Lecture 6: Promise and Inductive Thinking

built on 2019/05/09 at 13:41:16

Today's lecture covers *three* ideas: (1) how to specify the input/output behavior of a function, (2) how to think recursively—thinking in terms of smaller instances of the same problem, and (3) how to prove correctness of such recursive algorithms, using (mathematical) induction.

We will write $\hookrightarrow$ to mean "produces the value." Therefore, $2 + 2 \hookrightarrow 4$ means the expression $2 + 2$ produces the value 4, and similarly, $\texttt{foo}(2) + 5 \hookrightarrow 11$ means the left-hand side expression produces the value 11.

1 Contracts

An important theme of this course is learning to design correct algorithms and write correct programs. We now introduce a set of tools to help you specify precisely the behaviors of your code.

When we define a function, we also want to specify:

(1) *preconditions* (what the input is like, what other things look like); and
(2) *postconditions* (what's being returned, what effects have been made).

These are collectively called **contracts**; they are what we agree to provide. We'll see a number of techniques to show that the contracts are met.

When we write down a contract, we're focusing on the user-facing part: writing down preconditions and postconditions—and writing simple tests to ensure the implementation adheres to the spec.

For starters, we'll write a function that squares a given number. What do we promise to the user of this function? Perhaps, if you give it a number x, it will return x^2. In the lingo of pre/post conditions, we write:

```
// pre: x is a number of type long
// post: return a number of type long such that sqr(x) = x*x
long sqr(long x) {
  ....// indented code
}
```

Just writing down a contract doesn't make a function correct. But we'll try to get into the habit of writing pre- and post- conditions for all nontrivial functions we will write (or when asked). This is so that we are clear about what we want a function to do.

Blackbox Testing: We can test if the implementation meets the contracts by writing tests. (Technically, for the most part, an exhaustive test is *not* possible; there are just too many possibilities to try. Because of this, we can typically only look for counterexamples but we can't claim correctness with certainty.) We have seen how to use assertions to write tests.

Testing Fast Implementations Using An Easy One: Often, the task at hand will call for a fast implementation, which tends to be complex and difficult to code correctly. For this reason, it's imperative that we test the implementation extensively. But how're we to find a model solution to compare answers. It turns out, for most problems, you can write easy programs that may be slow; however, they are especially useful for testing your sophisticated implementation.

2 Recursive Algorithms: Think Backward

Recursion lies at the heart of many important algorithms design techniques and is a powerful pattern of thinking and reasoning. Quite contrary to the linear flow found in loops, thinking recursively requires thinking "backward." For this reason, the technique appears counterintuitive at first, but the basic idea itself is very simple and versatile. To use recursion, we need to answer two questions:

(Q1) How to solve small instances? (E.g. when the input size is 0, 1 or 2).

(Q2) How to tackle an instance in terms of smaller instances assuming we already know how to solve these smaller ones? More concretely, given a problem instance I of size n, if we assumed our algorithm can solve it for all problem size $< n$, could we solve this instance I?

To say that something is small or large, we need to define a measure of the instance's size. We're free to choose whatever measure we want, but it's generally a function of the input parameters.

In the terminologies of mathematical induction, the small instances are the base cases and the reduction from an instance to a smaller one, the inductive step.

As with other algorithms we have looked at in the past, we are interested in showing correctness (the algorithm works as intended) and analyzing its efficiency (what's the running time?). We have seen how to analyze the running time of recursive algorithms by writing another recursive function that describes its behavior—this is called recurrence relations.

We illustrate how to apply this algorithm design pattern with two examples: (i) raising a number to a non-negative power and (ii) finding the maximum number in a list.

2.1 Example I: Raising A Number To A Positive Integral Power

Our first example will be the familiar powering function. Specifically, we will implement a function $\texttt{pow}(b, w)$ which takes as input a non-zero real number $b \neq 0$ and a non-negative integer $w \geqslant 0$ and is to output the number b^w. To apply the above guideline, we attempt to answer the following two questions:

(Q1) **How to solve small instances?** First, we need a way to measure the size of the input. There are two parameters—b and w—in our input.

