Bit Questions

Basic Bit Level

[Bit Properties](#_Properties)

[Manipulation](#_Manipulation)

[Getting/Setting bits](#_Getting/Setting_bits)

[Brain Teasers](#_Brain_Teasers)

[Integers](#_Integers)

## Basic Bit

### Properties

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 resuls is 1s

Perform bitwise negation without using the ~ operator?

a ^ 1s = a ^ ~0

### 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 7654**3**210

1 00000001

1 << 3 0000**1**000

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 7654**3**210

1 00000001

1 << 3 0000**1**000

~(1 << 3) 1111**0**111

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 76543**210**

~0 11111111

~0 << 3 11111**000**

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 76543**210**

1 << 3 00001000

(1 << 3)-1 00000**111**

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 7**6543**210

(1 << 6-3+1) 00010000

(1 << 6-3+1)-1 00001111

((1 << 6-3)-1)<< 3 0**1111**000

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 7**6543**210

(1 << 6-3+1) 00010000

(1 << 6-3+1)-1 00001111

((1 << 6-3)-1)<< 3 01111000

~(((1 << 6-3)-1)<< 3) 1**0000**111

public static sbyte MaskSix(int i, int j)

=> (sbyte)~(((1 << j – i+1) - 1) << i);

Implement integer subtraction without using the – key.

We make use of the properties of twos complement

public static sbyte Subtract(sbyte a, sbyte b) =>

(sbyte)(a + ~b +1);

### Getting/Setting bits

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 76543**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

### Right Shift Based

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
* 1xxxxxxxx If a < b

If we perform a right arithmetic shift of 7 bits (sizeo 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);}

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 completment 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;

}

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);

}

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)));

Write a function to calculate

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;

}

### Least Significant set bit based

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.

|  |  |
| --- | --- |
|  | 000010**10** |
|  | 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

|  |  |
| --- | --- |
|  | 000010**1**0 |
|  | 00001001 |
|  | 000010**0**0 |

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.

|  |  |
| --- | --- |
|  | 000010**10** |
|  | 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.

|  |  |
| --- | --- |
|  | 00001100 |
|  | 00001011 |
|  | 00001111 |

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

|  |  |
| --- | --- |
|  | **00000001** |
|  | **00000010** |
|  | **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

|  |  |
| --- | --- |
|  | **00000000** |
|  | **00000001** |
|  | **00000011** |

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

|  |  |
| --- | --- |
|  | **00000001**  **00000000**  **00000000**  **00000010**  **00000001**  **00000000** |
|  | **00000100**  **00000011**  **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;

}

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 algorith 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 aglorithm invented by Brian Kernigan. It key idea is that if we siubtract 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 signigicant 1.

|  |  |
| --- | --- |
|  | **00000101**  **00000100**  **00000100**  **00011100**  **00000001**  **00000000** |
|  | **00000100**  **00000011**  **00000000** |

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 kernighans 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;

}

### Most Significant set bit based

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

|  |  |
| --- | --- |
|  | 01000000 |
|  | 01100000 |
|  | 01111000 |
|  | 01111111 |

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

Let e be the power we are looking for. Applying the result of the previous question we obtain a number The power we are looking for then becomes

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

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

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

### Miscelaneous

Given a decimal fraction such as 0.46 return a string representating its binary. If the number cannot be reprented 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();

}

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;

}

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 7**6**543210

x0 0**1**001111

i 6

The index of the non-trailing 1 is , the number of zeroes is and the number of ones is In order to unset the first non-trailing 1 which is at index 6 we create a mask

idx 7**6**543210

x0 0**1**001111

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 ones imediately to the right of. First we create a mask for the zeroing

idx 7**6**543210

x1 00001111

mask2 11000000

x2= x1 & mask2 00000000

Now we put the ones immediately to the right of i. Our mask is

idx 7**6**543210

x2 00000000

mask3 00111110

x3= x2 | 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;

}

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

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

maskodd  10101010

xodd = x & maskodd 10101000

idx 76543210

x 10111101

maskeven 01010101

xeven = x & maskeven 00010101

We shift the odd bits into even positions and even bits into odd positions

idx 76543210

xeven =  xeven <<< 1 00101010

xodd = xodd >>> 101010100

result = xeven |xodd 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);

}

## Integers

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

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;

}

User your function to write unsigned subtraction?

Write a function Log(double x, double b) that takes a double value and a double base and returns 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

Proof

Let and hence

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);

}

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

A number n represented in a base b number system will consist of k digits if and only if . In other words 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.

.

