Bit Questions

BASIC BIT LEVEL

- Bit Properties
- Manipulation
- Getting/Setting bits
- Brain Teasers
- Integers

Basic 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?

a ^ 1s = ~0

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
```

```
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
```

```
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
```

```
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
```

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

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
```

```
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
```

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

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	765432**2**10
n	00000**1**01
n >> 2	0000000**1**
1	00000001
(n >> 2)&1	00000001
((n >> 2)&1)>0	**true**

```
public bool GetBit(int n, int i) => ((n >> i) & 1) > 0;
```

Write a function set the bit at index i to 1

```
public int SetBit(int n, int i) => (1 << i) | n;
```

Write a function clear the bit at index i to 0

```
public int ClearBit(int n, int i) => ~(1 << i ) & n;
```

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

```
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

```
public int SetFromMsbToI(int n, int i)
{
  return (~0 << i) | n;
}
```

Write a function that clears all bits from 0 through to i inclusive

```
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

```
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	000010**10**
$10 - 1$	000010**01**

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

10	000010**1**0
$10 - 1$	00001001
$10 \,\&\, (10 - 1)$	000010**0**0

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	000010**10**
$10 - 1$	000010**01**

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
$10 \mid (10 - 1)$	00001111

Calculate Number of Set Bits

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

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
$x \mid = x \gg 1$	01100000
$x \mid = x \gg 2$	01111000
$x \mid = x \gg 4$	01111111

MSB > n

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

```
idx                        76543210
x0                         00010000
((1 << 5)-1)               00011111
~((1 << 5)-1)              11100000
x & ~((1 << 5)-1) != 0     true
```

Next Int Same 1 Count

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

```
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
x_0	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
x_0	01001111
mask1	10111111
$x_1 = x_0$&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
x_1	00001111
mask2	11000000
$x_2 = x_1$ & mask2	00000000

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

$$mask3 = \left(\left(1 \ll (i-1)\right) - 1\right) \ll (i - io)$$

idx	76543210
x_2	00000000
mask3	00111110
$x_3 = x_2$ | mask3	00111110

The source code is then

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

       int zeroCount = 0;
```

```
    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;
}
```

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;
}
```

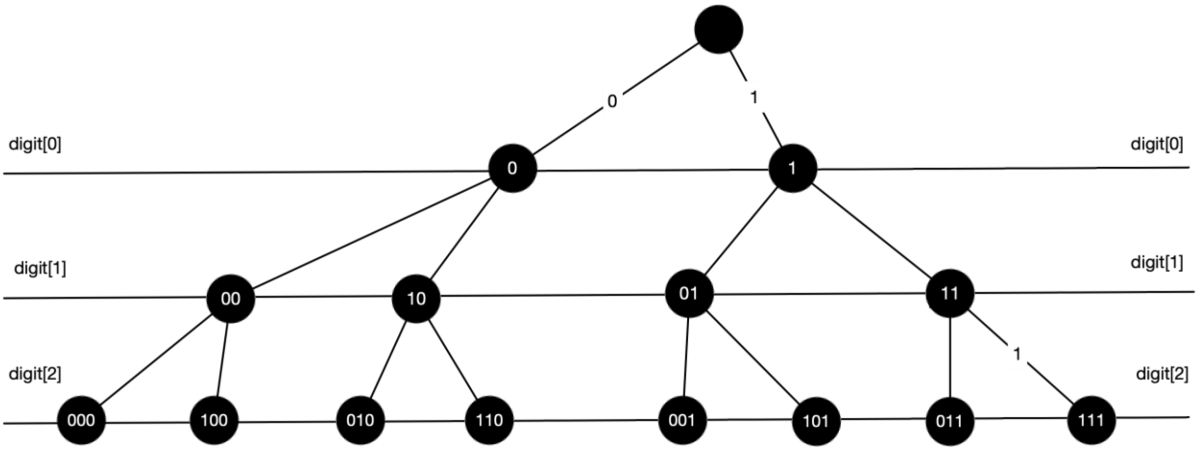
If the array is sorted we can use the relationship between each array element and its index to perform a binary search from O(logn)

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

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	8
0	1	2	3	4	5	6	8
0	1	2	3	4	5	6	8
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.




```
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

