2021山东省夏令营7.20晚讲座

动态规划怎样真正入门

莱州一中 孙铭远

2021.07.20

了解动态规划

- 动态规划几乎是OI赛事的必考内容
  - csp2020普及 T4
  - csp2020提高 T4
  - csp2019提高 D2T1（计数）D2T2（单调队列优化）
  - csp2019入门 T3（ 背包）
  - NOIP2018提高D1T2（背包）D1T3（树形）D2T2（状压）D2T3（树形）
  - NOIP2018普及T3（斜率优化）
  - NOIP2017提高D1T3（dp+最短路）D2T2（状压）
  - NOIP2017普及T4（二分+单调队列优化）
  - NOIP2016提高 D1T3（期望）
- 动态规划是常见的高分暴力（60-80pts）

了解动态规划

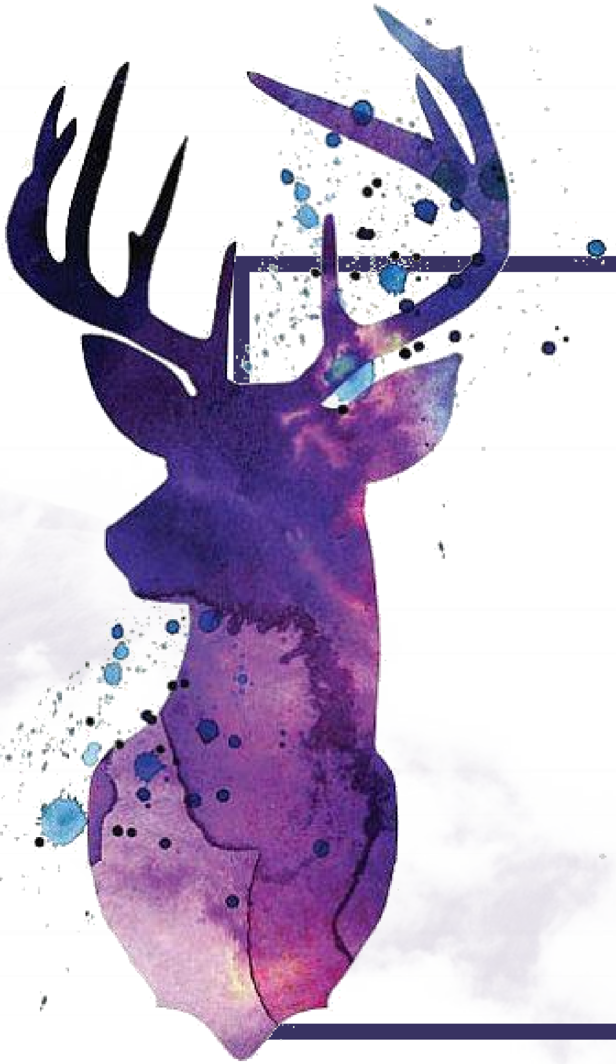
- 为什么动态规划有如此重要的位置?
  - 思维模式
  - 非常灵活
  - 容易复合

了解动态规划

- 从我校学弟学妹的理解状况来看，很多同学应该都处于晕晕乎乎的状态：
  - 什么？这个题是动态规划？
  - 我知道是动态规划，但是这个题的思路好奇特，状态是怎么列出来的？
  - 我跟正解列了一样的状态但是我没有优化。
  - 好的现在这道题我听懂了，但是再来一道题我还是不会做。
- 动态规划为什么这么难？
  - 难在很多同学对动态规划没有真正的理解，只是做一道算一道。
- 学习动态规划必须去反思、总结、梳理和归纳
- 这堂课分享我个人对于动态规划多年的归纳和理解。

了解动态规划

- 要完整的解决一道动态规划，有以下五步
  - 设立状态
  - 设计转移
  - 赋初值
  - 复杂度优化
  - 代码实现
- 这次报告，我会从最基础的定义开始，给大家细致的讲解动态规划。



