Recursively Defined Functions

A recursive defintion of function $f(\cdot)$, defines a value of function at some natural number n in terms of the function's value at some previous point(s).

Example 1. *Consider the* **fibonacci function** $F : \mathbb{N} \to \mathbb{N}$ *defined as follows:*

$$F(n) = \begin{cases} 0 & \textit{if } n = 0 \\ 1 & \textit{if } n = 1 \\ F(n-1) + F(n-2) & \textit{if } n > 1 \end{cases}$$

Notice that if we drop any of the conditions in the definition of fibonacci function, we end up with an improper definition that does not allow us to find the value of F at each natural number n.

For some of the recusively defined functions, we can find a closed-form formulas. In some cases, the closed-form formulas can be found by doing an initial exploration and then an educated guess. However, one cannot often rely on educated-guesses and needs more powerful tools (more on this later). The constructive nature of recursive defintions allows us to use induction to prove the equivalence of the the recursive definition and the closed-form formula.

Example 2. *Find a closed-form formula for the following recursively defined function.*

$$F(n) = \begin{cases} 1 & \textit{if } n = 1 \\ 1 + F(n-1) & \textit{if } n > 1 \end{cases}$$

Ans:
Repeated substitution:

$$\begin{aligned} F(n) &= 1 + F(n-1) \\ &= 1 + 1 + F(n-2) \\ &= 1 + 1 + 1 + F(n-3) \end{aligned}$$

Educated Guess: If we continue our substitution, a pattern starts to emmerge. After i substitutions $F(n) = i + F(n-i)$. If we check the validity of this claim using $i = 1$, we can see that it gives us the definition of the function F. But this is only guess, not proof.
Proof: We will prove our guess by induction. Formally, we will prove that $F(n) = n, \forall n \geq 1$.

Proof. Base Case: $n = 1$. By definitionn, $F(1) = 1 = n$.
Induction Step: Let $n > 1$ and assume $F(k) = n$ for $1 \leq k < n$ (IH).

$$F(n) = 1 + F(n-1) \qquad \text{(by recurrence since } n > 1) \qquad = 1 + n - 1 \qquad \text{(by IH since } 1 \leq n-1 < n) = n$$

Hence, by induction, $F(n) = n$ for all $n \geq 1$. □

□

Exercise. *Let $F(n)$ denote the fibonacci function. For any $n \in \mathbb{N}$, prove $F(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$, where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = \frac{1-\sqrt{5}}{2}$.*

Recursively defined functions are used to define the complexity (worst-case running time) of recursive algorithms.

Example 3 (Factorial)**.** *Consider the following recursive algorithm:*

```
function FACT(n)
    if $n == 0$ or $n == 1$: then
        return 1
    else
        return $n \times$ Fact(n-1)
    end if
end function
```

The Worst-case running time of factorial satisfies the following recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 0 \\ 1 + T(n-1) & \text{if } n > 1 \end{cases}$$

Can you spot the relation between the $T(n)$ and $F(n)$ in Example 2? What is the closed-form formula?

Example 4 (Binary search algorithm)**.** *Consider the following recursive implementation of binary search algorithm:*

```
function RECBSEARCH(x, A, s, f)
    if $s == f$ then
        if $x == A[s]$ then
            return s
        else
            return $-1$
        end if
    else
        $m = (s + f)\ /\ 2$                                    ▷ Integer Division
        if $x \le A[m]$ then
            return RecBSearch(x, A, s, m)
        else
            return RecBSearch(x, A, m + 1, f)
        end if
    end if
end function
```

We will discuss the correctness of this algorithm later in the course.

Now, Let's define the worst-case running time of RecBSearch over all inputs of size $n = f - s + 1$. (Why the "+1"? Otherwise size = 0 when $s = f-$ even though there is 1 element in that case!)

line 2, lines $3-4$, lines $5-6$, line 9 and line 10 are chunks of code. How about lines 11 and 13? These lines are recursive calls and hence has a worst-time complexity of: $T(\lceil \frac{n}{2} \rceil)$ steps for line 11 and $T(\lfloor \frac{n}{2} \rfloor)$ steps for line 13. Hence, $T(n)$ satisfies recurrence:

$$T(1) = 2$$
$$T(n) = 3 + \max\{T(\lfloor \frac{n}{2} \rfloor), T(\lceil \frac{n}{2} \rceil)\},\ n > 1$$

This equation doesn't seem like the kind of recursion that we have seen so far. Can we Simplify? If we knew T is increasing (i.e., $\forall x \le y,\ T(x) \le T(y)$)) then we could say $T(\lfloor \frac{n}{2} \rfloor) \le T(\lceil \frac{n}{2} \rceil)$ so $T(n) = 1 + T(\lceil \frac{n}{2} \rceil)$. But this would require proof!

Let's derive a closed form formula. Having a closed form formula may answer the validity of our assumption regarding $T(n)$. Assuming that T is strictly increasing, we can have a simplified recurrence for worst-case runtime as follows:

$$T(n) = \begin{cases} 2 & \text{if } n = 1 \\ 3 + T(\lceil \frac{n}{2} \rceil) & \text{if } n > 1 \end{cases}$$

Let's follow the steps of Example 2.
Repeated substitution: For now, we make the simplifying assumption that we can ignore ceilings so that we can come up with an educated guess.

$$\begin{aligned} T(n) &\approx 3 + T(\frac{n}{2}) \\ &\approx 3 + 3 + T(\frac{n}{4}) \\ &\approx 3 + 3 + 3 + T(\frac{n}{8}) \end{aligned}$$

Educated Guess: We can guess that after i substitutions $T(n) \approx 3i + T(\frac{n}{2^i})$. In the limit

$$\frac{n}{2^i} = 1 \Leftrightarrow n = 2^i \Leftrightarrow \log_2 n = i$$

If we substitute this in our formula:

$$T(n) \approx 3 \log_2 n + T(1) = 3 \log_2 n + 2.$$

This cannot be the exact solution, because of the simplifications that we made and also the fact that $T(n)$ is always an integer. But we expect $T(n) \in \Theta(\log n)$ (for exact value $T(n)$).
Proof: In order to prove that $T(n) \in \Theta(\log n)$, we have to prove that:

1. $T(n) \in \Omega(\log n)$, which means that we have to find constants $B, c > 0$ such that $T(n) \geq c \log n \; \forall n >= B$.
2. $T(n) \in O(\log n)$, which requires us to find constants $B, c > 0$ such that $T(n) \leq c \log n \; \forall n \geq B$.

The actual proofs will be done in the tutorial.