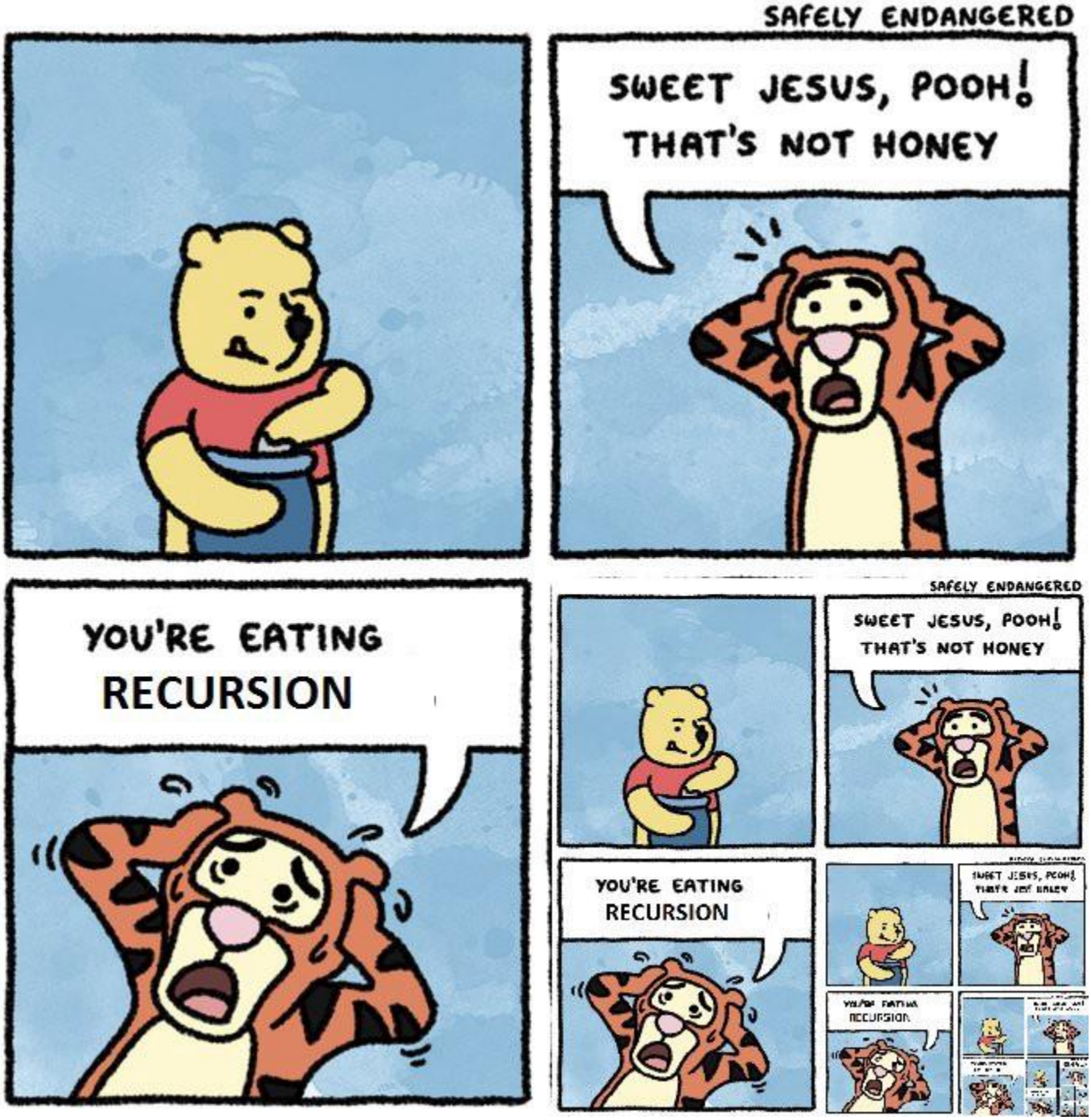
How to Think Recursively | Solving Recursion Problems in 4 Steps

Jack A. Chen May 12 · 9 min read

From: https://medium.com/@passbyvalue/how-to-think-recursively-solving-recursion-problems-in-4-steps-95a6d07aa866



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 answers in both Python and C++ are at 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 …

