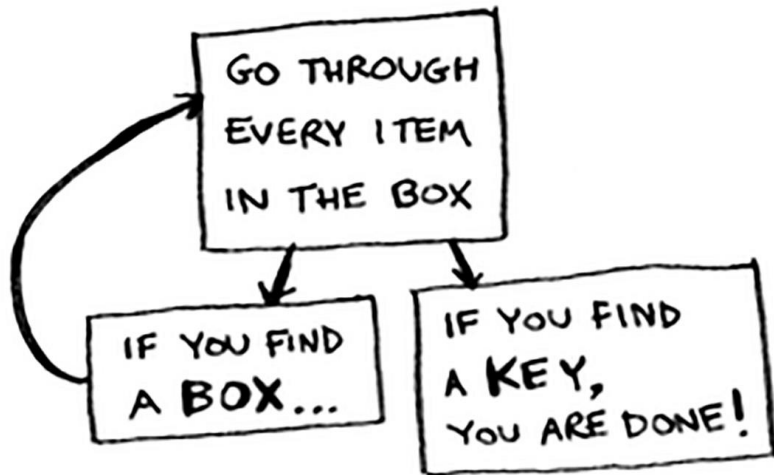
This box contains more boxes, with more boxes inside those boxes. The key is in a box somewhere. What's your algorithm to search for the key? Think of an algorithm before you read on



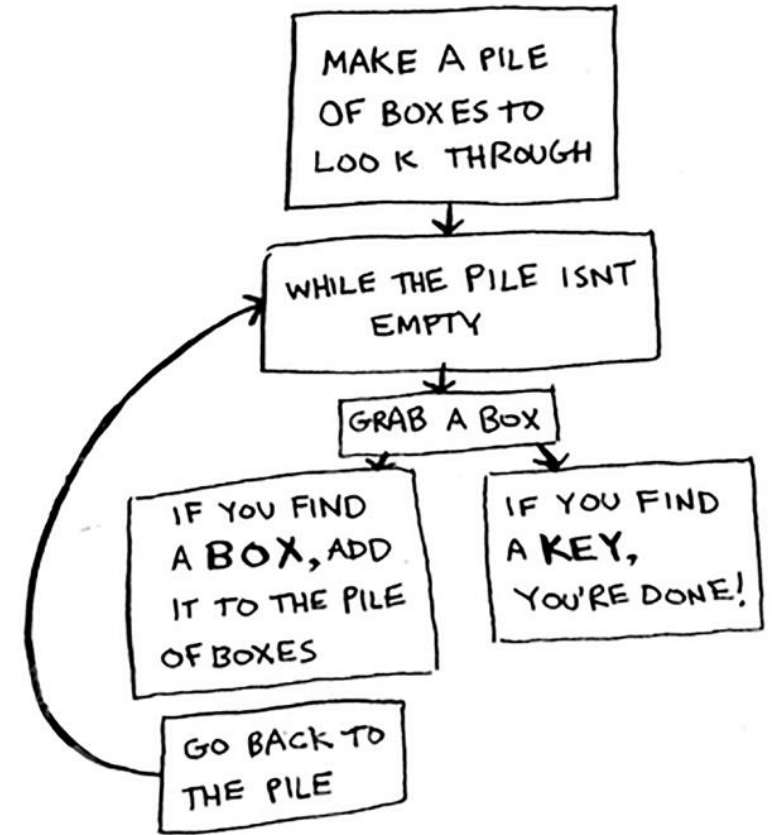
# Recursion

1. Make a pile of boxes to look through.
2. Grab a box, and look through it.
3. If you find a box, add it to the pile to look through later.
4. If you find a key, you're done!
5. Repeat.

Here's an alternate approach.



1. Look through the box.
2. If you find a box, go to step 1.
3. If you find a key, you're done!



# Recursion

- Which approach seems easier to you? The first approach uses a `while` loop. While the pile isn't empty, grab a box and look through it:

- 

```
def look_for_key(main_box):  
    pile = main_box.make_a_pile_to_look_through()  
    while pile is not empty:  
        box = pile.grab_a_box()  
        for item in box:  
            if item.is_a_box():  
                pile.append(item)  
            elif item.is_a_key():  
                print "found the key!"
```

