Introduction to Recursion

Daniel Liu

October 29, 2018

1 What is recursion?

Recursion is basically a function calling itself. It is best illustrated through an example:

Algorithm 1 Iterative Fibonnaci

```
1: function F(n)
2:     f_1 ← 1
3:     f_2 ← 1
4:     for i ∈ {3 ... n} do
5:         f_i ← f_{i−1} + f_{i−2}
6:     end for
7:     return f_n
8: end function
```

Algorithm 2 Recursive Fibonnaci

```
1: function F(n)
2:     if n = 1 then
3:         return n
4:     else
5:         return F(n − 1) + F(n − 2)
6:     end if
7: end function
```

The iterative variant calculates from the first two Fibonacci numbers, building up to the nth Fibonacci number. This is commonly known as the bottom-up approach. The recursive version defines the result using function calls, which can be thought of as placeholders for values. For many problems, using a recursive approach to reason about it can be beneficial—you can think of it as "asking" for a value that have not been calculated yet.

Now, we obviously cannot get values out of thin air. Each recursive function call defers the computation, layering the function calls until the base state is reached. Then, one by one, the functions start returning their results, going up

the stack of layers until the first function call is reached. As we start our function calls from the top, or in this case, from the original n, this is called the top-down approach. Note that our recursive Fibonacci implementation is *very* slow; we do two recursive calls every layer of the function call chain, leading to exponential growth in the total number of function calls! Without any optimizations, plain recursion is usually quite slow.

The main idea behind recursion is the transfer between states, where each state is *uniquely* represented through a recursive function's parameter(s). For our Fibonacci number example, each state can just be represented by the index n. For harder problems, the state can be much more complex.

2 A Very Recursive Problem

Alice recently got a n (let's say 20 for now) digit combination lock with a very peculiar property: only 1s and 0s are allowed as the lock combinations! Also, she only likes combinations with an odd number of 1s. Help her find out how many different combinations are possible![1]

This problem is fairly straightforward in terms of recursion—just go through each digit and choose either 1 or 0. At the very end, we check the n digit sequence to see if it has an odd number of 1s. Each state can be represented with an array and an index representing where we are in the array. For every step of the recursion, we advance the index and try adding both 1 and 0 to the array at the index.

Algorithm 3 Recursively generating combinations

```
1: r ← 0
2: n ← 20
3: function GENERATE(a, i)
4:     if i = n then
5:         if a has an odd number of 1s then
6:             r ← r + 1
7:         end if
8:     else
9:         GENERATE(a + [0], i + 1)
10:        GENERATE(a + [1], i + 1)
11:    end if
12: end function
13: GENERATE([], 0)
```

With an iterative implementation, n (in this case, 20) layers of for loops are required, which is quite a lot of code.

Since we branch into two different paths every single recursive call, and the maximum recursion depth is n, the time complexity should be some multiple of

[1] Please use recursion, not math.

$O(2^n)$. As we have to go through each combination to check if it has an odd number of 1s, the final complexity is $O(n2^n)$.

3 Analyzing Recursion: Merge Sort

Merge sort is a recursive searching algorithm that runs in $O(n \log n)$ time for an array of length n. It is important to note that merge sort uses the divide and conquer paradigm, and it does not run in exponential time, unlike many pure recursive algorithms.

Algorithm 4 Recursive merge sort

```
1: function MergeSort(a)
2:     if |a| = 1 then
3:         return a
4:     else
5:         l ← MergeSort(a_{0,...,|a|/2 − 1})
6:         r ← MergeSort(a_{|a|/2,...,|a|−1})
7:         a ← Merge l and r in O(|l| + |r|) time, sorted.
8:         return a
9:     end if
10: end function
```

Why is the time complexity $O(n \log n)$? Let's first estimate the recursion depth. Since we split the array up every single recursive function call, the recursion depth is $O(\log n)$. When merging two sorted subarrays into a larger sorted array, the time required is linear to the total number of elements in both arrays. If we group all recursive function calls with the same depth into a single layer, then the time complexity per layer is $O(n)$. Since we have $O(\log n)$ layers, the total time complexity is $O(n \log n)$.