Username: Pralay Patoria **Book:** Coding Interviews: Questions, Analysis & Solutions. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Bit Operations

People get accustomed to the decimal system in daily life. However, a number is stored as a binary in computers, which is a sequence of 0s and 1s. For example, the decimal 2 is 10 in binary, and decimal 10 is 1010 in binary.

Besides 2 and 10, numbers can be converted with other bases. For instance, the time system has 60 as its base to record seconds, minutes, and hours. The following problem is an interesting interview question about numbers with uncommon bases: in Microsoft Excel, "A" stands for the first column, "B" for the second column, …, "Z" for the 26^{th} column, "AA" for the 27^{th} column, "AB" for the 28^{th} column, and so on. Please write a function to get the column index according to a given string.

A column index in Excel is converted with base 26, so it essentially requires converting a number with base 26 to a decimal number.

Each bit in a binary number, 0 or 1, can be manipulated. Bit operations can be divided into two categories. The first category contains bitwise operations, which include AND, OR, XOR, and NOT. The first three are binary operators taking two input bits, which are summarized in Table 4-3.

Table 4-3. Bitwise AND, OR, and XOR

Bitwise AND (&)	0 & 0 = 0	1 & 0 = 0	0 & 1 = 0	1 & 1 = 1
Bitwise OR (|)	0 | 0 = 0	1 | 0 = 1	0 | 1 = 1	1 | 1 = 1
Bitwise XOR(^)	0 ^ 0 = 0	1 ^ 0 = 1	0 ^ 1 = 1	1 ^ 1 = 0

The bitwise NOT is a unary operator, which has only one argument. It produces the opposite of the input bit: a 1 if the input is a 0, or a 0 if the input is a 1.

The second category of bit operations contains shift operations. The left-shift operator in $m << n$ shifts m to the left by n bits and inserts n 0s at the lower-order bits. For example:

```
00001010 << 2 = 00101000

10001010 << 3 = 01010000
```

The right-shift operator in $m >> n$ shifts m to the right by n bits. It inserts n 0s at the higher-order bits if m is positive and inserts n 1s if m is negative. In the following examples, it takes signed 8-bit numbers as the left operand for simplicity:

```
00001010 >> 2 = 00000010

10001010 >> 3 = 11110001
```

The right operator >> is called a signed right-shift in Java, and a new operator >>> named unsigned right-shift is introduced. The operator in $m >>> n$ also shifts m to the right by n bits, and it always inserts 0s at the higher-order bits no matter what the sign of m is. In the following two examples, it takes signed 8-bit numbers as the left operand for simplicity again:

```
00001010 >>> 2 = 00000010

10001010 >>> 3 = 00010001
```

The unsigned right-shift operator >>> is available only in Java, but unavailable in C/C++/C#.

Number of 1s in Binary

Question 35 Please implement a function to get the number of 1s in the binary representation of an integer. For example, the integer 9 is 1001 in binary, so it returns 2 since there are two bits of 1.

It looks like a simple question about binary numbers, and we have many solutions for it. Unfortunately, the most intuitive solution for many candidates is incorrect. We should be careful.

Check the Rightmost Bit with Endless Loop

When candidates are given this problem during an interview, many of them find a solution in a short time: the solution checks whether the rightmost bit is 0 or 1, and then right-shifts the integer one bit and checks the rightmost bit again. It continues in a loop until the integer becomes 0.

How do you check whether the rightmost bit of an integer is 0 or 1? It is simple since we have the AND operation. There is only one bit of 1 in the binary representation of the integer 1, which is the rightmost bit. When we have the bitwise AND operation on an integer and 1, we can check whether the rightmost bit is 0 or 1. When the result of the AND operation is 1, it indicates the rightmost bit is 1; otherwise, it is 0. We can implement a function based on this solution quickly, as shown in Listing 4-20.

