### 1.3.4 Induction and Recursion

Failure to find a counterexample to a given algorithm does not mean "it is obvious" that the algorithm is correct. A proof or demonstration of correctness is needed. Often mathematical induction is the method of choice.

When I first learned about mathematical induction it seemed like complete magic. You proved a formula like $\sum_{i=1}^{n} i = n(n+1)/2$ for some basis case like 1 or 2, then *assumed* it was true all the way to $n-1$ before proving it was true for general $n$ using the assumption. That was a proof? Ridiculous!

When I first learned the programming technique of recursion it also seemed like complete magic. The program tested whether the input argument was some basis case like 1 or 2. If not, you solved the bigger case by breaking it into pieces and *calling the subprogram itself* to solve these pieces. That was a program? Ridiculous!

The reason both seemed like magic is because recursion *is* mathematical induction. In both, we have general and boundary conditions, with the general condition breaking the problem into smaller and smaller pieces. The *initial* or boundary condition terminates the recursion. Once you understand either recursion or induction, you should be able to see why the other one also works.

I've heard it said that a computer scientist is a mathematician who only knows how to prove things by induction. This is partially true because computer scientists are lousy at proving things, but primarily because so many of the algorithms we study are either recursive or incremental.

Consider the correctness of *insertion sort*, which we introduced at the beginning of this chapter. The *reason* it is correct can be shown inductively:

- The basis case consists of a single element, and by definition a one-element array is completely sorted.
- In general, we can assume that the first $n-1$ elements of array $A$ are completely sorted after $n-1$ iterations of insertion sort.
- To insert one last element $x$ to $A$, we find where it goes, namely the unique spot between the biggest element less than or equal to $x$ and the smallest element greater than $x$. This is done by moving all the greater elements back by one position, creating room for $x$ in the desired location. ∎

One must be suspicious of inductive proofs, however, because very subtle reasoning errors can creep in. The first are *boundary errors*. For example, our insertion sort correctness proof above boldly stated that there was a unique place to insert $x$ between two elements, when our basis case was a single-element array. Greater care is needed to properly deal with the special cases of inserting the minimum or maximum elements.

The second and more common class of inductive proof errors concerns cavallier extension claims. Adding one extra item to a given problem instance might cause the entire optimal solution to change. This was the case in our scheduling problem (see Figure 1.7). The optimal schedule after inserting a new segment may contain

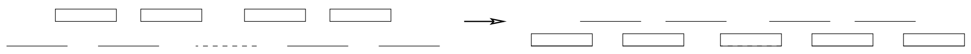

Figure 1.7: Large-scale changes in the optimal solution (boxes) after inserting a single interval (dashed) into the instance

none of the segments of any particular optimal solution prior to insertion. Boldly ignoring such difficulties can lead to very convincing inductive proofs of incorrect algorithms.

*Take-Home Lesson:* Mathematical induction is usually the right way to verify the correctness of a recursive or incremental insertion algorithm.

## Stop and Think: Incremental Correctness

*Problem:* Prove the correctness of the following recursive algorithm for incrementing natural numbers, i.e. $y \rightarrow y + 1$:

```
Increment(y)
    if $y = 0$ then return(1) else
        if ($y$ mod 2) = 1 then
            return($2 \cdot Increment(\lfloor y/2 \rfloor)$)
        else return($y + 1$)
```

*Solution:* The correctness of this algorithm is certainly *not* obvious to me. But as it is recursive and I am a computer scientist, my natural instinct is to try to prove it by induction.

The basis case of $y = 0$ is obviously correctly handled. Clearly the value 1 is returned, and $0 + 1 = 1$.

Now assume the function works correctly for the general case of $y = n-1$. Given this, we must demonstrate the truth for the case of $y = n$. Half of the cases are easy, namely the even numbers (For which ($y$ mod 2) $= 0$), since $y+1$ is explicitly returned.

For the odd numbers, the answer depends upon what is returned by $Increment(\lfloor y/2 \rfloor)$. Here we want to use our inductive assumption, but it isn't quite right. We have assumed that `increment` worked correctly for $y = n - 1$, but not for a value which is about half of it. We can fix this problem by strengthening our assumption to declare that the general case holds for all $y \leq n-1$. This costs us nothing in principle, but is necessary to establish the correctness of the algorithm.

Now, the case of odd $y$ (i.e. $y = 2m + 1$ for some integer $m$) can be dealt with as:

$$\begin{aligned} 2 \cdot \mathit{Increment}(\lfloor (2m+1)/2 \rfloor) &= 2 \cdot \mathit{Increment}(\lfloor m + 1/2 \rfloor) \\ &= 2 \cdot \mathit{Increment}(m) \\ &= 2(m+1) \\ &= 2m + 2 = y + 1 \end{aligned}$$

and the general case is resolved. ▮

## 1.3.5 Summations

Mathematical summation formulae arise often in algorithm analysis, which we will study in Chapter 2. Further, proving the correctness of summation formulae is a classic application of induction. Several exercises on inductive proofs of summations appear as exercises at the end this chapter. To make these more accessible, I review the basics of summations here.

Summation formula are concise expressions describing the addition of an arbitrarily large set of numbers, in particular the formula

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) + \ldots + f(n)$$

There are simple closed forms for summations of many algebraic functions. For example, since $n$ ones is $n$,

$$\sum_{i=1}^{n} 1 = n$$

The sum of the first $n$ integers can be seen by pairing up the $i$th and $(n-i+1)$th integers:

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n/2} (i + (n - i + 1)) = n(n+1)/2$$

Recognizing two basic classes of summation formulae will get you a long way in algorithm analysis:

- *Arithmetic progressions* – We already encountered arithmetic progressions when we saw $S(n) = \sum_{i}^{n} i = n(n+1)/2$ in the analysis of selection sort. From the big picture perspective, the important thing is that the sum is quadratic, not that the constant is 1/2. In general,

$$S(n,p) = \sum_{i}^{n} i^p = \Theta(n^{p+1})$$