Risk and Pricing Solutions

Bits, Bytes and Numbers

THIS DOCUMENT COVERS

- [Bit Operators](#)

a

Bits

Bit Operators

```
<<                          a          11010101
                            a << 2     01010100
>> (Signed Integers)        a          11010101
                            a >> 2     11110101

                            a          01010101
                            a >> 2     00010101
~                           a          00001101
                            ~a         11110010
&                           a          00001101
                            b          11101011
                            a&b        00001001

|                           a          00001101
                            b          11101011
                            a|b        11101111
^                           a          00001101
                            b          11101011
                            a^b        11100110
```
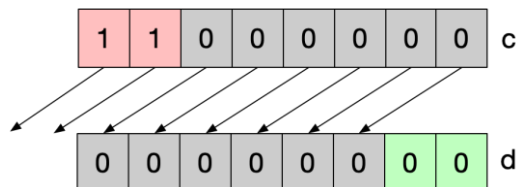
<< Left Shift

A left shift moves everything n place to the left. The left most n bits are dropped and the rightmost n bits are filled with zeros.

```
byte c = byte.MaxValue;
byte d = (byte)(a >> 2);
```
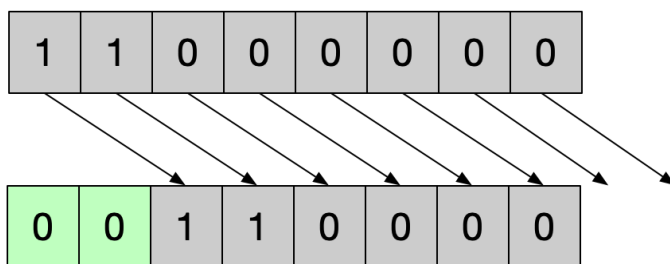
Figure 1 Left Shift

>> Right Shift

The right shift operator shifts all bits n places to the right. The rightmost n digits are dropped and the leftmost n digits are filled as follows. If the operand is unsigned the left n bits are filled with zeros

```
byte c = 128+64;
byte d = (byte)(c >> 2);
```


If the operand is signed the sign of the bits filled on the left matches the sign bit in the most significant position.

```
sbyte a = -96;
sbyte b = (sbyte)(a >> 2);
```
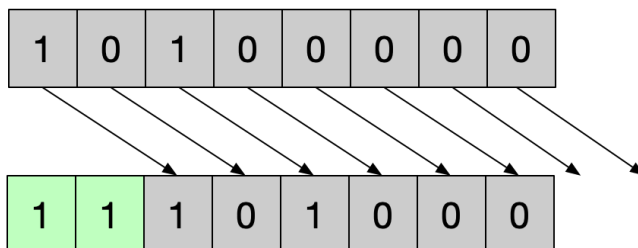

This extra complexity ensures that shifting right 1 place is equivalent to dividing by two when the operand is negative.

~ bitwise complement

Inverts all the bits

```
a      00001101
~a     11110010
```
& bitwise and

Copies a 1 into the result if the corresponding bits in each operand are 1

```
a      00001101
b      11101011
a&b    00001001
```
| bitwise or

```
a      00001101
b      11101011
a|b    11101111
```
^ exclusive or

```
a      00001101
b      11101011
a^b    11100110
```

~ bitwise complement

Bit Manipulation

`1 << i`	`1 << 3`	`0001000`	Create a mask with all zeros except a single 1 at bit location i
`~(1 << i)`	`~(1 << 3)`	`11110111`	Create a mask with all ones except a single 0 at bit location i
`~0 << n`	`~0 << 3`	`11111000`	Create a mask of all 1s except for 0s in the n least significant digits
`(1 << i)-1`	`(1 << 3)-1`	`00000111`	Create a mask of all 0s except for 1s in the n least significant digits
`(1 << j-i+1)-1)<<i`	`(1 << 4-2+1)-1)<<2`	`00011100`	Create a mask of all 0s except for digits i through j which contain 1s
`~(((1 << i - j + 1) - 1) << i))`	`~(((1 << i - j + 1) - 1) << i))`	`111000111`	Create a mask of all 1s except for digits i through j which contain 0s
`a+(~b+1)`	`5 + (~3+1)`	`2`	Perform subtraction without using the - key

Bit Properties

a ^ 0s = a

```
a            00001101
0s           00000000
a  ^  0s     00001101
```

a ^ 1s = ~a

```
a            00001101
1s           11111111
a  ^  1s     11110010
```

a ^ a = 0

```
a            00001101
0s           00000000
a  &  0s     00000000
```

a & 0s = 0

```
a            00001101
0s           00000000
a  &  0s     00000000
```

a & 1s = a

```
a            00001101
1s           11111111
a  &  1s     00001101
```

a & a = a

```
a            00001101
a            00001101
a  &  a      00001101
```

a | 1s = 1s

```
a            00001101
1s           11111111
a  |  1s     11111111
```

a | a = a

```
a            00001101
a            00001101
a  |  1s     00001101
```

a ^ ~a = 1s

```
a     00001101
~a    11110010
a^~a  11111111
```

Getting/Setting Bits

(1 << i) | a	(1 << 2) | 0b00101000 00101100		Set bit i to 1
~(1 << i) & a	~(1 << 2) & 0b00101100 00101000		Set bit i to 0
(a >> i) & 1	(0b00101100 >> 3) & 1 1		Get the value of bit i
(~0 << i) & a	(~0 << 3) & 0b10101111 10101000		Clear i least significant bits
((1 << i)-1) | a	((1 << 3)-1) | 0b10100000		Set i least signifcant bits
a & (a-1)	a 00111100 a-1 00111011 a&(a-1) 00111000		Clear the rightmost (least significant) 1 digit

Endianness

Most numeric types consist of multiple bytes. The order in which the bytes are arranged in memory is known as endianness. On a little-endian system, a numeric object's least to most significant bytes are arranged in order from lower memory addresses to higher memory addresses. Consider a .NET unsigned short which occupies 2 bytes or 16 bits

```
ushort a = 0x0102;
```

Figure 2 Endianness
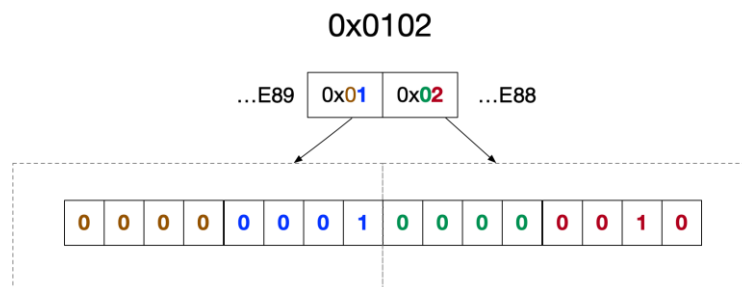
Manipulating Binary

TRICKS

Adding the same number

Performing integer addition where both operands are the same equal to multiplying by two which is equal to shifting left one place.

```
      00001101
+     00001101
      00011010
```

Multiplication

In binary multiplication is simply shiting the multiplicand left by a number of digits equal to the multiplier.

```
      00001101
*     00000011
      01101000
```

Bits – Questions

BIT MANIPULATION

…\linqpad\Queries\InterviewQuestions\Bits\2. Bit Manipulation

Implement this function to return a mask of all 0s except a single 1 in bit location i

The following shows how this works with i=3

```
idx          76543210
1            00000001
1 << 3       00001000
```

```csharp
public static sbyte MaskOne(int i) => (sbyte)(1 << i);
```

Implement this function to return a mask of all 1s except a single 0 in bit location i

The following shows how this works with i=3

```
idx          76543210
1            00000001
1 << 3       00001000
~(1 << 3)    11110111
```

```csharp
public static sbyte MaskTwo(int i) => (sbyte)(~(1 << i));
```

Implement this function to return a mask of all ones except for zeros in the i least significant bits from 0 to (i-1)

The following shows how this works with i=3

```
idx          76543210
~0           11111111
~0 << 3      11111000
```

```csharp
public static sbyte MaskThree(int n) => (sbyte)(~0 << n);
```

Implement this function to return a mask of all zeroes except for ones in the n least significant bits

The following shows how this works with i=3

```
idx          76543210
1 << 3       00001000
(1 << 3)-1   00000111
```

```csharp
public static sbyte MaskFour(int n) => (sbyte)((1 << n)-1);
```

Risk and Pricing Solutions

Implement this function to return a mask of all 0s except for digits i through j which

The following shows how this works with i=3, j=6

```
idx                       76543210
(1 << 6-3+1)              00010000
(1 << 6-3+1)-1            00001111
((1 << 6-3)-1)<< 3        01111000
```

```csharp
public static sbyte MaskFive(int i, int j) =>
   (sbyte)(((1 << j-i+1) - 1) << i);
```

Implement this function to return a mask of all 1s except for digits i through j which contain 0s.

The following shows how this works with i=3, j=6

```
idx                       76543210
(1 << 6-3+1)              00010000
(1 << 6-3+1)-1            00001111
((1 << 6-3)-1)<< 3        01111000
~(((1 << 6-3)-1)<< 3)     10000111
```

```csharp
public static sbyte MaskSix(int i, int j)
      => (sbyte)~(((1 << j - i+1) - 1) << i);
```

Risk and Pricing Solutions

BIT PROPERTIES

```
…\linqpad\Queries\InterviewQuestions\Bits\1. Properties of Bit Operators
```

What is the result of a ^ 0s?

```
a            00001101
0s           00000000
a ^ 0s       00001101
```
The result is a

What is the result of a ^ 1s?

```
a            00001101
1s           11111111
a ^ 1s       11110010
```
The result is ~a

What is the result of a^a?

```
a            00001101
0s           00000000
a & 0s       00000000
```
The result is 0s

What is the result of a&0s?

```
a            00001101
0s           00000000
a & 0s       00000000
```
The result is 0s

What is the result of a&1s?

```
a            00001101
1s           11111111
a & 1s       00001101
```
The result is a

What is the result of a | a?

```
a            00001101
a            00001101
a | 1s       00001101
```
The result is a

What is the result of a ^ ~a ?

```
a    00001101
~a   11110010
a^~a 11111111
```
The result is 1s

Perform bitwise negation without using the ~ operator?

Risk and Pricing Solutions

$a \wedge 1s = \sim 0$

Getting/Setting bits

`…\linqpad\Queries\InterviewQuestions\Bits\3. Getting and Setting`

Write a function that returns true or false, reflecting whether or not the bit at index i is 1 or 0 respectively

Consider the specific case where n = 5 and i=2

```
idx               76543210
n                 00000101
n >> 2            00000001
1                 00000001
(n >> 2)&1        00000001
((n >> 2)&1)>0    true
```

```csharp
public bool GetBit(int n, int i) => ((n >> i) & 1) > 0;
```
Write a function set the bit at index i to 1

```csharp
public int SetBit(int n, int i) => (1 << i) | n;
```
Write a function clear the bit at index i to 0

```csharp
public int ClearBit(int n, int i) => ~(1 << i ) & n;
```
Write a function that clears all bits from msb through to i inclusive

```csharp
public int ClearFromMsbToI(int n, int i)
{
  return ((1 << i )-1) & n;
}
```

Write a function that sets all bits from msb through to i inclusive

```csharp
public int SetFromMsbToI(int n, int i)
{
  return (~0 << i) | n;
}
```
Write a function that clears all bits from 0 through to i inclusive

```csharp
public int ClearFromLsbToI(int n, int i)
{
  return (~0 << i+1) & n;
}
```
Write a function that sets all bits from 0 through to i inclusive

```csharp
public int SetFromLsbToI(int n, int i)
{
  return ((1 << i+1)-1) | n;
}
```

BIT BASED INTERVIEW QUESTIONS

UNSET LEAST SIGNIFICANT SET BIT

Write an expression to unset the least significant/rightmost set bit

We make use of the fact that subtracting one from a binary number has the effect of unsetting the least significant set bit and setting all bits to the right of that bit.

10	00001010
10 − 1	00001001

If we then & the result of this operation with the original number the effect is to unset the least significant(rightmost) set bit

10	00001010
10 − 1	00001001
10 & (10 − 1)	00001000

SET ALL BITS TO RIGHT OF RIGHTMOST SET BIT

Write an expression to set all bits to the right of the least significant set bit

We make use of the fact that subtracting one from a binary number has the effect of unsetting the least significant set bit and setting all bits to the right of that bit.

