Invariants

William E. Skeith III

In our last lecture, we explored the process of writing a program from start to finish. Here's a high level summary:

1. We laid out some abstractions (the post-it notes and rules) to help us think in terms that will be easy to translate into programs.
2. Using our cleverness and intuition, we found a convincing sketch of a solution in terms of the above abstractions.
3. We scrutinized the details of our sketchy solution, realized it was a bit incomplete (the business about writing $-\infty$ on one of the notes at first), and patched up the small logical gaps.
4. We mechanically translated our abstract process to actual program code.

In this lecture, we'll go a bit deeper into the process, and introduce the idea of *invariants*, which can help out with steps 2 and 3 above. The idea can be summarized as follows:

1. Each variable in your program should have a clear **meaning** in your mind.
2. Throughout the program, take care to ensure each variable's **value is consistent with its meaning**.[1]

As simple as this sounds, it is not so easy to do. Having this type of clarity in your thinking is more or less tantamount to proving your program is going to work. Nevertheless it is a worthy goal, so let's get started.

Section 1

Example: Fibonacci Numbers

We will denote the terms of the *Fibonacci sequence* by $\{a_n\}_{n=0}^{\infty}$. If given two adjacent terms in the sequence, you can get the next one just by adding them. More formally:

$$a_{n+1} = \begin{cases} 1, & \text{if } n < 1 \\ a_n + a_{n-1} & \text{else} \end{cases} \tag{1}$$

[1]You may hear the phrase "maintaining the invariant", and this is what it means: keeping values consistent with desired meanings.

The following table gives the first few terms along with their index in the sequence.

n	0	1	2	3	4	5	6	...
a_n	1	1	2	3	5	8	13	...

(2)

The problem we'll address is to find a procedure for computing a_n when given n as input. Perhaps the most obvious or natural way to do this by hand would be to get a pen and paper, write the first few terms, and then just start adding and writing, and adding and writing, producing as much of Table (2) as you need. But as before, we'll introduce some rules which make the problem a little less trivial and more importantly, model some of the limitations you face when implementing this procedure on an actual computer. Let's once again think of this problem as a puzzle or game and again let's use our post-it note analogy. Here are the rules of the game:

1. As before, you can only write one number per post-it note, and you can't rely on your memory—anything you want to remember, you must write down.[2]
2. You should solve the puzzle with a small, fixed number of post-it notes, **irrespective of n**. Let's say 5 or fewer, but any small constant is fine.

We can see at once that the original method of just writing and adding over and over breaks the rules, as it would require an arbitrarily large number of post-it notes. This lecture has just begun, but it's already time for a break! Take a bit of time and see if you can figure out how to solve this problem with just a small number of post-it notes. The reader may appreciate what follows more thoroughly if she has tried to work out the answer on her own first. (Cue Jeopardy music...)

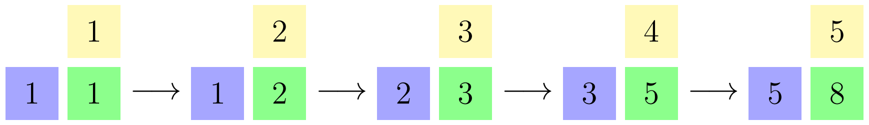
Now that you've had some time to think, let's walk through how a solution might develop. The basic intuition for how to get away with only a few notes is as follows: in order to figure out the next term of the sequence, we don't need the complete history – we only need the last two terms. So perhaps something like this[3] would work:

$$\boxed{1}\ \boxed{1} \longrightarrow \boxed{1}\ \boxed{2} \longrightarrow \boxed{2}\ \boxed{3} \longrightarrow \boxed{3}\ \boxed{5} \longrightarrow \boxed{5}\ \boxed{8}$$

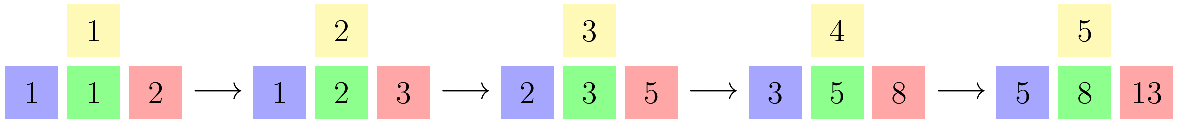
But there are a few problems with this. Can you see them? For one, note that since we can't remember anything we don't write down on a post-it note, we will have no idea what term we're on (what value of n the numbers correspond to). This is an easy fix: just add another post-it note, as follows.

[2]For sufficiently large numbers, this is a reasonable assumption, as most people have trouble remembering more than 10 digits upon hearing them just once.

