REASONING ABOUT CORRECTNESS

Although this book focuses primarily on the data side of computation, its study cannot truly be separated from the study of procedure. The next two chapters focus on two procedural concepts that are crucial to our study of data structures: reasoning about a procedure's correctness, and reasoning about a procedure's speed.

3.1. Invariants

For most of the algorithms we've seen thus far, their correctness has been fairly obvious, and we haven't needed to argue why each algorithm provided the correct answer. Some algorithms, though, are so clever that we really need a mathematical proof to be convinced that they work. One of the most useful tools for proving that something works is the notion of an *invariant*.

An invariant is a condition that is preserved during the course of executing a program. Invariants are particularly useful for reasoning about loops, where the invariant is a condition that holds at the beginning of each iteration. As an example, consider the following simple method; it's obvious enough to require no proof of its correctness, but we do best to begin by examining a simple loop before progressing to less obvious examples.

```java
public static int maxArray(int[] data) {
    int ret = data[0];
    for(int i = 1; i < data.length; i++) {
        if(data[i] > ret) ret = data[i];
    }
    return ret;
}
```

Here, the loop maintains an invariant that `ret` is the maximum of the first `i` elements of `data`, or more succinctly,

$$\texttt{ret} = \max_{0 \le j < i} \texttt{data}[j].$$

To be an invariant for a loop, two properties should hold. (If you've studied mathematical proof techniques, you will recognize the similarity between these two properties and the two parts of a proof by induction (a basis step and a induction step).)

- The invariant should be true when the computer first hits the **`while`** loop. In this case, at the beginning of the first iteration, `i` is 1, and the invariant asserts that `ret` would be the maximum considering only the first 1 element of `data`. Of course, the maximum among a set with just one number would be that number itself. Initialization of `ret`to `data[0]` prior to entering the **`while`** loop ensures that the invariant holds prior to the first iteration.

- At the beginning of each subsequent iteration, the invariant should hold assuming that it held at the beginning of the previous iteration. When we start an iteration in this example, the new `i` (call it `i`') will be one larger than the `i` of the previous iteration. We have two cases to consider.

  `data[i]` > `ret`:

  Since `ret` was the maximum of the first `i` elements at the iteration's start, and `data[i]` is larger than that, it must be that `data[i]` is the maximum of the first `i` + 1 = `i`' elements. Since `ret` will be `data[i]` after the iteration completes, the invariant will still hold then.

  `data[i]` ≤ `ret`:

  In this case, the program leaves `ret` unchanged. This is the proper thing to do because the maximum of the first `i`' = `i` + 1 elements is the same as the maximum of the first `i` elements.

If we have a loop invariant (which by definition must satisfy these two properties), then we can infer that the invariant will still hold just after the loop terminates. Since `i` will be`data.length` following the loop, the invariant implies that `ret` is the maximum value of the first `data.length` entries — that is, all entries — of the `data` array.

Technically, the argument is missing something. We know that the loop ends when `i` is no longer less than `data.length`, but this doesn't imply that `i` will equal`data.length`: It could be *greater than* `data.length`. To resolve this, we could add another fact to the invariant: that `i` is always less than or equal to `data.length`. This, coupled with the fact that `i` is no longer less than `data.length` after the loop completes, will imply that `i` must equal `data.length`. We must confirm that this additional assertion adheres to the two invariant properties: First, it is true when the computer first hits the **`while`** loop (`i` starts at 1, and `data.length` will be at least that); and second, since we know `i` is less than `data.length` at the beginning of each iteration, adding 1 to the integer value will not make it exceed `data.length`.

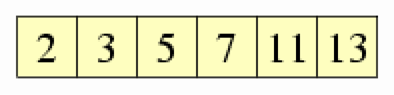
A complete proof of correctness would also include some argument that the loop will eventually terminate. This is not a technicality that will concern us, though.