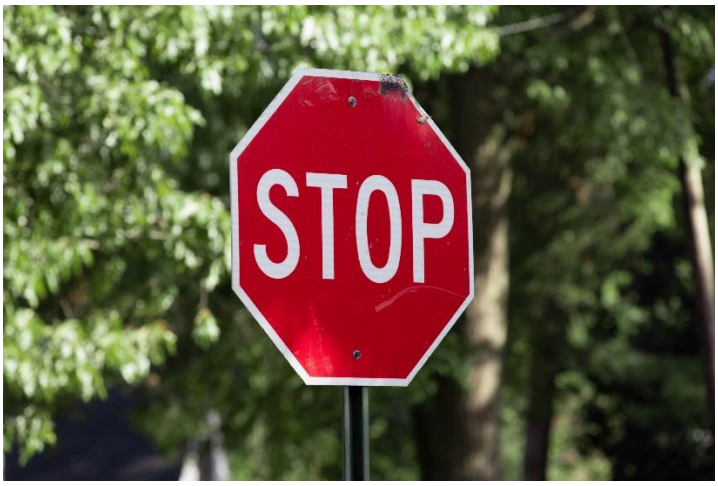


Photo by Will Porada on Unsplash

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. Assumptions will need to be made and is necessary for solving these types of problems.

How to do it

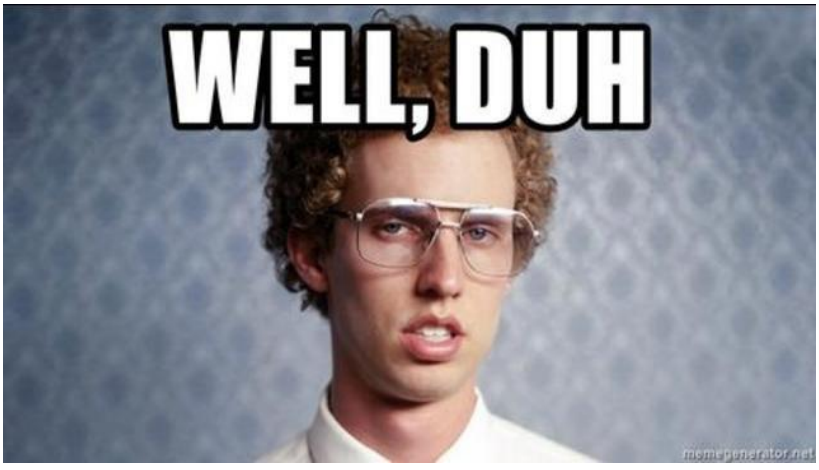
First, let's do one of the simplest recursion problems you can ever do.

Problem: Sum all values from 1 to n

```
function sumTo(n) {}
```

Step 1) Know what your function will 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.** Your function won't be fully defined until the end, so it's important to keep in mind what the function **should do.**

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

```
/*
  sumTo() takes an integer and returns an integer n
  that is the sum 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. Since our original function is called with the argument (n), our original problem is to solve for n. A subproblem in this case is anything less than n.

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, let's pick a subproblem that's close to n.

In this case, our problem needs 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.

Step 3) Take your answer from step 2, and combine 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 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. 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?

```
[Solution to Subproblem n-1]     = 1 + 2  ... n-2 + n-1
[Solution to Original Problem n] = 1 + 2  ... n-2 + n-1 + n
```

All we need to do is add n to the solution of our subproblem, which will solve our original problem.

```
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 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 it that requires no extra calculation ?

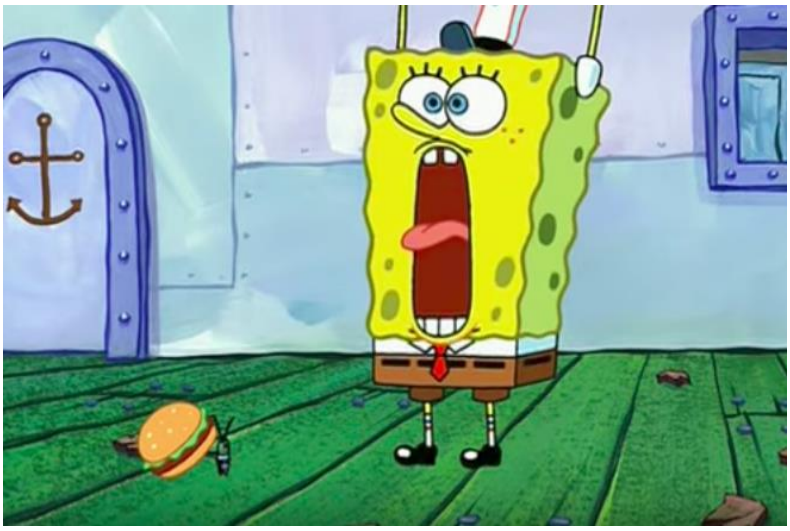
In our case, that would be n = 1. It's obvious that the sum of all values from 1 to 1 is 1, so why bother doing more recursion? That's where we define our base case.