As a first attempt, we'll measure the input size in w. Here also, we have the liberty of choosing the smallest instance that won't be solved recursively. Since our code only needs to work for $w \geqslant 0$, we'll pick $w = 0$ as the small instance we'll handle directly. For this, we know that $b^0 = 1$, so it's simple: `if w==0: return 1`.

(Q2) **How to tackle an instance in terms of smaller instances?** Say, for any $w \geqslant 0$, we already already knew how to compute $\texttt{pow}(b, 0) \hookrightarrow b^0$, $\texttt{pow}(b, 1) \hookrightarrow b^1, \ldots, \texttt{pow}(b, w-1) \hookrightarrow b^{w-1}$. The question to answer is: *can we compute b^w in terms of these?*

To solve this question, we look to algebraic identities. Here's an identity we learned years ago—for $w > 0$, $b^w = b^{w-1} \times b$. We can readily use it: Because we know how to compute b^{w-1} (simply calling $\texttt{pow}(b, w-1)$), to compute b^w, we can just multiply that number by b, according to the identity. Therefore, we have: for $w > 0$, `return pow(b, w-1) * b`.

Turning these ideas into code is straightforward. We only have to check whether we're in the recursive case or in the small-instance case, like so:

```
long pow(long b, long w) {
   if (w==0) return 1;
   else return pow(b, w-1)*b;
}
```

Time Complexity: We have seen how to write a recurrence relation for this. It is simply $T(w) = T(w-1) + O(1)$, which solves to $T(w) = O(w)$.

In conclusion: Our choice of size measure works (remember we picked w). We have written a recursive function for computing b^w, $w \geqslant 0$, with running time $O(w)$.

But we hope to do better. How can we do better?

2.1.1 Aggressively Reducing the Problem Size

In hopes to achieve a better running time, we want to understand why our first solution isn't fast. One culprit is that when we reduce the problem size, we express it in terms of a instance only one size smaller. Therefore, it's

natural to attempt to reduce the problem size more aggressively. *But how?*

Suppose we want to halve the problem size (i.e. halve w). We need to look for a different algebraic identity that gives us that. Here is where the following fact becomes handy:

Fact: if $x \geqslant 0$ is even, then $b^x = b^{x/2} \times b^{x/2}$ (note that $x/2$ is a whole number).

We'll formulate a recursive solution using this fact. Again, we need to answer the same two questions. Let's use the existing small-instance solution for now. We'll just focus on the large-instance case.

When we attempt to apply the fact, we quickly see that it doesn't apply for all x—only when x is even could we apply the identity. What are we to do when x is odd? One solution is to go back to the previous fact: $b^x = b^{x-1} \times b$, but for simplicity of the running time analysis, let's do the following:

> If x is odd, then $x-1$ is even and $(x-1)/2$ is a whole number. In Java, for odd x, $x/2$ is the same as $(x-1)/2$ under integer division. Therefore,
>
> $$b^x = b^{x-1} \times b = b^{\frac{x-1}{2}} \times b^{\frac{x-1}{2}} \times b = \texttt{pow}(b, x/2) * \texttt{pow}(b, x/2) * b$$
>
> where we have applied both facts.

Turning this into code is a simple exercise. Once again, we distinguish between $w == 0$ and $w > 0$:

```
long pow2(long b, long w) {
  if (w==0) return 1;
  else {
    if (w%2==0) return pow2(b, w/2) * pow2(b, w/2);
    else return pow2(b,w/2)*pow(b,w/2)*b;
  }
}
```

Can we analyze the time complexity of `pow2`? To do this, we resort to recurrence relations once more. Like before, we still measure the problem size in w and we let $T(w)$ denote the time to run $\texttt{pow2}(b, w)$ for any b.

With this definition, we have, once again, $T(0) = O(1)$. But the case when $w > 0$ is different. Two things can happen here depending on whether x is even or odd.

- *If x is even*, we make **two** calls to `pow2` and multiply the resulting values together. Here, to get the precise expression, we ask ourselves: what's the size we're calling it on? The answer is $\approx w/2$ as indicated in the code. Therefore, in the case that x is even, the time is $2T(w/2) + O(1)$.
- *If x is odd*, like above, we make **two** calls to `pow2` and multiply the resulting values together, then with b. Other than the two recursive calls, the rest of the steps take constant time. The two calls have the same size, which can be determined similarly to the even case. The size here is indeed $w \approx w/2$. So, the time in this case is $2T(w/2) + O(1)$.

