宫水三叶的刷题日记

打表

Author　：宫水三叶
Date　　：2021/10/07
QQ Group：703311589
WeChat　：oaoaya

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「打表」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「打表」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「打表」；
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 🍭🍭🍭

> **🍭 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：

```
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} <= n <= 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：

```java
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：

```java
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：$O(1)$

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 **

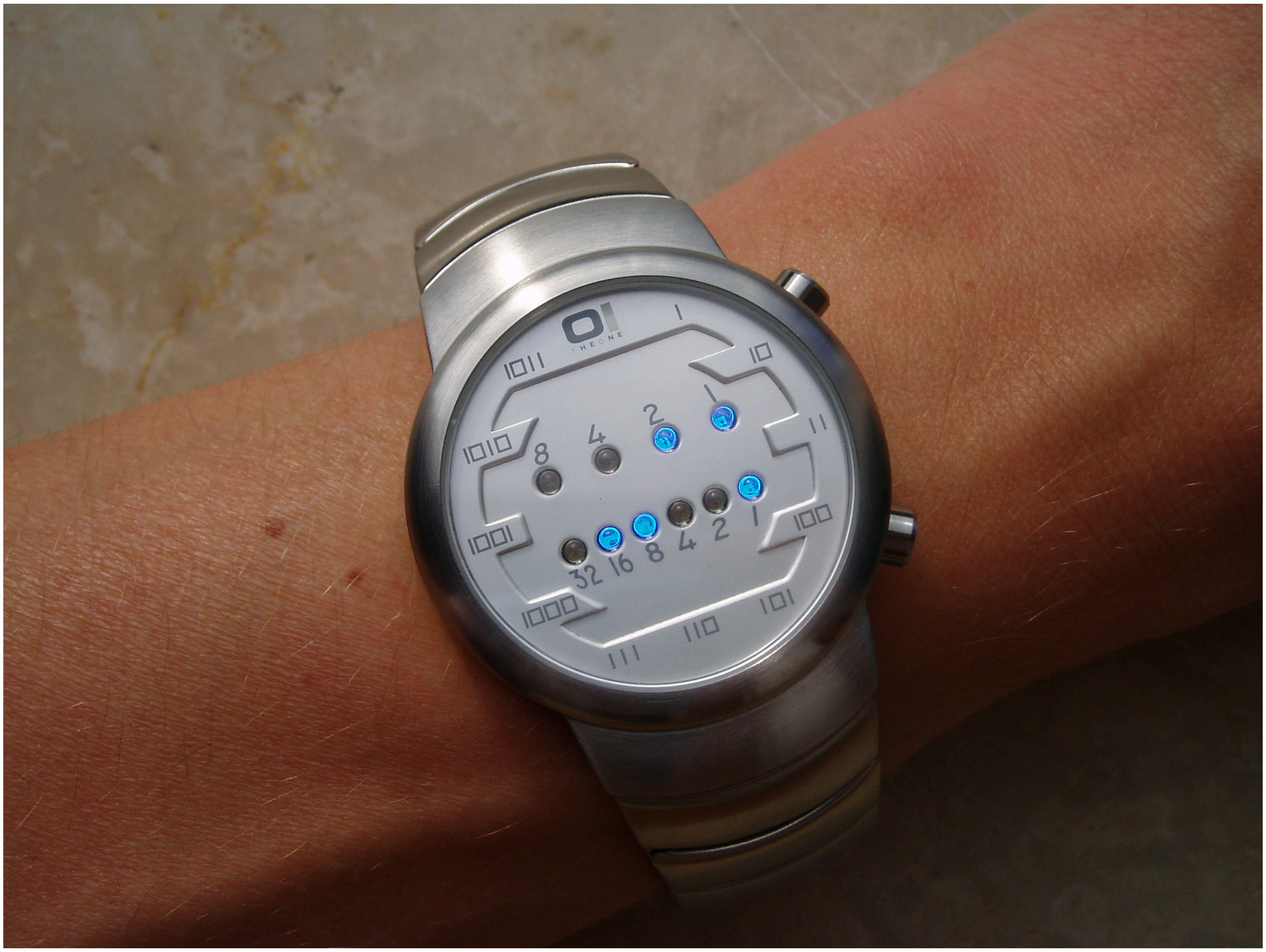
题目描述

这是 LeetCode 上的 **401. 二进制手表** ，难度为 **简单**。

Tag : 「打表」、「二进制」

二进制手表顶部有 4 个 LED 代表 小时（0-11），底部的 6 个 LED 代表 分钟（0-59）。每个 LED 代表一个 0 或 1，最低位在右侧。

例如，下面的二进制手表读取 “3:25” 。



（图源：WikiMedia - Binary clock samui moon.jpg ，许可协议：Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) ）

给你一个整数 turnedOn ，表示当前亮着的 LED 的数量，返回二进制手表可以表示的所有可能时间。你可以 按任意顺序 返回答案。

小时不会以零开头：

例如，“01:00” 是无效的时间，正确的写法应该是 “1:00” 。
分钟必须由两位数组成，可能会以零开头：

例如，“10:2” 是无效的时间，正确的写法应该是 “10:02” 。

示例 1：

输入：turnedOn = 1
输出：["0:01","0:02","0:04","0:08","0:16","0:32","1:00","2:00","4:00","8:00"]
示例 2：

输入：turnedOn = 9
输出：[]

提示：

- 0 <= turnedOn <= 10

打表

具体的，我们可以创建一个 `静态数据结构` 来存储打表信息（需确保全局唯一，即使存在多组测试数据只生成一次打表数据）。

然后在返回数据的时候直接 $O(1)$ 查表返回。

PS. 如果打表逻辑计算量接近 10^7 上限，可以考虑放到本地去做，这里数据量较少，直接放到 `static` 代码块去做即可。

代码：

