Perl script helps users survive aptitude tests

BOWLING FOR PERL

themaster444, Pixelio.de

Hardly anyone writes complicated algorithms anymore, but aptitude tests often require that candidates have the theory on tap. **BY MICHAEL SCHILLI**

The end of the credit crunch is nigh; time to apply for a new job! Once the economy gets back into gear, corporations will again have money burning holes in their pockets. And, to land the job of their dreams, candidates will have to dust off their CVs and bone up on questions posed at job interviews.

Bowling Ball Drop

A popular question that software companies in Silicon Valley love to ask applicants for programming jobs is this: Given two identical bowling balls – from which floor of a 100-story building could you drop either of them without it breaking? The idea of this task is to minimize the number of drop attempts required in the worst case.

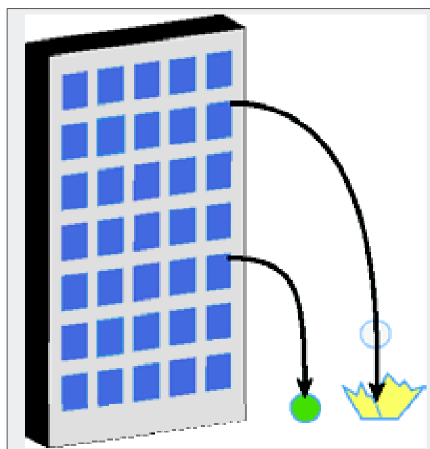

Figure 1: From which floor will the bowling ball survive the drop? Given 100 floors and a total of two balls, a maximum of 15 attempts will suffice.

A naive method would be to start on the first floor, drop the ball, and see if it survives. If it does, you would repeat the experiment on the second floor and slowly work your way up to the 100th floor. In the worst case, that is, if the bowling balls were tough enough to survive a drop from the 100th floor, you would need 100 drops to find the result.

Recall that you have two bowling balls, and you're allowed to break both of them, but you need to identify the critical floor precisely when you break the second one. (Hint: 15 attempts are all you should take – unless you want to fail the aptitude test and look for a job elsewhere.)

If you think you have what it takes to become a software developer in Silicon Valley, stop reading now, imagine the not totally stress-free scenario of a job interview, and keep your cool while you work through the options in your head. The clock's ticking …

The Decisive Trick

As I mentioned previously, a linear approach will solve the problem, but it takes far too many iterations. A binary search would be similar: If you divide the 100 floors by two and drop the first ball from the 50th floor, you would need 49 further drops to find the critical floor in a worst case scenario. After all, you're not allowed to break the second ball, if you don't have the answer afterwards.

The trick here is to divide the 100 floors into stretches of gradually decreasing size (Figure 1). The first stretch has a length of n floors, the second $n - 1$, the third $n - 2$, and so on, until one stretch includes the 100th floor. Using this division, you would work your way through the stretches from left to right, dropping the first bowling ball from the first floor of the stretch you are currently working on. If the ball breaks, you would drop the second ball from the second floor in the previous stretch (if this exists) until it breaks, or you reach the end of the stretch. This identifies the critical floor before you run out of balls.

The interesting property about the division shown in Figure 2 is that it always limits the number of attempts to identify the critical floor to a maximum of $n + 1$, no matter where the ball eventually breaks. If the critical point is, say, the last floor in the first stretch, the first ball would break on the second drop all told,

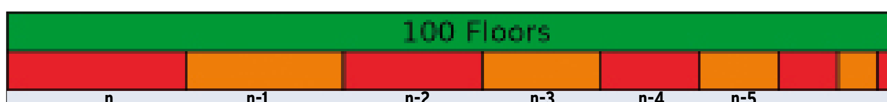

Figure 2: Dividing the 100 floors into stretches of continually decreasing size.

$$\frac{n(n+1)}{2} \geq 100$$

$$n^2 + n - 200 \geq 0$$

$$n_{1/2} = \frac{-1 \pm \sqrt{1^2 + 4 \cdot 1 \cdot 200}}{2} = 13.65, -14.65$$

Figure 3: The condition turns into a quadratic equation that can be solved via a well-known formula [2].

from the first floor of the second stretch. After this, the researcher would drop the second ball from the second to the *n*th floor of the first section. It would survive until the end, giving you *n* as the highest floor. The number of attempts here is two with the first ball, and $n - 1$ with the second, giving you a total of $n + 1$.

