7 Inductive Thinking for Algorithms Design

This chapter presents an important framework for algorithms design and analysis: design by inductive reasoning. Throughout history, (mathematical) induction has been successfully applied to help design algorithms, prove them correct, and establish their resource requirements analytically. It has proved to be versatile, generally intuitive, and rigorous, worth adding to the toolbox of any student of computer science.

We begin by reviewing a concept in programming known as recursion. We will briefly recap how induction works. Then, we will use these techniques creatively to help in algorithms design.

Inductive Thinking. When it comes to crafting an algorithm, induction and recursion are like soulmates made in heaven, complementing each other from the design and analysis to the implementation of an algorithm.

7.1 The Anatomy of Recursion

In practical terms, *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 will begin with two examples that illustrate the use of recursion.

The Factorial Function

As a first example, we will 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.,

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

with $0! = 1$. This means, for instance, $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.

The factorial 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, we will write out the expressions for $4!$ and $3!$ and observe their relationship:

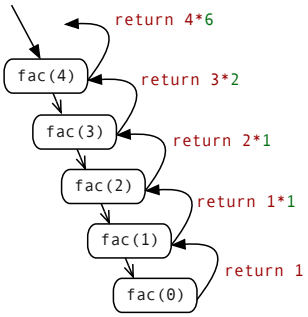
$$\begin{aligned} 3! &= \quad 3 \times 2 \times 1 \\ 4! &= 4 \times 3 \times 2 \times 1 \end{aligned}$$

Therefore, $4! = 4 \times (3 \times 2 \times 1) = 4 \cdot 3!$. In general, we can write the recursive function recursively (i.e., one which calls itself) as follows:

More generally, we have $n! = n \times (n-1)!$, where $0!$ is defined to be 1.

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

Recursion Trace of `fac(4)`.



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 is no circularity. Hence, the function will 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 is happening. An important tool in this case is a *recursion trace* diagram. It is best to describe this with an example: the recursion trace of `fac(4)` on the left shows how `fac` is called and what each instance of it returns when that call concludes. We can see, for example, that `fac(4)` calls `fac(3)` and depends on its return value. But `fac(0)`, our nonrecursive case, can return right away, without further calling itself.

How does this actually work? At this point, we may be wondering how the computer keeps track of where it is when it is the same function that is called again and again. After all, there are so many "incarnations" of `fac`. The answer is it keeps a stack: Each time the 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. In this way, it knows precisely where in the code to return to.

An English Ruler

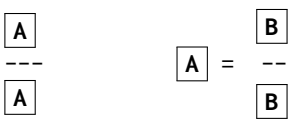
A 4-Level English Ruler.

```
---- 0
-
--
-
---
-
--
-
---- 1
```

Our first example was a nice, well-behaved mathematical function; however, when it comes to programming, there is not much of a practical reason that 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 example, we will focus on drawing the scale from 0 to 1 (excluding the 0 and 1 ticks).

Why might recursion be the natural choice for implementing this structure? Upon a closer look, the English ruler repeats the same pattern above and below the `---` line; the repeated pattern is rendered as **`A`** below. Moreover, the **`A`** portion further follows the same decomposition, where it is repeating the same pattern above and below the line `--`. This is denoted by **`B`** below.



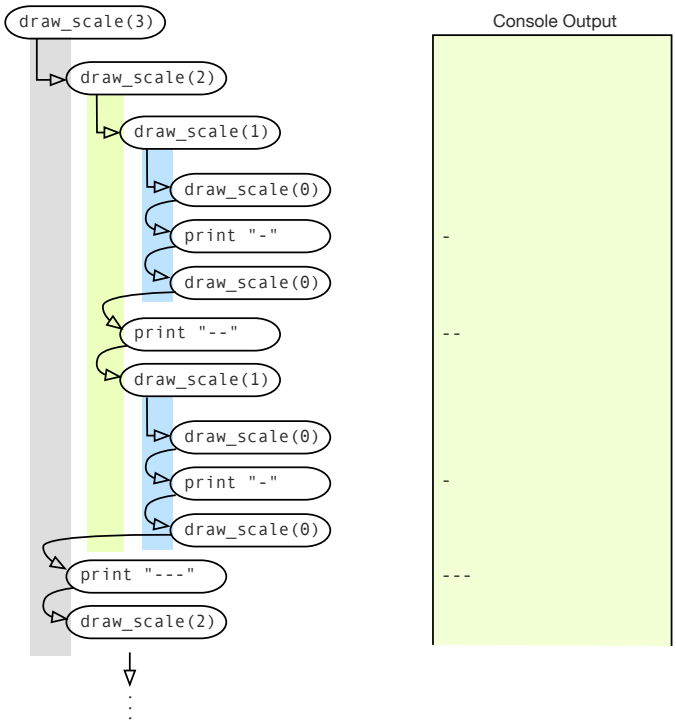
This nature of repetition suggests a recursive implementation. To draw the ruler, the function calls itself to draw **`A`**, prints out `---`, then calls itself again

