汉诺塔

题目大意

给定三个栈 A, B, C 和两个可以暂存一个元素的变量 L, R（相当于两只手）。初始时栈 A 中有 n 个元素（从栈底到栈顶单调递减，分别为 $n, n-1, \ldots, 1$），其余栈和变量均为空。要求在任意时刻各栈内都满足“越靠近栈底的元素越大”的单调性。

每次可以执行两种操作：拿起（从非空栈顶弹出一个元素，暂存到一只空闲的手中）和放下（将手中的元素压入某个栈中，需满足该栈为空或栈顶元素大于要压入的元素）。求将所有元素从 A 移动到 C 所需的最少“拿起”操作次数，结果对 998244353 取模。

解题思路

这道题的核心难点在于如何处理多出的两只手（容量为 1 的栈）以及解决在递推过程中的取模比较问题。本题的正解其实是**纯递推**，我们将推导过程分为动机分析、分治转移模型、物理限制推演，以及如何突破取模困境得出最终的线性递推关系四个部分。

1. 动机分析与核心思想

本题本质上是一个变形的多柱汉诺塔问题。经典汉诺塔中有多个容量无限的柱子，而本题中我们有三个标准栈 A, B, C 和两个容量仅为 1 的“退化柱” L, R。

面对这类求最少步数的问题，如果直接通过状态转移搜索（BFS）求解，状态空间呈指数级爆炸，对于 10^6 的数据必定超时。多柱汉诺塔问题的经典解法（即 Frame-Stewart 算法）其核心思想在于：**将整个盘子堆分为上下两部分，先借用所有空间把上部移走腾出空间，再把下部移到目标位置，最后把上部盖回来**。我们需要将这种分治递推结构完美映射到本题的条件中。

2. 经典分治法在变形汉诺塔中的映射

根据 Frame-Stewart 算法的框架，我们要把 A 栈中 n 个盘子全部移到 C 栈，可以分为以下三个阶段：

- **第一阶段（挪开上部大山）**：我们先挑出最顶部的 $n-m$ 个较小的盘子。利用场上所有的空间（栈 B、C 以及两只手 L、R），将这 $n-m$ 个盘子从 A 栈整体搬运并堆叠到 B 栈上。因为这本质上就是规模为 $n-m$ 的原问题（只是目标从 C 换成了 B），所以这一步所需的最少“拿起”次数就是 f_{n-m}。
- **第二阶段（转移底部大盘）**：经过第一阶段，栈 B 已经被较小的 $n-m$ 个盘子完全占据了。由于汉诺塔“大盘绝不能压小盘”的死规定，后续在移动剩下那些更大的盘子时，**任何大盘子都绝对不能往 B 栈里放**。这就相当于栈 B 在这个阶段被彻底“封印”了。我们只能利用剩下的栈 A、C 和两只手 L、R，把最底部的 m 个大盘子从 A 栈搬到 C 栈。我们设这种“废掉一个柱子”情况下的搬运代价为 g_m。
- **第三阶段（大山归位）**：底部大盘就位后，我们再次利用所有空间（因为大盘在 C 的底部，不影响小盘放上去），把 B 栈上的那 $n-m$ 个小盘子整体搬运到 C 栈的顶部。这同样是一个规模为 $n-m$ 的原问题，耗费次数依然是 f_{n-m}。

这三个阶段衔接在一起，总的“拿起”次数就等于 $f_{n-m} + g_m + f_{n-m} = 2 \times f_{n-m} + g_m$。为了找出将 n 个盘子移走的最优方案，我们显然需要尝试所有可能的划分点 m，看看究竟一次性把几个大盘子留在底部转移，才能让总代价最小：

$$f_n = \min_m \{2 \times f_{n-m} + g_m\}$$

3. 物理极限界限：如何用两只手极限转移 4 个盘子？

在第二步禁用 B 栈的情况下，仅靠 A, C 和两只手 L, R，最多能移动多少个大盘子？直觉上，废掉一根柱子，手里只能拿 2 个，加上目标柱 C 必须空着接收最大的盘子，最多只能转移 $2+1=3$ 个。但这是一个巨大的物理盲点：**源柱 A 本身也是可以作为中转站的！**只要严格遵守“大盘在下，小盘在上”，在最大盘子移动的瞬间，我们可以把小盘子反向压在 A 栈上。

