How many tests can you create and execute in an eight-hour workday? Five? Ten? Twenty-five? In a single day, your computer can generate and run tens of billions of tests. That's nine orders of magnitude more testing than humanly possible, all for the cost of electricity to keep the machine going. That's power. That's leverage. And that's why you should have a computer do your testing.

Four Steps to High-Volume Test Automation

We use an approach we call Behavior/Inputs/Outputs/Steering (BIOS) for creating our high-volume automated tests:

1. Describe the desired behavior of the system under test. What is the system supposed to do?
2. Choose how to generate inputs. How can we mechanically generate input values?
3. Design a way for the computer to verify system outputs. How can we mechanically verify results?
4. Steer the test generation toward areas you want to test. How can we focus the testing on areas we think are important?

Example 1: Testing a Square Root Function

Let's apply the BIOS approach to testing a simple square root function.

Step 1: Behavior: What is the system supposed to do?

The square root of a number is a non-negative value that, when multiplied by itself, gives the number. For example, the square root of 16 is 4.

The input domain for this particular function is all floating-point numbers from 0 to 1 billion.

Step 2: Inputs: How can we mechanically generate input values?

From those floating-point numbers, our test program selects two different sets of inputs:

1. Proceed systematically through a subset of values (such as 0, 1, 2, 3, …)
2. Choose sample values throughout the range

Step 3: Outputs: How can we mechanically verify results?

Because a square root is a non-negative number that, when multiplied by itself, gives the input value, our test program verifies two outcomes:

1. The output is greater than or equal to zero
2. Squaring the output produces a number very close to the input value (allowing for computer precision)

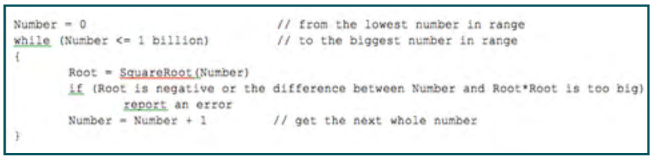
Step 4: Steering: How can we focus the testing on areas we think are important?

Our highest priority for these tests will be whole numbers, followed by other floating-point inputs. For the moment, we don't care about invalid inputs such as negative numbers or non-numeric strings. Based on our BIOS steps, we design two test programs:

1. Test program A systematically walks through all whole numbers from 0 to 1 billion
2. Test program B randomly samples floating-point numbers between 0 and 1 billion

Test Program A

Because we have only a billion whole numbers to test, we can simply walk through the list. A program based on the pseudocode in figure 1 is straightforward to implement and finishes in about ten seconds on a typical computer.

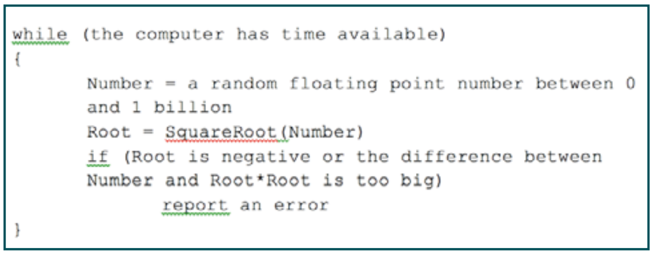


```
Number = 0                          // from the lowest number in range
while (Number <= 1 billion)         // to the biggest number in range
{
        Root = SquareRoot(Number)
        if (Root is negative or the difference between Number and Root*Root is too big)
                report an error
        Number = Number + 1         // get the next whole number
}
```

Figure 1: Pseudocode to systematically test whole numbers from 0 to 1 billion

Test Program B

Most programming languages support sixteen digits of precision, so there are about ten quintillion floating-point values between 0 and 1 billion. That's way too many to walk through. Figure 2 shows how to randomly sample values throughout the input range. Because there is no hard stopping point to our input generation, we will let the program run for as long as we don't have anything more important for our computer to do. If randomized inputs make you uneasy, take comfort in knowing that a a typical computer can test tens of billions of square roots per day, which is great coverage, no matter how you look at it.



```
while (the computer has time available)
{
        Number = a random floating point number between 0
        and 1 billion
        Root = SquareRoot(Number)
        if (Root is negative or the difference between
        Number and Root*Root is too big)
                report an error
}
```

Figure 2: Pseudocode to test floating-point numbers between 0 and 1 billion

What did we learn from testing the square root function?

- The test programs took only a few minutes to design and write but provided outstanding test coverage because the computer did the grunt work for us for free.
- Generating billions of test cases relieves us of the burden of trying to guess a specific value that might cause the function to show an error.
- We do not store test cases; we generate them as needed. This means we don't maintain a big inventory of static tests.
- We can steer the testing in new directions as required.

Example 2: Testing a Sorting Routine

Now that we've seen the basics of high-volume testing, let's see how it fares in a real-world example.

In 2004, a library of sorting routines called Nsort appeared on the web. [1] The library was well received and was eventually recommended in the book *Windows Developer Power Tools*. [2]

In 2013, we wondered whether a simple high-volume approach could find useful bugs in these published routines. We tried out high-volume testing on the fifteen sorting routines in the Nsort library, as well as a related implementation from another website. We revisit that experience here, following the four BIOS steps.