Listing 4-20. *C Code for Number of 1 in Binary (Version 1)*

```
int NumberOf1(int n) {

 int count = 0;

 while(n) {

     if(n & 1)

         count ++;
```

```
        n = n >> 1;

    }


    return count;

}
```

Interviewers may ask a question when they are told this solution: What is the result when the input integer is a negative number such as 0x80000000? When we right-shift the negative number 0x80000000 for a bit, it becomes 0xC0000000 rather than 0x40000000. The integer 0x8000000 is negative before the shift, so it is guaranteed to be negative after the shift. Therefore, when a negative integer is right-shifted, the first bit is set as 1 after the right-shift operation. If we continue to shift to the right side, a negative integer will be 0xFFFFFFFF eventually and it is trapped in an endless loop.

Left-Shift Operation on 1

If we are using Java, the unsigned right-shift operator >>> can be utilized to eliminate the problem of negative numbers.

We have to utilize other strategies if we are developing in C/C++. Instead of shifting the input integer *n* to right, we may shift the number 1 to left. We may check first the least important bit of the input number *n*, and then shift the number 1 to the left and continue to check the second least important bit of *n*. The code can be revised as shown in Listing 4-21.

Listing 4-21. *C Code for Number of 1 in Binary (Version 2)*

```
int NumberOf1(int n) {

 int count = 0;

 unsigned int flag = 1;

 while(flag) {

     if(n & flag)

         count ++;


     flag = flag << 1;

 }


 return count;

}
```

Minus One and Then Bitwise AND

Let's analyze what happens when you have a binary number minus 1. There is at least one bit 1 in a non-zero number. We first assume the rightmost bit is 1. It becomes 0 if you subtract 1 and other bits then keep unchanged.

Second, we assume the rightmost bit is 0. Since there is at least one 1 bit in a non-zero number, we suppose the m^{th} bit is the rightmost bit of 1. When it is minus 1, the m^{th} bit becomes 0, and all 0 bits behind the m^{th} bit become 1. For instance, the second bit of binary number 1100 is the rightmost 1 bit. When you take 1100 minus 1, the second bit becomes 0, and the third and fourth bits become 1, so the result is 1011.

In both situations above, the rightmost 1 bit becomes 0 when you subtract the 1. When there are some 0 bits on the right side, all of them become 1. The result of the bitwise AND operation on the original number and the minus 1 result is identical to the result gotten by modifying the rightmost 1 to 0. Take the binary number 1100 as an example again. Its result is 1011 when you apply minus 1. The result of the bitwise AND operation on 1100 and 1011 is 1000. If we change the rightmost 1 bit in the number 1100, it also becomes 1000 (Figure 4-6).

```
      1 1 0 0
(–)       1
-------------
      1 0 1 1
(&)   1 1 0 0
-------------
      1 0 0 0
```

Figure 4-6. *If we take 1100 minus 1 and then take the bitwise AND operation of the minus result 1011 and the original number 1100, the final result 1000 is the same as changing the rightmost 1 bit of the number 1100 to 0. (Numbers are in the binary representation.)*

The analysis can be summarized as follows: if we first apply minus 1 to a number and apply the bitwise AND operation to the original number and the minus result, the rightmost 1 bit becomes 0. We repeat these operations until the number becomes 0. We can develop the code in Listing 4-22 accordingly.

Listing 4-22. *C Code for Number of 1 in Binary (Version 3)*

```
int NumberOf1(int n) {

 int count = 0;


 while (n) {

     ++ count;

     n = (n - 1) & n;

 }


 return count;

}
```

The number of times in the `while` loops equals to the number of 1 in the binary format of input `n`.

Source Code:

035_NumberOf1.c

Test Cases:

- Positive integers (including the maximum integer as a boundary test case)
- Zero
- Negative integers (including the minimum integer as a boundary test case)

Question 36 Please check whether a number is a power product of 2 in only one C statement.

When a number is 2 to the power of k ($k \geq 0$), there is only one 1 bit in its binary representation. As discussed above, a statement (n-1)&n removes the only 1 bit in the number if n is 2^k. Therefore, the Boolean value of the statement `n != 0 && (n - 1) & n == 0` indicates whether n is 2^k.

Question 37 Given two integers, m and n, please calculate the number of bits in binary that need to be modified to change m to n. For example, the number 10 is 1010 in binary, and 13 is 1101 in binary. We can modify 3 bits of 1010 to get 1101 in binary.

