高级算法扩展(二) -- Bit Manipulation

136. Single Number
137. Single Number II
260. Single Number III
191. Number of 1 Bits
268. Missing Number
371. Sum of Two Integers
338. Counting Bits
231. Power of Two
342. Power of Four

你们的好朋友Eddie

Bit Manipulation(位运算)：

- 与 `&`
- 或 `|`
- 异或 `^`
- 左移 `<<`
- 右移 `>>`
    - 正数右移，高位用 0 补，负数右移，高位用 1 补
- 无符号右移 `>>>`
    - 当负数使用无符号右移时，用 0 进行补位
- 取非 `~`
    - 一元操作符

异或 `^` 运算的小技巧

> Use ^ to remove even exactly same numbers and save the odd, or save the distinct bits and remove the same.
>
> 去除出现偶数次的数字，保留出现奇数次的数字。

XOR 的一些技巧：

1. A ^ B = B ^ A
2. A ^ (B ^ C) = (A ^ B) ^ C
3. A ^ 0 = A
4. A ^ A = 0
5. 快速判断两个值是否相等： a ^ b == 0?
6. 位数翻转
   0 ^ 1 = 1
   1 ^ 1 = 0
   例如：
   翻转10100001的第6位：
   可以将该数与00100000(1 << 5)进行按位异或运算
   10100001 ^ 00100000 = 10000001
7. 我们使用异或来判断一个二进制数中1的数量是奇数还是偶数
   求10100001中1的数量是奇数还是偶数：
   1 ^ 0 ^ 1 ^ 0 ^ 0 ^ 0 ^ 0 ^ 1 = 1
   结果为1就是奇数个'1'，结果为0就是偶数个'1'
8. 不占用额外空间进行 swap:
   a = a ^ b;
   b = a ^ b;   //a ^ b ^ b = a ^ (b ^ b) = a ^ 0 = a;
   a = a ^ b;   //a ^ b ^ a = (a ^ a) ^ b = 0 ^ b = b;
9. num & (num - 1)   // 将 num（二进制） 中最右边的 '1' 变成 '0'.
10. num & ^(num - 1)   // 提取 num（二进制） 中最右边的那一位'1'，并将其他位变成 '0'.

XOR的性质：

XOR Truth Table		
Input A	**Input B**	**Output**
0	0	0
0	1	1
1	0	1
1	1	0

136. Single Number

Easy 5116 176 Add to List Share

Given a **non-empty** array of integers `nums`, every element appears *twice* except for one. Find that single one.

Follow up: Could you implement a solution with a linear runtime complexity and without using extra memory?

Example 1:

```
Input: nums = [2,2,1]
Output: 1
```

Example 2:

```
Input: nums = [4,1,2,1,2]
Output: 4
```

Example 3:

```
Input: nums = [1]
Output: 1
```

- If we take XOR of zero and some bit, it will return that bit
  - $a \oplus 0 = a$
- If we take XOR of two same bits, it will return 0
  - $a \oplus a = 0$
- $a \oplus b \oplus a = (a \oplus a) \oplus b = 0 \oplus b = b$

```java
public class Solution {
    //相同数字异或为0，0与a异或还是等于a本身
    public int singleNumber(int[] nums) {
        int res = 0;
        for (int num : nums) res ^= num;
        return res;
    }
}
```

137. Single Number II

Medium 2085 377 Add to List Share

Given a **non-empty** array of integers, every element appears *three* times except for one, which appears exactly once. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Example 1:

```
Input: [2,2,3,2]
Output: 3
```

Example 2:

```
Input: [0,1,0,1,0,1,99]
Output: 99
```

```java
    //HashMap space O(n) n is the distinct number in array
    public int singleNumber(int[] nums) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int num : nums) map.put(num, map.getOrDefault(num, 0) + 1);
        for (int k : map.keySet()) if (map.get(k) == 1) return k;
        return -1;
    }

    //bit mask space O(1)
    public int singleNumber(int[] nums) {
        int res = 0;
        int[] map = new int[32];
        //save to the map
        for (int i = 0; i < nums.length; i++) saveIntoMap(nums[i], map);
        //remove number appear 3 times
        for (int i = 0; i < 32; i++) map[i] = map[i] % 3;
        //the remaining is only one number
        for (int i = 0; i < 32; i++) res |= map[i] << i;
        return res;
    }

    private void saveIntoMap(int num, int[] map) {
        for (int i = 0; i < 32; i++)
            map[i] += (num >> i) & 1;
    }
```

260. Single Number III

Medium 1900 119 Add to List Share

Given an integer array `nums`, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once. You can return the answer in **any order**.