10	00001010
10 − 1	00001001

If we then | the result of this operation with the original number the effect is to set all bits to the right of the least sinificant set bit to 1.

10	00001100
10 − 1	00001011
10 | (10 − 1)	00001111

Risk and Pricing Solutions

Write a function to calculate the number of 1s in an integers binary representation

We can use a simple linear traversal of the integers bits counting the 1s as we go. Such as algorithm is constant time and always takes O(sizeof(int)*8).

```
public int BitCount(int a)
{
    int numBits = sizeof(int) * 8;
    int bitCount = 0;

    for (int i=0; i < numBits;i++)
    {
        if (((a >> i) & 1) > 0)
            bitCount++;
    }

    return bitCount;
}
```

But actually we can do better. The following describes a clever algorithm invented by Brian Kernigan. It key idea is that if we subtract 1 from any integer then the result is that ever bit from the lsb up to a and including the least significant 1 is flipped. If we then perform an & operation we are effectively removing the least 1.

5	00000101
$5 - 1$	00000100
$5 \wedge (5 - 1)$	00000100
2^1	00011100
$2^1 - 1$	00000001
$2^1 \wedge (2^1 - 1)$	00000000
2^2	00000100
$2^2 - 1$	00000011
$2^2 \wedge (2^2 - 1)$	00000000

Risk and Pricing Solutions

COUNT NUMBER OF DIFFERING BITS

Given two integers a and b find the number of bits you would need to change to modify x to be y?

A simple XOR is enough to give us the bit that differ between a and b. We can then use Kernighan's algorithm to count the number of bits

```
public int CountDifference(byte a, byte b)
{
    byte diff = (byte)(a ^ b);
    int count = 0;

    while (diff != 0)
    {
        diff = (byte)(diff & (diff-1));
        count++;
    }
    return count;
}
```

SET ALL BITS TO RIGHT OF MOST SIGNIFICANT BIT

Write an expression to set all bits to the right of the most significant set bit

x 01000000
$x \mathrel{|}= x \gg 1$ 01100000
$x \mathrel{|}= x \gg 2$ 01111000
$x \mathrel{|}= x \gg 4$ 01111111

MSB > N

Write code to calculate if msb if is in location > n

```
idx                     76543210
x_0                     00010000
((1 << 5)-1)            00011111
~((1 << 5)-1)           11100000
x & ~((1 << 5)-1)  |= 0   true
```

Risk and Pricing Solutions

NEXT INT SAME 1 COUNT

Given an integer find the next largest integer that has the same number of 1s in its binary representation.

```csharp
public byte NextLargestSame1Count(byte x)
{
    int onesCount = 0;

    for (int i = 0; i < sizeof(byte) * 8; i++)
    {
        // We have found the first non-trailing zero
        if (((x >> i) & 1) == 0)
        {
            if (onesCount > 0)
            {
                // Flip first non-trailing zero to 1
                x |= (byte)(1 << i);

                // Zero locations right of flipped digit
                x &= (byte)(~1 << (i-1));

                // add back in onesCount-1 1s in lsb locations
                x |= (byte)((1 << onesCount-1)-1);

                break;
            }
        }
        else
        {
            onesCount++;
        }
    }
    return x;
}
```

PREVIOUS INT SAME 1 COUNT

Given an integer find the next smallest integer that has the same number of 1s in its binary representation.

The key idea is that we need to swap a set bit with an unset bit. If the bit we unset is to the left (more significant) than the bit we set we have decreased the number. Consider the specific input of 62. The first step is to find the 1 that we will flip to a zero. In order to be valid the 1 bit must have a 0 bit to the right of it (less significant) We are hence looking for the first non-trailing 1. We walk from least significant bits to most significant counting zeroes on the way and stopping when we reach the first non trailing 1.

```
idx              76543210
x0               01001111
i                6
```

The index of the non-trailing 1 is $i = 6$, the number of zeroes is $iz = 2$ and the number of ones is $io + 1 - iz = 5$ In order to unset the first non-trailing 1 which is at index 6 we create a mask $mask1 = \sim(1 \ll i)$

```
idx              76543210
x0               01001111
mask1            10111111
x1= x0&mask1     00001111
```

Rather than look for a single bit to set to the right of i, we instead clear all bits to the right of i and insert $io - 1$ ones imediately to the right of. First we create a mask for the zeroing $mask2 = \sim 0 \ll i)$

```
idx              76543210
x1               00001111
mask2            11000000
x2= x1 & mask2   00000000
```

Now we put the $io - 1$ ones immediately to the right of i. Our mask is

$$mask3 = \Big((1 \ll (i - 1)) - 1\Big) \ll (i - io)$$

```
idx              76543210
x2               00000000
mask3            00111110
x3= x2 | mask3   00111110
```

The source code is then

```csharp
public byte NextSmallestSame1Count(byte x)
  {
      Console.WriteLine($"x      {Convert.ToString(x, 2).PadLeft(8, '0')}");

      int zeroCount = 0;
```

```csharp
    for (int i = 0; i < sizeof(byte) * 8; i++)
    {
        if (((x >> i) & 1) != 0)
        {
            // If this condition is true the bit at the
            // current index is set and there exists
            // unset bits to the right of it.
            // We can do a switch
            if (zeroCount > 0)
            {
                // 1. Unset the bit at the current index.
                //    To do this we form a mask of all 1s
                //    except at index i where it has a zero.
                //    The mask is anded with x to unset bit i
                byte mask1 =(byte)~(1 << i);
                x &= mask1;

                // 2. The index of the unset bit is i. We want to
                //    clear all bits to the right of i. That is
                //    we want to clear bits 0 through i-1 or
                //    the leftmost i bits. We define a mask that
                //    consists of i 0s in positions 0 through i-1
                //    and the rest 1s. We and the mask with x to clear
                byte mask2 =(byte)(~0 << i);
                x &= mask2;

                // 4. We originally had (i+1-zeroCount) 1 digits.
                //    We need to these  back in location i-1
                //    i-1-(i+1-zeroCount)
                int oneCount = i +1 - zeroCount;
                byte mask3 = (byte)((1 << oneCount) -1);

                // 5. Shift the mask into position
                mask3 = (byte)(mask3 << (i-oneCount));
                // 6. Apply the mask
                x |= mask2;
                break;
            }
        }
        else
        {
            zeroCount++;
        }
    }
    return x;
}
```

Risk and Pricing Solutions

MISSING INT IN ARRAY

An array holds all the values from 0 to n inclusive with the exception of one number in the range which is missing. Write code to find which one

The simplest solutions makes use of the fact that we know that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ We can then just walk the array summing as we go and then subtract the result from the know value. This technique is $O(n)$

```
public int FindMissing(int[] a)
{
    int n = a.Length;
    int expected = (n*(n+1))/2 ;
    int actual = a.Aggregate ((x, y) => x+y);

    return expected - actual;
}
```

If the array is sorted we can use the relationship between each array element and its index to perform a binary search from O(logn)

	lo	mid	hi		0	1	2	3	4	5	6	7
					0	1	2	3	4	5	6	7
0	0	3	7		0	1	2	3	4	5	6	8
1	3	5	7		0	1	2	3	4	5	6	8
2	5	6	7		0	1	2	3	4	5	6	8
3	6	6	7		0	1	2	3	4	5	6	8

```
public int SearchIterative(int[] ar)
{
    int lo = 0;
    int hi = ar.Length-1;
    int mi =0;

    // Special cases for off the front and back off the
    // sequence
    if (ar[lo] != 0) return 0;
    if (ar[hi] == hi) return hi +1;

    while ((hi-lo)>1)
    {
        mi = (lo+hi)/2;

        if ((ar[mi]-mi) != 0)
            hi = mi;
        else
            lo = mi;
    }

    return lo+1;
}
```
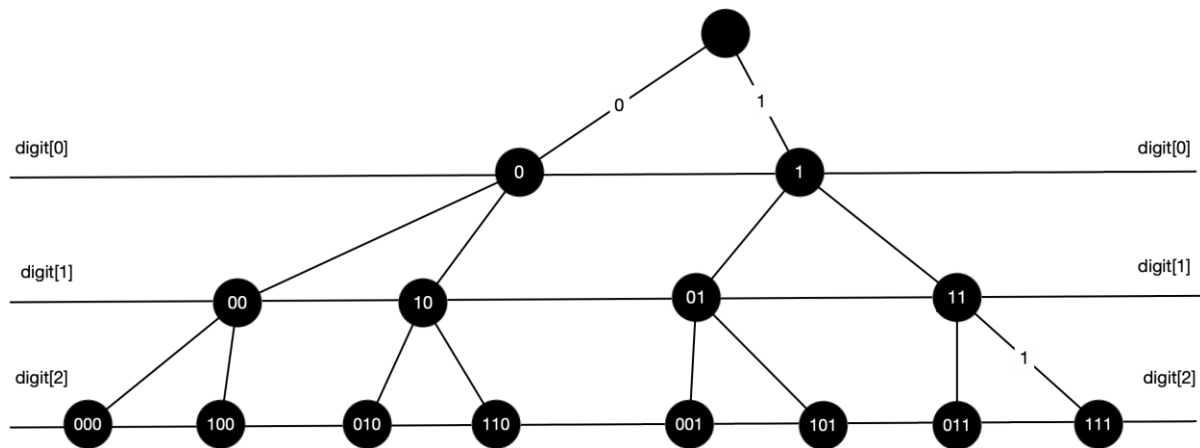
Finally if the array is not sorted we can use the relationship between the number of 1s and 0s at each bit position to work out which number is missing. This is by far the most complex technique but it does yield O(logn) on an unsorted array.


```csharp
public int SearchIterative(int[] ar)
{
    int numBits = (int)Math.Ceiling(Math.Log(ar.Count() + 2) /
Math.Log(2));
    int result = 0;

    List<int> searchArray = new List<int>(ar);
    for (int i = 0; i < numBits; i++)
    {
        List<int> ones = new List<int>();
        List<int> zeroes = new List<int>();

        for (int j = 0; j < searchArray.Count; j++)
        {
            if ((searchArray[j] >> i & 1) == 0)
                zeroes.Add(searchArray[j]);
            else
                ones.Add(searchArray[j]);
        }

        if (ones.Count >= zeroes.Count)
        {
            searchArray = zeroes;
        }
        else
        {
            searchArray = ones;
            result |= 1 << i;
        }

    }

    return result;
}
```

MAXIMUM XOR

Given an array of integers called ints and and an array of elements called elements that returns the maxium xor value of each value in elements against the elements in integers.

This is a tricky one and we need to use a data structure called a trie to solve it. Consider the case where our

SWAP EVEN AND ODD BITS

Write code to swap the even and odd bits of a given integer

The first stage is to separate out the even and odd digits. We use masks . Consider the specific example

```
idx              76543210
x                10111101
mask odd (0xaa)  10101010
mask even (0x55) 01010101
```
We apply the masks

```
idx                        76543210
x                          10111101
mask_odd                   10101010
x_odd = x & mask_odd       10101000

idx                        76543210
x                          10111101
mask_even                  01010101
x_even = x & mask_even     00010101
```
We shift the odd bits into even positions and even bits into odd positions

```
idx                          76543210
x_even = x_even <<< 1        00101010
x_odd = x_odd >>> 1          01010100
result = x_even | x_odd      01111110
```
The code is then

```csharp
sbyte SwapEvenAndOdd(sbyte x)
{
        // 1. Define the masks
        sbyte oddMask = unchecked((sbyte)0xaa);
        sbyte evenMask = unchecked((sbyte)0b01010101);

        // 2. Separate out the even and odd bits
        sbyte xEven = (sbyte)(x & evenMask);
        sbyte xOdd = (sbyte)(x & oddMask);

        // 3. Move odd bits into even positions and
        //    even bits into odd bit. Notice the cast to int
        //    to compensate fro C# having only arithmetic shift
        //    operators.
        xEven = (sbyte)(xEven << 1);
        xOdd = (sbyte)(((byte)xOdd) >> 1);
        return (sbyte)(xEven | xOdd);
}
```

LONGEST SEQUENCE OF 1S

Given an integer find the longest sequence of 1s you can form if you are allowed to flip one zero to a 1.