Hence, regardless of the parity of w (i.e., whether x is odd or even), we have $T(w) = 2T(w/2) + O(1)$. By our look-up table, this solves to $T(w) \in O(w)$. But wait... we haven't made any progress, have we?

2.2 Don't Do The Same Work Twice

Upon closer examination, we see that in both the odd and the even cases, we *unnecessarily* call the same $\texttt{pow2}(b, w/2)$ twice. This is wasteful, knowing that both calls to $\texttt{pow2}(b, w/2)$ will give the same result. Our next step, therefore, is to cut down on wasteful work: instead of computing $\texttt{pow2}(b, w/2)$ twice, we'll do it just once and save the result for further use. The idea is that if we store in t the value $t = \texttt{pow2}(b, w/2)$, then $\texttt{pow2}(b, w/2) \times \texttt{pow2}(b, w/2) = t \times t$. Hence, we rewrite the code as follows:

```
long pow3(long b, long w) {
  if (w==0) return 1;
  else {
    long t=pow3(b, w/2);
    if (w%2==0) return t*t;
```

```
    else return t*t*b;
  }
}
```

What's the running time complexity of `pow3`? The true and tried method for studying a recursive function's complexity is recurrence relations. For one more time, we'll let $T(w)$ be the time to run `pow3`(b, w) for any b.

Following the same argument as before, we have $T(0) = O(1)$. And for $w > 0$, we perform the following steps: (i) we compute $t =$ `pow3`$(b, w/2)$, (ii) depending on the parity of w, we either perform 1 or 2 multiplications. By now, we know that to compute t, we're calling `pow3` with problem size $\approx w/2$, so the cost of (i) is $T(w/2)$. Furthermore, the cost of (ii) is constant. We conclude that $T(w) = T(w/2) + O(1)$, which we know from the look-up table, solves to $T(w) \in O(\log w)$.

2.3 Proving Correctness of `pow`

As our first example, we'll apply this principle to prove correctness of the `pow` function. What we need to show is that for all b and for all $n \geqslant 0$, `pow`$(b, n) \hookrightarrow b^n$. More formally, we write down the following:

Theorem 2.1 *For all real number $b \neq 0$ and integer $w \geqslant 0$, the function* `pow`(b, w) *returns b^w.*

A quick glance at the code reveals that when calling `pow` with n, we only call `pow` with $n-1$, so let's use

$P(n) \equiv$ "for any b, `pow`$(b, n) \hookrightarrow b^n$".

since if $P(n-1)$ is true, we know that `pow`$(b, n-1) \hookrightarrow b^{n-1}$.

We attempt to formally prove that $P(n)$ holds for all $n \geqslant 0$ using mathematical induction as follows:

1. **Base Case.** We'll show that $P(0)$ holds. Specifically, we'll prove that for any b, `pow`$(b, 0) \hookrightarrow b^0$. To establish this, we examine the code. The `if w==0: return 1` statement indicates that we'll return 1. But since $1 = b^0$, we have that `pow`$(b, 0) \hookrightarrow b^0$, proving the base case of $P(0)$.
2. **Inductive Step.** For $n > 0$, we assume that the property holds for $n-1$ and establish that it holds for n. Consider the call `pow`(b, n), and remember that if $P(n-1)$ holds, we have `pow`$(b, n-1) \hookrightarrow b^{n-1}$. Because $n > 0$, we know that the else branch in the code is taken, so the algorithm returns `pow(b, n - 1)*b`. But by our assumption, we know `pow`$(b, n-1) \hookrightarrow b^{n-1}$. Hence, the return value of our algorithm is $b^{n-1} \times b = b^n$. And we have just established that for $n > 0$, if $P(n-1)$ is true, then $P(n)$ is true.

2.4 Proving Correctness of `pow3`

Having proved `pow` correct, we now turn to showing that `pow3`, a more efficient powering function, works as intended—i.e., for all b and integers $n \geqslant 0$, `pow3`$(b, n) \hookrightarrow b^n$.