to draw the other **A**. How would it draw **A**? It calls itself recursively to draw **B**, prints out `--`, then calls itself again to draw the other **B**. At smaller scales, we observe the same structure again and again, until there is no more to draw. This logic is expressed in the code on the right.

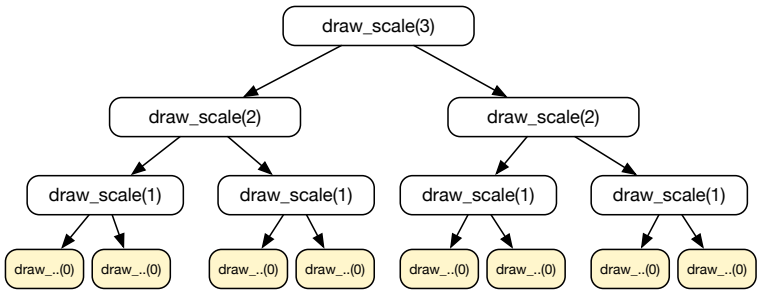
In code, we have:

```
void draw_scale(int n) {
  if (n > 0) {
    draw_scale(n-1);
    println("-".repeat(n));
    draw_scale(n-1);
  }
}
```

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. Below is a trace 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.



Often, a detailed trace of how the program executes is not necessary. For a more holistic view of the run, we will use a tree diagram sometimes known as the recursion tree. 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. Below is an example when the call is initiated with `draw_scale(3)`.



7.2 Mathematical Induction

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

Tips

One important point about the predicate $P(\cdot)$ is that it is a function that returns a Boolean value **True** or **False**. Therefore, do *not* take $P(n)$ and add it to a number.

An inductive proof is accomplished 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.

Domino cascade

https://www.flickr.com/photos/14516334@N00/297237720 CC BY-SA 2.0

Let us try to understand why these steps imply that $P(0), P(1), \ldots$ hold *ad infinitum*. Induction works much like the cascading effect in rows of dominoes. First, the base case indicates that $P(0)$ is true. Then, here is 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 works 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 $P(n)$?

As was justified by the base case, we know $P(0)$ is true. 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. So, 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. Because this chain reaction goes on infinitely, $P(n)$ holds for all integer $n \geqslant 0$.

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.

We will now take a look at some examples. In these examples, we are somewhat pedantic with the goal of pointing out the various features that we should think about or question them with great skepticism (when we read or write proof).

Example I: Summation

The well-known summation formula states that

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

As our first example on induction, we will show that this formula is indeed true. In particular, we will prove the following lemma:

Lemma 7.1. For all integer $n \geqslant 1$, the summation

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

To proceed, our plan will be 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. Let us try the simplest one first—just repeat the claim.

$$P(n) \equiv \text{“}1+2+3+\cdots+n = n(n+1)/2.\text{”}$$

What 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.

Using the summation notation, we would write

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

Proof. We will proceed by induction.

Base Case. The smallest n we claim is $n = 1$, so we will 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 $n = 1$. The inductive step will show it for $n > 1$. Let $n \geqslant 2$. 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

[†]This assumption is known as the *inductive hypothesis* (IH).

$$1+2+3+\cdots+(n-1) = \frac{(n-1)n}{2}. \tag{7.1}$$

To show that $P(n)$ holds, we need to verify that LHS equals RHS for the current n. We will look at the left-hand side (LHS) first. On the LHS 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

$$\begin{aligned}
&1+2+3+\cdots+(n-1)+n \\
&\quad= \Big[1+2+3+\cdots+(n-1)\Big]+n \\
&\quad= \frac{(n-1)n}{2}+n && \text{[IH implies (7.1)]} \\
&\quad= \frac{(n-1)n}{2}+\frac{2n}{2} && \text{[algebra]} \\
&\quad= \frac{n(n-1+2)}{2} = \frac{n(n+1)}{2} && \text{[algebra]}
\end{aligned}$$

which is equal to RHS, as we wish. Therefore, we conclude that $P(n-1)$ implies $P(n)$ for $n \geqslant 2$.