We can modify three bits of 1010 to get 1101 in binary because their last three bits are different. According to the definition of the bitwise XOR operation, it gets 1 when two input bits are different and gets 0 when they are same. Therefore, the result of XOR indicates the bit difference between m and n. For example, the XOR result of 1010 and 1101 is 0111. Three bits are different between these two numbers, and there are three 1 bits in their XOR result.

After we get the XOR result, the remaining task is to count 1s in a number that has been discussed already, as shown in Listing 4-23.

Listing 4-23. *C Code to Modify a Number to Another*

```
int bitsToModify(int number1, int number2) {
    int temp = number1 ^ number2;

    // the number of 1 bits in temp
    int bits = 0;
    while(temp != 0) {
        ++bits;
        temp = (temp - 1) & temp;
    }

    return bits;
}
```

Source Code:

037_ModifyANumberToAnother.c

Test Cases:

- One or two numbers are positive, zero, or negative

Numbers Occurring Only Once

Question 38 Let's assume all numbers except two occur twice in an array. How do you get those two numbers to occur only once in O(n) time and O(1) space?
For example, only two numbers, 4 and 6, in the array {2, 4, 3, 6, 3, 2, 5, 5} occur once, and the others numbers occur twice. Therefore, the output should be 4 and 6.

This is a very difficult interview question. When an interviewer notices that a candidate does not have any ideas after a few minutes, it is possible for the interviewer to modify the question a little bit: If all numbers except one occur twice in the array, how do you find the only number occurring only once?

Why does the interviewer emphasize the times a number occurs (once or twice)? It reminds us of the characteristic of the bitwise XOR operation: it gets 1 when two input bits are different and gets 0 when they are same. Therefore, XOR gets 0 on a pair of duplicated numbers. If numbers in an array where all numbers except one occur twice are XORed, the result is exactly the number occurring only once.

With the simplified problem solved, let's return back to the original one. We know how to solve it if we could partition the array into two sub-arrays, where only one number appears once and the others appear twice. The only problem is how to partition the array into two.

Supposing the two numbers occurring once in the array are *num1* and *num2*. If we take XOR operations on all numbers in the array, the result should be the same as the result of *num1^num2* because XOR gets 0 on pairs of duplicated numbers. The result of *num1^num2* is not 0 because *num1* and *num2* are two different numbers. There is a 1 bit at least in the XOR result.

Let's denote the first 1 bit in the XOR result as the i^{th} bit. Numbers in the array are partitioned according to their i^{th} bits. The i^{th} bits of all numbers in the first

sub-array are 1s, and the i^{th} bits of all numbers in the second sub-array are 0s. Two duplicated numbers in a pair are in either the first or the second sub-array, but *num1* and *num2* cannot be in the same sub-array.

Take the sample array {2, 4, 3, 6, 3, 2, 5, 5} as an example. The XOR result of all numbers in the array is 0010 in binary, of which the second bit from the right is 1. Numbers in the array are partitioned into two sub-arrays: the first one is {2, 3, 6, 3, 2} where the numbers all have the property that the second bit from the right is 1, and the second one is {4, 5, 5}, which are numbers where the second bit from the right is 0. If we use XOR operations on these two sub-arrays, we can get the two numbers 6 and 4 occurring only once.

It is time to write code after we have confirmed our ideas about how to partition an array, as shown in 4-24.

Listing 4-24. *Java Code to Get Numbers Occurring Only Once*

```
   void getOnce(int numbers[], NumbersOccurringOnce once){
    if (numbers.length < 2)
        return;

    int resultExclusiveOR = 0;
    for (int i = 0; i < numbers.length; ++ i)
        resultExclusiveOR ^= numbers[i];

    int indexOf1 = findFirstBitIs1(resultExclusiveOR);

    once.num1 = once.num2 = 0;
    for (int j = 0; j < numbers.length; ++ j) {
        if(isBit1(numbers[j], indexOf1))
            once.num1 ^= numbers[j];
        else
            once.num2 ^= numbers[j];
    }
}

// The first 1 bit from the rightmost
int findFirstBitIs1(int num){
    int indexBit = 0;
    while (((num & 1) == 0) && (indexBit < 32)) {
        num = num >> 1;
        ++ indexBit;
    }

    return indexBit;
}

// check whether the bit with index indexBit is 1
boolean isBit1(int num, int indexBit) {
    num = num >> indexBit;
    return (num & 1) == 1;
}

   class NumbersOccurringOnce {
    public int num1;
    public int num2;
}
```