We can use the same outline as the proof above. For starters, we'll use the name $P(n)$ and see if anything needs to changed. For the moment, let's use

$P(n) \equiv$ for any b, `pow3`$(b, n) \hookrightarrow b^n$.

It is easy to see that the base case will continue to hold, but crucially, we need to analyze the expressions $t \times t$ and $t \times t \times b$, where $t =$ `pow3(b, w/2)`. The trouble is that in the inductive step, we assume $P(n-1)$ and we attempt to show $P(n)$, but $P(n-1)$ *tells us nothing about what* `pow3(b, n/2)` *is like.*

We need something stronger.

Strengthen the Inductive Hypothesis: A closer look at the code of `pow3` shows that whenever we're trying to show $P(n)$, we only need to know about $P(n/2)$. While we could formulate a specialized form of induction that caters to this case, there's a common form of induction that will work in this case and more: *When just the previous term is not enough, make the assumption cover everything smaller.* This is known as strong induction.

When we carry out the inductive step, instead of assuming just $P(n)$, we'll assume everything before that is true as well. Specifically, this is what we assume:

For all $0 \leqslant n' < n$, $P(n')$ is true

In words, with this assumption being true, we know that

$$\begin{aligned} \texttt{pow3}(b,0) &\hookrightarrow b^0 \\ \texttt{pow3}(b,1) &\hookrightarrow b^1 \\ &\vdots \\ \texttt{pow3}(b,n-1) &\hookrightarrow b^{n-1} \end{aligned}$$

To prove $P(n)$ in general, we use the true and tried (strong) mathematical induction:

1. **Base Case.** We'll show that $P(0)$ holds. Specifically, we'll prove that for any b, $\texttt{pow3}(b,0) \hookrightarrow b^0$. To establish this, we proceed as before—the steps are straightforward.
2. **Inductive Step.** Let $n > 0$ be given. We assume the property holds for all $0 \leqslant n' < n$. What we need to know is that assuming this, we can prove $P(n)$.
   Consider the call $\texttt{pow3}(b,n)$. Let's first establish what value t takes on. Since $t = \texttt{pow3}(b,n/2)$, we allude to our assumption. *Why is this valid?* First, let's check the following:

   $$\texttt{n/2} = \lfloor n/2 \rfloor \leqslant \frac{n}{2} = \frac{n+0}{2} < \frac{n+n}{2} = n.$$

   Therefore, this means, among other things, that $\texttt{pow3}(b,n/2) \hookrightarrow b^{n/2}$. Hence, we know that $t = b^{n/2}$.
   Now if $n > 0$, there are two cases to consider as there is an `if` statement checking the parity of n. Before we proceed, we recall two facts that we reasoned about before:
   (A) If n is odd, $n/2 = \frac{n-1}{2}$. (B) If n is even, $n/2 = \frac{n}{2}$.
   We're ready to analyze these cases:
   - *If n is even*, we return $t \times t$. But because n is even, $n/2 = \frac{n}{2}$, so $t \times t$, which we return, equals $b^{n/2} \times b^{n/2} = b^{n/2} \times b^{n/2} = b^n$ by algebraic properties and what we have established about t. Therefore, the return value is b^n.
   - *If n is odd*, we return $t \times t \times b$. But because n is odd, $n/2 = \frac{n-1}{2}$, so $t \times t \times b$, which we return, equals

     $$b^{n/2} \times b^{n/2} \times b = b^{\frac{n-1}{2}} \times b^{\frac{n-1}{2}} \times b = b^{n-1} \times b = b^n$$

     by algebraic properties and what we have established about t. Therefore, the return value is b^n.

   Consequently, regardless of whether n is odd or even, $\texttt{pow3}(b,n) \hookrightarrow b^n$—and we have proved $P(n)$.

2.5 Example II: Finding The Maximum Value In a List

The second example deals with finding the largest number in a sequence (a Java array or a Python list). In particular, we'll write a function $\texttt{mymax}(A)$, which takes an array of numbers A and outputs the max value. This is easily done using a loop. But to practice inductive thinking, we'll attempt to cast this as a recursive process. To do so, we need to ask ourselves two questions:

(Q1) **How to solve small instances?** We begin by setting how we measure the input size. It makes sense to use the length as our measure.
As usual, we have the liberty of choosing the smallest instance that won't be solved recursively. It's natural to pick the smallest possible input—an array of size 1.
What's the maximum value in an array of size 1? It is the only element in that array. Hence, if the input is an array A, the answer in this case is $A[0]$.