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 lemma. □

Example II: Divisibility

There is a simple alternative proof if we remember the geometric sum formula:

$$\sum_{k=0}^{n-1} x^k = \frac{x^n - 1}{x - 1}$$

Our next example is a common fact about divisibility:

Lemma 7.2. For all natural numbers $n \geqslant 0$ and $x \neq 1$, $x^n - 1$ is divisible by $x - 1$.

We will use induction to prove this lemma. 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. Like before, we will try the simplest one first—just repeat the claim.

$$P(n) \equiv \text{ "for all } x \neq 1,\ x^n - 1 \text{ is divisible by } x - 1."$$

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. Notice that we quantify x ("for any x") as part of the predicate.

Divisibility. Remember that x is divisible by y if and only if there is an integer $\beta \in \mathbb{Z}$ such that $x = \beta \cdot y$.

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

Proof. Base Case. We begin by checking that $P(0)$ is true. For this to be true, we need to verify that when $n = 0$, 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

$$x^{n-1} - 1 = \beta(x - 1). \tag{7.2}$$

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

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

Therefore, we know that

$$\begin{aligned} x^n - 1 &= x(x^{n-1} - 1) + (x - 1) && \text{[algebra, above]} \\ &= x\big[\beta(x - 1)\big] + x - 1 && \text{[IH implies (7.2)]} \\ &= (x - 1)(\beta \cdot x + 1) && \text{[algebra]} \end{aligned}$$

But this shows that $x^n - 1 = \gamma(x - 1)$ where $\gamma = \beta \cdot x + 1 \in \mathbb{Z}$ is an

integer, 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 lemma. □

7.3 Induction-Inspired Algorithms

Inductive proofs tend to translate directly to an algorithm, easily implementable using recursion. We will explore this connection via an example.

We would like to design an algorithm that expresses any postage amount $n \geqslant 24$ as a sum of 5-cent and 7-cent stamps. We begin by proving the following theorem:

This problem is more generally known as **linear Diophantine equations**, whose simplest form asks whether there are integers x and y that satisfy the equation

$$\alpha \cdot x + \beta \cdot y = \gamma,$$

where α, β, γ are given integers.

Theorem 7.3. Given an unlimited supply of 5-cent stamps and 7-cent stamps, we can make any amount of postage that is at least 24 cents.

Although it can well be proved using a contradiction argument, we will march ahead with an inductive proof to demonstrate a few features. Even using induction, there are surely many routes to prove this theorem. We will 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 is at least 24 cents. So, if $n - 5 \geqslant 24$ (that is, $n \geqslant 29$), we are all set—as long as the theorem holds true. (So far, we do not yet know how to make the remaining $n - 5$, but so what! we know it can wishfully 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 $n = 24, 25, \ldots, 28$*?* We will try to manually come up with these solutions. A moment's thought shows

- $n = 24$ 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 $7 + 7 + 7 + 7$.

It appears that we have all the pieces. Let us now structure them as an inductive proof:

Proof. Consider the predicate

$$P(n) \equiv \text{"We can make } n \text{ cents in 5- and 7- cent stamps."}$$

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 do not know how directly, 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 these 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 is an n that is not covered by either of these, the proof is flawed.

We will now proceed to "decode" our proof of the theorem and implement an algorithm accordingly. The mapping is almost immediate: 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 (or inductively) and there are cases which we handle directly (or base cases). This means we can basically read off what the proof says and translate that into code. Below is an algorithm obtained by decoding the above theorem.

Code 7.1: Proof-decoded algorithm for expressing postage amounts

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

7.4 Inductive Thinking, Thinking Backwards

We have seen that an inductive proof requires covering two cases—base and inductive cases. To use recursion, we need to answer two questions:

(Q1) How to solve small instances? For example, how to solve it directly when the input size is 0, 1 or 2? This corresponds to base cases in an inductive proof.

(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? This corresponds to the inductive step in an inductive proof.

