## 2. 扩展的汉诺塔问题

### 2.1 问题分析

该问题的关键在于假设： 对于n个圆盘r根柱子的情况，可以先将m(1 <= m < n)个圆盘移动到第二根柱子上，再将n-m个圆盘移动到最后一根柱子上，然后再将m个圆盘从第二根柱子移动到最后一根柱子上。于是可以将原先规模为n个圆盘r根柱子的问题，分解为两个规模为m个圆盘r根柱子的问题和一个规模为n-m个圆盘r-1个柱子的问题。当问题规模小到只有三根柱子时，就可以按经典的汉诺塔问题求解。

问题的目标是找到最少移动次数的移动步骤。在假设的三步移动过程中，第一、三步的移动次数相同，如果用f(n,r)表示n个圆盘r根柱子的最小移动次数，那么可得递推式f(n,r)=min{2\*f(m,r)+f(n-m,r-1)} ，满足最优子结构。

此外，当柱子数为3时，为经典的汉诺塔问题，最少移动次数一定为2^盘子数-1。对于柱子数>3时，当只有1个盘子时，一定只要移动一次，2个盘子时，一定只要移动3次。我们就有了初始条件，如果我们设置一个二维数组dp来存放最少移动次数，使得行表示柱子数，列表示盘子数，第一行表示3个柱子，第一列表示1个盘子，即dp[r-3][n-1] = f(n,r)，那么我们就可以使用动态规划，从左往右，从上到下求出所有子问题的最少移动次数，直到求出f(n,r)。另外为了记录每个子问题最佳的移动盘子数量m，我们可以设置同样大小的二维数组best来记录每次得到的最优m。

经过动态规划求解得到两个数组后，我们就可以列举出最少次数条件下的移动步骤。利用递归方法，从问题规模为（n,r）开始自顶向下分解，直到可以用经典汉诺塔问题的方法来移动。

### 2.2 算法设计

根据以上的分析，我们的算法可以分为两部分进行：

1.动态规划求解最少移动次数和最优条件下移动盘子数。

2.根据前面的求解结果递归输出移动步骤。

#### 2.2.1模块设计

与算法相关的模块有hanoi和entity，以下列出了它们包含的类和对应的作用。

|  |  |
| --- | --- |
| Hanoi模块：算法求解 | |
| 类名 | **作用** |
| HanoiDP | 动态规划求最少移动和每次移动的盘子数 |
| HanoiInput | 封装算法的输入和输出 |
| HanoiSolver | 根据动态规划结果输出步骤和结果 |
| Peg | 存储柱子上的盘子编号和柱子的信息 |

|  |  |
| --- | --- |
| Entity模块：结果数据封装 | |
| 类名 | **作用** |
| HanoiResult | 保存结果和步骤信息，传给前端 |
| HanoiStep | 保存每一个步骤信息 |

#### 2.2.2 算法的输入

算法的输入包含柱子的个数和盘子的个数，并且柱子数范围：[6, 10]，盘子个数范围：[50，90]。

我们编写了HanoiInput类负责处理算法输入。类包含成员变量分别表示柱子数、盘子数、柱子上下限、盘子上下限。上下限默认为题目要求范围，也可以自行更改。

调用input(num\_tower,num\_disk)方法来输入数值赋值给成员变量。同时方法会检查输入数值的合法性，即是否在限定范围内：如果输入合法，方法返回true；否则返回false，并在结果中添加错误信息返回给前端。

完成输入后，调用getResult()方法来创建HanoiSolver求解器执行求解过程，并返回求解结果。

#### 2.2.3 动态规划求最优解

完成输入后，由HanoiSolver开始执行完整求解过程。

首先初始化柱子状态，我们用Peg类来封装每个柱子的id、是否有盘子和一个用来保存柱子上盘子的栈，设定起始柱子的id为0，并将所有盘子都加入到起始柱子上，编号小的在上，设置状态为非空。然后初始化其他的柱子，一开始都为空。

