

宫水三叶的刷题日记

数学

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「数学」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「数学」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「数学」；
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 [2. 两数相加](#)，难度为 中等。

Tag：「递归」、「链表」、「数学」、「模拟」

给你两个 非空 的链表，表示两个非负的整数。它们每位数字都是按照 逆序 的方式存储的，并且每个节点只能存储 一位 数字。

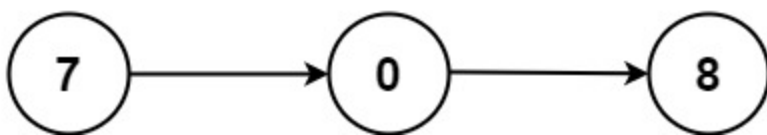
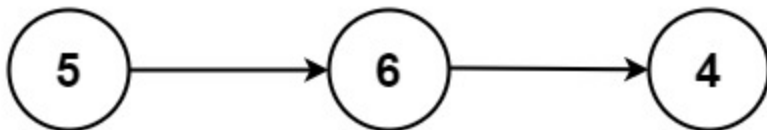
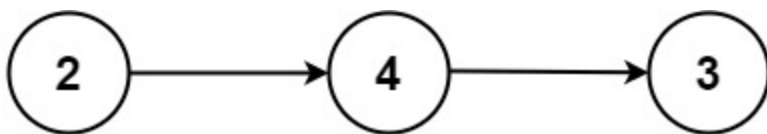
请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例 1：

刷题日记

公众号：宫水三叶的刷题日记



输入: $l1 = [2,4,3]$, $l2 = [5,6,4]$

输出: $[7,0,8]$

解释: $342 + 465 = 807$.

示例 2:

输入: $l1 = [0]$, $l2 = [0]$

输出: $[0]$

示例 3:

输入: $l1 = [9,9,9,9,9,9,9]$, $l2 = [9,9,9,9]$

输出: $[8,9,9,9,0,0,0,1]$

提示:

- 每个链表中的节点数在范围 $[1, 100]$ 内
- $0 \leq \text{Node.val} \leq 9$
- 题目数据保证列表表示的数字不含前导零

刷题日记

公众号: 宫水三叶的刷题日记

朴素解法（哨兵技巧）

这是道模拟题，模拟人工竖式做加法的过程：

从最低位至最高位，逐位相加，如果和大于等于 10，则保留个位数字，同时向前一位进 1 如果最高位有进位，则需在最前面补 1。

做有关链表的题目，有个常用技巧：添加一个虚拟头结点（哨兵），帮助简化边界情况的判断。

代码：

```
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode tmp = dummy;
        int t = 0;
        while (l1 != null || l2 != null) {
            int a = l1 != null ? l1.val : 0;
            int b = l2 != null ? l2.val : 0;
            t = a + b + t;
            tmp.next = new ListNode(t % 10);
            t /= 10;
            tmp = tmp.next;
            if (l1 != null) l1 = l1.next;
            if (l2 != null) l2 = l2.next;
        }
        if (t > 0) tmp.next = new ListNode(t);
        return dummy.next;
    }
}
```

- 时间复杂度： m 和 n 分别代表两条链表的长度，则遍历到的最远位置为 $\max(m, n)$ ，复杂度为 $O(\max(m, n))$
- 空间复杂度：创建了 $\max(m, n) + 1$ 个节点（含哨兵），复杂度为 $O(\max(m, n))$ （忽略常数）

注意：事实上还有可能创建 $\max(m, n) + 2$ 个节点，包含哨兵和最后一位的进位。但复杂度仍为 $O(\max(m, n))$ 。

题目描述

这是 LeetCode 上的 **6. Z 字形变换**，难度为 **中等**。

Tag：「模拟」、「数学」

将一个给定字符串 *s* 根据给定的行数 *numRows*，以从上往下、从左到右进行 Z 字形排列。

比如输入字符串为 "PAYPALISHIRING" 行数为 3 时，排列如下：

```
P   A   H   N
A P L S I I G
Y   I   R
```

之后，你的输出需要从左往右逐行读取，产生出一个新的字符串，比如 "PAHNAPLSIIGYIR"。

请你实现这个将字符串进行指定行数变换的函数

数：`string convert(string s, int numRows);`

示例 1：

```
输入：s = "PAYPALISHIRING", numRows = 3
输出："PAHNAPLSIIGYIR"
```

示例 2：

```
输入：s = "PAYPALISHIRING", numRows = 4
输出："PINALSIGYAHRPI"
解释：
P     I     N
A   L S   I G
Y A   H R
P     I
```

示例 3：

```
输入：s = "A", numRows = 1
输出："A"
```

提示：

宫水三叶
刷题日记

公众号：宫水三叶的刷题日记

- $1 \leq s.length \leq 1000$
 - s 由英文字母（小写和大写）、',' 和 '.' 组成
 - $1 \leq numRows \leq 1000$
-

直观规律解法

本题可以当成模拟题来做。

对需要打印的行 i 进行遍历，每一行的第一个字符可以直接确定：原字符开头的第 i 个字符。

然后计算出每行下一个字符的偏移量，这里需要分情况讨论：

- 对于第一行和最后一行：偏移量固定，不随着 z 的方向改变
- 对于其他行：偏移量随着 z 的方向发生变化

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 convert(String s, int r) {
        int n = s.length();
        if (n == 1 || r == 1) return s;

        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < r; i++) {
            if (i == 0 || i == r - 1) {
                int j = i;
                int rowOffset = (r - 1) * 2 - 1;
                while (j < n) {
                    sb.append(s.charAt(j));
                    j += rowOffset + 1;
                }
            } else {
                int j = i;

                int topRow = i;
                int topOffset = topRow * 2 - 1;

                int bottomRow = r - i - 1;
                int bottomOffset = bottomRow * 2 - 1;

                boolean flag = true;
                while (j < n) {
                    sb.append(s.charAt(j));
                    j += flag ? bottomOffset + 1 : topOffset + 1;
                    flag = !flag;
                }
            }
        }
        return sb.toString();
    }
}

```

- 时间复杂度：分别对 `s` 中的每个字符进行打印。复杂度为 $O(n)$
- 空间复杂度： $O(1)$

数学规律解法

在平时的练习中，对于这种找规律的题，当我们使用直观的思路推理出一个解法之后，可以尝试从数学的角度上去分析。

例如本题，我们可以不失一般性的将规律推导为「首项」和「公差公式」。

这通常能够有效减少一些判断。

分情况讨论：

- 对于第一行和最后一行：公差为 $2 * (n - 1)$ 的等差数列，首项是 i
- 对于其他行：两个公差为 $2 * (n - 1)$ 的等差数列交替排列，首项分别是 i 和 $2 * n - i - 2$

代码：

```
class Solution {
    public String convert(String s, int r) {
        int n = s.length();
        if (n == 1 || r == 1) return s;

        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < r; i++) {
            if (i == 0 || i == r - 1) {
                int pos = i;
                int offset = 2 * (r - 1);
                while (pos < n) {
                    sb.append(s.charAt(pos));
                    pos += offset;
                }
            } else {
                int pos1 = i, pos2 = 2 * r - i - 2;
                int offset = 2 * (r - 1);
                while (pos1 < n || pos2 < n) {
                    if (pos1 < n) {
                        sb.append(s.charAt(pos1));
                        pos1 += offset;
                    }
                    if (pos2 < n) {
                        sb.append(s.charAt(pos2));
                        pos2 += offset;
                    }
                }
            }
        }
        return sb.toString();
    }
}
```

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：分别对 `s` 中的每个字符进行打印。复杂度为 $O(n)$
- 空间复杂度： $O(1)$

总结

这种“找规律”的模拟题，和小学奥数题十分类似。要提高自己的做题水平，需要坚持两个方向：

1. 自己多在纸上画图找规律，这种题没有什么通用解法
2. 多做题，尽量对每种“规律”都有所接触

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 [7. 整数反转](#)，难度为 简单。

Tag：「数学」、「模拟」

给你一个 32 位的有符号整数 x ，返回将 x 中的数字部分反转后的结果。

如果反转后整数超过 32 位的有符号整数的范围 $[-2^{31}, 2^{31} - 1]$ ，就返回 0。

假设环境不允许存储 64 位整数（有符号或无符号）。

示例 1：

输入： $x = 123$
输出：321

示例 2：

输入： $x = -123$
输出：-321

示例 3：

输入： $x = 120$
输出：21

宫水三叶
刷题日记

公众号：宫水三叶的刷题日记

示例 4：

输入：x = 0

输出：0

提示：

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

不完美解法

在机试或者周赛这种需要快速 AC 的场景中，遇到这种从文字上进行限制的题目，可以选择性的忽略限制。

对于本题，题目从文字上限制我们只能使用 32 位的数据结构（int）。

但由于数据范围过大，使用 int 会有溢出的风险，所以我们使用 long 来进行计算，在返回再转换为 int。

代码：

```
class Solution {
    public int reverse(int x) {
        long ans = 0;
        while (x != 0) {
            ans = ans * 10 + x % 10;
            x = x / 10;
        }
        return (int)ans == ans ? (int)ans : 0;
    }
}
```

- 时间复杂度：数字 x 的位数，数字大约有 $\log_{10} x$ 位。复杂度为 $O(\log_{10} x)$
- 空间复杂度： $O(1)$

刷题日记

公众号：宫水三叶的刷题日记

完美解法

在「不完美解法」中，我们使用了不符合文字限制的 long 数据结构。

接下来我们看看，不使用 long 该如何求解。

从上述解法来看，我们在循环的 `ans = ans * 10 + x % 10` 这一步会有溢出的风险，因此我们需要边遍历边判断是否溢出：

- 对于正数而言：溢出意味着 `ans * 10 + x % 10 > Integer.MAX_VALUE`，对等式进行变化后可得 `ans > (Integer.MAX_VALUE - x % 10) / 10`。所以我们可以根据此变形公式进行预判断
- 对于负数而言：溢出意味着 `ans * 10 + x % 10 < Integer.MIN_VALUE`，对等式进行变化后可得 `ans < (Integer.MIN_VALUE - x % 10) / 10`。所以我们可以根据此变形公式进行预判断

代码：

```
class Solution {
    public int reverse(int x) {
        int ans = 0;
        while (x != 0) {
            if (x > 0 && ans > (Integer.MAX_VALUE - x % 10) / 10) return 0;
            if (x < 0 && ans < (Integer.MIN_VALUE - x % 10) / 10) return 0;
            ans = ans * 10 + x % 10;
            x /= 10;
        }
        return ans;
    }
}
```

- 时间复杂度：数字 x 的位数，数字大约有 $\log_{10} x$ 位。复杂度为 $O(\log_{10} x)$
- 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 [9. 回文数](#)，难度为 简单。

公众号: 宫水三叶的刷题日记

Tag：「数学」、「回文串」

给你一个整数 x ，如果 x 是一个回文整数，返回 `true`；否则，返回 `false`。

回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

例如，121 是回文，而 123 不是。

示例 1：

输入： $x = 121$
输出：`true`

示例 2：

输入： $x = -121$
输出：`false`
解释：从左向右读，为 -121 。从右向左读，为 $121-$ 。因此它不是一个回文数。

示例 3：

输入： $x = 10$
输出：`false`
解释：从右向左读，为 01 。因此它不是一个回文数。

示例 4：

输入： $x = -101$
输出：`false`

提示：

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

进阶：你能不将整数转为字符串来解决这个问题吗？

刷题日记

公众号：宫水三叶的刷题日记

字符串解法

虽然进阶里提到了不能用字符串来解决，但还是提供一下吧。

代码：

```
class Solution {
    public boolean isPalindrome(int x) {
        String s = String.valueOf(x);
        StringBuilder sb = new StringBuilder(s);
        sb.reverse();
        return sb.toString().equals(s);
    }
}
```

- 时间复杂度：数字 n 的位数，数字大约有 \log_{10}^n 位，翻转操作要执行循环。复杂度为 $O(\log_{10}^n)$
- 空间复杂度：使用了字符串作为存储。复杂度为 $O(\log_{10}^n)$

非字符串解法（完全翻转）

原数值 x 的不超过 `int` 的表示范围，但翻转后的值会有溢出的风险，所以这里使用 `long` 接收，最后对比两者是否相等。

代码：

```
class Solution {
    public boolean isPalindrome(int x) {
        if (x < 0) return false;
        long ans = 0;
        int t = x;
        while (x > 0) {
            ans = ans * 10 + x % 10;
            x /= 10;
        }
        return ans - t == 0;
    }
}
```

- 时间复杂度：数字 n 的位数，数字大约有 \log_{10}^n 位。复杂度为 $O(\log_{10}^n)$

- 空间复杂度： $O(1)$

非字符串解法（部分翻转）

如果在进阶中增加一个我们熟悉的要求：环境中只能存储得下 32 位的有符号整数。

那么我们就连 `long` 也不能用了，这时候要充分利用「回文」的特性：前半部分和后半部分（翻转）相等。

这里的前半部分和后半部分（翻转）需要分情况讨论：

- 回文长度为奇数：回文中心是一个独立的数，即
忽略回文中心后，前半部分 == 后半部分（翻转）。如 1234321 回文串
- 回文长度为偶数：回文中心在中间两个数中间，即 前半部分 == 后半部分（翻转）。
如 123321

代码：

```
class Solution {
    public boolean isPalindrome(int x) {
        // 对于 负数 和 x0、x00、x000 格式的数，直接返回 false
        if (x < 0 || (x % 10 == 0 && x != 0)) return false;
        int t = 0;
        while (x > t) {
            t = t * 10 + x % 10;
            x /= 10;
        }
        // 回文长度的两种情况：直接比较 & 忽略中心点（t 的最后一位）进行比较
        return x == t || x == t / 10;
    }
}
```

- 时间复杂度：数字 n 的位数，数字大约有 \log_{10}^n 位。复杂度为 $O(\log_{10}^n)$
- 空间复杂度： $O(1)$

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

刷题日记

公众号: 宫水三叶的刷题日记

题目描述

这是 LeetCode 上的 [29. 两数相除](#)，难度为 中等。

Tag：「数学」、「二分」

给定两个整数，被除数 dividend 和除数 divisor。

将两数相除，要求不使用乘法、除法和 mod 运算符。

返回被除数 dividend 除以除数 divisor 得到的商。

整数除法的结果应当截去（truncate）其小数部分，例如： $\text{truncate}(8.345) = 8$ 以及 $\text{truncate}(-2.7335) = -2$

示例 1:

输入: dividend = 10, divisor = 3

输出: 3

解释: $10/3 = \text{truncate}(3.33333...) = \text{truncate}(3) = 3$

示例 2:

输入: dividend = 7, divisor = -3

输出: -2

解释: $7/-3 = \text{truncate}(-2.33333...) = -2$

提示：

- 被除数和除数均为 32 位有符号整数。
- 除数不为 0。
- 假设我们的环境只能存储 32 位有符号整数，其数值范围是 $[-2^{31}, 2^{31} - 1]$ 。本题中，如果除法结果溢出，则返回 $2^{31} - 1$ 。

刷题日记

公众号: 宫水三叶的刷题日记

二分 + 倍增乘法

由于题目限定了我们不能使用乘法、除法和 mod 运算符。

我们可以先实现一个「倍增乘法」，然后利用对于 x 除以 y ，结果 x / y 必然落在范围 $[0, x]$ 的规律进行二分。

代码：

```
class Solution {
    public int divide(int a, int b) {
        long x = a, y = b;
        boolean isNeg = false;
        if ((x > 0 && y < 0) || (x < 0 && y > 0)) isNeg = true;
        if (x < 0) x = -x;
        if (y < 0) y = -y;
        long l = 0, r = x;
        while (l < r) {
            long mid = l + r + 1 >> 1;
            if (mul(mid, y) <= x) {
                l = mid;
            } else {
                r = mid - 1;
            }
        }
        long ans = isNeg ? -l : l;
        if (ans > Integer.MAX_VALUE || ans < Integer.MIN_VALUE) return Integer.MAX_VALUE;
        return (int)ans;
    }

    long mul(long a, long k) {
        long ans = 0;
        while (k > 0) {
            if ((k & 1) == 1) ans += a;
            k >>= 1;
            a += a;
        }
        return ans;
    }
}
```

- 时间复杂度：对 x 采用的是二分策略。复杂度为 $O(\log n)$
- 空间复杂度： $O(1)$

总结

这道题的解法，主要涉及的模板有两个。

一个是「二分」模板，一个是「快速乘法」模板。都是高频使用的模板。

其中「二分」模板其实有两套，主要是根据 `check(mid)` 函数为 `true` 时，需要调整的是 `l` 指针还是 `r` 指针来判断。

- 当 `check(mid) == true` 调整的是 `l` 时：计算 `mid` 的方式应该为 `mid = l + r + 1 >> 1`：

```
long l = 0, r = 1000009;
while (l < r) {
    long mid = l + r + 1 >> 1;
    if (check(mid)) {
        l = mid;
    } else {
        r = mid - 1;
    }
}
```

- 当 `check(mid) == true` 调整的是 `r` 时：计算 `mid` 的方式应该为 `mid = l + r >> 1`：

```
long l = 0, r = 1000009;
while (l < r) {
    long mid = l + r >> 1;
    if (check(mid)) {
        r = mid;
    } else {
        l = mid + 1;
    }
}
```

另外一个「快速乘法」模板，采用了倍增的思想：

宫水三叶
の
刷题日记

公众号：宫水三叶的刷题日记

```
long mul(long a, long k) {
    long ans = 0;
    while (k > 0) {
        if ((k & 1) == 1) ans += a;
        k >>= 1;
        a += a;
    }
    return ans;
}
```

关于「二分」模板的说明

- 为啥修改左边指针 l 的时候要进行 $+1$ 操作？

「模板一」的 $+1$ 操作主要是为了避免发生「死循环」，因为 $>>$ 和 直接使用 $/$ 一样，都属于「下取整」操作。

考虑 $l = 0, r = 1$ 的简单情况，如果不 $+1$ 的话， $l + r >> 1$ 等于 $0 + 1 / 2$ ， l 仍然是 0 ，陷入死循环。

- 为啥模板不考虑 `int` 溢出问题？

事实上，二分模板确实有考虑 `int` 溢出的写法，评论区我也有提到，但是一般我们不会去用那样的模板，因为太难记了。

而且如果数据范围确实存在 `int` 溢出的情况，正确的做法是使用 `long` 数据结构，因为你无法确保只会在「二分」逻辑中进行下标运算，这里改用模板，在别的地方也可能会溢出。

使用 `long` 的说明

评论区有小伙伴提醒三叶：假设我们的环境只能存储 32 位有符号整数，其数值范围是 $[-2^{31}, 2^{31} - 1]$

我认为这个提示有两层理解含义：

1. 实现过程中完全不能使用 `long`

2. 实现过程不限制使用 long，只是解释为什么某些情况下需要我们返回 $2^{31} - 1$

在本题，我是按照第二种解释方式进行理解。

当然也可以按照第一种解释方式进行理解，在 [7. 整数反转\(简单\)](#) 中，我就提供了实现过程中不使用 long 的「完美解决」方案。可以看看 ~

不使用 long 其实十分简单，只需要将越界判断放到循环里即可，建议你动手试试 ~

 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 [31. 下一个排列](#)，难度为 中等。

Tag：「模拟」、「数学」

实现获下一个排列的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须「原地」修改，只允许使用额外常数空间。

示例 1：

输入：nums = [1,2,3]

输出：[1,3,2]

示例 2：

输入：nums = [3,2,1]

输出：[1,2,3]

示例 3：

宫水三叶
刷题日记

公众号：宫水三叶的刷题日记

输入：nums = [1,1,5]

输出：[1,5,1]

示例 4：

输入：nums = [1]

输出：[1]

提示：

- $1 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 100$

朴素解法

找下一个排列的数。大家可以先想想大脑来是怎么完成这个找数的过程的。

我们会尽可能的将低位的数字变大，这样才符合「下一个排列」的定义。

也就是从低位往高位检查，观察某一位在「下一个排列」中是否可以被更大的数代替。

那么如何判断某一位能够被更大的数代替呢？

其实就是将 k 位到低位的所有数作为候选，判断是否有更大的数可以填入 k 位中。

假设当前我们检查到 k 位，要分析第 k 位在「下一个排列」中是否能被更大的数代替。

我们会先假定高位到 k 位的数不变，在 k 位到低位中是否有比 k 位上的数更大的数，如果有说明 k 在「下一个排列」中变大。

换句话说，我们要找的第 k 位其实就是从低位到高位第一个下降的数。

...

为了更好地理解，我们结合样例来分析，假设样例为 [1,3,5,4,1]：

1. 从后往前找，找到第一个下降的位置，记为 k 。注意 k 以后的位置是降序的。在

样例中就是找到 3

2. 从 k 往后找，找到最小的比 k 要大的数。找到 4
3. 将两者交换。注意此时 k 以后的位置仍然是降序的。
4. 直接将 k 以后的部分翻转（变为升序）。

注意：如果在步骤 1 中找到头部还没找到，说明该序列已经是字典序最大的排列。按照题意，我们要将数组重新排列成最小的排列。

代码：

```
class Solution {
    public void nextPermutation(int[] nums) {
        int n = nums.length;
        int k = n - 1;
        while (k - 1 >= 0 && nums[k - 1] >= nums[k]) k--;
        if (k == 0) {
            reverse(nums, 0, n - 1);
        } else {
            int u = k;
            while (u + 1 < n && nums[u + 1] > nums[k - 1]) u++;
            swap(nums, k - 1, u);
            reverse(nums, k, n - 1);
        }
    }
    void reverse(int[] nums, int a, int b) {
        int l = a, r = b;
        while (l < r) {
            swap(nums, l++, r--);
        }
    }
    void swap(int[] nums, int a, int b) {
        int c = nums[a];
        nums[a] = nums[b];
        nums[b] = c;
    }
}
```

- 时间复杂度：对数组线性遍历。复杂度为 $O(n)$
- 空间复杂度： $O(1)$

题目描述

这是 LeetCode 上的 [42. 接雨水](#)，难度为 **困难**。

Tag：「单调栈」、「数学」

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

示例 1：



输入：height = [0,1,0,2,1,0,1,3,2,1,2,1]

输出：6

解释：上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

示例 2：

输入：height = [4,2,0,3,2,5]

输出：9

提示：

- $n == \text{height.length}$
- $0 \leq n \leq 3 \times 10^4$
- $0 \leq \text{height}[i] \leq 10^5$

宫水三叶

刷题日记

公众号：宫水三叶的刷题日记

朴素解法

对于每根柱子而言，我们只需要找出「其左边最高的柱子」和「其右边最高的柱子」。

对左右的最高柱子取一个最小值，再和当前柱子的高度做一个比较，即可得出当前位置可以接下的雨水。

同时，边缘的柱子不可能接到雨水（某一侧没有柱子）。

这样的做法属于「暴力做法」，但题目没有给数据范围，我们无法分析到底能否 AC。

唯唯诺诺交一个，过了～（好题，建议加入蓝桥杯

代码：

```
class Solution {
    public int trap(int[] height) {
        int n = height.length;
        int ans = 0;
        for (int i = 1; i < n - 1; i++) {
            int cur = height[i];

            // 获取当前位置的左边最大值
            int l = Integer.MIN_VALUE;
            for (int j = i - 1; j >= 0; j--) l = Math.max(l, height[j]);
            if (l <= cur) continue;

            // 获取当前位置的右边最大值
            int r = Integer.MIN_VALUE;
            for (int j = i + 1; j < n; j++) r = Math.max(r, height[j]);
            if (r <= cur) continue;

            // 计算当前位置可接的雨水
            ans += Math.min(l, r) - cur;
        }
        return ans;
    }
}
```

- 时间复杂度：需要处理所有非边缘的柱子，复杂度为 $O(n)$ ；对于每根柱子而言，需要往两边扫描分别找到最大值，复杂度为 $O(n)$ 。整体复杂度为 $O(n^2)$
- 空间复杂度： $O(1)$

预处理最值解法

朴素解法的思路有了，我们想想怎么优化。

事实上，任何的优化无非都是「减少重复」。

想想在朴素思路中有哪些环节比较耗时，耗时环节中又有哪些地方是重复的，可以优化的。

首先对每根柱子进行遍历，求解每根柱子可以接下多少雨水，这个 $O(n)$ 操作肯定省不了。

但在求解某根柱子可以接下多少雨水时，需要对两边进行扫描，求两侧的最大值。每一根柱子都进行这样的扫描操作，导致每个位置都被扫描了 n 次。这个过程显然是可优化的。

换句话说：我们希望通过不重复遍历的方式找到任意位置的两侧最大值。

问题转化为：给定一个数组，如何求得任意位置的左半边的最大值和右半边的最大值。

一个很直观的方案是：直接将某个位置的两侧最大值存起来。

我们可以先从两端分别出发，预处理每个位置的「左右最值」，这样可以将我们「查找左右最值」的复杂度降到 $O(1)$ 。

整体算法的复杂度也从 $O(n^2)$ 下降到 $O(n)$ 。

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 trap(int[] height) {
        int n = height.length;
        int ans = 0;
        // 由于预处理最值的时候，我们会直接访问到 height[0] 或者 height[n - 1]，因此要特判一下
        if (n == 0) return ans;

        // 预处理每个位置左边的最值
        int[] lm = new int[n];
        lm[0] = height[0];
        for (int i = 1; i < n; i++) lm[i] = Math.max(height[i], lm[i - 1]);

        // 预处理每个位置右边的最值
        int[] rm = new int[n];
        rm[n - 1] = height[n - 1];
        for (int i = n - 2; i >= 0; i--) rm[i] = Math.max(height[i], rm[i + 1]);

        for (int i = 1; i < n - 1; i++) {
            int cur = height[i];

            int l = lm[i];
            if (l <= cur) continue;

            int r = rm[i];
            if (r <= cur) continue;

            ans += Math.min(l, r) - cur;
        }
        return ans;
    }
}

```

- 时间复杂度：预处理出两个最大值数组，复杂度为 $O(n)$ ；计算每根柱子可接的雨水，复杂度为 $O(n)$ 。整体复杂度为 $O(n)$
- 空间复杂度：使用了数组存储两侧最大值。复杂度为 $O(n)$

单调栈解法

前面我们讲到，优化思路将问题转化为：给定一个数组，如何求得任意位置的左半边的最大值和右半边的最大值。

但仔细一想，其实我们并不需要找两侧最大值，只需要找到两侧最近的比当前位置高的柱子就行

了。

针对这一类找最近值的问题，有一个通用解法：单调栈。

单调栈其实就是在栈的基础上，维持一个栈内元素单调。

在这道题，由于需要找某个位置两侧比其高的柱子（只有两侧有比当前位置高的柱子，当前位置才能接雨水），我们可以维持栈内元素的单调递减。

PS. 找某侧最近一个比其大的值，使用单调栈维持栈内元素递减；找某侧最近一个比其小的值，使用单调栈维持栈内元素递增 ...