To say that something is small or large, we need to define a measure of the instance's size. We are free to choose whatever measure we want, but it is 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. It is important that for all input sizes we care about, the instance is taken care of (i.e., it falls in one of the two cases).

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 is 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 nonnegative power and (ii) finding the maximum number in a list. In these examples, we will use the shorthand $\hookrightarrow$ to mean "produces a value of." For example, `foo(4) + 1` $\hookrightarrow$ 42 means the expression `foo(4) + 1` produces the value 42.

Example I: Raising A Number To A Positive Integral Power

For $a, x, y \in \mathbb{R}$,

$$a^{x+y} = a^x \times a^y$$

Our first example will be the familiar powering function. Specifically, we will implement a function $\mathsf{pow}(b, w)$ which takes as input a non-zero real number $b \neq 0$ and a nonnegative 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 knew how to compute $\mathsf{pow}(b, 0) \hookrightarrow b^0$, $\mathsf{pow}(b, 1) \hookrightarrow b^1, \ldots, \mathsf{pow}(b, w-1) \hookrightarrow b^{w-1}$. 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 `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:

```
1 long pow(long b, long w) {
2     if (w==0)
3         return 1;
4     return pow(b, w-1)*b;
5 }
```

If we rewrite $T(w)$ as $T(w) = T(w-1) + c$, $c = O(1)$, then

$$T(w) = \underbrace{c + c + \cdots + c}_{w \text{ times}} = c \cdot w = O(w).$$

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 $T(w) = O(w)$.

Summary. 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 $O(w)$. This is no better than writing a simple for-loop that multiplies `b` into an accumulator.

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

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 an 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: $b^x = b^{x-1} \times b$. For simplicity of the running time analysis, let us observe the following fact:

> **Fact:** If $x \geqslant 0$ 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,

$$b^x = b^{x-1} \times b = b^{\frac{x-1}{2}} \times b^{\frac{x-1}{2}} \times b = \mathsf{pow}(b, x/2) * \mathsf{pow}(b, x/2) * b$$

where we have applied both facts.

Turning this into code is a simple exercise. Once again, we distinguish between $w == 0$ and $w > 0$. For $w > 0$, we further distinguish whether w is even or odd. This gives rise the following Java implementation `pow2`.

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

Time Complexity. Can we analyze the time complexity of `pow2`? To do this, we resort to recurrence relations once more. Like before, we measure the problem size in w and let $T(w)$ denote the time to run $\mathsf{pow2}(b, w)$ 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 $w/2$ 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 $\approx w/2$. 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 $T(w) = 2T(w/2) + O(1)$. This solves to $T(w) \in O(w)$. But wait! In terms of running time, we haven't made any progress at all, have we?

Don't Do The Same Work Twice

Upon closer examination of `pow2`, we see that in both the odd and even cases, we *unnecessarily* call the same $\mathsf{pow2}(b, w/2)$ twice. This is wasteful, knowing that both calls to $\mathsf{pow2}(b, w/2)$ will give the same result.

Our next step, therefore, is to cut down on wasteful work: instead of calling the function twice with the same input, we'll do it just once and save the result in a variable for further use. We will use the variable `t` to store this value. Hence, we will rewrite the code as follows:

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

Time Complexity. What's the running time complexity of pow3? The true and tried method for studying a recursive function's complexity is recurrence relations. Let $T(w)$ be the time to run $\texttt{pow3}(b, w)$ for any b. We will set up the recurrence as follows.

Following the same argument as before, we have $T(0) = O(1)$. And for $w > 0$, we perform the following steps: (i) we compute $t = \texttt{pow3}(b, w/2)$, (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. Hence,

$$T(w) = T(w/2) + O(1), \text{ with } T(0) = O(1),$$

which solves to $T(w) \in O(\log w)$.

Because $T(w)$ is $T(w) = T(\frac{w}{2}) + c$, $c > 0$,

$$T(w) = \underbrace{c + c + \cdots + c}_{\log_2(w) \text{ times}} = c \cdot \log_2 w = O(\log w).$$

Correctness of Recursive Powering Functions

We have just established the running time of our variants of recursive powering functions. Next, we will show that these algorithms are correct—the code produces the right results as expected in the specifications. It is instructive to start with the simple pow algorithm and progress to pow3, which is somewhat more involved.