COPY SUBSECTION

POSITION OF RIGHTMOST SET BIT

POSITION OF SINGLE SET BIT

SWAP VARIABLES

Positional Number Systems

A positional number system represents any real number $\Re$ as a polynomial in the base of the number system.

$$\pm(d_\infty\beta^\infty+\ldots+d_1\beta^1 + d_0\beta^0 + d_{-1}\beta^{-1} + d_{-2}\beta^{-2}+\ldots d_{-\alpha}\beta^{-\alpha}) = \pm\left(\sum_{k=-\infty}^{\infty} d_k\beta^k\right)$$

When writing polynomial representations of numbers, we use a radix point to separate the whole and fractional parts. We can then drop the powers of the base β as the exponent is implicit in the position of the digit. If a power has no value, we still need to mark it with a co-efficient of zero. Our form becomes.

$$\pm(d_\infty\ldots d_1 d_0. d_{-1}d_{-2}\ldots d_{-\infty})_\beta$$

The following are some examples

- $+34.15_{10} = (3 \times 10^1 + 4 \times 10^0 + 1 \times 10^{-1} + 5 \times 10^{-2})_{10}$
- $-11.01_2 = -(1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2})_2 = -3.25_{10}$

Integers

If we drop the fractional part of the representation and consider only positive values, we have what programming languages refer to as the 'unsigned integers

$$(d_{n-1} \ldots d_1 d_0)_2 = (d_{n-1} 2^{n-1} + \ldots d_1 2^1 + d_0 2^0) = \left(\sum_{k=0}^{n-1} d_k n^k \right)$$

> **NOTE:** $\mathbb{N}$
>
> Mathematicians usually assume the natural numbers $\mathbb{N}$ exclude zero. We use the standard computer science convention that the set $\mathbb{N}$ includes zero.

Such a representation can distinguish between 2^n different values which we can use to represent positive integers in the range $[0, 2^n - 1]$ To highlight the approach consider the specific case of $n = 4$

```
00    0000
01    0001
02    0010
03    0011
04    0100
05    0101
06    0110
07    0111
08    1000
09    1001
10    1010
11    1011
12    1100
13    1101
14    1110
15    1111
```

It is useful to visualize such a system as a circle

Figure 3 Unsigned Integers
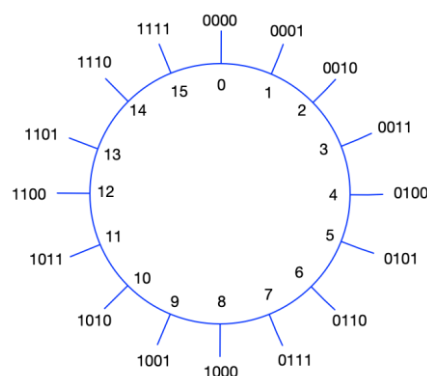
2s Complement Signed Integers

In the previous section we showed that an unsigned integer can be specified as the polynomial $w = (d_{n-1}2^{n-1} + \ldots d_1 2^1 + d_0 2^0) = (\sum_{k=0}^{n-1} d_k n^k)$ and that in this system the three-bit binary number 111_2 represents the decimal value 7_{10}.

If we allow both positive and negative values we obtain the set of integers $\mathbb{Z}$. In programming languages these are known as the signed integers.

$$\pm(d_{n-1}\ldots d_1 d_0)_2 = \pm(d_{n-1}2^{n-1} + \ldots d_1 2^1 + d_0 2^0) = \pm\left(\sum_{k=0}^{n-1} d_k n^k\right)$$

In order to provide support for signed integers most systems use a 2's complement notation. Twos complement is a way of encoding negative numbers into ordinary binary such that

addition still works. In a 2's complement signed representation we change our most significant digits weighting to $-1 \times d_n 2^{n-1}$ giving us

$$w = -1 \times d_n 2^{n-1} + \sum_{k=0}^{n-2} d_k 2^k$$

A n bit 2s complement representation supports the values from $-2^{n-1} \ldots (2^{n-1} - 1)$ In this system the three-bit binary number 111_2 is interpreted as the decimal value $-4 + 2 + 1 = -1_{10}$ If all the coefficients are set to 1, i.e. $(1 \ldots 1_1)$ the value is interpreted as -1. The following shows the values of a 4-bit 2's complement integer representation.

```
+00    0000    (−1 × 0 × 2³) + (0 × 2²) + (0 × 2¹) + (0 × 2⁰)
+01    0001    (−1 × 0 × 2³) + (0 × 2²) + (0 × 2¹) + (1 × 2⁰)
+02    0010
+03    0011
+04    0100
+05    0101
+06    0110
+07    0111    (−1 × 0 × 2³) + (1 × 2²) + (1 × 2¹) + (1 × 2⁰)
−08    1000    (−1 × 1 × 2³) + (0 × 2²) + (0 × 2¹) + (0 × 2⁰)
−07    1001
−06    1010
−05    1011
−04    1100
−03    1101
−02    1110
−01    1111    (−1 × 1 × 2³) + (1 × 2²) + (1 × 2¹) + (1 × 2⁰)
```

In the previous section on unsigned integers we saw that the maximum value than can be represented with a n-bit unsigned integer is $(2^n - 1) = 11 \ldots 1$. We also proved that subtracting a value from $(2^n - 1)$ is every bit is the same as flipping the bits

$$(2^n - 1) - b = \sim b$$

In our twos complement notation the binary value with a one in every bit is no longer $(2^n - 1)$ but instead is -1. Our equation then becomes

$$-1 - b = \sim b \therefore -b = \sim b + 1$$

In order to negate a positive 2's complement number we flip its bits and add one. We complement it and add one $-b = \sim b + 1$ When adding a pair of twos complement numbers where one of them is negative we simply move around the wheel the number of places in the positive direction of the twos complement binary representation.

Figure 4 Addition of negative unsigned integers

```
short Negate( short x )
{
    short neg = binaryAdd(~x, 1);
    return neg;
}

short binarySubtract( short x, short y )
{
    short minusY = Negate(y);

    return binaryAdd( x, minusY);
}
```

WHY TWOS COMPLEMENT IS POWERFUL

The most powerful aspect of 2's complement notation is that we can add positive and negative numbers. If we have n bits, we can represent 2^n values. If we move 2^n points around our modular system, we get back to the same number (overflow).

The algorithm to multiply a 2's complement number by -1 is to flip all its bits using the logical negation operator ~ and then add one. This is beautiful because we can use the same bitwise addition to perform addition and subtraction. To do subtraction just form the ones complement and then do normal addition

SUMMARY

- ◆ The most significant bit represents the sign
- ◆ Negating a value requires switching all its bits and then adding one
- ◆ 1 is represented by 001 and -1 is represented by 111
- ◆ N-bit implementation can represent numbers from -2^{n-1} to $2^{n-1} - 1$

Arithmetic Operators

UNSIGNED ADDITION

Addition $(a + b)$ is achieved by starting at a and moving b places clockwise around the wheel. Consider the specific case of $(3 + 2)_{10} = (0011 + 0010)_2$ We visualize this as follows
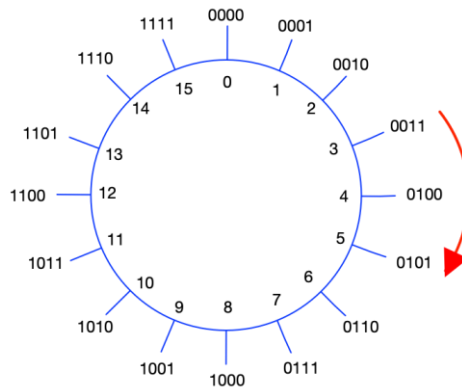
This is very simple binary addition

```
      0011
+     0010
      0101
```

The following C# code shows how we might achieve such addition

Figure 5 Adding Unsigned Integers

```csharp
public uint Add(uint a, uint b)
{
        uint carry = 0;
        uint result = 0;

        for (int bitIdx = 0; bitIdx < SizeInBits; bitIdx++)
        {
                // We deal in one binary digit at multiplicand time. By right
                // shifting multiplicand and bitIdx times we set the bit we want
                // into the least significant bit.
                uint aShifted = (a >> bitIdx);
                uint bShifted = (b >> bitIdx);

                // Now we make use of the fact that the number 1 has
                // in our unsigned representation consists of SizeInBits
                // zeros followed by multiplicand solitary one in the least significant
                // position. We can hence take our shifted valued and logically
                // and them with 1 to ensure we only have the digit values in the least
                // significant locations remaining.
                uint aDigit = aShifted & 1;
                uint bDigit = bShifted & 1;

                // We have three binary digits that feed into the current digit
                // {the multiplicand digit, the multiplier digit and the carry}
                //If one or all three
                // of these are one then the digit will be one, otherwise it will be
                // zero.
                uint sumBit = (aDigit ^ bDigit) ^ carry;

                // We now shift the bit into its correct location and add it into the
                // result
                result = result | (sumBit << bitIdx);

                // Finally calculate the carry for the next digit
                carry = (aDigit & bDigit) | (aDigit & carry) | (bDigit & carry);
        }

        return result;
}
```

Notice in our add method we do not deal with the overflow from the most significant bit. When we add one to the largest representable binary digit which consists of all ones the result is the smallest binary digit consisting of all zeros. In a four bit unsigned integer we would have as follows. Note the bold red overflow is discarded.

```
    1111
+   0001
   10000
```

By implementing add in this way we have created a modulo number system. If there are n bits in our unsigned integer, then addition is mod_{2^n}. For any unsigned integers a and m we have

$$(a + 2^n)_{mod_{2^n}} = a_{mod_{2^n}}$$

$$(a + m.2^n)_{mod_{2^n}} = a_{mod_{2^n}} \text{for}$$

Risk and Pricing Solutions

In our case adding $2^4 = 16$ to any value gets back to the same value. We show $4 + 2^4 = 4$
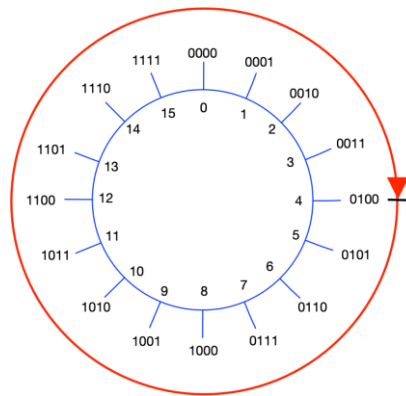
UNSIGNED SUBTRACTION

In our 4-bit integer notice what happens if we add $2^4 - 1 = 15$ to 4. We only rotate to 3. So, adding $2^4 - 1$ is the same as adding -1.
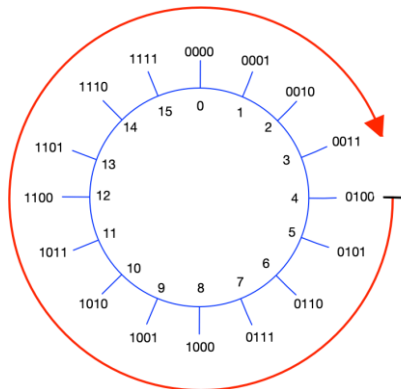
Similarly, adding $2^4 - 2$ is the same as subtracting 2 and adding $2^4 - b$ is the same as subtracting b. We noted in the previous section that $(a + 2^n)_{mod_2 n} = a_{mod_2 n}$ and so it is self-evident that

$$(a + [2^n - b])_{mod_2 n} = (a - b)_{mod_2 n}$$

This is a very useful result if we combine it with the following observation. Adding any binary number to its complement gives a number consisting solely of 1s.

$$b + \sim b = 11 \ldots 1$$

In our representation we have that $11 \ldots 1 = 2^n - 1$ hence it follows that

$$b + \sim b = 2^n - 1$$

Risk and Pricing Solutions

If we substitute this into the expression

$$(a + [2^n - b])_{mod_2 n} = (a - b)_{mod_2 n}$$

We get

$$(a - b)_{mod_2 n}(a + \sim b + 1)_{mod_2 n}$$

This means we can use our method for addition of unsigned integers to perform subtraction of unsigned integers. The following shows the simple C# code

```
public uint Subtract(uint a, uint b) => Add(a, Add(~b, 1));
```

b	B	~b	Adding	Subtraction
(Decimal)	(Binary)	(Binary)	(Clockwise)	(Anticlockwise)
0	0000	1111	15	$15 - 2^4 = -1$
1	0001	1110	14	$14 - 2^4 = -2$
2	0010	1101	13	$13 - 2^4 = -3$
3	0011	1100	12	$12 - 2^4 = -4$

Proof that $(a + \sim b + 1)_{mod_2 n} = (a - b)_{mod_2 n}$ ∴

From properties of modulo numbers we know that

$(a + 2^n)_{mod_2 n} = a_{mod_2 n}$ and hence

$(a - b + 2^n)_{mod_2 n} = (a - b)_{mod_2 n}$ rearranging

$(a + [2^n - b])_{mod_2 n} = (a - b)_{mod_2 n}$ adding and subtracting 1

$(a + [2^n - 1 - b] + 1)_{mod_2 n} = (a - b)_{mod_2 n}$

Risk and Pricing Solutions

DIVISION

Division is nothing but repeated subtraction. Integer division is defined using the following terms.

