Etude 5 Numbers

Group:

Oliver Westenra

Shaye McKaye

MIDPOINT

Q1) What is integer overflow?

When a calculation exceeds the maximum size that the integer type can store. 8-bit integers have a maximum value of 127 so anything higher than that will loop around to the lowest possible value that the integer can be, which is -128. As the number of bits increase the highest and lowest possible values get further away from zero but overflow can still occur.

Q2)How do you detect it?

Using the position of most-significant one-bit in the binary equivalent of the values being used. The maximum number of one-bits that the binary value of an integer in Java can have is 32, so if you use binary multiplication/addition/division/subtraction on the values you want to calculate and the result has a one-bit in the 33rd position or higher then you know that integer overflow has occurred.

Q3)How do you prevent it?

If both the values are negative then it applies the Math.abs() function to make sure the sum cannot go too low and result in integer underflow. If the low is greater than zero we use a subtraction formula to ensure that the sum cannot get too high and result in overflow.

Q4)Why are you confident your code is correct?

We have functions to test all the laws that are defined in the brief, and they all make mathematical sense. The methods described in the prevention question above make sure that overflow cannot happen as the equations can never get high or low enough for over or under flow to occur.

Q5)How hard was it to write a bulletproof version of this little function?

I rewrote my entire code because the original way I had was very confusing and then I found a much easier way to write the function. Translating this way into code was difficult but still easier than my original code.

Q6)When is it a good idea to use ‘safe’ languages (Lisp, Smalltalk, Python) and when is it a good idea to use numerically ‘unsafe’ languages (Java)?

If the program is going to be dealing with relatively small numbers or doing small computations then Java is fine. If there is a chance that the program will be dealing will large numbers or computations then probably best to use Lisp, Python etc as they deal with larger numbers better.

HYPOTENUSE

Q7)

Richard outlined what some outcomes mean in his write up for this etude. The result of the 15th example did not equal zero because the result of hyp() was as close to z as the formula could go. The final example overflowed and so the output became Infinity because the float data type could not store enough to hold that value.

0. 0.000000e+00

1. 0.000000e+00

2. 0.000000e+00

3. 0.000000e+00

4. 0.000000e+00

5. 0.000000e+00

6. 0.000000e+00

7. 0.000000e+00

8. 0.000000e+00

9. 0.000000e+00

10. 0.000000e+00

11. 0.000000e+00

12. 0.000000e+00

13. 0.000000e+00

14. 0.000000e+00

15. 1.073742e-07

16. 0.000000e+00

17. 0.000000e+00

18. 0.000000e+00

19. inf

Q8)

As the numbers get smaller it gets harder and harder to accurately calculate the square root of smaller numbers so the relative error gradually gets worse as is evident in output 21, 22 and 23 before the program finally underflows in the 24th example.

The 3 issues:

- The final iteration underflowed because it exceeded the minimum value that a float can hold.

- Subnormal numbers, in this case, are numbers that have an absolute value that is less than the absolute value of floats smallest normal number. The output contains some of these subnormal numbers which means that the output cannot have full precision.

- Round-off errors occur in this program because it cannot have complete precision when dealing with increasingly small numbers so the calculations get affected by round-off error and so the output cannot be exactly the expected output.

0. 0.000000e+00

1. 0.000000e+00

2. 0.000000e+00

3. 0.000000e+00

4. 0.000000e+00

5. 7.275958e-08

6. 9.094948e-08

7. 1.136869e-07

8. 1.421086e-07

9. 0.000000e+00

10. 1.110223e-07

11. 1.387779e-07

12. 8.673618e-08

13. 1.084202e-07

14. 6.776264e-08

15. 8.470330e-08

16. 1.058791e-07

17. 6.617446e-08

18. 8.271807e-08

19. 1.033976e-07

20. 1.292470e-07

21. 1.138989e-05

22. 1.138383e-03

23. 5.879122e-02

24. 1.000000e+00

Q9)Rerun all tests

25 iterations, dividing x, y and z by 10 after each iteration:

This alteration has prevented the underflow from the previous question and the relative error is relatively similar across all samples, except the ones that returned 0. Hypot() exists to make sure that we aren’t attempting to square root numbers that are less than 1. When square rooting numbers less than 1 eventually the result will end up equalling 1.

0. 0.000000e+00

1. 0.000000e+00

2. 7.450581e-08

3. 0.000000e+00

4. 1.164153e-07

5. 7.275958e-08

6. 9.094948e-08

7. 1.136869e-07

8. 1.421086e-07

9. 8.881785e-08

10. 1.110223e-07

11. 1.387779e-07

12. 8.673618e-08

13. 1.084202e-07

14. 6.776264e-08

15. 8.470330e-08

16. 1.058791e-07

17. 1.323489e-07

18. 1.654361e-07

19. 1.033976e-07

20. 6.462349e-08

21. 8.077936e-08

22. 1.009742e-07

23. 1.262177e-07

24. 7.888609e-08

Q10)

When I call the heron\_area function it overflows and so the numbers that are being calculated are being looped around to the minimum value of float. This results in a NaN value being outputted.