```
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

Integers

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*

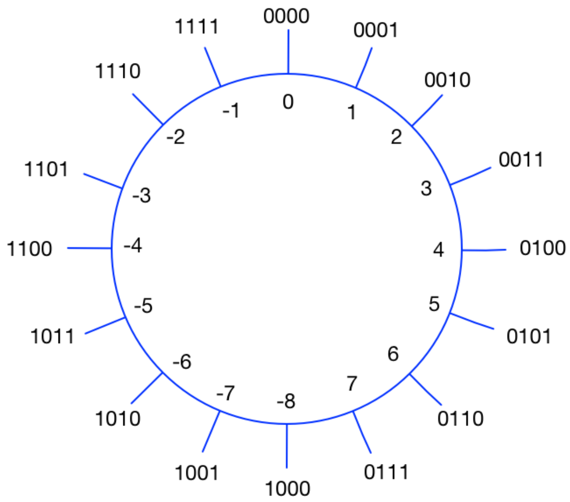
```
    1111
+   0001
   10000
```

```
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?

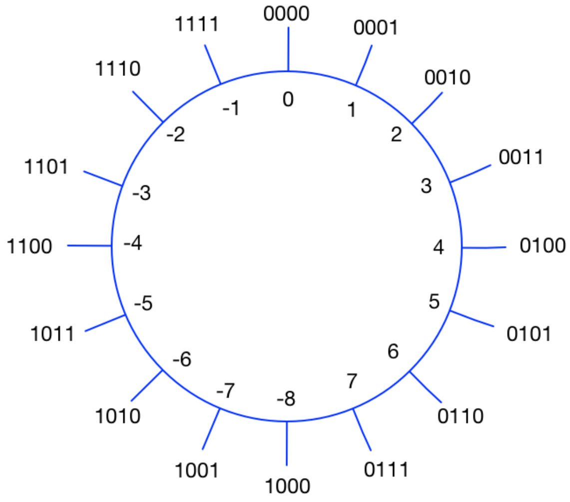
$$(a + [2^n - b])_{mod_{2^n}} = (a - b)_{mod_{2^n}}$$

$$b + \sim b = 11 \dots 1 = 2^n - 1$$

$$(a - b)_{mod_{2^n}} (a + \sim b + 1)_{mod_{2^n}}$$

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

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

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

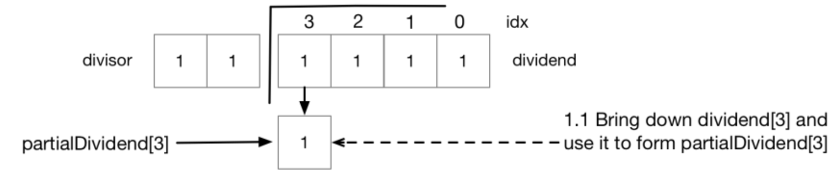
It is very slow

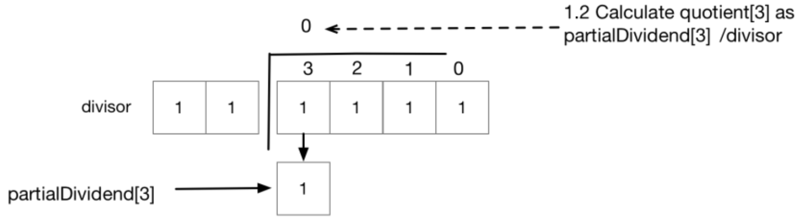
Use your function to do signed division?

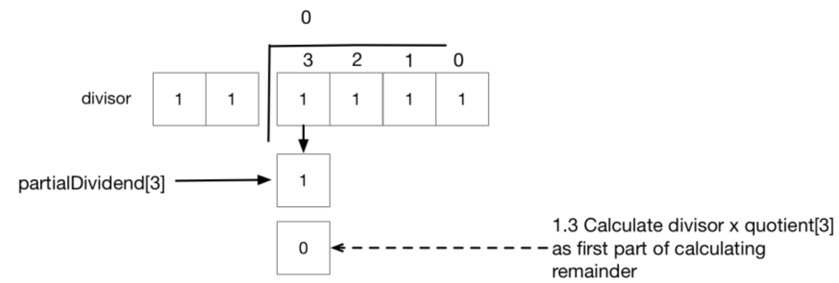
```
public (int q, int r) Divide(int dividend, int divisor)
{
	if (divisor == 0) throw new DivideByZeroException();

