

宫水三叶的刷题日记

# 位运算

Author : 宫水三叶

Date : 2021/10/07

QQ Group: 703311589

WeChat : oaoaya

刷题日记

公众号: 宫水三叶的刷题日记

噔噔噔噔，这是公众号「[宫水三叶的刷题日记](#)」的原创专题「位运算」合集。

本合集更新时间为 2021-10-07，大概每 2-4 周会集中更新一次。关注公众号，后台回复「位运算」即可获取最新下载链接。

💡下面介绍使用本合集的最佳使用实践：

## 学习算法：

1. 打开在线目录（[Github 版](#) & [Gitee 版](#)）；
2. 从侧边栏的类别目录找到「位运算」；
3. 按照「推荐指数」从大到小进行刷题，「推荐指数」相同，则按照「难度」从易到难进行刷题；
4. 拿到题号之后，回到本合集进行检索。

## 维持熟练度：

1. 按照本合集「从上往下」进行刷题。

学习过程中遇到任何困难，欢迎加入「每日一题打卡 QQ 群：703311589」进行交流   

## 题目描述

这是 LeetCode 上的 [137. 只出现一次的数字 II](#)，难度为 中等。

Tag：「哈希表」、「位运算」

给你一个整数数组 `nums`，除某个元素仅出现一次外，其余每个元素都恰出现三次。请你找出并返回那个只出现了一次的元素。

示例 1：

输入：`nums = [2,2,3,2]`

输出：3

宫水三叶  
の  
刷题日记

公众号：宫水三叶的刷题日记

示例 2：

输入：nums = [0,1,0,1,0,1,99]

输出：99

提示：

- $1 \leq \text{nums.length} \leq 3 \times 10^4$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
- nums 中，除某个元素仅出现 一次 外，其余每个元素都恰出现 三次

进阶：你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

## 哈希表

一个朴素的做法是使用「哈希表」进行计数，然后将计数为 1 的数字进行输出。

哈希表以「数值：数值出现次数」形式进行存储。

代码：

```
class Solution {
    public int singleNumber(int[] nums) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int x : nums) {
            map.put(x, map.getOrDefault(x, 0) + 1);
        }
        for (int x : map.keySet()) {
            if (map.get(x) == 1) return x;
        }
        return -1;
    }
}
```

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

宫水三叶  
刷题日记

公众号：宫水三叶的刷题日记

## 位数统计

哈希表解法的空间复杂度是  $O(n)$  的，而题目的【进阶】部分提到应当使用常数空间来做。

其中一个比较容易想到的做法，是利用 `int` 类型固定为 32 位。

使用一个长度为 32 的数组 `cnt` 记录下所有数值的每一位共出现了多少次 1，再对 `cnt` 数组的每一位进行  $\text{mod } 3$  操作，重新拼凑出只出现一次的数值。

举个🌰，考虑样例 `[1,1,1,3]`，1 和 3 对应的二进制表示分别是 `00..001` 和 `00..011`，存入 `cnt` 数组后得到 `[0,0,...,0,1,4]`。进行  $\text{mod } 3$  操作后得到 `[0,0,...,0,1,1]`，再转为十进制数字即可得「只出现一次」的答案 3。

代码：

```
class Solution {
    public int singleNumber(int[] nums) {
        int[] cnt = new int[32];
        for (int x : nums) {
            for (int i = 0; i < 32; i++) {
                if (((x >> i) & 1) == 1) {
                    cnt[i]++;
                }
            }
        }
        int ans = 0;
        for (int i = 0; i < 32; i++) {
            if ((cnt[i] % 3 & 1) == 1) {
                ans += (1 << i);
            }
        }
        return ans;
    }
}
```

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

宫水三叶

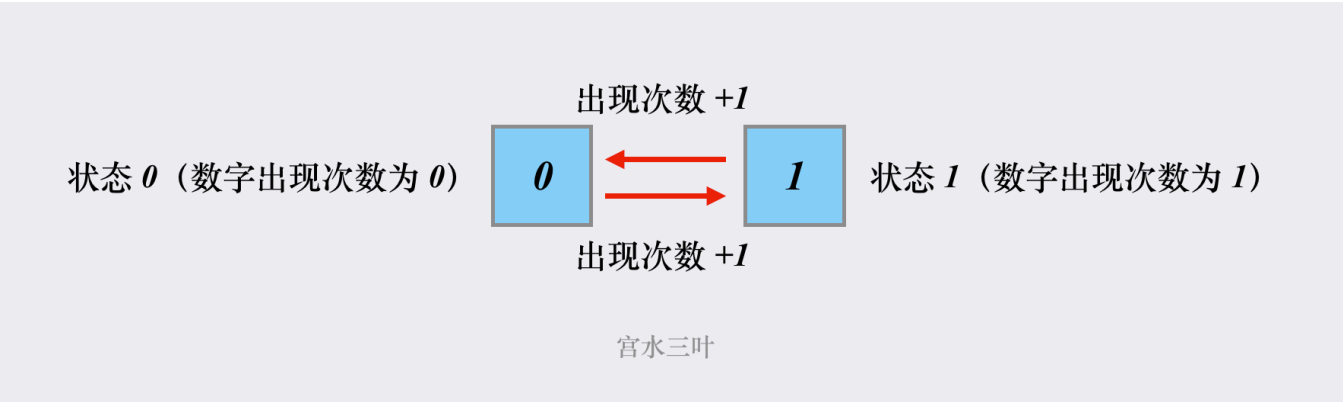
刷题日记

公众号：宫水三叶的刷题日记

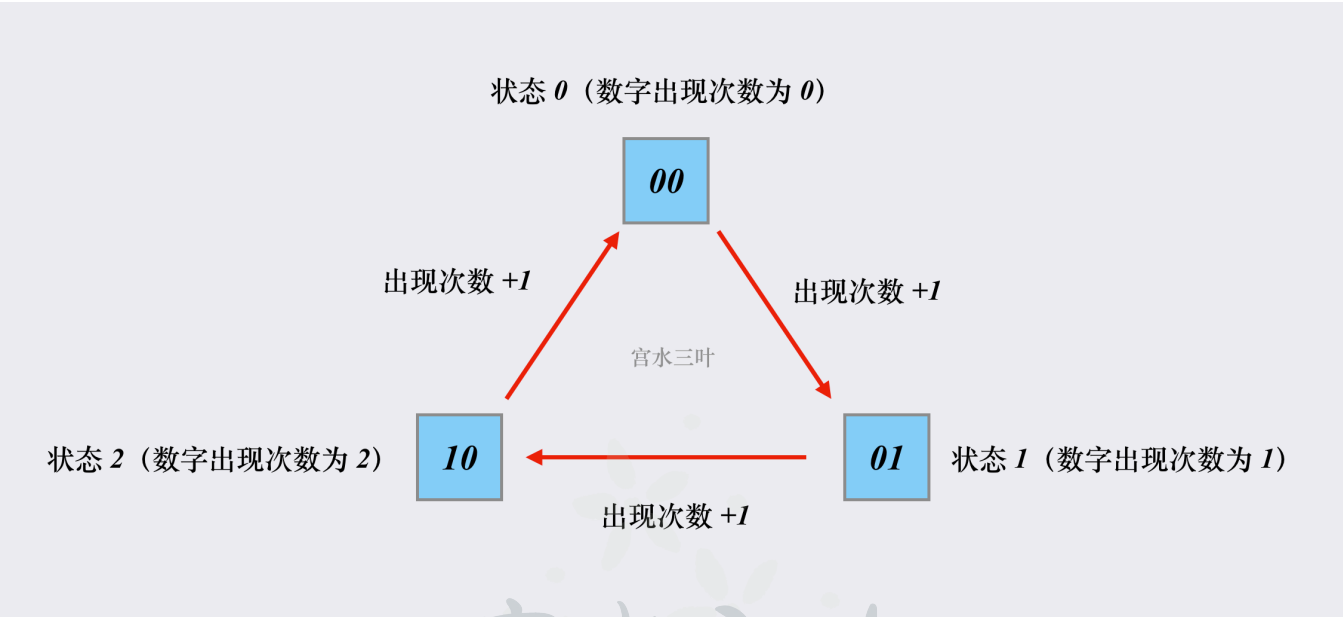
# DFA

如果我们考虑「除了某个元素只出现一次以外，其余每个元素均出现两次」的情况，那么可以使用「异或」运算。

利用相同数异或为 0 的性质，可以帮助我们很好实现状态切换：



本题是考虑「除了某个元素只出现一次以外，其余每个元素均出现三次」的情况，那么对应了「出现 0 次」、「出现 1 次」和「出现 2 次」三种状态，意味着至少需要两位进行记录，且状态转换关系为：