Step 1: Behavior: What is the system supposed to do?

Given a set of elements such as {5, 2, 6, 3, 1, 4}, a sorting routine should return the same elements arranged in order (e.g., {1, 2, 3, 4, 5, 6}).

Step 2: Inputs: How can we mechanically generate input values?

We can generate sets of various sizes by starting with an ordered set (e.g., {1, 2, 3, 4, 5, 6}) and shuffling that set into a random order, such as {5, 2, 6, 3, 1, 4}.

Step 3: Outputs: How can we mechanically verify results?

Our verification step leverages the fact that shuffling a set is easier than sorting it. When the shuffled set created in step 2 is handed to a sort routine, the resulting set should be identical to the original, ordered set.

Step 4: Steering: How can we focus the testing on areas we think are important?

For this example, we are interested in sets of different sizes. We will systematically increment set size from one element to five thousand elements. At each set size, we will perform several shuffles before moving on. We are not currently interested in duplicate elements, and we will limit the sort to increasing order.

Test Program C

Figure 3 shows pseudocode for testing the sort routines. Small sets sort quickly, and it is possible to generate all permutations. Large sets take longer to sort and have many more permutations, so the number of shuffles tested will be limited.

As we kicked off the test runs for the sixteen different sort routines, we wondered whether such a simple approach could find useful bugs. We were soon pleasantly surprised to find significant bugs in four of the routines, as shown in Table 1.

Why You Should Think and Let Your Computer Test

Most test automation today is stuck in the slow lane, with computers rigidly following scripts handcrafted by humans. High-volume automation breaks that mindset, allowing computers and humans to exploit their different strengths. Human ingenuity and computer horsepower can produce cheap, powerful testing that needs to be the next step for the software industry.

Isaac Asimov sounded a similar note thirty years ago in his essay "Intelligences Together" [3]:

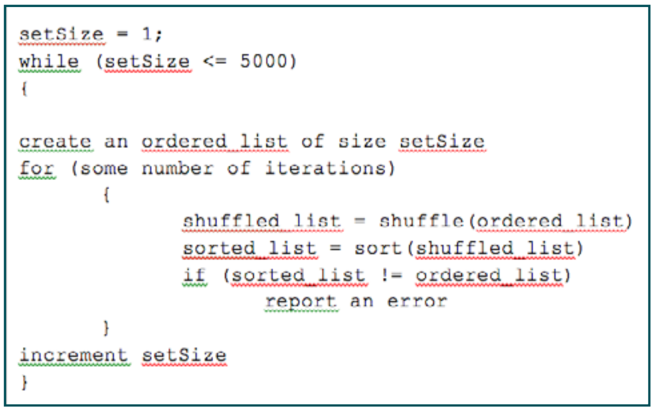
"Computers … are extraordinarily good in some ways. They possess capacious memories, have virtually instant and unfailing recall, and demonstrate the ability to carry through vast numbers of repetitive arithmetical operations without weariness or error…

"The human specialty … is the ability to see problems as a whole, to grasp solutions through intuition or insight; to see new combinations; to be able to make extraordinarily perceptive and creative guesses.

"Each variety of intelligence has its advantages and, in combination, human intelligence and computer intelligence—each filling in the gaps and compensating for the weaknesses of the other—can advance far more rapidly than either one could alone."

Asimov's insight rings true in software testing. No tester can execute a billion test cases by hand, and no computer can think up good test cases by itself. But testers and computers working together form a very powerful team.

We invite you to try out this style of testing yourself. A program to test the ShearSort routine is available for download. [4]



```
setSize = 1;
while (setSize <= 5000)
{

create an ordered list of size setSize
for (some number of iterations)
      {
            shuffled list = shuffle(ordered list)
            sorted list = sort(shuffled list)
            if (sorted list != ordered list)
                  report an error
      }
increment setSize
}
```

Figure 3: Pseudocode to test a sorting routine

Sort Routine Name	Example Input	Incorrect Output	Code Issue
FastQuickSort	1, 4, 5, 3, 2, 6	Unhandled Exception	Error decrementing array index
OddEvenTransportSort	2, 3, 1	2, 1, 3	Condition statement used "<" vs "<="
ShearSort	1, 2, 3, 4, 5, 6	1, 2, 3, 5, 4, 6	Condition statement used "<" vs "<="
ShearSortB	9, 2, 7, 4, 5, 6, 3, 8, 1	1, 2, 4, 3, 6, 7, 5, 8, 9	Condition statement used "<" vs "<="

Table 1: Summary of bugs found by automated high-volume testing of sorting routines

Drop the file contents into a Visual Studio Console project and see for yourself how easy and effective this style of testing can be. {end}

harryr@harryrobinson.net
doug.szabo@gmail.com

Sticky Notes

Click here to read more at StickyMinds.com.
- References

Wanted! A Few Great Writers!

I am looking for authors interested in getting their thoughts published in *Better Software*, a leading online magazine focused in the software development/IT industry. If you are interested in writing articles on one of the following topics, please contact me directly:

- Testing
- Agile methodology
- Project and people management
- DevOps
- Configuration management

I'm looking forward to hearing from you!

Ken Whitaker

Editor, *Better Software* magazine

kwhitaker@TechWell.com