然后调用HanoiDP对象求解最优解。根据前面的分析，我们采用动态规划求解，求解过程如下：

首先创建两个二维数组dpMove和dpBest分别用于保存最少移动次数和最少移动次数条件下的移动数量，大小均为(柱子数-2)\*盘子数，其中x行y列代表柱子数为x+3，盘子数为y+1的情况。

随后设置初始值，根据前面的分析，当柱子数为3时，最少移动次数一定为2^盘子数-1。对于柱子数>3时，当只有1个盘子时，一定只要移动一次，2个盘子时，一定只要移动3次。所以我们可以设置dpMove的第0行为2^(i+1)-1（i表示列号），设置第0列为1，第1列为3。

然后调用leastMoves (int disks, int pegs)方法利用动态规划填充两个数组。方法包含两个参数：disk——剩余盘子数，pegs——可用柱子数，结果返回当前情况的最少移动次数。方法首先进行越界检查，防止数组越界，如果disk<0或者pegs<3则返回-1。然后检查数组中是否已经计算好当前情况的最少移动次数，若是则直接返回结果。

随后根据递推方程f(n,r)=min{2\*f(m,r)+f(n-m,r-1)}（n为盘子数，r为柱子数，m为假设中一步移动的盘子数，1<=m<n），我们要遍历所有情况以找到最小移动次数。我们从一次移动n-1个盘子开始循环遍历到1个盘子，寻找其中最小的情况，并记录到数组中。此过程代码如下：

1. double least\_moves;

2. int moving\_disk;

3. /\*第一步和第三步相当于求解m个圆盘r根柱子的问题，第二步相当于求解n-m个圆盘r-1根柱子的问题\*/

4. least\_moves = 2\*leastMoves(disks-1,pegs) + leastMoves(1,pegs-1);

5. moving\_disk = disks-1;

6. /\* 遍历所有可能的情况以找到最佳情况 \*/

7. for (int r=disks-2;r>0;--r){

8. double moves = 2\*leastMoves(r,pegs) + leastMoves(disks-r,pegs-1);

9. if(moves<0){

10. /\*数据过大发生了数据溢出,跳过本次\*/

11. continue;

12. }

13. /\* 更新最佳解 \*/

14. if (moves<least\_moves){

15. least\_moves=moves;

16. moving\_disk = r;

17. }

18. }

19. /\* 更新数组并返回结果 \*/

20. dpBest[pegs-3][disks-1] = moving\_disk;

21. dpMove[pegs-3][disks-1] = (int)least\_moves;

22.

在实际测试过程中，发现出现了数据溢出的问题，于是增加了判断moves<0为溢出的情况，跳过本次循环。

#### 2.2.4 递归输出移动步骤

在完成上一步后，我们得到了最少移动次数和对应的移动盘子数，我们便可以据此来输出每一次移动的步骤。

我们定义了getSteps(int disk, int pegs, int sourceId, int buffId, int destId)方法来输出所有步骤。方法参数定义如下：

* disk 需要移动的盘子数
* pegs 包含的柱子数
* sourceId 起始柱子id
* buffId 中转柱子id
* destId 目标柱子id

初始参数为(盘子总数， 柱子总数,0, 1 ，柱子总数 -1)，表示从第一根柱子移动全部盘子到最后一根柱子。方法代码如下：

1. public void getSteps(int disk, int pegs, int sourceId, int buffId, int destId){

2. /\* 初始化 buffId, buffId 代表用作中转的柱子的id \*/

3. int bufferId = buffId;

4. /\*moveDisk表示需要移动多少个盘子\*/

5. int moveDisk = bestResult.dpBest[pegs - 3][disk - 1];