If the critical floor is the last floor in the fifth stretch, which has a length of $n - 4$, the first ball would break on the sixth attempt, from the first floor of the sixth stretch. Then, you only need $n - 4 - 1$ attempts to try the remaining floors in the fifth stretch because this stretch has a length of $n - 4$ and you previously tried the first floor without breaking the ball. The maximum number of attempts if the last ball drops safely from the last

Figure 4: When n = 14, the floors divide into these stretches.

floor in stretch five is $6 + n - 4 - 1$ (i.e., $n + 1$).

Searching for n

The algorithm is perfectly clear and performs at or better than $n + 1$ attempts in the worst case. But how do you decide on the value of *n* for a building with 100 floors? The task is to reduce the maximum number of attempts to a minimum, so this means that small values for *n* are preferred. However, the size of the stretches start at *n* drops in steps of 1, and if a stretch has a value of 1, without the sum total of all stretches reaching 100, you've run short.

Thus, the sum total of $n + (n - 1) + (n - 2) + \ldots + 1$ should be precisely 100 if possible. It can be above 100, but not below it. As you might be aware, the German mathematician Carl Friedrich Gauss presumably defined a formula for this series of numbers while he was still a kid. His teachers asked him to add up the numbers between 1 and 100, and the genial Gauss told them the answer within seconds without performing a single addition [1].

So, according to Gauss, $n(n + 1)/2 >= 100$. This leads to a quadratic equation, and the only positive solution for it, on the basis of the boilerplate math I was taught in school, is 13.65. Figure 3 shows how I solved it on my whiteboard. The smallest acceptable value for *n* is thus 14, and Figure 4 shows the derived distribution of the floor stretches I was looking for.

Molded in Perl

The bball-drop program in Listing 1 shows an implementation of the algorithm in Perl [3]. Line 10 determines the value for *n* by solving the quadratic equation and rounds up the resulting floating point value to the next integer with the *ceil()* function from the POSIX module.

Listing 1: bball-drop

```
01 #!/usr/local/bin/perl -w
02 use strict;
03 use POSIX qw(ceil);
04 use Log::Log4perl qw(:easy);
05
06 Log::Log4perl->easy_init(
07                      $INFO);
08 my $total = 100;
09
10 my $n = ceil(
11   ( -1 +
12     sqrt(1 + 4 * 2 * $total)
13   ) / 2
14 );
15
16 my $sum   = 1;
17 my @stops = ();
18 push @stops, $sum;
19
20 while ($sum + $n <= $total) {
21   push @stops, $sum + $n;
22   $sum += $n;
23   $n--;
24 }
25
26 my $last_ok_floor = (
27   defined $ARGV[0] ?
28   $ARGV[0] : 42 );
29
30 INFO "Pst, pst: Highest OK ",
31   floor is ", $last_ok_floor;
32
33 my $tries       = 0;
34 my $ok_floor    = 1;
35 my $smash_floor = $total + 1;
36
37 for my $stop (@stops) {
38
39   $tries++;
40   if ( !try_floor(
41     $stop, $last_ok_floor)) {
42     $smash_floor = $stop;
43     last;
44   }
45   $ok_floor = $stop;
46 }
47
48 for my $try_floor (
49   $ok_floor + 1 ..
50   $smash_floor - 1) {
51
52   $tries++;
53   if ( !try_floor(
54     $try_floor, $last_ok_floor
55   )) {
56     $smash_floor = $try_floor;
57     last;
58   }
59   $smash_floor =
60     $try_floor + 1;
61 }
62
63 INFO "Highest OK floor: ",
64   $smash_floor - 1,
65   " ($tries tries)";
66
67 ###############################
68 sub try_floor {
69 ###############################
70   my ($floor, $last_ok_floor)
71     = @_;
72
73   if($floor > $last_ok_floor){
74     INFO "Floor $floor: ",
75          "Wham, busted!";
76     return 0;
77   }
78
79   INFO "Floor $floor: Okay.";
80   return 1;
81 }
```

```
$ bball-drop 14
2009/10/10 13:36:05 Pst. pst: Highest OK floor is 14
2009/10/10 13:36:05 Floor 1: Okay.
2009/10/10 13:36:05 Floor 15: Whoa, busted!
2009/10/10 13:36:05 Floor 2: Okay.
2009/10/10 13:36:05 Floor 3: Okay.
2009/10/10 13:36:05 Floor 4: Okay.
2009/10/10 13:36:05 Floor 5: Okay.
2009/10/10 13:36:05 Floor 6: Okay.
2009/10/10 13:36:05 Floor 7: Okay.
2009/10/10 13:36:05 Floor 8: Okay.
2009/10/10 13:36:05 Floor 9: Okay.
2009/10/10 13:36:05 Floor 10: Okay.
2009/10/10 13:36:05 Floor 11: Okay.
2009/10/10 13:36:05 Floor 12: Okay.
2009/10/10 13:36:05 Floor 13: Okay.
2009/10/10 13:36:05 Floor 14: Okay.
2009/10/10 13:36:05 Highest OK floor: 14 (15 tries)
$
```