为了打破直觉盲区，我们来硬核推演一遍，如何极限榨取空间，将 4 个盘子（从底到顶为 $4, 3, 2, 1$）从 A 移到 C：

第一阶段：避难

- 左手 L 拿 1，右手 R 拿 2。（此时 A 剩 $[4, 3]$）
- R 把 2 放入 C，L 把 1 放入 C。（此时 C 有 $[2, 1]$，两手腾空！）

第二阶段：抽出最大盘

- L 拿 3，R 拿最大的 4。
- **关键点**：此时 A 彻底空了！最大盘 4 终于被拿在了右手 R 里！

第三阶段：极限微操，清空 C 栈现在 R 拿着 4，L 拿着 3，C 里面有 $[2,1]$。我们必须把 4 放进 C，但两手已满，C 又被占了，怎么办？利用空出来的 A！

- L 把 3 放入空的 A 栈。（此时 A 有 $[3]$，左手 L 腾空！）
- L 从 C 拿起 1。
- L 把 1 放入 A 栈。（压在 3 上面，完全合法！此时 A 有 $[3,1]$，L 再次腾空！）
- L 从 C 拿起 2。
- **见证奇迹的时刻**：此时 C 彻底空了！L 拿着 2，R 拿着 4，A 里安全停放着 $[3,1]$。

第四阶段：最大盘归位与收尾

- R 把最大的 4 稳稳地放入空无一物的 C 栈底部！（C 有 $[4]$，R 腾空）
- L 把 2 放入 C 栈。（C 有 $[4,2]$，L 腾空）
- 最后利用两只手，把 A 栈里的 $[3,1]$ 倒腾回 C 栈即可。

为什么不能是 5 个盘子？（>4 会导致死锁的原因）在这个极限微操中，之所以没有卡死，是因为从 C 往 A 退回盘子时，我们只退回了盘子 1，压在了盘子 3 上面（$1<3$，完全合法）。如果一开始试图转移的是 5 个盘子，那么在“清空 C 栈”阶段，C 里面将会有 $[3,2,1]$。当我们需要把它们退回 A 栈时，我们得先拿 1 压在空出来的盘子 4 上，然后再拿 2 压在 1 上——**盘子 2 压在盘子 1 上，大盘压小盘，直接违规死锁！**也就是说，在反向退回 A 栈时，由于只能一个一个倒序拿，导致上层的盘子一定比下层的大，A 栈在这个瞬间**最多只能安全地停放 1 个盘子**。加上手中能拿的 2 个盘子，以及正在转移的最大盘，物理极限刚好就是 $1+2+1=4$ 个！

我们通过简单的搜索即可得到 $m \le 4$ 时的常数代价：

$$g_1 = 1, \quad g_2 = 2, \quad g_3 = 5, \quad g_4 = 10$$

代入转移方程，得到了常数状态的动态规划式：

$$f_n = \min_{1 \le m \le 4} \{2 \times f_{n-m} + g_m\}$$

4. 纯递推的推导与取模困境的破局

似乎到这里，我们直接写一个 $\mathcal{O}(N)$ 的循环就能通过 10^6 的数据了。但这里隐藏着一个极其致命的陷阱：**取模运算与 min 操作是不兼容的！**

因为 f_n 呈指数级增长，很快就会溢出 。如果我们为了防止溢出而在中间对结果取模，那么取模后的值再送入 min 进行大小比较，纯属荒谬（比如真实的 $10^{10} \bmod MOD$ 可能会比真实的 10 还要小，导致转移决策严重错误）。

