Coding Review (I2P 2019)

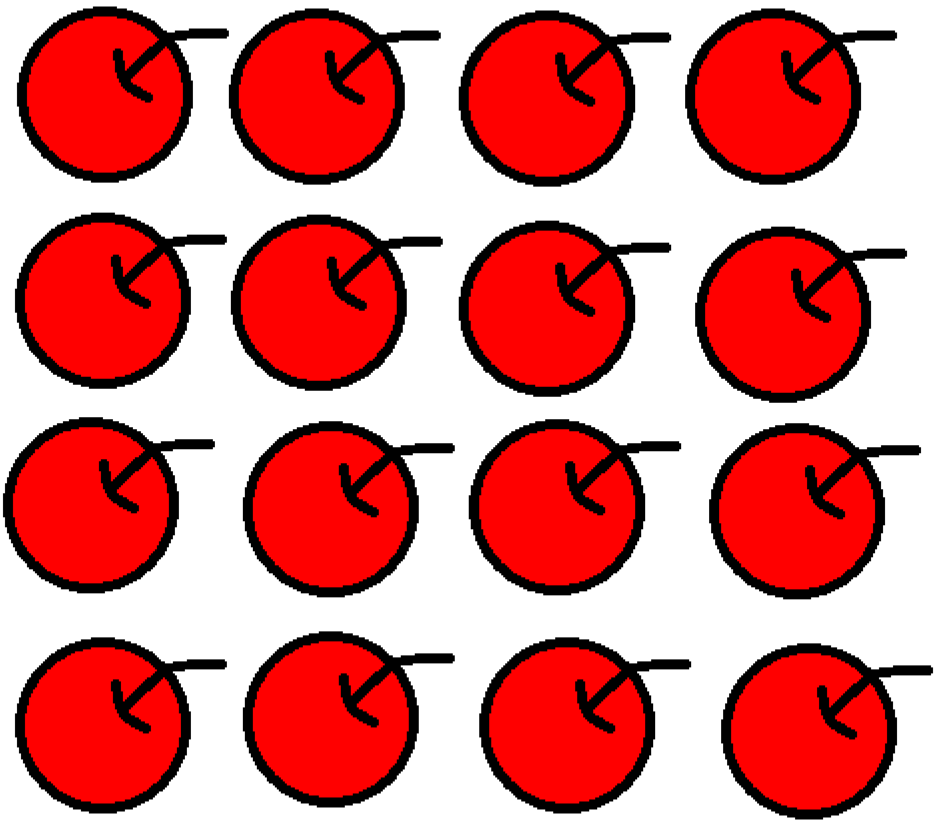
Binary Representations v1.0

Review of binary representations.

Number Basis (Radix)

Decimal, Binary, Octal, Hexadecimal are the common number basis seen in Computer Science courses.

- Decimal: All digits are 0 to 9, carry occurs at ten, so each digits have ten different values. (0, 1, ..., 9)
- Binary: All digits are 0 to 1, carry occurs at two, so each digits have two different values. (0, 1)
- Octal: All digits are 0 to 7, carry occurs at eight, so each digits have eight different values. (0, 1, ..., 7)
- Hexadecimal: All digits are 0 to 15 (F), carry occurs at sixteen, so each digits have sixteen different values. (0, 1, ..., 9, A, B, C, D, E, F)



1. So, if we have these apples, counting in Decimal should be:

   Answer: (Write your answer here!)

2. Counting in Binary should be:

Answer: (Write your answer here!)

3. Counting in Hexadecimal should be:

   Answer: (Write your answer here!)

Integer Conversion between Number Bases

Any Number Basis to Decimal

Since we're used to Decimal, so converting any other number basis to Decimal is an easy task for us. We can simply multiply each digits by the power of the number basis.

Take sixty one ($61_{10}, 111101_2, 3\mathrm{D}_{16}$) as example:

1. Decimal to decimal (61_{10})

   Answer: (Write your answer here!)

2. Binary to Decimal (111101_2)

   Answer: (Write your answer here!)

3. Hexadecimal to Decimal ($3\mathrm{D}_{16}$)

   Answer: (Write your answer here!)

Some more practices:

1. Binary to Decimal (1010111_2)

   Answer: (Write your answer here!)

2. Hexadecimal to Decimal ($6\mathrm{B}_{16}$)

   Answer: (Write your answer here!)

Decimal to Any Number Basis

Converting to any other number basis is also quite easy. We can simply keep dividing by the number basis and write down the remainder from right to left after each division. This is actually an inverse operation of converting to Decimal.

Take sixty one ($61_{10}, 111101_2, 3\mathrm{D}_{16}$) as example:

1. Decimal to Hexadecimal (61_{10})

   Answer: (Write your answer here!)

2. Decimal to Binary (61_{10})

   Answer: (Write your answer here!)

Some more practices:

1. Decimal to Binary (173_{10})

   Answer: (Write your answer here!)

2. Decimal to Hexadecimal (173_{10})

   Answer: (Write your answer here!)

Any Basis to Any Basis

You can convert use Decimal as an intermediate basis when converting between any basis. The direct conversion process is similar to the process above. You can try to come up with the direct conversion solution by yourself.

1. Binary to Hexadecimal (11111011_2)

   Answer: (Write your answer here!)

2. Hexadecimal to Binary (AC_{16})

   Answer: (Write your answer here!)

Binary and Hexadecimal are the number bases commonly used in programming. You can find the special relationship ($2^4 = 16$) between them, that is each four binary digits can be converted independently into one hexadecimal digit.

