QUIXEY

- Quixey Blog
- Tech Blog
- Seminar Series

Write Correct Code by Maintaining Invariants

October 24th, 2011 | Posted by Quixey in Technology | Uncategorized

In our engineering interviews, we often ask candidates to write a binary search on paper, and the candidates always do it with `lower` and `upper` variables. But they almost always fail to maintain consistent semantics for these variables, and their algorithms are usually buggy as a result.

For example, this Quixey Challenge problem demonstrates a bug we often see during interviews. Can you find it?

```
def find_in_sorted(arr, x):
    '''
        Returns index of x in arr
        Precondition: arr is sorted
    '''

    def binsearch(lower, upper):
        if lower == upper:
            return -1
        mid = lower + (upper - lower) // 2
        if x < arr[mid]:
            return binsearch(lower, mid)
        elif x > arr[mid]:
            return binsearch(mid, upper)
        else:
            return mid

    return binsearch(0, len(arr))
```

We tell our interview candidates to take their time and make sure their algorithm is correct before we judge it. They often respond that it's impossible to have confidence about their algorithm's correctness before running it. This "guess and check" approach to algorithm design is fine for quick hacks, but we're going to explain why it lacks something important.

Rice's Theorem

You've probably heard of the halting problem, and you've probably seen a proof of why a program's halting behavior is impossible to analyze in the general case. But you might not know there's a profound generalization called Rice's theorem which follows as a simple corollary.

Imagine you're given an arbitrary computer program. You know you might not ever be able to figure out whether it halts, but it seems like you should at least be able to figure out whether its behavior is equivalent to the program `print 'hello world'`, right?

Actually, Rice's theorem says that computer programs are resistant to understanding of any kind in the general case. So no matter how much static analysis and simulation you run on the code, no matter how many different sticks you poke it with, even the seemingly trivial property of being a Hello World implementation is generally impossible to discern.

As a programmer, you have to wade into the Ricean ocean of un-analyzable code, and find a way to float. Each line of code you write is an opportunity to fall off the surface of provable correctness into the chaotic froth.

Invariants

In light of Rice's Theorem, how can you manage to write correct programs? The answer is that you stay away from the Ricean "general case" by maintaining **invariants** – statements about your program that remain true as the program runs.

A common type of invariant you want to maintain is variable semantics. For example, a good binary search might maintain the invariant that the `lower` variable is always less than or equal to the algorithm's final output (except when the binary search turns up empty-handed).

Another common type of invariant applies to the program's control logic. For example, you can declare that any time the value we're binary-searching for isn't present in the input list, control must be routed to a single `return -1` statement. Many engineering candidates fail to maintain this invariant, and their algorithms crash when given an empty list. If you're careful to maintain the invariant above, an empty list won't even be a special case, because your algorithm will handle it the same way it handles an empty search interval demarcated by the values of `lower` and `upper`.

It's important to approach coding problems with invariants in mind. An invariant is like a fragile shell that protects the correctness of your code as long as it's properly maintained. And if you've ever tried gluing a cracked egg back together, you know what it's like to debug an algorithm with no well-maintained invariants.

When we watch an engineering candidate write code, we try to gauge how well they're thinking in terms of invariants. If we see them adding a bunch of special-case logic to their binary search, it's a telltale sign that they haven't developed an invariant-savvy mindset. The best programmers have a high-level understanding of their code, and know how to wield invariants to gracefully avoid off-by-one bugs. That's why we test engineering candidates by asking them to write a perfect binary search on paper.

Like Be the first of your friends to like this. Tweet 4

You can follow any responses to this entry through the RSS 2.0 You can leave a response, or trackback.

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *	
Email *	
Website	
Comment	

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <em> <i> <q cite=""> <strike> <strong>`

Post Comment

- **Twitter Updates**
    -

Follow us on Twitter

- **Categories**
    - App Trends
    - Culture
    - Favorite Apps
    - Functional Search
    - Functional Web
    - Industry News
    - Life-Changing Apps
    - Press
    - Quixey News
    - Seminar Series
    - Technology
    - Technology Trends
    - Tweet Awards
    - Uncategorized

  - Vision

- **Archives**

  - November 2013
  - October 2013
  - September 2013
  - August 2013
  - July 2013
  - June 2013
  - May 2013
  - April 2013
  - March 2013
  - February 2013
  - January 2013
  - December 2012
  - November 2012
  - October 2012
  - September 2012
  - August 2012
  - July 2012
  - June 2012
  - May 2012
  - April 2012
  - March 2012
  - February 2012
  - January 2012
  - December 2011
  - November 2011
  - October 2011
  - September 2011
  - August 2011
  - July 2011
  - June 2011
  - May 2011
  - April 2011
  - January 2011
  - October 2010
  - February 2010
  - December 2009
  - October 2009