How to Think Recursively | Solving Recursion Problems in 4 Steps

Jack Chen Follow

May 12, 2019 · 9 min read ★

Original comic from Safely Endangered. No idea who created this variant of the meme.

Disclaimer

This article is not meant to introduce more advanced concepts like dynamic programming. Instead, it will introduce the mentality required to start solving recursive problems.

The examples are in Javascript, but I'll write answers in Python and C++ in the bottom of this article.

What is recursion

Since you are reading this article, I'm assume that you have a vague idea of recursion, so I won't go deep into the context of it. Personally, I like to think of recursion as the following.

> Recursion is a way to solve a problem by solving smaller subproblems.

Before we start ...

Photo by Will Porada on Unsplash

Stop going through the function calls!

This is a large hurdle that hampers students when they learn recursion. They try to see what is happening at **every single** function call and try to trace each step in their solution.

You don't need to know what's happening in every step. If you want to start solving recursion problems, you must be willing to take a leap of a faith. You gotta ***believeee***. Assumptions will need to be made and is necessary for solving these types of problems.

How to do it

First, let's do one of the simplest recursion problems you can ever do.

Problem: Sum all values from 1 to n

```
function sumTo(n) {

}
```

Step 1) Know what your function should do

The first step to solve recursion problems, is to know what your function is suppose to do.

Reader: Do you think I'm an idiot?

This might seem obvious, but it's an important step that gets glossed over. You need to think about what your function ***should do***, not what it ***currently does***.

Looking at our sumTo() function, it's clear that the function ***should*** return an integer sum from 1 to n.

```
/*
  sumTo() takes an integer n and returns the sum of all integers
from 1 to n
*/

function sumTo(n) {

}
```

Step 2) Pick a subproblem and assume your function already works on it

… Subproblems…?

A ***sub problem*** is any problem that is **smaller** than your original problem.

Our original problem is to sum all values from 1 to n. A subproblem in this case is to sum all numbers up to a value ***smaller*** than n.

```
// If sumTo(n) was our original problem, these are all considered
subproblems because they are smaller versions of the original problem

sumTo(n-1)
sumTo(n-2)
sumTo(n-3)
...
sumTo(1)
```

But to make solving your problem easier, we need to select an appropriate subproblem.

Picking an appropriate subproblem

There are many ways to pick a subproblem. A good starting strategy is to choose a subproblem as ***close*** to the original as possible. Since we ***assume*** that sumTo() function ***already works***, why not pick a value that solves the bulk of the problem for us?

In this case, since our problem solves for n, then the best subproblem should solve for n-1.

Using n-1 as our subproblem

```
/*
  sumTo() takes an integer and returns an integer n
  that is the sum from 1 to n
*/

function sumTo (n) { // n is our original problem

  // Using n-1 as our subproblem, it returns the sum from 1 to n-1.
  const solutionToSubproblem = sumTo(n-1)
}
```

But wait! How can we use a function we haven't defined yet ???

You are correct, we have not defined anything yet, but that's what I meant in the beginning of the article. To solve a recursion problem, let's ***ASSUME*** that the function ***already works*** for any subproblem we want.

Because of our subproblem selection, we already have the sum of all values from 1 to n-1. All we need to do now is make that final leap.

Step 3) Take the answer to your subproblem, and use it to solve for the original problem.

We already solved our subproblem. So the next question is …

> **How do we take the solution to our subproblem, and use it to solve the original problem ?**

So far we have solved for 1 to n-1. But how do we use that to solve for n?

```
function sumTo (n) { // n is our original problem

  // Using n-1 as our subproblem, it returns the sum from 1 to n-1.
  const solutionToSubproblem = sumTo(n-1)
}
```

Let's think about our original problem again.

We want to find the sum from 1 to n, and we already have the solution from 1 to n -1.