Division Example

$$18 = (16 \times 1) + 2$$

$$2 = 18 - (16 \times 1) = 18 \,\%\, 2$$

$$1 = 18 - \frac{(18 - 2)}{16} = 18 \,/\, 2$$

Euclid's Division Algorithm

The simplest algorithm to perform integer division is to repeatedly subtract. The runtime of this operation is very slow O(q) where q is the quotient

```
private (int q, int r) UnsignedDivide(int dividend, int divisor)
{
    int quotient = 0;
    int remainder = dividend;

    while (remainder >= divisor)
    {
        remainder -= divisor;
        quotient++;
    }

    return (quotient, remainder);
}
```

Signed divide is nothing more than a decorator of the unsigned divide method

Risk and Pricing Solutions

```
public (int q, int r) Divide(int dividend, int divisor)
{
    if (divisor == 0) throw new DivideByZeroException();

    if ( dividend < 0 && divisor < 0 )
         return UnsignedDivide(-dividend,-divisor);

    if (dividend < 0)
    {
         (int q, int r) =UnsignedDivide(-dividend, divisor);
         return (-q,r);
    }

    if (divisor < 0)
    {
         (int q, int r) = UnsignedDivide(dividend, -divisor);
         return (-q, r);
    }

    return UnsignedDivide(dividend,divisor);
}
```

Long Division Algorithm (Any Base)

Consider $\dfrac{163}{16}$
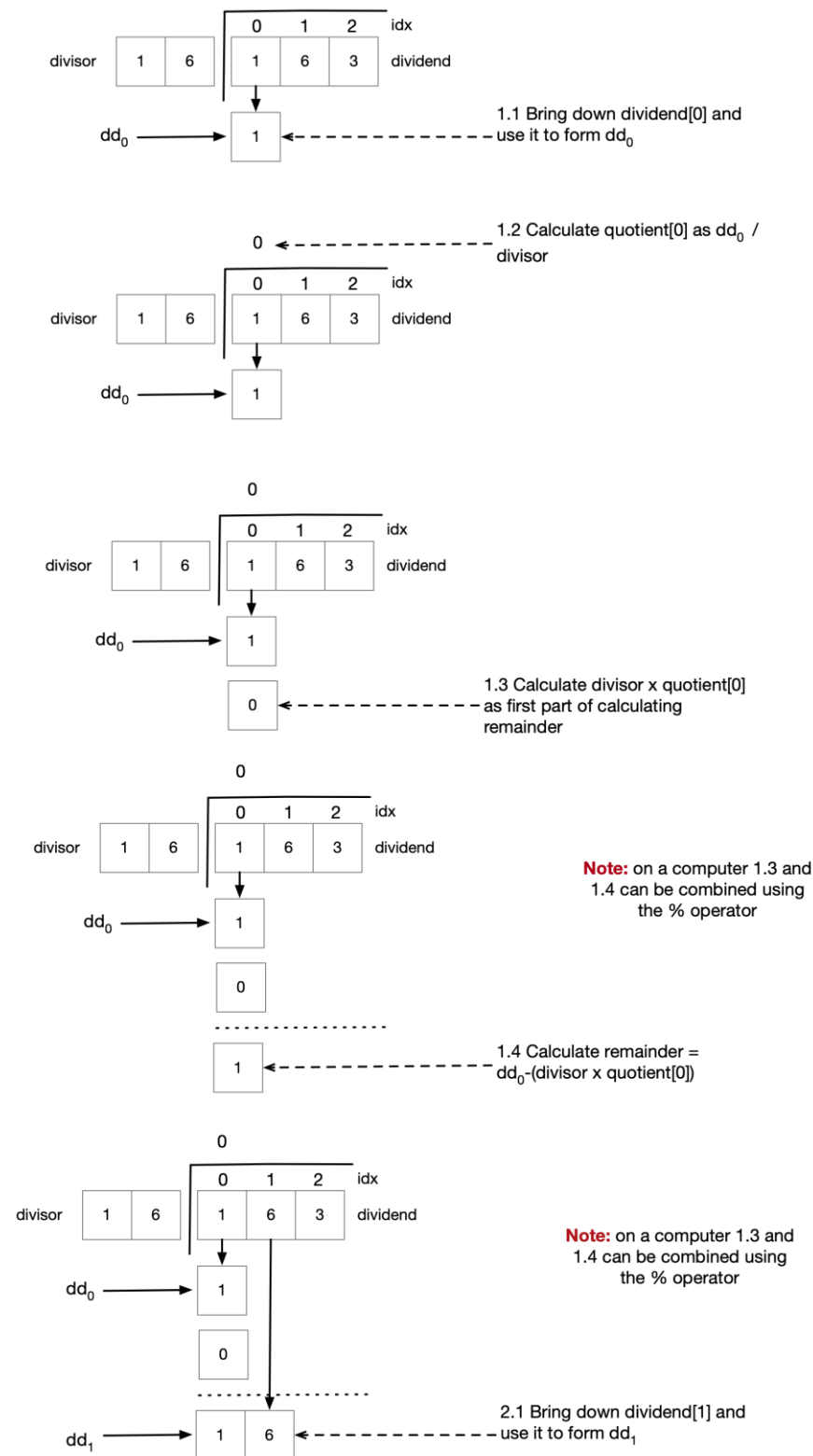
The following algorithm performs long division in any base.

```csharp
public (string quotient, string remainder) IntegerLongDivision(string dividend,
int divisor,
    int b = 10)
{
    StringBuilder quotient = new StringBuilder();
    int remainder = 0;
    int dd = 0;

    for (int idx = 0; idx < dividend.Length; idx++)
    {
        // idx.1 copy in next digit into temporary dividend dd
        dd = (dd * b) + dividend[idx].ToIntDigit();

        // idx.2 calculate partial quotient and set into quotient[idx]
        int partialQuotient = dd / divisor;
        quotient.Append(partialQuotient.ToChar());

        // idx.3 calculate this temporary as part of calculating the remainder
        int temp = partialQuotient * divisor;

        // idx.4 Calculate the remainder
        remainder = dd % divisor;

        // the remainder will form the basis of dd[idx+1]
        dd = remainder;
    }

    return (quotient.ToString(), remainder.ToChar().ToString());
}

public static class Extensions
{
    public static int ToIntDigit(this char c)
    {
        if (char.IsNumber(c)) return (int)char.GetNumericValue(c);

        return char.ToLower(c) - 'a' + 10;
    }

    public static char ToChar(this int i)
    {
        if (i >= 0 && i < 10)
            return (char)(i + '0');

        return (char)(i + 'a' - 10);
    }
}
```

Risk and Pricing Solutions

Long Division Algorithm Binary

If we want to do long division in binary the algorithm is very simple

```
public (int quotient, int remainder) UnsignedDivide(int dividend, int divisor)
{
        int numBits = sizeof(byte) * 8;
        int quotient = 0;

        int remainder = 0;

        for (int i = numBits-1; i >= 0; i--)
        {
                // Get the value of the dividend's bit index i
                byte d_i = (byte)((dividend >> i) & 1);

                // Shift the remainder left by 1 bit and add in the
                // bit i from the dividend
                remainder = ((remainder << 1) | d_i);

                // The value of the quotient at index i can only be 1 or 0.
                // It is 1 if the divisor is greater than or equal to
                // remainder, otherwise it is zero
                int q_i = (((remainder >= divisor) ? 1 : 0) << i);

                // copy q_i into the quotient
                quotient |= q_i;

                // If the quotient digit q_i is non zero we subtract the
                // divisor fro, the dividendTemp
                if ( q_i > 0 )
                        remainder -= divisor;
        }

        return (quotient,remainder);
}
```

Integer Long Division Algorithm Floating Point Result

```csharp
public string IntegerDivisionWithFloatingPointResult(string dividend, int divisor,
        int b = 10, int maxDigits = 8)
{
        StringBuilder quotient = new StringBuilder();
        int remainder = 1;
        int dd = 0;

        for (int idx = 0; (idx < dividend.Length || remainder > 0)
                    && idx < maxDigits; idx++)
        {
            // Add in a decimal point
            if (idx == dividend.Length)
                    quotient.Append(".");

            // idx.1 copy in next digit into temporary dividend dd
            if (idx < dividend.Length)
                    dd = (dd * b) + dividend[idx].ToIntDigit();
            else
                    // The integer dividend has no more digits so we just increase
                    // by a factor of b as we move to the right side of the point
                    // point
                    dd = (dd * b);

            // idx.2 calculate partial quotient and set into quotient[idx]
            int partialQuotient = dd / divisor;
            quotient.Append(partialQuotient.ToChar());

            // idx.3 calculate this temporary as part of calculating remainder
            int temp = partialQuotient * divisor;

            // idx.4 Calculate the remainder
            remainder = dd % divisor;

            // the remainder will form the basis of dd[idx+1]
            dd = remainder;
        }

        return quotient.ToString();
}
```

Integers – Questions

ADDITION

Write a function to add two signed integers. Do not use the + operator?

Note: Make sure you work through an example like this before writing the code

```
     1111
+    0001
    10000
```

```
public int Add(int a, int b)
{
    int numDigits = (sizeof(int) * 4)-1;

    int carry=0;
    int result = 0;

    for(int i=0; i < numDigits; i++)
    {
            int ai = (a >> i) & 1;
            int bi = (b >> i) & 1;

            // ri is a 1 if one or three of the
            // variables {ai,bi, carry} is a 1 otherwise
            // it is a zero. We use XOR
            int ri = ai ^ bi ^ carry;

            // the carry if any two of the three input are 1
            carry = (carry & ai) | (carry & bi) | (ai & bi);

            // Shift ri into position i and add to result
            result |= (ri << i);
    }

    return result;
}
```

Why does it work for negative integers?

Our addition is working on a modulo system where $a + 2^n = a$ and so

$a + (2^n - b) = a - b$ The following diagram shows how signed integers are represented in twos complement notation. Notice that $-b = 2^n - b$ so subtracting b is the same as adding $2^n - b$ if the binary is treated as an unsigned number.
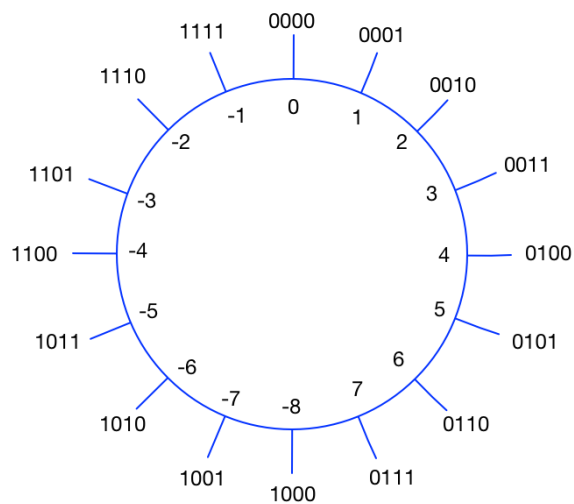
SUBTRACTION

User your function to write unsigned subtraction?