Proving Correctness of pow

We begin by proving correctness of the pow function. What we need to show is that for all b and for all $n \geqslant 0$, $\texttt{pow}(b, n) \hookrightarrow b^n$. More formally, we write down the following:

> **Theorem 7.4.** For all integer $b \neq 0$ and integer $w \geqslant 0$, the function $\texttt{pow}(b, w)$ 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 the predicate

$$P(n) \equiv \text{ "for any } b,\ \texttt{pow}(b, n) \hookrightarrow b^n\text{"}.$$

With this predicate, if $P(n-1)$ is true, we know that $\texttt{pow}(b, n-1) \hookrightarrow b^{n-1}$.

> **Proof.** 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, $\texttt{pow}(b,0) \hookrightarrow b^0$. 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 $\texttt{pow}(b,0) \hookrightarrow b^0$, 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 $\texttt{pow}(b,n)$, and remember that if $P(n-1)$ holds, we have $\texttt{pow}(b,n-1) \hookrightarrow b^{n-1}$. 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 $\texttt{pow}(b,n-1) \hookrightarrow b^{n-1}$. 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.

By induction, $P(n)$ holds for all integer $n \geqslant 0$. □

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 $n \geqslant 0$, $\texttt{pow3}(b,n) \hookrightarrow b^n$.

We can use the same outline as the previous proof. For starters, we'll use the name $P(n)$ and see if anything needs to be changed. We define the following predicate:

$$P(n) \equiv \text{ for any } b,\ \texttt{pow3}(b,n) \hookrightarrow b^n.$$

The base case clearly will continue to hold, but crucially, we need to analyze the expressions $t \times t$ and $t \times t \times b$, where $t = \texttt{pow3(b, w/2)}$. The trouble is that if in the inductive step, we assumed $P(n-1)$ and we attempted to show $P(n)$, then $P(n-1)$ *could tell 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, we assume:

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

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

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

Theorem 7.5. For all integer $b \neq 0$ and integer $w \geqslant 0$, the function `pow3(b,w)` returns b^w.

Proof. We proceed by strong induction.
Base Case. We'll show that $P(0)$ holds. Specifically, we'll prove that for any b, `pow3(b,0)` $\hookrightarrow b^0$. To establish this, we proceed as before—the steps are straightforward and are omitted.
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 `pow3(b,n)`. Let's first establish what value t takes on. Since $t =$ `pow3(b, n/2)`, we wish to allude to our assumption (inductive hypothesis) for the value of t. *But why is this valid?* First, because when we write `n/2` in Java code, the value obtained is $n/2$ with the fractional part truncated—or $\lfloor n/2 \rfloor$ in mathematical notation. To avoid potential confusion, define $k = \lfloor n/2 \rfloor$. We will use k when we wish to refer to the Java value `n/2`.
To apply the inductive hypothesis, we'll check the following:

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

This means, among other things, that $t =$ `pow3(b, n/2)` $\hookrightarrow b^k$. Hence, we know that $t = b^k$.
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:

$$\text{(A) If } n \text{ is odd, } k = \tfrac{n-1}{2}. \quad \text{(B) If } n \text{ is even, } k = \tfrac{n}{2}.$$

We're ready to analyze these cases:

- *If n is even*, we return $t \times t$. But because n is even, $k = \frac{n}{2}$, so $t \times t$, which we return, equals $b^k \times b^k = b^{n/2} \times b^{n/2} = b^n$ 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, $k = \frac{n-1}{2}$, so $t \times t \times b$, which we return, equals

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

  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, `pow3(b,n)` $\hookrightarrow b^n$—and we have proved $P(n)$. □

Example II: Finding The Maximum Value In a List

The second example deals with finding the largest number in a sequence (e.g., a Java array). In particular, we'll write a function `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, for an array A, the answer in this case is $A[0]$.

(Q2) **How to tackle an instance in terms of smaller instances?** Suppose our function is called with `mymax`(A) with length n.

Like we did before, assume that for all sizes *below* n, we already know how to solve `mymax`. Specifically, this means:

> For any array T, $0 \leqslant |T| < n$, we already know how to compute `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 $m = |A|/2$, 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:

```java
int mymax(int[] A) {
    if (A.length==1) return A[0];

    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));
}
```

Remarks. The time complexity of this function, however, is nothing to celebrate. If we were to analyze it, we would get $T(n) = 2T(n/2) + O(n)$, which solves to $O(n \log n)$. It can be improved if we avoid copying the array again and again. For now, let us focus on proving correctness.