Figure 5: Worst case, the algorithm needs 15 steps.

```
$ prove ./suite
./suite....ok
All tests successful.
Files=1, Tests=202,  7 wallclock secs
( 0.06 usr  0.02 sys +  5.31 cusr
  1.12 csys =  6.51 CPU)
Result: PASS
$
```

Figure 6: The test suite confirms that the script returns correct results for any combination of floors and never takes more than 15 attempts.

The *while* loop in lines 20-24 then creates an array, *@stops*, whose elements are set to the number of the first floor in a particular stretch. That is, for n = 14, *@stops* contains the values 1, 15, 28, ..., 91, 96, 100. The script calls the last floor from which the bowling ball survives the drop the *Highest OK floor* and accepts this number at the command line for test purposes (you would call the script by entering *bball-drop 99* to test the scenario in which the bowling balls can survive floor 99, for example, but no higher), or it defaults to the value of 42 set in line 28.

Log4perl in line 30 whispers the value to be discovered by the algorithm (*Pst, pst*) to the caller, then the script starts to work through the individual test cases. The *for* loop in lines 37-46 goes to the start of a stretch in the *@stop* array and calls the *try_floor()* routine with the floor to test and the secret maximum floor as arguments. If the tested height is greater than the maximum height, *try_floor()* returns a false to signify that the bowling ball you dropped has shattered into a thousand pieces.

If the *for* loop in lines 37-46 finds a floor that the first ball does not survive, it terminates by calling *last* and sets the *$smash_floor* variable to the floor that caused the damage. The algorithm then continues, with the next *for* loop in lines 48-61 taking the next floor from the stretch with the last successful drop and climbing up one floor at a time until the second ball bites the dust, or the end of the stretch is reached. If the ball breaks, the loop terminates with *last*, and the result is available in *$smash_floor*. Reducing this value by *1* gives you the highest floor from which the ball would survive the drop. If the end of the stretch is reached, the last successful floor is the searched-for maximum.

Check Your Results!

With delicate problems like this, bugs are almost inevitable, so you'd better verify the results with a regression test suite. The *suite* program (Listing 2) uses the Sysadm::Install module from CPAN to call the *bball-drop* script repeatedly with different maximum floor values between *0* and *100*. The *tap()* function calls the script and captures STDOUT, STDERR, and the script's return code. In a typical run, the bball-drop script prints out something like Figure 5, and the regular expression in line 14 picks up the result with the highest floor the bowling ball survived and the total number of steps required to ascertain the result.

The two Test::More commands in lines 19 and 22 verify whether the result identified by the script matches the preset test case parameter and whether the script keeps to the maximum permitted number of 15 attempts. The Test::More module from CPAN provides tried and trusted output in TAP format (*1 ok* for successful cases and *2 not ok* for unsuccessful ones). Reading this with the naked eye could be cumbersome, so the *prove* script, which comes with the module and with more recent Perl distributions, wraps a test harness around the TAP output to confirm that all 202 tests completed successfully (Figure 6). This is much easier than manually checking the 200+ lines output by *suite* for errors. As it turns out, all tests pass with flying colors. The candidate scores full points and can look forward to a successful career in IT! ■

Listing 2: suite

```
01 #!/usr/local/bin/perl -w
02 use strict;
03 use Sysadm::Install qw(:all);
04 use Test::More;
05
06 plan tests => 202;
07
08 for my $floor (0 .. 100) {
09
10  my ($stdout, $stderr, $rc) =
11    tap "bball-drop", $floor;
12
13  if ($stderr =~
14     /floor: (\d+) \((\d+)/) {
15
16   my ($result, $tries) =
17     ($1, $2);
18
19   is($floor, $result,
20    "result: $result $tries");
21
22   ok(
23    $tries <= 15,
24    "result: $result $tries"
25   );
26
27  } else {
28   die "Unmatched: $stderr";
29  }
30 }
```

INFO

[1] How young Johann Carl Friedrich Gauss solved adding integers in arithmetic progression: *http://en.wikipedia.org/wiki/Gauss#Early_years_.281777.E2.80.931798.29*

[2] Quadratic equation: *http://en.wikipedia.org/wiki/Quadratic_equation*

[3] Listings for this article: *ftp://www.linux-magazin.de/pub/listings/magazin/2009/12/Perl*