那么如何将上述 DFA 用表达式表示出来呢？有以下几种方法：

1. 用「真值表」写出「逻辑函数表达式」可参考 [这里](#)，化简过程可以参考 [卡诺图化简法](#)。
2. 把结论记住（这是一道经典的 DFA 入门题）。

3. 硬做，位运算也就那几种，不会「数字电路」也记不住「结论」，砸时间看着真值表不断调逻辑也是可以写出来的。

代码：

```
class Solution {
    public int singleNumber(int[] nums) {
        int one = 0, two = 0;
        for(int x : nums){
            one = one ^ x & ~two;
            two = two ^ x & ~one;
        }
        return one;
    }
}
```

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

更多精彩内容，欢迎关注：[公众号](#) / [Github](#) / [LeetCode](#) / [知乎](#) \*\*

## 题目描述

这是 LeetCode 上的 **190. 颠倒二进制位**，难度为 简单。

Tag：「位运算」、「模拟」

颠倒给定的 32 位无符号整数的二进制位。

提示：

- 请注意，在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。
- 在 Java 中，编译器使用二进制补码记法来表示有符号整数。因此，在上面的示例 2 中，输入表示有符号整数 -3，输出表示有符号整数 -1073741825。

进阶：

刷题日记

公众号: 宫水三叶的刷题日记

- 如果多次调用这个函数，你将如何优化你的算法？

示例 1：

输入：00000010100101000001111010011100

输出：00111001011110000010100101000000

解释：输入的**二进制串** 00000010100101000001111010011100 表示无符号整数 43261596，因此返回 964176192，其**二进制表示形式**为 00111001011110000010100101000000。

示例 2：

输入：1111111111111111111111111111101

输出：1011111111111111111111111111111

解释：输入的**二进制串** 1111111111111111111111111111101 表示无符号整数 4294967293，因此返回 3221225471 其**二进制表示形式**为 1011111111111111111111111111111。

提示：

- 输入是一个长度为 32 的二进制字符串

## 「对称位」构造

执行结果：通过 显示详情 >

执行用时：1 ms，在所有 Java 提交中击败了 100.00% 的用户

内存消耗：37.8 MB，在所有 Java 提交中击败了 98.44% 的用户

炫耀一下：



写题解，分享我的解题思路

一个简单的做法是对输入的  $n$  做诸位检查。

刷题日记

公众号：宫水三叶的刷题日记

如果某一位是 1 的话，则将答案相应的对称位置修改为 1。

代码：

```
public class Solution {
    public int reverseBits(int n) {
        int ans = 0;
        for (int i = 0; i < 32; i++) {
            int t = (n >> i) & 1;
            if (t == 1) {
                ans |= (1 << (31 - i));
            }
        }
        return ans;
    }
}
```

- 时间复杂度： $int$  固定 32 位，循环次数不随输入样本发生改变。复杂度为  $O(1)$
- 空间复杂度： $O(1)$

## 「逐位分离」构造

执行结果： **通过** [显示详情 >](#)

执行用时： **1 ms**，在所有 Java 提交中击败了 **100.00%** 的用户

内存消耗： **38.2 MB**，在所有 Java 提交中击败了 **37.55%** 的用户

炫耀一下：



[写题解，分享我的解题思路](#)

另外一种做法是，每次都使用  $n$  的最低一位，使用  $n$  的最低一位去更新答案的最低一位，使用完将  $n$  进行右移一位，将答案左移一位。

相当于每次都  $n$  的最低一位更新成  $ans$  的最低一位。



代码：

```
public class Solution {  
    public int reverseBits(int n) {  
        int ans = 0;  
        int cnt = 32;  
        while (cnt-- > 0) {  
            ans <<= 1;  
            ans += (n & 1);  
            n >>= 1;  
        }  
        return ans;  
    }  
}
```

- 时间复杂度： $int$  固定 32 位，循环次数不随输入样本发生改变。复杂度为  $O(1)$
- 空间复杂度： $O(1)$

## 「分组互换」

执行结果： **通过** [显示详情 >](#)

执行用时： **1 ms**，在所有 Java 提交中击败了 **100.00%** 的用户

内存消耗： **38 MB**，在所有 Java 提交中击败了 **91.43%** 的用户

炫耀一下：



[写题解，分享我的解题思路](#)

事实上，可以对于长度固定的  $int$  类型，我们可以使用「分组构造」的方式进行。

两位互换 -> 四位互换 -> 八位互换 -> 十六位互换。

代码：

刷题日记

公众号：宫水三叶的刷题日记

```

public class Solution {
    public int reverseBits(int n) {
        n = ((n & 0xAAAAAAAA) >>> 1) | ((n & 0x55555555) <<< 1);
        n = ((n & 0xCCCCCCCC) >>> 2) | ((n & 0x33333333) <<< 2);
        n = ((n & 0xF0F0F0F0) >>> 4) | ((n & 0x0F0F0F0F) <<< 4);
        n = ((n & 0xFF00FF00) >>> 8) | ((n & 0x00FF00FF) <<< 8);
        n = ((n & 0xFFFF0000) >>> 16) | ((n & 0x0000FFFF) <<< 16);
        return n;
    }
}

```

- 时间复杂度：如何进行互换操作取决于 *int* 长度。复杂度为  $O(1)$
- 空间复杂度： $O(1)$

**PS.** 类似的做法我在 [191. 位1的个数](#) 也介绍过。如果大家学有余力的话，建议大家在纸上模拟一下这个过程。如果不想深入，也可以当成模板背过（写法非常固定）。

但请不要认为「方法三」一定就比「方法一」等直接采用循环的方式更快。此类做法的最大作用，不是处理 *int*，而是处理更大位数的情况，在长度只有 32 位的 *int* 的情况下，该做法不一定就比循环要快（该做法会产生多个的中间结果，导致赋值发生多次，而且由于指令之间存在对 *n* 数值依赖，可能不会被优化为并行指令），这个道理和对于排序元素少的情况下，我们会选择「冒泡排序」而不是「归并排序」是一样的，因为「冒泡排序」常数更小。

更多精彩内容，欢迎关注：[公众号](#) / [Github](#) / [LeetCode](#) / [知乎](#)

## 题目描述

这是 LeetCode 上的 [191. 位1的个数](#)，难度为 简单。

Tag：「位运算」

编写一个函数，输入是一个无符号整数（以二进制串的形式），返回其二进制表达式中数字位数为 '1' 的个数（也被称为汉明重量）。

提示：

- 请注意，在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。

- 在 Java 中，编译器使用二进制补码记法来表示有符号整数。因此，在上面的 示例 3 中，输入表示有符号整数 -3。

#### 示例 1：

输入：00000000000000000000000000001011

输出：3

解释：输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。

#### 示例 2：

输入：00000000000000000000000010000000

输出：1

解释：输入的二进制串 00000000000000000000000010000000 中，共有一位为 '1'。

#### 示例 3：

输入：111111111111111111111111111101

输出：31

解释：输入的二进制串 111111111111111111111111111101 中，共有 31 位为 '1'。

#### 提示：

- 输入必须是长度为 32 的二进制串。

#### 进阶：

- 如果多次调用这个函数，你将如何优化你的算法？

---

## 「位数检查」解法

一个朴素的做法是，对 `int` 的每一位进行检查，并统计 1 的个数。

宫水三叶  
の  
刷题日记

公众号：宫水三叶的刷题日记

执行结果： **通过** [显示详情 >](#)

执行用时： **1 ms** ，在所有 Java 提交中击败了 **95.76%** 的用户

内存消耗： **35.1 MB** ，在所有 Java 提交中击败了 **97.46%** 的用户

炫耀一下：



[写题解，分享我的解题思路](#)

代码：

```
public class Solution {  
    public int hammingWeight(int n) {  
        int ans = 0;  
        for (int i = 0; i < 32; i++) {  
            ans += ((n >> i) & 1);  
        }  
        return ans;  
    }  
}
```

- 时间复杂度： $O(k)$ ， $k$  为 `int` 的位数，固定为 32 位
- 空间复杂度： $O(1)$

## 「右移统计」解法

对于方法一，即使  $n$  的高位均为是 0，我们也会对此进行循环检查。

因此另外一个做法是：通过 `n & 1` 来统计当前  $n$  的最低位是否为 1，同时每次直接对  $n$  进行右移并高位补 0。

当  $n = 0$  代表，我们已经将所有的 1 统计完成。

这样的做法，可以确保只会循环到最高位的 1。

执行结果： **通过** [显示详情 >](#)

执行用时： **0 ms** ，在所有 Java 提交中击败了 **100.00%** 的用户

内存消耗： **35.4 MB** ，在所有 Java 提交中击败了 **39.18%** 的用户

炫耀一下：



[写题解，分享我的解题思路](#)