Correctness of `mymax`

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

Theorem 7.6. 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 $n = |A|$.

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

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

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

1. **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 $P(1)$.

2. **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:]`, respectively. 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 $\max\{A[m], A[m+1], \ldots, A[(k+1)-1]\}$.

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

Code 7.2: Tail-recursive factorial

```
long factHelper(int n, long a) {
    if (n==0)
        return a;
    return factHelper(n-1, a*n);
}
long fact(int n) { return factHelper(n, 1); }
```

We conclude that the function's return value equals

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

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

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

7.5 Pulling Extra Properties Out Of Thin Air

Proving correctness of an algorithm inductively sometimes comes with a twist. The following example illustrates a technical point about inductive techniques. Consider the factorial function defined recursively on nonnegative 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 shown in Code 7.2.

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 $\texttt{factHelper}(n, 1) \hookrightarrow n!$. As in the previous examples, it is natural to use $P(n) \equiv$ "$\texttt{factHelper}(n, 1) \hookrightarrow n!$" as this is the only property we need from `factHelper`. Following the standard recipe, we have:

- **Base Case.** We'll show that $P(0)$ holds. As $\texttt{factHelper}(0, 1) \hookrightarrow 1 = 0!$, 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 $n > 0$, $\texttt{factHelper}(n, 1)$ returns $\texttt{factHelper}(n-1, n)$ since $a = 1$. But we have no idea what

`factHelper`$(n-1, n)$ is. Neither would it help to use strong induction on $P(n)$ like in previous examples.

The Fix? 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 `factHelper`$(n, a) \hookrightarrow \ldots$. This process is commonly known as strengthening the inductive hypothesis.

Before we can set a useful predicate, it helps to look at a few steps of `factHelper`(n, a). When `factHelper`(n, a) is called, we have:

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

where a pattern has emerged—`factHelper`$(n, a) \hookrightarrow n! \times a$. Therefore, we'll revise $P(n)$ to

$$P(n) \equiv \text{“for all } a, \texttt{factHelper}(n, a) \hookrightarrow n! \times a\text{”}.$$

Once we establish that $P(n)$ holds for all $n \geqslant 0$, we'll be golden. It is the reader's exercise to finish this proof.

Exercises

Exercise 7.1. Prove, using induction, that for $n \geqslant 1$,

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

Exercise 7.2. Find a closed form for the following sum and prove it using induction.

$$1 \cdot 2 + 2 \cdot 3 + \cdots + n(n+1).$$

Exercise 7.3. Let $a, b, n \in \mathbb{Z}_+$. Prove using induction that

$$\frac{1}{2}\left(a^n + b^n\right) \geqslant \left(\frac{a+b}{2}\right)^n$$

Exercise 7.4. Prove that for $n \geqslant 0$, `fact(n)` in Code 7.2 returns $n!$. *(Hint: prove by induction that for $n \geqslant 0$ and any a, `factHelper(n, a)` $\hookrightarrow a \times n!$.)*

Exercise 7.5. Consider the following snippet of Java code:

```
int sumHelper(int n , int a) {
    if (n==0) return a;
    else return sumHelper(n-1, a + n*n);
}

int sumSqr(int n) { return sumHelper(n, 0); }
```

Your Task: Prove that for $n \geqslant 1$, `sumSqr(n)` $\hookrightarrow 1^2 + 2^2 + 3^2 + \cdots + n^2$. To prove this, use induction to show that `sumHelper` computes the "right thing." *(Hint: How did we prove `factHelper` earlier?)*

Exercise 7.6. Consider the following function `foo`, which takes as input an integer $n \geqslant 1$ and returns a tuple of length 2 of integers:

```
def foo(n):
    if n == 1:
        return (1, 2)
    else:
        (p, q) = foo(n-1)
        return (q + p*n*(n+1), q*n*(n+1))
```

Your Task: Prove that for $n \geqslant 1$, `foo(n)` $\hookrightarrow (p, q)$ such that

$$\frac{p}{q} = 1 - \frac{1}{n+1}.$$

(*Hint:* induction on n.)

Exercise 7.7. There are *four* ways an L-shaped triomino can be arranged. Labeled by the missing corner, the four arrangements are:

Consider the following theorem:

> **Theorem:** Any 2^n-by-2^n grid with one painted cell can be tiled using L-shaped triominoes such that the entire grid is covered by triominoes but no triominoes overlap with each other nor the painted cell.