目录

- 问题模型
- 状态与转移
- 重叠子问题
- 转移成环
- 最优子结构
- 无后效性
- 正确性检验
- dp的时空复杂度
- 刷表法与填表法

问题模型

- 通常情况下，典型的动态规划问题都可以转化成如下模型：

  在满足某种条件的限制之下，求解某问题的最优解（或方案数）。

- 举例1：
  - 题意：[洛谷P1439]给定一个长度为n的序列a，求他的最长上升子序列。
  - 条件：子序列、上升。
  - 求解：LIS最长。
- 举例2：
  - 题意：[石子归并]给定n堆石子，每次可以把相邻的两堆合并成一堆，并获得新的一堆的石子的个数的得分，求将所有的石子合并成一堆能获得的最大得分。
  - 条件：每次合并相邻、最终合并成一堆。
  - 求解：最大得分。

问题模型

- 举例3
- 题意：背包问题
- 条件：总重量上限、每个物品选择限制（01、完全、多重、分组）
- 求解：最大价值

状态与转移

- 接下来，我们以最长上升子序列[LIS]为例，探讨一下状态与转移究竟是什么。
- 朴素的枚举做法?
  - 递归枚举每个数是否在子序列中，只有比当前子序列结尾大的数才可以加入；每搜索到一个上升子序列就记录一下长度，输出最长的子序列。
- 动态规划的思想?
  - 将朴素的枚举做法进行优化，把枚举的每一种具体的方案按照一定的规则分类，将它们一起求解而不是逐一枚举。
- 我们接下来看一下本题的具体做法。

状态与转移

- 设f[i]表示所有以i位置的数字为结尾上升子序列中，长度最长的上升子序列的长度。
- 我们从1~n依次扫过序列a，扫到第i位时，我们求f[i]的值：枚举前面的每一个位置j，如果a[i] > a[j]，说明i可以放到以j结尾的上升子序列的后面构成一个新的上升子序列； 如果a[i] <= a[j]，则j无法更新f[i]。
- 于是可以有以下的代码：

```
for(int i = 1; i <= n; ++i) {
    for(int j = 1; j < i; ++j) {
        if(a[i] > a[j]) {
            f[i] = max(f[i], f[j] + 1);
        }
    }
}
```

状态与转移

- f数组是什么?
  - 动态规划是将具体的方案按照一定的规则分类以优化算法的过程。
  - f数组的每一个单元：一个状态（一类方案）
  - 数组的下标：共同特点（状态信息）
  - 数组中存的值：所有方案的最优解（或方案数）
- 本题中，我们是如何设立f数组的?
  - 把方案按照结尾位置归为一类。
- 什么是状态?
  - 状态是将一类方案的共同特点提取出，用这些共同特点来表示这一类方案；也即，一个状态表示一个方案集合。
  - 对于一个状态，我们在数组里存的是该方案集合内所有方案的某个信息，通常是最优解或者方案个数。

状态与转移

- 动态规划解题的关键点在于设置合适的状态，有了状态，转移往往会比较自然的给出。
- 那么如何设立状态呢?
  - 一个角度是依据需求，往往有如下几个方面：
    - 记录与题目的限制条件相关的内容
    - 记录与所求答案相关的内容
    - 为了正确的、方便的转移，存储某些内容
  - 另一个角度是根据问题规模（重叠子问题特征），我们会在后面讨论这部分内容。
- 状态的设立还有如下规则：
  - 所设的状态要尽可能的简化。
  - 所设状态的初值要已知；所有状态的值求出后要能够得到答案。
  - 状态之间的转移有明确的顺序。（转移无环）
  - 转移不能受状态之外的信息所影响。（无后效性）

状态与转移

- 现在我们已经设立完状态了，那么如何求得答案呢?
  - 已知f[0]=0，所求为max{f[i]}。根据f[0]我们可以得到f[1]，根据f[0]和f[1]我们可以得到f[2]……最终我们可以得到所有的值。
- 什么是转移?
  - 通过转移来实现状态之间的转化（由已知求得未知），最终实现由初值（边界条件，如f[0]=0）得到答案。
- 状态转移方程:
  - 用数学化、公式化的形式表示转移的方式。优点是简洁直观，方便推导优化。
  - 举例:

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

$$f[l][r]=\max_{k=l}^{r-1} f[l][k]+f[k+1][r]+w[k]$$

$$f[i][V]=\max(f[i-1][V],f[i-1][V-v[i]]+w[i])$$

阶段?