当某个位置的元素弹出栈时，例如位置 `a`，我们自然可以得到 `a` 位置两侧比 `a` 高的柱子：

- 一个是导致 `a` 位置元素弹出的柱子（`a` 右侧比 `a` 高的柱子）
- 一个是 `a` 弹栈后的栈顶元素（`a` 左侧比 `a` 高的柱子）

当有了 `a` 左右两侧比 `a` 高的柱子后，便可计算 `a` 位置可接下的雨水量。

代码：

```
class Solution {
    public int trap(int[] height) {
        int n = height.length;
        int ans = 0;
        Deque<Integer> d = new ArrayDeque<>();
        for (int i = 0; i < n; i++) {
            while (!d.isEmpty() && height[i] > height[d.peekLast()]) {
                int cur = d.pollLast();

                // 如果栈内没有元素，说明当前位置左边没有比其高的柱子，跳过
                if (d.isEmpty()) continue;

                // 左右位置，并有左右位置得出「宽度」和「高度」
                int l = d.peekLast(), r = i;
                int w = r - l + 1 - 2;
                int h = Math.min(height[l], height[r]) - height[cur];
                ans += w * h;
            }
            d.addLast(i);
        }
        return ans;
    }
}
```

- 时间复杂度：每个元素最多进栈和出栈一次。复杂度为 $O(n)$
- 空间复杂度：栈最多存储 n 个元素。复杂度为 $O(n)$

面积差值解法

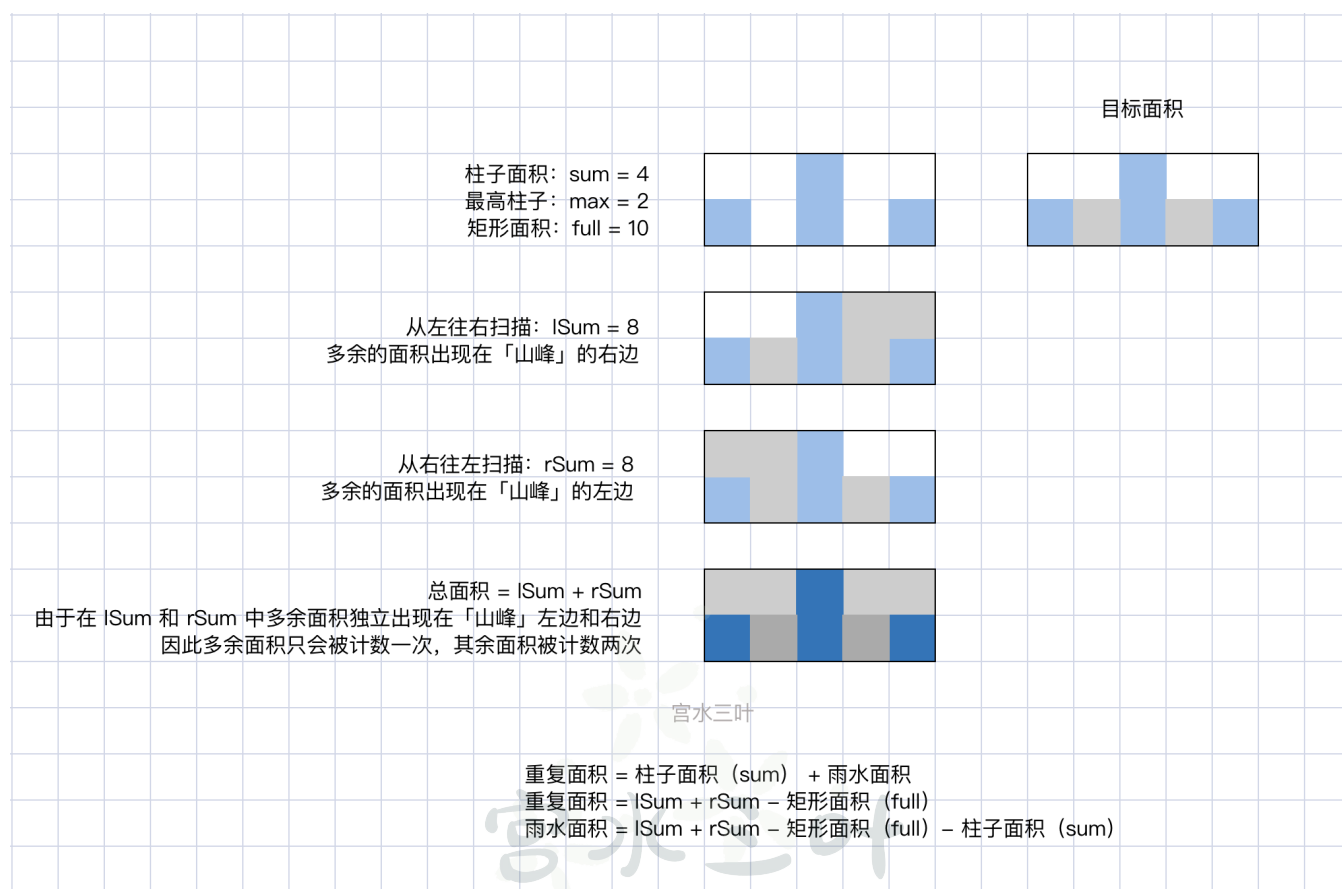
事实上，我们还能利用「面积差值」来进行求解。

我们先统计出「柱子面积」 sum 和「以柱子个数为宽、最高柱子高度为高的矩形面积」 $full$ 。

然后分别「从左往右」和「从右往左」计算一次最大高度覆盖面积 $lSum$ 和 $rSum$ 。

显然会出现重复面积，并且重复面积只会独立地出现在「山峰」的左边和右边。

利用此特性，我们可以通过简单的等式关系求解出「雨水面积」：



代码：

刷题日记

公众号: 宫水三叶的刷题日记

```

class Solution {
    public int trap(int[] height) {
        int n = height.length;

        int sum = 0, max = 0;
        for (int i = 0; i < n; i++) {
            int cur = height[i];
            sum += cur;
            max = Math.max(max, cur);
        }
        int full = max * n;

        int lSum = 0, lMax = 0;
        for (int i = 0; i < n; i++) {
            lMax = Math.max(lMax, height[i]);
            lSum += lMax;
        }

        int rSum = 0, rMax = 0;
        for (int i = n - 1; i >= 0; i--) {
            rMax = Math.max(rMax, height[i]);
            rSum += rMax;
        }

        return lSum + rSum - full - sum;
    }
}

```

- 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 **43. 字符串相乘**，难度为 **中等**。

Tag：「数学」、「模拟」

给定两个以字符串形式表示的非负整数 num1 和 num2，返回 num1 和 num2 的乘积，它们的乘积也表示为字符串形式。

示例 1:

输入: num1 = "2", num2 = "3"

输出: "6"

示例 2:

输入: num1 = "123", num2 = "456"

输出: "56088"

说明:

- num1 和 num2 的长度小于110。
- num1 和 num2 只包含数字 0-9。
- num1 和 num2 均不以零开头，除非是数字 0 本身。
- 不能使用任何标准库的大数类型（比如 BigInteger）或直接将输入转换为整数来处理。

模拟

本质上是道模拟题，模拟手算乘法的过程。

想要做出这道题，需要知道一个数学定理：

两个长度分别为 n 和 m 的数相乘，长度不会超过 $n + m$ 。

因此我们可以创建一个长度为 $n + m$ 的数组 `res` 存储结果。

另外，最后拼接结果时需要注意忽略前导零。

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 multiply(String n1, String n2) {
        int n = n1.length(), m = n2.length();
        int[] res = new int[n + m];
        for (int i = n - 1; i >= 0; i--) {
            for (int j = m - 1; j >= 0; j--) {
                int a = n1.charAt(i) - '0';
                int b = n2.charAt(j) - '0';
                int r = a * b;
                r += res[i + j + 1];
                res[i + j + 1] = r % 10;
                res[i + j] += r / 10;
            }
        }
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < n + m; i++) {
            if (sb.length() == 0 && res[i] == 0) continue;
            sb.append(res[i]);
        }
        return sb.length() == 0 ? "0" : sb.toString();
    }
}

```

- 时间复杂度：使用 `n` 和 `m` 分别代表两个数的长度。复杂度为 $O(n * m)$
- 空间复杂度：使用了长度为 `m + n` 的数组存储结果。复杂度为 $O(n + m)$

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 **149. 直线上最多的点数**，难度为 **困难**。

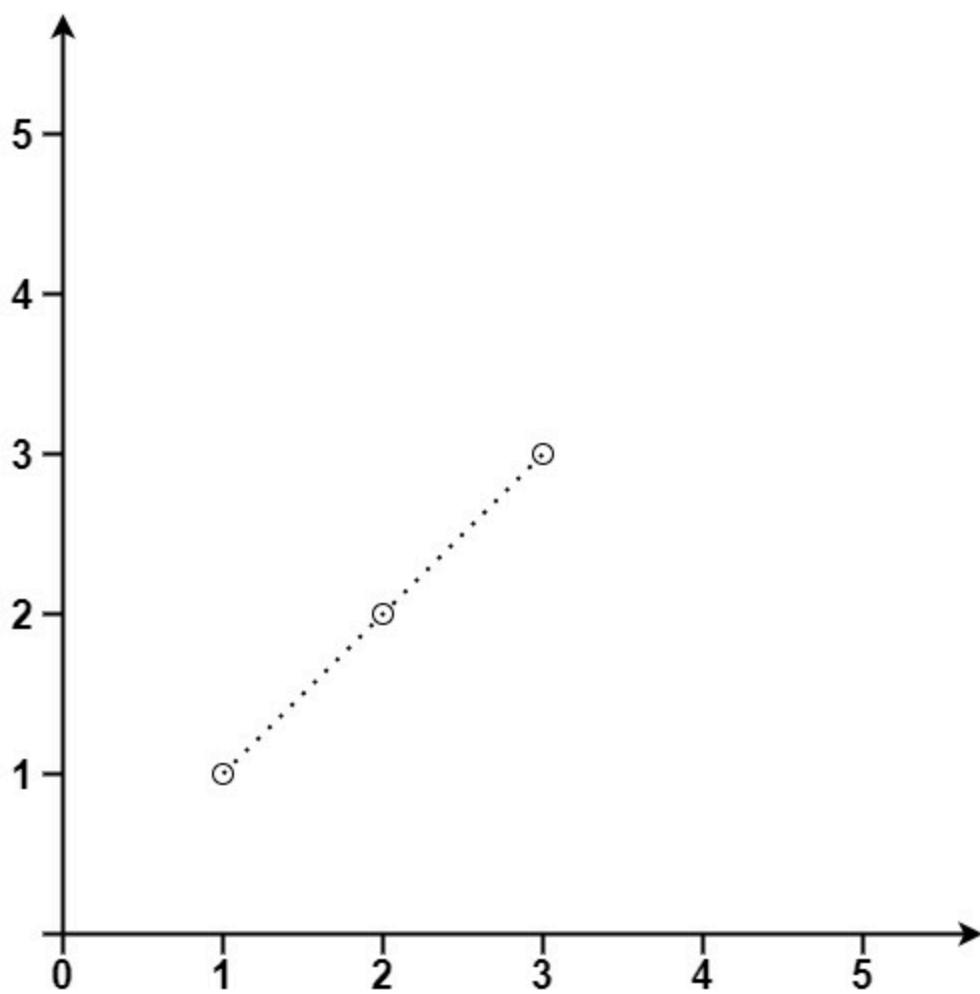
Tag：「数学」、「枚举」、「哈希表」

给你一个数组 `points`，其中 `points[i] = [xi, yi]` 表示 X-Y 平面上的一个点。求最多有多少个点在同一条直线上。

示例 1：

刷题日记

公众号: 宫水三叶的刷题日记



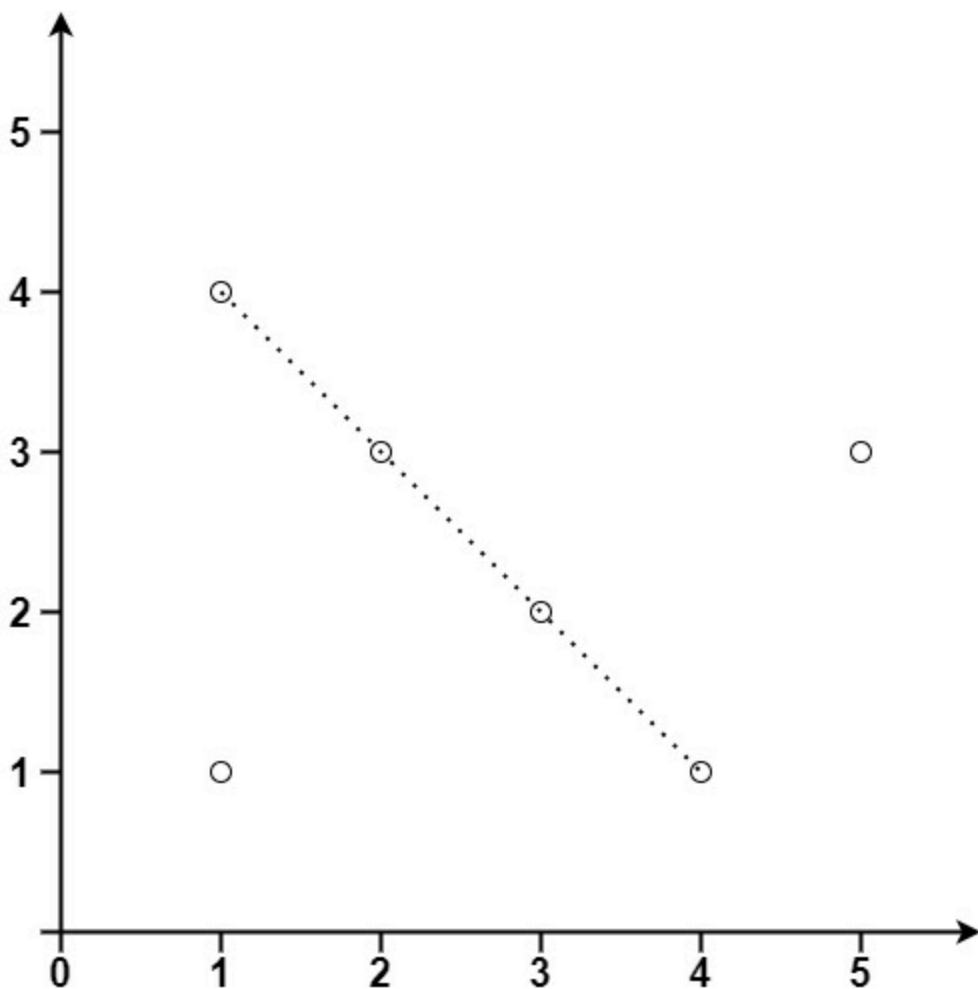
输入: `points = [[1,1],[2,2],[3,3]]`

输出: 3

示例 2:

宫水三叶
の
刷题日记

公众号: 宫水三叶的刷题日记



输入：points = [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]

输出：4

提示：

- $1 \leq \text{points.length} \leq 300$
- $\text{points}[i].\text{length} == 2$
- $-10^4 \leq x_i, y_i \leq 10^4$
- points 中的所有点 互不相同

朴素解法（枚举直线 + 枚举统计）

我们知道，两个点可以确定一条线。

因此一个朴素的做法是先枚举两条点（确定一条线），然后检查其余点是否落在该线中。

为了避免除法精度问题，当我们枚举两个点 i 和 j 时，不直接计算其对应直线的 **斜率** 和 **截距**，而是通过判断 i 和 j 与第三个点 k 形成的两条直线斜率是否相等（斜率相等的两条直线要么平行，要么重合，平行需要 4 个点来唯一确定，我们只有 3 个点，所以可以直接判定两直线重合）。

代码：

```
class Solution {
    public int maxPoints(int[][] ps) {
        int n = ps.length;
        int ans = 1;
        for (int i = 0; i < n; i++) {
            int[] x = ps[i];
            for (int j = i + 1; j < n; j++) {
                int[] y = ps[j];
                int cnt = 2;
                for (int k = j + 1; k < n; k++) {
                    int[] p = ps[k];
                    int s1 = (y[1] - x[1]) * (p[0] - y[0]);
                    int s2 = (p[1] - y[1]) * (y[0] - x[0]);
                    if (s1 == s2) cnt++;
                }
                ans = Math.max(ans, cnt);
            }
        }
        return ans;
    }
}
```

- 时间复杂度： $O(n^3)$
- 空间复杂度： $O(1)$

优化（枚举直线 + 哈希表统计）

根据「朴素解法」的思路，枚举所有直线的过程不可避免，但统计点数的过程可以优化。

具体的，我们可以先枚举所有可能出现的 **直线斜率**（根据两点确定一条直线，即枚举所有的「点对」），使用「哈希表」统计所有 **斜率** 对应的点的数量，在所有值中取个 max 即是答

案。

一些细节：在使用「哈希表」进行保存时，为了避免精度问题，我们直接使用字符串进行保存，同时需要将 **斜率** 约干净。

代码：

```
class Solution {
    public int maxPoints(int[][] ps) {
        int n = ps.length;
        int ans = 1;
        for (int i = 0; i < n; i++) {
            Map<String, Integer> map = new HashMap<>();
            // 由当前点 i 发出的直线所经过的最多点数量
            int max = 0;
            for (int j = i + 1; j < n; j++) {
                int x1 = ps[i][0], y1 = ps[i][1], x2 = ps[j][0], y2 = ps[j][1];
                int a = x1 - x2, b = y1 - y2;
                int k = gcd(a, b);
                String key = (a / k) + "_" + (b / k);
                map.put(key, map.getOrDefault(key, 0) + 1);
                max = Math.max(max, map.get(key));
            }
            ans = Math.max(ans, max + 1);
        }
        return ans;
    }
    int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }
}
```

- 时间复杂度：枚举所有直线的复杂度为 $O(n^2)$ ；令坐标值的最大差值为 m ，gcd 复杂度为 $O(\log m)$ 。整体复杂度为 $O(n^2 * \log m)$
- 空间复杂度： $O(n)$

 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 **166. 分数到小数**，难度为 **中等**。

公众号: 宫水三叶的刷题日记

Tag：「数学」、「模拟」、「哈希表」

给定两个整数，分别表示分数的分子 `numerator` 和分母 `denominator`，以字符串形式返回小数。

如果小数部分为循环小数，则将循环的部分括在括号内。

如果存在多个答案，只需返回 任意一个。

对于所有给定的输入，保证答案字符串的长度小于 10^4 。

示例 1：

输入：`numerator = 1, denominator = 2`

输出：`"0.5"`

示例 2：

输入：`numerator = 2, denominator = 1`

输出：`"2"`

示例 3：

输入：`numerator = 2, denominator = 3`

输出：`"0.(6)"`

示例 4：

输入：`numerator = 4, denominator = 333`

输出：`"0.(012)"`

示例 5：

输入：`numerator = 1, denominator = 5`

输出：`"0.2"`

宫水三叶
刷题日记

公众号：宫水三叶的刷题日记

提示：

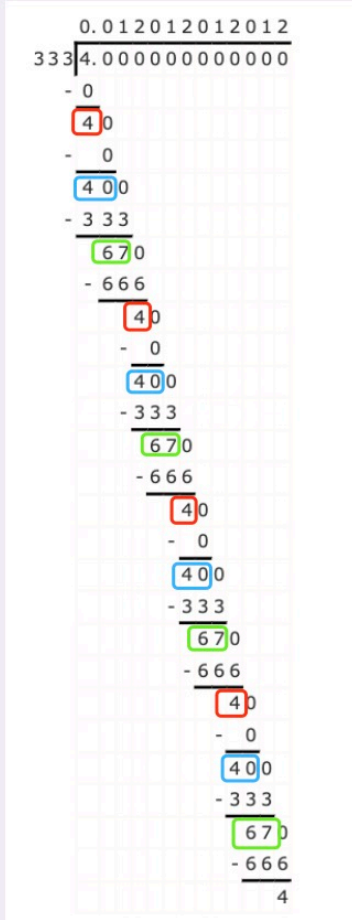
- $-2^{31} \leq \text{numerator}, \text{denominator} \leq 2^{31} - 1$
- $\text{denominator} \neq 0$

模拟

这是一道模拟 **竖式计算（除法）** 的题目。

首先可以明确，两个数相除要么是「有限位小数」，要么是「无限循环小数」，而不可能是「无限不循环小数」。

然后考虑人工计算两数相除是如何进行：



不断对余数补零（乘 10），再重新计算余数和除数的新余数
由于是一直对余数补零，往后的过程完全取决于当前余数是多少
一旦出现之前出现过的余数，说明产生了「循环小数」

宫水三叶

这引导我们可以在模拟竖式计算（除法）过程中，使用「哈希表」记录某个余数最早在什么位置出现过，一旦出现相同余数，则将「出现位置」到「当前结尾」之间的字符串抠出来，即是「循

环小数」部分。

PS. 到这里，从人工模拟除法运算的过程，我们就可以知道「为什么不会出现“无限不循环小数”」，因为始终是对余数进行补零操作，再往下进行运算，而余数个数具有明确的上限（有限集）。所以一直往下计算，最终结果要么是「出现相同余数」，要么是「余数为 0，运算结束」。

一些细节：

- 一个显然的条件是，如果本身两数能够整除，直接返回即可；
- 如果两个数有一个为“负数”，则最终答案为“负数”，因此可以起始先判断两数相乘是否小于 0，如果是，先往答案头部追加一个负号 `-`；
- 两者范围为 `int`，但计算结果可能会超过 `int` 范围，考虑 $numerator = -2^{31}$ 和 $denominator = -1$ 的情况，其结果为 2^{31} ，超出 `int` 的范围 $[-2^{31}, 2^{31} - 1]$ 。因此起始需要先使用 `long` 对两个入参类型转换一下。

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 fractionToDecimal(int numerator, int denominator) {
        // 转 long 计算，防止溢出
        long a = numerator, b = denominator;
        // 如果本身能够整除，直接返回计算结果
        if (a % b == 0) return String.valueOf(a / b);
        StringBuilder sb = new StringBuilder();
        // 如果其一为负数，先追加负号
        if (a * b < 0) sb.append('-');
        a = Math.abs(a); b = Math.abs(b);
        // 计算小数点前的部分，并将余数赋值给 a
        sb.append(String.valueOf(a / b) + ".");
        a %= b;
        Map<Long, Integer> map = new HashMap<>();
        while (a != 0) {
            // 记录当前余数所在答案的位置，并继续模拟除法运算
            map.put(a, sb.length());
            a *= 10;
            sb.append(a / b);
            a %= b;
            // 如果当前余数之前出现过，则将 [出现位置 到 当前位置] 的部分抠出来（循环小数部分）
            if (map.containsKey(a)) {
                int u = map.get(a);
                return String.format("%s(%s)", sb.substring(0, u), sb.substring(u));
            }
        }
        return sb.toString();
    }
}

```

- 时间复杂度：复杂度取决于最终答案的长度，题目规定了最大长度不会超过 10^4 ，整体复杂度为 $O(M)$
- 空间复杂度：复杂度取决于最终答案的长度，题目规定了最大长度不会超过 10^4 ，整体复杂度为 $O(M)$