There are two tasks in this exercise:

Subtask I: Prove this theorem using induction on n. *(Hint: Every 2^n-by-2^n grid is made up of four 2^{n-1}-by-2^{n-1} subgrids, each looking much like a subproblem that can be "recursively" solved.)*

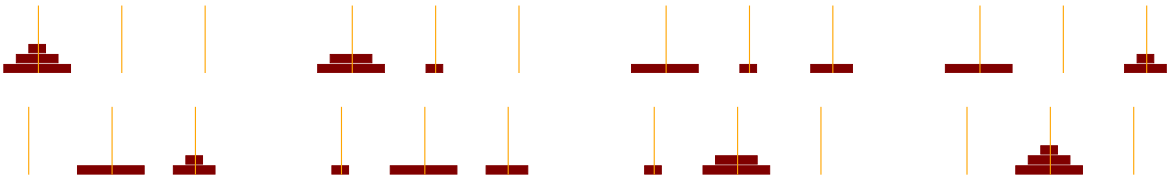


Figure 7.1: All configurations that take place in solving the Tower of Hanoi puzzle with $N = 3$.

Subtask II: Turn this proof into code. Specifically, write a function that takes an input the painted cell location and produces a plan to place triominoes according to the proof.

Exercise 7.8. Monks at a remote monastery are busy solving the Tower of Hanoi problem, a duty passed on for tens of generations. According to an old tale, when this group of monks finishes, P will be shown to equal NP and the world will come to an end.

In Tower of Hanoi, you are given N disks, labeled $0, 1, \ldots, N-1$ by their sizes. In addition, there are 3 pegs: Peg 0, Peg 1, and Peg 2. Initially, all disks are at Peg 0, neatly arranged from small (Disk 0) to large (Disk $N-1$), with the smallest one at the top and the largest one at the bottom. The goal is to transfer all these disks to Peg 1. You can move exactly one disk at a time. However, at all times, a bigger disk cannot be placed on top of a smaller one.

Solving this seemingly complex task turns out to be pretty simple. The following Python code prints out instructions that use the fewest number of moves—in other words, the best possible solution!

```
def solve_hanoi(n, from_peg, to_peg, aux_peg):
  if n>0:
    solve_hanoi(n-1, from_peg, aux_peg, to_peg)
    print('Move disk', n-1, 'from', from_peg, 'to', to_peg)
    solve_hanoi(n-1, aux_peg, to_peg, from_peg)

solve_hanoi(n, 0, 1, 2)
```

If you follow these steps, you can show that you'll use $2^N - 1$ moves in all. Now this is a large number and carrying out all the steps seems like eternity. So the monks, out of boredom, came up with a puzzle for you: Let's assume that they've strictly followed the instructions the Python program above generated. Given an intermediate configuration, can you figure out how many more steps they are going to need before completing the task?

Figure 7.1 shows all the configurations obtained by following the Python program's instructions for $N = 3$.

Subtask I: You'll begin by showing a useful property. Prove, using mathematical induction, that for any $n \geqslant 0$, `solve_hanoi(n, ...)` generates exactly $2^n - 1$ lines of instructions.

Subtask II: To solve the puzzle, you'll implement a function

```
public static long stepsRemaining(int[] diskPos)
```

that takes in the current positions of the disks and returns the number of steps that remain in the computer-generated instructions. The current positions are given as an array of integers: the length of the `diskPos` indicates how many disks there are, and `diskPos[i]` $\in \{0, 1, 2\}$ indicates the peg at which Disk `i` is. We guarantee that $0 \leqslant$ `diskPos.length` $\leqslant 63$. As examples (bastardizing Java syntax):

- `stepsRemaining({0})` should return 1.
- `stepsRemaining({2, 2, 1})` should return 3.
- `stepsRemaining({2, 2, 1, 1, 2, 2, 1})` should return 51.

(*Hint:* Solve the following inputs by hand: `{2, 2, 0}` and `{1, 2, 0}`. How many moves do we make before we move the largest disk?)

Performance Expectations: We expect your code to return within 1 second and use only a reasonable amount of memory (e.g., don't explicitly generate the whole instruction sequence).