代码：

```
public class Solution {
    public int hammingWeight(int n) {
        int ans = 0;
        while (n != 0) {
            ans += (n & 1);
            n >>= 1;
        }
        return ans;
    }
}
```

- 时间复杂度： $O(k)$ ， $k$  为 `int` 的位数，固定为 32 位，最坏情况  $n$  的二进制表示全是 1
- 空间复杂度： $O(1)$

## 「lowbit」解法

对于方法二，如果最高位 1 和 最低位 1 之间全是 0，我们仍然会诸次右移，直到处理到最高位的 1 为止。

那么是否有办法，只对位数为 1 的二进制位进行处理呢？

使用 `lowbit` 即可做到，`lowbit` 会在  $O(1)$  复杂度内返回二进制表示中最低位 1 所表示的数值。

例如  $(0000\dots111100)_2$  传入 `lowbit` 返回  $(0000\dots000100)_2$ ； $(0000\dots00011)_2$  传入 `lowbit` 返回  $(0000\dots00001)_2 \dots$

执行结果： 通过 [显示详情 >](#)

执行用时： **0 ms** ，在所有 Java 提交中击败了 **100.00%** 的用户

内存消耗： **35.4 MB** ，在所有 Java 提交中击败了 **39.18%** 的用户

炫耀一下：



[写题解，分享我的解题思路](#)

代码：

```
public class Solution {
    public int hammingWeight(int n) {
        int ans = 0;
        for (int i = n; i != 0; i -= lowbit(i)) ans++;
        return ans;
    }
    int lowbit(int x) {
        return x & -x;
    }
}
```

- 时间复杂度： $O(k)$ ， $k$  为 `int` 的位数，固定为 32 位，最坏情况  $n$  的二进制表示全是 1
- 空间复杂度： $O(1)$

## 「分组统计」解法

以上三种解法都是  $O(k)$  的，事实上我们可以通过分组统计的方式，做到比  $O(k)$  更低的复杂度。

执行结果： **通过** [显示详情 >](#)

执行用时： **0 ms** ，在所有 Java 提交中击败了 **100.00%** 的用户

内存消耗： **35.4 MB** ，在所有 Java 提交中击败了 **51.07%** 的用户

炫耀一下：



[写题解，分享我的解题思路](#)

代码：

```
public class Solution {
    public int hammingWeight(int n) {
        n = (n & 0x55555555) + ((n >> 1) & 0x55555555);
        n = (n & 0x33333333) + ((n >> 2) & 0x33333333);
        n = (n & 0x0f0f0f0f) + ((n >> 4) & 0x0f0f0f0f);
        n = (n & 0x00ff00ff) + ((n >> 8) & 0x00ff00ff);
        n = (n & 0x0000ffff) + ((n >> 16) & 0x0000ffff);
        return n;
    }
}
```

- 时间复杂度： $O(\log k)$ ， $k$  为 `int` 的位数，固定为 32 位
- 空间复杂度： $O(1)$

PS. 对于该解法，如果大家学有余力的话，还是建议大家在纸上模拟一下这个过程。如果不想深入，也可以当成模板背过（写法非常固定），但通常如果不是写底层框架，你几乎不会遇到需要一个  $O(\log k)$  解法的情况。

而且这个做法的最大作用，不是处理 `int`，而是处理更大位数的情况，在长度只有 32 位的

`int` 的情况下，该做法不一定就比循环要快（该做法会产生多个的中间结果，导致赋值发生多次，而且由于指令之间存在对  $n$  数值依赖，可能不会被优化为并行指令），这个道理和对于排序元素少的情况下，我们会选择「选择排序」而不是「归并排序」是一样的。

更多精彩内容，欢迎关注：[公众号](#) / [Github](#) / [LeetCode](#) / [知乎](#) \*\*

## 题目描述

这是 LeetCode 上的 [231. 2 的幂](#)，难度为 简单。

Tag：「数学」、「位运算」

给你一个整数  $n$ ，请你判断该整数是否是 2 的幂次方。如果是，返回 `true`；否则，返回 `false`。

如果存在一个整数  $x$  使得  $n == 2^x$ ，则认为  $n$  是 2 的幂次方。

示例 1：

输入： $n = 1$   
输出：`true`  
解释： $2^0 = 1$

示例 2：

输入： $n = 16$   
输出：`true`  
解释： $2^4 = 16$

示例 3：

输入： $n = 3$   
输出：`false`

示例 4：

输入： $n = 4$   
输出：`true`

宫水三叶  
の  
刷题日记

公众号：宫水三叶的刷题日记



示例 5：

输入：n = 5  
输出：false

提示：

- $-2^{31} \leq n \leq 2^{31} - 1$

进阶：你能够不使用循环/递归解决此问题吗？

---

## 朴素做法

首先小于等于 0 的数必然不是，1 必然是。

在处理完这些边界之后，尝试将  $n$  除干净，如果最后剩余数值为 1 则说明开始是 2 的幂。

代码：

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

- 时间复杂度： $O(\log n)$
- 空间复杂度： $O(1)$

---

## lowbit

熟悉树状数组的同学都知道，`lowbit` 可以快速求得  $x$  二进制表示中最低位 1 表示的值。

如果一个数  $n$  是 2 的幂，那么有 `lowbit(n) = n` 的性质（2 的幂的二进制表示中必然是最高位为 1，低位为 0）。

代码：

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

- 时间复杂度： $O(1)$
- 空间复杂度： $O(1)$

更多精彩内容，欢迎关注：[公众号](#) / [Github](#) / [LeetCode](#) / [知乎](#)

## 题目描述

这是 LeetCode 上的 **342. 4的幂**，难度为 **简单**。

Tag：「数学」、「位运算」

给定一个整数，写一个函数来判断它是否是 4 的幂次方。如果是，返回 true；否则，返回 false。

整数  $n$  是 4 的幂次方需满足：存在整数  $x$  使得  $n == 4^x$

示例 1：

输入： $n = 16$

输出：true

示例 2：

输入： $n = 5$

输出：false

示例 3：

宫水三叶  
の  
刷题日记

公众号：宫水三叶的刷题日记

输入：n = 1  
输出：true

提示：

$$\bullet -2^{31} \leq n \leq 2^{31} - 1$$

进阶：

你能不使用循环或者递归来完成本题吗？

---

## 基本分析

一个数  $n$  如果是 4 的幂，等价于  $n$  为质因数只有 2 的平方数。

因此我们可以将问题其转换：判断  $\sqrt{n}$  是否为 2 的幂。

判断某个数是否为 2 的幂的分析在（题解）[231. 2 的幂](#) 这里。

---

## sqrt + lowbit

我们可以先对  $n$  执行 `sqrt` 函数，然后应用 `lowbit` 函数快速判断  $\sqrt{n}$  是否为 2 的幂。

代码：

```
class Solution {
    public boolean isPowerOfFour(int n) {
        if (n <= 0) return false;
        int x = (int)Math.sqrt(n);
        return x * x == n && (x & -x) == x;
    }
}
```

宫水三叶  
の  
刷题日记

公众号：宫水三叶的刷题日记

```

class Solution {
    public boolean isPowerOfFour(int n) {
        if (n <= 0) return false;
        int x = getVal(n);
        return x * x == n && (x & -x) == x;
    }
    int getVal(int n) {
        long l = 0, r = n;
        while (l < r) {
            long mid = l + r >> 1;
            if (mid * mid >= n) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        return (int)r;
    }
}

```

- 时间复杂度：复杂度取决于内置函数 `sqrt`。一个简单的 `sqrt` 的实现接近于 P2 的代码。复杂度为  $O(\log n)$
- 空间复杂度： $O(1)$

\*\*🔍更多精彩内容，欢迎关注：[公众号](#) / [Github](#) / [LeetCode](#) / [知乎](#) \*\*

## 题目描述

这是 LeetCode 上的 **371. 两整数之和**，难度为 **中等**。

Tag：「位运算」

给你两个整数 `a` 和 `b`，不使用运算符 `+` 和 `-`，计算并返回两整数之和。

示例 1：

输入：a = 1, b = 2

输出：3

宫水三叶  
刷题日记

公众号：宫水三叶的刷题日记

## 示例 2：

输入：a = 2, b = 3

输出：5  
```

提示：

\* -1000 <= a, b <= 1000

---

### 位运算

<p class="mume-header " id="位运算"></p>

一个朴素的做法是使用「位运算」，利用二进制的「逢二进一」和「`int` 二进制表示长度为 32」，我们可以从低位往高位进行处理，处理

然后对两数当前位进行分情况讨论：

- \* 两个当前位均为 1：此时当前位是什么取决于前一位的进位情况，即有  $ans |= (t << i)$ ，同时进位  $t = 1$ ；
- \* 两个当前位只有一位是 1：此时当前位是什么取决于前一位的进位情况（整理后可统一为  $ans |= ((1 ^ t) << i)$ ：
  - \* 前一进位若为 1，结合该位为 1，则有当前位为 0，进位不变  $t = 1$ ；
  - \* 前一进位若为 0，结合该位为 1，则有当前位为 1，进位不变  $t = 0$ ；
- \* 两个当前位为 0：此时当前位是什么取决于前一位的进位情况，则有  $ans |= (t << i)$ ，同时进位  $t = 0$ 。

代码：