# Recursion

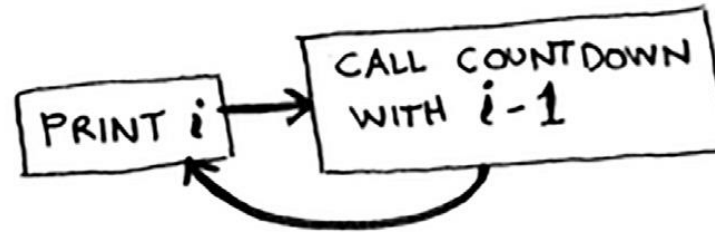
- The second way uses recursion. Recursion is where a function calls itself. Here's the second way in pseudocode:
- **def** look\_for\_key(box):
- **for** item in box:
- **if** item.is\_a\_box():
- look\_for\_key(item) ← **Recursion!**
- **elif** item.is\_a\_key():
- print "found the key!"

# Recursion

- Both approaches accomplish the same thing, but the second approach is clearer to me.
- Recursion is used when it makes the solution clearer.
- There's no performance benefit to using recursion; in fact, loops are sometimes better for performance.
- “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!”
- Many important algorithms use recursion, so it's important to understand the concept.

# Base case and recursive case

- Because a recursive function calls itself, it's easy to write a function incorrectly that ends up in an infinite loop. For example, suppose you want to write a function that prints a countdown, like this:
- 3...2...1  
You can write it recursively, like so:
- ```
def countdown(i):  
    print i  
    countdown(i-1)
```
- Write out this code and run it. You'll notice a problem: this function will run forever!





**Infinite loop**

(Press Ctrl-C to kill your script.)

# Base case and recursive case

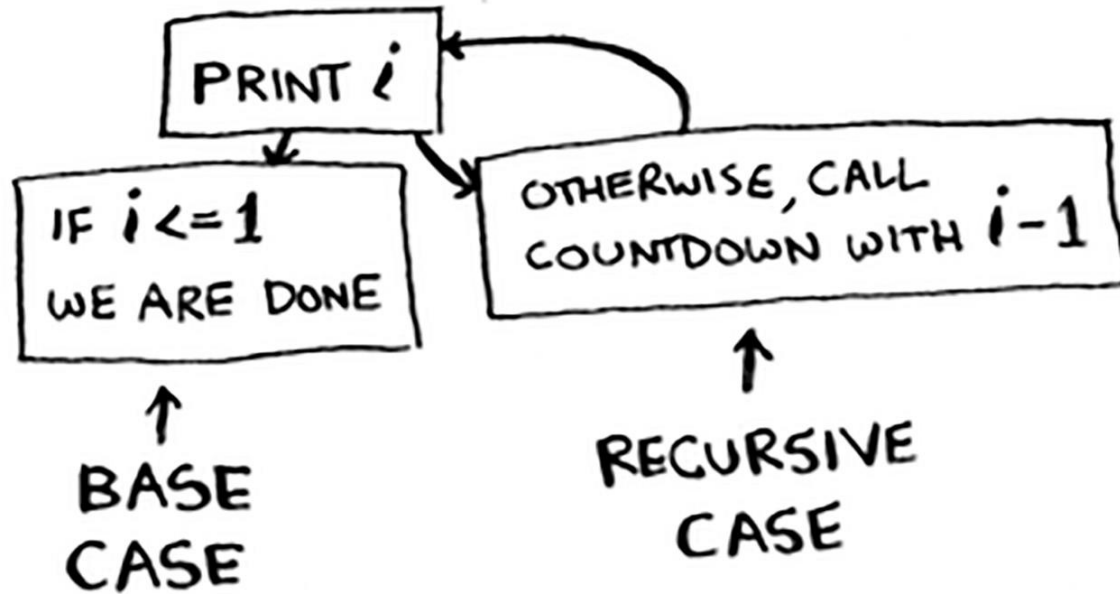
- When you write a recursive function, you have to tell it when to stop recursing.
- That's why every recursive function has two parts: **the base case**, and **the recursive case**.
- The recursive case is when the function calls itself.
- The base case is when the function doesn't call itself again ... so it doesn't go into an infinite loop.

Let's add a base case to the countdown function:

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

# Base case and recursive case

- Now the function works as expected. It goes something like this.





# Stack

- It's an important concept in programming. The call stack is an important concept in general programming, and it's also important to understand when using recursion.
- Suppose you're throwing a barbecue. You keep a todo list for the barbecue, in the form of a stack of sticky notes.
- Remember back when we talked about arrays and lists, and you had a todo list?
- You could add todo items anywhere to the list or delete random items. The stack of sticky notes is much simpler.
- When you insert an item, it gets added to the top of the list. When you read an item, you only read the topmost item, and it's taken off the list. So your todo list has only two actions: *push* (insert) and *pop* (remove and read)



# Stack

Let's see the todo list in action.