(Q2) **How to tackle an instance in terms of smaller instances?** Let's say our function is called with $\texttt{mymax}(A)$ with length n.
Like we did before, let's assume that for all sizes *below* n, we already know how to solve `mymax`. Specifically, this means:
For any array T, $0 \leqslant |\texttt{T}| < n$, we already know how to compute $\texttt{mymax}(T)$.

The question to answer now is: *can we compute* `mymax`(A) *in terms of* `mymax` *on smaller inputs?*
One immediate thought is to split the array at midpoint. Compute the max values for both sides and compare their results. Indeed, we have if $m = |\texttt{A}|/2$, then borrowing array slicing notation from Python, the max of A is

`max(mymax(left), mymax(right))` where `left=A[:m]` and `right=A[m:]`.

Importantly, this works because the problem size becomes smaller in both recursive calls—as we will prove in a bit.
Turning this into code is simple:

```
int mymax(int[] A) {
  if (A.length==1) return A[0];
  else {
    int m = A.length/2;
    int[] left = Arrays.copyOfRange(A, 0, m);
    int[] right = Arrays.copyOfRange(A, m, A.length);
    return Math.max(mymax(left), mymax(right));
  }
}
```

The time complexity of this function, however, is not great. We don't have time to analyze it in this lecture, but if you were to analyze it, you would get $T(n) = 2T(n/2) + O(n)$, which solves to $O(n \log n)$. We will see ideas on how to improve this in a future lecture. For now, let us focus on proving correctness.

2.5.1 Correctness of `mymax`

To show that `mymax` works as intended, we'll prove the following theorem[1]:

Theorem 2.2 *On input an array of numbers A, the* `mymax` *function returns the maximum value from A; that is, it returns* $\max\{A[0], A[1], \ldots, A[n-1]\}$ *where* $n = |\texttt{A}|$.

Proof: We proceed by strong induction on the length of the input array. Let $P(n)$ be the proposition

$P(n) \equiv$ for any array of numbers A of length n, the function `mymax` on input A returns $\max\{A[0], A[1], \ldots, A[n-1]\}$ where $n = |\texttt{A}|$.

To complete this proof, we will show that (1) the base cases hold and (2) the inductive step works:

— **Base Case:** We'll prove that $P(1)$ is true. When the array A has length $n = 1$, the code, by inspection, returns $A[0]$, which is the obvious maximum of all the elements of A, hence proving $P(1)$.

— **Inductive Step:** Let $k \geqslant 1$ be any integer. We assume that $P(\ell)$ holds for $0 \leqslant \ell \leqslant k$, and we wish to prove $P(k+1)$. That is, using this assumption, we want to prove that for any array A of numbers of length $k+1$, the `mymax` function returns the maximum element.

Examining the code, we see that because $k \geqslant 1$, the length $|A| = k+1$ is guaranteed to be at least two, so we enter into the else branch. Here we set m to $|A|/2$ and call `mymax` recursively on `left` and `right`, which are, in Python slicing notation, A`[:m]` and A`[m:]`, resp. First, we note that $m = |A|/2 = (k+1)/2 = \lfloor \frac{k+1}{2} \rfloor$. This means A`[:m]` has length

$$m = \left\lfloor \frac{k+1}{2} \right\rfloor \leqslant \frac{k+1}{2} \leqslant \frac{k+k}{2} \leqslant k,$$

so our inductive hypothesis (IH) applies for calling `mymax` on A`[:m]`. Therefore, by IH, we have that `mymax`$(A$`[:m]`$)$ returns $\max\{A[0], A[1], \ldots, A[m-1]\}$.

Moreover, A`[m:]` has length $n - m = k + 1 - \lfloor \frac{k+1}{2} \rfloor < k$ because $\lfloor \frac{k+1}{2} \rfloor \geqslant 1$ (as $k \geqslant 1$). Thus, our inductive hypothesis (IH) applies for calling `mymax` on A`[m:]`. Therefore, by IH, we have that `mymax`$(A$`[m:]`$)$ returns $\max\{A[m], A[m+1], \ldots, A[(k+1)-1]\}$.