```
``Java
class Solution {
    public int getSum(int a, int b) {
        int ans = 0;
        for (int i = 0, t = 0; i < 32; i++) {
            int u1 = (a >> i) & 1, u2 = (b >> i) & 1;
            if (u1 == 1 && u2 == 1) {
                ans |= (t << i);
                t = 1;
            } else if (u1 == 1 || u2 == 1) {
                ans |= ((1 ^ t) << i);
            } else {
                ans |= (t << i);
                t = 0;
            }
        }
        return ans;
    }
}
```

宫水三叶  
刷题日记

公众号：宫水三叶的刷题日记

- 时间复杂度： $O(C)$ ， $C$  为常数，固定为 32
- 空间复杂度： $O(1)$

---

## 递归

在彻底理解「解法一」后，不难发现「解法一」中本质是分别对两数的当前位进行“拆分”求解。

先计算原始的  $a$  和原始  $b$  在不考虑进位的情况下的结果，结果为  $a \oplus b$ ，然后在此基础上，考虑将进位累加进来，累加操作可以递归使用 `getSum` 来处理。

问题转化为如何求得  $a$  和  $b$  的进位值。

不难发现，当且仅当  $a$  和  $b$  的当前位均为 1，该位才存在进位，同时进位回应用到当前位的下一位（左边的一边，高一位），因此最终的进位结果为  $(a \& b) \ll 1$ 。

因此，递归调用 `getSum(a  $\oplus$  b, (a & b) << 1)` 我们可以得到最终答案。

最后还要考虑，该拆分过程何时结束。

由于在进位结果  $(a \& b) \ll 1$  中存在左移操作，因此最多执行 32 次的递归操作之后，该值会变为 0，而 0 与任何值  $x$  相加结果均为  $x$ 。

代码：

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

- 时间复杂度：令  $C$  为常数，固定为 32，最多执行  $C$  次的  $(a \& b) \ll 1$ ，递归过程结束。复杂度为  $O(C)$
- 空间复杂度： $O(1)$

## 题目描述

这是 LeetCode 上的 [405. 数字转换为十六进制数](#)，难度为 简单。

Tag：「位运算」、「模拟」

给定一个整数，编写一个算法将这个数转换为十六进制数。对于负整数，我们通常使用 补码运算 方法。

注意：

1. 十六进制中所有字母(a-f)都必须是小写。
2. 十六进制字符串中不能包含多余的前导零。如果要转化的数为0，那么以单个字符'0'来表示；对于其他情况，十六进制字符串中的第一个字符将不会是0字符。
3. 给定的数确保在32位有符号整数范围内。
4. 不能使用任何由库提供的将数字直接转换或格式化为十六进制的方法。

示例 1：

输入:26

输出:"1a"

示例 2：

输入:-1

输出:"ffffffff"

## 模拟 + 进制转换

首先，我们可以利用通用的进制转换思路来做，不断循环  $\text{num} \% k$  和  $\text{num} / k$  的操作来构造出  $k$  进制每一位。

但需要处理「补码」问题：对于负数的  $\text{num}$ ，我们需要先在  $\text{num}$  基础上加上  $2^{32}$  的偏移量，再进行进制转换。

代码：

```

class Solution {
    public String toHex(int _num) {
        if (_num == 0) return "0";
        long num = _num;
        StringBuilder sb = new StringBuilder();
        if (num < 0) num = (long)(Math.pow(2, 32) + num);
        while (num != 0) {
            long u = num % 16;
            char c = (char)(u + '0');
            if (u >= 10) c = (char)(u - 10 + 'a');
            sb.append(c);
            num /= 16;
        }
        return sb.reverse().toString();
    }
}

```

- 时间复杂度：复杂度取决于构造的十六进制数的长度，固定为  $C = 8$ 。整体复杂度为  $O(C)$
- 空间复杂度：复杂度取决于构造的十六进制数的长度，固定为  $C = 8$ 。整体复杂度为  $O(C)$

## 位运算 + 分组换算

将长度为 32 的二进制转换为 16 进制数，本质是对长度为 32 的二进制数进行分组，每 4 个一组（二进制  $(1111)_2$  表示 15，则使用长度为 4 的二进制可以表示 0-15）。

同时，由于我们是直接对长度为 32 的二进制进行分组换算（4 个为一组，共 8 组），而长度为 32 的二进制本身就是使用补码规则来表示的，因此我们无须额外处理「补码」问题。

具体的，我们将  $num$  与  $15 = (1111)_2$  进行  $\&$  运算，然后对  $num$  进行无符号右移 4 位来实现每 4 位处理。

代码：

宫水三叶  
の  
刷题日记

公众号：宫水三叶的刷题日记



```

class Solution {
    public String toHex(int num) {
        if (num == 0) return "0";
        StringBuilder sb = new StringBuilder();
        while (num != 0) {
            int u = num & 15;
            char c = (char)(u + '0');
            if (u >= 10) c = (char)(u - 10 + 'a');
            sb.append(c);
            num >>= 4;
        }
        return sb.reverse().toString();
    }
}

```

- 时间复杂度：复杂度取决于构造的十六进制数的长度，固定为  $C = 8$ 。整体复杂度为  $O(C)$
- 空间复杂度：复杂度取决于构造的十六进制数的长度，固定为  $C = 8$ 。整体复杂度为  $O(C)$

更多精彩内容，欢迎关注：[公众号](#) / [Github](#) / [LeetCode](#) / [知乎](#)

## 题目描述

这是 LeetCode 上的 **461. 汉明距离**，难度为 简单。

Tag：「位运算」

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数  $x$  和  $y$ ，计算它们之间的汉明距离。

注意：

$0 \leq x, y < 231$ .

示例:

宫水三叶  
刷题日记

公众号: 宫水三叶的刷题日记

输入:  $x = 1, y = 4$

输出: 2

解释:

```
1   (0 0 0 1)
4   (0 1 0 0)
      ↑   ↑
```

上面的箭头指出了对应二进制位不同的位置。

## 逐位比较

本身不改变  $x$  和  $y$ ，每次取不同的偏移位进行比较，不同则加一。

循环固定取满 32。

执行结果: **通过** [显示详情 >](#)

[添加备注](#)

执行用时: **0 ms**，在所有 Java 提交中击败了 **100.00%** 的用户

内存消耗: **35.3 MB**，在所有 Java 提交中击败了 **37.14%** 的用户

炫耀一下:



[写题解，分享我的解题思路](#)

代码:

宫水三叶  
の  
刷题日记

公众号: 宫水三叶的刷题日记

```
class Solution {  
    public int hammingDistance(int x, int y) {  
        int ans = 0;  
        for (int i = 0; i < 32; i++) {  
            int a = (x >> i) & 1, b = (y >> i) & 1;  
            ans += a != b ? 1 : 0;  
        }  
        return ans;  
    }  
}
```

- 时间复杂度： $O(C)$ ， $C$  固定为 32
- 空间复杂度： $O(1)$

## 右移统计

每次都统计当前  $x$  和  $y$  的最后一位，统计完则将  $x$  和  $y$  右移一位。

当  $x$  和  $y$  的最高一位 1 都被统计过之后，循环结束。

执行结果： **通过** [显示详情](#)

[添加备注](#)

执行用时： **0 ms**，在所有 Java 提交中击败了 **100.00%** 的用户

内存消耗： **35.3 MB**，在所有 Java 提交中击败了 **31.99%** 的用户

炫耀一下：



[写题解，分享我的解题思路](#)

代码：

宫水三叶  
の  
刷题日记

公众号：宫水三叶的刷题日记

```
class Solution {
    public int hammingDistance(int x, int y) {
        int ans = 0;
        while ((x | y) != 0) {
            int a = x & 1, b = y & 1;
            ans += a ^ b;
            x >>= 1; y >>= 1;
        }
        return ans;
    }
}
```

- 时间复杂度： $O(C)$ ， $C$  最多为 32
- 空间复杂度： $O(1)$

## lowbit

熟悉树状数组的同学都知道，`lowbit` 可以快速求得  $x$  二进制表示中最低位 1 表示的值。

因此我们可以先将  $x$  和  $y$  进行异或，再统计异或结果中 1 的个数。

执行结果： **通过** [显示详情 >](#)

[添加备注](#)

执行用时： **0 ms** ，在所有 Java 提交中击败了 **100.00%** 的用户

内存消耗： **35.1 MB** ，在所有 Java 提交中击败了 **75.88%** 的用户

炫耀一下：



[写题解，分享我的解题思路](#)

代码：

宫水三叶  
の  
刷题日记

公众号：宫水三叶的刷题日记