We make use of the fact that a − b = a + ~b +1

```
public int Subtract(int a, int b) => a + ~b +1;
```

Why does this work?

$$(a + [2^n - b])_{mod_2 n} = (a - b)_{mod_2 n}$$

$$b + {\sim}b = 11\ldots1 = 2^n - 1$$

$$(a - b)_{mod_2 n}(a + {\sim}b + 1)_{mod_2 n}$$

NEGATION

Write code to calculate a numbers twos complement
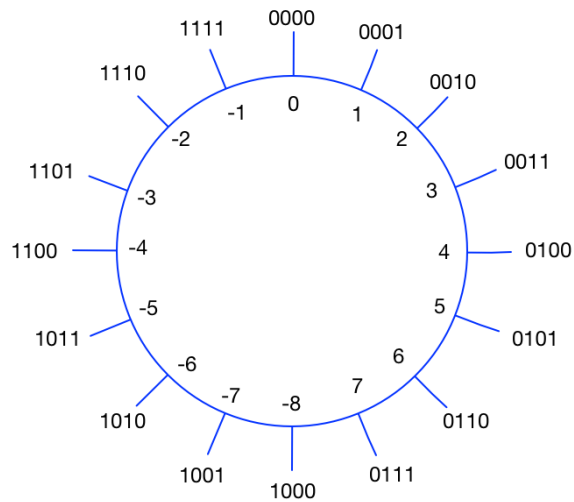
In binary as unsigned

$$b + 2^n = b$$

And we know

$$b + \sim b = 2^n - 1$$

Inserting 2 into 1

$$b + b + \sim b + 1 = b$$

Simplify

$$\sim b + 1 = -b$$

Risk and Pricing Solutions

MULTIPLICATION

Write code to do simple binary multiplication?

In binary multiplication is simply shifting the multiplicand left by a number of digits equal to the multiplier.

```
    00001101
*   00000011
    01101000
```

```
public int UnsignedMultiply(int multiplicant, int multiplier)
{
    int result =0;

    int numBits = sizeof(int)*8;

    for (int i = 0; i < numBits; i++)
    {
        if ( ((multiplier >> i ) & 1) > 0)
            result |= (multiplicant << i);
    }

    return result;
}
```

DIVISION BY REPEATED SUBTRACTION

Write code to do very simple division?

```
private (int q, int r) UnsignedDivide(int dividend, int divisor)
{
        int quotient = 0;
        int remainder = dividend;

        while (remainder >= divisor)
        {
                remainder -= divisor;
                quotient++;
        }

        return (quotient, remainder);
}
```

What is the performance of your algorithm?

This is very inefficient. O(q) where q is the quotient

It is very slow

Use your function to do signed division?

```
public (int q, int r) Divide(int dividend, int divisor)
{
        if (divisor == 0) throw new DivideByZeroException();

        if ( dividend < 0 && divisor < 0 )
                return UnsignedDivide(-dividend,-divisor);

        if (dividend < 0)
        {
                (int q, int r) =UnsignedDivide(-dividend, divisor);
                return (-q,r);
        }

        if (divisor < 0)
        {
                (int q, int r) = UnsignedDivide(dividend, -divisor);
                return (-q, r);
        }

        return UnsignedDivide(dividend,divisor);
}
```

DIVISION BY BINARY LONG DIVISION

Write code to perform binary long division unsigned

1.1 Bring down dividend[3] and use it to form partialDividend[3]

1.2 Calculate quotient[3] as partialDividend[3] /divisor

1.3 Calculate divisor x quotient[3] as first part of calculating remainder

Note: on a computer 1.3 and 1.4 can be combined using the % operator

1.4 Calculate remainder = partialDividend[3]-(divisor x quotient[3])

2.1 Bring down dividend[2] and use it to form partialDividend[2]

```
public (int quotient, int remainder) UnsignedDivide(int dividend, int
divisor)
{
    int numBits = sizeof(int) * 8;
    int quotient = 0;

    int remainder = 0;

    for (int i = numBits-1; i >= 0; i--)
    {
        // Get the value of the dividend's bit at index i
        int dividend_i = (int)((dividend >> i) & 1);

        // Form the partial dividend for this iteration
        // (partialDividend[i]) by combining the bit
        // at index i in the dividend (dividend[i])
        // the remainder from the previous iteration
        // shifted one bit left
        int partialDividend_i = (remainder << 1) | dividend_i;

        // The value of the quotient at index i (quotient[i])
        // can only be 1 or 0. It is 1 if the divisor is
        // greater than or equal to  partialDividend[i],
        //otherwise it is zero
        int quotient_i = ((partialDividend_i >= divisor) ? 1 : 0);

        // copy quotient[i] into the quotient
        quotient |= quotient_i <<i;

        // Calculate the product of quotient[i] and the divisor
        // as a part of calculating the remainder
        int productTemp = quotient_i * divisor;

        // The remainder from this iteration is then the
        // partialDividend[i] - (quotient_i * divisor) =
        //          partialDividend[i] % divisor
        remainder = partialDividend_i - productTemp;

        // Note the previous two statements can be much
        // simplified in the case of binary
        // which we do in Answer2
    }

    return (quotient,remainder);
}
```

We note however that the final two statements of the method that carry our the remainder can be greatly simplified because we are dealing with binary. The code becomes

```
public (int quotient, int remainder) UnsignedDivide(int dividend, int
divisor)
{
    int numBits = sizeof(int) * 8;
    int quotient = 0;

    int remainder = 0;

    for (int i = numBits-1; i >= 0; i--)
    {
            // Get the value of the dividend's bit at index i
            int dividend_i = (int)((dividend >> i) & 1);

            // Form the partial dividend for this iteration
            // (partialDividend[i]) by combining the bit
            // at index i in the dividend (dividend[i])
            // the remainder from the previous iteration
            // shifted one bit left
            int partialDividend_i = (remainder << 1) | dividend_i;

            // The value of the quotient at index i (quotient[i])
            // can only be 1 or 0. It is 1 if the divisor is
            // greater than or equal to  partialDividend[i],
            //otherwise it is zero
            int quotient_i = ((partialDividend_i >= divisor) ? 1 : 0);

            // copy quotient[i] into the quotient
            quotient |= quotient_i <<i;

            // Calculate the product of quotient[i] and the divisor
            // as a part of calculating the remainder
            int productTemp = quotient_i * divisor;

            remainder = partialDividend_i;

            // If the quotient digit q_i is non zero we subtract the
            // divisor fro, the dividendTemp
            if ( quotient_i > 0 )
                 remainder -= divisor;
    }

    return (quotient, remainder);
}
```

LONG DIVISION ANY BASE

Write code to perform integer division using a long division algorithm. The dividend is specified using a string. The base of the dividend and the divisor are given as simple ints?
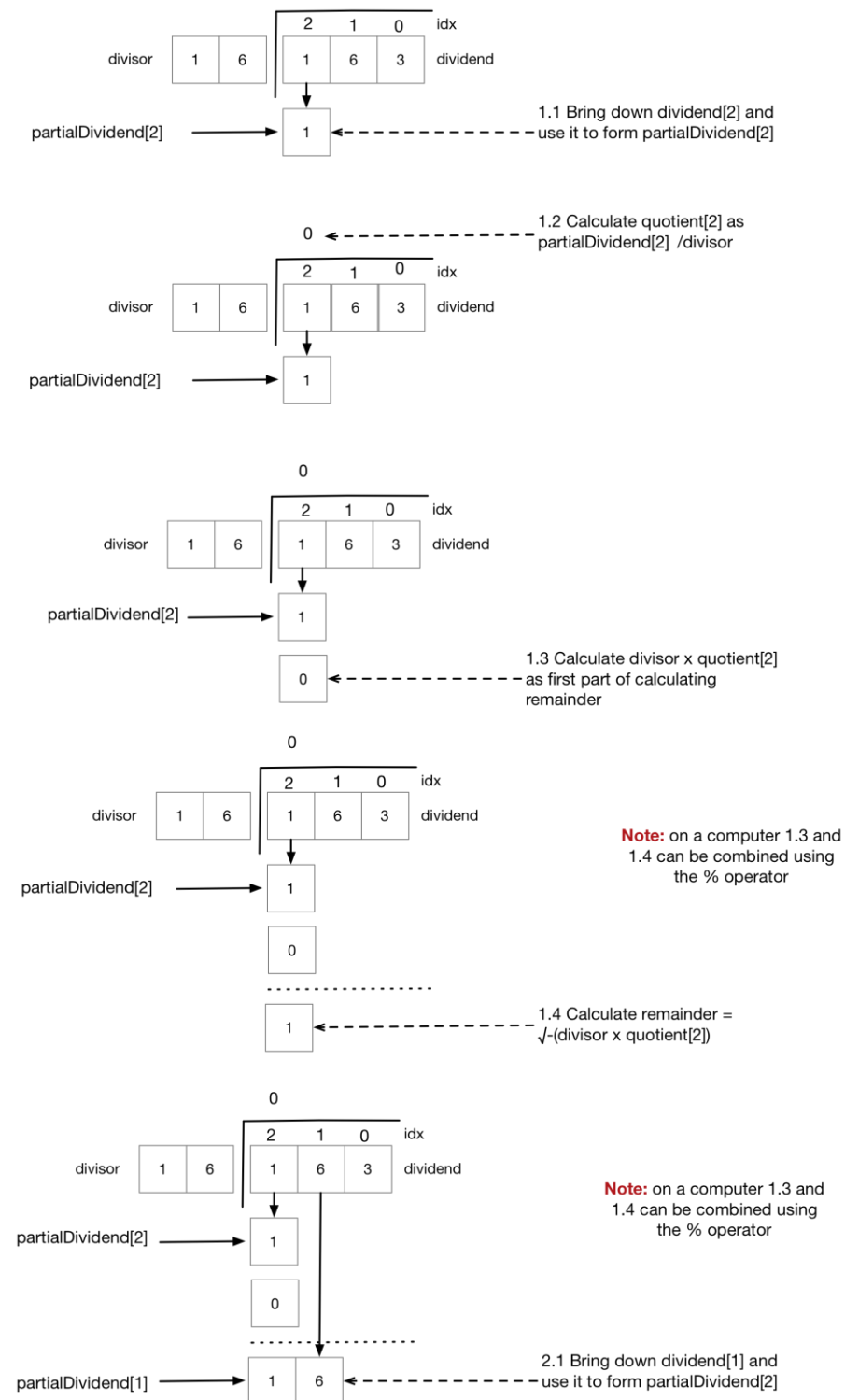
1.1 Bring down dividend[2] and use it to form partialDividend[2]

1.2 Calculate quotient[2] as partialDividend[2] /divisor

1.3 Calculate divisor x quotient[2] as first part of calculating remainder

Note: on a computer 1.3 and 1.4 can be combined using the % operator

1.4 Calculate remainder = √-(divisor x quotient[2])

Note: on a computer 1.3 and 1.4 can be combined using the % operator

2.1 Bring down dividend[1] and use it to form partialDividend[2]

```csharp
public (string quotient, string remainder) IntegerLongDivision(string
dividend, int divisor,
    int b = 10)
{
    StringBuilder quotient = new StringBuilder();
    int remainder = 0;
    int dd = 0;

    for (int idx = 0; idx < dividend.Length; idx++)
    {
        // Get the value of the character at index i and
        // convert it to an integer. This gives us a single
        // digit of the dividend
        int dividend_i = dividend[idx].ToIntDigit();

        // Form the partial dividend for this iteration my
        // shifting the remainder from the previous iteration
        // one position left and adding the dividend[i]
        int partialDividend_i = (remainder * b) + dividend_i;

        // Calculate partial quotient and set into quotient[idx]
        int quotient_i = partialDividend_i / divisor;
        quotient.Append(quotient_i.ToChar());

        // Calculate the remainder
        remainder = partialDividend_i % divisor;
    }

    return (quotient.ToString(), remainder.ToChar().ToString());
}

public static class Extensions
{
    public static int ToIntDigit(this char c)
    {
        if (char.IsNumber(c)) return (int)char.GetNumericValue(c);

        return char.ToLower(c) - 'a' + 10;
    }

    public static char ToChar(this int i)
    {
        if (i >= 0 && i < 10)
            return (char)(i + '0');

        return (char)(i + 'a' - 10);
    }
}
```

PARSE INT

Write code to parse an Integer

```csharp
public int ParseInteger(string s)
{
        int sign = 1;

        if (s[0] == '-')
        {
                s = s.Substring(1);
                sign = -1;
        }

        int x = 0;
        foreach (var c in s) x = (x*10) + c - '0';
        return x * sign;
}
```