Follow up: Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

Example 1:

```
Input: nums = [1,2,1,3,2,5]
Output: [3,5]
Explanation:  [5, 3] is also a valid answer.
```

Example 2:

```
Input: nums = [-1,0]
Output: [-1,0]
```

Example 3:

```
Input: nums = [0,1]
Output: [1,0]
```

隐藏条件留下的2个数字互不相同

其他所有数字出现2次，其他2个不同的数字只出现一次

原码　反码　补码（了解）

Binary code, Inverse code, Complement code,

最前是一位机器码:
正数是0，负数是1

原码:

	正数
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

	负数
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

“原码”储存的方式，方便了看的人类，却苦了计算机

1	0001

-1	1001

原码有两个问题:

A. 0不统一

有一个（+0），有一个（-0）

B. 破坏了加法

我们希望 （+1）和（-1）相加是0，但计算机只能算出0001+1001=1010 (-2)

为了解决 +k + (-k) = 0 问题，在“原码”的基础上，人们发明了“反码”

“反码”只针对负数：符号位置不变，其余位置取反

反码

	正数
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

	负数
-0	1111
-1	1110
-2	1101
-3	1100
-4	1011
-5	1010
-6	1001
-7	1000

原码

	负数
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

当“原码”变成“反码”时，完美的解决了“正负相加应该等于0”的问题

比如-1(1110)和1(0001)相加 1110(-1) +0001(1) =1111(-0)

进一步地

我们希望只有一个0，所以发明了"补码"，同样是针对"负数"做处理的

"补码"的意思是，从原来"反码"的基础上，补一个1，即（+1）

	正数
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

补码

	负数
~~0~~	~~0000~~
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

反码

	负数
-0	1111
-1	1110
-2	1101
-3	1100
-4	1011
-5	1010
-6	1001
-7	1000

原码

	负数
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

有得必有失，在补一位1的时候，要丢掉最高位

我们要处理"反码"中的"-0",当1111再补上一个1之后，变成了10000，丢掉最高位就是0000，刚好和左边正数的0，完美融合掉了

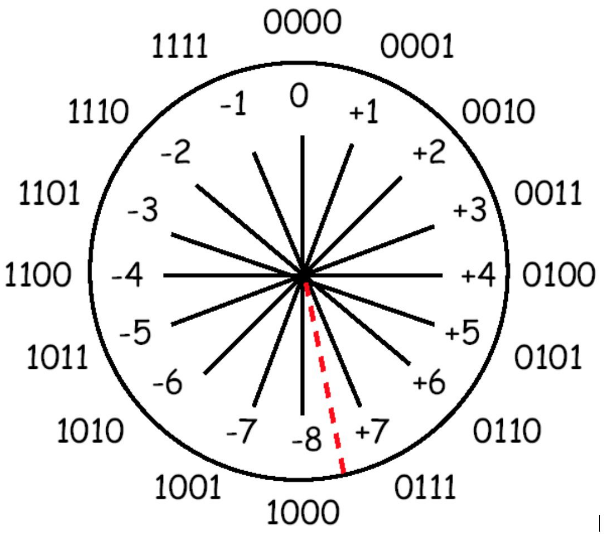
这样就解决了+0和-0同时存在的问题

另外"正负数相加等于0"的问题，同样得到满足

◆ 机器数的形式：原码、反码、**补码**。

整数	原码	反码	补码
1	0000 0001	0000 0001	0000 0001
-1	1000 0001	1111 1110	1111 1111
5	0000 0101	0000 0101	0000 0101
-5	1000 0101	1111 1010	1111 1011