**PUSH**  
(ADD A NEW ITEM  
TO THE TOP)



**POP**  
(REMOVE THE TOPMOST  
ITEM AND READ IT)



POP A TODO  
OFF THE STACK



IT SAYS "GET FOOD".  
YOU NEED TO GET  
BUNS, BURGERS AND  
BAKE A CAKE.



LET'S PUSH THESE  
TODOS ONTO THE STACK

# Stack

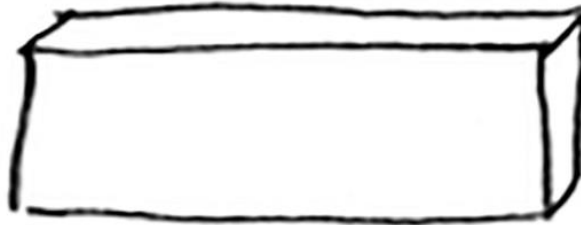
- Your computer uses a stack internally called the *call stack*. Let's see it in action. Here's a simple function:
- **def** greet(name):
- print "hello, " + name + "!" greet2(name)
- print "getting ready to say bye..." bye()

This function greets you and then calls two other functions. Here are those two functions:

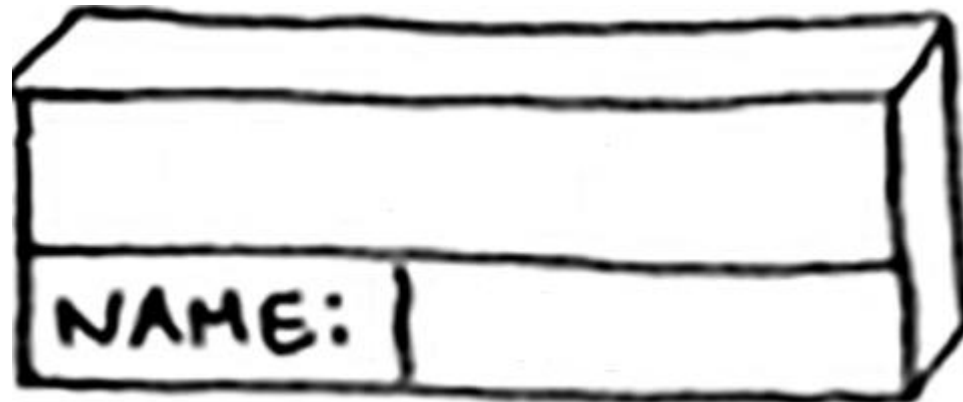
- **def** greet2(name):
- print "how are you, " + name + "?"
- **def** bye():
- print "ok bye!"

# Stack

- Suppose you call `greet("Ali")`. First, your computer allocates a box of memory for that function call.



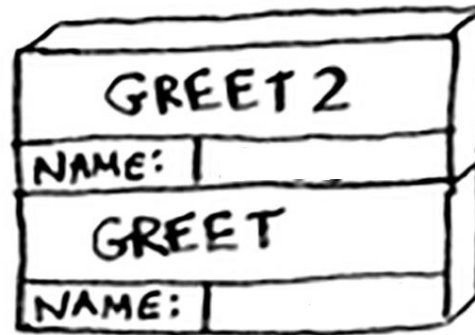
Now let's use the memory. The variable `name` is set to "Ali". That needs to be saved in memory



# Stack

- Every time you make a function call, your computer saves the values for all the variables for that call in memory like this. Next, you print `hello, Ali!` Then you call `greet2("Ali")`. Again, your computer allocates a box of memory for this function call

CURRENT  
FUNCTION  
CALL



# Stack

- Your computer is using a stack for these boxes. The second box is added on top of the first one. You print `how are you, Ali?` Then you return from the function call. When this happens, the box on top of the stack gets popped off.



# Stack

- Now the topmost box on the stack is for the `greet` function, which means you returned back to the `greet` function.
- When you called the `greet2` function, the `greet` function was *partially completed*.
- *When you call a function from another function, the calling function is paused in a partially completed state.* All the values of the variables for that function are still stored in memory.
- Now that you're done with the `greet2` function, you're back to the `greet` function, and you pick up where you left off. First you print getting ready to say bye... You call the `bye` function.



