Theory: Recursion basics

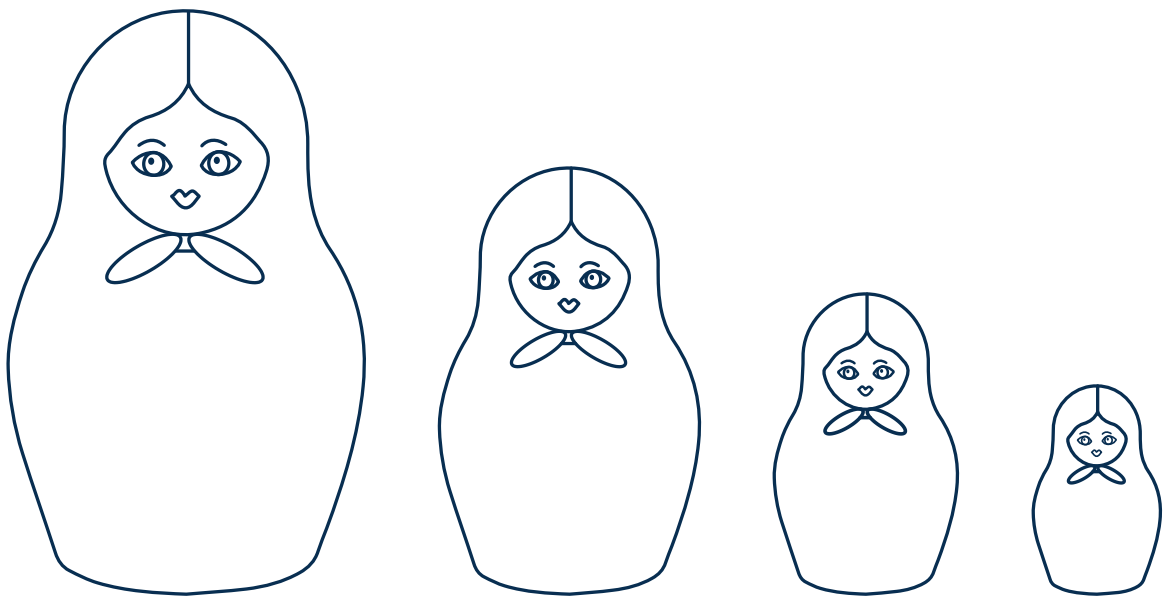
13 minutes 6 / 6 problems solved

Start practicing

In short, **recursion** in programming is when **a function calls itself**. It has a case where it terminates and a set of rules to reduce other cases to the first case. A function that can do it is called a **recursive function**. Sounds a little abstract? Let's try to get the main idea on the example.

§1. Recursive matryoshka

Think of it like a Russian doll, matryoshka. It's a doll, or, more accurately, a set of dolls placed one inside another. You open the first doll, and there's the second, open this one and get the third, and so on until you get to this last doll, which won't open.



If we want to find the smallest doll, we take our big matryoshka and try opening it: if it gives in, we go on and on opening our dolls, until we finally find that tiny one. Recursion works in pretty much the same way, so let's use it as a metaphor to understand more complicated principles.

Imagine that you got a set of dolls like this as a present, and you want not only to find the smallest one but also to count them all. How many dolls do you have? No clue. Let's say we have x dolls. As a true recursion enthusiast, you decide to count them recursively. Each time you are optimistic, so you ask yourself a question: "Is this doll the smallest one?" You manage to open the doll x, but you are not losing hope. "Oh well, maybe the doll $x - 1$ won't open". Finally, you get to the tiniest doll and exclaim: "Here is an unopenable doll! This doll is the first!" Now you understand that the doll that you opened the last was actually the second doll, then the third... And then you can continue till you find x.

Once again: first, you open them one after another, and only when you get to the smallest one you can count them, retracing your steps. If you were a recursive function designed to count the matryoshkas, you would work exactly as described.

§2. Designing a function

Most (if not all) programming languages have recursion (in other words, they allow a function to call itself). It is very convenient to know how to create recursive functions, so let's now create an algorithm to count the dolls. Each recursive function consists of the following steps:

- A trivial base case stops the recursion. This is the case we know the result for. For example, if we find a doll we can't open, we take it and proudly state: "it's our smallest doll!"
- A reduction step (one or more, imagine that our doll contains two dolls inside it!) gets us from the current problem to a simpler one. For example, if our doll can be opened, we open it and look at what is inside.

Current topic:
Recursion basics
Topic depends on:
Computer algorithms
Topic is required for:
Divide and conquer
Depth-first search
Merge sort
Tree
Recursion in Python
Recursion

If we are talking about the trivial base of our $count()$ function, it is just one doll. In other words, if x equals 1, then we can stop thinking and just be happy.

We are now imagining the matryoshkas from the real world, so let's just say the following: if x is not equal to 1, then it is bigger. In this case, we need to try our $count()$ function on $x - 1$. Just bear in mind that we will need to add this one doll that we opened to the result later.

```
How to count X dolls:
    If X is 1, the result is 1.
    If X is not 1, see: "How to count X-1 dolls" + 1.
```

But wait, who counts like this? Good question! And here is when we get to our next point: is recursion a good alternative?

§3. Advantages and disadvantages

Many recursive functions can be written another way: we could simply go through all numbers from 1 to n and compute the function for each number. For example, we can open the largest doll, say "One", throw the doll away, and repeat these steps until we found the last one. This way of computing is called **the loop**. But which way is more efficient?

It depends on the programming language. As a rule, in Python and Java, loops are more efficient in terms of **time** and **memory**. Recursion is slower and "heavier" because each call of a function takes additional memory, and recursive functions usually get called many times.

In that case, why recursion? Well, it has one certain advantage over loops: in some cases, it is intuitive. If you are certain that some function uses itself, it is much faster to write 3-4 lines of recursive code than to think how exactly a loop should behave. If you are short on time but don't have to worry about memory consumption, recursion is your choice.

So, recursion is usually slower and less memory-efficient, but it saves the developers' time.

Let's look at one classic (and more practical, to be perfectly honest) problem that recursion can effectively deal with.

§4. The factorial example

A classic example of recursion is a math function called **the factorial**.

The **factorial** of a non-negative integer n is equal to the **product of all positive integers from 1 to n inclusively.** Also, by definition, **the factorial of 0 is 1**. Let's take a normal number though: for example, the factorial of 5 (written as $5!$) is $1 * 2 * 3 * 4 * 5 = 120$.

So, we know the factorial of 0 and the factorial of 1. Also, we could say that **the factorial of any number $n > 1$ is equal to n multiplied by the factorial of $n - 1$**. For example:

$$4! = 4 * 3! = 4 * 3 * 2! = 4 * 3 * 2 * 1! = 4 * 3 * 2 * 1 = 24$$

If we choose the "function language", it will look like:

$$4! = 4 * 3! = 4 * 3 * 2! = ... = 4 * 3 * 2 * 1 = 24$$

Do you see what's happening? When writing a function to compute the factorial, we could do it recursively! There is no need to think a lot because the recursive function can be created just using the definition of **factorial**. We have a trivial case (for 0 or 1, our function returns 1), and the reduction step (if our number $n > 1$, the function returns $n * (n - 1)!$). And now try to imagine the factorial function with loops. It isn't that obvious, is it?

As you can see, recursion is a simple yet very powerful idea. Enjoy!

Report a typo

529 users liked this theory. **12** didn't like it. **What about you?**

Table of contents:

Start practicing

Comments (15) Hints (6) Useful links (2) Show discussion