Questions

for

Functional Data Structures — Fall 2013

1 Computing the roots of quadratics

In high-school most of learn that the equation $ax^2 + bx + c = 0$ has two solutions given by

$$r_i = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{1}$$

In fact, if $b^2 < 4ac$ there are no real solutions, and if $b^2 = 4ac$ there is one (repeated) solution. Furthermore, if $|ac| \ll b^2$ then Equation (1) is a poor way to compute the smaller root.

It is also the case that if r_1 and r_2 are the roots of $ax^2 + bx + c = 0$ then $-r_1$ and $-r_2$ are the roots of $ax^2 - bx + c = 0$. It is also true that $ar_1r_2 = c$.

Putting all of these facts together we can come up with the following algorithm for computing the roots of a quadratic.

1. If $b > 0$, replace b with $-b$, solve, and then return the negative of the roots found.
2. If $a = 0$, we don't have a quadratic. Fail.
3. Otherwise [$a \neq 0$, $b < 0$] let $D = b^2 - 4ac$.
4. If $D < 0$, the quadratic has no real solution. Fail.
5. Set $r_1 = (\sqrt{D} + |b|)/(2a)$.
6. Set $r_2 = c/(r_1 a)$.
7. Return r_1, r_2.

\+ **Question 1.** Write this algorithm in Scheme and Standard ML.

Figure 1: UTF-16 encoding of Codes 0x10000 to 0x10$FFFF$

Suppose that the code point in binary is $v_1\,v_2v_3v_4v_5x_0x_1x_2x_3\,x_4x_5x_6x_7\,x_8x_9x_Ax_B\,x_Cx_Dx_Ex_F$. (where at least one of the vs must be non-zero.) Let $u = v - 1$ where v is the number whose binary representation is $v_1\,v_2v_3v_4v_5$. Then $0 \leq u < 16$ and can be written in binary as $u_1\,u_2u_3u_4$. The two 16-bit numbers representing this code point are

1	0	1	1	1	0	u_1	u_2	u_3	u_4	x_0	x_1	x_2	x_3	x_4	x_5
1	0	1	1	1	1	x_6	x_7	x_8	x_9	x_A	x_B	x_C	x_D	x_E	x_F

2 UTF-8 and Unicode

Unicode (or the Universal Character Set) contains information about most of the world's printed character. Each character has a number (often referred to as its *code point*) between 0 and 0x10$FFFF$, which allows for almost 17×2^{16} possible characters (some of the slots are permanently disallowed). The first 128 (0x00–0x3F) of these are identical to ASCII.

One method of storing Unicode character data is to allocate a 32-bit word to each character. However, this tends to be space inefficient.

Another method is to use 16-bit words, representing each valid character in the range 0x0–0x$FFFF$ as itself, and representing the characters with codes larger than 0x10000 as a pair of 16-bit words. This is how UTF-16 works, and is essentially what Java and parts of the Windows™ operating system do. See Figure 1 for details.

A third method is to represent each character as a sequence of 1 to 4 (8-bit) bytes, letting the ASCII characters stand for themselves, and using a special encoding for characters with code points greater than or equal to 128.

This is what UTF-8 does. Non-ASCII characters start with a sequence of 1s equal to the number of bytes used, followed by a 0, followed by bits of the actual code point. Subsequent bytes start with 10. (See Figures 2–5 for details.)

+ **Question 2.** Write a function to convert a list of integers (representing UCS code points) to a list of values in the range 0–255 representing the corresponding UTF-8 encoding.

0	x	x	x	x	x	x	x

Figure 2: UTF-8 encoding of ASCII

Encoding of $uuuuxxxxxx_2$ (At least one of the us must be non-zero.)

1	1	0	u	u	u	u	x	1	0	x	x	x	x	x	x

Figure 3: UTF-8 encoding of Codes 0x80 to 0x7FF

Encoding of $uuuuxxxxxx_2$ (At least one of the us must be non-zero.)

1	1	1	0	u	u	u	u
1	0	u	x	x	x	x	x
1	0	x	x	x	x	x	x

Figure 4: UTF-8 encoding of Codes 0x800 to 0x$FFFF$

Encoding of $uuuuxxxxxx_2$ (At least one of the us must be non-zero.)

1	1	1	1	0	u	u	u
1	0	u	u	x	x	x	x
1	0	x	x	x	x	x	x
1	0	x	x	x	x	x	x

Figure 5: UTF-8 encoding of Codes 0x10000 to 0x10FFF