6. /\* 如果盘子小于等于2个或者柱子小于等于3个或者移动的盘子数<1 就没必要再寻找下一次需要移动的盘子数,使用经典的汉诺塔解法即可 \*/

7. if (disk <= 2 || pegs <= 3 || moveDisk < 1) {

8. /\* 按经典汉诺塔问题来求解 \*/

9. classicSolve(disk, TOTAL\_PEG[sourceId], TOTAL\_PEG[bufferId], TOTAL\_PEG[destId]);

10. }

11. else {

12. /\*从起始柱子移动moveDisk个盘子到中转柱子\*/

13. getSteps(moveDisk, pegs, sourceId, destId, bufferId);

14. /\*

15. 完成第一步移动后，需要指定新的中转柱子

16. 此时原先的中转柱子已经使用

17. 所以要找到下一步移动用的中转柱子

18. \*/

19. int newBufferId=bufferId;

20. if(!TOTAL\_PEG[bufferId].isEmpty()){

21. if(getAvailableBuff(destId).length!=0)

22. newBufferId = getAvailableBuff(destId)[0];

23. }

24. /\* 再将起始柱子上disk - moveDisk个盘子移到目标柱子上 \*/

25. getSteps(disk - moveDisk, pegs - 1, sourceId, newBufferId, destId);

26.

27. /\* 最后将中转柱子上的moveDisk个柱子移动到目标柱子 \*/

28. getSteps(moveDisk, pegs, bufferId, sourceId, destId);

29. }

30.

31. }

32.

算法首先获得dpBest数组保存的对应的移动的盘子数。

然后判断问题规模，如果盘子数<=2或者柱子数<=3或者移动盘子数为0，都表示此时问题规模已经是经典的汉诺塔问题，可以直接用经典汉诺塔问题的移动步骤完成，并打印出每次移动步骤。

如果问题规模还没有小到能直接解决，则按照假设的规则，第一步从起始柱子移动对应数量盘子到中转柱子，即递归调用getSteps(moveDisk, pegs, sourceId, destId, bufferId)。

然后因为此时中转柱子已经被盘子占用，我们需要先选取另一个空的中转柱子，从目标柱子以外的所有空柱子中选取即可，这里我们取第一个空闲柱子。然后执行假设中的第二步，将起始柱子上剩余的盘子移动到目标柱子上，同样也是递归调用getSteps(disk - moveDisk, pegs - 1, sourceId, newBufferId, destId)。

最后执行假设的第三步，即把中转柱子上的盘子移动到目标柱子上，也是递归调用getSteps(moveDisk, pegs, bufferId, sourceId, destId)。

至此，所有移动步骤就完成了。最后将最少移动次数、所有步骤的信息都存至HanoiResult对象当中，返回给前端。

#### 2.2.5 结果的表示

结果表示涉及到两个类HanoiResult和HanoiStep，HanoiStep用于记录每一步的移动和所有柱子的状态，HanoiResult保存求解结果和所有的步骤，即HanoiStep对象数组。求解完成后，结果和步骤会被封装进HanoiResult对象中并返回给前端页面。

下面是对这两个类的成员变量的描述：

|  |  |  |
| --- | --- | --- |
| HanoiResult | | |
| 成员 | 类型 | 作用 |
| numTower | Int | 柱子数量 |
| numDisk | Int | 盘子数量 |
| leastMove | Int | 最少移动次数 |
| totalStep | Int | 总步骤数，即打印的步数 |
| errorCount | Int | 总错误数，即错误的移动，当把编号更大的盘子移动到编号小的盘子上时出现，用于测试算法正确性 |
| msg | String | 向前端传递的消息，主要包含用户输入不合法等错误信息 |
| stepList | List<HanoiStep> | 保存所有的步骤信息 |

|  |  |  |
| --- | --- | --- |
| HanoiStep | | |
| 成员 | 类型 | 作用 |
| currentStep | Int | 当前步数 |
| Source | Int | 移动起点柱子id |
| destination | Int | 移动目标柱子id |
| pegStackList | List<Stack<Integer>> | 存储了所有柱子上的盘子 |