Lecture 5: Anatomy of Recursion, Induction

built on 2019/05/06 at 15:31:59

In addition to loops, we have seen that repetition can be achieved using recursion. Today is the first lecture in the sequel to better understand recursion. We'll see how a recursive function is run and start thinking about how we can reason about recursive programs.

1 Recursion

Simply put, *recursion* is a technique where a function makes one or more calls to itself. In nature, we can look to fractal arts for inspiration. In computing, recursion provides an important and expressive mechanism for performing and reasoning about tasks that contain certain repetition.

We'll begin with two examples that illustrate the use of recursion.

1.1 The Factorial Function

As a first we'll, example consider the factorial function to demonstrate the mechanics of recursion. The factorial function, denoted by $n!$, is given by the product of integers from 1 to n, i.e., $n! = 1 \times 2 \times 3 \times ... \times n$ with $0! = 1$. This means $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. This function has an important physical interpretation as it represents the number of ways one can arrange n distinct items into a sequence. For example, there are 3! ways to arrange `a`, `b`, and `c` into a sequence:

abc, acb, bac, bca, cab, cba.

As it turns out, this function can be expressed in a recursive manner quite naturally. To see this, let's write out the expressions for 4! and 3! and observe their relationship: $3! = 3\times2\times1$ and $4! = 4\times3\times2\times1 = 4\times(3\times2\times1) = 4\cdot3!$

Therefore, we can write the recursive function recursively (i.e., one which calls itself) as follows:

```
int fac(int n) {
  if (n==0) return 1;
  else return n * fac(n-1);
}
```

```
def fac(n):
    if n==0: return 1
    else: return n*fac(n-1)
```

Notice that our implementation achieves repetition without using any loops. It does so by means of function calling. Importantly, each time the function calls itself, the argument is made smaller by one, so there's no circularity. Hence, it should terminate eventually.

We can run this function and observe that it gives the results we expect; however, to understand how they actually work, we want to be able to visualize what's going on. An important tool in this case is a *recursion trace* diagram. It is best to describe this with an example. Let's see the recursion trace for `fac(5)`.

1.1.1 How this actually works?

At this point, you may be wondering how the computer keeps track of where it is. After all, there are so many "incarnations" of `fac`. The answer is, it keeps a stack: Each time a function is called, all of its parameters and variables are put on a new plate and placed on top of the stack. As it continues the execution, the code might call more functions (more plates on the stack) or return. At the point of return, the top-most plate—at the top of the stack—is removed and the program at the (now) top of the stack continues to run.

1.2 English Ruler

Our first example was a nice, well-behaved mathematical function; however, when it comes to programming, there is no reason a priori why anyone would want to write factorial recursively, over say, a simple for-loop.

The second example is concerned with drawing marks on a ruler where the solution is naturally recursive; doing it otherwise would, in fact, be more cumbersome.

For the purpose of this lecture, we'll focus on drawing the scale from 0 to 1 (excluding the 1 tick):

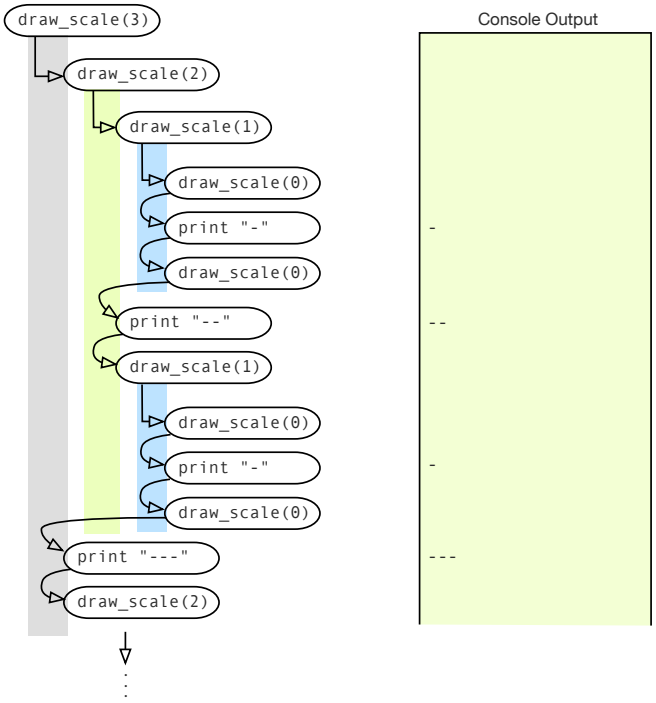
```
---- 0
-
--
-
---
-
--
-
---- 1
.
.
.
---- n
```