	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);
}
```

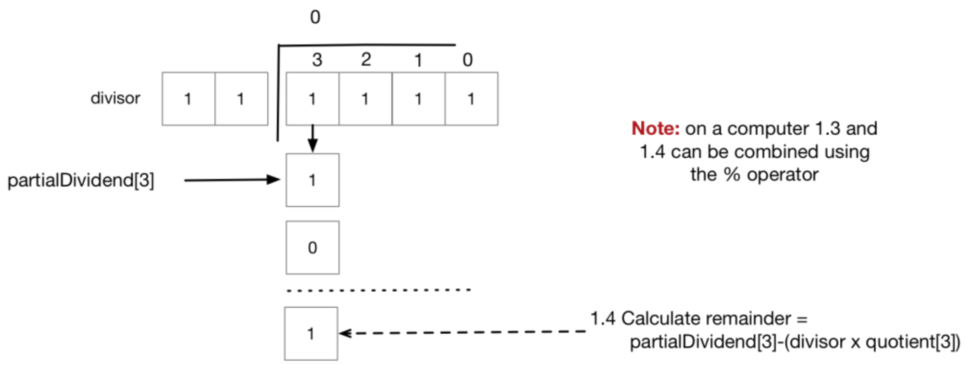
Division By Binary Long Division

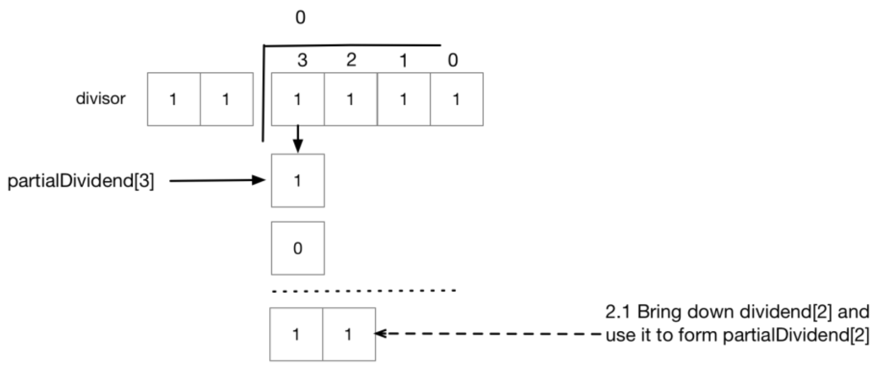
Write code to perform binary long division unsigned











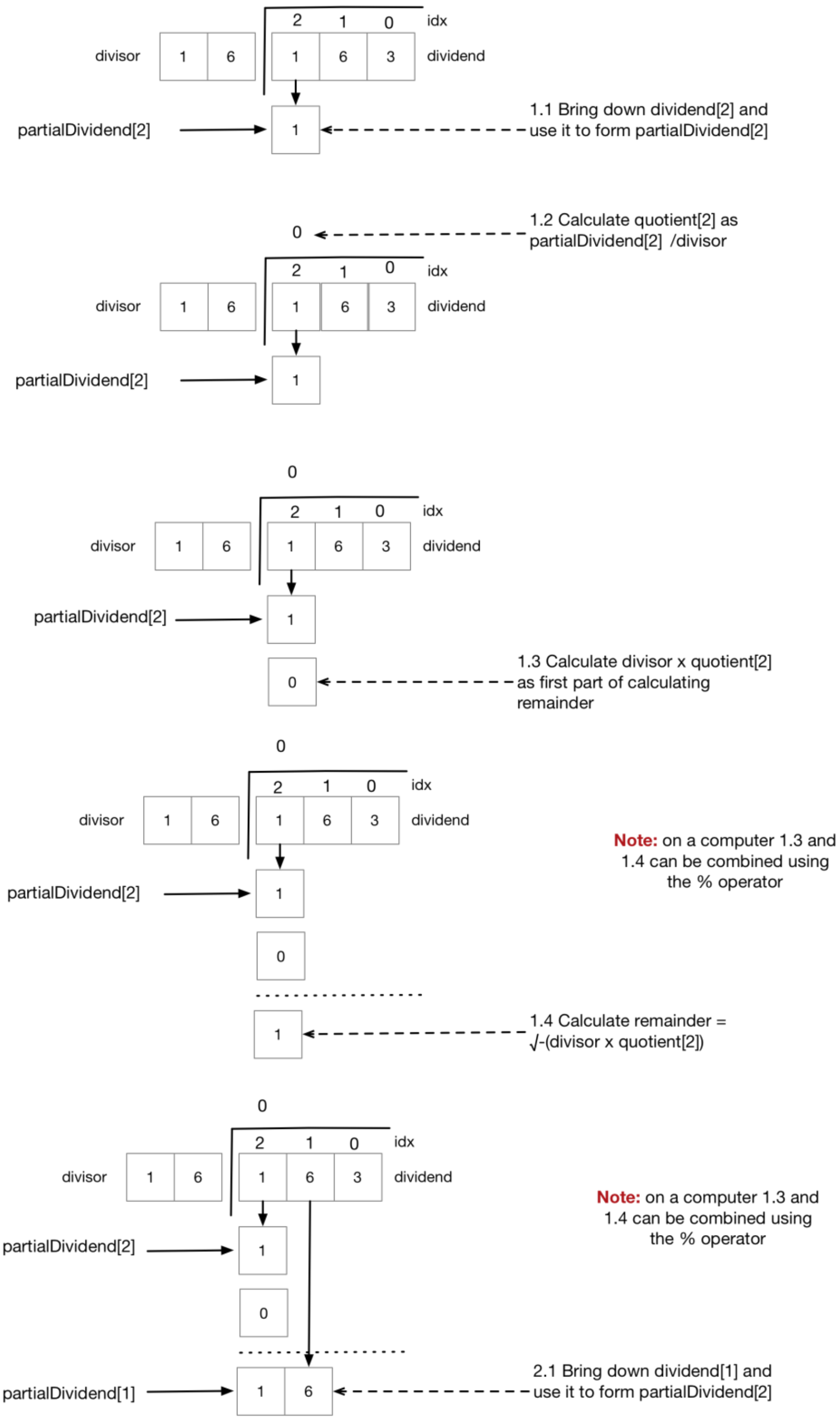
```
public (int quotient, int remainder) UnsignedDivide(int dividend, int
divisor)
{
    int numBits = sizeof(int) * 8;
    int quotient = 0;