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 **223. 矩形面积**，难度为 **中等**。

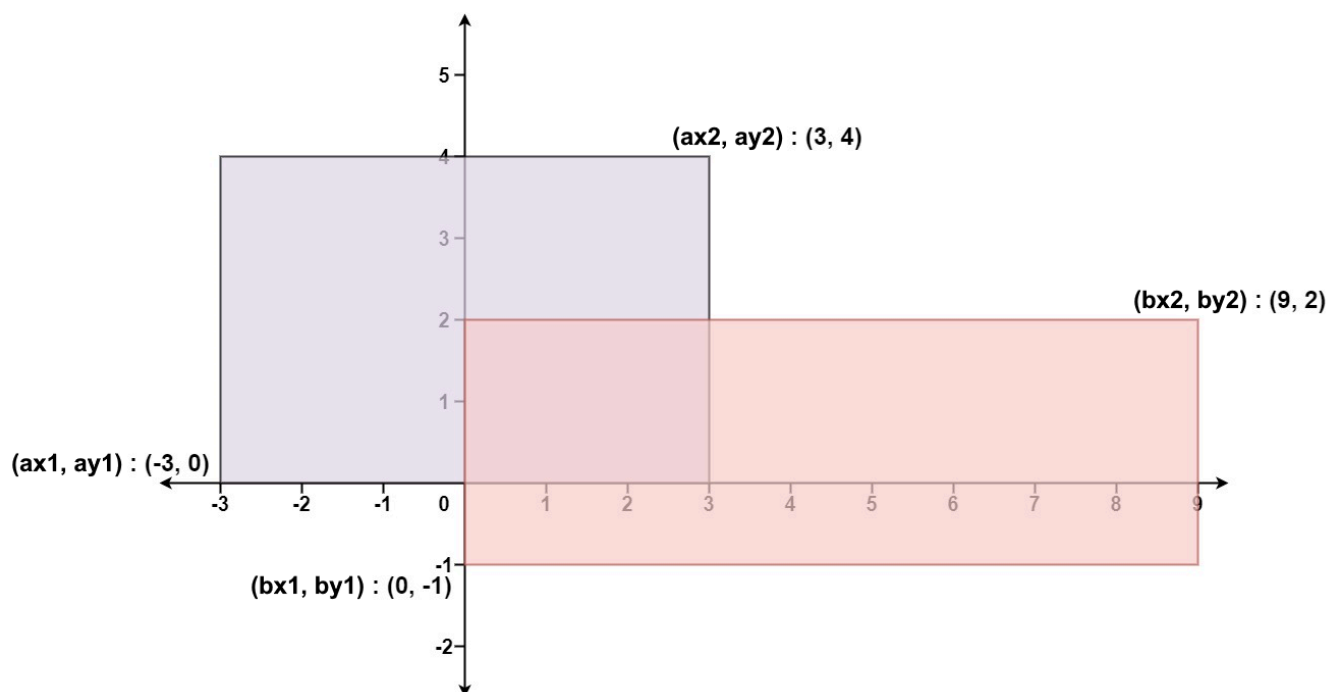
Tag：「容斥原理」

给你 二维 平面上两个 由直线构成的 矩形，请你计算并返回两个矩形覆盖的总面积。

每个矩形由其 左下 顶点和 右上 顶点坐标表示：

- 第一个矩形由其左下顶点 $(ax1, ay1)$ 和右上顶点 $(ax2, ay2)$ 定义。
- 第二个矩形由其左下顶点 $(bx1, by1)$ 和右上顶点 $(bx2, by2)$ 定义。

示例 1：



输入：ax1 = -3, ay1 = 0, ax2 = 3, ay2 = 4, bx1 = 0, by1 = -1, bx2 = 9, by2 = 2

输出：45

示例 2：

输入：ax1 = -2, ay1 = -2, ax2 = 2, ay2 = 2, bx1 = -2, by1 = -2, bx2 = 2, by2 = 2

输出：16

提示：

- $-10^4 \leq ax1, ay1, ax2, ay2, bx1, by1, bx2, by2 \leq 10^4$

容斥原理

首先在给定左下顶点和右上顶点的情况下，计算矩形面积为 $(x_2 - x_1) * (y_2 - y_1)$ 。

因此，起始时我们可以先直接算得给定的两个矩形的面积 A 和 B ，并进行累加。

剩下的，我们需要求得两矩形的交集面积，利用「容斥原理」，减去交集面积，即是答案。

求交集矩形面积，可以转换为求两矩形在坐标轴上的重合长度，若两矩形在 X 轴上的重合长度为 x ，在 Y 轴上的重合长度为 y ，则有重合面积为 $C = x * y$ 。同时考虑两矩形在任一坐标轴上没有重合长度，则不存在重合面积，因此需要将重合长度与 0 取 max。

最终答案为 $A + B - C$ 。

代码；

```
class Solution {
    public int computeArea(int ax1, int ay1, int ax2, int ay2, int bx1, int by1, int bx2,
        int x = Math.max(0, Math.min(ax2, bx2) - Math.max(ax1, bx1));
        int y = Math.max(0, Math.min(ay2, by2) - Math.max(ay1, by1));
        return (ax2 - ax1) * (ay2 - ay1) + (bx2 - bx1) * (by2 - by1) - (x * y);
    }
}
```

- 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 **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`

示例 5：

输入： $n = 5$
输出：`false`

提示：

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

进阶：你能够不使用循环/递归解决此问题吗？

宫水三叶
の
刷题日记

公众号：宫水三叶的刷题日记

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： $O(1)$

宫水三叶
刷题日记

公众号：宫水三叶的刷题日记

题目描述

这是 LeetCode 上的 [233. 数字 1 的个数](#)，难度为 **困难**。

Tag：「动态规划」、「数位 DP」、「模拟」

给定一个整数 n ，计算所有小于等于 n 的非负整数中数字 1 出现的个数。

示例 1：

输入：n = 13

输出：6

示例 2：

输入：n = 0

输出：0

提示：

$$\bullet 0 \leq n \leq 2 * 10^9$$

基本分析

这是一道经典的「数位 DP」模板题的简化版，原题在 [这里](#)。

这几天每日一题出得挺好，「序列 DP」、「区间 DP」、「数位 DP」轮着来 😊

但由于本题只需求 1 出现的次数，而不需要求解 0 到 9 的出现次数，同时意味着不需要考虑统计 0 次数时的前导零边界问题。

因此，也可以不当作数位 DP 题来做，只当作一道计数类模拟题来求解。

刷题日记

公众号: 宫水三叶的刷题日记

计数类模拟

回到本题，我们需要计算 $[1, n]$ 范围内所有数中 1 出现的次数。

我们可以统计 1 在每一位出现的次数，将其累加起来即是答案。

举个 🍌，对于一个长度为 m 的数字 n ，我们可以计算其在「个位（从右起第 1 位）」、「十位（第 2 位）」、「百位（第 3 位）」和「第 m 位」中 1 出现的次数。

假设有 $n = abcde$ ，即 $m = 5$ ，假设我们需要统计第 3 位中 1 出现的次数，即可统计满足 $— — 1 — —$ 形式，同时满足 $1 \leq — — 1 — — \leq abcde$ 要求的数有多少个，我们称 $1 \leq — — 1 — — \leq abcde$ 关系为「大小要求」。

我们只需对 c 前后出现的值进行分情况讨论：

- 当 c 前面的部分 $< ab$ ，即范围为 $[0, ab)$ ，此时必然满足「大小要求」，因此后面的部分可以任意取，即范围为 $[0, 99]$ 。根据「乘法原理」，可得知此时数量为 $ab * 100$ ；
- 当 c 前面的部分 $= ab$ ，这时候「大小关系」主要取决于 c ：
 - 当 $c = 0$ ，必然不满足「大小要求」，数量为 0；
 - 当 $c = 1$ ，此时「大小关系」取决于后部分，后面的取值范围为 $[0, de]$ ，数量为 $1 * (de + 1)$ ；
 - 当 $c > 1$ ，必然满足「大小关系」，后面的部分可以任意取，即范围为 $[0, 99]$ ，数量为 $1 * 100$ ；
- 当 c 前面的部分 $> ab$ ，必然不满足「大小要求」，数量为 0。

其他数位的分析同理。

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 int countDigitOne(int n) {
        String s = String.valueOf(n);
        int m = s.length();
        if (m == 1) return n > 0 ? 1 : 0;
        // 计算第 i 位前缀代表的数值，和后缀代表的数值
        // 例如 abcde 则有 ps[2] = ab; ss[2] = de
        int[] ps = new int[m], ss = new int[m];
        ss[0] = Integer.parseInt(s.substring(1));
        for (int i = 1; i < m - 1; i++) {
            ps[i] = Integer.parseInt(s.substring(0, i));
            ss[i] = Integer.parseInt(s.substring(i + 1));
        }
        ps[m - 1] = Integer.parseInt(s.substring(0, m - 1));
        // 分情况讨论
        int ans = 0;
        for (int i = 0; i < m; i++) {
            // x 为当前位数值，len 为当前位后面长度为多少
            int x = s.charAt(i) - '0', len = m - i - 1;
            int prefix = ps[i], suffix = ss[i];
            int tot = 0;
            tot += prefix * Math.pow(10, len);
            if (x == 0) {
            } else if (x == 1) {
                tot += suffix + 1;
            } else {
                tot += Math.pow(10, len);
            }
            ans += tot;
        }
        return ans;
    }
}

```

- 时间复杂度： $O(m)$
- 空间复杂度： $O(m)$

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 **263. 丑数**，难度为 简单。

公众号: 宫水三叶的刷题日记

Tag：「数学」、「模拟」

给你一个整数 n ，请你判断 n 是否为丑数。如果是，返回 true；否则，返回 false。

丑数 就是只包含质因数 2、3 和/或 5 的正整数。

示例 1：

输入：n = 6

输出：true

解释：6 = 2 × 3

示例 2：

输入：n = 8

输出：true

解释：8 = 2 × 2 × 2

示例 3：

输入：n = 14

输出：false

解释：14 不是丑数，因为它包含了另外一个质因数 7。

示例 4：

输入：n = 1

输出：true

解释：1 通常被视为丑数。

提示：

• $-2^{31} \leq n \leq 2^{31} - 1$

宫水三叶
の
刷题日记

公众号：宫水三叶的刷题日记

朴素解法

输入范围是 $-2^{31} \leq n \leq 2^{31} - 1$ ，我们只需要对输入进行分情况讨论即可：

- 如果 n 不是正整数（即小于等于 0）：必然不是丑数，直接返回 false。
- 如果 n 是正整数：我们对 n 执行 2 3 5 的整除操作即可，直到 n 被除干净，如果 n 最终为 1 说明是丑数，否则不是丑数。

注意，2 3 5 先除哪一个都是可以的，因为乘法本身具有交换律。

代码：

```
class Solution {
    public boolean isUgly(int n) {
        if (n <= 0) return false;
        while (n % 2 == 0) n /= 2;
        while (n % 3 == 0) n /= 3;
        while (n % 5 == 0) n /= 5;
        return n == 1;
    }
}
```

- 时间复杂度：当 n 是以 2 为底的对数时，需要除以 $\log n$ 次。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 **313. 超级丑数**，难度为 中等。

Tag：「优先队列」、「多路归并」

超级丑数 是一个正整数，并满足其所有质因数都出现在质数数组 primes 中。

给你一个整数 n 和一个整数数组 primes，返回第 n 个 超级丑数。

题目数据保证第 n 个 超级丑数 在 32-bit 带符号整数范围内。

示例 1：

输入： $n = 12$, $\text{primes} = [2, 7, 13, 19]$

输出：32

解释：给定长度为 4 的质数数组 $\text{primes} = [2, 7, 13, 19]$ ，前 12 个超级丑数序列为： $[1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32]$ 。

示例 2：

输入： $n = 1$, $\text{primes} = [2, 3, 5]$

输出：1

解释：1 不含质因数，因此它的所有质因数都在质数数组 $\text{primes} = [2, 3, 5]$ 中。

提示：

- $1 \leq n \leq 10^6$
- $1 \leq \text{primes.length} \leq 100$
- $2 \leq \text{primes}[i] \leq 1000$
- 题目数据 保证 $\text{primes}[i]$ 是一个质数
- primes 中的所有值都 互不相同，且按 递增顺序 排列

基本分析

类似的题目在之前的每日一题也出现过。

本题做法与 [264. 丑数 II](#) 类似，相关题解在 [这里](#)。

回到本题，根据丑数的定义，我们有如下结论：

- 1 是最小的丑数。
- 对于任意一个丑数 x ，其与任意给定的质因数 $\text{primes}[i]$ 相乘，结果仍为丑数。

刷题日记

公众号：宫水三叶的刷题日记

优先队列（堆）

有了基本的分析思路，一个简单的解法是使用优先队列：

1. 起始先将最小丑数 1 放入队列
2. 每次从队列取出最小值 x ，然后将 x 所对应的丑数 $x * primes[i]$ 进行入队。
3. 对步骤 2 循环多次，第 n 次出队的值即是答案。

为了防止同一丑数多次进队，我们需要使用数据结构 *Set* 来记录入过队列的丑数。

代码：

```
class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
        Set<Long> set = new HashSet<>();
        PriorityQueue<Long> q = new PriorityQueue<>();
        q.add(1L);
        set.add(1L);
        while (n-- > 0) {
            long x = q.poll();
            if (n == 0) return (int)x;
            for (int k : primes) {
                if (!set.contains(k * x)) {
                    set.add(k * x);
                    q.add(k * x);
                }
            }
        }
        return -1; // never
    }
}
```

- 时间复杂度：令 $primes$ 长度为 m ，需要从优先队列（堆）中弹出 n 个元素，每次弹出最多需要放入 m 个元素，堆中最多有 $n * m$ 个元素。复杂度为 $O(n * m \log(n * m))$
- 空间复杂度： $O(n * m)$

多路归并

从解法一中不难发现，我们「往后产生的丑数」都是基于「已有丑数」而来（使用「已有丑数」

乘上「给定质因数」 $primes[i]$)。

因此，如果我们所有丑数的有序序列为 $a_1, a_2, a_3, \dots, a_n$ 的话，序列中的每一个数都必然能够被以下三个序列（中的至少一个）覆盖（这里假设 $primes = [2, 3, 5]$ ）：

- 由丑数 * 2 所得的有序序列： $1 * 2, 2 * 2, 3 * 2, 4 * 2, 5 * 2, 6 * 2, 8 * 2 \dots$
- 由丑数 * 3 所得的有序序列： $1 * 3, 2 * 3, 3 * 3, 4 * 3, 5 * 3, 6 * 3, 8 * 3 \dots$
- 由丑数 * 5 所得的有序序列： $1 * 5, 2 * 5, 3 * 5, 4 * 5, 5 * 5, 6 * 5, 8 * 5 \dots$

我们令这些有序序列为 arr ，最终的丑数序列为 ans 。

如果 $primes$ 的长度为 m 的话，我们可以使用 m 个指针来指向这 m 个有序序列 arr 的当前下标。

显然，我们需要每次取 m 个指针中值最小的一个，然后让指针后移（即将当前序列的下一个值放入堆中），不断重复这个过程，直到我们找到第 n 个丑数。

当然，实现上，我们并不需要构造出这 m 个有序序列。

我们可以构造一个存储三元组的小根堆，三元组信息为 (val, i, idx) ：

- val ：为当前列表指针指向具体值；
- i ：代表这是由 $primes[i]$ 构造出来的有序序列；
- idx ：代表丑数下标，存在关系 $val = ans[idx] * primes[i]$ 。

起始时，我们将所有的 $(primes[i], i, 0)$ 加入优先队列（堆）中，每次从堆中取出最小元素，那么下一个该放入的元素为 $(ans[idx + 1] * primes[i], i, idx + 1)$ 。

另外，由于我们每个 arr 的指针移动和 ans 的构造，都是单调递增，因此我们可以通过与当前最后一位构造的 $ans[x]$ 进行比较来实现去重，而无须引用常数较大的 `Set` 结构。

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 nthSuperUglyNumber(int n, int[] primes) {
        int m = primes.length;
        PriorityQueue<int[]> q = new PriorityQueue<>((a,b)->a[0]-b[0]);
        for (int i = 0; i < m; i++) {
            q.add(new int[]{primes[i], i, 0});
        }
        int[] ans = new int[n];
        ans[0] = 1;
        for (int j = 1; j < n; ) {
            int[] poll = q.poll();
            int val = poll[0], i = poll[1], idx = poll[2];
            if (val != ans[j - 1]) ans[j++] = val;
            q.add(new int[]{ans[idx + 1] * primes[i], i, idx + 1});
        }
        return ans[n - 1];
    }
}

```

- 时间复杂度：需要构造长度为 n 的答案，每次构造需要往堆中取出和放入元素，堆中有 m 个元素，起始时，需要对 $primes$ 进行遍历，复杂度为 $O(m)$ 。整体复杂度为 $O(\max(m, n \log m))$
- 空间复杂度：存储 n 个答案，堆中有 m 个元素，复杂度为 $O(n + m)$

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 [326. 3的幂](#)，难度为 简单。

Tag：「数学」、「打表」

给定一个整数，写一个函数来判断它是否是 3 的幂次方。如果是，返回 *true*；否则，返回 *false*。

整数 n 是 3 的幂次方需满足：存在整数 x 使得 $n == 3^x$

示例 1：

宫水三叶
刷题日记

公众号：宫水三叶的刷题日记

输入：n = 27

输出：true

示例 2：

输入：n = 0

输出：false

示例 3：

输入：n = 9

输出：true

示例 4：

输入：n = 45

输出：false

提示：

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

数学

一个不能再朴素的做法是将 n 对 3 进行试除，直到 n 不再与 3 呈倍数关系，最后判断 n 是否为 $3^0 = 1$ 即可。

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 boolean isPowerOfThree(int n) {
        if (n <= 0) return false;
        while (n % 3 == 0) n /= 3;
        return n == 1;
    }
}
```

- 时间复杂度： $O(\log_3 n)$
- 空间复杂度： $O(1)$

倍数 & 约数

题目要求不能使用循环或递归来做，而传参 n 的数据类型为 `int`，这引导我们首先分析出 `int` 范围内的最大 3 次幂是多少，约为 $3^{19} = 1162261467$ 。

如果 n 为 3 的幂的话，那么必然满足 $n * 3^k = 1162261467$ ，即 n 与 1162261467 存在倍数关系。

因此，我们只需要判断 n 是否为 1162261467 的约数即可。

注意：这并不是快速判断 x 的幂的通用做法，当且仅当 x 为质数可用。

代码：

```
class Solution {
    public boolean isPowerOfThree(int n) {
        return n > 0 && 1162261467 % n == 0;
    }
}
```

- 时间复杂度： $O(1)$
- 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

打表

另外一个更容易想到的「不使用循环/递归」的做法是进行打表预处理。

使用 `static` 代码块，预处理出不超过 `int` 数据范围的所有 3 的幂，这样我们在跑测试样例时，就不需要使用「循环/递归」来实现逻辑，可直接 $O(1)$ 查表返回。

代码：

```
class Solution {
    static Set<Integer> set = new HashSet<>();
    static {
        int cur = 1;
        set.add(cur);
        while (cur <= Integer.MAX_VALUE / 3) {
            cur *= 3;
            set.add(cur);
        }
    }
    public boolean isPowerOfThree(int n) {
        return n > 0 && set.contains(n);
    }
}
```

- 时间复杂度：将打表逻辑交给 *OJ* 执行的话，复杂度为 $O(\log_3 C)$ ， C 固定为 2147483647；将打表逻辑放到本地执行，复杂度为 $O(1)$
- 空间复杂度： $O(\log_3 n)$

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：

```
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;
    }
}
```

```
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 [446. 等差数列划分 II - 子序列](#)，难度为 **困难**。

Tag：「动态规划」、「序列 DP」、「容斥原理」、「数学」

给你一个整数数组 `nums`，返回 `nums` 中所有等差子序列的数目。

如果一个序列中至少有三个元素，并且任意两个相邻元素之差相同，则称该序列为等差序列。

- 例如，`[1, 3, 5, 7, 9]`、`[7, 7, 7, 7]` 和 `[3, -1, -5, -9]` 都是等差序列。
- 再例如，`[1, 1, 2, 5, 7]` 不是等差序列。

数组中的子序列是从数组中删除一些元素（也可能不删除）得到的一个序列。

- 例如，`[2,5,10]` 是 `[1,2,1,2,4,1,5,10]` 的一个子序列。

题目数据保证答案是一个 32-bit 整数。

示例 1：

输入：`nums = [2,4,6,8,10]`

输出：7

解释：所有的等差子序列为：

`[2,4,6]`

`[4,6,8]`

`[6,8,10]`

`[2,4,6,8]`

`[4,6,8,10]`

`[2,4,6,8,10]`

`[2,6,10]`

示例 2：

输入：`nums = [7,7,7,7,7]`

输出：16

解释：数组中的任意子序列都是等差子序列。

提示：

- $1 \leq \text{nums.length} \leq 1000$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

基本分析

从题目描述来看，我们可以确定这是一个「序列 DP」问题，通常「序列 DP」需要 $O(n^2)$ 的时间复杂度，而某些具有特殊性质的「序列 DP」问题，例如 LIS 问题，能够配合贪心思路 + 二分做到 $O(n \log n)$ 复杂度。再看一眼数据范围为 10^3 ，基本可以确定这是一道复杂度为 $O(n^2)$ 的「序列 DP」问题。

动态规划 + 容斥原理

既然分析出是序列 DP 问题，我们可以先猜想一个基本的状态定义，看是否能够「不重不漏」的将状态通过转移计算出来。如果不行，我们再考虑引入更多的维度来进行求解。

先从最朴素的猜想出发，定义 $f[i]$ 为考虑下标不超过 i 的所有数，并且以 $nums[i]$ 为结尾的等差序列的个数。

不失一般性的 $f[i]$ 该如何转移，不难发现我们需要枚举 $[0, i - 1]$ 范围内的所有数，假设当前我们枚举到 $[0, i - 1]$ 中的位置 j ，我们可以直接算出两个位置的差值 $d = nums[i] - nums[j]$ ，但我们不知道 $f[j]$ 存储的子序列数量是差值为多少的。

同时，根据题目我们要求的是所有的等差序列的个数，而不是求差值为某个具体值 x 的等差序列的个数。换句话说，我们需要记录下所有差值的子序列个数，并求和才是答案。

因此我们的 $f[i]$ 不能是一个数，而应该是一个「集合」，该集合记录下了所有以 $nums[i]$ 为结尾，差值为所有情况的子序列的个数。

我们可以设置 $f[i] = g$ ，其中 g 为一个「集合」数据结构，我们期望在 $O(1)$ 的复杂度内查的某个差值 d 的子序列个数是多少。

这样 $f[i][j]$ 就代表了以 $nums[i]$ 为结尾，并且差值为 j 的子序列个数是多少。

当我们多引入一维进行这样的状态定义后，我们再分析一下能否「不重不漏」的通过转移计算出所有的动规值。

不失一般性的考虑 $f[i][j]$ 该如何转移，显然序列 DP 问题我们还是要枚举区间 $[0, i - 1]$ 的所有数。

和其他的「序列 DP」问题一样，枚举当前位置前面的所有位置的目的是，为了找到当前位置的数，能够接在哪一个位置的后面，形成序列。

对于本题，枚举区间 $[0, i - 1]$ 的所有数的含义是：枚举以 $nums[i]$ 为子序列结尾时，它的前一个值是什么，也就是 $nums[i]$ 接在哪个数的后面，形成等差子序列。

这样必然是可以「不重不漏」的处理到所有以 $nums[i]$ 为子序列结尾的情况的。

至于具体的状态转移方程，我们令差值 $d = nums[i] - nums[j]$ ，显然有（先不考虑长度至少为 3 的限制）：

$$f[i][d] = \sum_{j=0}^{i-1} (f[j][d] + 1)$$

含义为：在原本以 $nums[j]$ 为结尾的，且差值为 d 的子序列的基础上接上 $nums[i]$ ，再加上新的子序列 $(nums[j], nums[i])$ ，共 $f[j][d] + 1$ 个子序列。

最后对所有的哈希表的「值」对进行累加计数，就是以任意位置为结尾，长度大于 1 的等差子序列的数量 ans 。

这时候再看一眼数据范围 $-2^{31} \leq nums[i] \leq 2^{31} - 1$ ，如果从数据范围出发，使用「数组」充当集合的话，我们需要将数组开得很大，必然会爆内存。

但同时有 $1 \leq nums.length \leq 1000$ ，也就是说「最小差值」和「最大差值」之间可能相差很大，但是差值的数量是有限的，不会超过 n^2 个。

为了不引入复杂的「离散化」操作，我们可以直接使用「哈希表」来充当「集合」。

每一个 $f[i]$ 为一个哈希表，哈希表的以 `{d: cnt}` 的形式进行存储，`d` 为子序列差值，`cnt` 为子序列数量。

虽然相比使用数组，哈希表常数更大，但是经过上述分析，我们的复杂度为 $O(n^2)$ ，计算量为 10^6 ，距离计算量上界 10^7 还保有一段距离，因此直接使用哈希表十分安全。