Note that a single loop can have many valid invariants satisfying the two properties. For example, another invariant to the above loop is `i` ≥ 1: It holds in starting the first iteration (when `i` is 1), and in each subsequent iteration, `i` only increases. (Because of overflow considerations, a complete argument would depend on the fact that it stops once `i` reaches`data.length`.) For that matter, 1 + 1 = 2 is another invariant, as it *is* something that is true at the beginning of each iteration; of course, this is an invariant for all loops, so it's hardly an interesting one. Because there are many possible invariants for a loop, to refer to *the* invariant for a loop is technically invalid.

Despite such intricacies, we nonetheless talk about *the* invariant for a loop. When we do, we are talking about the invariant that encapsulates the important information about what the loop does. In the case of our code for computing the maximum, the first invariant we examined would be this invariant. In short, if you're asked to identify the invariant for a loop, you shouldn't expect anybody to be impressed (or, on a test, to give you any credit) for responding, 1 + 1 = 2.

3.1.1. Case study: Searching an array

Onward to more interesting examples. Consider the problem of determining where within an array of numbers a particular value lies. Our method should return the index of the value, or −1 if the requested value doesn't appear at all in the array. For example, suppose we have the following array.



2	3	5	7	11	13

Given a request to find 5, we want to respond with 2, the index within the array where 5 can be found.

The simple solution, called sequential search, simply starts at the beginning of the array and searches forward until the number is found. If the method reaches the end of the array, it returns −1.

```
public static int sequentialSearch(int[] data, int value) {
    for(int i = 0; i < data.length; i++) {
        if(data[i] == value) return i;
    }
    return -1;
}
```

Here, the invariant to the loop is that `value` does not appear in the first `i` elements of the array `data`. This is trivially true at the beginning, when `i` is 0, so the invariant satisfies the first required property. For the second property, assuming that the invariant holds at the beginning of one iteration means assuming that `value` is not among the first `i` elements. We will reach the next iteration only if `data[i]` — that is, the (`i` + 1)st value of `i` — is also not `value`. Thus, at the beginning of the next iteration, even though `i` has gone up by one, `value` is still not among the first `i` elements — and the invariant still holds.

That invariant was not a difficult example. Notice how it is similar to the invariant of `maxArray`: In both cases, the loop has an index (`i`) that iterates through subsequent integers, and the invariant simply gives information about the cumulative job that the algorithm has completed thus far. Our next example, though, does not follow this model as closely.

Suppose that we want to search an array that we already know is in increasing order. While we could use the sequential search algorithm, the binary search algorithm is a more efficient way to solve the problem. In this algorithm, we keep track of a range of indices (marked by `a` and `c` in our program) where `value` could lie. With each iteration, we take the midpoint of the interval, see how the value in `data` at that index compares to `value`, and discard either the lower half or the upper half based on the result.

It's a simple idea, but the binary search algorithm is notoriously difficult to program. Computer scientist Jon Bentley taught programming seminars to many professionals, and he regularly began by giving the participants half an hour to write the binary search algorithm on paper. He found that, after giving the assignment to over a hundred programmers, about 90% of participants discovered a mistake in their logic while participating in his seminar (Jon Bentley, *Programming Pearls*, Addison-Wesley, 1986, 36). (He was relying on their own reports — he didn't check the other 10% himself.) In another study, Donald Knuth surveyed a broad range of early computing literature and found six different versions published between 1946 and 1958; but he found that the first *correct* binary search code was not published until 1962 (Donald Knuth, *The Art of Computer Programming*, vol. 3, Addison-Wesley, 1973, 419). Here's a version written in Java.

```java
// precondition: data array must be in increasing order
public static int binarySearch(int[] data, int value) {
    int a = 0;
    int c = data.length - 1;
    while(a != c) {
        int b = (a + c) / 2;
        if(data[b] >= value) c = b;
        else                 a = b + 1;
    }
    if(data[a] == value) return a;
    else                 return -1;
}
```

Given the difficulty that others have had writing the program, though, you shouldn't trust this code without a proof. And for that, we need an invariant. Ours will have two parts, which we label (a) and (b).