```java
class Solution {
    // 打表逻辑，也可以放到本地做
    // 注意使用 static 修饰，确保打表数据只会被生成一次
    static Map<Integer, List<String>> map = new HashMap<>();
    static {
        for (int h = 0; h <= 11; h++) {
            for (int m = 0; m <= 59; m++) {
                int tot = getCnt(h) + getCnt(m);
                List<String> list = map.getOrDefault(tot, new ArrayList<String>());
                list.add(h + ":" + (m <= 9 ? "0" + m : m));
                map.put(tot, list);
            }
        }
    }
    static int getCnt(int x) {
        int ans = 0;
        for (int i = x; i > 0; i -= lowbit(i)) ans++;
        return ans;
    }
    static int lowbit(int x) {
        return x & -x;
    }
    public List<String> readBinaryWatch(int t) {
        return map.getOrDefault(t, new ArrayList<>());
    }
}
```

- 时间复杂度：$O(1)$
- 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 **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 <= n <= 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 $j <= i$。

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

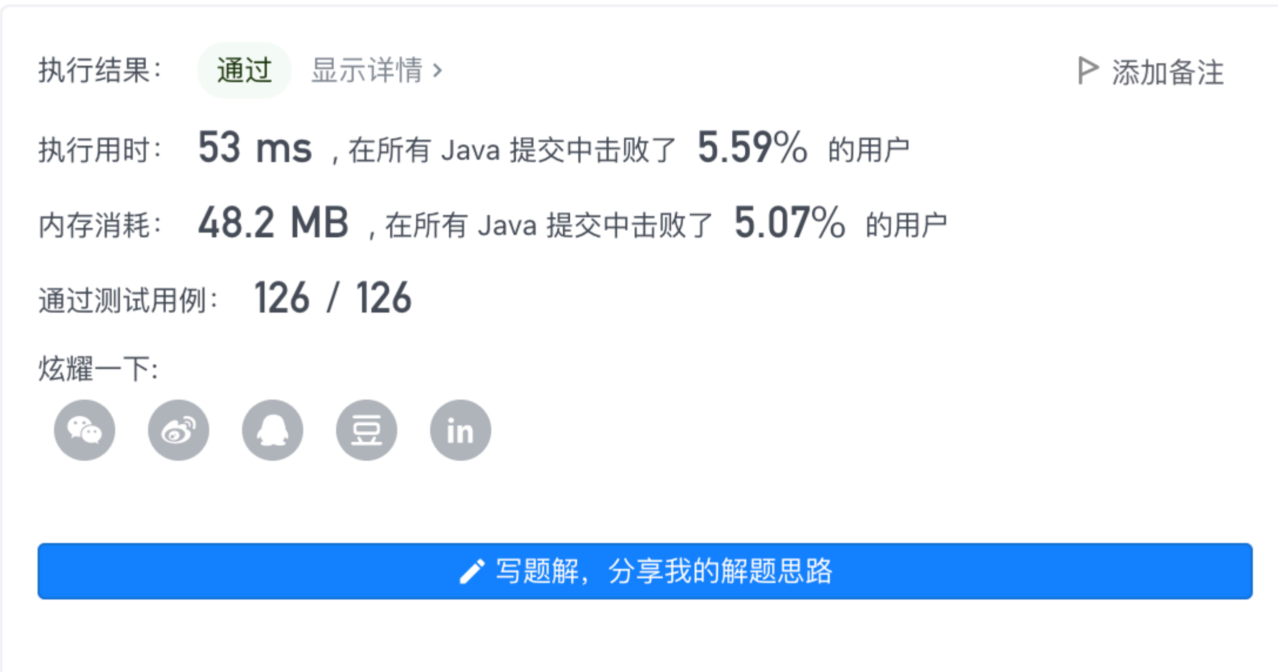
$f[i][j] = f[i-j][j] + 1$；

- 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 $f[i][j] = \min(f[i][x] + 1), 0 \le x < i$。

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。



代码：

```java
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）
   ...