到这里，我们解决了不考虑「长度为至少为 3」限制的原问题。

那么需要考虑「长度为至少为 3」限制怎么办？

显然，我们计算的 ans 为统计所有的「长度大于 1」的等差子序列数量，由于长度必然为正整数，也就是统计的是「长度大于等于 2」的等差子序列的数量。

因此，如果我们能够求出长度为 2 的子序列的个数的话，从 *ans* 中减去，得到的就是「长度为至少为 3」子序列的数量。

长度为 2 的等差子序列，由于没有第三个数的差值限制，因此任意的数对 (j, i) 都是一个合法的长度为 2 的等差子序列。

而求长度为 n 的数组的所有数对，其实就是求首项为 0，末项为 $n - 1$ ，公差为 1，长度为 n 的等差数列之和，直接使用「等差数列求和」公式求解即可。

代码：

```
class Solution {
    public int numberOfArithmeticSlices(int[] nums) {
        int n = nums.length;
        // 每个 f[i] 均为哈希表，哈希表键值对为 {d : cnt}
        // d : 子序列差值
        // cnt : 以 nums[i] 为结尾，且差值为 d 的子序列数量
        List<Map<Long, Integer>> f = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            Map<Long, Integer> cur = new HashMap<>();
            for (int j = 0; j < i; j++) {
                Long d = nums[i] * 1L - nums[j];
                Map<Long, Integer> prev = f.get(j);
                int cnt = cur.getOrDefault(d, 0);
                cnt += prev.getOrDefault(d, 0);
                cnt++;
                cur.put(d, cnt);
            }
            f.add(cur);
        }
        int ans = 0;
        for (int i = 0; i < n; i++) {
            Map<Long, Integer> cur = f.get(i);
            for (Long key : cur.keySet()) ans += cur.get(key);
        }
        int a1 = 0, an = n - 1;
        int cnt = (a1 + an) * n / 2;
        return ans - cnt;
    }
}
```

- 时间复杂度：DP 过程的复杂度为 $O(n^2)$ ，遍历所有的哈希表的复杂度上界不会超过 $O(n^2)$ 。整体复杂度为 $O(n^2)$
- 空间复杂度：所有哈希表存储的复杂度上界不会超过 $O(n^2)$

题目描述

这是 LeetCode 上的 [470. 用 Rand7\(\) 实现 Rand10\(\)](#)，难度为 **中等**。

Tag：「位运算」、「数学」

已有方法 rand7 可生成 1 到 7 范围内的均匀随机整数，试写一个方法 rand10 生成 1 到 10 范围内的均匀随机整数。

不要使用系统的 Math.random() 方法。

示例 1:

输入：1

输出：[7]

示例 2:

输入：2

输出：[8,4]

示例 3:

输入：3

输出：[8,1,10]

提示:

1. rand7 已定义。
2. 传入参数: n 表示 rand10 的调用次数。

进阶:

- rand7()调用次数的期望值是多少？

- 你能否尽量少调用 `rand7()` ？

基本分析

给定一个随机生成 $1 \sim 7$ 的函数，要求实现等概率返回 $1 \sim 10$ 的函数。

首先需要知道，在输出域上进行定量整体偏移，仍然满足等概率，即要实现 $0 \sim 6$ 随机器，只需要在 `rand7` 的返回值上进行 -1 操作即可。

但输出域的 拼接/叠加 并不满足等概率。例如 `rand7() + rand7()` 会产生 $[2, 14]$ 范围内的数，但每个数并非等概率：

- 产生 2 的概率为： $\frac{1}{7} * \frac{1}{7} = \frac{1}{49}$
- 产生 4 的概率为： $\frac{1}{7} * \frac{1}{7} + \frac{1}{7} * \frac{1}{7} + \frac{1}{7} * \frac{1}{7} = \frac{3}{49}$

在 $[2, 14]$ 这 13 个数里面，等概率的数值不足 10 个。

因此，你应该知道「执行两次 `rand7()` 相加，将 $[1, 10]$ 范围内的数进行返回，否则一直重试」的做法是错误的。

k 进制诸位生成 + 拒绝采样

上述做法出现概率分布不均的情况，是因为两次随机值的不同组合「相加」的会出现相同的结果（ $(1, 3)$ 、 $(2, 2)$ 、 $(3, 1)$ 最终结果均为 4）。

结合每次执行 `rand7` 都可以看作一次独立事件。我们可以将两次 `rand7` 的结果看作生成 7 进制的两位。从而实现每个数值都唯一对应了一种随机值的组合（等概率），反之亦然。

举个🍌，设随机执行两次 `rand7` 得到的结果分别是 4（第一次）、7（第二次），由于我们要 7 进制的数，因此可以先对 `rand7` 的执行结果进行 -1 操作，将输出域偏移到 $[0, 6]$ （仍为等概率），即得到 3（第一次）和 6（第二次），最终得到的是数值 $(63)_7$ ，数值 $(63)_7$ 唯一对应了我们的随机值组合方案，反过来随机值组合方案也唯一对应一个 7 进制的数值。

那么根据「进制转换」的相关知识，如果我们存在一个 `randK` 的函数，对其执行 n 次，我们能够等概率产生 $[0, K^n - 1]$ 范围内的数值。

回到本题，执行一次 `rand7` 只能产生 $[0, 6]$ 范围内的数值，不足 10 个；而执行 2 次 `rand7` 的话则能产生 $[0, 48]$ 范围内的数值，足够 10 个，且等概率。

我们只需要判定生成的值是否为题意的 $[1, 10]$ 即可，如果是的话直接返回，否则一直重试。

代码：

```
class Solution extends SolBase {
    public int rand10() {
        while (true) {
            int ans = (rand7() - 1) * 7 + (rand7() - 1); // 进制转换
            if (1 <= ans && ans <= 10) return ans;
        }
    }
}
```

- 时间复杂度：期望复杂度为 $O(1)$ ，最坏情况下为 $O(\infty)$
- 空间复杂度： $O(1)$

进阶

1. 降低对 `rand7` 的调用次数

我们发现，在上述解法中，范围 $[0, 48]$ 中，只有 $[1, 10]$ 范围内的数据会被接受返回，其余情况均被拒绝重试。

为了尽可能少的调用 `rand7` 方法，我们可以从 $[0, 48]$ 中取与 $[1, 10]$ 成倍数关系的数，来进行转换。

我们可以取 $[0, 48]$ 中的 $[1, 40]$ 范围内的数来代指 $[1, 10]$ 。

首先在 $[0, 48]$ 中取 $[1, 40]$ 仍为等概率，其次形如 $x1$ 的数值有 4 个（1、11、21、31），形如 $x2$ 的数值有 4 个（2、12、22、32）... 因此最终结果仍为等概率。

代码：

宫水三叶
刷题日记

公众号：宫水三叶的刷题日记

```
class Solution extends SolBase {
    public int rand10() {
        while (true) {
            int ans = (rand7() - 1) * 7 + (rand7() - 1); // 进制转换
            if (1 <= ans && ans <= 40) return ans % 10 + 1;
        }
    }
}
```

- 时间复杂度：期望复杂度为 $O(1)$ ，最坏情况下为 $O(\infty)$
- 空间复杂度： $O(1)$

2. 计算 rand7 的期望调用次数

在 $[0, 48]$ 中我们采纳了 $[1, 40]$ 范围内的数值，即以调用两次为基本单位的话，有 $\frac{40}{49}$ 的概率被接受返回（成功）。

成功的概率为 $\frac{40}{49}$ ，那么需要触发成功所需次数（期望次数）为其倒数 $\frac{49}{40} = 1.225$ ，每次会调用两次 rand7，因而总的期望调用次数为 $1.225 * 2 = 2.45$ 。

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：

HammingDistance(4, 14) + HammingDistance(4, 2) + HammingDistance(14, 2) = 2 + 2 + 2 = 6

示例 2：

```
输入：nums = [4,14,4]
```

```
输出：4
```

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。

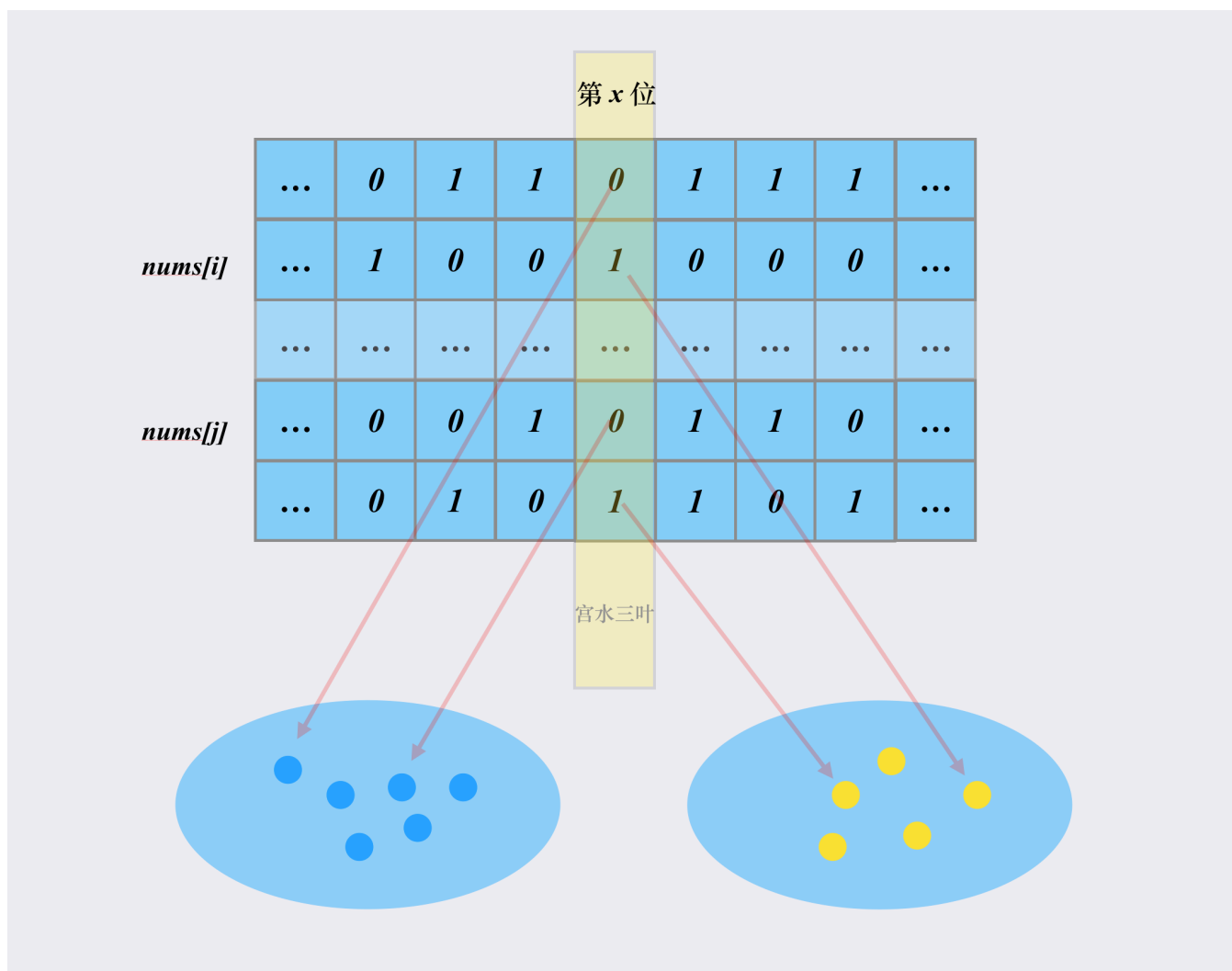
我们考虑某个 $calc(x)$ 如何求得：

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 **483. 最小好进制**，难度为 **困难**。

Tag：「数学」、「推公式」

对于给定的整数 n ，如果 n 的 k ($k \geq 2$) 进制数的所有数位全为 1，则称 k ($k \geq 2$) 是 n 的一个好进制。

以字符串的形式给出 n ，以字符串的形式返回 n 的最小好进制。

示例 1：

输入："13"

输出："3"

解释：13 的 3 进制是 111。

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："4681"

输出："8"

解释：4681 的 8 进制是 11111。

示例 3：

输入："1000000000000000000"

输出："999999999999999999"

解释：1000000000000000000 的 9999999999999999 进制是 11。

提示：

- n 的取值范围是 $[3, 10^{18}]$ 。
- 输入总是有效且没有前导 0。

基本分析

设 $(n)_{10}$ 的 k 进制表示共有 Len 位，那么根据「进制转换」相关知识，必然有如下等式：

$$(n)_{10} = (11\dots 11)_k = k^0 + k^1 + k^2 + \dots + k^{Len-1}$$

当 n 给定的情况下， k 随着 Len 减小而增大，由此我们可以分析出 k 的上界：

- 当 Len 取 1 的时候： $k^0 = n$ ，即 $n = 1$ ，与题目给定的数据范围冲突，不可取；
- 当 Len 取 2 的时候： $k^0 + k^1 = n$ ，即 $k = n - 1$ ，为合法值。

因此 k 的上界为 $n - 1$ ，同时我们知道长度 Len 满足等式 $Len \geq 2$ 。

然后我们再分析一下 Len 的上界。

根据 k 与 Len 大小变化关系，同时已知 $k \geq 2$ ，不难分析当 k 取最小值 2 的时候， Len 可得最大值（同时 n 的最大值为 10^{18} ），可分析出 $Len \leq \lceil \log_2 n \rceil$ ， $\log_2 10^{18}$ 不超过 60。

因此可以采取枚举 Len 的做法。

刷题日记

公众号：宫水三叶的刷题日记

枚举 Len

根据分析，我们可以在 $[2, 60]$ 范围内从大到小枚举 Len ，当取得第一位合法的 Len 时， k 为最小合法值。

剩下的问题在于如何在给定 Len 的情况下，求得 k 为多少。

前面分析到 $Len \geq 2$ ，令 $s = Len - 1$ ，再根据 [二项式定理](#) 可得：

$$n = k^0 + k^1 + k^2 + \dots + k^s < C_s^0 * k^0 + C_s^1 * k^1 + \dots + C_s^s * k^s = (k + 1)^s$$

同时结合 $n > k^s$ ，可得 $k^s < n < (k + 1)^s$ ，整理后可得：

$$k < n^{\frac{1}{s}} < k + 1$$

因此，对于任意的 $s = Len - 1$ 都有唯一的解为 $n^{\frac{1}{s}}$ （正整数），我们只需要验证 $n^{\frac{1}{s}}$ 是否为正整数即可。

一些细节：实现上为了方便，不处理 $k = n - 1$ 的边界问题，我们可以调整枚举下界为 3，当枚举不出合法 k 时，直接返回 $n - 1$ 作为答案。

代码：

```
class Solution {
    public String smallestGoodBase(String n) {
        long m = Long.parseLong(n);
        int max = (int)(Math.log(m) / Math.log(2) + 1);
        for (int len = max; len >= 3; len--) {
            long k = (long)Math.pow(m, 1.0 / (len - 1));
            long res = 0;
            for (int i = 0; i < len; i++) res = res * k + 1;
            if (res == m) return String.valueOf(k);
        }
        return String.valueOf(m - 1);
    }
}
```

- 时间复杂度： $O(\log^2 n)$
- 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

题目描述

这是 LeetCode 上的 [523. 连续子数组和](#)，难度为 **中等**。

Tag：「前缀和」

给你一个整数数组 `nums` 和一个整数 `k`，编写一个函数来判断该数组是否含有同时满足下述条件的连续子数组：

- 子数组大小 至少为 2，且
- 子数组元素总和为 `k` 的倍数。

如果存在，返回 `true`；否则，返回 `false`。

如果存在一个整数 `n`，令整数 `x` 符合 $x = n * k$ ，则称 `x` 是 `k` 的一个倍数。

示例 1：

```
输入：nums = [23,2,4,6,7], k = 6
输出：true
解释：[2,4] 是一个大小为 2 的子数组，并且和为 6。
```

示例 2：

```
输入：nums = [23,2,6,4,7], k = 6
输出：true
解释：[23, 2, 6, 4, 7] 是大小为 5 的子数组，并且和为 42。
42 是 6 的倍数，因为  $42 = 7 * 6$  且 7 是一个整数。
```

示例 3：

```
输入：nums = [23,2,6,4,7], k = 13
输出：false
```

提示：

- $1 \leq \text{nums.length} \leq 10^5$
- $0 \leq \text{nums}[i] \leq 10^9$

宫水三叶
刷题日记

公众号：宫水三叶的刷题日记

- $0 \leq \text{sum}(\text{nums}[i]) \leq 2^{31} - 1$
- $1 \leq k \leq 2^{31} - 1$

基本分析

这是一道很经典的前缀和题目，类似的原题也在蓝桥杯出现过，坐标在 [K 倍区间](#)。

本题与那道题不同在于：

- [K 倍区间] 需要求得所有符合条件的区间数量；本题需要判断是否存在。
- [K 倍区间] 序列全是正整数，不需要考虑 0 值问题；本题需要考虑 0 值问题。

数据范围为 10^4 ，因此无论是纯朴素的做法 ($O(n^3)$) 还是简单使用前缀和优化的做法 ($O(n^2)$) 都不能满足要求。

我们需要从 k 的倍数作为切入点来做。

预处理前缀和数组 sum ，方便快速求得某一段区间的和。然后假定 $[i, j]$ 是我们的目标区间，那么有：

$$\text{sum}[j] - \text{sum}[i - 1] = n * k$$

经过简单的变形可得：

$$\frac{\text{sum}[j]}{k} - \frac{\text{sum}[i - 1]}{k} = n$$

要使得两者除 k 相减为整数，需要满足 $\text{sum}[j]$ 和 $\text{sum}[i - 1]$ 对 k 取余相同。

也就是说，我们只需要枚举右端点 j ，然后在枚举右端点 j 的时候检查之前是否出现过左端点 i ，使得 $\text{sum}[j]$ 和 $\text{sum}[i - 1]$ 对 k 取余相同。

前缀和 + HashSet

具体的，使用 `HashSet` 来保存出现过的值。

让循环从 2 开始枚举右端点（根据题目要求，子数组长度至少为 2），每次将符合长度要求的

位置的取余结果存入 `HashSet` 。

如果枚举某个右端点 j 时发现存在某个左端点 i 符合要求，则返回 `True` 。

代码：

```
class Solution {
    public boolean checkSubarraySum(int[] nums, int k) {
        int n = nums.length;
        int[] sum = new int[n + 1];
        for (int i = 1; i <= n; i++) sum[i] = sum[i - 1] + nums[i - 1];
        Set<Integer> set = new HashSet<>();
        for (int i = 2; i <= n; i++) {
            set.add(sum[i - 2] % k);
            if (set.contains(sum[i] % k)) return true;
        }
        return false;
    }
}
```

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

拓展（求方案数）

蓝桥杯官网登录经常性罢工，我登录十几次都没登录上去，这里直接截图了 [K 倍区间] 的题目给大家。

宫水三叶
の
刷题日记

公众号：宫水三叶的刷题日记

问题描述

给定一个长度为 N 的数列， A_1, A_2, \dots, A_N ，如果其中一段连续的子序列 $A_i, A_{i+1}, \dots, A_j (i \leq j)$ 之和是 K 的倍数，我们就称这个区间 $[i, j]$ 是 K 倍区间。

你能求出数列中总共有多少个 K 倍区间吗？

输入格式

第一行包含两个整数 N 和 K 。 $(1 \leq N, K \leq 100000)$

以下 N 行每行包含一个整数 A_i 。 $(1 \leq A_i \leq 100000)$

输出格式

输出一个整数，代表 K 倍区间的数目。

样例输入

5 2

18

2

3

4

5

样例输出

6

写了代码，但很可惜没 OJ 可以测试 🤔

比较简单，应该没啥问题，可以直接参考 😊

代码：

宫水三叶
の
刷题日记

公众号：宫水三叶的刷题日记

```

import java.util.*;
class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt(), k = sc.nextInt();
        long[] s = new long[n + 1];
        for (int i = 1; i <= n; i++) s[i] = s[i - 1] + sc.nextLong();
        long ans = 0;
        Map<Long, Integer> map = new HashMap<>();
        map.put(0L, 1);
        for (int i = 1; i <= n; i++) {
            long u = s[i] % k;
            if (map.containsKey(u)) ans += map.get(u);
            map.put(u, map.getOrDefault(u, 0) + 1);
        }
        System.out.println(ans);
    }
}

```

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

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 **552. 学生出勤记录 II**，难度为 **困难**。

Tag：「动态规划」、「状态机」、「记忆化搜索」、「矩阵快速幂」、「数学」

可以用字符串表示一个学生的出勤记录，其中的每个字符用来标记当天的出勤情况（缺勤、迟到、到场）。

记录中只含下面三种字符：

- 'A'：Absent，缺勤
- 'L'：Late，迟到
- 'P'：Present，到场

如果学生能够同时满足下面两个条件，则可以获得出勤奖励：

公众号: 宫水三叶的刷题日记

- 按**总出勤**计，学生缺勤（‘A’）严格 少于两天。
- 学生**不会存在** 连续 3 天或 连续 3 天以上的迟到（‘L’）记录。

给你一个整数 n ，表示出勤记录的长度（次数）。请你返回记录长度为 n 时，可能获得出勤奖励的记录情况 数量。

答案可能很大，所以返回对 $10^9 + 7$ 取余的结果。

示例 1：

输入： $n = 2$

输出：8

解释：

有 8 种长度为 2 的记录将被视为可奖励：

"PP" , "AP" , "PA" , "LP" , "PL" , "AL" , "LA" , "LL"

只有"AA"不会被视为可奖励，因为缺勤次数为 2 次（需要少于 2 次）。

示例 2：

输入： $n = 1$

输出：3

示例 3：

输入： $n = 10101$

输出：183236316

提示：

- $1 \leq n \leq 10^5$

基本分析

根据题意，我们知道一个合法的方案中 A 的总出现次数最多为 1 次，L 的连续出现次数最多

为 2 次。

因此在枚举/统计合法方案的个数时，当我们决策到某一位应该选什么时，我们关心的是当前方案中已经出现了多少个 **A**（以决策当前能否填入 **A**）以及连续出现的 **L** 的次数是多少（以决策当前能否填入 **L**）。

记忆化搜索

枚举所有方案的爆搜 **DFS** 代码不难写，大致的函数签名设计如下：

```
/**
 * @param u 当前还剩下多少位需要决策
 * @param acnt 当前方案中 A 的总出现次数
 * @param lcnt 当前方案中结尾 L 的连续出现次数
 * @param cur 当前方案
 * @param ans 结果集
 */
void dfs(int u, int acnt, int lcnt, String cur, List<String> ans);
```

实际上，我们不需要枚举所有的方案数，因此我们只需要保留函数签名中的前三个参数即可。

同时由于我们在计算某个 $(u, acnt, lcnt)$ 的方案数时，其依赖的状态可能会被重复使用，考虑加入记忆化，将结果缓存起来。

根据题意， n 的取值范围为 $[0, n]$ ， $acnt$ 取值范围为 $[0, 1]$ ， $lcnt$ 取值范围为 $[0, 2]$ 。

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;
    int[][][] cache;
    public int checkRecord(int n) {
        cache = new int[n + 1][2][3];
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j < 2; j++) {
                for (int k = 0; k < 3; k++) {
                    cache[i][j][k] = -1;
                }
            }
        }
        return dfs(n, 0, 0);
    }
    int dfs(int u, int acnt, int lcnt) {
        if (acnt >= 2) return 0;
        if (lcnt >= 3) return 0;
        if (u == 0) return 1;
        if (cache[u][acnt][lcnt] != -1) return cache[u][acnt][lcnt];
        int ans = 0;
        ans = dfs(u - 1, acnt + 1, 0) % mod; // A
        ans = (ans + dfs(u - 1, acnt, lcnt + 1)) % mod; // L
        ans = (ans + dfs(u - 1, acnt, 0)) % mod; // P
        cache[u][acnt][lcnt] = ans;
        return ans;
    }
}

```

- 时间复杂度：有 $O(n * 2 * 3)$ 个状态需要被计算，复杂度为 $O(n)$
- 空间复杂度： $O(n)$

状态机 DP

通过记忆化搜索的分析我们发现，当我们在决策下一位是什么的时候，依赖于前面已经填入的 A 的个数以及当前结尾处的 L 的连续出现次数。

也就是说，状态 $f[u][acnt][lcnt]$ 必然被某些特定状态所更新，或者说由 $f[u][acnt][lcnt]$ 出发，所能更新的状态是固定的。

因此这其实是一个状态机模型的 DP 问题。

根据「更新 $f[u][acnt][lcnt]$ 需要哪些状态值」还是「从 $f[u][acnt][lcnt]$ 出发，能够更新哪些状态」，我们能够写出两种方式（方向）的 DP 代码：

一类是从 $f[u][acnt][lcnt]$ 往回找所依赖的状态；一类是从 $f[u][acnt][lcnt]$ 出发往前去更新所能更新的状态值。

无论是何种方式（方向）的 DP 实现都只需搞清楚「当前位的选择对 $acnt$ 和 $lcnt$ 的影响」即可。

代码：

宫水三叶
の
刷题日记

公众号：宫水三叶的刷题日记

// 从 f[u][acnt][lcnt] 往回找所依赖的状态