- 我个人认为阶段在动态规划中并不是一个重要的点。
  - 阶段的作用是明确动态规划进行的顺序（转移的顺序），会在状态中占据一维，导致和状态混淆。
  - 阶段的本质是重叠子问题每个子问题，但是具有高度的套路性，导致在解决动态规划问题时，不需要刻意的思考阶段。
    - 序列上的动规：从1~n扫序列a（线性dp），按照区间从小到大（区间dp）
    - 二维网格的动规：扫描每一个格子/bfs序/dfs序。
    - 树上动规：按照子树从小到大（从儿子到父亲）/从大到小（从父亲到儿子）。
    - DAG上的动规：拓扑序。
    - 数位dp：按照数位从高到低
- 这次授课，我把阶段的存在意义拆成了两点，一是从问题规模的角度建立状态，二是dp要有明确的转移顺序。

小结

- 问题模型
  - 在满足某种条件的限制之下，求解某问题的最优解（或方案数）。
- 状态
  - 状态的本质是一类在某些方面拥有共同特点的方案的集合。
  - f数组内存的是这类方案的最优解或方案个数。
  - 设立状态的角度有：条件限制、问题规模。
  - 状态的设立要：尽可能的简化、有初值能推出答案、转移有序无环、无后效性。
- 转移
  - 转移是状态之间相互转化的方式。
  - 状态转移方程： 用数学化、公式化的形式表示转移的方式。

重叠子问题

- 题目大意[洛谷P2758] 设A和B是两个仅含小写字母的字符串，长度分别为n，m。我们要用最少的操作次数，将字符串A转换为字符串B。n，m<=1000，这里所说的操作共有三种：
  - 1、删除任意一个一个字符；
  - 2、在任意位置插入一个字符；
  - 3、将任意一个字符改为另一个字符。

重叠子问题

- 大致思路：我们考虑最终B[1]处字符的来源：A[1]、新插入的、把某个字符修改得到、删掉一些字符后后面的字符补位得到的。我们分别讨论一下这四种情况：
  - 若是A[1]，那么继续做A[2~n]转换为B[2~m]即可。
  - 若新插入的，那么剩下的问题就变成了把A[1~n]转换为B[2~m]，可以继续进行。
  - 若为把某个字符修改得到的，一定会修改A[1]（为什么？）。故问题变成把A[2~n]转换为B[2~m]。
  - 若为删掉一些字符由后面的补位，那么可以看作先删除A[1]然后问题转化为将A[2~n]转化为B[1~m]。

重叠子问题

- 具体做法：设f[i][j]表示A已经考虑完第i位，B考虑完第j位时（即A[1~i]转化成了B[1~j]）的最小操作次数。
- 考虑如何转移：
  - 若A[i]=B[j]：f[i][j]=min(f[i][j], f[i-1][j-1])。
  - 以及还有共有的转移：f[i][j]=min{f[i][j], f[i-1][j-1]+1,f[i-1][j]+1,f[i][j-1]+1}。
- 代码如下：

```
memset(f, 0x3f, sizeof(f));
f[0][0] = 0;
for(int i = 1; i <= n; ++i) {
    for(int j = 1; j <= m; ++j) {
        if(a[i] == b[j]) f[i][j] = min(f[i][j], f[i - 1][j - 1]);
        f[i][j] = min(f[i][j], f[i - 1][j] + 1);
        f[i][j] = min(f[i][j], f[i][j - 1] + 1);
        f[i][j] = min(f[i][j], f[i - 1][j - 1] + 1);
    }
}
cout << f[n][m] << endl;
```