```
class Solution {
    int lowbit(int x) {
        return x & -x;
    }
    public int hammingDistance(int x, int y) {
        int ans = 0;
        for (int i = x ^ y; i > 0; i -= lowbit(i)) ans++;
        return ans;
    }
}
```

- 时间复杂度： $O(C)$ ， $C$  最多为 32
- 空间复杂度： $O(1)$

更多精彩内容，欢迎关注：[公众号](#) / [Github](#) / [LeetCode](#) / [知乎](#)

## 题目描述

这是 LeetCode 上的 [477. 汉明距离总和](#)，难度为 中等。

Tag：「位运算」、「数学」

两个整数的 汉明距离 指的是这两个数字的二进制数对应位不同的数量。

给你一个整数数组 nums，请你计算并返回 nums 中任意两个数之间汉明距离的总和。

示例 1：

输入：nums = [4,14,2]

输出：6

解释：在二进制表示中，4 表示为 0100，14 表示为 1110，2 表示为 0010。（这样表示是为了体现后四位之间关系）  
所以答案为：

$\text{HammingDistance}(4, 14) + \text{HammingDistance}(4, 2) + \text{HammingDistance}(14, 2) = 2 + 2 + 2 = 6$

示例 2：

输入：nums = [4,14,4]

输出：4

宫水三叶  
刷题日记

公众号：宫水三叶的刷题日记

提示：

```
1 <= nums.length <= $10^5$  
0 <= nums[i] <= $10^9$
```

## 按位统计

我们知道，汉明距离为两数二进制表示中不同位的个数，同时每位的统计是相互独立的。

即最终的答案为  $\sum_{x=0}^{31} calc(x)$ ，其中  $calc$  函数为求得所有数二进制表示中的某一位  $x$  所产生的不同位的个数。

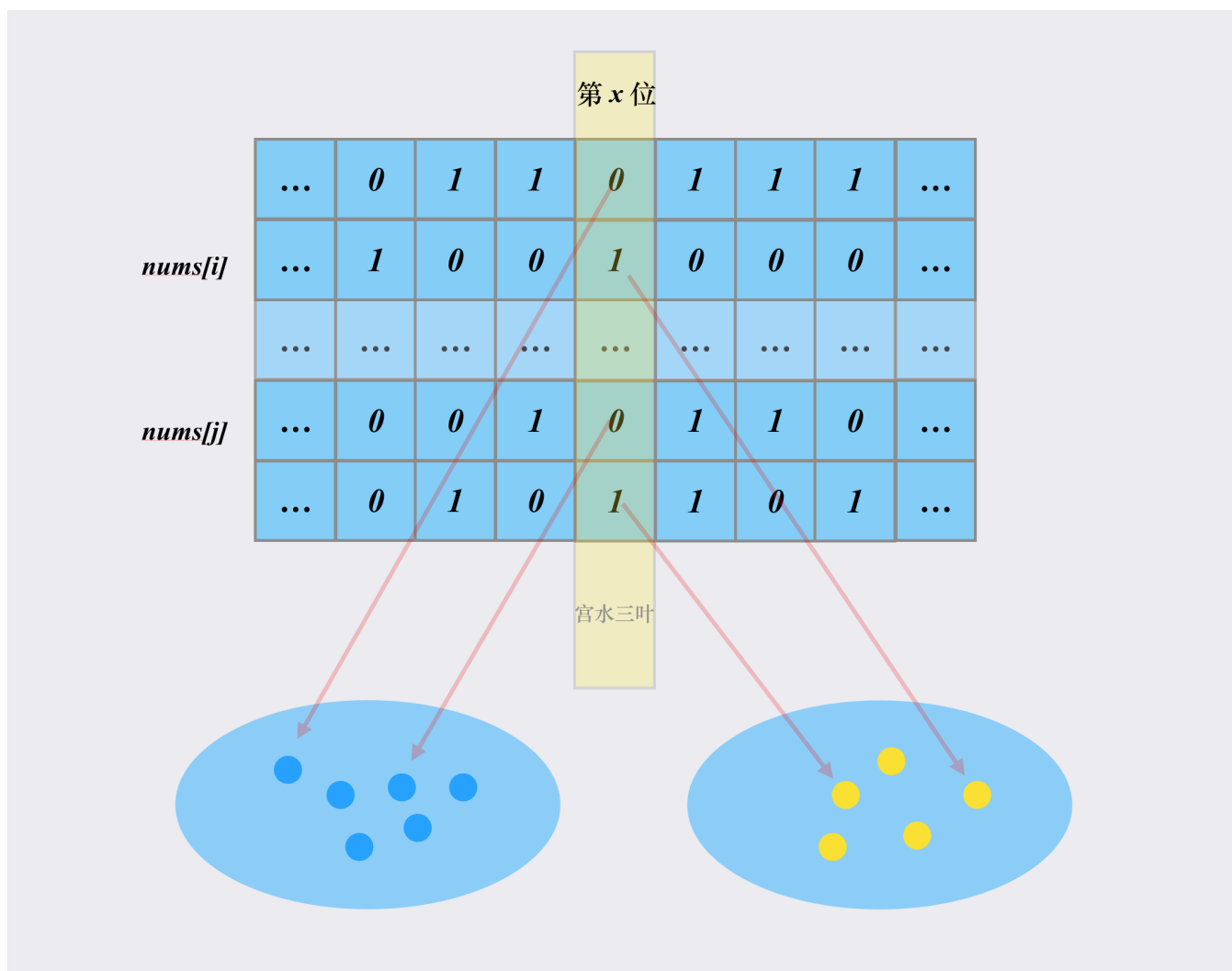
我们考虑某个  $calc(x)$  如何求得：

事实上，对于某个 `nums[i]` 我们只关心在 `nums` 中有多少数的第  $x$  位的与其不同，而不关心具体是哪些数与其不同，同时二进制表示中非 0 即 1。

这指导我们可以建立两个集合  $s_0$  和  $s_1$ ，分别统计出 `nums` 中所有数的第  $x$  位中 0 的个数和 1 的个数，集合中的每次计数代表了 `nums` 中的某一元素，根据所在集合的不同代表了其第  $x$  位的值。那么要找到在 `nums` 中有多少数与某一个数的第  $x$  位不同，只需要读取另外一个集合的元素个数即可，变成了  $O(1)$  操作。那么要求得「第  $x$  位所有不同数」的对数的个数，只需要应用乘法原理，将两者元素个数相乘即可。

宫水三叶  
の  
刷题日记

公众号：宫水三叶的刷题日记



前面说到每位的统计是相对独立的，因此只要对「每一位」都应用上述操作，并把「每一位」的结果累加即是最终答案。

代码：

宫水三叶  
の  
刷题日记

公众号：宫水三叶的刷题日记

