**1.Why mid=begin+(end-begin)/2 is better than mid= (begin + end)/2**

**Lot of programmers use formula mid=beg+(end-begin)/2 instead of using the simpler formula mid= (start + end)/2 for finding the middle element in the array or list.**

**Why do they use the former one?**

**(start + end) might overflow, while (end – start) cannot.**

*Because latter does not work when* **start** *and* **end** *are pointer?*

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.

So, what's the bug? Here's a standard binary search, in Java.

1: public static int binary Search(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 (231 - 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 230 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;

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 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. 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.

Resources:

1.https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html