Floating Point Numbers

CSC 28 – Discrete Structures for Computer Science

Last modified 29 January 2009

This course is called *Discrete Structures*. To understand why, consider the meaning of the two words. According to the American Heritage Dictionary, "discrete" means "defined for a finite or countable set of values", whereas "structure" has a meaning you might expect, "something made up of a number of parts that are held or put together in a particular way". The phrase "discrete structure" therefore might reasonably be interpreted as "something made of multiple parts, where each part comes from a finite set".

We see discrete structures everywhere in computer science. The fundamental data types such as `char`, `int` and `double` are each represented in a computer as a sequence of a certain number of bits (eg, a java `int` is 32 bits), and each bit comes from the set $\{0, 1\}$. So, by definition, each of these types is a discrete structure. Because all a computer is capable of doing is manipulating these fundamental types, one could argue that computers cannot do anything but manipulate discrete structures. But, that is too shallow of an examination. The pervasive nature of discrete structures continues at every level. When you define a class in java, you are creating a structure to hold related data, and each data member comes from some finite set of values. Every computer program is a structure of instructions, each coming from a finite set. Every circuit is a structure, etc, etc. Everything a computer does is by way of layer upon layer of discrete structure.

Since discrete structures are so important in computer science, we must become adept at describing and manipulating them. That begins with sets, and continues with various ways to structure the elements of a set: sequences, functions, relations, logic, circuits. We will also study some relevant reasoning techniques, in particular proof methods and counting principles.

Let's begin with an example discrete structure used extensively in computer applications: floating-point numbers. The most common way to work with a real number on a computer is to declare a variable of type `float` or `double` and assign the real number to the variable. The value stored in the variable, however, is not always exactly what was assigned. Let's see why and have a closer look at how discrete structures are used to get work done.

An Example: Floating Point Representation

When you declare a variable to be of type `float` or `double` in C, C++ or java, the compiler sets aside 32 or 64 bits (respectively) to hold the value. For the computer to write a number to those bits and later read a number back from them, the computer must follow a consistent interpretation. In 1985, the Institute of Electrical and Electronic Engineers (IEEE) published a standard for floating-point representation, IEEE-754. A `float` variable uses *single-precision* (32 bits) representation while a `double` uses *double-precision* (64 bits). We will look at the single-precision representation first.

Let's say we had 32 bits that we knew were a single-precision representation of a floating-point number,

$$b_1 b_2 b_3 \ldots b_{32} \, .$$

The structure of single-precision numbers is that there are three parts: the first bit, then the next eight bits and then the final 23 bits:

$$b_1 \quad | \quad b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 \quad | \quad b_{10} b_{11} \ldots b_{32} \, .$$

We then interpret each of the three parts, each in a different manner.

Sign bit. The bit b_1 represents the sign of the floating-point number. If $b_1 = 0$ then the number is not-negative. If $b_1 = 1$ then the number is negative. We will be plugging the sign bit into a formula later, so let's say that $s = b_1$ (ie, $s = 0$ or $s = 1$ depending on the value of b_1).

Exponent. The next eight bits are interpreted in "excess-127" notation. To do so, first interpret the eight bits as if they were a standard binary number, and then subtract 127. Let's call the resulting number e. Written as a formula, let $e = (b_2 \times 2^7 + b_3 \times 2^6 + \cdots + b_9 \times 2^0) - 127$.

Significand. The final 23 bits are interpreted as powers of $1/2$ (ie, fixed-point notation, much as digits to the right of a decimal point are interpreted with powers of $1/10$). Let $f = b_{10}/2^1 + b_{11}/2^2 + b_{12}/2^3 + \cdots + b_{32}/2^{23}$.

Once these three variables s, e, f are determined, the floating-point number represented by $b_1 b_2 b_3 \ldots b_{32}$ can easily be calculated as the result of the formula

$$(-1)^s \times (1 + f) \times 2^e \; .$$

Example 1. What real number does 11000001110100000000000000000000 represent when interpreted as single-precision floating-point? First, we split it up into its constituent parts:

$$1 \quad 10000011 \quad 10100000000000000000000 \; .$$

Next, we extract s, e, f. Because $b_1 = 1$, we know $s = 1$ and the final result will be negative. Because 10000011 is $128 + 2 + 1 = 131$, we know $e = 131 - 127 = 4$. Finally, we know $f = 1/2 + 1/8 = 5/8$ because only b_{23} and b_{25} are 1. Plugging these values into the equation gives us

$$\begin{aligned} (-1)^s \times (1 + f) \times 2^e &= (-1)^1 \times (1 + \frac{5}{8}) \times 2^4 \\ &= -\frac{13}{8} \times 16 \\ &= -26 \; . \end{aligned}$$