Exercise 7.9. Consider the following program:

```
void printRuler(int n) {
    if (n > 0) {
        printRuler(n-1);
        // print n dashes
        System.out.println("-".repeat(n));
        printRuler(n-1);
    }
}
```

We would like to know the total number of dashes printed for a given n. If we are to write a recurrence for that, we will get

$$g(n) = 2g(n-1) + n, \text{ with } g(0) = 0,$$

where the additive n term stems from the fact that we print exactly n dashes in that function call.

It may seem hopeless to try to solve this recurrence directly, but you may recall that the number of instructions taken to solve tower of Hanoi on n discs is given by the recurrence

$$f(n) = 2f(n-1) + 1, \text{ with } f(0) = 0.$$

It is known that $f(n)$ has a closed-form of $f(n) = 2^n - 1$.

The two recurrences are strikingly similar. In this problem, we'll analyze $g(n)$ using our knowledge of $f(n)$. Since $f(n)$ and $g(n)$ have similar recurrences, differing only in an n term, we're going to guess that

$$g(n) = a \cdot f(n) + b \cdot n + c \tag{7.3}$$

The following steps will guide you through determining the values of a, b, and c—and verifying that our guess indeed works out. It is especially instructive to show your work carefully.

(i) We'll first figure out the value of c. What do you get when plugging in $n = 0$ into equation (7.3)? It helps to remember that $f(0) = g(0) = 0$. (*Hint:* c should be 0.)

(ii) To figure out the values of a and b, we'll plug in $g(n)$ from equation (7.3) into the recurrence $g(n) = 2g(n-1) + n$. You should be able write it as

$$\underbrace{(\ldots)}_{=P} n + \underbrace{(\ldots)}_{=Q} = 0$$

and solve for a and b such that $P = 0$ and $Q = 0$.

Keep in mind: Because $f(n) = 2f(n-1) + 1$, we know that $f(n) - 2f(n-1) = 1$.

(iii) Derive a closed form for $g(n)$.

(iv) Use induction to verify that your closed form for $g(n)$ actually works.

Exercise 7.10. *(adapted from a problem in ACM Regionals, Greater NY 2008)*

This problem will give you more practice in writing recursive programs, in the context of solving a wacky problem.

Let $N > 0$ be an integer. We say that a list X of positive integers is a *partition* of N if the elements of X add up to exactly N. For example, each of `[1,2,4]` and `[2,3,2]` is a partition of 7.

As you might know already, a list is *palindromic* if it reads the same forward and backward. Of the above example partitions, `[1,2,4]` is not palindromic, but `[2,3,2]` is palindromic. What's more, we know that if X is palindromic, then the *first half* (precisely the first `len(X)/2` numbers) is the reverse of the last half (precisely the last `len(X)/2` numbers).

In this task, we're interested in partitions that are palindromic recursively. A partition is *recursively palindromic* if it is palindromic itself and its first half is recursively palindromic or empty. For example, there are 6 recursively palindromic partitions of 7:

```
[7], [1,5,1], [2,3,2], [1,1,3,1,1], [3,1,3], [1,1,1,1,1,1,1]
```

Your task: Implement a function `int countRPal(int N)` that takes as input a number N (an integer between 1 and 100, inclusive) and returns a list of all recursively palindromic partitions of N. We will only test your function with N between 1 and 100 (inclusive). As an example, calling `countRPal(7)` should return 6. *(Hint: There are* 9,042 *partitions that are recursively palindromic for* $N = 99$*.)*

Performance Expectations: We expect your code to be reasonably fast. On machines in year 2020, for the largest N (i.e., $N = 99$), your program should not take more than 2 seconds.

Chapter Notes

The idea of inductive proofs is ancient, dating back to Plato's era. Induction is a now-standard tool in mathematical reasoning, as well as in the study

of programming languages. For excellent lessons on inductive reasoning, readers are encouraged to look at *Invitation to discrete mathematics* [MN98] and *Mathematics for Computer Science* [LLM15].

As a programming tool, recursion—the counterpart of inductive reasoning in computing—is widely adopted as a programming strategy. It is also an indispensable tool in algorithms design and reasoning about algorithms and data structures. The classic textbook on algorithms by Udi Manber [Man89] discusses using induction neatly as a primary means to derive and think about algorithms. Readers are recommended to read excellent chapters (recursions and backtracking) in a recent book by Jeff Erikson [Eri19] for further and more advanced examples.