◆ 补码计算方式：正数补码=反码=原码，负数补码=反码+1。

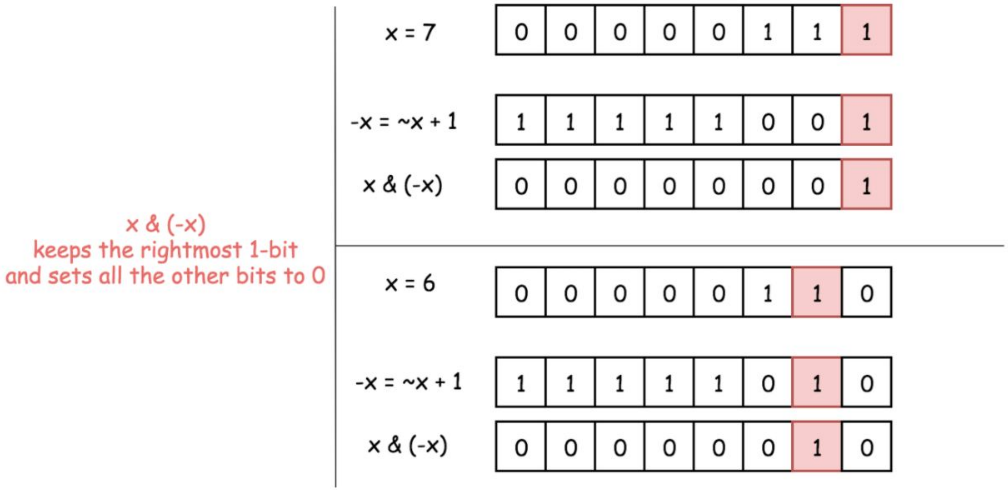


num & (num - 1)

num = ________100...00
num - 1 = ________011...11

num & (num - 1) = ________000...00

// 将 num 中最右边的 1 变成 0.



num & ~(num - 1);

num = ________100...00
num - 1 = ________011...11

~(num - 1) = =======100...00
num = ________100...00

num & ~(num - 1) = 00000100...00

// 提取 num 中最右边的那一位1，并将其他位变成 0.

```java
public class Solution {
    public int[] singleNumber(int[] nums) {
        int xor = 0; //这里只留下的2个不同数字的异或
        for (int i = 0; i < nums.length; i++) xor ^= nums[i];
        //这里是找出了两个不同数字在某一位的不同，根据这一位可以讲二者分开来
        int lastDigit = xor & (-xor);              //直接用补码
        // int lastDigit = xor & (~xor + 1);      //补码=反码+1
        int group1 = 0, group2 = 0;
        for (int i = 0; i < nums.length; i++) {
            if ((lastDigit & nums[i]) == 0) group1 ^= nums[i];
            else group2 ^= nums[i];
        }
        return new int[]{group1, group2};
    }
}
```

371. Sum of Two Integers

Medium 1414 2425 Add to List Share

Calculate the sum of two integers *a* and *b*, but you are **not allowed** to use the operator `+` and `-`.

Example 1:

```
Input: a = 1, b = 2
Output: 3
```

Example 2:

```
Input: a = -2, b = 3
Output: 1
```

```java
public class Solution {
    // 011  3
    //+101  5
    //1000  8
    public int getSum(int a, int b) {
        while (b != 0) {
            int carry = a & b;  // 101 & 011 = 001        //把a，b相同位置的1保留下来因为我们要处理进位      二周目我们进位后和上次不进位的计算
            a = a ^ b;          // a = 101 ^ 011 = 110    //把a，b不同位置的1保留下来因为不进位，我们直接加法计算    不进位直接加
            b = carry << 1;     // b = 00 << 1 = 0        //相同位置有1证明需要进位，我们进位c一位                 直到我们没有进位的部分
        }
        return a;          // a = 11 = 3
    }

    // Recursive
    public int getSum(int a, int b) {
        return b == 0 ? a : getSum(a ^ b, (a & b) << 1);
    }
```

268. Missing Number

Easy 2177 2341 Add to List Share

Given an array `nums` containing `n` distinct numbers in the range `[0, n]`, return *the only number in the range that is missing from the array.*

Follow up: Could you implement a solution using only `O(1)` extra space complexity and `O(n)` runtime complexity?

