The wonders of recursion (...and more)!

Jatin Abacousnac

October 2023

1 Recursion

We start with the lovechild of mathematical induction and algorithm: recursion. Think of nested Russian dolls, of trippy fractal videos with self-repeating structures, think of broccoli. Huh?

Recursion is the act of embedding a process within itself. The innermost layer of recursion is called the base case, and is the one that needs to be addressed first, even though the base case is the computation that will be performed last. The classic example of how to go about doing recursion is the factorial example. How do you compute the factorial of n? If you've never heard of recursion, you might approach the problem this way:

```python
def factorial(n):
    """factorial with for loop"""
    product = 1
    for i in range(1, n+1):
        product *= i
    return product
```

It will do the job.

Now, let's do it recursively. This is the magical part. You start by realizing that

$$n! = n(n-1)! \tag{1}$$

So, if you can write the function for $n-1$, then you can write it for n. Likewise, if you can write it for $n-2$, you can write it for $n-1$, etc. The complication arises when $n = 0$ because the factorial of a negative number is not defined, and that becomes our base case.

```python
def factorial(n):
    """factorial with recursion"""
    if n < 2:
        return 1
    else:
        return n*factorial(n-1)
```

Ta-da! This was awesome, and the code looks concise/elegant/[insert other pleasant adjective]. You might automatically think that recursion is the solution to all of your problems. Sadly, that is not the truth because these computations come at a heavy cost. There are situations where recursion might actually help save computational time, but this is not one of those. Given that at every step of the recursion a function call is made, doing recursive computation in this example is extremely time intensive. Can we save time and use recursion? We can improve it so every time it does the factorial calculation, it does a speedier job.

2 Memoization

What if I gave my function a semblance of memory, and it wouldn't need to recompute things it's done before? That would save a whole lot of time, wouldn't it? Here we go!

```python
factorial_dict = {} #a dictionary which will save my results
def factorial(n)
    """factorial with recursion + memoization"""
    if n < 2:
        return 1
    else:
        factorial_dict[n] = n*factorial(n-1)
        return factorial_dict[n]
```

Of course, the caveat is that the first time it is run, it won't be faster than the un-memoized version. The second time you run it for some large n, with the dictionary having been populated with the factorials of numbers $\leq n$, the computation will speed up! Oh, another caveat: there is a trade-off between speed and memory. Powered by this knowledge, you might want to attack recursive problems with recursion (and maybe memoization, as well). However, like I've hinted previously, it's not always the greatest idea in the world. Let's use the darling of all sequences as an example: the Fibonacci sequence. The Fibonacci sequence goes as 1, 1, 2, 3, 5, 8, 13, etc. The recursive relation is

$$F_n = F_{n-2} + F_{n-1}. \tag{2}$$

Using recursion,

```
def fibonacci(n):
"""fibonacci with recursion"""
    if n < 3:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Very good. If we wish to use memoization,

```
fibonacci_dict = {}
def fibonacci(n):
"""fibonacci with recursion + memoization"""
    if n < 3:
        return 1
    else:
        fibonacci_dict[n] = fibonacci(n-1) + fibonacci(n-2)
        return fibonacci_dict[n]
```

This should be faster. But it turns out doing it the so-called stupid way, with *for* loops is the smarter way. It even has a name.

3 Bottoms-Up Approach

```
def fibonacci(n):
"""fibonacci with bottoms-up approach"""
    the_list = [1,1]
    for i in range(2, n):
        the_list.append(the_list[i-1] + the_list[i-2])
    return the_list[-1]
```

Even though you might look at this and go "oh! that was trivial!," it is important to note how this approach is more suitable for the fibonacci problem, and how not having to make function calls is an advantage. So when does recursion even help? It helps when you are invited to a party and someone invites you to play Towers of Hanoi.

4 Towers of Hanoi

I love this game.

Say we have n disks and 3 pegs, with disk 1 being the smallest, disk n being the largest, etc. At the start, the pegs are sorted (biggest on the bottom, smallest on top) and placed on the leftmost peg (call it **A**). The goal is to transfer all the disks from **A** to the rightmost (**C**). At no point is a bigger peg allowed to be stacked upon a smaller one. The middle peg is labeled **B**. We want to achieve this goal optimally.

The key is to realize that disk n has to be moved to peg **C**, eventually. The only preceding state that allows that is for peg **C** to be empty, and for peg **B** to contain disks 1 through $n-1$. This is true for every n. So, if you wanted to solve for $n = 4$, you'd need disks 1, 2, and 3 to find themselves in peg **B**. If you were solving for $n = 3$, you'd need to stack disks 1 and 2 in peg **B**, etc. It turns out that it will take half the number of moves to get to that point, with the next move being the one where the largest disk is moved. The second key point to realize is that for the first half of moves, the biggest disk is as good as non-existent. So, the first half of the problem goes from being an n problem to an $n-1$ problem. Now, if we are reduced to the $n-1$ problem, then we would need to stack disk 1 through $n-2$ in peg **C** to solve the first half problem (enabling disk $n-1$ to get to **B**). Which is only possible if disks 1 through $n-3$ find themselves on peg **C**. See the recursion?

The n problem can be solved by solving the $n-1$ problem (with the caveat that each preceding problem be solved on an alternating peg). In terms of the total number of moves T_n,

$$T_n = 2T_{n-1} + 1 \tag{3}$$

$T_1 = 1, T_2 = 3, T_3 = 7, ...$ Generally,

$$T_n = 2^n - 1. \tag{4}$$

All of that is fantastic. So how would we solve it recursively on a computer? First the base case: if we only have 1 disk, it obviously goes straight to **C**. Since each preceding problem is solved on the alternative peg, we switch pegs **B** and **C** for the $n-1$ case. When the $n-2$ recursive call will be made, **B** and **C** will be back to the original positions as in the n case. But that only solves half the problem. The remaining half is solved by symmetry by noticing that all the moves in the first half can be replicated by now switching pegs **A** and **B**, and that gets us to our goal.

