Computers good with numbers? это курам на смех!

This étude has you looking at three simple calculations and writing a report in which you explain what goes wrong and what you can do about it. All the programming must be done in C, C++, or Fortran (with the -ffloat-store compiler option if you are using gcc) or in Java using public strictfp class or Scala using the @strictfp annotation or the /fp:strict compiler option with C♯.

Midpoint

Let's define midpoint$(x, y) = (x + y)/2$. That's pretty simple if you are dealing with rational numbers or the mathematical real numbers. But we're going to think about binary search and bitmapped displays and consider integers, and $(0 + 1)/2$ isn't an integer. So let's define

$$\begin{aligned} \text{trunc}(v) &= \text{sign}(v) \times \text{floor}(\text{abs}(v)) \\ \text{midpoint}(x, y) &= \text{trunc}(\frac{x + y}{2}) \end{aligned}$$

Note that we are defining the midpoint of two ints to be an int. When midpoint(2, 3) returns 2 instead of 2.5, that's because it is *supposed* to.

Integer division in Java[1] is defined to truncate towards 0. That's exactly what we want here. In C integer division is implementation-defined and we have to use the library function div to get portable code.

Last time this étude was used, this section presented a series of definitions, asked you to test them, and to explain the results of the tests. The cause was always *INTEGER OVERFLOW*. This time you have to come up with a definition that works. Your report must be accompanied by a source file so that we can re-run your tests.

Here are some laws that midpoint must satisfy and *your program must test*. You should look for more.

- The definition of midpoint(x, y) is symmetric, so for any x and y midpoint$(x, y) =$ midpoint(y, x).

[1] See section 15.17.2 of the JLS

- If $x \leq y$ then $x \leq \text{midpoint}(x, y) \leq y$.
- Put another way, $\min(x, y) \leq \text{midpoint}(x, y) \leq \max(x, y)$
- As a consequence, if x and y are any legal ints whatsoever, $\text{midpoint}(x, y)$ is defined. Explain carefully why your definition never experiences integer overflow.
- As another consequence, $\text{midpoint}(x, x) = x$.
- $\text{midpoint}(x, -x) = 0$.
- $\text{midpoint}(x, 0) = x/2$.
- $\text{midpoint}(x, y) = -\text{midpoint}(-x, -y)$, provided x, $-x$, y, and $-y$ are all representable integers, since truncation towards zero is symmetric around zero.

You are to submit a source file with your implementation(s) of midpoint() and your test code.

Your implementation of midpoint() **must** use only plain 32-bit integer arithmetic without overflow detection; you should *know* that no overflows will occur in your code.

Your test code **must** try many values (too many for a human to check) and **must not** rely on a human being scrutinising printed output. Use your programming language's version of an "assert" statement.

You are to submit a report. (See the COSC326 web page about about reports in general.) For this section, the report will cover

(Q1) What is integer overflow?

(Q2) How do you detect it? (In C and Java 1.7 this is a trick question.)

(Q3) How do you prevent it?

(Q4) Why are you confident your code is correct? Note: you **must** have code in your program to **test** the laws listed above for a sufficiently large and wide sampling of x and y.

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

(Q6) Question for reflection: if it's this hard to get a simple thing like midpoint right, how much confidence can we have in *any* integer calculation without formal verification or testing? Programming languages like Lisp, Smalltalk, Python, and Perl6 support integers with no fixed

size limit; when is it a good idea to use a "safe" language and when is it a good idea to use a numerically "unsafe" language like Java?

Hypotenuse

Given two points (x_1, y_1) and (x_2, y_2) in the Euclidean plane, the distance between them is $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. The C library has a function called hypot to compute $\sqrt{x^2 + y^2}$ and java.lang.Math has the same function. Convenience is only one of the reasons for having it.