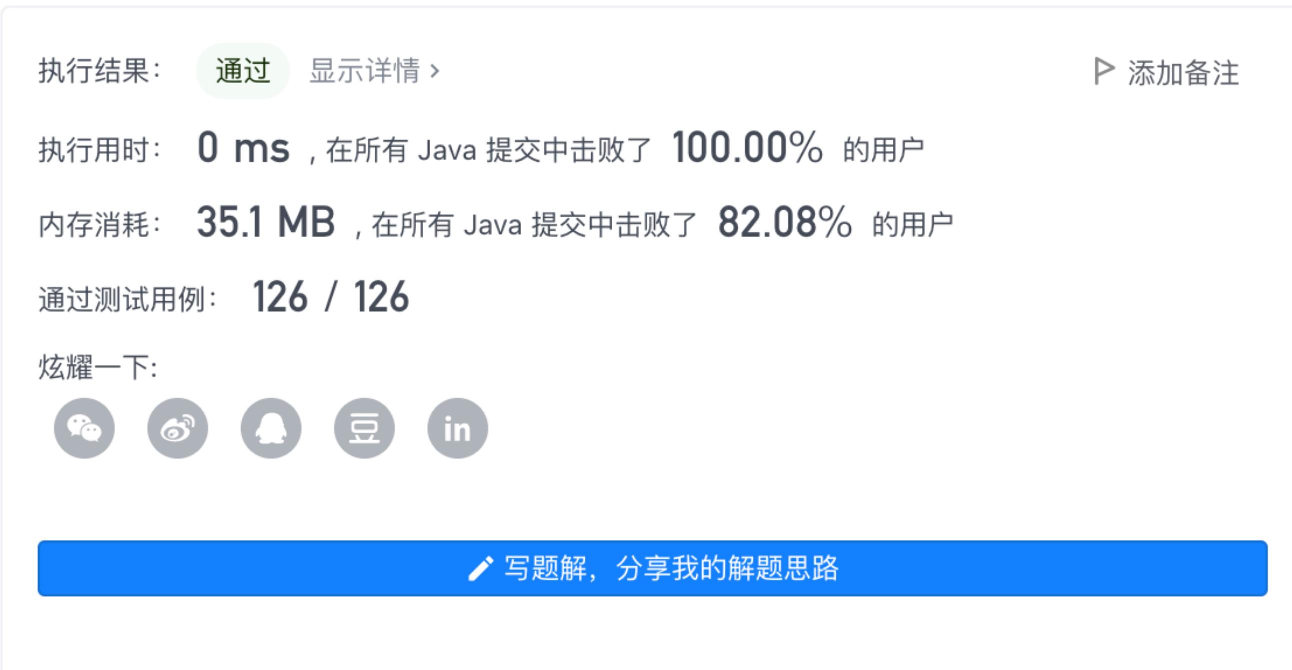
最终经过 k 次“动作”之后，得到长度为 n 的记事本长度，即有：

$$n = k_1 * k_2 * ... * k_x$$

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

对于任意一个 k_i（合数）而言，根据定理 $a * b >= a + b$ 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。



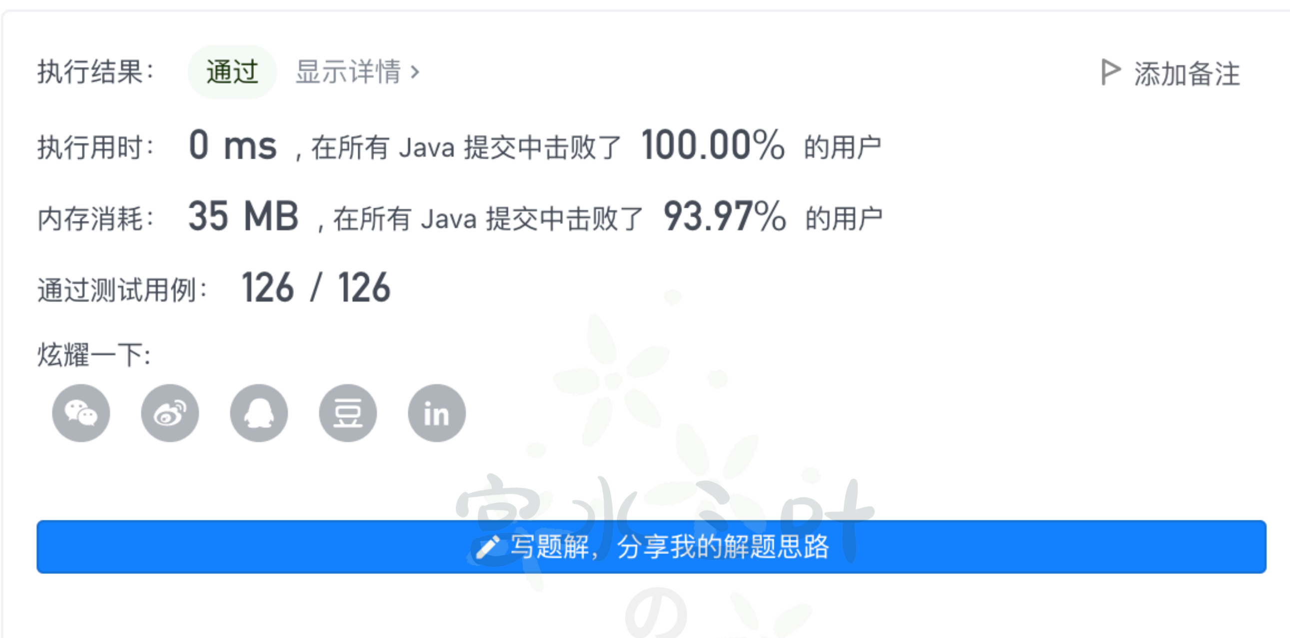
代码：

```java
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。



代码：