How do we get the sum from 1 to n, if we have the solution from 1 to n-1?

```
sumTo(n-1) // 1 + 2  ... n-2 + n-1
sumTo(n)   // 1 + 2  ... n-2 + n-1 + n
```

With some basic algebra, all we need to do is add n to the solution of our subproblem, which will solve our original problem.

```
// What we've already determined
sumTo(n-1) // 1 + 2  ... n-2 + n-1
sumTo(n)   // 1 + 2  ... n-2 + n-1 + n

// Using our solution to the subproblem to solve the original
sumTo(n-1) + n  // 1 + 2  ... n-2 + n-1 + n
sumTo(n)        // 1 + 2  ... n-2 + n-1 + n
```

Or in code …

```
function sumTo (n) { // n is our original problem
  const solutionToSubproblem = sumTo(n-1) // n-1 is our subproblem

  return solutionToSubproblem + n
}
```

As you saw, we took the solution to our subproblem and found how it's used to solve the original problem. This is known as finding the **recurrence relation.**

Step 4) You have already solved 99% of the problem. The remaining 1%? Base case.

Your function is calling itself, so it will probably run forever. That is why we need to add a base case to stop it.

What is a base case and how do we determine a base case?

A base case is a way for us to stop the recursion. Usually, it can be a simple if-else statement in the beginning of the function.

The condition prevents more function calls if it has reached its base case. To pick a base case, think of the following.

> What is the ***EASIEST POSSIBLE VALUE*** you can put into the function that requires no extra calculation ?

In our case, that would be n = 0.

It's obvious that the sum of all values from 0 to 0, is 0, so why bother doing more recursion? That's where we define our base case.

```
function sumTo (n) {
  if (n === 0) { return 0 }
  const solutionToSubproblem = sumTo(n-1)

  return solutionToSubproblem + n
}
```

Now that we have a base case. There is now a point where the recursion stops.

THAT'S IT

VICTORY SCREECH !!! — SpongeBob SquarePants S3E1

That's all there is to our answer. In summary, solving the recursion problems involves the following

- Keeping in mind what the function ***should*** do, not what it currently does
- Identifying the proper subproblems
- Use the solution to your subproblems to solve the original problem
- Writing a base case

More examples

Problem: Reverse a string

```
function reverse(s) {

}
```

What ***should*** the function do? The function ***should*** return a reversed copy of a string.

```
// Reverses a string s
function reverse(s) {

}
```

Our problem is to reverse a string s. Let's think of a subproblem that would make solving this problem easy. Let's use "Hello" as an example.

```
const s = "Hello"
```

Again, let's ***assume*** reverse() already works. To make our lives easier, we pick a subproblem that solves the bulk of the problem for us. In this case, let's just call reverse() on everything but the first letter.

```
reverse("Hello") // "Hello" is our original problem"
reverse("ello")  // "ello" can be used as our subproblem
```

So if our subproblem is the original string, without the first letter, what

```
reverse("Hello") // "olleH"
reverse("ello")  // "olle"
```

How do we use our subproblem to solve our original? In this case, we can append the first letter of our original problem to the end of the subproblem solution.

```
// What we determined so far
reverse("Hello") // "olleH"
reverse("ello")  // "olle"

// Using our subproblem to solve the original
reverse("Hello")      // "olleH"
reverse("ello") + "H" // "olleH"
```

Using our example, let's abstract this answer to work for any string s.

```
function reverse (s) {
  const subproblem = s.slice(1, s.length) // exclude first letter
  const reversedSubproblem = reverse(subproblem)

  return reversedSubproblem + s[0]
}
```

Lastly, let's determine our base case. For our problem, what is the ***simplest*** value that we can pass in that ***doesn't need*** extra computation? The answer is the empty string. What is the reverse of an empty string? Well, the empty string of course.

```
function reverse (s) {
  if (s === '') { return '' } // Base case

  const subproblem = s.slice(1, s.length)
  const reversedSubproblem = reverse(subproblem)

  return reversedSubproblem + s[0]
}
```