[1] *Technical Note:* We will be a bit sloppy with how max works. Everything we handwave here can be fully formalized as max can be seen as an associative binary operator (And if this paragraph means nothing to you, you should just skip it.)

We conclude that the function's return value equals

$$\begin{aligned}
&\max\left\{\texttt{mymax}(A\texttt{[:m]}), \texttt{mymax}(A\texttt{[m:]})\right\} \\
&\quad = \max\left\{ \max\left\{A[0], A[1], \ldots, A[m-1]\right\}, \max\left\{A[m], A[m+1], \ldots, A[(k+1)-1]\right\}\right\} \\
&\quad = \max\{A[0], A[1], \ldots, A[(k+1)-1]\},
\end{aligned}$$

which is the maximum of the elements of A, as desired, hence proving $P(k+1)$.

Hence, $P(n)$ for all $n \geqslant 0$, which concludes the proof for the theorem. ■

3 Pulling Extra Properties Out Of Thin Air

The following example illustrates a technical point about inductive techniques. Consider the factorial function defined recursively on non-negative integers:

$$n! = n \times (n-1)!, \text{ where } 0! = 1.$$

This recursive definition lends itself to a natural recursive implementation. But often, a different style of recursion—known as tail recursion—is preferred. A tail-recursive function is one where the code makes at most one recursive call and for each call, no additional operations are performed after the recursive call returns. One can implement factorial as a tail-recursive function as follows:

```
def fact_helper(n, a):
    if n==0: return a
    else: return fact_helper(n-1, a*n)

def fact(n): return fact_helper(n, 1)
```

The particulars of this implementation is not important for now; we'll only use it to showcase an aspect of inductive proof. We want to show that $\texttt{fact}(n) \hookrightarrow n!$ and specifically, we want to use induction to show that $\texttt{fact\_helper}(n,1) \hookrightarrow n!$. As in the previous examples, it is natural to use $P(n) \equiv$ "$\texttt{fact\_helper}(n,1) \hookrightarrow n!$" as this is the only property we need from `fact_helper`. Following the standard process, we have:

- **Base Case.** We'll show that $P(0)$ holds. As $\texttt{fact\_helper}(0,1) \hookrightarrow 1 = 0!$, we have proved the base case.
- **Inductive Step.** For $n > 0$, we assume that the property holds for $n-1$ and establish that it holds for n. When $n > 0$, $\texttt{fact\_helper}(n,1)$ returns $\texttt{fact\_helper}(n-1,n)$ since $a = 1$. But we have no idea what $\texttt{fact\_helper}(n-1,n)$ is. Neither would it help to use strong induction on $P(n)$ like in previous examples.

What we need is a statement that covers not only $a = 1$ but also other values of a. (Because we don't know what a the function will be called with, it's best to make it general.) We'll revise it so that our property works for any n and a—that is, we want a property of the form $\texttt{fact\_helper}(n,a) \hookrightarrow ...$

It helps to look at a few steps of $\texttt{fact\_helper}(n,a)$. When $\texttt{fact\_helper}(n,a)$ is called, we have:

$$\begin{aligned}
&\texttt{fact_helper}(n,a) \\
\to\ &\texttt{fact_helper}(n-1, n\times a) \\
\to\ &\texttt{fact_helper}(n-2, n\times(n-1)\times a) \\
&\vdots \\
\to\ &\texttt{fact_helper}(1, n\times(n-1)\times(n-2)\times 2\times a) \\
\to\ &\texttt{fact_helper}(0, n\times(n-1)\times(n-2)\times 2\times 1\times a) \\
\hookrightarrow\ &n!\times a,
\end{aligned}$$

where a pattern has emerged—$\texttt{fact\_helper}(n,a) \hookrightarrow n! \times a$. Therefore, we'll revise $P(n)$ to

$P(n) \equiv$ "for all a, $\texttt{fact\_helper}(n, a) \hookrightarrow n! \times a$".

Once we establish that $P(n)$ holds for all $n \geqslant 0$, we'll be golden.

Exercise: Complete this proof.