CHANGE OF BASE

Given a string representation of an integer N in base convert it to a string representation of an integer in base β. For example given the input "10" with $\lambda = 10$ and $\beta = 2$ it would return "1010"

Let $N = \pm(a_n\lambda^\infty + \ldots + a_2\lambda^1 + a_1\lambda^0)_\lambda$ then we want to find the coefficients c_i such that

$$N = \pm(c_n\beta^\infty + \ldots + c_2\beta^1 + c_1\beta^0)_\beta$$

We have a number N

$$N = (a_n a_{n-1} \ldots a_2 a_1)_\lambda$$

That we want to convert to base β such that

$$N = (c_m c_{m-1} \ldots c_1 c_1)_\beta$$

We can rewrite this as

$$N = c_1 + \beta\big(c_2 + \beta(c_3 + \ldots + \beta(c_m))\ldots\big)_\beta$$

If we divide it by β then the remainder is clearly c_1 and the quotient is

$$c_2 + \beta\big(c_3 + \beta(c_4 + \ldots + \beta(c_m))\ldots\big)_\beta$$

If we repeat this until the quotient is zero we can read off the value of c_1 to c_n giving us the required number in the new base $(c_m c_{m-1} \ldots c_1 c_1)_\beta$

Integer change of base

```csharp
private string ConvertIntegralPart(string input, int l, int b)
{
    var result = new StringBuilder();

    // Calculate the decimal equivalent
    var idx =0;
    var d = input[idx].ToIntDigit();
    while(++idx < input.Length) d = (d * l) + input[idx].ToIntDigit();

    var quotient = d;
    do
    {
        var r = quotient % b ;
        quotient = quotient / b;
        result.Append(r.ToChar());
    }
    while (quotient > 0 );

    var chars = result.ToString().ToCharArray().Reverse().ToArray();
    return new string(chars);
}
```

LOG ANY BASE

Write a function Log(double x, double b) that takes a double value and a double base and returns $Log_b x$ It should work for any valid real base.

We only have natural logarithm and logarithm base 10 in the mathematics package but we can make use of the following to calculate any base from the natural logarithm or the base 10 logarithm

$$log_a x = \frac{log_b x}{log_b a}$$

Proof

Let $x = a^y$ and hence $log_a x = y$

$$log_b x = log_b a^y$$

$$log_b x = y \times log_b a$$

$$log_b x = log_a x \times log_b a$$

$$log_a x = \frac{log_b x}{log_b a}$$

The C# source code is then given by

Logarithm any base

```
public double Log(double x, double b)
{
    return Math.Log(x) / Math.Log(b);
}
```

DIGITS REQUIRED TO REPRESENT INTEGER IN BASE B

Given an integer value calculate the number of digits d required to represent that integer in base b nuber system

A number n represented in a base b number system will consist of k digits if and only if $b^{k-1} \leq n < b^k$. In other words b^{k-1} is the smallest number that requires k digits. Based on these facts we can derive expressions that calculate the number of digits k required to represent n in base b.

Expression using the floor function

Taking logarithms our inequality becomes.

$$k - 1 \leq log_b\, n < k.$$

From the properties of the floor function we know that $\lfloor x \rfloor = m \leftrightarrow m \leq x < m + 1$ and hence in our case

$$\lfloor log_b\, n \rfloor = k - 1$$

Expression using the ceiling function

We can achive a similar result that uses the ceiling function by adding one to the inequality $b^{k-1} \leq n < b^k$. so we get

$$b^{k-1} < n + 1 \leq b^k \text{ and taking logarithms we get}$$

$$k - 1 < log_b(n + 1) \leq k$$

From the properties of the ceiling function we know that $\lceil x \rceil = m \leftrightarrow m - 1 < x \leq m$ and hence that

$$\lceil log_b(n + 1) \rceil = k$$

Number of digits code

The following code uses the ceiling function approach. It requires a function that gives the logarithm of any base.

```
public double DigitsRequired(int x, int b)
    => Math.Ceiling(Log(x+1,b));

public double Log(double x, double b)
    => Math.Log(x) / Math.Log(b);
```

Risk and Pricing Solutions

LOG BASE 2 FLOOR (INTEGER)

Write a function to calculate $\lfloor \log_2 x \rfloor$

The brute force algorithm simply shifts right one digit at a time until we reach zero. The number of ties we can do this gives us the position of the most significant set bit and hence the number we are looking for

```
public byte IntLog(byte x)
{
    if (x<=0) throw new ArgumentException();
    byte shiftCount =0;

    while (x >0)
    {
        x >>=1;
        shiftCount++;
    }

    return (byte)(shiftCount -1);
}
```

Risk and Pricing Solutions

More efficient log2(n)

```
public int LogOpt(int x)
{
    int e = 0;

    if ((x &(~((1<<16)-1))) != 0)
    {
        // We have set digits in location 16-31 so we don't
        // care about the digits in locations 0-15. Add 16
        // and shift right to home in on exact location
        x >>= 16; e += 16;
    }

    if ((x &(~((1<<8)-1))) != 0)
    {
        // We have set digits in location 8-15 so we don't
        // care about the digits in locations 0-7. Add 8
        // and shift right to home in on exact location
        x >>= 8; e += 8;
    }

    if ((x &(~((1<<4)-1))) != 0)
    {
        // We have set digits in location 4-7 so we don't
        // care about the digits in locations 0-3. Add 4
        // and shift right to home in on exact location
        x >>= 4; e += 4;
    }

    if ((x &(~((1<<2)-1))) != 0)
    {
        // We have set digits in location 2-3 so we don't
        // care about the digits in locations 0-1. Add 2
        // and shift right to home in on exact location
        x >>= 2; e += 2;
    }

    if ((x &(~((1<<1)-1))) != 0)
    {
        // Finally is the digit in slot index 0 or 1
        e += 1;
    }

    return e;
}
```

MINIMUM OF TWO INTEGERS NO BRANCHING

Write code to find the minimum of two signed integers. You may not use Math.min or branching constructs.

Consider the case where we have two signed 8 bit integers a and b. If we take their difference (a-b) then the result can be classified as

- ♦ 0xxxxxxx   If a >= b or 1xxxxxxx   If a < b

If we perform a right arithmetic shift of 7 bits (sizeof the int -1) we get either

- ♦ 00000000   If a >= b
- ♦ 11111111   If a < b

Now if we & the result of this shift with the original difference. ((a-b) >> 7) & a-b

- ♦ 0   If a >= b
- ♦ a-b   If a < b

Now we add in b

- ♦ 0+b=b   If a >= b
- ♦ a-b+b =a   If a < b

So we have returned b if a>=b and a if a < b which was the original aim

```
public sbyte Min(sbyte a, sbyte b)
{
    // Take the differnce a-b. The result is one of two forms
    // a) 0xxxxxxx if a >= b
    // b) 1xxxxxxx if a < b
    sbyte difference = (sbyte)(a-b);

    // The result of the right shift is then one of two things
    // a) 00000000 if a >= b
    // b) 11111111 if a < b
    sbyte mask = (sbyte)(difference >> (sizeof(sbyte)*8-1));

    // Now if we & the mask and (a-b) we get one of of two things
    // a) 00000000 if a >= b
    // b) a-b       if a < b
    sbyte temp = (sbyte)(mask & difference);

    // If we add b to this temp variable we get one of two things which
    // is what we wanted
    // a) 0+b=b       if a >= b
    // b) a-b+b=a     if a < b
    return (sbyte)(temp + b);}
```

MAXIMUM OF TWO INTEGERS NO BRANCHING

Write code to find the maximum of two signed integers. You may not use Math.min or branching constructs.

Risk and Pricing Solutions

This is the same as the previous code except for we take the complement of the shift.

```
public sbyte Max(sbyte a, sbyte b)
{
    // Take the differnce a-b. The result is one of two forms
    // a) 0xxxxxxx if a >= b
    // b) 1xxxxxxx if a < b
    sbyte difference = (sbyte)(a-b);

    // The result of the complemented right shift is
    //  then one of two things
    // a) 11111111 if a >= b
    // b) 00000000 if a < b
    sbyte mask = (sbyte)~(difference >> (sizeof(sbyte)*8-1));

    // Now if we & the mask and (a-b) we get one of of two things
    // a) a-b      if a >= b
    // b) 0        if a < b
    sbyte temp = (sbyte)(mask & difference);

    // If we add b to this temp variable we get one of two things which
    // is what we wanted
    // a) a-b+b=a    if a >= b
    // b) 0+b=b      if a < b
    sbyte result = (sbyte)(temp + b);
    return result;
}
```

Risk and Pricing Solutions

INTEGER ABSOLUTE VALUE NO BRANCHING

Write code to find the absolute value of an integer without branching.

We first use our old shift right routine to form a mask. If x is positive the mask is 0s and if x is negative the mask is all 1s.

```
x=5                 00000101
mask = x>>7         00000000

x=-5                11111011
mask = x>>7         11111111
```

Now if we xor the mask with x we get one of two things. If the mask is 0s then the result is just x. If the mask is negative the result is ~x because xor with 1s is the same as the complement operator.

```
x=5                 00000101
mask = x>>7         00000000
mask ^ x            00000101

x=-5                11111011
mask = x>>7         11111111
mask ^ x            00000100
```

The final trick is to subtract the mask from the result of the xor. If the mask is zero then the subtraction has no effect and we return x. If the mask is 1s this represents -1 in 2s complement. In the negative case we have x ^ 1s -1 which is the same as positive x.

```
x=5                 00000101
mask = x>>7         00000000
mask ^ x            00000101
(mask ^ x)-mask     00000101

x=-5                11111011
mask = x>>7         11111111
mask ^ x            00000100
(mask ^ x)-mask     00000101
```

The code is then

```
public sbyte AbsoluteValue(sbyte x)
{
    sbyte mask = (sbyte)(x >> 7);
    return (sbyte)((mask ^ x ) - mask);
}
```

CALCULATE SIGN OF INTEGER

Write code to calculate the sign of an integer?

```csharp
public sbyte GetSign(sbyte a)
    => (sbyte)(a >> ((sizeof(sbyte) * 8)-1));

public sbyte GetSign2(sbyte a) =>
    (sbyte)(1 |(a >> ((sizeof(sbyte) * 8)-1)));
```

IS POWER OF 2

Write a function to check if a given unsigned integer is a power of 2

We make use of the fact the binary representation of any power of 2 is a single 1 followed by all zeros

$2^0 = 1$	00000001
$2^1 = 2$	00000010
$2^2 = 4$	00000100

Secondly we note that subtractive 1 from such a representation flips the single 1 to zero and changes all zeros following it to 1s

$2^0 - 1$	00000000
$2^1 - 1 = 1$	00000001
$2^2 - 1 = 3$	00000011

Finally we use the fact that ANDing the two forms gives a result of zero.

2^0	00000001
$2^0 - 1$	<u>00000000</u>
$2^0 \wedge (2^0 - 1)$	00000000
2^1	00000010
$2^1 - 1$	<u>00000001</u>
$2^1 \wedge (2^1 - 1)$	00000000
2^2	00000100
$2^2 - 1$	<u>00000011</u>
$2^2 \wedge (2^2 - 1)$	00000000

The code is given as follows. Note the special case for zero which is not a power of 2

```csharp
public bool IsPowerOfTwo(uint a)
{
  return (a != 0) && (a & (a-1)) == 0;
}
```

LARGEST POWER OF 2 <= X

Write statements to calculate the largest power of 2 less than or equal to x

y	01000000
$y \mathrel{|}= y \gg 1$	01100000
$y \mathrel{|}= y \gg 2$	01111000
$y \mathrel{|}= y \gg 4$	01111111

Let e be the power we are looking for. Applying the result of the previous question we obtain a number $y = (2 \times e) - 1$ The power we are looking for then becomes $\frac{(y+1)}{2}$

SMALLEST POWER OF 2 >= X

Write statements to calculate the smallest power of 2 greater than or equal to x

y	01000000
$y \mathrel{|}= y \gg 1$	01100000
$y \mathrel{|}= y \gg 2$	01111000
$y \mathrel{|}= y \gg 4$	01111111

Let e be the power we are looking for. Applying the result of the previous question we obtain a number $y = e - 1$ The power we are looking for then becomes $y+1$

Risk and Pricing Solutions

Floats

> "The exact meaning of single-, double-, and extended-precision is implementation-defined. Choosing the right precision for a problem where the choice matters requires significant understanding of floating-point computation. If you don't have that understanding, get advice, take the time to learn, or use double and hope for the best"
>
> Bjarne Stroustrup – The C++ Programming Language

Real numbers from the set $\Re$ form the basis of most scientific calculations. Any real number can be written in normalized scientific form.

$$mantissa \times \beta^e, 1.0 \leq mantissa < \beta, e \in \mathbb{Z}$$

The mantissa is a real number whose value is greater than or equal to 1.0 and less than β. The exponent is an integer. An example of a number in this form is

$$(3.1456224 \times 10^4)_{10}$$

Given an infinite number of digits in the fractional part of the mantissa any real number can be represented in this general form.