# Stack

- A box for that function is added to the top of the stack. Then you print `ok bye!` and return from the function call.



- And you're back to the `greet` function. There's nothing else to be done, so you return from the `greet` function too. This stack, used to save the variables for multiple functions, is called the *call stack*.

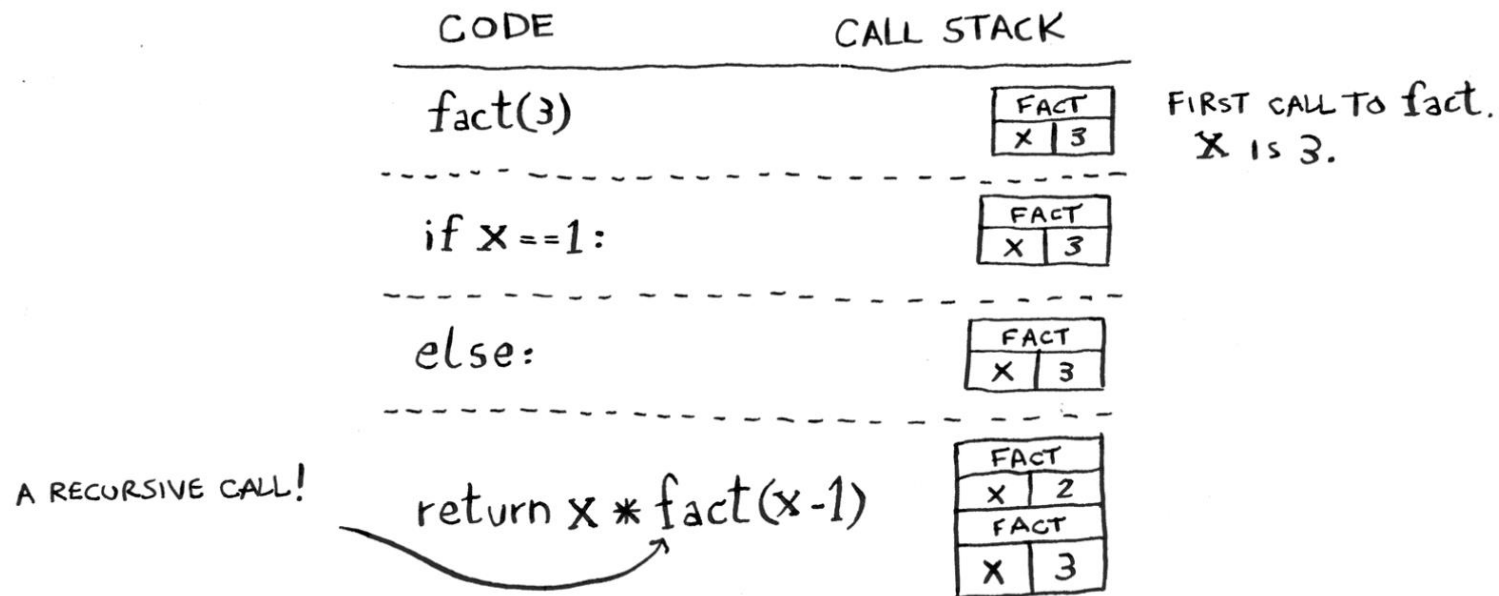


# Base case and recursive case

- Recursive functions use the call stack too! Let's look at this in action
- with the `factorial` function. `factorial(5)` is written as  $5!$ , and it's defined like this:  $5! = 5 * 4 * 3 * 2 * 1$ .
- Similarly, `factorial(3)` is  $3 * 2 * 1$ . Here's a recursive function to calculate the factorial of a number:
- **`def fact(x):`**
- **`if x == 1:`**
- **`return 1`**
- **`else:`**
- **`return x * fact(x-1)`**

# Base case and recursive case

- Now you call `fact(3)`. Let's step through this call line by line and see how the stack changes. Remember, the topmost box in the stack tells you what call to `fact` you're currently on