```java
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];
    }
}
```

```java
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,
    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 **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 Tn 定义如下：

T0 = 0, T1 = 1, T2 = 1, 且在 n >= 0 的条件下 Tn+3 = Tn + Tn+1 + Tn+2

给你整数 n，请返回第 n 个泰波那契数 T_n 的值。

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 <= n <= 37
- 答案保证是一个 32 位整数，即 answer <= 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：

```java
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：

```java
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：$O(n)$

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（$i >= 3$）：

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

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：

```java
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 **1646. 获取生成数组中的最大值** ，难度为 **中等**。

Tag : 「模拟」、「打表」

给你一个整数 n 。按下述规则生成一个长度为 n + 1 的数组 nums ：

- nums[0] = 0
- nums[1] = 1
- 当 2 <= 2 * i <= n 时，nums[2 * i] = nums[i]
- 当 2 <= 2 * i + 1 <= n 时，nums[2 * i + 1] = nums[i] + nums[i + 1]

返回生成数组 nums 中的 最大 值。

示例 1：

```
输入：n = 7

输出：3

解释：根据规则：
  nums[0] = 0
  nums[1] = 1
  nums[(1 * 2) = 2] = nums[1] = 1
  nums[(1 * 2) + 1 = 3] = nums[1] + nums[2] = 1 + 1 = 2
  nums[(2 * 2) = 4] = nums[2] = 1
  nums[(2 * 2) + 1 = 5] = nums[2] + nums[3] = 1 + 2 = 3
  nums[(3 * 2) = 6] = nums[3] = 2
  nums[(3 * 2) + 1 = 7] = nums[3] + nums[4] = 2 + 1 = 3
因此，nums = [0,1,1,2,1,3,2,3]，最大值 3
```

示例 2：

```
输入：n = 2

输出：1

解释：根据规则，nums[0]、nums[1] 和 nums[2] 之中的最大值是 1
```

示例 3：

```
输入：n = 3

输出：2

解释：根据规则，nums[0]、nums[1]、nums[2] 和 nums[3] 之中的最大值是 2
```

提示：

- 0 <= n <= 100

模拟

按照题意模拟一遍，得到数列 $nums$，再从 $nums$ 中找出最大值即可。

代码：

```
class Solution {
    public int getMaximumGenerated(int n) {
        if (n == 0) return 0;
        int[] nums = new int[n + 1];
        nums[0] = 0;
        nums[1] = 1;
        for (int i = 0; i < n; i++) {
            if (2 * i <= n) nums[2 * i] = nums[i];
            if (2 * i + 1 <= n) nums[2 * i + 1] = nums[i] + nums[i + 1];
        }
        int ans = 0;
        for (int i : nums) ans = Math.max(ans, i);
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

打表

利用数据范围，可以直接使用 `static` 进行打表构造。

代码：

```java
class Solution {
    static int N = 110;
    static int[] nums = new int[N];
    static {
        nums[0] = 0;
        nums[1] = 1;
        for (int i = 0; i < N; i++) {
            if (2 * i < N) nums[2 * i] = nums[i];
            if (2 * i + 1 < N) nums[2 * i + 1] = nums[i] + nums[i + 1];
        }
        for (int i = 0, max = 0; i < N; i++) {
            nums[i] = max = Math.max(max, nums[i]);
        }
    }
    public int getMaximumGenerated(int n) {
        return nums[n];
    }
}
```

- 时间复杂度：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 **

💡**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」，回台回复「打表」获取下载链接。

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