```
class Solution {
    int mod = (int)1e9+7;
    public int checkRecord(int n) {
        int[][][] f = new int[n + 1][2][3];
        f[0][0][0] = 1;
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < 2; j++) {
                for (int k = 0; k < 3; k++) {
                    if (j == 1 && k == 0) { // A
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j - 1][0]) % mod;
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j - 1][1]) % mod;
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j - 1][2]) % mod;
                    }
                    if (k != 0) { // L
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j][k - 1]) % mod;
                    }
                    if (k == 0) { // P
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j][0]) % mod;
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j][1]) % mod;
                        f[i][j][k] = (f[i][j][k] + f[i - 1][j][2]) % mod;
                    }
                }
            }
        }
        int ans = 0;
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 3; k++) {
                ans += f[n][j][k];
                ans %= mod;
            }
        }
        return ans;
    }
}
```

宫水三叶
の
刷题日记

公众号: 宫水三叶的刷题日记

```

// 从 f[u][acnt][lcnt] 出发往前去更新所能更新的状态值
class Solution {
    int mod = (int)1e9+7;
    public int checkRecord(int n) {
        int[][][] f = new int[n + 1][2][3];
        f[0][0][0] = 1;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < 2; j++) {
                for (int k = 0; k < 3; k++) {
                    if (j != 1) f[i + 1][j + 1][0] = (f[i + 1][j + 1][0] + f[i][j][k]) % mod;
                    if (k != 2) f[i + 1][j][k + 1] = (f[i + 1][j][k + 1] + f[i][j][k]) % mod;
                    f[i + 1][j][0] = (f[i + 1][j][0] + f[i][j][k]) % mod; // P
                }
            }
        }
        int ans = 0;
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 3; k++) {
                ans += f[n][j][k];
                ans %= mod;
            }
        }
        return ans;
    }
}

```

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

矩阵快速幂

之所以在动态规划解法中强调更新状态的方式（方向）是「往回」还是「往前」，是因为对于存在线性关系（同时又具有结合律）的递推式，我们能够通过「矩阵快速幂」来进行加速。

矩阵快速幂的基本分析之前在 [\(题解\) 1137. 第 N 个泰波那契数](#) 详细讲过。

由于 *acnt* 和 *lcnt* 的取值范围都很小，其组合的状态只有 $2 * 3 = 6$ 种，我们使用 $idx = acnt * 3 + lcnt$ 来代指组合（通用的二维转一维方式）：

- $idx = 0$: $acnt = 0$ 、 $lcnt = 0$;
- $idx = 1$: $acnt = 1$ 、 $lcnt = 0$;

...
 • $idx = 5 : acnt = 1 \setminus lcnt = 2 ;$

最终答案为 $ans = \sum_{idx=0}^5 f[n][idx]$ ，将答案依赖的状态整理成列向量：

$$g[n] = \begin{bmatrix} f[n][0] \\ f[n][1] \\ f[n][2] \\ f[n][3] \\ f[n][4] \\ f[n][5] \end{bmatrix}$$

根据状态机逻辑，可得：

$$g[n] = \begin{bmatrix} f[n][0] \\ f[n][1] \\ f[n][2] \\ f[n][3] \\ f[n][4] \\ f[n][5] \end{bmatrix} = \begin{bmatrix} f[n-1][0] * 1 + f[n-1][1] * 1 + f[n-1][2] * 1 + f[n-1][3] * 0 + f[n-1][4] * 0 + f[n-1][5] * 0 \\ f[n-1][0] * 1 + f[n-1][1] * 0 + f[n-1][2] * 0 + f[n-1][3] * 0 + f[n-1][4] * 1 + f[n-1][5] * 1 \\ f[n-1][0] * 0 + f[n-1][1] * 1 + f[n-1][2] * 0 + f[n-1][3] * 0 + f[n-1][4] * 1 + f[n-1][5] * 0 \\ f[n-1][0] * 1 + f[n-1][1] * 1 + f[n-1][2] * 1 + f[n-1][3] * 1 + f[n-1][4] * 0 + f[n-1][5] * 0 \\ f[n-1][0] * 0 + f[n-1][1] * 0 + f[n-1][2] * 0 + f[n-1][3] * 1 + f[n-1][4] * 0 + f[n-1][5] * 1 \\ f[n-1][0] * 0 + f[n-1][1] * 0 + f[n-1][2] * 0 + f[n-1][3] * 0 + f[n-1][4] * 0 + f[n-1][5] * 0 \end{bmatrix}$$

我们令：

$$mat = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

根据「矩阵乘法」即有：

$$g[n] = mat * g[n-1]$$

起始时，我们只有 $g[0]$ ，根据递推式得：

$$g[n] = mat * mat * \dots * mat * g[0]$$

再根据矩阵乘法具有「结合律」，最终可得：

$$g[n] = mat^n * g[0]$$

计算 mat^n 可以套用「快速幂」进行求解。

代码：

```
class Solution {
    int N = 6;
    int mod = (int)1e9+7;
    long[][] mul(long[][] a, long[][] b) {
        int r = a.length, c = b[0].length, z = b.length;
        long[][] ans = new long[r][c];
        for (int i = 0; i < r; i++) {
            for (int j = 0; j < c; j++) {
                for (int k = 0; k < z; k++) {
                    ans[i][j] += a[i][k] * b[k][j];
                    ans[i][j] %= mod;
                }
            }
        }
        return ans;
    }
    public int checkRecord(int n) {
        long[][] ans = new long[][]{
            {1}, {0}, {0}, {0}, {0}, {0}
        };
        long[][] mat = new long[][]{
            {1, 1, 1, 0, 0, 0},
            {1, 0, 0, 0, 0, 0},
            {0, 1, 0, 0, 0, 0},
            {1, 1, 1, 1, 1, 1},
            {0, 0, 0, 1, 0, 0},
            {0, 0, 0, 0, 1, 0}
        };
        while (n != 0) {
            if ((n & 1) != 0) ans = mul(mat, ans);
            mat = mul(mat, mat);
            n >>= 1;
        }
        int res = 0;
        for (int i = 0; i < N; i++) {
            res += ans[i][0];
            res %= mod;
        }
        return res;
    }
}
```

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： $O(\log n)$
- 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 [633. 平方数之和](#)，难度为 **中等**。

Tag：「数学」、「双指针」

给定一个非负整数 c ，你要判断是否存在两个整数 a 和 b ，使得 $a^2 + b^2 = c$ 。

示例 1：

输入： $c = 5$

输出：`true`

解释： $1 * 1 + 2 * 2 = 5$

示例 2：

输入： $c = 3$

输出：`false`

示例 3：

输入： $c = 4$

输出：`true`

示例 4：

输入： $c = 2$

输出：`true`

宫水三叶
刷题日记

公众号：宫水三叶的刷题日记

示例 5：

输入：c = 1

输出：true

提示：

- $0 \leq c \leq 2^{31} - 1$

基本分析

根据等式 $a^2 + b^2 = c$ ，可得知 **a** 和 **b** 的范围均为 $[0, \sqrt{c}]$ 。

基于此我们会有以下几种做法。

枚举

我们可以枚举 **a**，边枚举边检查是否存在 **b** 使得等式成立。

这样做的复杂度为 $O(\sqrt{c})$ 。

代码：

```
class Solution {
    public boolean judgeSquareSum(int c) {
        int max = (int) Math.sqrt(c);
        for (int a = 0; a <= max; a++) {
            int b = (int) Math.sqrt(c - a * a);
            if (a * a + b * b == c) return true;
        }
        return false;
    }
}
```

- 时间复杂度： $O(\sqrt{c})$
- 空间复杂度： $O(1)$

宫水三叶
の

刷题日记

公众号：宫水三叶的刷题日记

双指针

由于 a 和 b 的范围均为 $[0, \sqrt{c}]$ ，因此我们可以使用「双指针」在 $[0, \sqrt{c}]$ 范围进行扫描：

- $a^2 + b^2 == c$ ：找到符合条件的 a 和 b ，返回 *true*
- $a^2 + b^2 < c$ ：当前值比目标值要小， $a++$
- $a^2 + b^2 > c$ ：当前值比目标值要大， $b--$

代码：

```
class Solution {
    public boolean judgeSquareSum(int c) {
        int a = 0, b = (int)Math.sqrt(c);
        while (a <= b) {
            int cur = a * a + b * b;
            if (cur == c) {
                return true;
            } else if (cur > c) {
                b--;
            } else {
                a++;
            }
        }
        return false;
    }
}
```

- 时间复杂度： $O(\sqrt{c})$
- 空间复杂度： $O(1)$

费马平方和

费马平方和：奇质数能表示为两个平方数之和的充分必要条件是該质数被 4 除余 1。

翻译过来就是：当且仅当一个自然数的质因数分解中，满足 $4k+3$ 形式的质数次方数均为偶数时，该自然数才能被表示为两个平方数之和。

因此我们对 c 进行质因数分解，再判断满足 $4k+3$ 形式的质因子的次方数是否均为偶数即

可。

代码：

```
public class Solution {
    public boolean judgeSquareSum(int c) {
        for (int i = 2, cnt = 0; i * i <= c; i++, cnt = 0) {
            while (c % i == 0 && ++cnt > 0) c /= i;
            if (i % 4 == 3 && cnt % 2 != 0) return false;
        }
        return c % 4 != 3;
    }
}
```

- 时间复杂度： $O(\sqrt{c})$
- 空间复杂度： $O(1)$

我猜你问

- 三种解法复杂度都一样，哪个才是最优解呀？

前两套解法是需要「真正掌握」的，而「费马平方和」更多的是作为一种拓展。

你会发现从复杂度上来说，其实「费马平方和」并没有比前两种解法更好，但由于存在对 `c` 除质因数操作，导致「费马平方和」实际表现效果要优于同样复杂度的其他做法。但这仍然不成为我们必须掌握「费马平方和」的理由。

三者从复杂度上来说，都是 $O(\sqrt{c})$ 算法，不存在最不最优的问题。

- 是否有关于「费马平方和」的证明呢？

想要看 莱昂哈德·欧拉 对于「费马平方和」的证明在 [这里](#)，我这里直接引用 费马 本人的证明：

我确实发现了一个美妙的证明，但这里空白太小写不下。

- 我就是要学「费马平方和」，有没有可读性更高的代码？

有的，在这里。喜欢的话可以考虑背过：

```
public class Solution {
    public boolean judgeSquareSum(int c) {
        for (int i = 2; i * i <= c; i++) {
            int cnt = 0;
            while (c % i == 0) {
                cnt++;
                c /= i;
            }
            if (i % 4 == 3 && cnt % 2 != 0) return false;
        }
        return c % 4 != 3;
    }
}
```

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 **645. 错误的集合**，难度为 **简单**。

Tag：「模拟」、「哈希表」、「数学」、「桶排序」

集合 **s** 包含从 **1** 到 **n** 的整数。不幸的是，因为数据错误，导致集合里面某一个数字复制了成了集合里面的另外一个数字的值，导致集合 丢失了一个数字 并且 有一个数字重复。

给定一个数组 **nums** 代表了集合 **S** 发生错误后的结果。

请你找出重复出现的整数，再找到丢失的整数，将它们以数组的形式返回。

示例 1：

输入：nums = [1,2,2,4]
输出：[2,3]

示例 2：

输入：nums = [1,1]
输出：[1,2]

宫水三叶
刷题日记

公众号：宫水三叶的刷题日记

提示：

- $2 \leq \text{nums.length} \leq 10^4$
- $1 \leq \text{nums}[i] \leq 10^4$

计数

一个朴素的做法是，使用「哈希表」统计每个元素出现次数，然后在 $[1, n]$ 查询每个元素的出现次数。

在「哈希表」中出现 2 次的为重复元素，未在「哈希表」中出现的元素为缺失元素。

由于这里数的范围确定为 $[1, n]$ ，我们可以使用数组来充当「哈希表」，以减少「哈希表」的哈希函数执行和冲突扩容的时间开销。

执行结果： **通过** [显示详情 >](#)

[▶ 添加备注](#)

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： **40.3 MB**，在所有 Java 提交中击败了 **15.82%** 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[] findErrorNums(int[] nums) {
        int n = nums.length;
        int[] cnts = new int[n + 1];
        for (int x : nums) cnts[x]++;
        int[] ans = new int[2];
        for (int i = 1; i <= n; i++) {
            if (cnts[i] == 0) ans[1] = i;
            if (cnts[i] == 2) ans[0] = i;
        }
        return ans;
    }
}

```

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

数学

我们还可以利用数值范围为 $[1, n]$ ，只有一个数重复和只有一个缺失的特性，进行「作差」求解。

- 令 $[1, n]$ 的求和为 tot ，这部分可以使用「等差数列求和公式」直接得出： $tot = \frac{n(1+n)}{2}$ ；
- 令数组 $nums$ 的求和值为 sum ，由循环累加可得；
- 令数组 $nums$ 去重求和值为 set ，由循环配合「哈希表/数组」累加可得。

最终答案为 (重复元素, 缺失元素) = (sum-set, tot-set)。

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 >](#)

[添加备注](#)

执行用时： **2 ms** ，在所有 Java 提交中击败了 **91.50%** 的用户

内存消耗： **40.3 MB** ，在所有 Java 提交中击败了 **21.15%** 的用户

炫耀一下：



[写题解，分享我的解题思路](#)

代码：

```
class Solution {
    public int[] findErrorNums(int[] nums) {
        int n = nums.length;
        int[] cnts = new int[n + 1];
        int tot = (1 + n) * n / 2;
        int sum = 0, set = 0;
        for (int x : nums) {
            sum += x;
            if (cnts[x] == 0) set += x;
            cnts[x] = 1;
        }
        return new int[]{sum - set, tot - set};
    }
}
```

- 时间复杂度： $O(n)$
- 空间复杂度： $O(n)$

桶排序

因为值的范围在 $[1, n]$ ，我们可以运用「桶排序」的思路，根据 $nums[i] = i + 1$ 的对应关系使用 $O(n)$ 的复杂度将每个数放在其应该落在的位置里。

然后线性扫描一遍排好序的数组，找到不符合 $nums[i] = i + 1$ 对应关系的位置，从而确定重复元素和缺失元素是哪个值。

执行结果： **通过** [显示详情 >](#)

[添加备注](#)

执行用时： **2 ms** ，在所有 Java 提交中击败了 **91.50%** 的用户

内存消耗： **39.8 MB** ，在所有 Java 提交中击败了 **73.23%** 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[] findErrorNums(int[] nums) {
        int n = nums.length;
        for (int i = 0; i < n; i++) {
            while (nums[i] != i + 1 && nums[nums[i] - 1] != nums[i]) {
                swap(nums, i, nums[i] - 1);
            }
        }
        int a = -1, b = -1;
        for (int i = 0; i < n; i++) {
            if (nums[i] != i + 1) {
                a = nums[i];
                b = i == 0 ? 1 : nums[i - 1] + 1;
            }
        }
        return new int[]{a, b};
    }
    void swap(int[] nums, int i, int j) {
        int tmp = nums[i];
        nums[i] = nums[j];
        nums[j] = tmp;
    }
}

```

- 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 **650. 只有两个键的键盘**，难度为 **中等**。

Tag：「动态规划」、「线性 DP」、「数学」、「打表」

最初记事本上只有一个字符 'A'。你每次可以对这个记事本进行两种操作：

- Copy All（复制全部）：复制这个记事本中的所有字符（不允许仅复制部分字符）。
- Paste（粘贴）：粘贴上一次复制的字符。

给你一个数字 n ，你需要使用最少的操作次数，在记事本上输出 恰好 n 个 'A'。返回能够打印出 n 个 'A' 的最少操作次数。

示例 1：

输入：3

输出：3

解释：

最初，只有一个字符 'A'。

第 1 步，使用 Copy All 操作。

第 2 步，使用 Paste 操作来获得 'AA'。

第 3 步，使用 Paste 操作来获得 'AAA'。

示例 2：

输入：n = 1

输出：0

提示：

- $1 \leq n \leq 1000$

动态规划

定义 $f[i][j]$ 为经过最后一次操作后，当前记事本上有 i 个字符，粘贴板上有 j 个字符的最小操作次数。

由于我们粘贴板的字符必然是经过 Copy All 操作而来，因此对于一个合法的 $f[i][j]$ 而言，必然有 $j \leq i$ 。

不失一般性地考虑 $f[i][j]$ 该如何转移：

- 最后一次操作是 Paste 操作：此时粘贴板的字符数量不会发生变化，即有 $f[i][j] = f[i-j][j] + 1$ ；
- 最后一次操作是 Copy All 操作：那么此时的粘贴板的字符数与记事本上的字符数

相等（满足 $i = j$ ），此时的 $f[i][j] = \min(f[i][x] + 1), 0 \leq x < i$ 。

我们发现最后一个合法的 $f[i][j]$ （满足 $i = j$ ）依赖与前面 $f[i][j]$ （满足 $j < i$ ）。

因此实现上，我们可以使用一个变量 min 保存前面转移的最小值，用来更新最后的 $f[i][j]$ 。

再进一步，我们发现如果 $f[i][j]$ 的最后一次操作是由 `Paste` 而来，原来粘贴板的字符数不会超过 $i/2$ ，因此在转移 $f[i][j]$ （满足 $j < i$ ）时，其实只需要枚举 $[0, i/2]$ 即可。

执行结果： 通过 [显示详情 >](#)

[添加备注](#)

执行用时： **53 ms**，在所有 Java 提交中击败了 **5.59%** 的用户

内存消耗： **48.2 MB**，在所有 Java 提交中击败了 **5.07%** 的用户

通过测试用例： **126 / 126**

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 INF = 0x3f3f3f3f;
    public int minSteps(int n) {
        int[][] f = new int[n + 1][n + 1];
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= n; j++) {
                f[i][j] = INF;
            }
        }
        f[1][0] = 0; f[1][1] = 1;
        for (int i = 2; i <= n; i++) {
            int min = INF;
            for (int j = 0; j <= i / 2; j++) {
                f[i][j] = f[i - j][j] + 1;
                min = Math.min(min, f[i][j]);
            }
            f[i][i] = min + 1;
        }
        int ans = INF;
        for (int i = 0; i <= n; i++) ans = Math.min(ans, f[n][i]);
        return ans;
    }
}

```

- 时间复杂度： $O(n^2)$
- 空间复杂度： $O(n^2)$

数学

如果我们将「1 次 Copy All + x 次 Paste」看做一次“动作”的话。

那么一次“动作”所产生的效果就是将原来的字符串变为原来的 $x + 1$ 倍。

最终的最小操作次数方案可以等价以下操作流程：

1. 起始对长度为 1 的记事本字符进行 1 次 Copy All + $k_1 - 1$ 次 Paste 操作（消耗次数为 k_1 ，得到长度为 k_1 的记事本长度）；
2. 对长度为 k_1 的记事本字符进行 1 次 Copy All + $k_2 - 1$ 次 Paste 操作（消耗次数为 $k_1 + k_2$ ，得到长度为 $k_1 * k_2$ 的记事本长度）
- ...

最终经过 k 次“动作”之后，得到长度为 n 的记事本长度，即有：

$$n = k_1 * k_2 * \dots * k_x$$

问题转化为：如何对 n 进行拆分，可以使得 $k_1 + k_2 + \dots + k_x$ 最小。

对于任意一个 k_i （合数）而言，根据定理 $a * b \geq a + b$ 可知进一步的拆分必然不会导致结果变差。

因此，我们只需要使用「试除法」对 n 执行分解质因数操作，累加所有的操作次数，即可得到答案。

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 **82.08%** 的用户

通过测试用例： **126 / 126**

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 minSteps(int n) {
        int ans = 0;
        for (int i = 2; i * i <= n; i++) {
            while (n % i == 0) {
                ans += i;
                n /= i;
            }
        }
        if (n != 1) ans += n;
        return ans;
    }
}
```

- 时间复杂度： $O(\sqrt{n})$
- 空间复杂度： $O(1)$

打表

我们发现，对于某个 $minSteps(i)$ 而言为定值，且数据范围只有 1000，因此考虑使用打表来做。

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 MB**，在所有 Java 提交中击败了 **93.97%** 的用户

通过测试用例： **126 / 126**

炫耀一下：



[写题解，分享我的解题思路](#)

代码：

刷题日记

公众号：宫水三叶的刷题日记

```
class Solution {
    static int N = 1010;
    static int[] g = new int[N];
    static {
        for (int k = 2; k < N; k++) {
            int cnt = 0, n = k;
            for (int i = 2; i * i <= n; i++) {
                while (n % i == 0) {
                    cnt += i;
                    n /= i;
                }
            }
            if (n != 1) cnt += n;
            g[k] = cnt;
        }
        // System.out.println(Arrays.toString(g)); // 输出打表结果
    }
    public int minSteps(int n) {
        return g[n];
    }
}
```

```
class Solution {
    static int[] g = new int[]{0, 0, 2, 3, 4, 5, 5, 7, 6, 6, 7, 11, 7, 13, 9, 8, 8, 17, 8, 16, 13, 10, 14, 14, 9, 12, 12, 16, 14, 19, 17, 14, 12};
    public int minSteps(int n) {
        return g[n];
    }
}
```

- 时间复杂度：将打表逻辑配合 `static` 交给 OJ 执行，复杂度为 $O(C * \sqrt{C})$ ， C 为常数，固定为 1010；将打表逻辑放到本地执行，复杂度为 $O(1)$
- 空间复杂度： $O(C)$

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 **789. 逃脱阻碍者**，难度为 **中等**。

Tag: 「数学」

你在进行一个简化版的吃豆人游戏。你从 $[0, 0]$ 点开始出发，你的目的地是 $\text{target} = [\text{xtarget}, \text{ytarget}]$ 。地图上有一些阻碍者，以数组 ghosts 给出，第 i 个阻碍者从 $\text{ghosts}[i] = [x_i, y_i]$ 出发。所有输入均为 整数坐标。

每一回合，你和阻碍者们可以同时向东，西，南，北四个方向移动，每次可以移动到距离原位置 1 个单位 的新位置。当然，也可以选择 不动。所有动作 同时 发生。

如果你可以在任何阻碍者抓住你 之前 到达目的地（阻碍者可以采取任意行动方式），则被视为 逃脱成功。如果你和阻碍者同时到达了一个位置（包括目的地）都不算是逃脱成功。

只有在你有可能成功逃脱时，输出 `true`；否则，输出 `false`。

示例 1：

输入：`ghosts = [[1,0],[0,3]]`, `target = [0,1]`

输出：`true`

解释：你可以直接一步到达目的地 $(0, 1)$ ，在 $(1, 0)$ 或者 $(0, 3)$ 位置的阻碍者都不可能抓住你。

示例 2：

输入：`ghosts = [[1,0]]`, `target = [2,0]`

输出：`false`

解释：你需要走到位于 $(2, 0)$ 的目的地，但是在 $(1, 0)$ 的阻碍者位于你和目的地之间。

示例 3：

输入：`ghosts = [[2,0]]`, `target = [1,0]`

输出：`false`

解释：阻碍者可以和你同时达到目的地。

示例 4：

宫水三叶
の
刷题日记

公众号：宫水三叶的刷题日记

输入：ghosts = [[5,0],[-10,-2],[0,-5],[-2,-2],[-7,1]], target = [7,7]

输出：false

示例 5：

输入：ghosts = [[-1,0],[0,1],[-1,0],[0,1],[-1,0]], target = [0,0]

输出：true

提示：

- $1 \leq \text{ghosts.length} \leq 100$
- $\text{ghosts}[i].\text{length} == 2$
- $-10^4 \leq x_i, y_i \leq 10^4$
- 同一位置可能有多个阻碍者。
- $\text{target.length} == 2$
- $-10^4 \leq x_{\text{target}}, y_{\text{target}} \leq 10^4$

数学

从数据范围 $-10^4 \leq x_{\text{target}}, y_{\text{target}} \leq 10^4$ 且每次只能移动一个单位（或不移动）就注定了不能使用朴素的 BFS 进行求解。

朴素的 BFS 是指每次在玩家移动一步前，先将阻碍者可以一步到达的位置“置灰”（即设为永不可达），然后判断玩家是否能够到达 target 。

朴素 BFS 本质是模拟，由于棋盘足够大，步长只有 1，因此该做法显然会 TLE。

是否有比模拟更快的做法呢？

根据「树的直径」类似的证明，我们可以证明出「如果一个阻碍者能够抓到玩家，必然不会比玩家更晚到达终点」。

为了方便，我们设玩家起点、阻碍者起点、终点分别为 s 、 e 和 t ，计算两点距离的函数为 $\text{dist}(x, y)$ 。

刷题日记

公众号：宫水三叶的刷题日记

假设玩家从 s 到 t 的路径中会经过点 k ，当且仅当 $dist(e, k) \leq dist(s, k)$ ，即「阻碍者起点与点 k 的距离」小于等于「玩家起点与点 k 的距离」时，阻碍者可以在点 k 抓到玩家。

由于「玩家到终点」以及「阻碍者到终点」的路径存在公共部分 $dist(k, t)$ ，可推导出：

$$dist(e, k) + dist(k, t) \leq dist(s, k) + dist(k, t)$$

即得证 如果一个阻碍者能够抓到玩家，那么该阻碍者必然不会比玩家更晚到达终点。

由于步长为 1，且移动规则为上下左右四联通方向，因此 $dist(x, y)$ 的实现为计算两点的曼哈顿距离。

代码：