# Base case and recursive case

NOW WE ARE IN  
THE SECOND CALL  
TO `fact`. `X` IS 2

`if x == 1:`

|      |   |
|------|---|
| FACT |   |
| X    | 2 |
| FACT |   |
| X    | 3 |

← THE TOPMOST FUNCTION  
CALL IS THE CALL WE  
ARE CURRENTLY IN

`else:`

|      |   |
|------|---|
| FACT |   |
| X    | 2 |
| FACT |   |
| X    | 3 |

← NOTE: BOTH FUNCTION CALLS  
HAVE A VARIABLE NAMED `X`  
AND THE VALUE OF `X`  
IS DIFFERENT IN BOTH

`return x * fact(x-1)`

|      |   |
|------|---|
| FACT |   |
| X    | 1 |
| FACT |   |
| X    | 2 |
| FACT |   |
| X    | 3 |

← YOU CAN'T ACCESS  
THIS CALL'S `X`  
FROM THIS CALL  
AND VICE VERSA

# Base case and recursive case

if  $x == 1$ :

| FACT |   |
|------|---|
| x    | 1 |
| FACT |   |
| x    | 2 |
| FACT |   |
| x    | 3 |

WOW, WE MADE  
THREE CALLS TO  
fact, BUT WE  
HAD NOT FINISHED  
A SINGLE CALL UNTIL  
NOW!

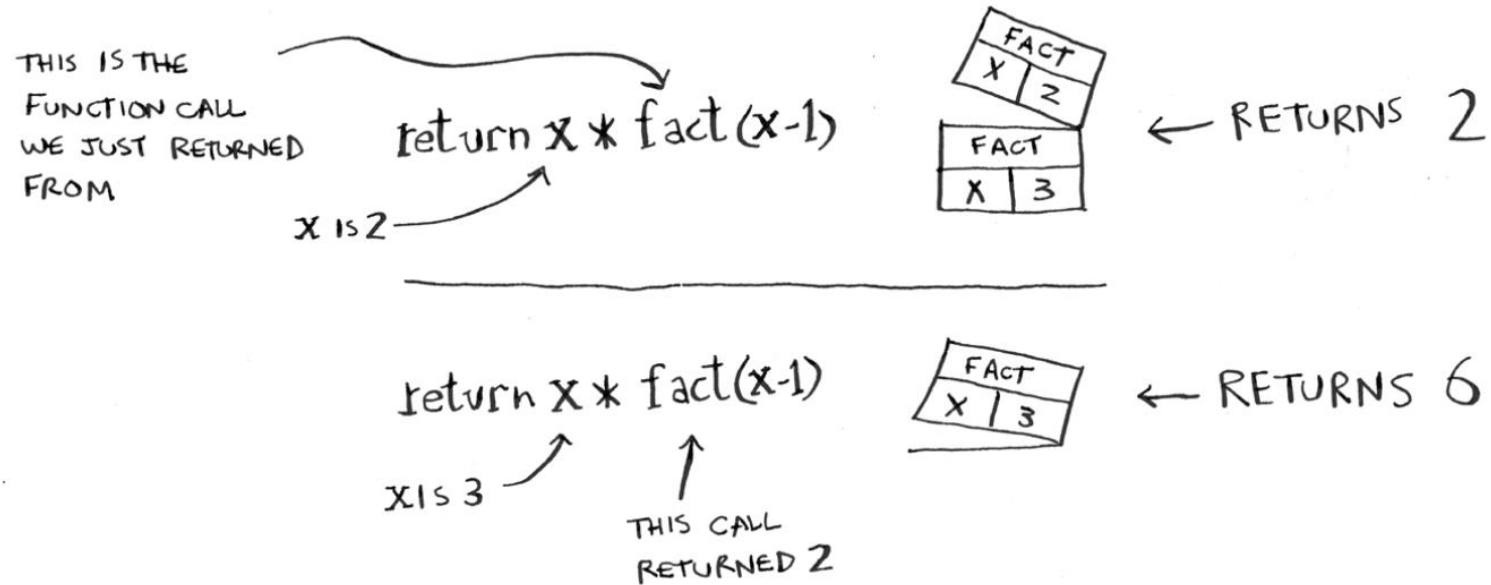
return 1

| FACT |   |
|------|---|
| x    | 1 |
| FACT |   |
| x    | 2 |
| FACT |   |
| x    | 3 |