[3]Here, and in what follows, we illustrate our "solution sketches" by simply showing a few iterations of how the post-it notes' contents would change over time, ignoring the little details of the steps required to make the changes. We will fill in the details as we go.



The yellow note above keeps track of the value of n for the green note below it. It's looking pretty good, but there is one other potential violation of the rules: when we add the contents of the blue and green notes, where do we put it? If we just write it back on the green note, then you are erasing the number you want to put next on the blue note. An easy fix is to add another note (the red one) to store the sum of the blue and green notes.[4]

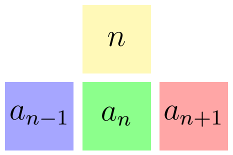


The diagram above is still a pretty sketchy description of the process – for example, at each step, 4 different notes seem to change at once, yet most of us only have 2 hands, and out of those, only one that writes legibly and hence we must prescribe the *order* in which to change the notes. However, we have a good idea of what variables we will need, and that's usually enough to try to formalize invariants.

The question to ask yourself is this: **what do each of my post-it notes mean?** They all have a specific role in the process, and it is helpful to put this into words. Here's how I would write the invariant for what we've done so far. We'll define everything in terms of the contents of the yellow note (denoted by n). No matter what number n is on the yellow note, we want the following to be true:

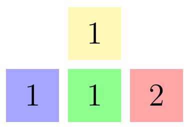
1. The green note contains a_n, where n is the value on the yellow note.
2. The blue note contains a_{n-1}.
3. The red note could be thought of as temporary storage, but it will be used to store the next term, $a_{n+1} = a_n + a_{n-1}$.

Note how at each step our post-its look like a fragment of Table (2), and our view just slides from left to right. Here's a pictorial summary of the invariant (the meaning / role of the variables):

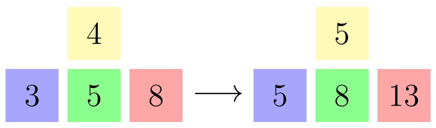


With a good-looking invariant in hand, let's now try to fill out the remaining details of our process. Getting things set up is easy enough: just write a 1 on the yellow, blue and green notes and a 2 on the red one:

[4]Alternatively, you could perform a subtraction to find it, but there is still the issue that it might be difficult to carry out the addition of what's on the green note as you are erasing it...



Notice that the invariant holds with these settings. If $n = 1$, then $a_n = a_{n-1} = 1$ and $a_{n+1} = 2$. Good. Now let's think through the mechanical details of changing the post-it notes' contents. In each "step" of our sketch, all 4 notes change, for example:



Notice that the yellow note always changes in a very simple way—you just add one to it. But the moment you do that, the other notes are "wrong", since their values no longer match their desired meanings (our invariant is broken). Everything on the bottom row is one step behind. Since we can change only one note at a time, which one should change first? Probably the blue one, since the current value on the blue note is no longer needed. So we copy the green to the blue and the red to the green, and lastly, we can add the blue and green together placing the result in the red note. Figure 1 shows the transformation in pictures.

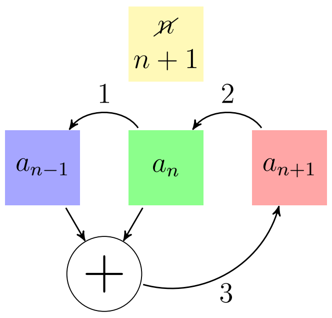
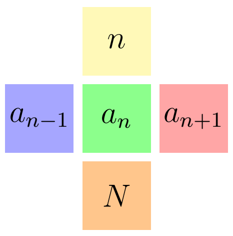


Figure 1: How to maintain the invariant after incrementing n. The numbers above the arrows indicate the order in which they happen (the $n \mapsto n + 1$ is not labeled, as it could happen before or after the other operations).

Let's finally write down the steps of our solution, and while we're at it, we can fill in a few tiny details that were omitted, like the fact that we never stored the input anywhere... To minimize the changes in notation, let's call the input N. We'll use 5 post-it notes for this. One more picture, and then the summary.



1. Get 5 post-it notes, as shown above. Write 1 on the blue, green and yellow notes, 2 on the red, and write the input N on the orange.
2. While the value on the yellow note is less than what's on the orange (while $n < N$), add one to the yellow note and perform the transformation from figure 1. (Note that figure 1 is a 3 step process.)
3. When the yellow note has the same value as the orange ($n = N$), report the contents of the green note as the answer a_N.