为了解决这个取模的死结，我们需要关注递推的实质：**到底在转移中，是哪个 m 取到了最小值？**

我们可以用不取模的程序打印出前几十项的真实最优决策情况：当 $n \le 7$ 时，转移处于边界调整期，其最优解为：$1,2,4,6,9,13,17$。但是，**当 $n \ge 8$ 时，无一例外，所有的最优转移全部来自于 $m=4$！**

为什么最优决策总是 $m=4$ 呢？其实逻辑非常直观：每挪开一次上面的“大山”（即顶部的 $n-m$ 个盘子，耗费 $2 \times f_{n-m}$），我们就应该尽可能多地把底部的盘子一次性全塞过去。而底部的容量物理极限就是 4 个盘子，代价仅仅是 $g_4 = 10$。相较于呈指数级暴涨的 f_{n-m}，“挪开大山”的代价过于高昂，因此最大化利用底部的极限物理空间（即永远选择 $m=4$）绝对是最优的贪心策略。

这就引出了本题最终的**纯递推公式**：当 $n \ge 8$ 时，状态转移方程完全去除了不可取模的 min 函数，退化为一个确定的、常系数线性递推方程：

$$f_n = 2 \times f_{n-4} + 10$$

有了这个严格的递推关系，取模难题迎刃而解！因为递推式中不再有任何比较操作，我们可以毫无顾忌地在每一步转移后直接取模：

$$f_n = (2 \times f_{n-4} + 10) \pmod{998244353}$$

至此，我们得到了一个完美且大道至简的 $\mathcal{O}(N)$ 纯递推解法。对于 $N = 10^6$ 的数据规模，$\mathcal{O}(N)$ 的时间复杂度能够在几毫秒内轻松跑完。即便后续遇到恶意的 $N = 10^{18}$ 的加强版数据，由于这是一个常系数线性递推，我们也可以无缝衔接**矩阵快速幂**进行对数级别的加速。但在本题的数据范围内，朴素的纯递推循环已经是最契合、最漂亮的解法。

代码实现

完整代码供以验证思路：

```cpp
#include <iostream>
#include <vector>

using namespace std;

const int MOD = 998244353;
const int MAXN = 1000000;

vector<long long> f(MAXN + 5);

void init() {
    // 阶段一：前 7 项为无规律的边界状态，Hardcode 写入
    f[1] = 1;
    f[2] = 2;
    f[3] = 4;
    f[4] = 6;
    f[5] = 9;
    f[6] = 13;
    f[7] = 17;

    // 阶段二：n >= 8 时，进入严格的线性递推状态
    // 最优决策固定为 m = 4，彻底避免了 min 操作与取模冲突的问题
    for (int i = 8; i <= MAXN; i++) {
        f[i] = (2 * f[i - 4] + 10) % MOD;
    }
}

void solve() {
    int n;
    cin >> n;
    cout << f[n] << "\n";
}

int main() {
    // 优化输入输出
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    // 预处理打表，O(N) 纯递推
    init();

    int t;
    if (cin >> t) {
        while (t--) {
            solve();
        }
    }
    return 0;
}
```

复杂度分析

- **时间复杂度**：通过固定最优决策点，我们将不可取模的 min 动态规划简化为了 $\mathcal{O}(N)$ 的纯线性递推。预处理阶段需执行 N 次循环，时间复杂度为 $\mathcal{O}(N)$。后续每次查询直接利用数组寻址，时间复杂度为 $\mathcal{O}(1)$。总时间复杂度为 $\mathcal{O}(N+T)$，对于 $N = 10^6$ 的数据规模，可以在毫秒级别内跑完。
- **空间复杂度**：需要一个大小为 N 的一维数组来存储所有的答案表，空间复杂度为 $\mathcal{O}(N)$，占用不到 10 MB，远低于题目 65536 KB 的空间限制。