## Model the Binary Search Algorithm

Here is a standard binary search, expressed in Java:

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:     }

Seems straightforward, right?

In fact, it has a bug.

When this binary search algorithm is implemented in C the bug causes an array index out of bounds with unpredictable results. In Java, it throws ArrayIndexOutOfBoundsException.

Have you figured out what the bug is?

The bug is in this line:

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

It fails for large values low and high. Specifically, it fails if the sum of low and high is greater than the maximum positive int value (231 - 1). The sum overflows to a negative value, and the value stays negative when divided by two.

This bug can manifest itself for arrays whose length (in elements) is 230 or greater (roughly a billion elements). Such array sizes are common these days.

The binary-search bug applies equally to merge sort, 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.

More … <https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html>

## Alloy Model of the Binary Search Algorithm

The model nicely shows the overflow problem. The model has **Int** fields for low and high:

**one** **sig** Variables {
 low: **Int** **lone** -> Time,
 high: **Int** **lone** -> Time,
 mid: **Int** **lone** -> Time,
 midValue: A **lone** -> Time,
 answer: **Int** **lone** -> Time
}

When the model is run, the size of **Int** is specified, e.g.,

**run** { **some** x: **univ** | binarySearch [x] } **for** 10 **but** 5 **Int**

That specifies **Int** is a 5-bit signed integer, which means its range of values is: -16 to 15.

The **run** command also specifies “for 10,” which means the array is to have 10 values (with indices 0 to 9). Suppose that during the binary search low = 7 and high = 9. The algorithm says to add low and high, then divide by 2.

7 + 9 = 16

But 16 is too large since **Int**’s largest allowable value is 15. So, the sum overflows and the result is negative:

7 + 9 = -16

On the other hand, if the run command specifies 6 **Int**,

**run** { **some** x: **univ** | binarySearch [x] } **for** 10 **but** 6 **Int**

then **Int** is a 6-bit signed integer with this range of values: -32 to 31. For an array of 10 values, the sum of any two indices will never exceed **Int**’s maximum value.

## Lessons Learned

No matter how big we set the size of **Int**, a sufficiently large array will have indices that, when low and high are added together the sum will exceed the maximum value for **Int**. Clearly, computing mid by adding low and high (and dividing by 2) is not the right approach.

## Here’s the Alloy Model

See the examples folder: examples/binary-search/binarySearch-v3.als