    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?



```
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

```
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 $\boldsymbol{\beta}$. For example given the input "10" with $\boldsymbol{\lambda} = \mathbf{10}$ and $\boldsymbol{\beta} = \mathbf{2}$ 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

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

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

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

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

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

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

```
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*

$b^{k-1} < n + 1 \leq b^k$ *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);
```

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);
}
```

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

- 0xxxxxxxx If a >= b or 1xxxxxxxx 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.

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;
}
```

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?

```
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
$2^0 - 1$	00000000
$2^0 \wedge (2^0 - 1)$	00000000
2^1	
$2^1 - 1$	00000010
$2^1 \wedge (2^1 - 1)$	00000001
	00000000
2^2	00000100
$2^2 - 1$	00000011
$2^2 \wedge (2^2 - 1)$	00000000

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

```
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
$y \mid = y \gg 1$	01100000
$y \mid = y \gg 2$	01111000
$y \mid = y \gg 4$	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
$y \mid = y \gg 1$	01100000
$y \mid = y \gg 2$	01111000
$y \mid = y \gg 4$	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*

Floats

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

```
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();
}
```

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 $\boldsymbol{\beta}$. For example given the input "0.75" with $\boldsymbol{\lambda} = 10$ and $\boldsymbol{\beta} = 2$ 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 repeate 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?

```
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();
}
```

Number Theory

Is Prime Naïve

Calculate is prime using simple brute force. What is the runtime?