Here is one way to produce it using recursion:

```
---- 0
-
--
-
---
-
--
-
```

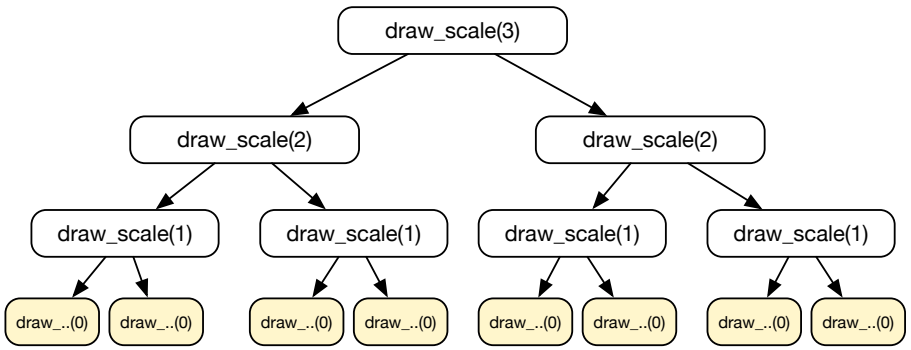
```
def draw_scale(tick_length):
    if tick_length>0:
        draw_scale(tick_length-1)
        print "-"*tick_length
        draw_scale(tick_length-1)

print ("-"*4), "0"
draw_scale(3)
```



It helps to understand how this function actually works. Because each `draw_scale(.)` call makes two calls to itself, the flow of this program is substantially more complex than that of factorial. To right of the code shows a trace of the first few steps of how this function is called starting with `draw_scale(3)`. Notice how position in the code is remembered and resumed right after the execution flow returns from an inner call.

To get a more holistic view of what's happening, we'll use a tree diagram to illustrate the run. The top most node is where this recursive execution begins. As each arrow indicates a call, we know that the top node makes two calls, each making two further calls, etc. etc.



2 Proof and Proof by Induction

What's a proof? To explain what a proof is, we start by talking about a statement. For example, one may claim that every number greater than 1 is divisible by a prime number. To demonstrate that such a statement is true, we give a proof. Simply put then, a proof is a convincing demonstration that an assertion we make is true.

Often we work with universally-quantified statements such as

- For all n and $x \neq 1$, $x^n - 1$ is divisible by $x - 1$.
- For every input `lst`, the function `prod(lst)` returns the product
  $$\texttt{lst}[0] \times \texttt{lst}[1] \times \cdots \times \texttt{lst}[n-1]$$
  where $n = \texttt{len}(\texttt{lst})$.

To prove such a statement, we've seen in Discrete Math that we can resort to

1. direct proof. let the input be what it is and attack it head on.
2. proof by contradiction. assume for a contradiction that this is false. But then blah blah blah and the world is doomed. So it must have been true.
3. proof by induction. assume the small ones are true, argue that if it's true for n, it must be true for $n+1$, and conclude all must be true.

In this course, we'll most often deal with induction, which we'll now revisit.

2.1 Mathematical Induction

Mathematical induction is one of the most basic forms of induction, yet a very powerful technique. In the simplest formulation, it is used to prove a certain property over natural numbers $0, 1, 2, \ldots$. In other words, we use it to show that a property $P(n)$—parameterized by n—holds for all non-negative integers $n \geqslant 0$.

> One important point about the predicate $P(\cdot)$ is that it is a function that returns a Boolean value **True** or **False**. Please don't take $P(n)$ and add it to a number.