```

class Solution {
    public int totalHammingDistance(int[] nums) {
        int ans = 0;
        for (int x = 31; x >= 0; x--) {
            int s0 = 0, s1 = 0;
            for (int u : nums) {
                if (((u >> x) & 1) == 1) {
                    s1++;
                } else {
                    s0++;
                }
            }
            ans += s0 * s1;
        }
        return ans;
    }
}

```

- 时间复杂度： $O(C * n)$ ， $C$  固定为 32
- 空间复杂度： $O(1)$

更多精彩内容，欢迎关注：[公众号](#) / [Github](#) / [LeetCode](#) / [知乎](#)

## 题目描述

这是 LeetCode 上的 **526. 优美的排列**，难度为 **中等**。

Tag：「位运算」、「状压 DP」、「动态规划」

假设有从 1 到  $N$  的  $N$  个整数，如果从这  $N$  个数字中成功构造出一个数组，使得数组的第  $i$  位 ( $1 \leq i \leq N$ ) 满足如下两个条件中的一个，我们就称这个数组为一个优美的排列。

条件：

- 第  $i$  位的数字能被  $i$  整除
- $i$  能被第  $i$  位上的数字整除

现在给定一个整数  $N$ ，请问可以构造多少个优美的排列？

示例1:



输入：2

输出：2

解释：

第 1 个优美的排列是 [1, 2]：

第 1 个位置 ( $i=1$ ) 上的数字是 1，1 能被  $i$  ( $i=1$ ) 整除

第 2 个位置 ( $i=2$ ) 上的数字是 2，2 能被  $i$  ( $i=2$ ) 整除

第 2 个优美的排列是 [2, 1]：

第 1 个位置 ( $i=1$ ) 上的数字是 2，2 能被  $i$  ( $i=1$ ) 整除

第 2 个位置 ( $i=2$ ) 上的数字是 1，1 能被  $i$  ( $i=2$ ) 整除

说明：

- $N$  是一个正整数，并且不会超过 15。

## 状态压缩 DP

利用数据范围不超过 15，我们可以使用「状态压缩 DP」进行求解。

使用一个二进制数表示当前哪些数已被选，哪些数未被选，目的是为了可以使用位运算进行加速。

我们可以通过一个具体的样例，来感受下「状态压缩」是什么意思：

例如  $(000...0101)_2$  代表值为 1 和值为 3 的数字已经被使用了，而值为 2 的节点尚未被使用。

然后再来看看使用「状态压缩」的话，一些基本的操作该如何进行：

假设变量  $state$  存放了「当前数的使用情况」，当我们需要检查值为  $k$  的数是否被使用时，可以使用位运算  $a = (state \gg k) \& 1$ ，来获取  $state$  中第  $k$  位的二进制表示，如果  $a$  为 1 代表值为  $k$  的数字已被使用，如果为 0 则未被访问。

定义  $f[i][state]$  为考虑前  $i$  个数，且当前选择方案为  $state$  的所有方案数量。

一个显然的初始化条件为  $f[0][0] = 1$ ，代表当我们不考虑任何数 ( $i = 0$ ) 的情况下，一个数都不被选择 ( $state = 0$ ) 为一种合法方案。

不失一般性的考虑  $f[i][state]$  该如何转移，由于本题是求方案数，我们的转移方程必须做到

「不重不漏」。

我们可以通过枚举当前位置  $i$  是选哪个数，假设位置  $i$  所选数值为  $k$ ，首先  $k$  值需要同时满足如下两个条件：

- $state$  中的第  $k$  位为 1；
- 要么  $k$  能被  $i$  整除，要么  $i$  能被  $k$  整除。

那么根据状态定义，位置  $i$  选了数值  $k$ ，通过位运算我们可以直接得出决策位置  $i$  之前的状态是什么： $state \& (\neg(1 \ll (k - 1)))$ ，代表将  $state$  的二进制表示中的第  $k$  位置 0。

最终的  $f[i][state]$  为当前位置  $i$  选择的是所有合法的  $k$  值的方案数之和：

$$f[i][state] = \sum_{k=1}^n f[i-1][state \& (\neg(1 \ll (k - 1)))]$$

一些细节：由于给定的数值范围为  $[1, n]$ ，但实现上为了方便，我们使用  $state$  从右往左的第 0 位表示数值 1 选择情况，第 1 位表示数值 2 的选择情况 ... 即对选择数值  $k$  做一个  $-1$  的偏移。

代码：

宫水三叶  
の  
刷题日记

公众号：宫水三叶的刷题日记

```

class Solution {
    public int countArrangement(int n) {
        int mask = 1 << n;
        int[][] f = new int[n + 1][mask];
        f[0][0] = 1;
        for (int i = 1; i <= n; i++) {
            // 枚举所有的状态
            for (int state = 0; state < mask; state++) {
                // 枚举位置 i (最后一位) 选的数值是 k
                for (int k = 1; k <= n; k++) {
                    // 首先 k 在 state 中必须是 1
                    if (((state >> (k - 1)) & 1) == 0) continue;
                    // 数值 k 和位置 i 之间满足任一整除关系
                    if (k % i != 0 && i % k != 0) continue;
                    // state & (~(1 << (k - 1))) 代表将 state 中数值 k 的位置置零
                    f[i][state] += f[i - 1][state & (~(1 << (k - 1)))];
                }
            }
        }
        return f[n][mask - 1];
    }
}

```

- 时间复杂度：共有  $n * 2^n$  的状态需要被转移，每次转移复杂度为  $O(n)$ ，整体复杂度为  $O(n^2 * 2^n)$
- 空间复杂度： $O(n * 2^n)$

## 状态压缩 DP（优化）

通过对朴素的状态 DP 的分析，我们发现，在决策第  $i$  位的时候，理论上我们应该使用的数字数量也应该为  $i$  个。

但这一点在朴素状态 DP 中并没有体现，这就导致了我们在决策第  $i$  位的时候，仍然需要对所有的  $state$  进行计算检查（即使是那些二进制表示中 1 的出现次数不为  $i$  个的状态）。

因此我们可以换个思路进行枚举（使用新的状态定义并优化转移方程）。

定义  $f[state]$  为当前选择数值情况为  $state$  时的所有方案的数量。

这样仍然有  $f[0] = 1$  的初始化条件，最终答案为  $f[(1 << n) - 1]$ 。

不失一般性考虑  $f[state]$  如何计算：

从当前状态  $state$  进行出发，检查  $state$  中的每一位 1 作为最后一个被选择的数值，这样仍然可以确保方案数「不重不漏」的被统计，同时由于我们「从小到大」对  $state$  进行枚举，因此计算  $f[state]$  所依赖的其他状态值必然都已经被计算完成。

同样的，我们仍然需要确保  $state$  中的那一位作为最后一个的 1 需要与所放的位置成整除关系。

因此我们需要一个计算  $state$  的 1 的个数的方法，这里使用 *lowbit* 实现即可。

最终的  $f[state]$  为当前位置选择的是所有合法值的方案数之和：

$$f[state] = \sum_{i=0}^n f[state \& (\neg(1 \ll i))]$$

代码：

宫水三叶  
の  
刷题日记

公众号：宫水三叶的刷题日记

```

class Solution {
    int getCnt(int x) {
        int ans = 0;
        while (x != 0) {
            x -= (x & -x); // lowbit
            ans++;
        }
        return ans;
    }
    public int countArrangement(int n) {
        int mask = 1 << n;
        int[] f = new int[mask];
        f[0] = 1;
        // 枚举所有的状态
        for (int state = 1; state < mask; state++) {
            // 计算 state 有多少个 1 (也就是当前排序长度为多少)
            int cnt = getCnt(state);
            // 枚举最后一位数值为多少
            for (int i = 0; i < n; i++) {
                // 数值在 state 中必须是 1
                if (((state >> i) & 1) == 0) continue;
                // 数值 (i + 1) 和位置 (cnt) 之间满足任一整除关系
                if ((i + 1) % cnt != 0 && cnt % (i + 1) != 0) continue;
                // state & ~(1 << i) 代表将 state 中所选数值的位置置零
                f[state] += f[state & ~(1 << i)];
            }
        }
        return f[mask - 1];
    }
}

```

- 时间复杂度：共有  $2^n$  的状态需要被转移，每次转移复杂度为  $O(n)$ ，整体复杂度为  $O(n * 2^n)$
- 空间复杂度： $O(2^n)$

## 总结

不难发现，其实两种状态压缩 DP 的思路其实是完全一样的。

只不过在朴素状压 DP 中我们是显式的枚举了考虑每一种长度的情况（存在维度  $i$ ），而在状压 DP（优化）中利用则  $state$  中的 1 的个数中蕴含的长度信息。

## 题目描述

这是 LeetCode 上的 [1178. 猜字谜](#)，难度为 **困难**。

Tag：「状态压缩」、「位运算」、「哈希表」

外国友人仿照中国字谜设计了一个英文版猜字谜小游戏，请你来猜猜看吧。

字谜的谜面 puzzle 按字符串形式给出，如果一个单词 word 符合下面两个条件，那么它就可以算作谜底：

- 单词 word 中包含谜面 puzzle 的第一个字母。
- 单词 word 中的每一个字母都可以在谜面 puzzle 中找到。

例如，如果字谜的谜面是 “abcdefg”，那么可以作为谜底的单词有 “faced”，“cabbage”，和 “baggage”；而 “beefed”（不含字母 “a”）以及 “based”（其中的 “s” 没有出现在谜面中）都不能作为谜底。

返回一个答案数组 answer，数组中的每个元素 answer[i] 是在给出的单词列表 words 中可以作为字谜谜面 puzzles[i] 所对应的谜底的单词数目。

示例：

输入：