The following naïve implementation is O(n)

```
public static bool IsPrimeNaive(int n)
{
     if (n <= 1) return false;

     for (int i = 2; i < n; i++)
     {
           if (n % i ==0)
                 return false;
     }
     return true;
}
```

The runtime is O(N)

Is Prime Simple Optimisation

Optimise it. What is the runtime?

```
public bool IsPrimeUsingSquareRoot(int n)
{
      if (n < 2)
            return false;

      if (n == 2)
            return true;

      // The definition of a prime is an integer x
      // which is not exactly divisible by any
      // number other than itself and one. If a
      // number x is not prime it can be written as
      // the product of two factors a x b. If both
      // a and b were greater than the square root of
      // x then a x b would also be greater than x and hence
      // a x b is not x. SO testing all factors up to floor(root(x))
      // is sufficient as if one factor is floor(root(x)) the other factor must
      // be less than that

      // hence test the n-2 integers from
      // 2,..., Floor(Root(N))
      return Enumerable.Range(2, (int)Math.Floor(Math.Sqrt(n)))
            .All(i => n % i > 0);
}
```

`The runtime is O(Root(n))`

Fundamental Theorem Of Arithmetic

What is the fundamental theorem of arithmetic?

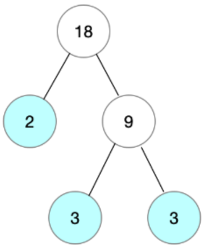
Any integer is either prime itself prime or can be expressed as a product of prime factors

$$x = p_1^{a_1} p_2^{a_2} \dots p_n^{a_n}$$

Where $p_1 \dots p_n$ *are successive primes and* $a_1 \dots a_n$ *are powers of that prime. For any given p, the corresponding* a *can be zero.*

How do we find the prime factorisation?

By continually dividing through



$$18 = 2^1 \times 3^2$$

Highest Common Factor

What is the HCF of x and y?

The biggest integer that divides into x and y

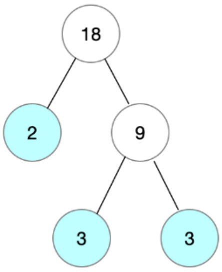
Give a definition of HCF in term of prime numbers?

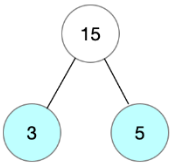
$$x = p_1^{a_1} p_2^{a_2} \dots p_n^{a_n}$$

$$y = p_1^{b_1} p_2^{b_2} \dots p_n^{n}$$

$$hcm(x, y) = p_1^{min(a_1,b_1)} p_2^{min(a_2,b_2)} \dots p_\infty^{min(a_n,b_n)}$$

Calculate HCF of 18 and 15





$18 = 2^1 \times 3^2$, $15 = 2^0 \times 3^1 \times 3^5$

$hcf(15,18) = 2^{\min(0,1)} \times 3^{\min(1,2)} \times 5^{\min(0,1)} = 3$

Lowest Common Multiple

What is the LCM of x and y?

The smallest number that is a product of both x and y

Give a definition of LCM in term of prime numbers?

Given two integers x and y and their corresponding prime factorisations

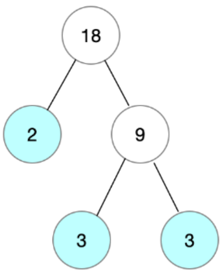
$$x = p_1^{a_1} p_2^{a_2} \dots p_n^{a_n}$$

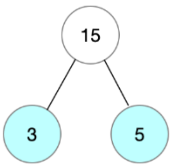
$$y = p_1^{b_1} p_2^{b_2} \dots p_n^{n}$$

We can calculate the lowest common multiple as

$$lcm(x, y) = p_1^{max(a_1, b_1)} p_2^{max(a_2, b_2)} \dots p_\infty^{max(a_n, b_n)}$$

Calculate the LCM of 18 and 15





$18 = 2^1 \times 3^2$

$15 = 2^0 \times 3^1 \times 3^5$

$lcm(15,18) = 2^{\max(0,1)} \times 3^{\max(1,2)} \times 5^{\max(0,1)} = 2 \times 3^2 \times 5^1 = 90$

Relating HCF and LCM

Give an expression relating HCF and LCM

$$lcm(x, y) \times hcf(x, y) = x \times y$$

Prove it

Given two integers x and y and their corresponding prime factorisations

$$x = p_1^{a_1} p_2^{a_2} \dots p_n^{a_n}$$

$$y = p_1^{b_1} p_2^{b_2} \dots p_n^{n}$$

We can show there is a relationship between lcm and hcf.