Fraction Conversion between Number Bases

1. Binary to Decimal (10.1101_2)

   Answer: (Write your answer here!)

2. Decimal to Binary (3.6875_{10})

   Answer: (Write your answer here!)

3. Decimal to Binary (0.87_{10})

   Answer: (Write your answer here!)

Online calculators can be found here:

- Binary to Decimal converter
- Decimal to Binary converter
- Exploring Binary

Binary Representations

Representation of Integers

Now we know how to represent positive integers in binary format, but how about negative integers?

We can simply add a bit to represent whether the number is negative. However, it's not straight forward to do additions and subtractions in the hardware level.

So we use 2's complement representation to make addition easier. The concept is simple, we want $N + (-N) = 0$, so if we have the binary representation of N, we can calculate $(-N)$ by inverting the digits and adding one.

1. What is the 2's complement binary representation of $(-6)_{10}$ if we have a total of four bits?

   Answer: (Write your answer here!)

4-bit 2's Complement Table:

Bit Pattern	**2's Complement**
0111	7
0110	6
...	...
0010	2
0001	1
0000	0
1111	-1
1110	-2
...	...
1001	-7
1000	-8

It's easy to see that the representable range for N-bit signed integer is $[-2^{N-1}, 2^{N-1} - 1]$. The fact that the positive range is smaller than the negative range can be easily seen by the left-most bit, since 0 is only represented once here.

Representation of Fractions

For representing fractions, we can simply use a signed-bit since the addition cannot be simplified as seen in 2's complement.

Floating point representation is used instead of Fixed point representation to increase the representable range.

We can further use Excess Notation to speed up comparison between numbers. The concept is to make the exponent term directly comparable through bit-patterns, that is, make the bit pattern of the smallest representable exponent as $0...0$, and keep adding up. This exponent is put at front of the mantissa so that we can compare two floating-points like integers. See the table below:

Excess-8 Notation Table:

Bit Pattern	Excess-8 Notation
1111	7
1110	6
...	...
1010	2
1001	1
1000	0
0111	-1
0110	-2
...	...
0001	-7
0000	-8

1. What is the floating-point representation of -2.625_{10}, with 4-bit fraction and 3-bit exponent using IEEE Standard.

   Answer: (Write your answer here!)

2. What is the truncation error of 0.1_{10} (direct truncation), with 4-bit fraction and 3-bit exponent using IEEE Standard.

   Answer: (Write your answer here!)

For further information, there's also a method for 2's complement multiplication, which is much harder to think intuitively. And there's also a 1's complement representation, which is quite straight forward. For floating-point representation there are Infinities, NaNs and Denormalized Numbers.

To wrap up, the common used representations are:

- Signed Magnitude
- One's Complement
- Two's Complement
- Excess-N Notation

Overflow & Underflow & Truncation

- In unsigned representation

  When two unsigned numbers are added, overflow occurs if there is a carry out of the leftmost bit.

- In signed 2's complement representation

Overflow occurs when both operands are positive and the result is negative. Or both operands are negative but the result is positive.

- In float representation

  Underflow occurs when the result of a floating point representation is smaller than the smallest value representable.

  When a number cannot be precisely represented, the number is truncated.

Try it by Yourself

Here's a C code that shows the binary representation of any variable.

```c
#include <stdio.h>
#include <limits.h>

void printBits_LittleEndian(size_t const size, void const* const ptr) {
    unsigned char* b = (unsigned char*) ptr;
    unsigned char byte;
    int i, j;

    for (i = size - 1; i >= 0; i--) {
        for (j = 7; j >= 0; j--) {
            byte = (b[i] >> j) & 1;
            printf("%u", byte);
        }
    }
    puts("");
}

void printBits_BigEndian(size_t const size, void const* const ptr) {
    unsigned char* b = (unsigned char*) ptr;
    unsigned char byte;
    int i, j;

    for (i = size - 1; i >= 0; i--) {
        for (j = 0; j < 8; j++) {
            byte = (b[i] >> j) & 1;
            printf("%u", byte);
        }
    }
    puts("");
}

int main(void) {
    int i = 23;
    unsigned int ui = UINT_MAX;
    float f = 23.45f;
    puts("For Little Endian:");
    printBits_LittleEndian(sizeof i, &i);
    printBits_LittleEndian(sizeof ui, &ui);
```

```
    printBits_LittleEndian(sizeof f, &f);
    // puts("For Big Endian:");
    // printBits_BigEndian(sizeof i, &i);
    // printBits_BigEndian(sizeof ui, &ui);
    // printBits_BigEndian(sizeof f, &f);
    return 0;
}
```

The output should be:

```
For Little Endian:
00000000000000000000000000010111
11111111111111111111111111111111
01000001101110111001100110011010
```

If your output is different than above, try un-commenting the Big Endian code below and see the results. This part is just for fun and you can see the actual binary representation for any type of variables. You'll learn more about Endians in Computer Architecture course.

If you look closely enough, you'll find that IEEE-754 actually uses a Excess-127 (127 bias) notation instead of Excess-128. If you are interested in the rationale of this, see:

- Why does the IEEE 754 standard use a 127 bias?
- What is a “bias value” of floating-point numbers?

By thinking the process inside out and think of the intuition behind the representations, you never need to memorize these representations or calculations.

For the next assignment, we'll review the basic concept of arrays and pointers.

Epilogue

> Q: Why do programmers confuse Halloween and Christmas?
>
> A: Because Oct 31 equals Dec 25.

If there's any typo, please discuss on iLMS or email `j3soon@gapp.nthu.edu.tw`, I appreciate your help.