Extended Introduction to Computer Science CS1001.py

Module B Floating Point Representation

Instructors: Elhanan Borenstein, Michal Kleinbort

Teaching Assistants: Noam Parzanchevsky,

Omri Porat, Moshe Shechner

School of Computer Science

Tel-Aviv University

Spring Semester

http://tau-cs1001-py.wikidot.com

* Slides based on a course designed by Prof. Benny Chor

Plan

- Real numbers: bounded precision representation
  - Fixed point vs floating point

Wonders of real numbers in Python

- This is a very disturbing phenomenon:

```
>>> 0.1+0.1 == 0.2
True
>>> 0.1+0.1+0.1 == 0.3
False
```

And indeed,

```
>>> 0.1+0.1
0.2
>>> 0.1+0.1+0.1
0.30000000000000004
```

We need some understanding of how real numbers are represented in the computer's memory. This understanding will be very partial, in our course.

Fixed Point Arithmetic

- A simple way to represent real numbers in the computer's memory would be a fixed point representation.
  (**think about writing the number Pi on a piece of paper**)

- Suppose we want to allow n decimal digits for such numbers
- We allocate a fixed number of digits to the left of the decimal point.
- Denote this number i ($0 < i < n - 1$):

$$d_0 d_1 d_2 \ldots d_{i-1} . d_i d_{i+1} \ldots d_{n-1}$$

- The value of i can be regarded as a scaling factor.

  For example, if $n = 3$ and $i = 1$, then 1.23 is represented as 123 (the scaling factor 1/100 is implicit).

Fixed Point Arithmetic

- However, there are some major disadvantages to this method, which is why it is hardly used today in modern systems.

- First, this method bounds numbers to a fixed order of magnitude and precision. For example, distances between galaxies and diameters of atomic nucleus cannot be both expressed with the same fixed scaling factor.

- This also means that frequent rounding due to arithmetical operations may cause loss of precision.

  For example, if the scaling factor is 1/100 and $n = 3$, multiplying 1.23 by 0.26 entails multiplying 123 by 26 to yield 3198, which is scaled to 0.3198 and rounded to 0.32.

Floating Point Arithmetic

Floating Point Arithmetic

- Today, most real number arithmetic is done using floating points representation. This method allows different orders of magnitude and precision within the same type.

- Over the years, a variety of floating point representations have been used in computers. However, since the 1990s, the most commonly encountered representation is that defined by the IEEE 754 Standard, some of which will be presented next.

- The basic idea:

  3,498,523 can be represented as 3.498523×10^{6}

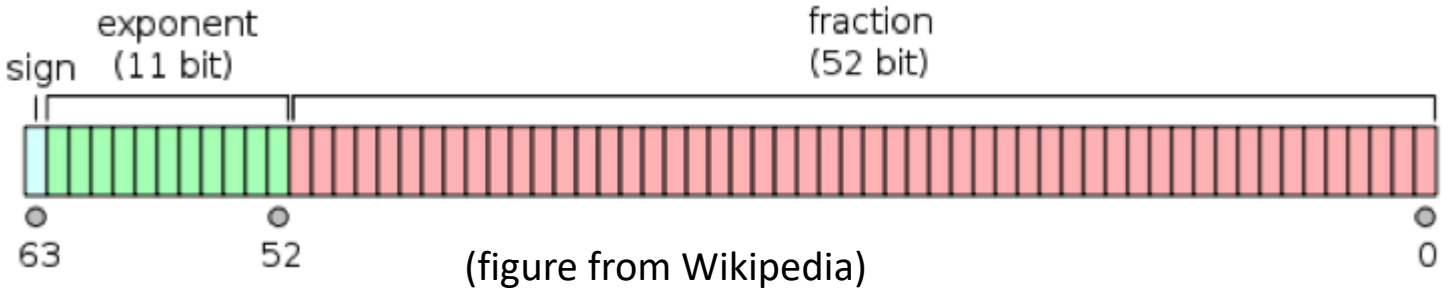
  -0.0432 can be represented as -4.32×10^{-2}

Representation of Floating Point Numbers

- Suppose we deal with a machine of 64 bit words. A floating point number is typically represented by

$$\text{sign} \cdot 2^{\text{exponent-1023}} \cdot (1+\text{fraction})$$

- The sign is ± 1 (1 indicates negative, 0 indicates non-negative).
- The exponent is an 11 bit integer, so $-1023 \leq \text{exponent-1023} \leq 1024$
- The fraction is a sum of negative powers of 2, represented by 52 bits, $0 \leq fraction \leq \sum_{i=1}^{52} 2^{-i} = 1 - 2^{-52}$



(figure from Wikipedia)

Example #1

Which number is represented here?