THIS IS THE FIRST BOX  
TO GET POPPED OFF THE  
STACK, WHICH MEANS  
IT'S THE FIRST CALL WE  
RETURN FROM

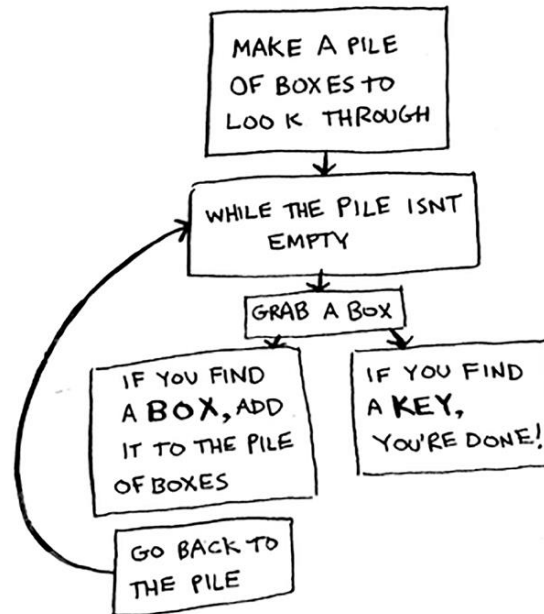
RETURNS 1

# Base case and recursive case



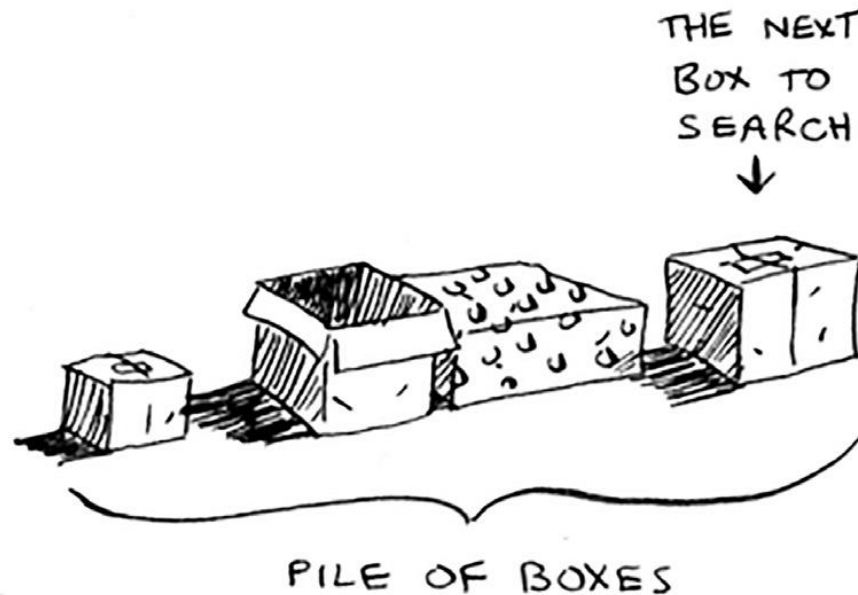
# Base case and recursive case

- Notice that each call to `fact` has its own copy of `x`. You can't access a different function's copy of `x`.
- The stack plays a big part in recursion. In the opening example, there were two approaches to find the key. Here's the first way again.



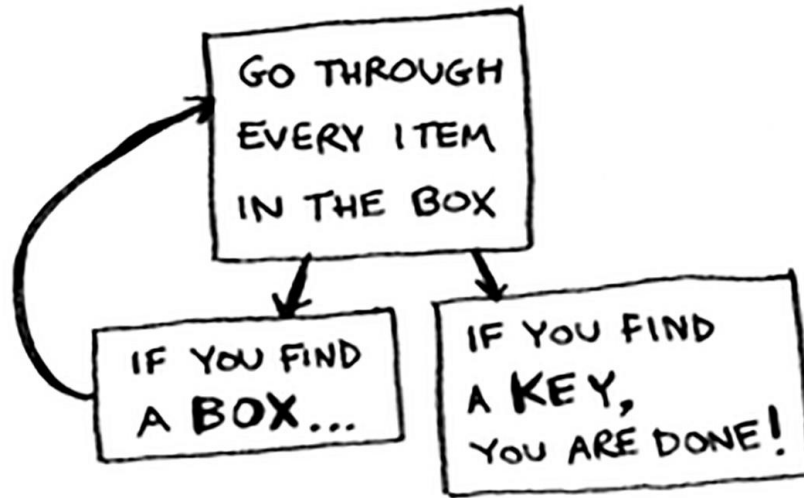
# Base case and recursive case

- This way, you make a pile of boxes to search through, so you always know what boxes you still need to search.