- (a) For every index *i* where $0 \le i <$ `a`, `data[`*i*`]` is less than `value`; and
- (b) for every index *i* where `c` $\le i <$ `data.length`, `data[`*i*`]` is at least `value`.

For this to be correct, it needs to satisfy the two properties of the invariant.

The first property is that the invariant must hold when we first hit the loop. Clause (a) of the invariant holds trivially: In fact, since `a` starts at 0, there are no indices *i* where $0 \le i <$ `a`, so anything is true for all such indices. We hit a snag, though, when we look at clause (b). Since `c` starts at `data.length` − 1, there *is* an index *i* where `c` $\le i <$ `data.length`— namely, `c` itself. And it could be that `data[c]` is less than `value`. So the invariant doesn't hold.

Hmm. If this code works, the invariant will be something different. We'll shoehorn another clause in to force our candidate invariant to be true at the beginning.

- (a) For every index *i* where $0 \le i <$ `a`, `data[`*i*`]` is less than `value`; and
- *either* (b) for every index *i* where `c` $\le i <$ `data.length`, `data[`*i*`]` is at least `value`, ***or*** **(c) all values in `data` are less than `value`.**

As we reach the loop, `data[c]` will either be less than `value`, or it will be at least `value`. If `data[c]` is less than `value`, then all values in `data` are less than `value` (since `data` must come to us in increasing order), and clause (c) holds true. On the other hand, if `data[c]` is at least `value`, then clause (b) holds. In either case, the entire invariant now holds when we first reach the loop.

Now to the second property of invariants. We'll suppose that the invariant holds at the beginning of the iteration; we want to show that it also holds for the values of `a` and `c`following the iteration. We'll use `a` and `c` to represent the values of the variables' respective values previous to the iteration, and `a`' and `c`' to represent the values at the iteration's end. We have two cases to consider.

`data[b]` ≥ `value`

In this case, `a`' is the same as `a`, and so clause (a) of the invariant will still hold. Note that (c) has nothing to do with `a` or `c`, so that won't change either. But the truth value of (b) depends on `c`, so its truth value may change; we need to show that it won't. Since `c`' will be `b`, `data[c']` will be at least `value`. The array is increasing, so all values in`data` following entry `c`' will be at least `data[c']`, which is itself at least `value`. Thus clause (b) holds, and from that follows the invariant.

`data[b]` < `value`

In this case, `c`' is the same as `c`, so the meaning of clauses (b) and (c) do not change. One of them held as the iteration started, and the same one will hold for the next iteration too. We have only to handle clause (a). Since `a`' is `b`, we know that `data[a']` is less than `value`. Moreover, since the array is increasing, all values in `data` preceding entry `a`' must be at most `data[a']`, which itself is less than `value`. Thus clause (a) holds, and from that follows the invariant.

With both properties proven, we have confirmed that our invariant is valid.

Since the loop coninues as long as `a != c`, `a` and `c` will be equal once the loop completes. Since they'll be equal then, we can replace `c` by `a` in the invariant, and we'll get something that is true once the loop finishes:

- (a) For every index *i* where $0 \le i <$ `a`, `data[i]` is less than `value`; and
- either (b) for every index *i* where `a` $\le i <$ `data.length`, `data[i]` is at least `value`, or (c) all values in `data` are less than `value`.

We now want to know whether the method returns the proper value. For this, note that the invariant implies that either (b) or (c) holds. If (c) holds, then the correct return value is -1, and this is indeed what the method returns, since `data[a]` won't match `value`. The other possibility is that (b) holds. If the entry at `a` matches `value`, then returning `a` (as the method does) is of course correct. But suppose `data[a]` doesn't equal `value`. From (a), we know that nothing in `data` before index `a` matches `value`. We already know from (b) that `data[a]` must be greater than or equal to `value`, and since we're supposing it's not equal, it must be greater. Since the array is increasing, this means that all the values in `data`after index `a` are also greater than `value`. So `value` can't appear before index `a`, at index `a`, or after index `a`. None of the entries of `data` match `value`, and returning -1 is correct.

Source : http://www.toves.org/books/data/ch03-inv/index.html