重叠子问题

- 解题思路?
  - 本题并没有什么特别明显的限制，从限制入手并不是很好设状态。
  - 我们在解题过程中所做的总结起来就是：讨论第一步做了什么，然后分析问题转化成了什么。
  - 问题转化成了一个相同且规模缩小的问题。
  - 所以我们按照问题规模设立状态，设f[i][j]表示……
- 有关重叠子问题：
  - 重叠子问题是我们设状态的一个切入点。这类题目有明显的重复性，我们可以进行一步操作而使问题缩小，从而实现逐步求解。
  - 举例：区间dp，树上dp。

转移成环

- [luogu4042]
  - 你在进行一个打怪兽的游戏，共有n种怪兽，你有两种攻击方式。
  - 对于一个i号怪兽，你可以花费ai的代价把它打死，或者花费bi的代价把它变成ci个di号怪兽。
  - 起初只有一个1号怪兽，求最少花费多少的代价把它打死。
  - n<=2×10^5,Σci<=10^6
- 设f[i]表示杀掉一个i怪物所消耗的最小代价，那么有如下的转移方程

$$f[i]=\min(a[i],b[i]+c[i]\times f[d[i]])$$

- 由于怪物之间的转化没有固定的顺序（如只能由小编号转化成大编号等），所以我们没有办法确定一个枚举状态求解的顺序。

转移成环

- 动态规划问题可以抽象到图论上理解：
  - 将每个状态看作是一个点，若A状态能转移到B状态，那么连一条A向B的权值是这次转移对答案的贡献的有向边。
  - 一般的动态规划都会获得一个有向无环图，dp初值是起点，所求是终点，求解dp的过程转化成了求最短路。
  - 转移无环的dp建成的图是DAG图，所以求最短路只需按照拓扑序扫一遍状态即可。事实上我们写dp就是这样做的，只是没有显式的把图建出来。
  - dp的难点：设立状态和转移，相当于建图。
- 但是对于一些dp，没有阶段，建成的图是有向有环图，这时我们就要用spfa来实现了。
- 实现起来只需要像最短路那样：每个状态一旦被更新就加到队列中，每次取队首尝试更新它能更新的所有状态。（本质上是迭代）

转移成环

- 本题解法
  - 令f[i]初值为a[i]，然后将所有的f[i]加入队列中跑spfa，最后f[1]就是答案。
  - dijkstra不行吗?
  - 有的题可以，有的题根据题意整点trick修改一下做法可以，有的题不行。
- 对于设完状态之后发现转移成环的情况，一般有如下解决方法
  - 用spfa来跑dp（本题方法）
  - 思考是否有些转移是不必要的，一定不优。也即，有些转移一定不会对最终的答案产生贡献，可以忽略掉。
  - 思考方向有问题，重设状态。

最优子结构

- [洛谷P4342简化版]
  - 给定一个长度为n的运算序列a（如3+5*4），但是运算顺序不是先乘后加，而是从左至右。（如3+5*4=32）。
  - 你可以给这个序列加括号改变运算顺序（如3+(5*4)=23），你可以任意的加括号（即任意钦定运算符的运算顺序），求最后的运算结果最大是多少。
  - n<=200，a[i]可以是负数。
- （原版是环上问题，由于断环为链的技巧不在我们今天的讨论范围之内，故修改）。
- 为了方便表示，我们统一让下标i指的是第i个数字。

最优子结构

- 设f[l][r]表示将l~r这一段先加括号算出来的最大值。
- 那么可以有如下的状态转移方程：

$$f[l][r]=\max_{k=l}^{r-1}\left(f[l][k]\oplus f[k+1][r],f[l][k]\otimes f[k+1][r]\right)$$

- 注意在这里⊗表示的是如果k和k+1之间的符号是乘才有此项，⊕ 同理。
- 这个做法是正确的吗？
  - 我们忽略了一种情况：负负得正。
  - 我们要求的是最大值，但是最大值未必由最大值更新而来。
- 最优解不能推出最优解的情况叫做不满足最优子结构。