# Base case and recursive case

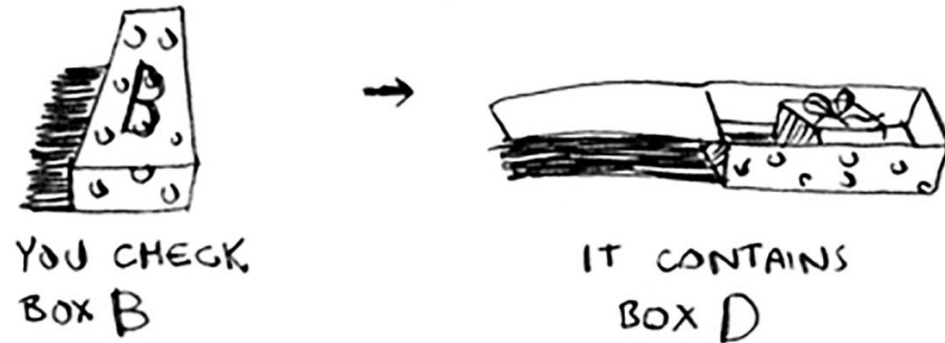
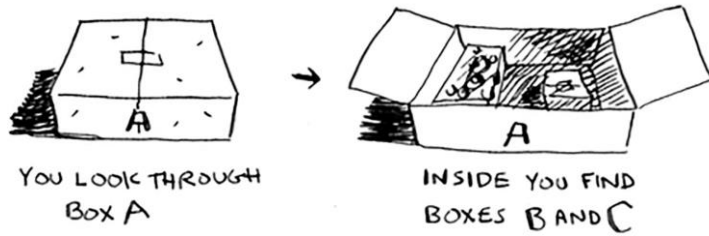
- But in the recursive approach, there's no pile.



- If there's no pile, how does your algorithm know what boxes you still have to look through? Here's an example.



# Base case and recursive case

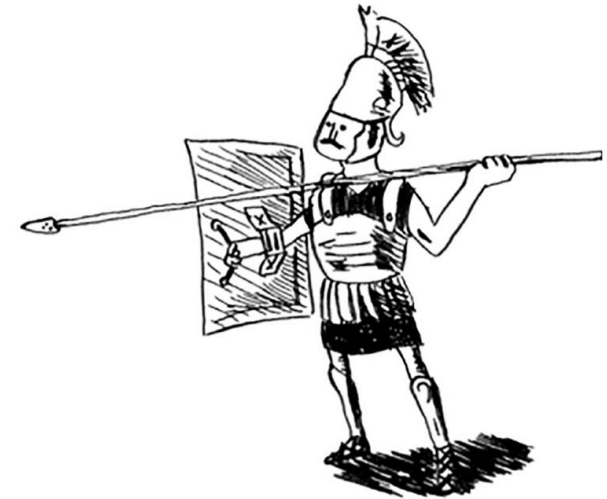


# Base case and recursive case

- The “pile of boxes” is saved on the stack!
- This is a stack of half completed function calls, each with its own half-complete list of boxes to look through.
- Using the stack is convenient because you don’t have to keep track of a pile of boxes yourself—the stack does it for you.
- Using the stack is convenient, but there’s a cost: saving all that info can take up a lot of memory.
- Each of those function calls takes up some memory, and when your stack is too tall, that means your computer is saving information for many function calls.
- At that point, you have two options:
- You can rewrite your code to use a loop instead.
- You can use something called *tail recursion*.