Section 2

An Actual Program

If our post-it note solution really followed the rules, then we should have very little trouble expressing the solution in an actual programming language: each step of our post-it note solution has an equivalent in the programming language (this was one of the main points of the first lecture). By now perhaps this is obvious to you, but for completeness, we give complete C++ program code anyway. I would strongly encourage you to program the solution on your own before looking at the code below.

```
#include <iostream>
using std::cin;
using std::cout;

int main()
{
    int n = 1;        /* yellow note: $n$ */
    int abefore = 1;  /* blue note: $a_{n-1}$ */
    int acurrent = 1; /* green note: $a_n$ */
    int aafter = 2;   /* red note: $a_{n+1}$ */
    int N;            /* orange note: input */
    cin >> N; /* read input (we will compute and output $a_N$) */
    /* NOTE: at this point, our invariant is set!  All variables
     * have values consistent with their meanings. */
    while (n < N) {
        n++;
        /* At this point, our invariant is broken.  Let's fix it: */
        abefore = acurrent;          /* figure 1, step 1 */
        acurrent = aafter;           /* figure 1, step 2 */
        aafter = abefore + acurrent; /* figure 1, step 3 */
    }
    /* At this point you know $n$ == $N$, and hence our answer is on the
     * green note which we know (thanks to the invariant!) contains $a_n$ */
    cout << acurrent << "\n";
    /* NOTE: we are making use of the fact that a_0 = a_1, else
     * the answer might be wrong for N = 0. */
    return 0;
}
```

Section 3
Retrospective

The idea of *invariants* was introduced, and an attempt to illustrate their role in problem solving was given. A few final thoughts, and answers to questions you might have:

Where do invariants fit? The Fibonacci example will hopefully be representative: we spent a lot of time sketching out different ideas before trying to write an invariant. Surely the problem solving process is a bit different for everyone, but probably most programmers do not immediately think in terms of invariants. For most people, I think the process begins with an informal, intuitive phase where the main ideas are generated, and invariants come in afterwards to help formalize the process and make sure it is going to work.

Why go to the trouble? There is one particular benefit I would like to point out: if you can think clearly enough about your process to formulate invariants, you will suffer fewer headaches when programming. If anything goes wrong with your code, it is likely a simple misunderstanding of some mechanics of the language. (In fact, it **must** be such a misunderstanding since after all, you have basically proved your process correct!) But if you are not yet comfortable with the language **and** you're not even sure that the idea itself will give correct results, then debugging can be miserable. You have no idea where to begin. You can give `gdb` a try, but if you don't even know what you're looking for, you won't be able to tell when or where the computation has gone sour! If on the other hand, you see that an invariant you thought should hold no longer does, you're likely well on your way to fixing your code. I'm sure not every programmer will agree with this,[5] but here is my advice: **Think first. Write code later.**

Should I worry if I still have no idea what invariants are? Parts of this lecture were admittedly a little meta. Don't worry if you have trouble generalizing the examples you see here right away—it will get easier with experience. It is still valuable for you to start trying to think in these terms now.

[5]Well, probably Donald Knuth wouldn't argue – he once wrote in a comment "Beware of bugs in the above code; I have only proved it correct, not tried it."

Section 4

Practical stuff + exercises

1. Revisit some of our first programs, like computing the max, min and sum of numbers from standard input, and see if you can formalize invariants for them (they'll be very straightforward, but it's still good to think through it). Do the same for the exercise about computing the second smallest number given on standard input.
2. Our C++ program computes one extra term that it doesn't need to (the final contents of the red note are never used). Modify the program to stop sooner and use the red note as output.
3. Can you write a C++ program for the Fibonacci sequence that only uses 4 variables in total? What about 3? For each solution, see if you can formalize how the invariant changes. (*Hint:* for starters, note that you can do without the red note.)
4. (**NOTE:** *this exercise will likely provide a nontrivial challenge for most students. It's just for fun, so don't worry too much if you can't figure it out.*) Note that the number of steps required for us to compute a_N depended linearly on N (that is, #steps $\approx cN$ for some constant c). There is a very clever way to compute a_N using approximately $\log N$ steps.[6] There are two main ingredients you'll need, described below.
   (a) For integers a, b, see if you can find a way to compute a^b using only $\approx \log b$ multiplications. For inspiration, take a look at the following common programming mistake many beginners make when trying to compute powers of a number a:
   ```
   for (int i = 0; i < n; i++) {
       a *= a;
   }
   ```
   What are the intermediate values of a in terms of the original? (You surely don't get a^i!) How could you use them to compute a^b efficiently?
   (b) Also helpful is a matrix algebra formulation of the sequence. Play around with the matrix
   $$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$
   and see if anything jumps out at you as being related to the Fibonacci sequence. (*Hint:* compute $A^2, A^3, \ldots$)
   (c) Combine (a) and (b) above, and you should have a far more efficient algorithm for computing any term of the sequence.

[6] Depending on what you call a "step", it will actually be a small polynomial in $\log N$.