Nearly All Binary Searches and Mergesorts are Broken

Joshua Bloch
Google Inc.

I remember vividly Jon Bentley's first Algorithms lecture at CMU, where he asked all of us incoming Ph.D. students to write a binary search, and then dissected one of our implementations in front of the class. Of course it was broken, as were most of our implementations. This made a real impression on me, as did the treatment of this material in his wonderful Programming Pearls (Addison-Wesley, 1986; Second Edition, 2000). The key lesson was to carefully consider the invariants in your programs.

Fast forward to 2006. I was shocked to learn that the binary search program that Bentley proved correct and subsequently tested in Chapter 5 of Programming Pearls contains a bug. Once I tell you what it is, you will understand why it escaped detection for two decades. Lest you think I'm picking on Bentley, let me tell you how I discovered the bug: The version of binary search that I wrote for the JDK contained the same bug. It was reported to Sun recently when it broke someone's program, after lying in wait for nine years or so.

So what's the bug? Here's a standard binary search, in Java. (It's one that I wrote for the java.util.Arrays):

```
1:     public static int binarySearch(int[] a, int key) {
2:         int low = 0;
3:         int high = a.length - 1;
4:
5:         while (low <= high) {
6:             int mid = (low + high) / 2;
7:             int midVal = a[mid];
8:
9:             if (midVal < key)
10:                 low = mid + 1;
11:            else if (midVal > key)
12:                high = mid - 1;
13:            else
14:                return mid; // key found
15:        }
16:        return -(low + 1);  // key not found.
17:    }
```

The bug is in this line:

```
 6:             int mid =(low + high) / 2;
```

In Programming Pearls Bentley says that the analogous line "sets m to the average of l and u, truncated down to the nearest integer." On the face of it, this assertion might appear correct, but it fails for large values of the int variables low and high. Specifically, it fails if the sum of low and high is greater than the maximum positive int value (2^31 - 1). The sum overflows to a negative value, and the value stays negative when divided by two. In C this causes an array index out of bounds with unpredictable results. In Java, it throws ArrayIndexOutOfBoundsException.

This bug can manifest itself for arrays whose length (in elements) is 2^30 or greater (roughly a billion elements). This was inconceivable back in the '80s, when Programming Pearls was written, but it is common these days at Google and other places. In Programming Pearls, Bentley says "While the first binary search was published in 1946, the first binary search that works correctly for all values of n did not appear until 1962." The truth is, very few correct versions

have ever been published, at least in mainstream programming languages.

So what's the best way to fix the bug? Here's one way:

```
 6:             int mid = low + ((high - low) / 2);
```

Probably faster, and arguably as clear is:

```
 6:             int mid = (low + high) >>> 1;
```

In C and C++ (where you don't have the >>> operator), you can do this:

```
 6:             mid = ((unsigned int)low + (unsigned int)high)) >> 1;
```

And now we know the binary search is bug-free, right? Well, we strongly suspect so, but we don't know. It is not sufficient merely to prove a program correct; you have to test it too. Moreover, to be really certain that a program is correct, you have to test it for all possible input values, but this is seldom feasible. With concurrent programs, it's even worse: You have to test for all internal states, which is, for all practical purposes, impossible.

The binary-search bug applies equally to mergesort, and to other divide-and-conquer algorithms. If you have any code that implements one of these algorithms, fix it now before it blows up. The general lesson that I take away from this bug is humility: It is hard to write even the smallest piece of code correctly, and our whole world runs on big, complex pieces of code.

We programmers need all the help we can get, and we should never assume otherwise. Careful design is great. Testing is great. Formal methods are great. Code reviews are great. Static analysis is great. But none of these things alone are sufficient to eliminate bugs: They will always be with us. A bug can exist for half a century despite our best efforts to exterminate it. We must program carefully, defensively, and remain ever vigilant.

Update 17 Feb 2008: Thanks to Antoine Trux, Principal Member of Engineering Staff at Nokia Research Center Finland for pointing out that the original proposed fix for C and C++ (Line 6), was not guaranteed to work by the relevant C99 standard (INTERNATIONAL STANDARD - ISO/IEC - 9899 - Second edition - 1999-12-01, 3.4.3.3), which says that if you add two signed quantities and get an overflow, the result is undefined. The older C Standard, C89/90, and the C++ Standard are both identical to C99 in this respect. Now that we've made this change, we know that the program is correct;)

EOF