To prove this, we accomplish it in two steps:

1. **Base Case.** We show that $P(0)$ holds—the property holds for 0.
2. **Inductive Step.** We then assume that the property holds for $n-1$ and establish that it holds for n.

Let us try to understand why these steps imply that $P(0), P(1), \ldots$ hold *ad infinitum*. First, the base case indicates that $P(0)$ is true. Now here's the critical idea. The reasoning in the inductive step gives us the following:

For any $n > 0$, if $P(n-1)$ is true, we know $P(n)$ will be true.

And remember that this holds for any $n > 0$. Therefore, this means that with $n = 1$, the statement reads:

If $P(0)$ is true, we know $P(1)$ will be true.

With $n = 2$, the statement reads:

If $P(1)$ is true, we know $P(2)$ will be true.

We can keep doing this for all $n > 0$. But what does this actually mean in terms of the truth of the property $P(n)$?

Let's recall that $P(0)$ is true (justified by the base case). But we know that "If $P(0)$ is true, then $P(1)$ will be true." Together with the knowledge that that $P(0)$ is true, this means $P(1)$ is also true. Thus far, we know $P(0)$ *and* $P(1)$ are both true.

Okay, we know that $P(1)$ is true. But we also know that "If $P(1)$ is true, then $P(2)$ will be true." Therefore, we have $P(2)$ is true.

This "chain reaction" goes on infinitely. Therefore, $P(n)$ holds for all $n \geqslant 0$.

There are small variations of this process, where we start the base case at 1, or where there are two base cases 0 and 1. This depends on the nature of the property we want to prove. However, if we start the base case at 1, we can only conclude that $P(1), P(2), \ldots$ hold.

Let's look at a few examples. In these examples, we are intentionally excessively pedantic with the goal of pointing out to you various features that you should think about or question them with great skepticism (when you read or write proof on your own).

2.2 Example I: Summation

A few lectures ago, we saw this formula $1 + 2 + 3 + \cdots + n = n(n+1)/2$. As our first example on induction, let's show that this formula is indeed true. In particular, we'll prove the following claim:

Claim 2.1 *For all integer $n \geqslant 1$, the summation*

$$1 + 2 + 3 + \cdots + n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

To proceed, we'll sketch a plan as follows. Since n is universally-quantified and a natural number, it seems natural to induct on n. But first, we need a predicate $P(\cdot)$ that we'll use for induction. We'll try the simplest one first—just repeat the claim.

$P(n) \equiv$ "$1 + 2 + 3 + \cdots + n = n(n+1)/2$."

Why is the value of $P(1)$? Notice how, as defined, when we plug in a value of n into $P(n)$, we expect this function to return True or False—not a number or anything else.

Base Case. The smallest n we claim is $n = 1$, so let's check that $P(1)$ is indeed true. For this to be true, we must show that the left-hand side (LHS) of the equation is equal to the right-hand side (RHS). When $n = 1$, the LHS is 1 by itself. The RHS is $1(1+1)/2 = 1$. Hence, LHS is equal to RHS, and so $P(1)$ has been verified.

Inductive Step. We have verified the base case(s) up to $n = 1$. Therefore, we'll use the inductive step to show it for $n > 1$. Let $n \geqslant 2$. We will assume $P(n-1)$ is true and show that under this assumption, $P(n)$ is also true. By this assumption (that is, $P(n-1)$ is true), we know that

$$1 + 2 + 3 + \cdots + (n-1) = \frac{(n-1)n}{2}. \tag{2.1}$$

Like in the base case, to show that $P(n)$ holds, we need to verify that LHS equals RHS for the current n. Let's look at LHS first. On the left-hand side of the equation, we have $1 + 2 + 3 + \cdots + (n-1) + n$—the first n positive numbers. On the right hand side, we have $n(n+1)/2$.