$$lcm(x, y) = p_1^{max(a_1, b_1)} p_2^{max(a_2, b_2)} \dots p_\infty^{max(a_n, b_n)}$$

$$hcf(x, y) = p_1^{min(a_1, b_1)} p_2^{min(a_2, b_2)} \dots p_\infty^{min(a_n, b_n)}$$

$$hcf(x, y) \times lcm(x, y) = p_1^{min(a_1, b_1) \times max(a_1, b_1)} p_2^{min(a_2, b_2) \times max(a_1, b_1)} \dots p_\infty^{min(a_n, b_n) \times max(a_n, b_n)}$$

$$hcf(x, y) \times lcm(x, y) = p_1^{a_1 \times b_1} p_2^{a_2 \times b_2} \dots p_n^{a_n \times b_n} = x \times y$$

So we now know that

$$lcm(x, y) = \frac{x \times y}{hcf(x, y)}$$

Why is this useful?

We have efficient algorithms for calculating the hcf, whereas we do not have efficient algorithms for carrying out prime factorisation

What is a the basis for Euclids algorithm for HCF?

$$gcd(a, b) = gcd(b, a\%b)$$

Highest Common Factor

Implement Euclids algorithm for HCF. What is the runtime?

```
/// <summary>
    /// Implementation of Euclids algorithm
    /// </summary>
    /// <param name="a"></param>
    /// <param name="b"></param>
    /// <returns></returns>
    public static int HighestCommonFactor(int a, int b)
    {
      if (a < b)
      {
        return HighestCommonFactor(b, a);
      }
      else
      {
        int remainder = a % b;

        if (remainder == 0)
        {
          return b;
        }
        else
        {
          return HighestCommonFactor(b, remainder);
        }
      }
    }
```

Lowest Common Multiple

Calculate LCM using the algorithm from the previous section?

$$lcm(x, y) = \frac{x \times y}{hcf(x, y)}$$

Fibonacci Iterative

Write Fibonacci iterative

```
public static int FibonacciIterative(int n)
{
    // f0 f1 f2 f3 f4
    //  0  1  1  2  3

    // fn  = Fibonacci(n)
    // fn1 = Fibonacci(n+1)
    // fn2 = Fibonacci(n+2)
    int fn = 0, fn1 = 1;

    for (int i = 0; i < n; i++)
    {
        int fn2 = fn + fn1;
        fn = fn1;
        fn1 = fn2;
    }
    return fn;
}
```

Analyse the runtime?

O(N)

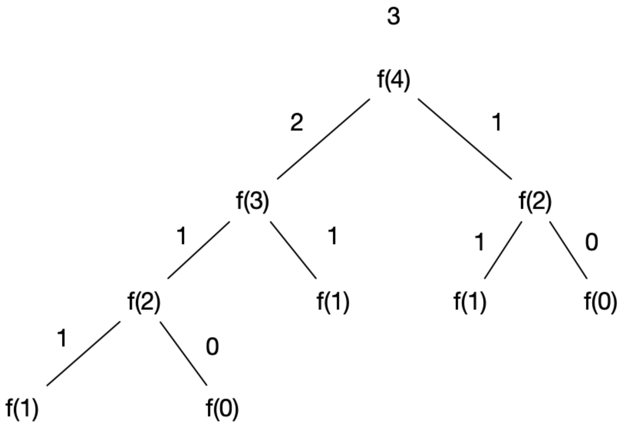
Fibonacci Recursive

Write Fibonacci recursive

```
public static int FibonacciRecursive(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;

    return FibonacciRecursive(n - 1) + FibonacciRecursive(n - 2);
}
```

Analyse the runtime?

Consider the following call graph of f(4)



The runtime is upper bounded by $O(2^n)$. *There is a slighter tighter bound*

Improve the performance of the recursive algorithm

This is O(2n) =O(n)

```
public static int FibonacciRecursiveMemo(int i)
{
    var cache = new int[i + 1];

    int F(int x)
    {
        if (x == 0 || x == 1) return x;

        if (cache[x] == 0) cache[x] = F(x - 1) + F(x - 2);

        return cache[x];
    }

    return F(i);
}
```

Floating Point

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

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