0 01111111110 1000000000000000000000000000000000000000000000000000

- The sign bit is 0 → sign = +1
- exponent = $01111111110_2 = 1022_{10}$
- The fraction bits are 100…0 →

$$\text{fraction} = \Sigma_{i=1}^{52} b_i \cdot \frac{1}{2^i} = 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} + \ldots + 0 \cdot \frac{1}{2^{52}} = \frac{1}{2}$$

$$\text{sign} \cdot 2^{\text{exponent}-1023} \cdot (1+\text{fraction}) =$$

$$(+1) \cdot 2^{1022-1023} \cdot \left(1 + \frac{1}{2}\right) = 2^{-1} \cdot 1.5 = 0.75$$

Example #2

Which number is represented here?

0 10000000001 0110000000000000000000000000000000000000000000000000

- The sign bit is 0 → sign = +1
- exponent = $10000000001_2 = 1025_{10}$
- The fraction bits are 01100...0 →

$$\text{fraction} = \Sigma_{i=1}^{52} b_i \cdot \frac{1}{2^i} = 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 1 \cdot \frac{1}{8} + 0 \cdot \frac{1}{16} + \cdots + 0 \cdot \frac{1}{2^{52}}$$
$$= \frac{1}{4} + \frac{1}{8} = 0.375$$

$$\text{sign} \cdot 2^{\text{exponent}-1023} \cdot (1+\text{fraction}) =$$
$$(+1) \cdot 2^{1025-1023} \cdot (1 + 0.375) = 2^2 \cdot 1.375 = 5.5$$

Floating Point Arithmetic

- Accuracy is of course still bounded (determined by the “physical word size'', the operating system you are using ,the version of the interpreter, etc., etc.).

- Indeed, floating point arithmetic carries many surprises for the unwary. This follows from the fact that floating numbers are represented as a number in binary, namely the sum of a fixed number of powers of two.

- The bad news is that even very simple rational numbers cannot be represented this way.
  - For example, the decimal 0.1=1/10 cannot be represented as a sum of powers of two, since the denominator has prime factors other than 2, in this case, 5.

Wonders of Floating Point Arithmetic in Python

- A confusing issue is that when we type 0.1 (or, equivalently, 1/10) to the interpreter, the reply is 0.1.

```
>>> 1/10
0.1
```

- This does not mean that 0.1 is represented exactly as a floating point number. It just means that Python's designers have built the display function to act this way. In fact the inner representation of 0.1 on most machines today is $3602879701896397 \cdot 2^{-55}$.

```
>>> 1/10 == 3602879701896397 * 2**-55
True
```

- And so is the inner representation of 0.10000000000000001 .
  Since the two have the same inner representation, no wonder that display treats them the same

```
>>> 0.10000000000000001
0.1
```

Arithmetic of Floating Point Numbers

- The speed of floating point operations, commonly measured in terms of FLOPS, is an important characteristic of a computer system, especially for applications that involve intensive numerical calculations.
- Addition is done by first representing both numbers with the same exponent, then adding the weighted fractions, then converting back so that the fraction is smaller than 1 (recall $0 \le \text{fraction} \le 1-2^{-52}$).
- Multiplication is done by multiplying the two fractions, and adding the two exponents.

Arithmetic of Floating Point Numbers

- The speed of floating point operations, commonly measured in terms of FLOPS, is an important characteristic of a computer system, especially for applications that involve intensive numerical calculations.
- Addition is done by first representing both numbers with the same exponent, then adding the weighted fractions, then converting back so that the fraction is smaller than 1 (recall $0 \leq \text{fraction} \leq 1\text{-}2^{-52}$).
- Multiplication is done by multiplying the two fractions, and adding the two exponents.
- Subtraction and division are analogous to addition and multiplication, correspondingly.
- Arithmetical operations may lead to substantial loss of precision:

```
>>> a = 1111111111111111111111111111111111111111111111.0
>>> b = 1111111111111111111111111111.0
>>> a-b
1.1111111111111112e+48
>>> a-b == a
True
```

Relaxed Equality for Floating Point Arithmetic

- We may attempt to solve some non intuitive issues with floating point numbers by redefining equality to mean equality up to some epsilon.

```
def float_eq(num1, num2, epsilon=10**(-10)):
    """re-defines equality of floating point numbers"""
    assert(isinstance(num1, float) and \
           isinstance(num2, float))

    return abs(num1-num2) < epsilon
```

- This indeed solves one problem

```
>>> float_eq(0.1+0.1+0.1, 0.3)
True
```

- But now this new equality relation is not transitive:

```
>>> float_eq(0.0, 2*11**(-10))
True
>>> float_eq(2*11**(-10), 4*11**(-10))
True
>>> float_eq(0.0, 4*11**(-10))
False
```

Floating Point Arithmetic vs. Rational Arithmetic

- So, no matter how we turn it around, floating point arithmetic introduces some challenges and problems we are not very used to in "everyday mathematical life".

- Questions:
  1. Can we do something about this?
  2. Should we bother?

- Answers:
  1. Yes, we can! (for example unbounded precision, rational arithmetic)
  2. Usually not: Typically the results of rational (unbounded precision) arithmetic for numerical computations are very similar to results from floating point arithmetic. Yet rational arithmetic is not for free -- huge numerators and denominators tend to form, slowing down computation significantly, for no good reason.

  Still, in some singular (and fairly rare) cases, numerical computations can be unstable , and the outcomes of floating point vs. rational arithmetic can be very different .

And Even More Wonders of Floating Point Arithmetic (for reference only)

- As we have pointed out, floating point precision is determined by the “physical word size'', the operating system you are using, the version of the interpreter, etc. etc.
- For example, the following results were obtained on a MacBook Pro, running MAC OSX version 10.6.8, Build 10K549, using Python version 3.2.2 and the IDLE interpreter, and also on a Lenovo X1, running Windows 7 Professional 64 bit, using Python version 3.3.0 and the IDLE interpreter,

```
>>> 2**-(1023+51)==0.0
False
>>> 2**-(1023+52)==0.0
True
```

- Can you explain these results?

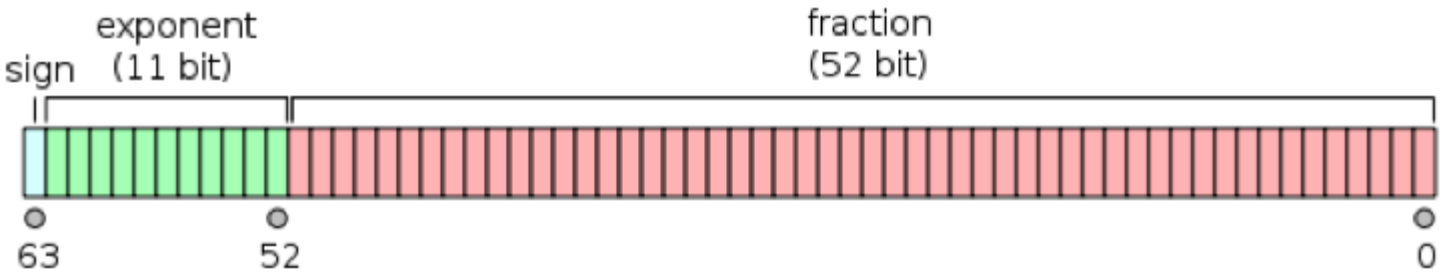
Representation of Floating Point Numbers (for reference only)

- The largest floating point number in a 64 bit word is smaller than 2^{1024} , and larger than 2^{1023}.
- We can try looking for this maximum:

```
>>> 2.0**1023
8.98846567431158e+307
>>> 2.0**1023*2.0
inf
>>> 2.0**1024
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    2.0**1024
OverflowError: (34, 'Result too large')
```

Exercise

- How many different floating point values are in $[0,1]$?



sign · $2^{\text{exponent-1023}}$ · (1+fraction)

  - Sign must be +
  - Recall $0 \leq fraction < 1$, and so $1 \leq 1 + fraction < 2$
    - 2^{52} different fractions possible
  - Any $exponent < 1023$ will yield a negative power of 2, thus the resulting floating point number is in $[0,1]$
    - There are 1022 such exponents
- Summing up we get $2^{52} \cdot 1022 < 2^{52} \cdot 2^{10} = 2^{62}$

Helpful Resources

- Two's complement:
  - https://www.youtube.com/watch?v=4qH4unVtJkE&ab_channel=BenEater
- Floating point numbers
  - https://www.youtube.com/watch?v=PZRI1IfStY0&ab_channel=Computerphile
- Zero and infinity

נציין שיש כמה מקרים מיוחדים שחורגים מהתבנית שהוסברה לעיל. לא דיברנו עליהם כלל בכיתה (ולא צריך להכיר אותם). לסקרנים:

- אם exponent מורכב כולו מ 0-ים ו - faction מורכב כולו 0-ים אז בהתאם ל sign נייצג את 0.0 ואת -0.0. המקרה המיוחד הזה חשוב על מנת לייצג את 0 כ float, משום שבעזרת הסכימה הרגילה לא ניתן לייצג את 0: תמיד מכפילים ב $(1 + fraction)$ שגדול ממש מ-0, והביטוי $2^{exponent-1023}$ הוא תמיד חיובי ממש.
- אם exponent מורכב כולו מ 1-ים ו- fraction מורכב כולו מ 0-ים אז בהתאם ל sign נייצג את "פלוס-אינסוף" ו:"מינוס-אינסוף". למען האמת, המספר המקסימלי שאינו "מקרה מיוחד" יכיל exponent שמתאים לערך $2^{11} - 2 = 2046$, כלומר יהיה גדול מ 2^{1023} וקטן מ 2^{1024}.

נשים לב שלכל מספר שאינו חורג מהתבנית שהוסברה לעיל, למעט 0, יש ייצוג יחיד .