♢

The IEEE-754 standard specifies some special bit patterns:

$$\begin{array}{llll} 0 & 00000000 & 00000000000000000000000 & = \quad 0 \\ 0 & 11111111 & 00000000000000000000000 & = \quad +\infty \\ 1 & 11111111 & 00000000000000000000000 & = \quad -\infty, \end{array}$$

and a few more (eg, not-a-number and denormalized numbers), but in this class we will only look at numbers that use the usual interpretation method.

A problem trickier than interpreting a bit pattern is finding the single-precision representation of a given real number. Let's say x is a number and we want to write its single-precision representation. The general strategy is to rewrite x as a formula $(-1)^s \times (1 + f) \times 2^e$ and then extract the correct bit pattern from the values s, e, f. The only wrinkle in this strategy is that $1 + f$ must be no smaller than one and less than two because those are the only values representable by $1 + f$ (why?). Figure out s, e, f by following this algorithm (assuming x is not a value requiring one of the special patterns just mentioned):

1. Let $s = 0$ if x is positive, and let $s = 1$ if x is negative.
2. Write $|x| \times 2^0$. This is the magnitude of x.
3. If $|x| < 1$, then double x and decrease 2's exponent by 1. This maintains the value of the product, but gets $|x|$ closer to the range $[1 \ldots 2)$.
4. If $|x| \geq 2$, then halve x and increase 2's exponent by 1. This maintains the value of the product, but gets $|x|$ closer to the range $[1 \ldots 2)$.
5. Repeat Steps 3 or 4 until $1 \leq |x| < 2$, then figure out the values for e and f and write out the appropriate bit pattern for s, e, f.

It may be impossible to exactly represent the desired value because (i) the number is too big or too small to be represented using an 8-bit exponent, or (ii) f may fall between numbers representable using the bits $b_{10} \ldots b_{32}$. If this is the case, the number should be represented using the nearest number that is representable.

Example 2. What is the single-precision representation of $-\frac{7}{16}$? Because the number is negative, we know $s = 1$. Next, we write the magnitude of our number in the form $|x| \times 2^0$.

$$\frac{7}{16} \times 2^0$$

Since 7/16 is less than 1, we need to double it until it is at least 1, decreasing the exponent of 2 once for each doubling. (If the magnitude were two or more, then we would have to halve it until it was less than two, increasing the exponent of 2 each time.)

$$\begin{aligned} \frac{7}{16} \times 2^0 &= \frac{14}{16} \times 2^{-1} \\ &= \frac{28}{16} \times 2^{-2} \end{aligned}$$

This can be rewritten $(1 + \frac{3}{4}) \times 2^{-2}$, which tells me $e = -2$ and $f = 3/4$. Because e needs to be written in excess-127 notation, we determine its bit pattern by adding 127 to -2. Because f needs to be written as a fixed-point fraction, we need to find what powers of 1/2 sum to 3/4 (ie, 1/2 + 1/4 = 3/4). The final bit pattern is

1 01111101 11000000000000000000000.

♢

Double-precision works in much the same way, except that the exponent part is 11 bits long and gets written in excess-1023 notation, and the significand is 52 bits long.

Implications

There are some limitations to this representation that people should be aware of.

Limited precision. A pattern of 32 bits can only have 2^{32} different values. Since the real numbers are infinite, there is an infinity of real numbers that can't be represented precisely. If an unrepresentable number (such as 1/3) is stored as a floating-point number, it is truncated to a nearby value. This means that floating-point operations should always be treated as close approximations rather than precise results.

Absolute precision. For each value of e, there are 2^{23} values of f, and so 2^{23} representable values from 2^e to 2^{e+1}. This means that when e is at its smallest ($e = -127$), sequential floating-point numbers are 2^{-150} apart, but when e is at its largest ($e = 128$) numbers are 2^{105} apart. This makes small numbers appear more precise because we can never be too far off from the number we wish to represent, whereas large numbers may be way way off. This is true in an absolute sense, but untrue in a relative sense. Worst-case *percentage* error is constant throughout all floating-point numbers.

Although we will do nothing more with floating-point numbers in this class, they do make a good example of what we mean by a discrete structure. This course will continue to examine and manipulate discrete structures, and by so doing you will become better and better at understanding how to examine and manipulate computers themselves.