Problem: Return the nth term in the Fibonacci Sequence

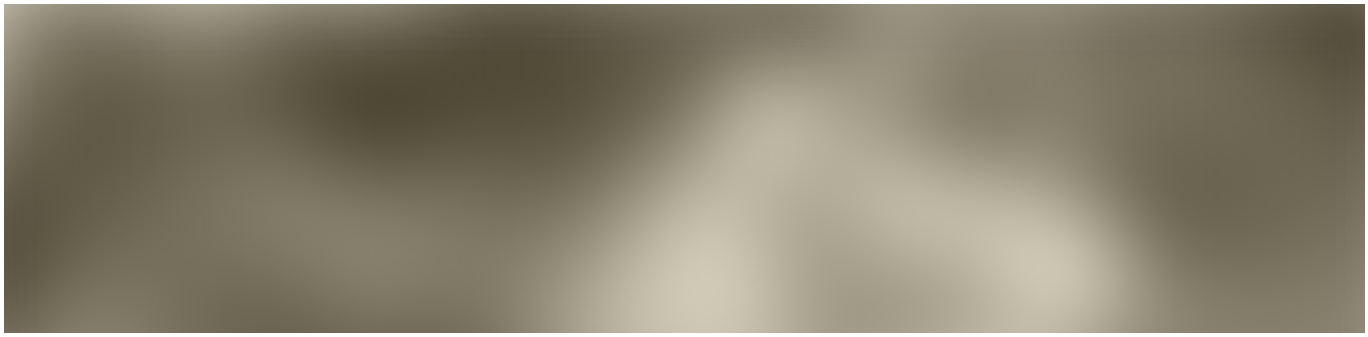

Photo by rolf neumann on Unsplash

Ah yes, the infamous Fibonacci Problem …

```
// Fibonacci Sequence
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

function fibTerm(n) {

}
```

Once you familiarize how the fibonacci sequence works, you will notice that the nth term is the sum of its two ***previous*** terms.

For example …

```
5 is the sum of the two previous values 3 and 2.
34 is the sum of the two previous values 21 and 13.
233 is the sum of the two previous values 144 and 89.
```

Unlike our other problems, our original problem requires us to solve **two subproblems**.

```
// Fibonacci Sequence (using n = 7 as an example)
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...
               ^  ^   ^
             n-2 n-1 n

The nth term, is the sum of the n-1 and n-2 term.
```

Assuming that our function already works, we use it to calculate the two previous terms.

```
// Fibonacci Sequence
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

function fibTerm(n) {
  const term1 = fibTerm(n-1)  // Our two sub problems
  const term2 = fibTerm(n-2)
}
```

To solve our original problem, let's return the sum of our two subproblems.

```
// Fibonacci Sequence
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

function fibTerm(n) {
  const term1 = fibTerm(n-1)
  const term2 = fibTerm(n-2)

  // Solving the original problem using our subproblem solutions
  return term1 + term2
}
```

Finding the base case might be a bit tricky. In our case, we have two base cases, n = 0 and n = 1. The reason is that those values don't have two previous terms to calculate from.

Using n = 0 and n = 1 as our base case, we complete our function.

```
// Fibonacci Sequence
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...
```

```
function fibTerm(n) {
  if (n === 0) { return 0 }
  if (n === 1) { return 1 }

  const term1 = fibTerm(n-1)
  const term2 = fibTerm(n-2)

  return term1 + term2 // Solving the original problem
}
```

C++ / Python Solutions

Problem: Sum from 1 to n

```
// C++
int sumTo(int n) {
  if (n == 1) { return 1 }
  return n + sumTo(n)
}

// Python
def sumTo(n) {
  if (n == 1 ) { return 1 }
  return n + sumTo(n)
}
```

Problem: Reverse a string from 1 to n

```
// C++
string reverse(string n) {
  if (n == '') { return ''}
  return reverse(n.substr(1, n.length()-1 ) + n[0]
}

// Python
def reverse(n) {
  if (n == '') { return ''}
  return reverse(n[1:]) + n[0]
}
```

Problem: Find the nth term of the Fibonacci Sequence

```
// C++
int fibTerm(int n) {
  if (n == 0) { return 0 }
  if (n == 1) { return 1 }
  return fibTerm(n-1) + fibTerm(n-2)
}

// Python
def fibTerm(n) {
  if (n == 0) { return 0 }
  if (n == 1) { return 1 }
  return fibTerm(n-1) + fibTerm(n-2)
}
```