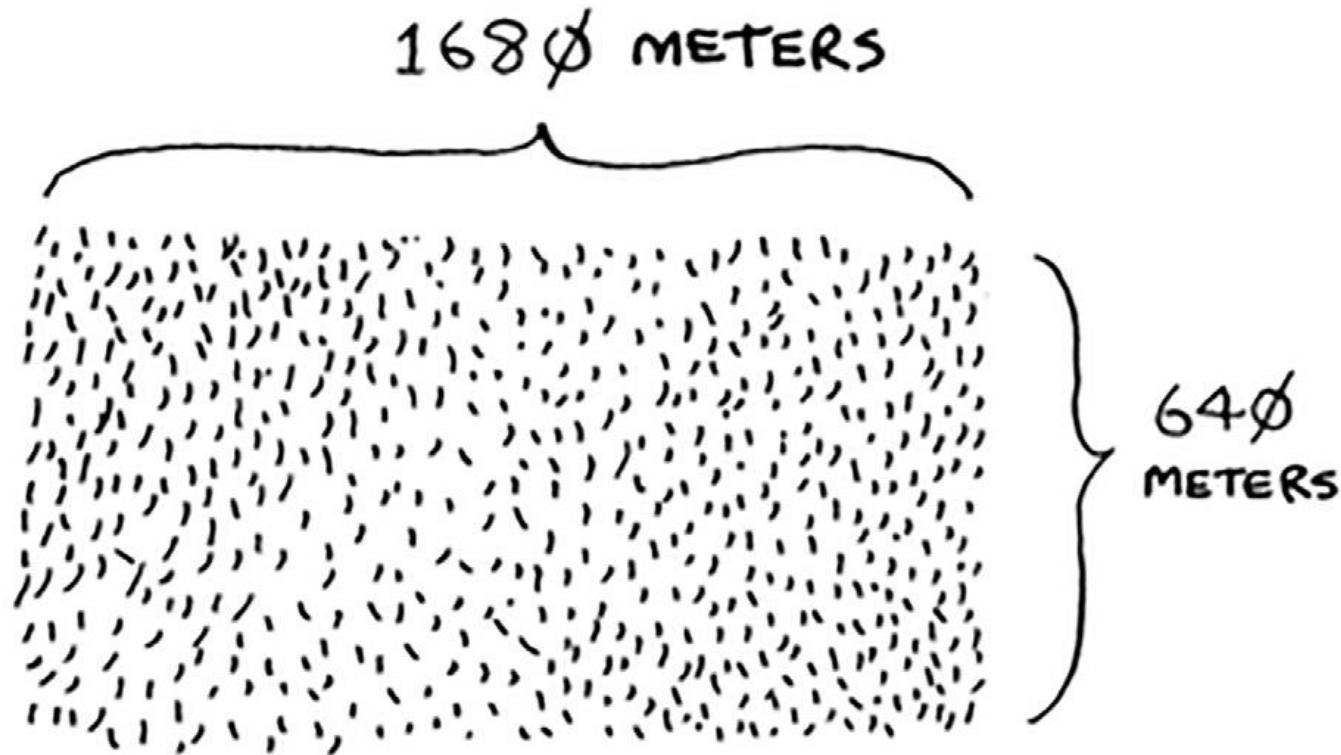
# Divide and Conquer (D&C)

- We'll explore *divide and conquer* (D&C), a well-known recursive technique for solving problems.
- D&C can take some time to grasp. So, we'll do three examples.
- First I'll show you a visual example. Then, I'll do a code example that is less pretty but maybe easier.
- Finally, we'll go through quicksort, a sorting algorithm that uses D&C.



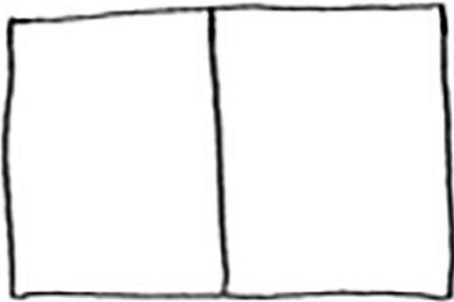
# Divide and Conquer (D&C)

- Suppose you're a farmer with a plot of land

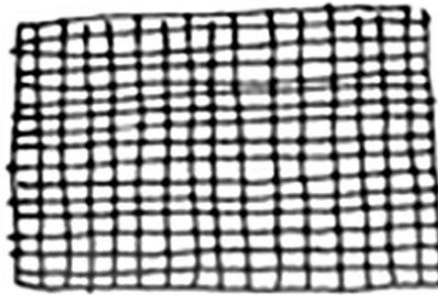


# Divide and Conquer (D&C)

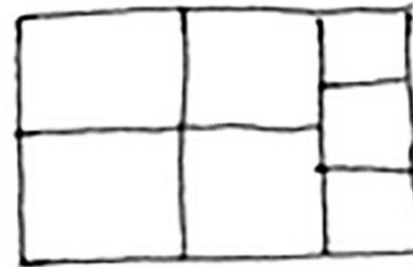
- You want to divide this farm evenly into square plots. You want the plots to be as big as possible. So none of these will work.



BOXES ARE  
NOT SQUARE



BOXES ARE TOO  
SMALL



ALL BOXES MUST  
BE SAME SIZE

- How do you figure out the largest square size you can use for a plot of land? Use the D&C strategy! D&C algorithms are recursive algorithms. To solve a problem using D&C, there are two steps: