SIZE MATTERS: LESSONS FROM A BROKEN BINARY SEARCH*

Eric Shade
Computer Science Department
Missouri State University
901 S. National, Springfield, MO 65897
EricShade@MissouriState.edu

ABSTRACT

As reported in the Google Research Blog, nearly all binary search and merge sort implementations are broken because of an integer overflow bug. The bug does not appear unless the array being searched/sorted is quite large (over a billion elements), which is why it has remained undetected for decades. There is a tendency to dismiss such overflow bugs as mere trivia, but this is a mistake. This paper reviews the bug, some common but incorrect solutions, and a correct solution. More important, it distills lessons to be learned from the bug, and provides specific recommendations for computer science educators to help their students cope with such bugs in the future.

INTRODUCTION

This paper is divided into three parts, comprising seven sections. The first part summarizes the Google Research Blog post announcing an integer overflow bug in the classical binary search algorithm. The second part looks at the general problem of computing the average of two integers (which is where the overflow occurs), reviews several non-solutions to the problem, then provides a general solution. The third part of the paper (1) explores the likelihood that this bug, or one like it, will be with us for some time; (2) discusses programming language features that can eliminate it; and (3) provides several specific recommendations for computer science educators to ensure that their students are well-prepared to avoid or cope with such bugs in the future.

BROKEN BINARY SEARCH

Here is the standard binary search algorithm [2] expressed in Java, in which *int* is a 32-bit two's-complement signed integer type and arrays use zero-origin indexing. A formal specification of this algorithm is beyond the scope of this paper; we will assume that it is required to work correctly for any array that will fit in main memory and whose length (number of elements) is representable as an *int*.

```
bool binarySearch(int[] A, int target) {
    for (int lo = 0, int hi = A.length – 1; lo <= hi; ) {
        int mid = (lo + hi) / 2;
        if (target == A[mid]) return true;
        else if (target < A[mid]) hi = mid – 1;
        else lo = mid + 1;
    }
    return false;
}
```

As reported in the Google Research Blog, this code has a bug [3]. If the array has length $2^{30} + 1 =$ 1,073,741,825 or larger, then the sum *lo*+*hi* may be as large as $2x(2^{30}) = 2^{31}$, which overflows to a negative integer since the largest representable positive integer is $2^{31}– 1$. Note that this is not a little "corner" bug; it affects arrays of over a billion different lengths (those strictly between 2^{30} and 2^{31}).

One might argue that binary search is an idealized algorithm that assumes idealized arithmetic, and that this is a flawed implementation of that idealized algorithm. That is a philosophically defensible view, but it is no help in practice. The author has seen binary search presented in dozens of textbooks. Among those that gave concrete implementations in a programming language with fixed-precision integers, all contained the bug (assuming the preconditions above). Among those that expressed the algorithm in language-independent pseudocode, none had an effective explanation of the overflow problem. In practice, a programmer is almost certain to copy, transcribe, or reinvent an implementation with the same bug.

THE BLOGOSPHERE REACTION

The blog post has 31 moderated comments, which seems to be a very small number, though several dozen other blogs/articles linked to it. The comments fall into several categories: surprise that the bug has lain dormant for so long, misunderstandings about the bug, failed attempts to fix the bug, and dismissals of the bug as irrelevant or an implementation detail. One wrote, "Also, who the &#$% does a binary search on an array of billions of items?"

The notion that the bug is irrelevant is deeply flawed. First, integer overflow is an explanation but not an excuse. Assuming our preconditions that the array can be stored in memory and the size is representable as an *int*, then there is no *a priori* reason why binary search should not work. It can and does work if implemented properly. There is a prevalent but mistaken notion that because computer arithmetic is only an approximation to "real" arithmetic, any problems related to it are somehow inevitable or

beyond a programmer's control. Second, the notion that efficient in-memory algorithms cannot or should not be used on large data structures is ludicrous. A billion is approximately 2^{30}, so a binary search of an array that size will take only about 30 steps, which is blindingly fast. Many computers today have 2GiB or more of RAM, which is enough to store a billion-element byte array (some space will be taken by the operating system and other applications). 5GiB is enough to store a billion-element *int* or *float* array. Desktop computers can be purchased today with 32GiB of RAM. The mere fact that the bug was discovered means that *somebody* wanted to search such a large array.

COMPUTING THE AVERAGE OF TWO INTEGERS

The general problem is to compute the average of two integers *a* and *b*; assume that they are both 32-bit signed *ints* or both 32-bit unsigned *ints*. Though Java only supports signed *ints*, C and C++ support unsigned *ints*, so it is worth having a solution that works for both. Of course one could always cast *a* and *b* to a larger type (say, a 64-bit *long*) and use the obvious formula (*a*+*b*)/2, but since most languages only provide a small set of integer types, there may not always be a larger type to cast to. Further, since the average must lie between *a* and *b*, it is reasonable to expect that it can be computed without using higher precision, and in fact it can be.

Recall that &, |, and ^ are bitwise "and", bitwise inclusive-or, and bitwise exclusive-or. C, C++, and Java all have a >> right-shift operator that shifts either zeros or copies of the sign bit in from the left, depending on the type (not value) of the left operand. Since Java's primitive integer types are all signed, >> is always a signed shift, so it also provides the >>> unsigned right-shift operator.

The obvious formula to compute the average of *a* and *b* is (*a*+*b*)/2. It fails if *a* and *b* are very large, because the sum *a*+*b* will overflow. This is the cause of the binary search bug. For example, if *a* is one billion and *b* is two billion, the computed average is –647,483,648, despite the fact that the correct average of 1500 million is representable as a 32-bit signed *int*.

If *a* and *b* are both *ints* with nonnegative values, then either (*a* + *b*) >>> 1 or *a* + (*b*–*a*)/2 will correctly compute the average. The former is the solution presented in the Google blog; in that solution, *a*+*b* may overflow into 32 bits, but the unsigned right shift brings it back to 31 bits. In the latter solution, the difference *b*–*a* cannot overflow. (Recall that a sum cannot overflow if the operands have different signs; equivalently, a difference cannot overflow if the operands have the same sign.)

The author prefers the formula (*a* + *b*) >>> 1 because >>> is a warning that something tricky is going on. Since *a* + (*b*–*a*)/2 is algebraically equal to (*a*+*b*)/2, it's possible that someone might "optimize" the former to the latter, not realizing that the order of operations is crucial. (For that matter, a highly-optimizing compiler might erroneously do the same thing.)

If *a* and *b* are *ints* that may be negative, then neither formula will work in general. The first formula will always produce a positive result since it shifts a zero into the sign bit. In the second formula, *b*–*a* may overflow if *a* and *b* have different signs.

If *a* and *b* are both unsigned, then again neither solution will work in general: in the first formula, *a*+*b* may overflow to 33 bits, and in the second, *b*–*a* will "wrap" to a large positive integer if *b* is less than *a*. For example, if *a* = 1 and *b* = 0 (as 32-bit unsigned *ints*), then

a + (*b*–*a*)/2 = 1 + (0–1)/2 = 1 + 4294967295/2 = 1 + 2147483647 = 2147483648

The simplest formula that will work correctly in all cases is

(*a*&*b*) + (*a*^*b*)/2,

which requires four machine instructions and cannot overflow. The derivation is simple and relies on the basic fact that *a*+*b* = (*a*&*b*) + (*a*|*b*); the term *a*&*b* computes the individual bit carries and the term *a*|*b* computes the individual bit sums [1,5,7]. Recall that (*a*|*b*) = (*a*^*b*) + (*a*&*b*). Then

a+*b* = (*a*&*b*) + (*a*|*b*) = (*a*&*b*) + ((*a*^*b*) + (*a*&*b*)) = 2*(*a*&*b*) + (*a*^*b*),

and therefore

(*a*+*b*)/2 = (2*(*a*&*b*) + (*a*^*b*))/2 = (*a*&*b*) + (*a*^*b*)/2.

The most-significant bits of *a*&*b* and *a*^*b* must be different, so the sum cannot overflow.

A RECURRING PROBLEM

This bug and its solution seem destined to be rediscovered many times. First, as indicated above, memory sizes on modern hardware easily allow data structures big enough to trigger the bug to fit in memory. It may still be a while before such large data structures are common, of course. The inevitable transition to 64-bit computing will not help, because in many languages such as Java, *int* will still be a 32-bit type. The rules of C/C++ are loose enough that *int* could be a 64-bit type on 64-bit machines, but it may well remain at 32 bits for backwards compatibility with 32-bit code.

Second, it seems unlikely that a proper solution will appear in many textbooks. Binary search is a very simple algorithm, and the broken version is "obviously" correct. It would take longer to clearly explain the overflow issue (especially for a solution using bitwise operations) than it does to explain the algorithm as a whole. The author expects to see the broken version for many years to come, perhaps with a footnote that one must take care to avoid overflow in computing the average.

Third, though computer programmers know intellectually that overflow and roundoff errors can happen, there is still a tendency to think in terms of idealized arithmetic, especially for formulas as trivial as (*a*+*b*)/2. That formula shouts "average of *a* and *b*" as if it were a single concept.

Fourth, it is not clear how the solution can be effectively disseminated to working programmers. C, C++, and Java all provide binary search methods, but not all programmers use them, either because they don't know they're available or because they're perceived as too slow (they involve a call to a comparator for each element probed). And this won't help for other algorithms that rely on computing averages. A more direct approach for Java is to provide methods Integer.*average*(*int*, *int*) and

Long.*average*(*long*, *long*) that encapsulate the solution, but again, how are programmers to know they exist? (Regrettably, Java does not have such methods, nor does any other language known to the author.) Why would programmers look for such methods when a simple (though incorrect) formula is readily available?

This is a general problem. For example, in [4], numerical analyst William Kahan explains why the high-school formula for the area of a triangle can give wildly inaccurate results in some cases, provides numerically stable formulas of comparable complexity, then questions how programmers are to learn of their existence when the high-school formula is mathematically correct (assuming exact arithmetic) and has "withstood the test of time." Though Kahan's paper is concerned with inaccuracies in floating-point arithmetic (a huge problem that deserves more attention), the basic issues of identifying the problem in an "obvious" algorithm and disseminating the solution are the same.

RETURN OF THE BIGNUMS?

Some programming languages, including most Lisp dialects and some scripting languages like Python and Ruby, provide built-in arbitrary-precision integers. (In Lisp terminology, the integers are divided into *fixnums*, which are small enough to fit in a machine word along with some tag bits, and *bignums*, which are not.) In such a language, the binary search bug disappears. However, the prevalent attitude about bignums has been that they are an esoteric and expensive luxury, best provided via a class/library (such as Java's BigInteger) if at all. But bignums are useful in modern applications; cryptographic algorithms are but one example. And the overhead of bignum arithmetic is modest: all modern CPUs provide cheap tests for overflow, and together with good branch prediction, the fixed overhead for bignum arithmetic can be reduced to one or two machine instructions.

The author believes that built-in bignums are worth the price for general software development. (Having a separate library is better than nothing, but unless using bignums is just as simple as using *ints*, programmers aren't likely to use them.) With the modern and powerful high-level languages we have today, it is ridiculous that programs should fail due to simple arithmetic overflow. Humans should not be wasting their time worrying about how many bits fit into an *int* or how best to add two numbers; that is a job for computers. Unfortunately, the most widely-used languages use fixed-precision integers and can't easily be changed without breaking millions of lines of existing code, so we must cope with fixed-precision integers for the foreseeable future.

There is a close analogy with garbage collection, which was long considered to be too slow for general use. *Real* programmers managed their own memory. But it is now widely understood that garbage collection is a huge productivity boost for programmers, eliminates most if not all memory errors, and imposes an acceptable cost in space and time. It is suitable for all but the most demanding real-time applications.

RECOMMENDATIONS FOR COMPUTER SCIENCE EDUCATORS

This author was surprised by the bug, but readily understood it and was aware of the solution. A much bigger concern is whether today's computer science students are

capable of identifying and fixing such bugs. (The number of incorrect solutions in the comments on the Google blog was disheartening.) Whether or not you view this particular bug as significant, it is representative of a large class of integer arithmetic bugs that are likely to be exposed in the next decade as we transition from 32-bit to 64-bit computing and more often reach the limits of 32-bit integers. We as computer science educators must provide our students with the fundamental skills to cope with such bugs. Some educators may feel that the solution above using bitwise operators is too "tricky" to deserve coverage in a curriculum, but surely we can agree that students should be capable of understanding the bug and researching a solution. Though we are faced with pressure, both from students and employers, to teach more features, we cannot abandon essential computer science concepts. Following is a set of concrete recommendations.

Teach the powers of two. Require students to memorize the powers of two up to 1024, and to know that 2^{10} is approximately a thousand (or one K), 2^{20} is approximately a million (or one M), 2^{30} is approximately a billion (or one G), and 2^{40} is approximately a quadrillion (or one T). Require them to know these facts throughout the curriculum, not just in one course. If a quiz or exam only requires arithmetic using powers of two, do not allow them to use a calculator. Armed with these facts, it will be obvious to students that you can't use an *int* to store the capacity of a hard drive, and that 2GiB of RAM is plenty of space to store a billion bytes.

Teach students that "int" means more than just "no decimals." Some textbooks oversimplify the distinction between *int* and *float* (or *double*) in this way. An *int* cannot have decimals, but it also has a limit. Since your students now know their powers of two, you can explain what this limit is. The limit is not meaningless trivia, it is an essential property of the type.

Show students examples of overflow and how to cope with it. Seeing it makes it real. A computer architecture course is an obvious place to discuss overflow, but it's also important for students to see overflows "in the wild", using their usual programming language. Emphasize that the possibility of overflow does not make integer arithmetic imprecise or unpredictable. Integer arithmetic is perfectly well defined and offers no surprises to those who understand it.

Teach the difference between int, long, and bignums. The author has seen students write programs using only *longs* in a misguided attempt to avoid arithmetic errors. The idea is that *long* is somehow "safe" because it is the widest primitive integer type. This doesn't make all arithmetic errors go away; rather, it leads to a false sense of security and unnecessarily large and slow programs. There is no way to avoid the dangers inherent in fixed-precision integers. Students must learn those dangers and learn how to select an appropriate integer type. If no fixed-precision type suffices, they must know how to use bignums.

Require students to learn at least one language with native bignum support. This can easily be done in a programming languages course. If a new generation of programmers comes to appreciate bignums, it will accelerate their adoption in new languages or revisions to existing languages. For example, it's not hard to imagine that some future version of Java will provide "auto-BigInteger" support, in which integer literals too large to fit in a *long* are automatically coerced to a BigInteger, and BigIntegers may be freely mixed with other integer types using standard operators.

Review core libraries. Despite the fact that students generally prefer to write as few lines of code as possible, the author finds that they often tend to reinvent the wheel. Though the GUI and data structure libraries are of course important, a trip through the core libraries can be highly profitable. For example, Java's wrapper classes Byte, Short, Integer, and Long provide a number of useful methods like *bitCount* and *signum*, and fields like *SIZE* and *MAX_VALUE*; the Collections class is an often-overlooked treasure trove of methods like *binarySearch*, *copy*, *frequency*, *sort*, and *swap*.

Use large test cases when grading programs. When using only small data sets, every data structure is compact enough and every algorithm is fast enough. Students who graduate never having manipulated a data structure larger than a few dozen elements will have no concept of space and time and will be ill-prepared for the real world. It will not even occur to them that someone could or might want to work with huge data sets. (Recall the blog commenter who asked why anyone would use binary search on such a large array.) Using large data sets makes both space and time real. If the data structures are too sloppy, they may not fit in memory. If the algorithm is too slow, it will take too long to run. For example, if you require students to implement a sorting algorithm in time $O(n \log n)$, how can you be sure that they meet the time bound if you only use small arrays? If you sort an array of size 100000, the difference between an $O(n \log n)$ algorithm and an $O(n^2)$ one will be readily apparent.

CONCLUSION

There are several reasons why the binary search bug is noteworthy. First, it has lain dormant for decades in a simple algorithm that has heretofore been treated as gospel. Second, it manifests itself on large arrays, with sizes beyond what most programmers work with on a regular basis, but well within the memory limits of modern computers. Programmers who have not encountered such large arrays may very well do so within the next few years. Third, most programmers and computer science students use high-level programming languages that abstract away many low-level details such as memory management, which can lead to a tendency to dismiss *all* low-level details as trivia. But fixed-precision integers and their sizes are low-level details that are *not* abstracted away in most languages, and are the source of the bug. Fourth, computer science educators are under pressure to teach more “stuff”, which can lead to a decreased emphasis on fundamentals. But without an understanding of the fundamentals, such bugs cannot easily be understood, anticipated, or fixed. This paper has explored several lessons to be learned from the binary search bug in light of these considerations.

REFERENCES

[1] Beeler, M., Gosper, R.W., and Schroeppel, R., *HAKMEM,* MIT AI Memo 239, 1972.

[2] Bentley, J., *Programming Pearls 2/e*, Boston, MA: Addison-Wesley, 1999.

[3] Bloch, J., Extra, Extra – Read All About It: Nearly All Binary Searches and Mergesorts are Broken, *Google Research Blog*,

http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html, June 2006.

[4] Kahan, W., Miscalculating Area and Angles of a Needle-like Triangle, http://www.cs.berkeley.edu/~wkahan/Triangle.pdf, March 2000.

[5] Ohannessian, R., Bob's page of mildly useful but still pretty neat code snippets, http://bob.allegronetwork.com/prog/tricks.html#average, August 2006.

[6] Spolsky, J., The Perils of JavaSchools, http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html, December 2005.

[7] Warren Jr., H.S., *Hacker's Delight*, Upper Saddle River, NJ: Addison-Wesley, 2003.