Restricting the representation size

In practice we do not have the luxury of an infinite number of digits in the mantissa and hence it is common to use a more restricted representation. In full generality, if we use a base β and have p digits of precision in the mantissa/significand we can **represent** a real number using the representation.

$$\mp(d_0.d_1d_2...d_{p-1}) \times \beta^e, (1 \leq d_0 < \beta, 0 \leq d_{i:=1..(p-1)} < \beta, e \in z, e_{min} \leq e \leq e_{max})$$

This is the same as the following.

$$\mp(d_0\beta^0 + d_1\beta^{-1} + \cdots + d_{p-1}\beta^{-(p-1)}) \times \beta^e, (1 \leq d_0 < \beta, 0 \leq d_{i:=0..(p-1)} < \beta, e \in z)$$

DEFINITIONS

1.	Mantissa/Significand	$d_0.d_1d_2...d_{p-1}$
2.	P	The number of digits in the mantissa/significand
3.	e_{min} and e_{max}	The max and minimum exponents
4.	β	The base

Properties of the finite representation

NUMBER OF DISTINCT VALUES

How many different numbers can a normalised finite form represent? One key point of the standard form is that that the digit d_0 before the decimal point must be in the set $[1, \beta - 1]$ where the digits $d_2....d_3$ can be zero$[0, \beta - 1]$. The total number of different representable values is given as

$$2 \times (\beta - 1) \times \beta^{(p-1)} \times (e_{max} - e_{min} + 1)$$

Sign   Integral   Fractional   Exponent

LARGEST AND SMALLEST VALUES

The largest representable value is $(\beta - \beta^{-(p-1)})\beta^{e_{max}}$. The smallest representable value is $-(\beta - \beta^{-(p-1)})\beta^{e_{max}}$. The smallest non-negative value is $1.0 \times \beta^{e_{min}}$.
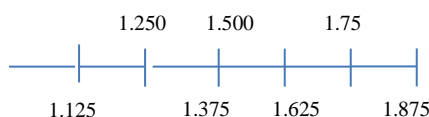
NEAREST VALUES

Example 1

Consider the following case where we use base 2, 4 digits for the mantissa and 2 digits for the exponent.
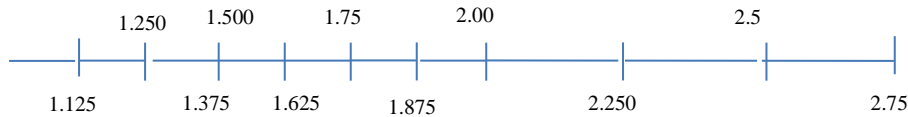
$$\mp(d_0.d_1d_2d_3) \times 2^e$$

If e is 0 then the distance between the two nearest values is $2^{-3} \times 2^0 = 0.125_{10}$

If we increase e to 1 the distance between the two nearest values becomes $2^{-3} \times 2^1 = 0.250_{10}$

Risk and Pricing Solutions

For a given exponent e and a given number of binary digits p in the mantissa the distance between the nearest two points in our representation is $2^e \times 2^{-(p-1)}$

Example 2

Consider a floating-point representation with base $\beta = 10, p = 3, e \in \{1, 0, -1\}$ The difference between the nearest two numbers depends on the exponent e.

$1.01 \times 10^1 = 10.1$	0.1
$1.01 \times 10^1 = 10.2$	0.1
$1.01 \times 10^1 = 10.3$	0.1
$1.01 \times 10^0 = 1.01$	0.01
$1.01 \times 10^0 = 1.02$	0.01
$1.01 \times 10^0 = 1.03$	0.01
$1.01 \times 10^{-1} = 0.101$	0.001
$1.01 \times 10^{-1} = 0.102$	0.001
$1.01 \times 10^{-1} = 0.103$	0.001

Generalising

Generalizing, for any base β, precision p and exponent e the distance between the nearest two values is $\beta^e \times \beta^{-(p-1)}$.

SUMMARY OF PROPERTIES OF FINITE REPRESENTATIONS

- ◆ Possible different values $2 \times (\beta - 1) \times \beta^{(p-1)} \times (e_{max} - e_{min} + 1)$

- ◆ Smallest positive value $1.0 \times \beta^{e_{min}}$

- ◆ Largest value $(\beta - \beta^{-(p-1)})\beta^{e_{max}}$

- ◆ Smallest value $-(\beta - \beta^{-(p-1)})\beta^{e_{max}}$

- ◆ Difference between nearest 2 values $\beta^e \times \beta^{-(p-1)}$

Risk and Pricing Solutions

Examples

BASE 2

If we use base 2 with 4 digits for the mantissa and $e_{max} = 1, e_{min} = -1$ our finite normalized scientific representation becomes

$$\mp(d_0.d_1d_2d_3) \times 2^e$$

The leading integer digit of the mantissa must be non-zero in normalized notation and such a binary digit is in the set[0,1]the only valid value it can take is 1. All the other digits in the mantissa and exponent can be either 0 or 1 giving us a total number of representable values as the product of

- ◆ 1   values of the integer part of the mantissa
- ◆ 2^3 values of the fractional part of the mantissa
- ◆ 2^2 values of the exponent
- ◆ 2   positive and negative values of the exponent
- ◆ 2   positive and negative values of the mantissa

Giving a total number of representable values of

$$[1 \times 2^3 \times 2^2 \times 2 \times 2] = 128$$

We note an important point here. We used 4 bits for the mantissa and 2 bits for the exponent, one bit for the sign of the mantissa and one bit for the sign of the exponent coming to a total of 8 bits. However the total number of representable values is only $128 = 2^7$. This is because the leading integer digit of the mantissa has to be one. (remember in normalized scientific notation the integer digit must be greater than or equal to one and less than β. If $\beta$is 2 then only the integer digit 1 meets this criteria). We need one bit less in the representation. When we look at computer representation of floating point numbers later we will meet this again.

- ◆ Smallest non-zero positive value $1.0 \times 2^{-1} = 0.25_{10}$
- ◆ Largest representable value $(2 - 2^{-3})2^1 = 3.75_{10}$
- ◆ Smallest representable value $-(2 - 2^{-3})2^1 = -3.75_{10}$
- ◆ Difference between nearest 2 values $2^e \times 2^{-3}$

Risk and Pricing Solutions

BASE 10

If we use base 10 with 3 digits for the mantissa and $e_{max} = 99, e_{min} = -99$ our finite normalized scientific representation becomes

- ◆ Smallest non-zero positive value 1.0×10^{-99}
- ◆ Largest representable value $(10 - 10^{-2})10^{99} = 9.99 \times 10^{99}$
- ◆ Smallest representable value $-(10 - 10^{-2})10^{99} = -9.99 \times 10^{99}$
- ◆ Difference between nearest 2 values $\beta^e \times 10^{-2}$

Risk and Pricing Solutions

Representation error

Internally real numbers are stored in binary representation, i.e. our base is 2.

$$\mp\left(d_0.d_1d_{2\ldots}d_{p-1}\right) \times 2^e, \left(d_0 = 1, d_{i:=1..(p-1)} \in \{0,1\}\right)$$

All floating-point numbers are **rational numbers** which means they have a terminating expansion in the relevant base. As such most real numbers cannot be expressed exactly. Any number with an infinite expansion cannot be represented.

Also a number which has a finite expansion in one base can have non-finite expansion in another base. If the base is 2, as in binary floating point only **rational** numbers whose denominators are powers of 2 can be represented.

$$\frac{1}{2}, \frac{1}{4}, \frac{3}{4}, \frac{1}{8}, \frac{3}{8}, \frac{5}{8}, \frac{7}{8}$$

Converting a base 10 fraction such as 0.1 to binary floating point will result in an infinite expansion which can only be approximated with a finite number of digits. The following sections how to measure this error in approximation.

ABSOLUTE ERROR

If we are approximating some real number x with a floating point representation float(x) the absolute error in the approximation is given by.

$$Absolute\ error = float(x) - x$$

We noted earlier that the difference between the two nearest values in a given representation is defined as

$$\beta^e \times \beta^{-(p-1)}.$$

If we need to round a real number to the nearest machine representable number, an upper bound on the maximum absolute error is half this space or $\frac{\beta^{e-(p-1)}}{2}$

UNITS OF THE LAST PLACE (ULPS)

Often, we are interested in the absolute error in terms of the precision of the mantissa, ignoring the exponent part. We often talk of the error in "units of the last place" which means the error in units of the last place of the mantissa. If our mantissa has precision of 4 decimal digits our floating-point representation of 54.13298 is given by 54.13. The absolute difference is given by

```
z                           5.413298  × 10¹ = 54.13298
f                           5.413     × 10¹ = 54.13
z-f                         0.000298 × 10¹  = 00.00298
```

The error in units of the last place would be .298. Since one unit in the last place is 0.01 our absolute error of 0.00298 is equal to 0.298 units in the last place. In general we can calculate the value of 1ulp as

$$1ulp = \beta^{-(p-1)}\beta^e$$

In our case we had $1ulp = \beta^{-(p-1)}\beta^e = 10^{-3}10^1 = 10^{-2} = 0.01$

In our general form where we approximate a number x using a floating-point number float(x) with base β and p digits, the error in units in the last place becomes.

$$ulps = \left| d_0.d_1d_{2\ldots}d_{p-1} - \frac{x}{\beta^e} \right| \frac{1}{\beta^{-(p-1)}}$$

Of course, given a number in units of the last place we simply multiply by the exponent term to get the absolute error

$$absolute\ error = ulps \times \beta^{-(p-1)}\beta^e$$

If we have procedure that guarantees that the floating point number chosen to approximate our real number x is the closest floating point number then the error in terms of "units of the last place" can be at most ½ times the value of the unit of the last place

$$0.5ulp = \frac{\beta}{2}\beta^e\beta^{-p}$$

RELATIVE ERROR

One problem with absolute error is that it does not consider the scale of the number being approximated. Relative error includes the magnitude of the value we are approximating.

$$Relative\ error = \left| \frac{float(x) - x}{x} \right|$$

Using the example from the previous section the relative error is given as

$$\frac{3.1459 - 3.14}{3.1459} = 0.000506$$

Now to see the relationship between absolute and relative error consider approximating the following two real numbers with the nearest floating point representatives; 9.995 and 0.005 (As we are looking at relative error the magnitude given by β^e can be ignored) The relative error of the two numbers are

$$\frac{9.995 - 9.99}{9.995} = \frac{0.005}{9.995} = 0.0005$$

$$\frac{1.005 - 1.00}{1.005} = \frac{0.005}{1.005} = 0.005$$

So, although all numbers with a given β^e have the same maximum absolute error, there relative error varies by a factor of β^e This is known as the wobble.

FROM UNITS OF THE LAST PLACE TO RELATIVE ERROR

For any chosen value of e a number of the form $\mp(d_0.d_1d_2..._{d_{p-1}}) \times \beta^e$ can vary in value from $\mp(1.00...0) \times \beta^e$ all the way to $\mp((\beta-1).(\beta-1)(\beta-1)...(\beta-1)) \times \beta^e \approx \beta^e \times \beta$ As such for any chosen value of e, where the error in terms of ulps is fixed the relative error will vary from $\frac{ulps \times \beta^e}{\beta^e \beta}$ up to $\frac{ulps \times \beta^e}{\beta^e}$ If we have chosen the nearest floating point value to the real value then we can say that the error measured in ulps is 0.5 then our relative error will vary from $\frac{1}{2}\beta^{-p}$ up to $\frac{\beta}{2}\beta^{-p}$

Put another way. For a fixed value of error in ulps the relative error can vary by a factor of β because the mantissa can vary from 1.0 up to just under $\beta - B^{-p} \approx B$.