Example 1:

```
Input: nums = [3,0,1]
Output: 2
Explanation: n = 3 since there are 3 numbers, so all numbers are in the range
[0,3]. 2 is the missing number in the range since it does not appear in nums.
```

Example 2:

```
Input: nums = [0,1]
Output: 2
Explanation: n = 2 since there are 2 numbers, so all numbers are in th
[0,2]. 2 is the missing number in the range since it does not appear i
```

```java
public class Solution {
    public int missingNumber(int[] nums) {
        int miss = 0;
        for (int num : nums) miss ^= num;
        for (int i = 1; i <= nums.length; i++) miss ^= i;
        return miss;
    }

    public int missingNumber(int[] nums) {
        int expectedSum = nums.length * (nums.length + 1) / 2;
        int actualSum = 0;
        for (int num : nums) actualSum += num;
        return expectedSum - actualSum;
    }
```

191. Number of 1 Bits

Easy 1052 558 Add to List Share

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

Note:

- Note that in some languages such as Java, there is no unsigned integer type. In this case, the input will be given as a signed integer type. It should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using 2's complement notation. Therefore, in **Example 3** above, the input represents the signed integer. `-3`.

Follow up: If this function is called many times, how would you optimize it?

Example 1:

```
Input: n = 00000000000000000000000000001011
Output: 3
Explanation: The input binary string 00000000000000000000000000001011 has a total
of three '1' bits.
```

Example 2:

```
Input: n = 00000000000000000000000010000000
Output: 1
Explanation: The input binary string 00000000000000000000000010000000 has a total
of one '1' bit.
```

Example 3:

```
Input: n = 11111111111111111111111111111101
Output: 31
Explanation: The input binary string 11111111111111111111111111111101 has a total
of thirty one '1' bits.
```

```java
public class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int count = 0, mask = 1;
        for (int i = 0; i < 32; i++) {
            if ((n & mask) != 0) count++;
            mask <<= 1;
        }
        return count;
    }

    public int hammingWeight(int n) {
        int count = 0;
        while (n != 0) {
            count++;
            n &= (n - 1); //最右边一位1置为0
        }
        return count;
    }

    public int hammingWeight(int n) {
        return Integer.bitCount(n);
    }
}
```

338. Counting Bits

Medium 3170 180 Add to List Share

Given a non negative integer number **num**. For every numbers **i** in the range **0 ≤ i ≤ num** calculate the number of 1's in their binary representation and return them as an array.

Example 1:

```
Input: 2
Output: [0,1,1]
```

Example 2:

```
Input: 5
Output: [0,1,1,2,1,2]
```

```java
//P(x) = P(x / 2) + (x \mod 2) P(x)=P(x/2)+(xmod2)


public class Solution {
    //也是最右边位不管01都去掉，找之前计算出的数据
    public int[] countBits(int num) {
        int[] res = new int[num + 1];
        for (int i = 1; i <= num; ++i)
            res[i] = res[i / 2] + (i % 2); // x / 2 is x >> 1 and x % 2 is x & 1
        return res;
    }
```

Follow up:

- It is very easy to come up with a solution with run time **O(n*sizeof(integer))**. But can you do it in linear time **O(n)** /possibly in a single pass?
- Space complexity should be **O(n)**.
- Can you do it like a boss? Do it without using any builtin function like **__builtin_popcount** in c++ or in any other language.

```java
    //这个方法把从右边数第一次出现的1变成了0，这样就找到了之前已经计算出来的状态。然后我们比这个状态多一个1，就是我们之前抹去的最右边最高位的1
    public int[] countBits(int num) {
        int[] ans = new int[num + 1];
        for (int i = 1; i <= num; ++i)
          ans[i] = ans[i & (i - 1)] + 1;
        return ans;
    }
```

231. Power of Two

Easy 1077 205 Add to List Share

Given an integer `n`, write a function to determine if it is a power of two.

Example 1:

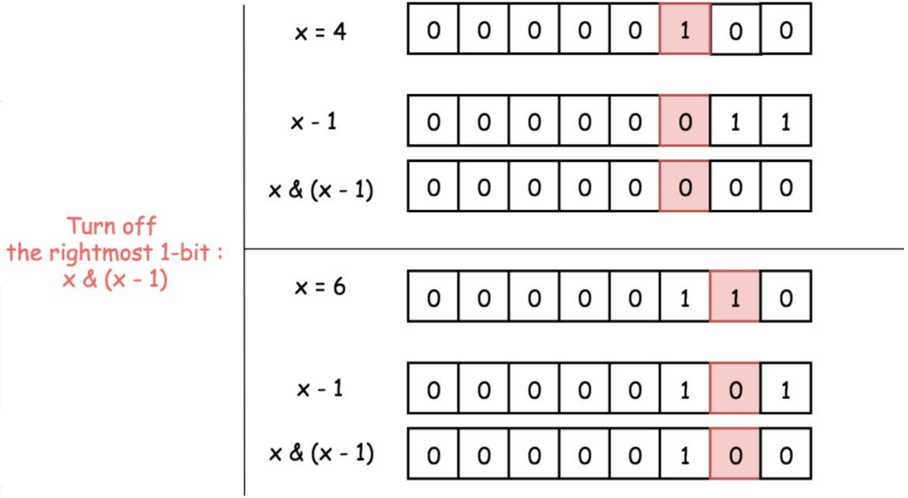
```
Input: n = 1
Output: true
Explanation: 2^0 = 1
```