But consider that LHS can be written as

$$\begin{aligned}
1 + 2 + 3 + \cdots + (n-1) + n &= \Big[1 + 2 + 3 + \cdots + (n-1)\Big] + n \\
&= \frac{(n-1)n}{2} + n && \text{[IH implies (2.1)]} \\
&= \frac{(n-1)n}{2} + \frac{2n}{2} && \text{[algebra]} \\
&= \frac{n(n-1+2)}{2} = \frac{n(n+1)}{2} && \text{[algebra]}
\end{aligned}$$

which is equal to RHS, as we wish. Therefore, we conclude that $P(n-1) \implies P(n)$.

Conclusion: Having shown the base case and the inductive step, we conclude using the principle of mathematical induction that $P(n)$ holds for all $n \geqslant 1$, hence proving the claim.

2.3 Example II: Divisibility

Let's prove one of the statements we mentioned earlier using induction:

Claim 2.2 *For all natural numbers $n \geqslant 0$ and $x \neq 1$, $x^n - 1$ is divisible by $x - 1$.*

To proceed, we'll sketch a plan as follows. Since n is universally-quantified and a natural number, it seems natural to induct on n. But first, we need a predicate $P(\cdot)$ that we'll use for induction. We'll try the simplest one first—just repeat the claim.

$P(n) \equiv$ "for any $x \neq 1$, $x^n - 1$ is divisible by $x - 1$."

Once again, notice how, as defined, when we plug in a value of n into $P(n)$, we expect this function to return True or False—not a number or anything else.

Now, to apply induction, we just need two more things:

Base Case. We'll check that $P(0)$ is true. For this to be true, we need to verify that when $n = 1$, for any $x \neq 1$, $x^n - 1$ is divisible by $x - 1$. But this is easy to see:

$$x^n - 1 = x^0 - 1 = 1 - 1 = 0 = 0(x - 1),$$

so $x^0 - 1$ is divisible by $x - 1$.

Inductive Step. Let $n \geqslant 1$. For this step, we will assume $P(n-1)$ is true and show that under this assumption, $P(n)$ is also true. By this assumption (that is, $P(n-1)$ is true), we know that $x^{n-1} - 1$ is divisible by $x - 1$. In other words, we can choose a number β such that

$$x^{n-1} - 1 = \beta(x - 1). \tag{2.2}$$

To conclude that $P(n)$ holds, we need to show that $x^n - 1$ is divisible by $x - 1$. *But is this true?* We don't yet know. We do know, however, that $x^n - 1$ can be written (by algebra) as

$$x^n - 1 = x^{n-1} \cdot x - 1 = x^{n-1} \cdot x + 0 - 1 = x^{n-1} \cdot x \underbrace{-x + x}_{=0} - 1 = x(x^{n-1} - 1) + (x - 1).$$

Therefore, we know that

$$\begin{aligned} x^n - 1 &= x(x^{n-1} - 1) + (x - 1) && \text{[algebra, above]} \\ &= x\big[\beta(x - 1)\big] + x - 1 && \text{[IH implies (2.2)]} \\ &= (x - 1)(\beta \cdot x + 1) && \text{[algebra]} \end{aligned}$$

But this shows that $x^n - 1 = \gamma(x-1)$ where $\gamma = \beta \cdot x + 1$ is a number, so we conclude that under the assumption of $P(n-1)$, $x^n - 1$ is divisible by $x - 1$.

Conclusion: Having shown the base case and the inductive step, we can conclude using the principle of mathematical induction that $P(n)$ holds for all $n \geqslant 0$, hence proving the claim.

3 Induction and Algorithms

Let's move on to a completely different example.

Theorem 3.1 *Given an unlimited supply of 5-cent stamps and 7-cent stamps, we can make any amount of postage at least* 24 *cents.*

At this point, you might ask *how?* The answer, as we'll see, lies in decoding our proof of this theorem. Although it can well be proved using a contradiction argument, we'll march ahead with an inductive proof to demonstrate a few features.

Even using induction, there are surely many routes to prove this theorem. We'll look at one which is simple but rather elucidating.