最优子结构

- 我们现在尝试修复一下这个做法。
  - 既然最大值还有可能由两个最小值相乘得到，那么我们在维护最大值的同时再开个数组维护最小值就可以了。
  - 设f[l][r]表示将l~r这一段先加括号算出来的最大值，g[l][r]表示最小值。
  - 那么可以有如下的状态转移方程：

$$f[l][r]=\max_{k=l}^{r-1}\left(f[l][k]\oplus f[k+1][r],f[l][k]\otimes f[k+1][r],g[l][k]\otimes g[k+1][r]\right)$$

$$g[l][r]=\min_{k=l}^{r-1}\left(g[l][k]\oplus g[k+1][r],f[l][k]\otimes g[k+1][r],g[l][k]\otimes f[k+1][r],g[l][k]\otimes g[k+1][r]\right)$$

  - 对于复杂的转移情况，通常需要手动枚举一下各种情况。

最优子结构

- 明明求的是最大值，为什么要维护最小值g呢?
  - 已知短段的最大值求不出长段的最大值。
  - 对于这种已知最优解求不出最优解的情况，我们称为不满足最优子结构。
- 不满足最优子结构的问题的解决方法?
  - 考虑最优解有可能是什么转移过来的，即我们少考虑了哪些情况，并开设数组同时维护（如例题）。
  - 思考方向有问题，重设状态。

无后效性

- [CF626F 简化版]
- 给定一个长度为n的序列a，你需要把它划分成若干个子序列（可以不连续，可以长度为1，但原顺序不能变），每个子序列的价值为结尾元素减起始元素，使所有子序列的价值和最大。
- n<=200
- 这个题有一个显然的性质，每个子序列一定是上升的。
    - 如果把问题转化成把序列划分成若干个上升子序列，那么会使题目的难度加大（因为限制更紧）。
    - 而正确的想法是尝试通过这个优美性质去简化题目，但是对于本题实际上不可行。
- 所以说，在这道题上，我们不考虑这个性质，大家想想怎么做？

无后效性

- 首先：考虑从左向右扫这个序列，每个位置可以新开一个子序列做开头、加入已有的子序列做中间值、加入已有的子序列做结尾。
- 观察发现：对于不同的子序列，中间值是什么我们是不需要管的，而对答案产生影响的是起始元素当时放的是什么。
- 遇到问题：把当时开头放了什么记入状态，显然是不现实的。而直接设f[i][j]表示当前在第i位，目前有j个子序列已开启未闭合，又会因为信息不全而无法统计答案。
- 所遇问题的本质：
  - 影响转移的信息无法全部记到状态里，即转移受到了状态之外的信息所影响，也即不满足无后效性。

无后效性

- 什么是无后效性?
  - 无后效性的名字含义为：当前不会对之后产生效应。
  - 前面提到过我们每个状态存的是一类方案，这一类方案作为一个状态A转移到状态B的转移系数、转移所选择的决策应当完全相同。
  - 转移不能受状态之外的信息所影响，即影响转移的所有要素都已计入状态。
- 本题解法
  - 观察发现，贡献的形式实际上是t-s的样子，我们可以拆成t和-s，即新开子序列的时候让答案-a[i]，关闭子序列的时候答案+a[i]。
  - 状态转移方程如下：（设f[i][j]表示当前在第i位，目前有j个子序列已开启未闭合）

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

  - 复杂度O(n^2)
  - 这满足无后效性的性质。

无后效性

- 有后效性怎么办?
  - 把影响转移的部分加到状态里，增多状态，细化方案的分类。
  - 费用提前计算。（本题方法）
  - 重设状态，改换思路。

小结

- 重叠子问题
  - 它是思考动态规划题目的重要角度，也是设立状态的常见角度。
- 转移成环
  - dp与图论有共通之处
  - dp本应要求有明确的转移顺序，但从图论的角度考虑时，可以处理一些转移成环的情况。
- 最优子结构
  - 最优子结构是我们在日常做题中最容易忽略的一点，但它决定了我们做法的正确性。
- 无后效性
  - 无后效性的本质是要求我们的状态划分的足够细致，使得可以正确的转移求解，但是太细了又会影响复杂度。
  - 有后效性是很棘手的情况，提前计算费用并不是都像例题那样简单。