Example 2:

```
Input: n = 16
Output: true
Explanation: 2^4 = 16
```

Example 3:

```
Input: n = 3
Output: false
```




```java
class Solution {

    public boolean isPowerOfTwo(int n) {
        return n > 0 && (n & (n - 1)) == 0;
    }

    public boolean isPowerOfTwo(int n) {
        if (n <= 0) return false;
        while (n % 2 == 0) n /= 2;
        return n == 1;
    }
}
```

342. Power of Four

Easy 732 243 Add to List Share

Given an integer (signed 32 bits), write a function to check whether it is a power of 4.

Example 1:

```
Input: 16
Output: true
```

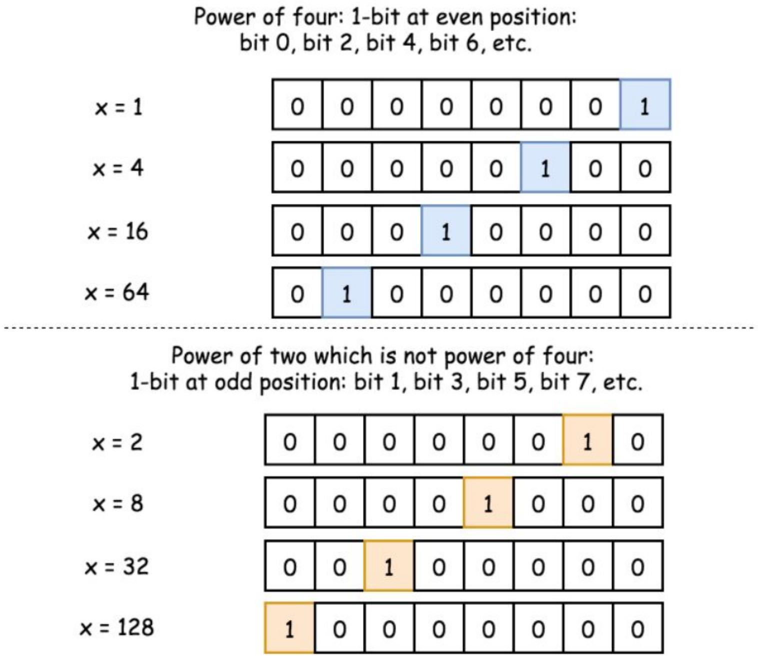
Example 2:

```
Input: 5
Output: false
```

Hence power of four would make a zero in a bitwise AND with number $(101010...10)_2$:

$$4^a \wedge (101010...10)_2 == 0$$

> How long should be $(101010...10)_2$ if x is a signed integer? 32 bits. To write shorter, in 8 charaters instead of 32, it's common to use hexadecimal representation: $(101010...10)_2 = (aaaaaaaa)_{16}$.




```java
public class Solution {
    public boolean isPowerOfFour(int num) {
        // return (Math.log(num) / Math.log(4)) % 1 == 0;
        return (num > 0) && ((num & (num - 1)) == 0) && ((num & 0xaaaaaaaa) == 0);
    }
}
```

Summany

1. 计算机有一套机制用二进制表示（正/负）整数/小数
2. 平时我们写代码不用刻意写 ‘<<’ 或者 ‘>>’ 等移位运算符, 因为编译器会自动做优化。
   如果要写, 建议加括号。
3. 熟记真值表。
4. XOR 有很多性质，最重要的是：定义本身和交换律、结合律。
   其他性质可从定义和交换律、结合律推导出来。
5. XOR 的性质产生了很多等式, 这些等式有时候可以写成状态转移方程,
   可以把原问题转换成dp问题。