```
class Solution {
    int dist(int x1, int y1, int x2, int y2) {
        return Math.abs(x1 - x2) + Math.abs(y1 - y2);
    }
    public boolean escapeGhosts(int[][] gs, int[] t) {
        int cur = dist(0, 0, t[0], t[1]);
        for (int[] g : gs) {
            if (dist(g[0], g[1], t[0], t[1]) <= cur) return false;
        }
        return true;
    }
}
```

- 时间复杂度：令 gs 长度为 n ，计算曼哈顿距离复杂度为 $O(1)$ ，整体复杂度为 $O(n)$
- 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 [810. 黑板异或游戏](#)，难度为 困难。

Tag：「博弈论」、「数学」、「异或」

黑板上写着一个非负整数数组 `nums[i]`。

Alice 和 Bob 轮流从黑板上擦掉一个数字，Alice 先手。如果擦除一个数字后，剩余的所有数字按位异或运算得出的结果等于 0 的话，当前玩家游戏失败。（另外，如果只剩一个数字，按位异或运算得到它本身；如果无数字剩余，按位异或运算结果为 0。）

换种说法就是，轮到某个玩家时，如果当前黑板上所有数字按位异或运算结果等于 0，这个玩家获胜。

假设两个玩家每步都使用最优解，当且仅当 Alice 获胜时返回 `true`。

示例：

输入：nums = [1, 1, 2]

输出：false

解释：

Alice 有两个选择：擦掉数字 1 或 2。

如果擦掉 1，数组变成 [1, 2]。剩余数字按位异或得到 $1 \text{ XOR } 2 = 3$ 。那么 Bob 可以擦掉任意数字，因为 Alice 会成为擦掉最后一个数字的人，即她会输掉游戏。

如果 Alice 擦掉 2，那么数组变成 [1, 1]。剩余数字按位异或得到 $1 \text{ XOR } 1 = 0$ 。Alice 仍然会输掉游戏。

提示：

- $1 \leq N \leq 1000$
- $0 \leq \text{nums}[i] \leq 2^{16}$

基本分析

这是一道「博弈论」题。

如果没接触过博弈论，其实很难想到，特别是数据范围为 10^3 ，很具有迷惑性。

如果接触过博弈论，对于这种「判断先手后手的必胜必败」的题目，博弈论方向是一个优先考虑的方向。

根据题意，如果某位玩家在操作前所有数值异或和为 0，那么该玩家胜利。要我们判断给定序列时，先手是处于「必胜态」还是「必败态」，如果处于「必胜态」返回 `True`，否则返回 `False`。

对于博弈论的题目，通常有两类的思考方式：

1. 经验分析：见过类似的题目，猜一个性质，然后去证明该性质是否可推广。
2. 状态分析：根据题目给定的规则是判断「胜利」还是「失败」来决定优先分析「必胜态」还是「必败态」时具有何种性质，然后证明性质是否可推广。

博弈论

对于本题，给定的是判断「胜利」的规则（在给定序列的情况下，如果所有数值异或和为 0 可立即判断胜利，其他情况无法立即判断胜负），那么我们应该优先判断何为「先手必胜态」，如果不好分析，才考虑分析后手的「必败态」。

接下来是分情况讨论：

1. 如果给定的序列异或和为 0，游戏开始时，先手直接获胜：

由此推导出性质一：给定序列 `nums` 的异或和为 0，先手处于「必胜态」，返回 `True`。

2. 如果给定序列异或和不为 0，我们需要分析，先手获胜的话，序列会满足何种性质：

显然如果要先手获胜，则需要满足「先手去掉一个数，剩余数值异或和必然不为 0；同时后手去掉一个数后，剩余数值异或和必然为 0」。

换句话说，我们需要分析什么情况下「经过一次后手操作」后，序列会以上述情况 1 的状态，回到先手的局面。

也就是反过来分析想要出现「后手必败态」，序列会有何种性质。

假设后手操作前的异或和为 Sum ($Sum \neq 0$)，「后手必败态」意味着去掉任意数字后异或和为 0。

同时根据「相同数值异或结果为 0」的特性，我们知道去掉某个数值，等价于在原有异或和的基础上异或上这个值。

则有：

$$Sum' = Sum \oplus nums[i] = 0$$

公众号：宫水三叶的刷题日记

由于是「后手必败态」，因此 i 取任意一位，都满足上述式子。

则有：

$$Sum \oplus nums[0] = \dots = Sum \oplus nums[k] = \dots = Sum \oplus nums[n-1] = 0$$

同时根据「任意数值与 0 异或数值不变」的特性，我们将每一项进行异或：

$$(Sum \oplus nums[0]) \oplus \dots \oplus (Sum \oplus nums[k]) \oplus \dots \oplus (Sum \oplus nums[n-1]) = 0$$

根据交换律进行变换：

$$(Sum \oplus Sum \oplus \dots \oplus Sum) \oplus (nums[0] \oplus \dots \oplus nums[k] \oplus \dots \oplus nums[n-1]) = 0$$

再结合 Sum 为原序列的异或和可得：

$$(Sum \oplus Sum \oplus \dots \oplus Sum) \oplus Sum = 0, Sum \neq 0$$

至此，我们分析出当处于「后手必败态」时，去掉任意一个数值会满足上述式子。

根据「相同数值偶数次异或结果为 0」的特性，可推导出「后手必败态」会导致交回到先手的序列个数为偶数，由此推导后手操作前序列个数为奇数，后手操作前一个回合为偶数。

到这一步，我们推导出想要出现「后手必败态」，先手操作前的序列个数应当为偶数。

那么根据先手操作前序列个数为偶数（且异或和不为 0），是否能够推导出必然出现「后手必败态」呢？

显然是可以的，因为如果不出现「后手必败态」，会与我们前面分析过程矛盾。

假设先手操作前异或和为 Xor （序列数量为偶数，同时 $Xor \neq 0$ ），如果最终不出现「后手必败态」的话，也就是先手会输掉的话，那么意味着有 $Xor \oplus nums[i] = 0$ ，其中 i 为序列的任意位置。利用此性质，像上述分析那样，将每一项进行展开异或，会得到奇数个 Xor 异或结果为 0，这与开始的 $Xor \neq 0$ 矛盾。

由此推导出性质二：只需要保证先手操作前序列个数为偶数时就会出现「后手必败态」，从而确保先手必胜。

综上，如果序列 `nums` 本身异或和为 0，天然符合「先手必胜态」的条件，答案返回 `True`；如果序列 `nums` 异或和不为 0，但序列长度为偶数，那么最终会出现「后手必败态」，推导出先手必胜，答案返回 `True`。

代码：

```
class Solution {  
    public boolean xorGame(int[] nums) {  
        int sum = 0;  
        for (int i : nums) sum ^= i;  
        return sum == 0 || nums.length % 2 == 0;  
    }  
}
```

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

总结

事实上，在做题的时候，我也是采取「先假定奇偶性，再证明」的做法，因为这样比较快。

但「假定奇偶性」这一步是比较具有跳跃性的，这有点像我前面说到的「经验分析解法」，而本题解证明没有做任何的前置假定，单纯从「先手必胜态」和「后手必败态」进行推导，最终推导出「先手序列偶数必胜」的性质，更符合前面说到的「状态分析解法」。

两种做法殊途同归，在某些博弈论问题上，「经验分析解法」可以通过「归纳」&「反证」很好分析出来，但这要求选手本身具有一定的博弈论基础；而「状态分析解法」则对选手的题量要求低些，逻辑推理能力高些。

两种方法并无优劣之分，都是科学严谨的做法。

我十分建议大家将此题解与 [官方题解](#) 一同阅读，体会两种分析方法的区别。

 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 [879. 盈利计划](#)，难度为 **困难**。

Tag：「动态规划」、「容斥原理」、「数学」、「背包问题」、「多维背包」

集团里有 n 名员工，他们可以完成各种各样的工作创造利润。

第 i 种工作会产生 $profit[i]$ 的利润，它要求 $group[i]$ 名成员共同参与。如果成员参与了其中一项工作，就不能参与另一项工作。

工作的任何至少产生 $minProfit$ 利润的子集称为 盈利计划 。并且工作的成员总数最多为 n 。

有多少种计划可以选择？因为答案很大，所以 返回结果模 $10^9 + 7$ 的值。

示例 1：

输入： $n = 5$, $minProfit = 3$, $group = [2,2]$, $profit = [2,3]$

输出：2

解释：至少产生 3 的利润，该集团可以完成工作 0 和工作 1，或仅完成工作 1。
总的来说，有两种计划。

示例 2：

输入： $n = 10$, $minProfit = 5$, $group = [2,3,5]$, $profit = [6,7,8]$

输出：7

解释：至少产生 5 的利润，只要完成其中一种工作就行，所以该集团可以完成任何工作。
有 7 种可能的计划： (0) ， (1) ， (2) ， $(0,1)$ ， $(0,2)$ ， $(1,2)$ ，以及 $(0,1,2)$ 。

提示：

- $1 \leq n \leq 100$
- $0 \leq minProfit \leq 100$
- $1 \leq group.length \leq 100$
- $1 \leq group[i] \leq 100$
- $profit.length == group.length$
- $0 \leq profit[i] \leq 100$

动态规划

这是一类特殊的多维费用背包问题。

将每个任务看作一个「物品」，完成任务所需要的人数看作「成本」，完成任务得到的利润看作

「价值」。

其特殊在于存在一维容量维度需要满足「不低于」，而不是常规的「不超过」。这需要我们对于某些状态作等价变换。

定义 $f[i][j][k]$ 为考虑前 i 件物品，使用人数不超过 j ，所得利润至少为 k 的方案数。

对于每件物品（令下标从 1 开始），我们有「选」和「不选」两种决策：

- 不选：显然有：

$$f[i-1][j][k]$$

- 选：首先需要满足人数达到要求（ $j \geq \text{group}[i-1]$ ），还需要考虑「至少利润」负值问题：

如果直接令「利润维度」为 $k - \text{profit}[i-1]$ 可能会出现负值，那么负值是否为合法状态呢？这需结合「状态定义」来看，由于是「利润至少为 k 」，因此属于「合法状态」，需要参与转移。

由于我们没有设计动规数组存储「利润至少为负权」状态，我们需要根据「状态定义」做一个等价替换，将这个「状态」映射到 $f[i][j][0]$ 。这主要是利用所有的任务利润都为“非负数”，所以不可能出现利润为负的情况，这时候「利润至少为某个负数 k 」的方案数其实是完全等价于「利润至少为 0」的方案数。

$$f[i-1][j - \text{group}[i-1]][\max(k - \text{profit}[i-1], 0)]$$

最终 $f[i][j][k]$ 为上述两种情况之和。

然后考虑「如何构造有效起始值」问题，还是结合我们的「状态定义」来考虑：

当不存在任何物品（任务）时，所得利用利润必然为 0（满足至少为 0），同时对人数限制没有要求。

因此可以让所有 $f[0][x][0] = 1$ 。

代码（一维空间优化代码见 P2）：

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 profitableSchemes(int n, int min, int[] gs, int[] ps) {
        int m = gs.length;
        long[][][] f = new long[m + 1][n + 1][min + 1];
        for (int i = 0; i <= n; i++) f[0][i][0] = 1;
        for (int i = 1; i <= m; i++) {
            int a = gs[i - 1], b = ps[i - 1];
            for (int j = 0; j <= n; j++) {
                for (int k = 0; k <= min; k++) {
                    f[i][j][k] = f[i - 1][j][k];
                    if (j >= a) {
                        int u = Math.max(k - b, 0);
                        f[i][j][k] += f[i - 1][j - a][u];
                        f[i][j][k] %= mod;
                    }
                }
            }
        }
        return (int)f[m][n][min];
    }
}

```

```

class Solution {
    int mod = (int)1e9+7;
    public int profitableSchemes(int n, int min, int[] gs, int[] ps) {
        int m = gs.length;
        int[][] f = new int[n + 1][min + 1];
        for (int i = 0; i <= n; i++) f[i][0] = 1;
        for (int i = 1; i <= m; i++) {
            int a = gs[i - 1], b = ps[i - 1];
            for (int j = n; j >= a; j--) {
                for (int k = min; k >= 0; k--) {
                    int u = Math.max(k - b, 0);
                    f[j][k] += f[j - a][u];
                    if (f[j][k] >= mod) f[j][k] -= mod;
                }
            }
        }
        return f[n][min];
    }
}

```

- 时间复杂度： $O(m * n * min)$
- 空间复杂度： $O(m * n * min)$

动态规划（作差法）

这个方案足足调了快一个小时 😓

先是爆 `long`，然后转用高精度后被卡内存，最终改为滚动数组后勉强过了（不是，稳稳的过了，之前调得久是我把 `N` 多打了一位，写成 1005 了，`N` 不打错的话，不滚动也是能过的 😓😓😓）

基本思路是先不考虑最小利润 `minProfit`，求得所有只受「人数限制」的方案数 `a`，然后求得考虑「人数限制」同时，利润低于 `minProfit`（不超过 `minProfit - 1`）的所有方案数 `b`。

由 `a - b` 即是答案。

代码：

宫水三叶
の
刷题日记

公众号：宫水三叶的刷题日记

```

import java.math.BigInteger;
class Solution {
    static int N = 105;
    static BigInteger[][] f = new BigInteger[2][N];
    static BigInteger[][][] g = new BigInteger[2][N][N];
    static BigInteger mod = new BigInteger("1000000007");

    public int profitableSchemes(int n, int min, int[] gs, int[] ps) {
        int m = gs.length;

        for (int j = 0; j <= n; j++) {
            f[0][j] = new BigInteger("1");
            f[1][j] = new BigInteger("0");
        }
        for (int j = 0; j <= n; j++) {
            for (int k = 0; k <= min; k++) {
                g[0][j][k] = new BigInteger("1");
                g[1][j][k] = new BigInteger("0");
            }
        }

        for (int i = 1; i <= m; i++) {
            int a = gs[i - 1], b = ps[i - 1];
            int x = i & 1, y = (i - 1) & 1;
            for (int j = 0; j <= n; j++) {
                f[x][j] = f[y][j];
                if (j >= a) {
                    f[x][j] = f[x][j].add(f[y][j - a]);
                }
            }
        }
        if (min == 0) return (f[m&1][n]).mod(mod).intValue();

        for (int i = 1; i <= m; i++) {
            int a = gs[i - 1], b = ps[i - 1];
            int x = i & 1, y = (i - 1) & 1;
            for (int j = 0; j <= n; j++) {
                for (int k = 0; k < min; k++) {
                    g[x][j][k] = g[y][j][k];
                    if (j - a >= 0 && k - b >= 0) {
                        g[x][j][k] = g[x][j][k].add(g[y][j - a][k - b]);
                    }
                }
            }
        }
    }
}

```

刷题日记

公众号: 宫水三叶的刷题日记

```
        return f[m&1][n].subtract(g[m&1][n][min - 1]).mod(mod).intValue();
    }
}
```

- 时间复杂度：第一遍 DP 复杂度为 $O(m * n)$ ；第二遍 DP 复杂度为 $O(m * n * min)$ 。整体复杂度为 $O(m * n * min)$
- 空间复杂度： $O(m * n * min)$

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 **1006. 笨阶乘**，难度为 **中等**。

Tag：「数学」、「栈」

通常，正整数 n 的阶乘是所有小于或等于 n 的正整数的乘积。

例如， $\text{factorial}(10) = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1$ 。

相反，我们设计了一个笨阶乘 clumsy：在整数的递减序列中，我们以一个固定顺序的操作符序列来依次替换原有的乘法操作符：乘法($*$)，除法($/$)，加法($+$)和减法($-$)。

例如， $\text{clumsy}(10) = 10 * 9 / 8 + 7 - 6 * 5 / 4 + 3 - 2 * 1$ 。然而，这些运算仍然使用通常的算术运算顺序：我们在任何加、减步骤之前执行所有的乘法和除法步骤，并且按从左到右处理乘法和除法步骤。

另外，我们使用的除法是地板除法（floor division），所以 $10 * 9 / 8$ 等于 11。这保证结果是一个整数。

实现上面定义的笨函数：给定一个整数 N ，它返回 N 的笨阶乘。

示例 1：

宫水三叶
刷题日记

公众号: 宫水三叶的刷题日记

输入：4

输出：7

解释：7 = 4 * 3 / 2 + 1

示例 2：

输入：10

输出：12

解释：12 = 10 * 9 / 8 + 7 - 6 * 5 / 4 + 3 - 2 * 1

提示：

- $1 \leq N \leq 10000$
- $-2^{31} \leq \text{answer} \leq 2^{31} - 1$ （答案保证符合 32 位整数）

通用表达式解法

第一种解法是我们的老朋友解法了，使用「双栈」来解决通用表达式问题。

事实上，我提供这套解决方案不仅仅能解决只有 `+ - ()`（[224. 基本计算器](#)）或者 `+ - * /`（[227. 基本计算器 II](#)）的表达式问题，还能解决 `+ - * / ^ % ()` 的完全表达式问题。

甚至支持自定义运算符，只要在运算优先级上进行维护即可。

对于「表达式计算」这一类问题，你都可以使用这套思路进行解决。我十分建议你加强理解这套处理逻辑。

对于「任何表达式」而言，我们都使用两个栈 `nums` 和 `ops`：

- `nums`：存放所有的数字
- `ops`：存放所有的数字以外的操作

然后从前往后做，对遍历到的字符做分情况讨论：

- 空格：跳过
- (：直接加入 ops 中，等待与之匹配的)
-)：使用现有的 nums 和 ops 进行计算，直到遇到左边最近的一个左括号为止，计算结果放到 nums
- 数字：从当前位置开始继续往后取，将整个连续数字整体取出，加入 nums
- + - * / ^ %：需要将操作放入 ops 中。在放入之前先把栈内可以算的都算掉（只有「栈内运算符」比「当前运算符」优先级高/同等，才进行运算），使用现有的 nums 和 ops 进行计算，直到没有操作或者遇到左括号，计算结果放到 nums

我们可以通过 🍎 来理解 只有「栈内运算符」比「当前运算符」优先级高/同等，才进行运算 是什么意思：

因为我们是从前往后做的，假设我们当前已经扫描到 $2 + 1$ 了（此时栈内的操作为 + ）。

1. 如果后面出现的 $+ 2$ 或者 $- 1$ 的话，满足「栈内运算符」比「当前运算符」优先级高/同等，可以将 $2 + 1$ 算掉，把结果放到 nums 中；
2. 如果后面出现的是 $* 2$ 或者 $/ 1$ 的话，不满足「栈内运算符」比「当前运算符」优先级高/同等，这时候不能计算 $2 + 1$ 。

更为详细的讲解可以看这篇题解：[使用「双栈」解决「究极表达式计算」问题](#)

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 clumsy(int n) {
        Deque<Integer> nums = new ArrayDeque<>();
        Deque<Character> ops = new ArrayDeque<>();
        // 维护运算符优先级
        Map<Character, Integer> map = new HashMap<>(){
            put('*', 2);
            put('/', 2);
            put('+', 1);
            put('-', 1);
        };
        char[] cs = new char[]{'*', '/', '+', '-'};
        for (int i = n, j = 0; i > 0; i--, j++) {
            char op = cs[j % 4];
            nums.addLast(i);
            // 如果「当前运算符优先级」不高于「栈顶运算符优先级」，说明栈内的可以算
            while (!ops.isEmpty() && map.get(ops.peekLast()) >= map.get(op)) {
                calc(nums, ops);
            }
            if (i != 1) ops.add(op);
        }
        // 如果栈内还有元素没有算完，继续算
        while (!ops.isEmpty()) calc(nums, ops);
        return nums.peekLast();
    }

    void calc(Deque<Integer> nums, Deque<Character> ops) {
        int b = nums.pollLast(), a = nums.pollLast();
        int op = ops.pollLast();
        int ans = 0;
        if (op == '+') ans = a + b;
        else if (op == '-') ans = a - b;
        else if (op == '*') ans = a * b;
        else if (op == '/') ans = a / b;
        nums.addLast(ans);
    }
}

```

- 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

数学解法（打表技巧分析）

这次在讲【证明】之前，顺便给大家讲讲找规律的题目该怎么做。

由于是按照特定顺序替换运算符，因此应该是有一些特性可以被我们利用的。

通常我们需要先实现一个可打表的算法（例如上述的解法一，这是为什么掌握「通用表达式」解法具有重要意义），将连续数字的答案打印输出，来寻找规律：

```
Solution solution = new Solution();
for (int i = 1; i <= 10000; i++) {
    int res = solution.clumsy(i);
    System.out.println(i + " : " + res);
}
```

宫水三叶
の
刷题日记

公众号：宫水三叶的刷题日记

✓ Tests passed: 1 of 1 test – 2 s 160 ms

400 : 401
401 : 403
402 : 404
403 : 402
404 : 405
405 : 407
406 : 408
407 : 406
408 : 409
409 : 411
410 : 412
411 : 410
412 : 413
413 : 415
414 : 416
415 : 414
416 : 417
417 : 419
418 : 420
419 : 418
420 : 421
421 : 423
422 : 424
423 : 422
424 : 425
425 : 427
426 : 428
427 : 426
428 : 429
429 : 431
430 : 432
431 : 430
432 : 433
433 : 435
434 : 436
435 : 434
436 : 437
437 : 439
438 : 440
439 : 438

似乎 n 与 答案比较接近，我们考虑将两者的差值输出：

刷题日记

公众号: 宫水三叶的刷题日记

```
Solution solution = new Solution();
for (int i = 1; i <= 10000; i++) {
    int res = solution.clumsy(i);
    System.out.println(i + " : " + res + " : " + (res - i));
}
```

✓ Tests passed: 1 of 1 test - 2 s 272 ms

```
400 : 401 : 1
401 : 403 : 2
402 : 404 : 2
403 : 402 : -1
404 : 405 : 1
405 : 407 : 2
406 : 408 : 2
407 : 406 : -1
408 : 409 : 1
409 : 411 : 2
410 : 412 : 2
411 : 410 : -1
412 : 413 : 1
413 : 415 : 2
414 : 416 : 2
415 : 414 : -1
416 : 417 : 1
417 : 419 : 2
418 : 420 : 2
419 : 418 : -1
420 : 421 : 1
421 : 423 : 2
422 : 424 : 2
423 : 422 : -1
424 : 425 : 1
425 : 427 : 2
426 : 428 : 2
427 : 426 : -1
428 : 429 : 1
429 : 431 : 2
430 : 432 : 2
431 : 430 : -1
432 : 433 : 1
433 : 435 : 2
434 : 436 : 2
435 : 434 : -1
436 : 437 : 1
437 : 439 : 2
438 : 440 : 2
439 : 438 : -1
```

咦，好像发现了什么不得了的东西。似乎每四个数，差值都是 [1, 2, 2, -1]

再修改我们的打表逻辑，来验证一下（只输出与我们猜想不一样的数字）：

```

Solution solution = new Solution();
int[] diff = new int[]{1,2,2,-1};
for (int i = 1; i <= 10000; i++) {
    int res = solution.clumsy(i);
    int t = res - i;
    if (t != diff[i % 4]) {
        System.out.println(i + " : " + res);
    }
}

```

```

1 : 1
2 : 2
3 : 6
4 : 7

```

只有前四个数字被输出，其他数字都是符合我们的猜想规律的。

到这里我们已经知道代码怎么写可以 AC 了，十分简单。

代码：