```python
def toh(n, A, B, C):
"""recursive implementation of towers of hanoi with 3 pegs"""
if n == 1:
    print(f"disk 1: {A} -> {C}") #base case
else:
    toh(n-1, A, C, B) #solve for the first half n-1 solution
    print(f"disk {n}: {A} -> {C}")
    toh(n-1, B, A, C) #use symmetry to flip first half n-1 solution
```

The output for the $n = 4$ case will look like

```
disk 1: A -> B
disk 2: A -> C
disk 1: B -> C
disk 3: A -> B
disk 1: C -> A
disk 2: C -> B
disk 1: A -> B
disk 4: A -> C
disk 1: B -> C
disk 2: B -> A
disk 1: C -> A
disk 3: B -> C
disk 1: A -> B
disk 2: A -> C
disk 1: B -> C
```

5 Divide and Conquer algorithms

There is a famous puzzle which goes as follows: There are 25 horses, and 5 racetracks. At most, 5 horses can race on a track. What is the minimum number of races it takes to determine which of the 25 horses are the fastest three?

Sol: We want a sorted list of horses (except that we only care about the top 3 elements). The way to solve it is to make as few comparisons as we can. Step 1 is the most natural way to proceed: divide the group of horses into 5 groups: call them A, B, C, D, E, and make each group race, one at a time. So far, 5 races. The winners from

A-E are worth considering. But for all we know the top three all took part in the same race. So, we end up with 15 horses that ought to be considered. Now take the top 5 and make them race one another. This is a pivotal race (let's call it the semi-final) because out of the 15 remaining horses, it takes 6 horses immediately out of contention, the 6 being the ones which originally raced with the two horses which ended up being last in the semi-final. Now, clearly the horse who won the semi is the fastest of them all. Let's call this horse Bojack. The 2nd and 3rd placed horse could well be the 2nd and 3rd fastest overall, but it could also be that the two horses which ended up being behind Bojack in his very first race should be in contention as well. The final contending horse will be the one which ended up being behind the 2nd placed horse in the semi. The final race takes place among these 5 contending horses, and the top two are automatically 2nd and 3rd. Answer: 7 races.

While we won't be attempting to solve this problem as efficiently as possible, it does offer some insight into solving similar problems.

5.1 Quicksort

The idea is simple. Say we have a list of integers that need to be sorted. We solve the problem by sorting sublists of this giant list. How are these sublists created? The first step is the designation of one of the elements as the pivot (choose the middle element, for instance). Any comparison will be made with regards to the pivot. One of approaching this problem is to create 3 sublists: one with values < pivot, one with values > and another with values equal.

```
def quicksort(the_list):
"""quicksort (recursive)"""
if len(the_list) <= 1:
    return the_list
else:
    pivot = the_list[len(the_list)//2] #middle element is pivot
    l = [i for i in the_list if i < pivot]
    r = [i for i in the_list if i > pivot]
    m = [i for i in the_list if i == pivot]
    the_list = quicksort(l) + m + quicksort(r)
    return the_list
```

5.2 Mergesort

Yeah yeah yeah I get it. Now that I mentioned quicksort, it's time for mergesort. Here, the idea is to split a list into two. That smaller sublist is also split into two, and this is done until we can no longer split. Then, the two adjacent sublists each of element 1 are compared to one another and merged into a sublist of length 2 where the smaller element is to the left and the larger to the right. We keep doing that for progressively larger merged sublists recursively.

```
def mergesort(the_list):
"""mergesort (recursive)"""
    if len(the_list) <= 1:
        return the_list
    else:
        m_index = len(the_list)//2 #the middle index
        l = mergesort(the_list[:m_index])
        r = mergesort(the_list[m_index:])
        return do_merge(l, r)

def do_merge(l, r):
"""mergesort helper function where it all happens"""
    merged = []
    i, j = 0,0
    while (i<len(l)) & (j<len(r)):
        if l[i]<r[j]:
            merged.append(l[i])
            i+=1
        else:
            merged.append(r[j])
            j+=1
    merged.extend(l[i:])
    merged.extend(r[j:])
    return merged
```