CSC207H Lecture 11

Sadia Sharmin

Nov 22, 2016

Some simple arithmetic

- `1.5 + 0.5 = ?`
- `1.25 + 1.0 = ?`
- `0.1 + 0.1 + 0.1 = ?`

Let's try printing these out in Java.

Is Java broken?

Its not Java. Check this out in Python:

```
>>> x = 0.1
>>> sum = x + x + x
>>> print sum == 0.3
False
>>> sum
0.30000000000000004
>>> bigger = 1.0
>>> s = 1.0e-6
>>> sum1 = s + s + s + s + s + s + s + s + s + s + bigger
>>> sum2 = bigger + s + s + s + s + s + s + s + s + s + s
>>> sum1 == sum2
False
>>> sum1
1.00001
>>> sum2
1.0000099999999992
```

What's going on?

- Let's talk a little bit about how numbers are stored in computer hardware vs. how we usually think about them (binary vs. decimal).
- First, consider an int like 42. Hardware doesnt directly represent 4s or 2s – everything is binary.
- `42 = 1x2^5 + 0x2^4 +1x2^3 + 0x2^2 + 1x2^1 + 0x2^0`
- So 42 can be represented by 101010 (base 2).
- What about fractions?

Binary vs. Decimal Fractions

- Floating-point numbers are represented in computer hardware as base 2 (binary) fractions.
- For example, the decimal fraction `0.125` has value `1/10 + 2/100 + 5/1000`
- Similarly, the binary fraction `0.001` has value `0/2 + 0/4 + 1/8`
- These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Source:
https://docs.python.org/3/tutorial/floatingpoint.html

Binary vs. Decimal Fractions

- What is $1/10$ as a decimal number?
- What is $1/3$ as a decimal number?
- What is $1/10$ as a binary number?

Binary vs. Decimal Fractions

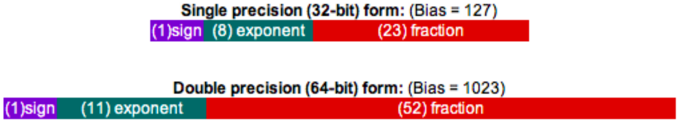
- Just like how $1/3$ does not have an exact decimal representation, many fractions do not have an exact binary representation
- For $1/3 = 0.333333$, no matter how many digits you add, you will never represent exactly $1/3$, but a better and better approximation
- In binary, 0.1 in decimal is equal to the infinitely repeating fraction 0.0001100110011001100110011001100110011...
- We have finite memory. How can we represent numbers that take an infinite number of bits?

Historical aside

- 30 years ago, computer manufacturers each had their own standard for floating point.
- Problem? Writing portable software!
- Advantage to manufacturers? Customers got locked in to their particular computers.
- In the late 1980s, the IEEE produced the standard that now virtually all follow.
- Kahan spearheaded the effort, and won the 1989 Turing Award for it.

IEEE-754 Floating Point

- Like a binary version of scientific notation
- Single-precision float uses 32 bits as follows:
  - **1 bit for the sign**: 1 for negative and 0 for positive
  - **8 bits for the exponent e** - To allow for negative exponents,127 is added to the exponent. We say that the exponent is biased by 127. So the range of possible exponents is not 0 to 28-1 = 0 to 255, but (0-127) to (255-127) = -127 to 128.
  - **23 bits for the significand or mantissa**



- A double is more precise as it uses 64 bits

Converting to IEEE-754 Floating Point

References:

- `http://www.oxfordmathcenter.com/drupal7/node/43`
- `https://www.youtube.com/watch?v=tx-M_rqhuUA`

Back to the example

- As we saw, 0.1 cannot be represented exactly in binary, leading to the unexpected result
- And adding a very small quantity to a very large quantity can mean the smaller quantity falls off the end of the mantissa
- But if we add small quantities to each other, this doesn't happen. And if they accumulate into a larger quantity, they may not be lost when we finally add the big quantity in.

Examples

- This seems contrived, but consider some value that accumulates in a loop.
- Code: Totalling.java
- Or consider adding up a list of doubles, what should you do?
- Code: ArrayTotal.java

Lessons

- When adding floating point numbers, add the smallest first.
- More generally, try to avoid adding dissimilar quantities.
- Specific scenario: When adding a list of floating point numbers, sort them first.

Example

- Suppose you want to have a loop that deals with numbers 0.1 to 1.0
- Code: LoopCounter.java

Lessons

- Dont use floating point variables to control what is essentially a counted loop.
- Also, use fewer arithmetic operations where possible.

Example

- A very simple program that just prints the same variable using different formats.
- Code: Examine.java

Example

- We shouldnt be surprised by now to find out that 4/5 cant be represented exactly in a float. Lots of things cant.
- But the represented value should be off by a tiny bit.
- What are all these extra digits??
  `4/5 = 1.10011001 10011001 10011001*1001100...x 2^(-1)`
  *(this is the 23rd bit)
- This gets rounded to
  `1.10011001100110011001101 x 2^(-1)`
- When we print, it gets converted back to decimal, which is:
  `0.800000011920928955078125000000`
- Code: Examine.java

Lesson

Dont ask for more precision in your output than you are holding.

Why does this matter? Example: Patriot missile accident

- In 1991, an American missile failed to track and destroy an incoming missile.
- The system tracked time in tenths of seconds. The error in approximating 0.1 with 24 bits was magnified in its calculations.
- At the time of the accident, the error corresponded to .34 seconds. A Patriot missile travels about half a km in that time.

Source: http://www.ima.umn.edu/~arnold/disasters/patriot.html

Summary of lessons

- Use double instead of float.
- When adding floating point numbers, add the smallest first.
- More generally, try to avoid adding dissimilar quantities.
- Specific scenario: When adding a list of floating point numbers, sort them first.
- Dont use floating point variables to control what is essentially a counted loop.
- Use fewer arithmetic operations where possible.
- Dont ask for more precision in your output than you are holding.