How do we get started? We can first think about what we would actually do if we had to make n cents in postage. For instance, we could try to use one 5-cent stamp and try to make the rest $n - 5$ in postage. The theorem says that we can make any amount as long as it's at least 24 cents. So if $n - 5 \geqslant 24$ (that is, $n \geqslant 29$), we are set—as long as the theorem holds true. (So far we don't yet know how to make the remaining $n - 5$, but so what! we know it can be done, by the theorem.) At this stage, the remaining considerations are for $n = 24, 25, 26, 27, 28$—since we already know how to do it for $n \geqslant 29$.

Do we know how to make $n = 24, 25, \ldots, 28$? Let's try them! A moment's thought shows that

- $n = 24$ is $7 + 7 + 5 + 5$.
- $n = 25$ is $5 + 5 + 5 + 5 + 5$.

- $n = 26$ is $7 + 7 + 7 + 5$.
- $n = 27$ is $7 + 5 + 5 + 5 + 5$.
- $n = 28$ is $7 + 7 + 7 + 7$.

Let's structure this as an inductive proof:

Proof: Consider $P(n) \equiv$ "We can make n cents in 5- and 7- cent stamps.". We want to show $P(n)$ for all $n \geqslant 24$ using induction.

Base Cases. For the base cases, as outlined, we want to show that we can do $n = 24, 25, 26, 27$, and 28. To prove this, we show that

- $n = 24$ is *two* 7-cent stamps and *two* 5-cent stamps, proving $P(24)$.
- $n = 25$ is *five* 5-cent stamp, proving $P(25)$.
- $n = 26$ is $7 + 7 + 7 + 5$, proving $P(26)$.
- $n = 27$ is $7 + 5 + 5 + 5 + 5$, proving $P(27)$.
- $n = 28$ is $7 + 7 + 7 + 7$, proving $P(28)$.

Inductive Step. Let $n \geqslant 29$ be given. Assume that for any integer $24 \leqslant k < n$, $P(k)$ is true. Then, we know that for this n,

$$24 \leqslant n - 5 < n \implies P(n-5) \text{ is true}$$

This means that "we can make $n - 5$ cents in 5- and 7- cent stamps." We don't know just how, but we know it can be done by the inductive hypothesis. By adding an extra 5-cent stamp, we can make n cents in 5- and 7- cent stamps, proving $P(n)$. ■

Notice that: we really need this many base cases because otherwise there would be n in the inductive step where $n - 5$ falls nowhere.

In general, the cases in an inductive proof always fall into two categories: base case(s) or inductive case. If there's an n that isn't covered by these, your proof is flawed.

If we think about inductive proofs in this way, we'll see that this is very similar to recursive programs. There are cases which we handle recursively and there are cases which we handle directly.

For this particular problem (and for many others), the mapping is almost immediate. We can basically read off what the proof says and translate that into code.

```
List<Integer> change(int n) {
  switch (n) {
    case 24: return Arrays.asList(7,7,5,5);
    case 25: return Arrays.asList(5,5,5,5,5);
    case 26: return Arrays.asList(7,7,7,5);
    case 27: return Arrays.asList(7,5,5,5,5);
    case 28: return Arrays.asList(7,7,7,7);
    default:
       List<Integer> nminus5 =
           new ArrayList<>(change(n-2));
       nminus5.add(5);
       return nminus5;
  }
}
```

```
def change(n):
  assert(n>=24)
  if n==24: return [7,7,5,5]
  elif n==25: return [5,5,5,5,5]
  elif n==26: return [7,7,7,5]
  elif n==27: return [7,5,5,5,5]
  elif n==28: return [7,7,7,7]
  else:
   nminus5 = change(n-5)
   return nminus5 + [5]
```

Congratulations! We have turned an inductive proof into code. There are 2 groups: $n = 24$ through 28 are solved directly, and for $n \geqslant 29$, it is solved recursively—or should we say inductively?

We also have a recipe for discovering inductive proofs:

1. Just write down the boilerplate—base case, inductive hypothesis, inductive steps, etc. etc.
2. Think about reducing a really big n to something smaller. Don't care about base cases for now.
3. Fill in the holes, including the base cases.
4. Rewrite the whole thing for readability.