```
words = ["aaaa","asas","able","ability","actt","actor","access"],  
puzzles = ["aboveyz","abrodyz","abslute","absoryz","actresz","gaswxyz"]
```

输出：[1,1,3,2,4,0]

解释：

- 1 个单词可以作为 "aboveyz" 的谜底："aaaa"
- 1 个单词可以作为 "abrodyz" 的谜底："aaaa"
- 3 个单词可以作为 "abslute" 的谜底："aaaa", "asas", "able"
- 2 个单词可以作为 "absoryz" 的谜底："aaaa", "asas"
- 4 个单词可以作为 "actresz" 的谜底："aaaa", "asas", "actt", "access"
- 没有单词可以作为 "gaswxyz" 的谜底，因为列表中的单词都不含字母 'g'。

提示：

- $1 \leq \text{words.length} \leq 10^5$

- $4 \leq \text{words}[i].\text{length} \leq 50$
- $1 \leq \text{puzzles.length} \leq 10^4$
- $\text{puzzles}[i].\text{length} == 7$
- $\text{words}[i][j]$ ,  $\text{puzzles}[i][j]$  都是小写英文字母。
- 每个  $\text{puzzles}[i]$  所包含的字符都不重复。

---

## 朴素位运算解法(TLE)

根据「谜底」和「谜面」的对应条件：

- 单词 `word` 中包含谜面 `puzzle` 的第一个字母。
- 单词 `word` 中的每一个字母都可以在谜面 `puzzle` 中找到

`puzzle` 本身长度只有 7 位，而且不重复；我们可以发现对应条件与 `word` 的重复字母无关。

因此我们可以使用「二进制」数来表示每一个 `word` 和 `puzzle`：

一个长度为 26 的二进制数来表示（直接使用长度为 32 的 `int` 即可，使用低 26 位），假如有 `str = "abz"` 则对应了 `100...011` (共 26 位，从右往左是 a - z)。

至此我们可以已经可以得出一个朴素解法的思路了：

1. 预处理除所有的 `word` 对应的二进制数字。计算量为  $50 * 10^5$ ，数量级为  $10^6$
2. 对每个 `puzzle` 进行条件判定（每一个 `puzzle` 都需要遍历所有的 `word` 进行检查）。计算量为  $10^5 * 10^4$ ，数量级为  $10^9$

计算机单秒的计算量为  $10^7$  左右（OJ 测评器通常在  $10^6 \sim 10^7$  之间），哪怕忽略常数后，我们的总运算也超过了上限，铁定超时。

代码：

宫水三叶  
の  
刷题日记

公众号: 宫水三叶的刷题日记

```

class Solution {
    public List<Integer> findNumOfValidWords(String[] ws, String[] ps) {
        // 预处理出所有的 word 所对应的二进制数值
        List<Integer> list = new ArrayList<>();
        for (String w : ws) list.add(getBin(w));
        // 判定每个 puzzles 有多少个谜底
        List<Integer> ans = new ArrayList<>();
        for (String p : ps) ans.add(getCnt(list, p));
        return ans;
    }
    // 判定某个 puzzles 有多少个谜底
    int getCnt(List<Integer> ws, String str) {
        int ans = 0;
        // 获取当前 puzzles 对应的二进制数字
        int t = getBin(str);
        // 当前 puzzles 的首个字符在二进制数值中的位置
        int first = str.charAt(0) - 'a';
        for (int w : ws) {
            // check 条件一：单词 word 中包含谜面 puzzle 的第一个字母
            if ((w >> first & 1) == 0) continue;
            // check 条件二：单词 word 中的每一个字母都可以在谜面 puzzle 中找到
            if ((w & t) == w) ans++;
        }
        return ans;
    }
    // 将 str 所包含的字母用二进制标识
    // 如果 str = abz 则对应的二进制为 100...011 (共 26 位，从右往左是 a - z)
    int getBin(String str) {
        int t = 0;
        char[] cs = str.toCharArray();
        for (char c : cs) {
            // 每一位字符所对应二进制数字中哪一位
            int u = c - 'a';
            // 如果当前位置为 0，代表还没记录过，则进行记录（不重复记录）
            if ((t >> u & 1) == 0) t += 1 << u;
        }
        return t;
    }
}

```

- 时间复杂度： $O(words.length * (words[i].length + puzzles.length))$
- 空间复杂度：每个 word 对应了一个 int，每个 puzzle 对应了一个答案。复杂度为  $O(words.length + puzzles.length)$



## 哈希表 & 位运算解法

因此我们需要优化上述步骤 1 或者步骤 2。显然超时的主要原因是步骤 2 计算量太多了。

一个很显眼的突破口是利用 `puzzles[i].length == 7`，同时判定条件 1 对 `puzzle` 的首字母进行了限定。

对于一个确定的 `puzzle` 而言，我们要找它有多少个「谜底」。可以通过枚举它所有可能的「谜底」，再去 `words` 里面找每一个「谜底」出现了多少次。

结合题意的话，就是固定住 `puzzle` 的首位，去枚举其余后面的 6 位的所有的可能性（每一位都有保留和不保留两种选择），即枚举子集的过程。

你可能还是无法理解，其实就是一个通过 `puzzle` 反推 `word` 的过程：

举个🍌吧，假如我们有 `puzzle` 是 `gabc`（假定现在的 `puzzle` 长度只有 4），那么可能的 `word` 有哪些？

1. 首先要满足条件一，也就是 `word` 必然包含首字母 `g`；
2. 然后是条件二，`word` 中的每一位都在 `puzzle` 出现过，因此可能的 `word` 包括 `g`、`ga`、`gb`、`gc`、`gab`、`gac`、`gbc`、`gabc`。

使用 1 和 0 代表 `puzzle` 每一位选择与否的话，其实就是对应了 1000、1100、1010、1001、1110、1101、1011、1111。

搞明白了这个过程之后，我们需要对 `words` 进行词频统计，我们可以使用「哈希表」记录相同含义的 `word` 出现了多少次（相同含义的意思是包含字母类型一样的 `word`，因为答案和 `word` 的重复字符无关）

这样做的复杂度/计算量是多少呢？

1. 统计所有 `word` 的词频。计算量为  $50 * 10^5$ ，数量级为  $10^6$
2. 对应每个 `puzzle` 而言，由于其长度确定为 7，因此所有枚举所有可能「谜底」的数量不为  $2^6=64$  个，可以看做是  $O(1)$  的，检查每个可能的「谜底」在 `words` 出现次数是通过哈希表，也是近似  $O(1)$  的。因此在确定一个 `puzzle` 的答案时，与 `words` 的长度无关。计算量为  $10^4$ ，数量级为  $10^4$

计算机单秒的计算量为  $10^7$  左右（OJ 测评器通常在  $10^6 \sim 10^7$  之间），因此可以过。

代码：

刷题日记

公众号: 宫水三叶的刷题日记

```

class Solution {
    public List<Integer> findNumOfValidWords(String[] ws, String[] ps) {
        // 转用「哈希表」来统计出所有的 word 所对应的二进制数值
        Map<Integer, Integer> map = new HashMap<>();
        for (String w : ws) {
            int t = getBin(w);
            map.put(t, map.getOrDefault(t, 0) + 1);
        }
        // 判定每个 puzzle 有多少个谜底
        List<Integer> ans = new ArrayList<>();
        for (String p : ps) ans.add(getCnt(map, p));
        return ans;
    }

    int getCnt(Map<Integer, Integer> map, String str) {
        int ans = 0;
        int m = str.length();
        char[] cs = str.toCharArray();
        // 当前 puzzle 的首个字符在二进制数值中的位置
        int first = cs[0] - 'a';
        // 枚举「保留首个字母」的所有子集
        // 即我们需要先固定 puzzle 的首位字母，然后枚举剩余的 6 位是否保留
        // 由于是二进制，每一位共有 0 和 1 两种选择，因此共有  $2^6$  种可能性，也就是  $2^6 = 1 \ll (7 - 1)$ 
        // i 代表了所有「保留首个字母」的子集的「后六位」的二进制表示
        for (int i = 0; i < (1 << (m - 1)); i++) {
            // u 代表了当前可能的谜底。先将首字母提取出来
            int u = 1 << first;
            // 枚举「首个字母」之后的每一位
            for (int j = 1; j < m; j++) {
                // 如果当前位为 1，代表该位置要保留，将该位置的字母追加到谜底 u 中
                if (((i >> (j - 1)) & 1) != 0) u += 1 << (cs[j] - 'a');
            }
            // 查询这样的字符是否出现在 `words` 中，出现了多少次
            if (map.containsKey(u)) ans += map.get(u);
        }
        return ans;
    }

    // 将 str 所包含的字母用二进制标识
    // 如果 str = abz 则对应的二进制为 100...011 (共 26 位，从右往左是 a - z)
    int getBin(String str) {
        int t = 0;
        char[] cs = str.toCharArray();
        for (char c : cs) {
            // 每一位字符所对应二进制数字中哪一位
            int u = c - 'a';
            // 如果当前位置为 0，代表还没记录过，则进行记录 (不重复记录)
            if ((t >> u & 1) == 0) t += 1 << u;
        }
    }
}

```