Here is a definition, using single precision floats. You will also need -ffloat-store in C (or C++) if using gcc, or strictfp in Java.

```
#include <math.h>

float hyp(float x, float y) {
    return sqrtf(x*x + y*y);
}
```

This is obviously right, right? So let's test it. We know that $3^2 + 4^2 = 9 + 16 = 25 = 5^2$.

```
#include <stdio.h>

int main(void) {
    float x = 3.0f, y = 4.0f, z = 5.0f;
    for (int i = 0; i < 20; i++) {
        float e = fabsf(hyp(x, y) - z)/z;
        printf("%2d  %e\n", i, e);
        x *= 10.0f, y *= 10.0f, z *= 10.0f;
    }
    return 0;
}
```

(Q7) Compile and run this program. The point of -ffloat-store/strictfp is to ensure that you get the same answers I do. When I run it, two of the relative error answers are not zero. Why aren't they zero? (Note: we are only interested in the correctness of hyp(). The relative error calculation is what it is, and it does not go wrong. This question is about why hyp() isn't exactly equal to z.)

(Q8) Change the program to divide by 10.0f instead of multiplying and change 20 to 25, so that we're testing with small to very small numbers. Most of the answers are not zero. There are at least three different things happening here. Which iterations are affected by what issues?

In an attempt to deal with some of these issues, people generally write

```
float hyp(float x, float y) {
    float a = fabsf(x), b = fabsf(y);
    if (a > b) {
        b = b/a;
        return sqrtf(1.0f + b*b)*a;
    } else
    if (a < b) {
        a = a/b;
        return sqrtf(1.0f + a*a)*b;
    } else { // This works even when a == b == 0.0f
        return a*sqrtf(2.0f);
    }
}
```

(Q9) Rerun all the earlier test cases. What, if anything has changed? What does this tell you about why hypot() exists?

Include your answers to Q7–Q9 in your report, and *show the numbers*. You must show the numbers *readably*. That means no screen-shots, no funny colouring, just plain black text on a white background like the rest of the document, in neat tables.

Heron's formula

Suppose we are given a triangle with sides a, b, and c. Heron's formula for the area of a triangle is

$$\sqrt{s(s-a)(s-b)(s-c)}$$

where

$$s = \frac{a+b+c}{2}$$

Suppose we are given an isosceles triangle, where $a = b$. The area of a triangle is one half the base times the altitude, and in this case the base is c and the altitude is h where

$$\begin{aligned} h &= \sqrt{a^2 - (c/2)^2} \\ &= a\sqrt{1 - (c/(2a))^2} \end{aligned}$$

So we have two formulas for the area of an isosceles triangle.

```
float heron_area(float a, float c) {
    float s = (a+a+c)/2.0f;
    return (s-a)*sqrtf(s*(s-c));
}

float baseht_area(float a, float c) {
    float d = c/(2.0f*a);
    return sqrtf(1.0f - d*d)*a*c*0.5f;
}
```

(Q10) Test these functions with $c = 1$ and $a = 1, 10, \ldots, 10^{19}$. One of these functions goes catastrophically wrong; which one and why?

Task

Include your answer to Q10 in your report, and *show the numbers.*

(Pair 2)

Notes

Computers were originally built to do numeric calculations. In our computer architecture and operating systems paper you were taught how computers represent 32-bit integers and floating-point numbers. In this étude you experience some consquences of those representations.

As noted, programming languages like Lisp and Smalltalk have supported integers with no fixed limit, and rational numbers too, for a long time.

Some more modern languages do too. Java has BigInteger, but no rational numbers. C++, using the GMP library, has big integers and rational numbers, and using the MPFR library, it even supports floating-point numbers of whatever precision you want. Computer Algebra systems like SAGE (which is open source) make such libraries especially easy to use. **It doesn't have to be like C!**

The issues you need to watch out for are

- division by zero (integer and floating point), which happens not to occur in this étude;
- overflow (integer and floating point);
- underflow;
- subnormal numbers;
- catastrophic cancellation; and
- roundoff error.

A common mistake is for students to attribute everything to roundoff error, and another common mistake is to assume that using wider precision fixes everything.

Testing

You can't afford to test midpoint() exhaustively for 32-bit x and y, but you *could* exhaustively test a 16-bit version (or, in C, an 8-bit version using signed char). You *must* test values near the limits INT_MIN and INT_MAX. C defines INT_MIN and INT_MAX in ⟨limits.h⟩.

This étude shows that very simple tests can uncover serious problems. Don't ever think you can't come up with useful tests.

It also shows the importance of testing. The initial version of hyp is "obviously correct", but isn't, because floating point arithmetic does not have unbounded range or unbounded precision. (Java's BigDecimal lacks square roots, so we can't use that.)