正确性检验

- 动态规划的正确性只需要保证两点：
  - 有合适的初值，根据终值可以求出答案。（状态）
  - 根据初值可以推出正确的终值。（转移）
- 对于检验转移是否正确我们有一个类似于数学归纳法的思想：
  - 假设某一阶段以前的信息都已经正确得到，通过转移方程，我们可以得到下一阶段的正确信息，那么所设计的动态规划就是正确的。
  - 新手的常见误区就是考虑很多层的转移。随着dp的运行过程深入的模拟对人脑而言几乎是不可能的事情，最终根本无法证明。

时空复杂度的计算

- 时间复杂度=预处理复杂度+状态复杂度×转移复杂度。
- 空间复杂度<=状态复杂度（比如滚动数组）。
- 时空复杂度是检验dp优劣的重要标准，一切对于动态规划的优化都是针对复杂度的优化。

填表法

- 做法：先枚举目前要求信息的状态，再枚举哪些状态能转移到它。
- 代码实现：（以重叠子问题的例题为例）

```
memset(f, 0x3f, sizeof(f));
f[0][0] = 0;
for(int i = 1; i <= n; ++i) {
    for(int j = 1; j <= m; ++j) {
        if(a[i] == b[j]) f[i][j] = min(f[i][j], f[i - 1][j - 1]);
        f[i][j] = min(f[i][j], f[i - 1][j] + 1);
        f[i][j] = min(f[i][j], f[i][j - 1] + 1);
        f[i][j] = min(f[i][j], f[i - 1][j - 1] + 1);
    }
}
cout << f[n][m] << endl;
```

- 优点：简洁易读，多转一，方程清晰便于推导优化；适用范围更广泛（很微小）。

刷表法

- 做法：先枚举已知信息的状态，再枚举它能转移到的状态。
- 代码实现：（以例题2为例）

```
memset(f, 0x3f, sizeof(f));
f[0][0] = 0;
for(int i = 0; i <= n; ++i) {
    for(int j = 0; j <= m; ++j) {
        if(a[i + 1] == b[j + 1]) f[i + 1][j + 1] = min(f[i + 1][j + 1], f[i][j]);
        f[i + 1][j] = min(f[i + 1][j], f[i][j] + 1);
        f[i][j + 1] = min(f[i][j + 1], f[i][j] + 1);
        f[i + 1][j + 1] = min(f[i + 1][j + 1], f[i][j] + 1);
    }
}
cout << f[n][m] << endl;
```

- 缺点：比较混乱，不能直观看到转移到某个状态的所有状态，不方便针对转移进行优化；对于某些（极少数）动态规划不适用。

刷表法的用处

- 大家将来会用到的推式子优化、前缀和优化、斜率优化、单调性优化、数据结构维护等高级优化算法都是针对填表法，对于刷表法则难以或无法实现。
- 那么刷表法为什么还会存在于算法竞赛中呢？
- 刷表法有一种特殊的优化
  - 我们设立的状态有时不是所有都是有用的，有的状态是废状态（根本不存在满足特点的方案，比如求方案数值为0，求最小值值为INF），这样的状态我们完全可以不管它。
  - 填表法的时候，我们枚举一个未知信息的状态，然后枚举能转移到它的状态来转移。由于当前状态未知，所以我们不知道它是不是废状态，无论如何都要进行一遍转移，这样我们的动态规划的所有转移是跑满的。
  - 刷表法的时候，我们枚举一个已知信息的状态，一旦发现它是废状态，就可以跳过，不用它更新其它状态（因为没有用），这样实际进行的转移就会减少，代码速度更快。

汇 报 完 毕

感 谢 聆 听

Q&A

- 考虑到很多同学不好意思当众发问，这是我的联系方式，我们可以线上交流。
- QQ：1901030119
- 微信：15376962875

扫一扫二维码，加我QQ好友。