```
    }  
    return t;  
}  
}
```

- 时间复杂度： $O(words.length * words[i].length + puzzles.length)$
- 空间复杂度：`word` 和 `puzzle` 分别具有最大长度和固定长度，使用空间主要取决于量数组的长度。复杂度为  $O(words.length + puzzles.length)$

---

## 位运算说明

$a \gg b \& 1$  代表检查  $a$  的第  $b$  位是否为 1，有两种可能性 0 或者 1

$a += 1 \ll b$  代表将  $a$  的第  $b$  位设置为 1 (当第  $b$  位为 0 的时候适用)

如不想写对第  $b$  位为 0 的前置判断， $a += 1 \ll b$  也可以改成  $a |= 1 \ll b$

PS. 1 的二进制就是最低位为 1，其他位为 0 哦

以上两个操作在位运算中出现频率超高，建议每位同学都加深理解。

---

## 点评

这道题解发到 LeetCode 之后，很多同学反映还是看不懂，还是不理解。

于是我重新思考了这道题的每一个环节。

这道题之所是 Hard，是因为考察的都是违反人性“直觉”的东西：

1. 状态压缩：对一个单词出现过哪些字母，不能采用我们直观中的 map/set 进行记录，而要利用一个长度为 26 的二进制数来记录，对于某个字母需要计算在二进制数中的哪一位，如果出现过用 1 表示，没出现过用 0 表示
2. 正难则反：不能从 `words` 数组出发，去检查有哪些 `word` 符合要求；而要反过来从 `puzzle` 出发，去枚举当前 `puzzle` 所有合法的 `word`，再去确定这些合法的 `word` 在真实的 `words` 数组中出现了多少次

大家要尽量去理解这种思路的合理性，当这种思路也形成意识的时候，这种题也就不难了。

\*\*🔍更多精彩内容，欢迎关注：[公众号](#) / [Github](#) / [LeetCode](#) / [知乎](#) \*\*

## 题目描述

这是 LeetCode 上的 **1711. 大餐计数**，难度为 **中等**。

Tag：「哈希表」、「位运算」

大餐 是指 恰好包含两道不同餐品 的一餐，其美味程度之和等于 2 的幂。

你可以搭配 任意 两道餐品做一顿大餐。

给你一个整数数组 `deliciousness`，其中 `deliciousness[i]` 是第  $i$  道餐品的美味程度，返回你可以用数组中的餐品做出的不同 大餐 的数量。结果需要对  $10^9 + 7$  取余。

注意，只要餐品下标不同，就可以认为是不同的餐品，即便它们的美味程度相同。

示例 1：

输入：`deliciousness = [1,3,5,7,9]`

输出：4

解释：大餐的美味程度组合为  $(1,3)$ 、 $(1,7)$ 、 $(3,5)$  和  $(7,9)$ 。  
它们各自的美味程度之和分别为 4、8、8 和 16，都是 2 的幂。

示例 2：

输入：`deliciousness = [1,1,1,3,3,3,7]`

输出：15

解释：大餐的美味程度组合为 3 种  $(1,1)$ ，9 种  $(1,3)$ ，和 3 种  $(1,7)$ 。

提示：

- $1 \leq \text{deliciousness.length} \leq 10^5$

- $0 \leq \text{deliciousness}[i] \leq 2^{20}$

## 枚举前一个数（TLE）

一个朴素的想法是，从前往后遍历 *deliciousness* 中的所有数，当遍历到下标 *i* 的时候，回头检查下标小于 *i* 的数是否能够与 *deliciousness*[*i*] 相加形成 2 的幂。

这样的做法是  $O(n^2)$  的，防止同样的数值被重复计算，我们可以使用「哈希表」记录某个数出现了多少次，但这并不改变算法仍然是  $O(n^2)$  的。

而且我们需要一个 `check` 方法来判断某个数是否为 2 的幂：

- 朴素的做法是对 *x* 应用试除法，当然因为精度问题，我们需要使用乘法实现试除；
- 另一个比较优秀的做法是利用位运算找到符合「大于等于 *x*」的最近的 2 的幂，然后判断是否与 *x* 相同。

两种做法差距有多大呢？方法一的复杂度为  $O(\log n)$ ，方法二为  $O(1)$ 。

根据数据范围  $0 \leq \text{deliciousness}[i] \leq 2^{20}$ ，方法一最多也就是执行不超过 22 次循环。

显然，采用何种判断 2 的幂的做法不是关键，在 OJ 判定上也只是分别卡在 60/70 和 62/70 的 TLE 上。

但通过这样的分析，我们可以发现「枚举前一个数」的做法是与 *n* 相关的，而枚举「可能出现的 2 的幂」则是有明确的范围，这引导出我们的解法二。

代码：

宫水三叶  
の  
刷题日记

公众号：宫水三叶的刷题日记

```

class Solution {
    int mod = (int)1e9+7;
    public int countPairs(int[] ds) {
        int n = ds.length;
        long ans = 0;
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < n; i++) {
            int x = ds[i];
            for (int other : map.keySet()) {
                if (check(other + x)) ans += map.get(other);
            }
            map.put(x, map.getOrDefault(x, 0) + 1);
        }
        return (int)(ans % mod);
    }
    boolean check(long x) {
        // 方法一
        // long cur = 1;
        // while (cur < x) {
        //     cur = cur * 2;
        // }
        // return cur == x;

        // 方法二
        return getVal(x) == x;
    }
    long getVal(long x) {
        long n = x - 1;
        n |= n >>> 1;
        n |= n >>> 2;
        n |= n >>> 4;
        n |= n >>> 8;
        n |= n >>> 16;
        return n < 0 ? 1 : n + 1;
    }
}

```

- 时间复杂度： $O(n^2)$
- 空间复杂度： $O(n)$

宫水三叶

刷题日记

公众号: 宫水三叶的刷题日记

## 枚举 2 的幂（容斥原理）

根据对朴素解法的分析，我们可以先使用「哈希表」对所有在 *deliciousness* 出现过的数进行统计。

然后对于每个数  $x$ ，检查所有可能出现的 2 的幂  $i$ ，再从「哈希表」中反查  $t = i - x$  是否存在，并实现计数。

一些细节：如果哈希表中存在  $t = i - x$ ，并且  $t = x$ ，这时候方案数应该是  $(cnts[x] - 1) * cnts[x]$ ；其余一般情况则是  $cnts[t] * cnts[x]$ 。

同时，这样的计数方式，我们对于二元组  $(x, t)$  会分别计数两次（遍历  $x$  和 遍历  $t$ ），因此最后要利用容斥原理，对重复计数的进行减半操作。

代码：

```
class Solution {
    int mod = (int)1e9+7;
    int max = 1 << 22;
    public int countPairs(int[] ds) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int d : ds) map.put(d, map.getOrDefault(d, 0) + 1);
        long ans = 0;
        for (int x : map.keySet()) {
            for (int i = 1; i < max; i <= 1) {
                int t = i - x;
                if (map.containsKey(t)) {
                    if (t == x) ans += (map.get(x) - 1) * 1L * map.get(x);
                    else ans += map.get(x) * 1L * map.get(t);
                }
            }
        }
        ans >>= 1;
        return (int)(ans % mod);
    }
}
```

- 时间复杂度：根据数据范围，令  $C$  为  $2^{21}$ 。复杂度为  $O(n * \log C)$
- 空间复杂度： $O(n)$

刷题日记

公众号：宫水三叶的刷题日记

## 枚举 2 的幂（边遍历边统计）

当然，我们也可以采取「一边遍历一边统计」的方式，这样取余操作就可以放在遍历逻辑中去做，也就顺便实现了不使用 *long* 来计数（以及不使用 `%` 实现取余）。


代码：

```
class Solution {
    int mod = (int)1e9+7;
    int max = 1 << 22;
    public int countPairs(int[] ds) {
        Map<Integer, Integer> map = new HashMap<>();
        int ans = 0;
        for (int x : ds) {
            for (int i = 1; i < max; i <= 1) {
                int t = i - x;
                if (map.containsKey(t)) {
                    ans += map.get(t);
                    if (ans >= mod) ans -= mod;
                }
            }
            map.put(x, map.getOrDefault(x, 0) + 1);
        }
        return ans;
    }
}
```

- 时间复杂度：根据数据范围，令  $C$  为  $2^{21}$ 。复杂度为  $O(n * \log C)$
- 空间复杂度： $O(n)$

---

 更多精彩内容，欢迎关注：[公众号](#) / [Github](#) / [LeetCode](#) / [知乎](#) \*\*

 **更新 Tips：**本专题更新时间为 2021-10-07，大概每 2-4 周 集中更新一次。

最新专题合集资料下载，可关注公众号「[宫水三叶的刷题日记](#)」，后台回复「位运算」获取下载链接。

觉得专题不错，可以请作者吃糖 ：

刷题日记

公众号: 宫水三叶的刷题日记





“给作者手机充个电”

YOLO 的赞赏码

版权声明：任何形式的转载请保留出处 [Wiki](#)。