Source Code:

```
038_NumbersOccuringOnce.java
```

Test Cases:

- Functional cases: Numbers in a array appear twice/four times except two numbers occurring once
- Boundary cases: An array only has two unique numbers

Question 39 Two numbers out of *n* numbers from 1 to *n* are missing. The remaining *n*-2 numbers are restored in an array, not in any particular order. Please write a method (or a function) to find the missing two numbers.

Based on Arithmetic Calculation

Supposing the two missing numbers are *num1* and *num2*. It is easy to get the sum when adding all numbers in the array denoted as S_1 as well as the result of 1+2+…+*n* denoted as S_2. Let's denote S_2-S_1 as *s*. It is also easy to get the product when multiplying all numbers in the array denoted as P_1 as well as the result of 1×2×…×*n* denoted as P_2. Similarly, we denote P_2/P_1 as *p*.

Now we have the following pair of linear equations:

$$\begin{cases} num1 + num2 = s \\ num1 \times num2 = p \end{cases}$$

After solving this linear equations, we get *num1*= $\frac{s+\sqrt{s^2-4p}}{2}$ and *num2*= $\frac{s-\sqrt{s^2-4p}}{2}$. Therefore, this solution can be implemented with the code shown in Listing 4-25.

Listing 4-25. *Java Code to Get Two Missing Numbers (Version 1)*

```
void findMissing_solution1(int numbers[], NumbersOccurringOnce missing){
 int sum1 = 0;
 int product1 = 1;
 for(int i = 0; i < numbers.length; ++i){
     sum1 += numbers[i];
     product1 *= numbers[i];
 }

 int sum2 = 0;
    int product2 = 1;
 for(int i = 1; i <= numbers.length + 2; ++i){
     sum2 += i;
     product2 *= i;
 }

 int s = sum2 - sum1;
 int p = product2 / product1;

 missing.num1 = (s + (int)(Math.sqrt(s * s - 4 * p))) / 2;
 missing.num2 = s - missing.num1;
}
```

This solutions works when the range of numbers from 1 to *n* is relatively narrow. However, it causes overflow errors to calculate 1+2+…+*n* and 1×2×…×*n* when *n* is large. Let's look for a more robust solution.

Based on Bit Operations

There are *n*-2 numbers in an array in the range from 1 to *n* missing two numbers *num1* and *num2*. We define another array with size 2*n*-2, of which the first *n*-2 numbers are copies of numbers in the original array and others are numbers from 1 to *n*. All numbers except two numbers, *num1* and *num2*, occur twice in the new array, and *num1* and *num2* occur only once. Therefore, this problem is equivalent to the preceding problem: find two numbers occurring once in an array where others occur twice. Let's solve it based on bit operations with the method `getOnce`, borrowed from the solution of the preceding problem, as shown in Listing 4-26.

Listing 4-26. *Java Code to Get Two Missing Numbers (Version 2)*

```
void findMissing_solution2(int numbers[], NumbersOccurringOnce missing){
 int originalLength = numbers.length;
 int extendedLength = originalLength * 2 + 2;
 int extention[] = new int[extendedLength];

 for(int i = 0; i < originalLength; ++i)
```

```
            extention[i] = numbers[i];

        for(int i = originalLength; i < extendedLength; ++i)

            extention[i] = i - originalLength;



        getOnce(extention, missing);

    }
```

Source Code:

```
039_FindTwoMissingNumbers.java
```

Test Cases:

- Functional tests: Two numbers out of *n* numbers from 1 to *n* are missing in an array
- Boundary tests: An array only has one or two numbers
- Robust tests: The maximum number *n* is very big