宫水三叶的刷题日记

矩阵快速幂

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「矩阵快速幂」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「矩阵快速幂」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「矩阵快速幂」；
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 **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：

- 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 <= n <= 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：

```
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：

```java
// 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;
    }
}
```

```
// 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]) % m
                    if (k != 2) f[i + 1][j][k + 1] = (f[i + 1][j][k + 1] + f[i][j][k]) % m
                    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）：

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

...

- $idx = 5$ ：$acnt = 1$、$lcnt = 2$；

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：

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

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：

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

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：

```java
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;
    }
}
```

- 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 **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：

```
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：

```
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 **

💡 **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「矩阵快速幂」获取下载链接。

觉得专题不错，可以请作者吃糖🍭🍭🍭：