We now consider a numerical example to cement these formulas. Consider the special case where we use a base of 10 and mantissa of 4 digits. Assuming we always choose the correct nearest floating point number then the error in ulps will be 0.5. In our case the last place has value 10^{-3} . Our chosen value of e is 3 so from our ulp error to absolute error we multiply 0.5×10^{-3} by 10^3 giving us an absolute error of 0.5 units. If we consider the value 1.000×10^3 our relative error becomes $\frac{0.5 \times 10^{-3} \times 10^3}{1.000 \times 10^3} = 0.5 \times 10^{-3} = \frac{1}{2}\beta^{-p}$ On the other hand if we consider the value 9.999×10^3 and our relative error becomes $\frac{0.5 \times 10^{-3} \times 10^3}{9.999 \times 10^3} =$ $0.5 \times 10^{-2} = \frac{\beta}{2}\beta^{-p}$ Both of these confirm what we expect. A fixed absolute ulp for a given exponent e gives a relative error that varies by a factor of B depending on the value of the mantissa

Risk and Pricing Solutions

Single Precision Float

If we consider single precision number $x = \mp q \times 10^m$ the valid values the 32 bits are allocated as

$$\mp(d_0.d_1d_{2\ldots}d_{23}) \times 2^{e_8e_7\ldots e_1}$$

♦ 1 bit represents the sign of the number $\mp$
♦ 23 bits for the fractional part of the mantissa
♦ 8 bit signed number for the exponent

The storage however is a little peculiar. We might expect that using 8 bits for the exponent would allow use to have 256 different values. However four values are reserved for special values such as plus and minus zero and plus and minus infinity.

The representation is $(-1)^s \times 2^{c-127} \times (1.f)_2$ where $-126 \leq c \leq 127$ (0 and 255 are used for special values) and $1 \leq (1.f)_2 \leq (1.11111111111111111111111)_2 = 2 - 2^{-23}$ The largest possible value representable is hence $(2 - 2^{-23})2^{127} \approx 2^{128} \approx 3.4 \times 10^{38}$. The smallest positive number becomes $(1)2^{-126} \approx 1.2 \times 10^{-38}$

The binary machine number $\varepsilon = 2^{-23}$ is the machine epsilon and is hence the smallest positive value such that $1 + \varepsilon \neq 1$. Because $2^{-23} \approx 1.2 \times 10^{-7}$ we can infer that single precision floating point has accuracy to six significant decimal figures.

So the mantissa can represent from 1 to $2 - 2^{-23}$ in increments of 2^{-23} which in decimal in approximately from 1 to $2 - 1.2 \times 10^{-7}$ in increments of 1.2×10^{-7} so since any single precision mantissa representation can be up to 1.2×10^{-7} from the real number. As such the precision of the mantissa is 6 significant figures.

Converting Between Bases

Give a number N in base λ we want to convert it to a new base β. Given

$$N = \pm(a_n\lambda^\infty+\ldots+a_2\lambda^1 + a_1\lambda^0 + b_1\lambda^{-1} + b_2\lambda^{-2}+\ldots b_\alpha\lambda^{-\alpha})_\lambda$$

we want to find the coefficients c_i and d_i such that

$$N = \pm(c_n\beta^\infty+\ldots+c_2\beta^1 + c_1\beta^0 + d_1\beta^{-1}db_2\beta^{-2}+\ldots d_\alpha\beta^{-\alpha})_\beta$$

When doing the conversion we consider the integral and fractional part separately.

INTEGRAL PART

Looking first at the integral part we have a number N

$$N = (a_n a_{n-1}\ldots a_2 a_1)_\lambda$$

We want to convert it to base β such that

$$N = (c_m c_{m-1}\ldots c_1 c_1)_\beta$$

We can rewrite this as

$$N = c_1 + \beta\left(c_2 + \beta\left(c_3+\ldots+\beta(c_m)\right)\ldots\right)_\beta$$

If we divide it by β then the remainder is clearly c_1 and the quotient is

$$c_2 + \beta\left(c_3 + \beta\left(c_4+\ldots+\beta(c_m)\right)\ldots\right)_\beta$$

If we repeat this until the quotient is zero we can read off the value of c_1 to c_n giving us the required number in the new base $(c_m c_{m-1}\ldots c_1 c_1)_\beta$

Let us consider the scenario where we want to convert the decimal number 2748 to hexadecimal. We first divide our decimal number by the new base 16

```
          171
      ┌────────
  16  │ 2748
      │ 1600
      └────────
        1148
         112
      ────────
          28
          16
      ────────
          12
```

So after this first division we know that

    1) $2748 = [(171 \times 16)] + 12$

We can't represent 171 as we only have sixteen symbols so we to divide 171 by 16

```
           10
       ┌────────
   16  │ 171
       │ 160
       └────────
          11
```

So now we know that

    2) $171 = [(10 \times 16)] + 11$

Inserting ii) into i) we get

    3) $2748 = [(\{[10 \times 16] + 11\} \times 16)] + 12 = (16^2 \times 10) + (16^1 \times 11) + (16^0 \times 12)$

Which we know is a positional number ABC_{16}

Similarly we can do the same for base 2

FRACTIONAL PART

Consider the situation where we have a fraction part $0 < x < 1$ in some base λ and we want to find the digits d_k in the representation

$$x = \sum_{k=1}^{\infty} d_h \beta^{-k} = (0. d_1 d_2 d_3 \dots)_\beta$$

We first note that

$$\beta x = (d_1. d_2 d_3 \dots)_\beta$$

So if we take our fractional part and multiply it by β then the resulting integral component is the d_1 we can similarly repeat the process to find the digits $d_2 .. d_m$

EXAMPLE 1 CONVERT 0.526_{10} **TO BASE 8**

i) $\quad 8 \times 0.526_{10} = 4.208 \therefore 0.526_{10} = \frac{1}{8}4 + \frac{1}{8}(0.208)$

$$8 \times 0.208_{10} = 1.664 \therefore 0.208_{10} = \frac{1}{8}1 + \frac{1}{8}(0.664)$$

$$8 \times 0.664_{10} = 5.312 \therefore 0.664_{10} = \frac{1}{8}5 + \frac{1}{8}(0.312)$$

$$8 \times 0.312_{10} = 2.496 \therefore 0.312_{10} = \frac{1}{8}2 + \frac{1}{8}(0.496)$$

$$0.526_{10} \approx 0.4152_8$$

Floats – Questions

DECIMAL FRACTION TO BINARY FRACTION

Given a decimal fraction such as 0.46 return a string representation its binary. If the number cannot be represented exactly in binary in n bits throw an exception

```csharp
private string ConvertIntegralPart(double b, int maxDigits)
{
  StringBuilder result = new StringBuilder("0.");
  if (b >= 1.0) throw new ArgumentException("Input must be a fraction");

  double frac = 0.5;


  while (b >=0 && maxDigits-- > 0)
  {
      if (b >=frac)
      {
            result.Append("1");
            b-= frac;
      }
      else
      {
            result.Append("0");
      }

      frac /= 2;
  }

  return result.ToString();
}
```

Risk and Pricing Solutions

PARSE FLOAT

CHANGE FRACTIONAL BASE

Given a string representation of an fraction N in base convert it to a string representation of a fraction in base β. For example given the input "0.75" with $\lambda = 10$ and $\beta = 2$ it would return "0.11"

Consider the situation where we have a fraction part $0 < x < 1$ in some base λ and we want to find the digits d_k in the representation

$$x = \sum_{k=1}^{\infty} d_h \beta^{-k} = (0.d_1 d_2 d_3 \ldots)_\beta$$

We first note that

$$\beta x = (d_1.d_2 d_3 \ldots)_\beta$$

So if we take our fractional part and multiply it by β then the resulting integral component is the d_1 we can similarly repeat the process to find the digits $d_2 .. d_m$

Fractional change of base

```
private string ConvertFractionalPart(string input, int l, int b,
    int maxDigits=16)
{
    var fractionString = input.Split('.')[1];
    var result = new StringBuilder("0.");

    // Calculate the decimal Fraction
    double decimalFraction = 0.0;
    for (int i = 0; i < fractionString.Length; i++)
    {
        decimalFraction +=
            fractionString[i].ToIntDigit() * Math.Pow(l,-(i+1));

    }

    int digitIdx=0;
    while (decimalFraction > 0.0 && digitIdx++ < maxDigits)
    {
        decimalFraction = (decimalFraction * b);
        int digit = (int)decimalFraction;
        result.Append(digit.ToChar());

        decimalFraction -= digit;
    }

    return result.ToString();
}
```

CHANGE FRACTIONAL BASE

DIVISION TO FLOATING POINT

Modify your answer from the previous section to return a floating point result rather than quotient and remainder?

```csharp
public string IntegerDivisionWithFloatingPointResult(string dividend, int divisor,
      int b = 10, int maxDigits = 8)
{
      StringBuilder quotient = new StringBuilder();
      int remainder = 1;
      int dd = 0;

      for (int idx = 0; (idx < dividend.Length || remainder > 0)
                  && idx < maxDigits; idx++)
      {
            // Add in a decimal point
            if (idx == dividend.Length)
                  quotient.Append(".");

            // idx.1 copy in next digit into temporary dividend dd
            if (idx < dividend.Length)
                  dd = (dd * b) + dividend[idx].ToIntDigit();
            else
                  // The integer dividend has no more digits so we just increase
                  // by a factor of b as we move to the right side of the point
                  // point
                  dd = (dd * b);

            // idx.2 calculate partial quotient and set into quotient[idx]
            int partialQuotient = dd / divisor;
            quotient.Append(partialQuotient.ToChar());

            // idx.3 calculate this temporary as part of calculating remainder
            int temp = partialQuotient * divisor;

            // idx.4 Calculate the remainder
            remainder = dd % divisor;

            // the remainder will form the basis of dd[idx+1]
            dd = remainder;
      }

      return quotient.ToString();
}
```

EVALUATING ARITHMETIC EXPRESSIONS

Write code to evaluate arithmetic expressions?

```csharp
public double Evaluate(string expression)
{
        Stack<double> values = new Stack<double>();
        Stack<string> ops = new Stack<string>();

        string[] tokens = expression.Split(' ');

        foreach (var token in tokens)
        {
                if (operators.Contains(token))
                        ops.Push(token);
                else if (token.Equals(")"))
                {
                        double arg1 = values.Pop();
                        double arg2 = values.Pop();

                        switch(ops.Pop())
                        {
                                case "+":
                                        values.Push(arg1+arg2);
                                        break;
                                case "-":
                                        values.Push(arg1 - arg2);
                                        break;
                                case "*":
                                        values.Push(arg1 * arg2);
                                        break;
                                case "/":
                                        values.Push(arg1 / arg2);
                                        break;
                        }
                }
                else if (token.Equals("("))
                {
                        // Do Nothing
                }
                else
                {
                        values.Push(double.Parse(token));
                }
        }

        return values.Pop();
}
```

PRECISION AND RANGE

PRECISION OF FLOAT

What is the precision of a single precision point floating point number and why?

Six significant figures

The binary machine number $\varepsilon = 2^{-23}$ is the machine epsilon and is hence the smallest positive value such that $1 + \varepsilon \neq 1$. Because $2^{-23} \approx 1.2 \times 10^{-7}$ which if we write it out we see

0.00000012 *If we see this value what it really means is that the value is*

$$0.00000018 > x > 0.00000006$$

So only the sixth significant figure is accurate.

RANGE OF FLOAT

What is the range of a single precision floating point and why?

From $\approx 2^{128}$ to $\approx -(2^{128})$ which is approximately from 3.4×10^{38} to $-(3.4 \times 10^{38})$

The reason being that the largest absolute value representable in single precision is given by $(2 - 2^{-23})2^{127} \approx 2^{128} \approx 3.4 \times 10^{38}$ as the mantissa has 23 bits and the exponent has 8 bits.

PRECISION OF DOUBLE

What is the precision of a double precision point floating point number and why?

The binary machine number $\varepsilon = 2^{-53}$ is the machine epsilon and is hence the smallest positive value such that $1 + \varepsilon \neq 1$. Because $2^{-53} \approx 1.1 \times 10^{-16}$ so only to the 15 significant figure is correct.