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

  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

- Knowing what the function **should** do
- Identifying the proper subproblems
- Use the solution to your subproblems to solve the original problem
- Writing a base case

Uh… I still don't get it

Fair enough, we can go through some more examples!

Problem: Reverse a string

```
function reverse(s) {}
```

The function reverses takes in a string and returns a reversed copy.

```
// 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.

```
Original Problem s = "Hello"
```

A good subproblem is to reverse all letters of s, except the first one.

```
[Original Problem] s            = "Hello"
[Subproblem] s w/o first letter = "ello"
```

So our subproblem is the substring of s without the first letter.

```
function reverse (s) {
  const substring = s.slice(1, s.length)
  const subProblemSolution = reverse(substring)
}
```

Now, let's take solution to our subproblem and use it to solve the original.

How do we use our solution to the subproblem to solve our original?

```
// Using s = "Hello" as an example and "ello" as our subproblem
[Solution to our subproblem   "ello"]  = "olle"
[Solution to original Problem "Hello"] = "olleH"
```

We can take the first letter of our original solution and add it to the end of the solution to our subproblem.

```
function reverse (s) {
  const subproblem = s.slice(1, s.length)
  const subProblemSolution = reverse(subproblem)  return subProblemSolution +
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

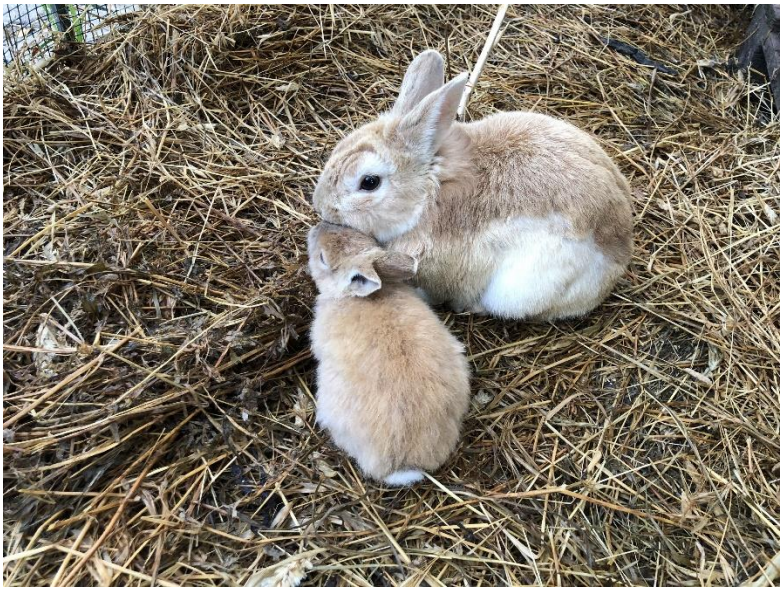


Photo by rolf neumann on Unsplash

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) {}
```

Let's recall that the function should an integer n and returns the nth term in the Fibonacci sequence.

For example ...

```
fibTerm(0) // Should be 0
fibTerm(1) // Should be 1
fibTerm(4) // Should be 3
fibTerm(6) // Should be 21
```

Unlike our other problems, we will need to solve **two subproblems**. Observe that every term in the sequence is the sum of the previous two terms.

```
// 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 nThe 7th term, is the sum of the 6th and 5th term.
```

Thus, our problem requires solving two subproblems, n-1 and n-2.

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

Finding the base case might be a bit tricky. We may think that n = 0 is our base case, but n = 1 is also our base case. The reason is that the fibTerm(1) requires finding fibTerm(0) and fibTerm(-1). As we can see, fibTerm(-1) is nonsense.

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)
}
```

WRITTEN BY Jack A. Chen

Full Stack Engineer @ Digit