From the properties of the floor function we know that and hence in our case

Expression using the ceiling function

We can achive a similar result that uses the ceiling function by adding one to the inequality . so we get

and taking logarithms we get

From the properties of the ceiling function we know that and hence that

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);

Given a string representation of an integer N in base convert it to a string representation of an integer in base . For example given the input “10” with and it would return “1010”

Let then we want to find the coefficients  such that

We have a number N

That we want to convert to base such that

We can rewrite this as

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

If we repeat this until the quotient is zero we can read off the value of to giving us the required number in the new base

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);

}

Given a string representation of an fraction N in base convert it to a string representation of a fraction in base . For example given the input “0.75” with and it would return “0.11”

Consider the situation where we have a fraction part in some base and we want to find the digits in the representation

We first note that

So if we take our fractional part and multiply it by then the resulting integral component is the we can similarly repeate the process to find the digits

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();

}

Write the simplest possible code to perform unsigned integer division?

Division is just repeated subtraction so we have

public (uint q, uint r) Divide(uint dividend, uint divisor)

{

uint quotient = 0;

uint remainder = dividend;

while (divisor <= remainder)

{

remainder = remainder - divisor;

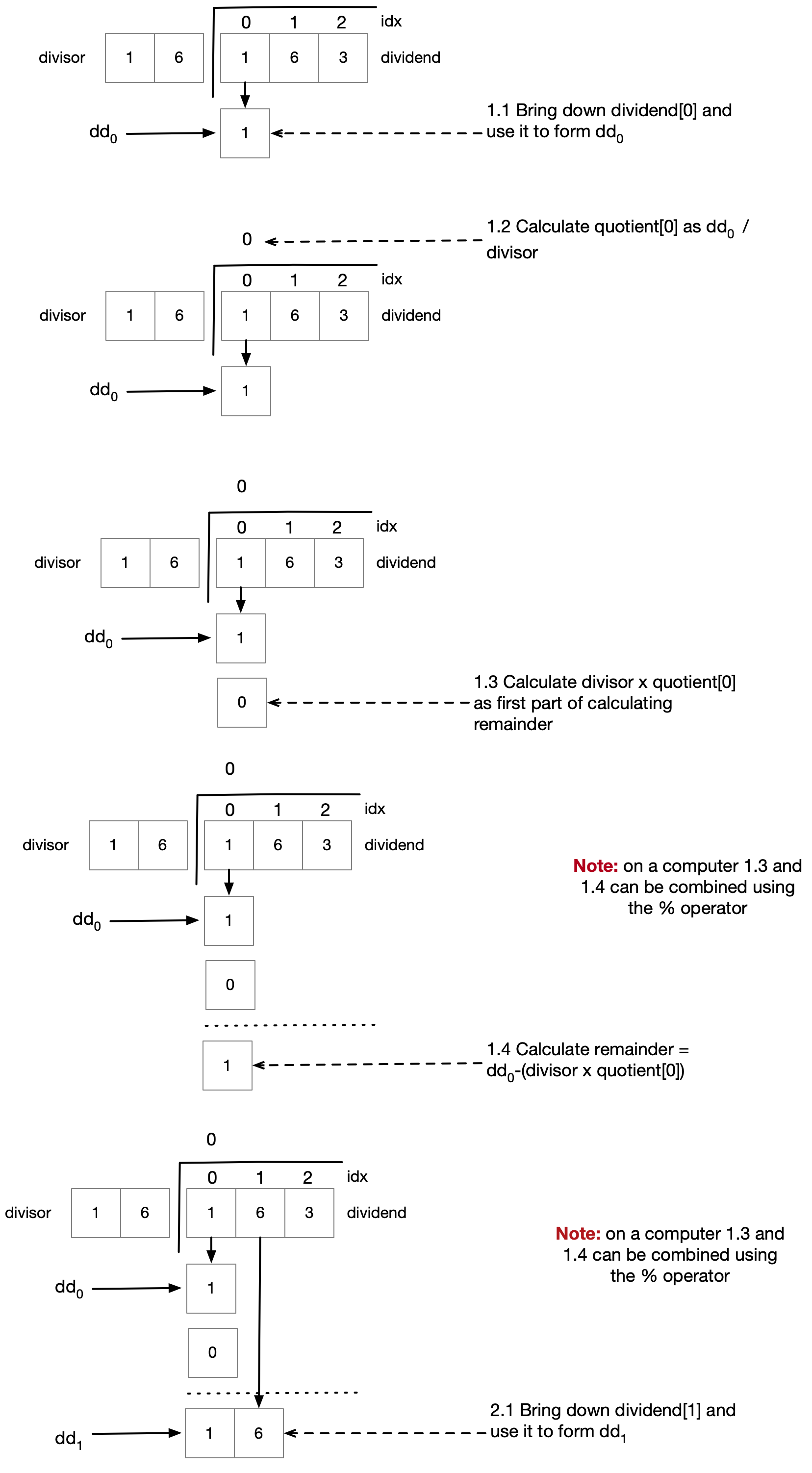
quotient++;

}

return (quotient,remainder);

}

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++)

{

// 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);

}

}

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