```

class Solution {
    public int clumsy(int n) {
        int[] special = new int[]{1,2,6,7};
        int[] diff = new int[]{1,2,2,-1};
        if (n <= 4) return special[(n - 1) % 4];
        return n + diff[n % 4];
    }
}

```

- 时间复杂度： $O(1)$
- 空间复杂度： $O(1)$

证明

讲完我们的【实战技巧】之后，再讲讲如何证明。

上述的做法比较适合于笔试或者比赛，但是面试，通常还需要证明做法为什么是正确的。

我们不失一般性的分析某个 n ，当然这个 n 必须是大于 4，不属于我们的特判值。

然后对 n 进行讨论（根据我们的打表猜想去证明规律是否可推广）：

1. $n \% 4 == 0$: $f(n) = n * (n - 1) / (n - 2) + \dots + 5 - 4 * 3 / 2 + 1 = n + 1$ ，即 $\text{diff} = 1$
2. $n \% 4 == 1$: $f(n) = n * (n - 1) / (n - 2) + \dots + 6 - 5 * 4 / 3 + 2 - 1 = n + 2$ ，即 $\text{diff} = 2$
3. $n \% 4 == 2$: $f(n) = n * (n - 1) / (n - 2) + \dots + 7 - 6 * 5 / 4 + 3 - 2 * 1 = n + 2$ ，即 $\text{diff} = 2$
4. $n \% 4 == 3$: $f(n) = n * (n - 1) / (n - 2) + \dots + 8 - 7 * 6 / 5 + 4 - 3 * 2 / 1 = n - 1$ ，即 $\text{diff} = -1$

上述的表达式展开过程属于小学数学内容，省略号部分的项式的和为 0，因此你只需要关注我写出来的那部分。

至此，我们证明了我们的打表猜想具有「可推广」的特性。

甚至我们应该学到：证明可以是基于猜想去证明，而不必从零开始进行推导。

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 [1104. 二叉树寻路](#)，难度为中等。

Tag：「二叉树」、「模拟」、「数学」

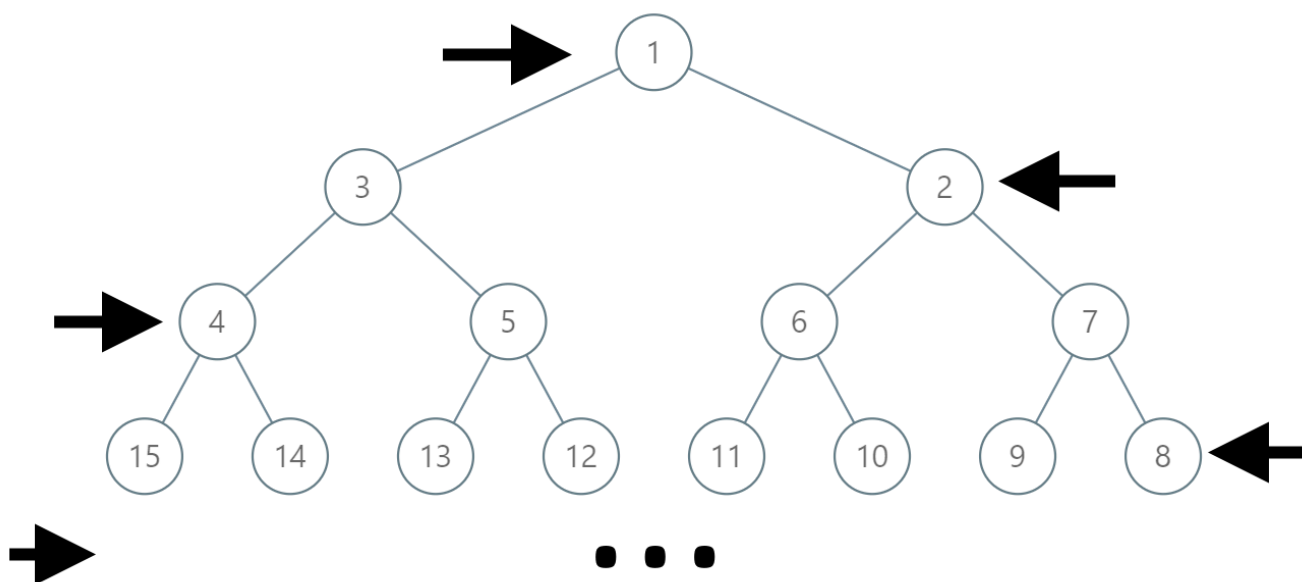
在一棵无限的二叉树上，每个节点都有两个子节点，树中的节点逐行依次按“之”字形进行标记。

如下图所示，在奇数行（即，第一行、第三行、第五行……）中，按从左到右的顺序进行标记；

而偶数行（即，第二行、第四行、第六行……）中，按从右到左的顺序进行标记。

刷题日记

公众号: 宫水三叶的刷题日记



给你树上某一个节点的标号 `label`，请你返回从根节点到该标号为 `label` 节点的路径，该路径是由途经的节点标号所组成的。

示例 1：

输入：`label = 14`

输出：`[1,3,4,14]`

示例 2：

输入：`label = 26`

输出：`[1,2,6,10,26]`

提示：

- $1 \leq \text{label} \leq 10^6$

模拟

一个朴素的做法是根据题意进行模拟。

利用从根节点到任意一层都是满二叉树，我们可以先确定 `label` 所在的层级 `level`，然后计

算出当前层起始节点值（最小值）和结束节点值（最大值）。

再利用「每层节点数量翻倍」&「隔层奇偶性翻转」，寻址出上一层的节点下标（令每层下标均「从左往右」计算，并从 1 开始），直到构造出答案（寻址到根节点）。

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 {
    // 第 level 层的起始节点值
    int getStart(int level) {
        return (int)Math.pow(2, level - 1);
    }
    // 第 level 层的结束节点值
    int getEnd(int level) {
        int a = getStart(level);
        return a + a - 1;
    }
    public List<Integer> pathInZigZagTree(int n) {
        // 计算 n 所在层级
        int level = 1;
        while (getEnd(level) < n) level++;

        int[] ans = new int[level];
        int idx = level - 1, cur = n;
        while (idx >= 0) {
            ans[idx--] = cur;
            int tot = (int)Math.pow(2, level - 1);
            int start = getStart(level), end = getEnd(level);
            if (level % 2 == 0) {
                // 当前层为偶数层，则当前层节点「从右往左」数值递增，相应计算上一层下标也应该「从右往左」
                int j = tot / 2;
                for (int i = start; i <= end; i += 2, j--) {
                    if (i == cur || (i + 1) == cur) break;
                }
                int prevStart = getStart(level - 1);
                while (j-- > 1) prevStart++;
                cur = prevStart;
            } else {
                // 当前层为奇数层，则当前层节点「从左往右」数值递增，相应计算上一层下标也应该「从左往右」
                int j = 1;
                for (int i = start; i <= end; i += 2, j++) {
                    if (i == cur || (i + 1) == cur) break;
                }
                int prevEnd = getEnd(level - 1);
                while (j-- > 1) prevEnd--;
                cur = prevEnd;
            }
            level--;
        }
        List<Integer> list = new ArrayList<>();
        for (int i : ans) list.add(i);
        return list;
    }
}

```

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：确定 n 所在层级复杂度为 $O(\log n)$ ；构造答案最坏情况下每个节点会被遍历一次，复杂度为 $O(n)$
- 空间复杂度： $O(1)$

数学

上述解法复杂度上界取决于「由当前行节点位置确定上层位置」的线性遍历。

如果二叉树本身不具有奇偶性翻转的话，显然某个节点 x 的父节点为 $\lfloor x/2 \rfloor$ ，但事实上存在奇偶性翻转，而在解法一中我们已经可以 $O(1)$ 计算某一层的起始值和结束值，有了「起始值 & 结束值」和「当前节点所在层的相对位置」，只需要利用“对称性”找到父节点在上层的相应位置，然后根据相应位置算出父节点值即可。

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 getStart(int level) {
        return (int)Math.pow(2, level - 1);
    }
    int getEnd(int level) {
        int a = getStart(level);
        return a + a - 1;
    }
    public List<Integer> pathInZigZagTree(int n) {
        int level = 1;
        while (getEnd(level) < n) level++;
        int[] ans = new int[level];
        int idx = level - 1, cur = n;
        while (idx >= 0) {
            ans[idx--] = cur;
            int loc = ((1 << (level)) - 1 - cur) >> 1;
            cur = (1 << (level - 2)) + loc;
            level--;
        }
        List<Integer> list = new ArrayList<>();
        for (int i : ans) list.add(i);
        return list;
    }
}

```

- 时间复杂度：复杂度上界取决于确定 n 所在层级。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 **1137. 第 N 个泰波那契数**，难度为 简单。

Tag：「动态规划」、「递归」、「递推」、「矩阵快速幂」、「打表」

泰波那契序列 T_n 定义如下：

$T_0 = 0, T_1 = 1, T_2 = 1$, 且在 $n \geq 0$ 的条件下 $T_{n+3} = T_n + T_{n+1} + T_{n+2}$

给你整数 n ，请返回第 n 个泰波那契数 T_n 的值。

公众号: 宫水三叶的刷题日记

示例 1：

输入：n = 4

输出：4

解释：

$$T_3 = 0 + 1 + 1 = 2$$

$$T_4 = 1 + 1 + 2 = 4$$

示例 2：

输入：n = 25

输出：1389537

提示：

- $0 \leq n \leq 37$
- 答案保证是一个 32 位整数，即 $\text{answer} \leq 2^{31} - 1$ 。

迭代实现动态规划

都直接给出状态转移方程了，其实就是道模拟题。

使用三个变量，从前往后算一遍即可。

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 tribonacci(int n) {
        if (n == 0) return 0;
        if (n == 1 || n == 2) return 1;
        int a = 0, b = 1, c = 1;
        for (int i = 3; i <= n; i++) {
            int d = a + b + c;
            a = b;
            b = c;
            c = d;
        }
        return c;
    }
}

```

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

递归实现动态规划

也就是记忆化搜索，创建一个 `cache` 数组用于防止重复计算。

代码：

```

class Solution {
    int[] cache = new int[40];
    public int tribonacci(int n) {
        if (n == 0) return 0;
        if (n == 1 || n == 2) return 1;
        if (cache[n] != 0) return cache[n];
        cache[n] = tribonacci(n - 1) + tribonacci(n - 2) + tribonacci(n - 3);
        return cache[n];
    }
}

```

- 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

矩阵快速幂

这还是一道「矩阵快速幂」的板子题。

首先你要对「快速幂」和「矩阵乘法」概念有所了解。

矩阵快速幂用于求解一般性问题：给定大小为 $n * n$ 的矩阵 M ，求答案矩阵 M^k ，并对答案矩阵中的每位元素对 P 取模。

在上述两种解法中，当我们要求解 $f[i]$ 时，需要将 $f[0]$ 到 $f[n - 1]$ 都算一遍，因此需要线性的复杂度。

对于此类的「数列递推」问题，我们可以使用「矩阵快速幂」来进行加速（比如要递归一个长度为 $1e9$ 的数列，线性复杂度会被卡）。

使用矩阵快速幂，我们只需要 $O(\log n)$ 的复杂度。

根据题目的递推关系 ($i \geq 3$)：

$$f(i) = f(i - 1) + f(i - 2) + f(i - 3)$$

我们发现要求解 $f(i)$ ，其依赖的是 $f(i - 1)$ 、 $f(i - 2)$ 和 $f(i - 3)$ 。

我们可以将其存成一个列向量：

$$\begin{bmatrix} f(i - 1) \\ f(i - 2) \\ f(i - 3) \end{bmatrix}$$

当我们整理出依赖的列向量之后，不难发现，我们想求的 $f(i)$ 所在的列向量是这样的：

$$\begin{bmatrix} f(i) \\ f(i - 1) \\ f(i - 2) \end{bmatrix}$$

利用题目给定的依赖关系，对目标矩阵元素进行展开：

$$\begin{bmatrix} f(i) \\ f(i - 1) \\ f(i - 2) \end{bmatrix} = \begin{bmatrix} f(i - 1) * 1 + f(i - 2) * 1 + f(i - 3) * 1 \\ f(i - 1) * 1 + f(i - 2) * 0 + f(i - 3) * 0 \\ f(i - 1) * 0 + f(i - 2) * 1 + f(i - 3) * 0 \end{bmatrix}$$

那么根据矩阵乘法，即有：

$$\begin{bmatrix} f(i) \\ f(i-1) \\ f(i-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} f(i-1) \\ f(i-2) \\ f(i-3) \end{bmatrix}$$

我们令

$$Mat = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

然后发现，利用 Mat 我们也能实现数列递推（公式太难敲了，随便列两项吧）：

$$Mat * \begin{bmatrix} f(i-1) \\ f(i-2) \\ f(i-3) \end{bmatrix} = \begin{bmatrix} f(i) \\ f(i-1) \\ f(i-2) \end{bmatrix}$$

$$Mat * \begin{bmatrix} f(i) \\ f(i-1) \\ f(i-2) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i) \\ f(i-1) \end{bmatrix}$$

再根据矩阵运算的结合律，最终有：

$$\begin{bmatrix} f(n) \\ f(n-1) \\ f(n-2) \end{bmatrix} = Mat^{n-2} * \begin{bmatrix} f(2) \\ f(1) \\ f(0) \end{bmatrix}$$

从而将问题转化为求解 Mat^{n-2} ，这时候可以套用「矩阵快速幂」解决方案。

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 {
    int N = 3;
    int[][] mul(int[][] a, int[][] b) {
        int[][] c = new int[N][N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                c[i][j] = a[i][0] * b[0][j] + a[i][1] * b[1][j] + a[i][2] * b[2][j];
            }
        }
        return c;
    }
    public int tribonacci(int n) {
        if (n == 0) return 0;
        if (n == 1 || n == 2) return 1;
        int[][] ans = new int[][]{
            {1,0,0},
            {0,1,0},
            {0,0,1}
        };
        int[][] mat = new int[][]{
            {1,1,1},
            {1,0,0},
            {0,1,0}
        };
        int k = n - 2;
        while (k != 0) {
            if ((k & 1) != 0) ans = mul(ans, mat);
            mat = mul(mat, mat);
            k >>= 1;
        }
        return ans[0][0] + ans[0][1];
    }
}

```

- 时间复杂度： $O(\log n)$
- 空间复杂度： $O(1)$

打表

当然，我们也可以将数据范围内的所有答案进行打表预处理，然后在询问时直接查表返回。

但对这种题目进行打表带来的收益没有平常打表题的大，因为打表内容不是作为算法必须的一个

环节，而直接是作为该询问的答案，但测试样例是不会相同的，即不会有两个测试数据都是 $n = 37$ 。

这时候打表节省的计算量是不同测试数据之间的相同前缀计算量，例如 $n = 36$ 和 $n = 37$ ，其 35 之前的计算量只会被计算一次。

因此直接为「解法二」的 `cache` 添加 `static` 修饰其实是更好的方式：代码更短，同时也能起到同样的节省运算量的效果。

代码：

```
class Solution {
    static int[] cache = new int[40];
    static {
        cache[0] = 0;
        cache[1] = 1;
        cache[2] = 1;
        for (int i = 3; i < cache.length; i++) {
            cache[i] = cache[i - 1] + cache[i - 2] + cache[i - 3];
        }
    }
    public int tribonacci(int n) {
        return cache[n];
    }
}
```

- 时间复杂度：将打表逻辑交给 *OJ*，复杂度为 $O(C)$ ， C 固定为 40。将打表逻辑放到本地进行，复杂度为 $O(1)$
- 空间复杂度： $O(n)$

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 [1310. 子数组异或查询](#)，难度为 **中等**。

Tag：「数学」、「树状数组」、「前缀和」

有一个正整数数组 `arr`，现给你一个对应的查询数组 `queries`，其中 `queries[i] = [Li, Ri]`。

对于每个查询 i ，请你计算从 L_i 到 R_i 的 XOR 值（即 $\text{arr}[L_i] \text{ xor } \text{arr}[L_i+1] \text{ xor } \dots \text{ xor } \text{arr}[R_i]$ ）作为本次查询的结果。

并返回一个包含给定查询 `queries` 所有结果的数组。

示例 1：

输入：`arr = [1,3,4,8]`，`queries = [[0,1],[1,2],[0,3],[3,3]]`

输出：`[2,7,14,8]`

解释：

数组中元素的二进制表示形式是：

$1 = 0001$

$3 = 0011$

$4 = 0100$

$8 = 1000$

查询的 XOR 值为：

$[0,1] = 1 \text{ xor } 3 = 2$

$[1,2] = 3 \text{ xor } 4 = 7$

$[0,3] = 1 \text{ xor } 3 \text{ xor } 4 \text{ xor } 8 = 14$

$[3,3] = 8$

示例 2：

输入：`arr = [4,8,2,10]`，`queries = [[2,3],[1,3],[0,0],[0,3]]`

输出：`[8,0,4,4]`

提示：

- $1 \leq \text{arr.length} \leq 3 * 10^4$
- $1 \leq \text{arr}[i] \leq 10^9$
- $1 \leq \text{queries.length} \leq 3 * 10^4$
- $\text{queries}[i].\text{length} == 2$
- $0 \leq \text{queries}[i][0] \leq \text{queries}[i][1] < \text{arr.length}$

刷题日记

公众号：宫水三叶的刷题日记

基本分析

令数组 `arr` 和数组 `queries` 的长度分别为 `n` 和 `m`。

`n` 和 `m` 的数据范围均为 10^4 ，因此 $O(m * n)$ 的暴力做法我们不用考虑了。

数据范围要求我们做到「对数复杂度」或「线性复杂度」。

本题主要利用异或运算中的「相同数值进行运算结果为 0」的特性。

对于特定数组 $[a_1, a_2, a_3, \dots, a_n]$ ，要求得任意区间 $[l, r]$ 的异或结果，可以通过 $[1, r]$ 和 $[1, l - 1]$ 的异或结果得出：

$$\text{xor}(l, r) = \text{xor}(1, r) \oplus \text{xor}(1, l - 1)$$

本质上还是利用集合（区间结果）的容斥原理。只不过前缀和需要利用「减法（逆运算）」做容斥，而前缀异或是利用「相同数值进行异或结果为 0（偶数次的异或结果为 0）」的特性实现容斥。

对于「区间求值」问题，之前在【题解】307. 区域和检索 - 数组可修改 也做过总结。

针对不同的题目，有不同的方案可以选择（假设有一个数组）：

1. 数组不变，求区间和：「前缀和」、「树状数组」、「线段树」
2. 多次修改某个数，求区间和：「树状数组」、「线段树」
3. 多次整体修改某个区间，求区间和：「线段树」、「树状数组」（看修改区间的数据范围）
4. 多次将某个区间变成同一个数，求区间和：「线段树」、「树状数组」（看修改区间的数据范围）

虽然「线段树」能解决的问题最多，但「线段树」代码很长，且常数很大，实际表现不算好。我们只有在不得不用的情况下才考虑「线段树」。

本题我们使用「树状数组」和「前缀和」来求解。

树状数组

使用「树状数组」分段记录我们某些区间的「异或结果」，再根据 `queries` 中的询问将分段

「异或结果」汇总（执行异或运算），得出最终答案。

代码：

```
class Solution {
    int n;
    int[] c = new int[100009];
    int lowbit(int x) {
        return x & -x;
    }
    void add(int x, int u) {
        for (int i = x; i <= n; i += lowbit(i)) c[i] ^= u;
    }
    int query(int x) {
        int ans = 0;
        for (int i = x; i > 0; i -= lowbit(i)) ans ^= c[i];
        return ans;
    }
    public int[] xorQueries(int[] arr, int[][] qs) {
        n = arr.length;
        int m = qs.length;
        for (int i = 1; i <= n; i++) add(i, arr[i - 1]);
        int[] ans = new int[m];
        for (int i = 0; i < m; i++) {
            int l = qs[i][0] + 1, r = qs[i][1] + 1;
            ans[i] = query(r) ^ query(l - 1);
        }
        return ans;
    }
}
```

- 时间复杂度：令 `arr` 数组长度为 `n`，`qs` 数组的长度为 `m`。创建树状数组复杂度为 $O(n \log n)$ ；查询的复杂度为 $O(m \log n)$ 。整体复杂度为 $O((n + m) \log n)$
- 空间复杂度： $O(n)$

前缀异或

「树状数组」的查询复杂度为 $O(\log n)$ ，而本题其实不涉及「修改操作」，我们可以使用「前缀异或」来代替「树状数组」。

虽说「树状数组」也有 $O(n)$ 的创建方式，但这里使用「前缀异或」主要是为了降低查询的复杂度。

代码：

```
class Solution {
    public int[] xorQueries(int[] arr, int[][] qs) {
        int n = arr.length, m = qs.length;
        int[] sum = new int[n + 1];
        for (int i = 1; i <= n; i++) sum[i] = sum[i - 1] ^ arr[i - 1];
        int[] ans = new int[m];
        for (int i = 0; i < m; i++) {
            int l = qs[i][0] + 1, r = qs[i][1] + 1;
            ans[i] = sum[r] ^ sum[l - 1];
        }
        return ans;
    }
}
```

- 时间复杂度：令 `arr` 数组长度为 `n`，`qs` 数组的长度为 `m`。预处理前缀和数组复杂度为 $O(n)$ ；查询的复杂度为 $O(m)$ 。整体复杂度为 $O(n + m)$
- 空间复杂度： $O(n)$

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 **1442. 形成两个异或相等数组的三元组数目**，难度为 **中等**。

Tag：「数学」、「前缀和」

给你一个整数数组 `arr`。

现需要从数组中取三个下标 `i`、`j` 和 `k`，其中 $(0 \leq i < j \leq k < arr.length)$ 。

`a` 和 `b` 定义如下：

- $a = arr[i] \oplus arr[i + 1] \oplus \dots \oplus arr[j - 1]$
- $b = arr[j] \oplus arr[j + 1] \oplus \dots \oplus arr[k]$

注意： \wedge 表示 按位异或 操作。

请返回能够令 $a == b$ 成立的三元组 (i, j, k) 的数目。

示例 1：

输入：`arr = [2,3,1,6,7]`

输出：`4`

解释：满足题意的三元组分别是 $(0,1,2)$ ， $(0,2,2)$ ， $(2,3,4)$ 以及 $(2,4,4)$

示例 2：

输入：`arr = [1,1,1,1,1]`

输出：`10`

示例 3：

输入：`arr = [2,3]`

输出：`0`

示例 4：

输入：`arr = [1,3,5,7,9]`

输出：`3`

示例 5：

输入：`arr = [7,11,12,9,5,2,7,17,22]`

输出：`8`

提示：

- $1 \leq \text{arr.length} \leq 300$
- $1 \leq \text{arr}[i] \leq 10^8$

宫水三叶

の

刷题日记

公众号：宫水三叶的刷题日记

基本分析

数据范围是 10^2 ，三元组包含 i 、 j 和 k 三个下标，因此通过「枚举下标」并「每次循环计算异或结果」的 $O(n^4)$ 朴素做法不用考虑了。

相信做过 [1310. 子数组异或查询](#) 的同学不难想到可以使用「树状数组」或者「前缀异或」来优化我们「每次循环计算异或结果」的过程。

由于不涉及修改操作，我们优先使用「前缀异或」。经过这样优化之后的复杂度是 $O(n^3)$ ，可以过。

前缀异或

预处理出「前缀异或」数组，并枚举三元组的下标。

本质上是利用集合（区间结果）的容斥原理。只不过前缀和需要利用「减法（逆运算）」做容斥，而前缀异或是利用「相同数值进行异或结果为 0（偶数次的异或结果为 0）」的特性实现容斥。

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 countTriplets(int[] arr) {
        int n = arr.length;
        int[] sum = new int[n + 1];
        for (int i = 1; i <= n; i++) sum[i] = sum[i - 1] ^ arr[i - 1];
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = i + 1; j <= n; j++) {
                for (int k = j; k <= n; k++) {
                    int a = sum[j - 1] ^ sum[i - 1];
                    int b = sum[k] ^ sum[j - 1];
                    if (a == b) ans++;
                }
            }
        }
        return ans;
    }
}

```

- 时间复杂度： $O(n^3)$
- 空间复杂度： $O(n)$

前缀异或 & 哈希表

我们重新审视一下这道题。

题目其实是要我们取得连续的一段区间 $[i, k]$ ，并在这一段中找到分割点 j ，使得区间内分割点左边的异或结果为 a ，分割点右边的异或结果为 b 。并最终让 a 和 b 相等。

由 a 与 b 相等，我们可以推导出 $a \oplus b = 0$ ，再结合 a 和 b 的由来，可以推导出 $[i, k]$ 连续一段的异或结果为 0。

再结合我们预处理的「前缀异或」数组，可得：

$$Xor(i, k) = sum[k] \oplus sum[i - 1] = 0$$

根据公式和「相同数值异或结果为 0」特性，我们可以知道 $sum[k]$ 和 $sum[i - 1]$ 数值相等，因此我们可以使用「哈希表」记录每个出现过的异或结果对应的下标集合，从而实现在确定 k 的情况下，通过 $O(1)$ 的复杂度找到所有符合条件的 i 。

需要注意的是，因为我们「前缀异或」数组的下标是从 1 开始，所以我们需要先往「哈希表」存入一个哨兵 0 作为边界，当然这一步不需要特殊操作，只需要让 k 从 0 开始执行循环即可（利用「前缀异或」数组中下标 0 的值本身为 0）。

代码：

```
class Solution {
    public int countTriplets(int[] arr) {
        int n = arr.length;
        // 预处理前缀异或数组
        int[] sum = new int[n + 1];
        for (int i = 1; i <= n; i++) sum[i] = sum[i - 1] ^ arr[i - 1];
        int ans = 0;
        // 记录出现过的异或结果，存储格式：{ 异或结果 : [下标1, 下标2 ...] }
        Map<Integer, List<Integer>> map = new HashMap<>();
        for (int k = 0; k <= n; k++) {
            List<Integer> list = map.getOrDefault(sum[k], new ArrayList<>());
            for (int idx : list) {
                int i = idx + 1;
                ans += k - i;
            }
            list.add(k);
            map.put(sum[k], list);
        }
        return ans;
    }
}
```

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 countTriplets(int[] arr) {
        int n = arr.length;
        // 事实上，甚至可以不预处理「前缀异或数组」，使用一个变量 xor 边遍历边计算即可
        int xor = 0, ans = 0;
        Map<Integer, List<Integer>> map = new HashMap<>();
        for (int k = 0; k <= n; k++) {
            if (k >= 1) xor ^= arr[k - 1];
            List<Integer> list = map.getOrDefault(xor, new ArrayList<>());
            for (int idx : list) {
                int i = idx + 1;
                ans += k - i;
            }
            list.add(k);
            map.put(xor, list);
        }
        return ans;
    }
}

```

- 时间复杂度： $O(n^2)$
- 空间复杂度： $O(n)$

**🔍 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 **1049. 最后一块石头的重量 II**，难度为 **中等**。

Tag：「动态规划」、「背包问题」、「01 背包」、「数学」

有一堆石头，用整数数组 stones 表示。其中 stones[i] 表示第 i 块石头的重量。

每一回合，从中选出任意两块石头，然后将它们一起粉碎。假设石头的重量分别为 x 和 y，且 $x \leq y$ 。那么粉碎的可能结果如下：

- 如果 $x == y$ ，那么两块石头都会被完全粉碎；
- 如果 $x \neq y$ ，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 $y - x$ 。

最后，最多只会剩下一块石头。返回此石头 最小的可能重量。如果没有石头剩下，就返回 0。

示例 1：

输入：stones = [2,7,4,1,8,1]

输出：1

解释：

组合 2 和 4，得到 2，所以数组转化为 [2,7,1,8,1]，

组合 7 和 8，得到 1，所以数组转化为 [2,1,1,1]，

组合 2 和 1，得到 1，所以数组转化为 [1,1,1]，

组合 1 和 1，得到 0，所以数组转化为 [1]，这就是最优值。

示例 2：

输入：stones = [31,26,33,21,40]

输出：5

示例 3：

输入：stones = [1,2]

输出：1

提示：

- $1 \leq \text{stones.length} \leq 30$
- $1 \leq \text{stones}[i] \leq 100$

基本分析

看到标题，心里咯噔了一下 🤔

一般性的石子合并问题通常是只能操作相邻的两个石子，要么是「区间 DP」要么是「四边形不等式」，怎么到 LeetCode 就成了中等难度的题目（也太卷了 😓）

仔细看了一下题目，可对任意石子进行操作，重放回的重量也不是操作石子的总和，而是操作石子的差值。

哦，那没事了～ 🤔

也是基于此启发，我们可以这样进行分析。

假设想要得到最优解，我们需要按照如下顺序操作石子：

$[(sa, sb), (sc, sd), \dots, (si, sj), (sp, sq)]$ 。

其中 $abcdijpq$ 代表了石子编号，字母顺序不代表编号的大小关系。

如果不考虑「有放回」的操作的话，我们可以划分为两个石子堆（正号堆/负号堆）：

- 将每次操作中「重量较大」的石子放到「正号堆」，代表在这次操作中该石子重量在「最终运算结果」中应用 $+$ 运算符
- 将每次操作中「重量较少/相等」的石子放到「负号堆」，代表在这次操作中该石子重量在「最终运算结果」中应用 $-$ 运算符

这意味我们最终得到的结果，可以为原来 $stones$ 数组中的数字添加 $+/-$ 符号，所形成的「计算表达式」所表示。

那有放回的石子重量如何考虑？

其实所谓的「有放回」操作，只是触发调整「某个原有石子」所在「哪个堆」中，并不会真正意义上的产生「新的石子重量」。

什么意思呢？

假设有起始石子 a 和 b ，且两者重量关系为 $a \geq b$ ，那么首先会将 a 放入「正号堆」，将 b 放入「负号堆」。重放回操作可以看作产生一个新的重量为 $a - b$ 的“虚拟石子”，将来这个“虚拟石子”也会参与某次合并操作，也会被添加 $+/-$ 符号：

- 当对“虚拟石子”添加 $+$ 符号，即可 $+(a - b)$ ，展开后为 $a - b$ ，即起始石子 a 和 b 所在「石子堆」不变
- 当对“虚拟石子”添加 $-$ 符号，即可 $-(a - b)$ ，展开后为 $b - a$ ，即起始石子 a 和 b 所在「石子堆」交换

因此所谓不断「合并」&「重放」，本质只是在构造一个折叠的计算表达式，最终都能展开扁平化为非折叠的计算表达式。

综上，即使是包含「有放回」操作，最终的结果仍然可以使用「为原来 $stones$ 数组中的数字添加 $+/-$ 符号，形成的“计算表达式”」所表示。

刷题日记

公众号：宫水三叶的刷题日记

动态规划

有了上述分析后，问题转换为：为 *stones* 中的每个数字添加 $+/ -$ ，使得形成的「计算表达式」结果绝对值最小。

与（题解）494. 目标和 类似，需要考虑正负号两边时，其实只需要考虑一边就可以了，使用总和 *sum* 减去决策出来的结果，就能得到另外一边的结果。

同时，由于想要「计算表达式」结果绝对值，因此我们需要将石子划分为差值最小的两个堆。

其实就是对「计算表达式」中带 $-$ 的数值提取公因数 -1 ，进一步转换为两堆石子相减总和，绝对值最小。

这就将问题彻底切换为 01 背包问题：从 *stones* 数组中选择，凑成总和不超过 $\frac{sum}{2}$ 的最大价值。

其中「成本」&「价值」均为数值本身。

整理一下：

定义 $f[i][j]$ 代表考虑前 i 个物品（数值），凑成总和不超过 j 的最大价值。

每个物品都有「选」和「不选」两种决策，转移方程为：

$$f[i][j] = \max(f[i-1][j], f[i-1][j - stones[i-1]] + stones[i-1])$$

与完全背包不同，01 背包的几种空间优化是不存在时间复杂度上的优化，因此写成 朴素二维、滚动数组、一维优化 都可以。

建议直接上手写「一维空间优化」版本，是其他背包问题的基础。

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 lastStoneWeightII(int[] ss) {
        int n = ss.length;
        int sum = 0;
        for (int i : ss) sum += i;
        int t = sum / 2;
        int[][] f = new int[n + 1][t + 1];
        for (int i = 1; i <= n; i++) {
            int x = ss[i - 1];
            for (int j = 0; j <= t; j++) {
                f[i][j] = f[i - 1][j];
                if (j >= x) f[i][j] = Math.max(f[i][j], f[i - 1][j - x] + x);
            }
        }
        return Math.abs(sum - f[n][t] - f[n][t]);
    }
}

```

```

class Solution {
    public int lastStoneWeightII(int[] ss) {
        int n = ss.length;
        int sum = 0;
        for (int i : ss) sum += i;
        int t = sum / 2;
        int[][] f = new int[2][t + 1];
        for (int i = 1; i <= n; i++) {
            int x = ss[i - 1];
            int a = i & 1, b = (i - 1) & 1;
            for (int j = 0; j <= t; j++) {
                f[a][j] = f[b][j];
                if (j >= x) f[a][j] = Math.max(f[a][j], f[b][j - x] + x);
            }
        }
        return Math.abs(sum - f[n&1][t] - f[n&1][t]);
    }
}

```

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 lastStoneWeightII(int[] ss) {
        int n = ss.length;
        int sum = 0;
        for (int i : ss) sum += i;
        int t = sum / 2;
        int[] f = new int[t + 1];
        for (int i = 1; i <= n; i++) {
            int x = ss[i - 1];
            for (int j = t; j >= x; j--) {
                f[j] = Math.max(f[j], f[j - x] + x);
            }
        }
        return Math.abs(sum - f[t] - f[t]);
    }
}

```

- 时间复杂度： $O(n * \sum_{i=0}^{n-1} stones[i])$
- 空间复杂度： $O(n * \sum_{i=0}^{n-1} stones[i])$

**🔍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 **1486. 数组异或操作**，难度为 **简单**。

Tag：「数学」、「模拟」

给你两个整数， n 和 $start$ 。

数组 $nums$ 定义为： $nums[i] = start + 2*i$ （下标从 0 开始）且 $n == nums.length$ 。

请返回 $nums$ 中所有元素按位异或（XOR）后得到的结果。

示例 1：

宫水三叶
の
刷题日记

公众号：宫水三叶的刷题日记

输入：n = 5, start = 0

输出：8

解释：数组 nums 为 [0, 2, 4, 6, 8]，其中 $(0 \wedge 2 \wedge 4 \wedge 6 \wedge 8) = 8$ 。
"^" 为按位异或 XOR 运算符。

示例 2：

输入：n = 4, start = 3

输出：8

解释：数组 nums 为 [3, 5, 7, 9]，其中 $(3 \wedge 5 \wedge 7 \wedge 9) = 8$ 。

示例 3：

输入：n = 1, start = 7

输出：7

示例 4：

输入：n = 10, start = 5

输出：2

提示：

- $1 \leq n \leq 1000$
- $0 \leq \text{start} \leq 1000$
- $n == \text{nums.length}$

模拟

数据范围只有 10^3 ，按照题目要求从头模拟一遍即可。

代码：

宫水三叶

刷题日记

公众号：宫水三叶的刷题日记

```

class Solution {
    public int xorOperation(int n, int start) {
        int ans = start;
        for (int i = 1; i < n; i++) {
            int x = start + 2 * i;
            ans ^= x;
        }
        return ans;
    }
}

```

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

数学

上述解法数据范围出到 10^8 大概率会发生 TLE。

如果数据范围出到 10^8 的话，本题难度应该会归为「中等」或「困难」。

事实上，本题存在「数学规律」解法。

原式为 $start \oplus (start + 2) \oplus (start + 4) \oplus \dots \oplus (start + 2 * (n - 1))$ 。

我们发现原式中只有数值 2 是固定系数（由题目给定），考虑将其进行提出。

得到新式子 $s \oplus (s + 1) \oplus (s + 2) \oplus \dots \oplus (s + (n - 1)) * 2$ ，其中 $s = start / 2$ 。

之所以进行这样的转换操作，是因为我们想要利用 $1 \oplus 2 \oplus 3 = 0$ 的异或性质。

但是转换到了这一步，我们发现「新式子」与「原式子」其实并不相等。

我们需要考虑两者之间的差值关系：

不难发现，将「原式」转化成「新式」的集体除以 2 的操作相当于将每个 *item* 的进行「右移一位」，同时「异或运算」是每位独立计算的，因此「右移一位」不会影响移动部分的计算结果。

本质上，「原式」转化成「新式」是将最终答案 `ans` 进了「右移」一位的操作。因此如果要重新得到 `ans`，我们需要将其重新「左移」一位，将最后一位异或结果补回。

即 `原式结果 = 新式结果 << 1 | e`， e 为最后一位异或结果（只能是 0 或者 1，其余高位为 0）。

我们重新观察「原式」发现式子中每个 *item* 奇偶性相同，这意味着其二进制的最低位相同。

根据 `n` 和 `start` 的奇偶数搭配，不难得最后一位 `e = n & start & 1`。

剩下的问题在于如何在不遍历的情况下计算「新式」结果，前面说到转化的目的是为了利用 $1 \oplus 2 \oplus 3 = 0$ 异或特性。

事实上，这个式子存在一般性的推广结论： $4i \oplus (4i + 1) \oplus (4i + 2) \oplus (4i + 3) = 0$ 。

因此只需要对最后一项进行 `%4` 讨论即可，这部分属于「结论」，详见代码的 `calc` 部分。

总结一下，假设我们最终的答案为 `ans`。整个处理过程其实就是把原式中的每个 *item* 右移一位（除以 2），计算 `ans` 中除了最低一位以外的结果；然后再将 `ans` 进行一位左移（重新乘以 2），将原本丢失的最后一位结果重新补上。补上则是利用了 `n` 和 `start` 的「奇偶性」的讨论。

代码：

```
class Solution {
    int calc(int x) {
        if (x % 4 == 0) return x;
        else if (x % 4 == 1) return 1;
        else if (x % 4 == 2) return x + 1;
        else return 0;
    }
    public int xorOperation(int n, int start) {
        // 整体除以 2，利用 %4 结论计算 ans 中除「最低一位」的结果
        int s = start >> 1;
        int prefix = calc(s - 1) ^ calc(s + n - 1);
        // 利用「奇偶性」计算 ans 中的「最低一位」结果
        int last = n & start & 1;
        int ans = prefix << 1 | last;
        return ans;
    }
}
```

- 时间复杂度： $O(1)$
- 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

题目描述

这是 LeetCode 上的 [1588. 所有奇数长度子数组的和](#)，难度为 简单。

Tag：「前缀和」、「数学」

给你一个正整数数组 `arr`，请你计算所有可能的奇数长度子数组的和。

子数组 定义为原数组中的一个连续子序列。

请你返回 `arr` 中 所有奇数长度子数组的和 。

示例 1：

输入：`arr = [1,4,2,5,3]`

输出：58

解释：所有奇数长度子数组和它们的和为：

`[1] = 1`

`[4] = 4`

`[2] = 2`

`[5] = 5`

`[3] = 3`

`[1,4,2] = 7`

`[4,2,5] = 11`

`[2,5,3] = 10`

`[1,4,2,5,3] = 15`

我们将所有值求和得到 $1 + 4 + 2 + 5 + 3 + 7 + 11 + 10 + 15 = 58$

示例 2：

输入：`arr = [1,2]`

输出：3

解释：总共只有 2 个长度为奇数的子数组，`[1]` 和 `[2]`。它们的和为 3。

示例 3：

宫水三叶
刷题日记

公众号：宫水三叶的刷题日记

输入：arr = [10,11,12]

输出：66

提示：

- $1 \leq \text{arr.length} \leq 100$
- $1 \leq \text{arr}[i] \leq 1000$

前缀和

枚举所有长度为奇数的子数组，我们可以通过「枚举长度 - 枚举左端点，并计算右端点」的两层循环来做。

而对于区间 $[l, r]$ 的和问题，可以直接再加一层循环来做，这样复杂度来到了 $O(n^3)$ ，但本题数据范围只有 100，也是可以过的。

对于此类区间求和问题，我们应当想到使用「前缀和」进行优化：使用 $O(n)$ 的复杂度预处理出前缀和数组，每次查询 $[l, r]$ 区间和可以在 $O(1)$ 返回。

代码：

```
class Solution {
    public int sumOddLengthSubarrays(int[] arr) {
        int n = arr.length;
        int[] sum = new int[n + 1];
        for (int i = 1; i <= n; i++) sum[i] = sum[i - 1] + arr[i - 1];
        int ans = 0;
        for (int len = 1; len <= n; len += 2) {
            for (int l = 0; l + len - 1 < n; l++) {
                int r = l + len - 1;
                ans += sum[r + 1] - sum[l];
            }
        }
        return ans;
    }
}
```

- 时间复杂度： $O(n^2)$

刷题日记

公众号：宫水三叶的刷题日记

- 空间复杂度： $O(n)$

数学

事实上，我们可以统计任意值 $arr[i]$ 在奇数子数组的出现次数。

对于原数组的任意位置 i 而言，其左边共有 i 个数，右边共有 $n - i - 1$ 个数。

$arr[i]$ 作为某个奇数子数组的成员的充要条件为：其所在奇数子数组左右两边元素个数奇偶性相同。

于是问题转换为如何求得「 $arr[i]$ 在原数组中两边连续一段元素个数为奇数的方案数」和「 $arr[i]$ 在原数组两边连续一段元素个数为偶数的方案数」。

由于我们已经知道 $arr[i]$ 左边共有 i 个数，右边共有 $n - i - 1$ 个数，因此可以算得组合数：

- 位置 i 左边奇数个数的方案数为 $(i + 1)/2$ ，右边奇数个数的方案数为 $(n - i)/2$ ；
- 位置 i 左边偶数（非零）个数的方案数为 $i/2$ ，右边偶数（非零）个数的方案数为 $(n - i - 1)/2$ ；
 - 考虑左右两边不选也属于合法的偶数个数方案数，因此在上述分析基础上对偶数方案数自增 1。

至此，我们得到了位置 i 左右奇数和偶数的方案数个数，根据「如果 $arr[i]$ 位于奇数子数组中，其左右两边元素个数奇偶性相同」以及「乘法原理」，我们知道 $arr[i]$ 同出现在多少个奇数子数组中，再乘上 $arr[i]$ 即是 $arr[i]$ 对答案的贡献。

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 sumOddLengthSubarrays(int[] arr) {
        int n = arr.length;
        int ans = 0;
        for (int i = 0; i < n; i++) {
            int l1 = (i + 1) / 2, r1 = (n - i) / 2; // 奇数
            int l2 = i / 2, r2 = (n - i - 1) / 2; // 偶数
            l2++; r2++;
            ans += (l1 * r1 + l2 * r2) * arr[i];
        }
        return ans;
    }
}

```

- 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 **1734. 解码异或后的排列**，难度为 **中等**。

Tag：「数学」、「异或」

给你一个整数数组 perm，它是前 n 个正整数的排列，且 n 是个奇数。

它被加密成另一个长度为 n - 1 的整数数组 encoded，满足 $encoded[i] = perm[i] \text{ XOR } perm[i + 1]$ 。比方说，如果 $perm = [1,3,2]$ ，那么 $encoded = [2,1]$ 。

给你 encoded 数组，请你返回原始数组 perm。题目保证答案存在且唯一。

示例 1：

输入：encoded = [3,1]

输出：[1,2,3]

解释：如果 $perm = [1,2,3]$ ，那么 $encoded = [1 \text{ XOR } 2, 2 \text{ XOR } 3] = [3,1]$

示例 2：

输入：encoded = [6,5,4,6]

输出：[2,4,1,5,3]

提示：

- $3 \leq n < 10^5$
- n 是奇数。
- $\text{encoded.length} == n - 1$

基本分析

我们知道异或运算有如下性质：

1. 相同数值异或，结果为 0
2. 任意数值与 0 进行异或，结果为数值本身
3. 异或本身满足交换律

本题与 [1720. 解码异或后的数组](#) 的主要区别是没有给出首位元素。

因此，求得答案数组的「首位元素」或者「结尾元素」可作为本题切入点。

数学 & 模拟

我们定义答案数组为 `ans[]`，`ans[]` 数组的长度为 n ，且 n 为奇数。

即有 $[ans[0], ans[1], ans[2], \dots, ans[n-1]]$ 。

给定的数组 `encoded[]` 其实是 $[ans[0] \oplus ans[1], ans[1] \oplus ans[2], \dots, ans[n-3] \oplus ans[n-2], ans[n-2] \oplus ans[n-1]]$ ，长度为 $n-1$ 。

由于每相邻一位会出现相同的数组成员 `ans[x]`，考虑“每隔一位”进行异或：

1. 从 `encoded[]` 的第 0 位开始，每隔一位进行异或：可得 $ans[0] \oplus ans[1] \oplus$

- $\dots \oplus ans[n - 2]$ ，即除了 `ans[]` 数组中的 $ans[n - 1]$ 以外的所有异或结果。
2. 利用 `ans[]` 数组是 n 个正整数的排列，我们可得 $ans[0] \oplus ans[1] \oplus \dots \oplus ans[n - 2] \oplus ans[n - 1]$ ，即 `ans[]` 数组中所有元素的异或结果。

将两式进行「异或」，可得 $ans[n - 1]$ 。

有了结尾元素后，问题变为与 1720. 解码异或后的数组 类似的模拟题。

代码：

```
class Solution {
    public int[] decode(int[] encoded) {
        int n = encoded.length + 1;
        int[] ans = new int[n];
        // 求得除了 ans[n - 1] 的所有异或结果
        int a = 0;
        for (int i = 0; i < n - 1; i += 2) a ^= encoded[i];
        // 求得 ans 的所有异或结果
        int b = 0;
        for (int i = 1; i <= n; i++) b ^= i;
        // 求得 ans[n - 1] 后，从后往前做
        ans[n - 1] = a ^ b;
        for (int i = n - 2; i >= 0; i--) {
            ans[i] = encoded[i] ^ ans[i + 1];
        }
        return ans;
    }
}
```

- 时间复杂度： $O(n)$
- 空间复杂度：构建同等数量级的答案数组。复杂度为 $O(n)$

**🔍 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 **1738. 找出第 K 大的异或坐标值**，难度为 **中等**。

Tag：「Top K」、「数学」、「前缀和」

给你一个二维矩阵 `matrix` 和一个整数 `k`，矩阵大小为 $m \times n$ 由非负整数组成。

矩阵中坐标 (a, b) 的值可由对所有满足 $0 \leq i \leq a < m$ 且 $0 \leq j \leq b < n$ 的元素 `matrix[i][j]`（下标从 0 开始计数）执行异或运算得到。

请你找出 `matrix` 的所有坐标中第 k 大的值（ k 的值从 1 开始计数）。

示例 1：

输入：`matrix = [[5,2],[1,6]]`，`k = 1`

输出：7

解释：坐标 $(0, 1)$ 的值是 $5 \text{ XOR } 2 = 7$ ，为最大的值。

示例 2：

输入：`matrix = [[5,2],[1,6]]`，`k = 2`

输出：5

解释：坐标 $(0, 0)$ 的值是 $5 = 5$ ，为第 2 大的值。

示例 3：

输入：`matrix = [[5,2],[1,6]]`，`k = 3`

输出：4

解释：坐标 $(1, 0)$ 的值是 $5 \text{ XOR } 1 = 4$ ，为第 3 大的值。

示例 4：

输入：`matrix = [[5,2],[1,6]]`，`k = 4`

输出：0

解释：坐标 $(1, 1)$ 的值是 $5 \text{ XOR } 2 \text{ XOR } 1 \text{ XOR } 6 = 0$ ，为第 4 大的值。

提示：

宫水三叶
刷题日记

公众号：宫水三叶的刷题日记

- $m == \text{matrix.length}$
- $n == \text{matrix}[i].\text{length}$
- $1 \leq m, n \leq 1000$
- $0 \leq \text{matrix}[i][j] \leq 10^6$
- $1 \leq k \leq m * n$

基本分析

根据题意，我们知道其实就是求「所有子矩阵中第 k 大的异或和」，同时规定所有子矩阵的左上角端点为 $(0, 0)$ 。

数据范围为 10^3 ，因此「枚举所有右下角」并「每次计算子矩阵异或和」的朴素做法 $O(m^2 * n^2)$ 不用考虑。

要在全局中找最优，「枚举所有右下角」过程不可避免，我们可以优化「每次计算子矩阵异或和」的过程。

这个分析过程与 [1310. 子数组异或查询](#) 类似。

异或作为不进位加法，可以利用「偶数次异或结果为 0」的特性实现类似「前缀和」的容斥。这使得我们可以在 $O(1)$ 的复杂度内计算「某个子矩阵的异或和」。

二维前缀异或 & 优先队列（堆）

创建二维数组 $sum[][]$ ，令 $sum[i][j]$ 为以 (i, j) 为右下角的子矩阵的异或和，我们可以得出计算公式：

$$sum[i][j] = sum[i-1][j] \oplus sum[i][j-1] \oplus sum[i-1][j-1] \oplus matrix[i-1][j-1]$$

剩下的问题是，如果从所有的「子矩阵异或和」找到第 k 大的值。

变成了 [Top K](#) 问题，可以使用「排序」或「堆」进行求解。

具体的，我们可以建立一个大小为 k 的小根堆，在计算二维前缀异或时，判断当前「子矩阵异或和」是否大于堆顶元素：

- 大于堆顶元素：当前子矩阵异或和可能是第 k 大的值，堆顶元素不可能为第 k 大的值。将堆顶元素弹出，并将当前子矩阵和加入堆中
- 小于堆顶元素：不会是第 k 大的值，直接丢弃。
- 等于堆顶元素：有相同元素在堆中，直接丢弃。

最终的堆顶元素即为答案。

代码：

```
class Solution {
    public int kthLargestValue(int[][] mat, int k) {
        int m = mat.length, n = mat[0].length;
        int[][] sum = new int[m + 1][n + 1];
        PriorityQueue<Integer> q = new PriorityQueue<>(k, (a, b) -> a - b);
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                sum[i][j] = sum[i - 1][j] ^ sum[i][j - 1] ^ sum[i - 1][j - 1] ^ mat[i - 1][j - 1];
                if (q.size() < k) {
                    q.add(sum[i][j]);
                } else {
                    if (sum[i][j] > q.peek()) {
                        q.poll();
                        q.add(sum[i][j]);
                    }
                }
            }
        }
        return q.peek();
    }
}
```

- 时间复杂度： $O(m * n * \log k)$
- 空间复杂度： $O(m * n)$

**🔍 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「数学」获取下载链接。

公众号: 宫水三叶的刷题日记

觉得专题不错，可以请作者吃糖🍬🍬🍬：



“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](#)。