动 态 规 划

——从入门到经典

常州市第一中学　林厚从

摘要

- 动态规划的思想、概念
- 从搜索到动态规划——数字三角形
- 经典问题1——最长不下降子序列及其应用
- 经典问题2——最长公共子序列
- 举一反三——加工小木棍、WZK走路、书的复制、传纸条
- 经典问题3——01背包及其应用
- 经典问题4——区间DP及其应用
- 经典问题5——环形结构（数字游戏）
- 错误应用DP举例——MOD4取余、LKF吹泡泡
- 容易忽略DP问题——Floyd算法（最短路径）
- 总结与后续知识展望

历史

- 动态规划（DP，Dynamic Programming）最早由 Richard Bellman 在1940s 在讨论多阶段决策问题时提出。
- 1957年，他出版了名著《Dynamic Programming》，这是该领域的第一本著作。
- 动态规划首次进入信息学奥赛是IOI 1994（数字三角形），在国内首次出现是NOI 1995，是OI考试的重要知识点。
- 动态规划是运筹学的一个分支，是求解“决策过程最优值”问题的一种重要方法，在数学、计算机科学和经济学中应用广泛。

思想

- 和分治法一样，动态规划也是通过把一个复杂的问题分解为相对简单的子问题的方式求解，再组合出答案的方法。
- 不过，**分治法是将问题划分成一些独立的子问题**，递归地求解各子问题，然后再合并子问题的解而得到原问题的解。如二分查找、快速排序问题。
- 与此不同，**动态规划适用于子问题不是独立的情况**。也就是子问题**包含公共的子子问题**（如fibnacci数列问题求解第n项时，需要调用第n-1和第n-2项，而求第n-1项时又要调用第n-2项，此处公共子问题就是第n-2项）。在这种情况下，采用传统的递归分治法会做很多不必要的**冗余计算**，即重复地求解公共的子问题（如F[i]），所以效率比较低。

思想

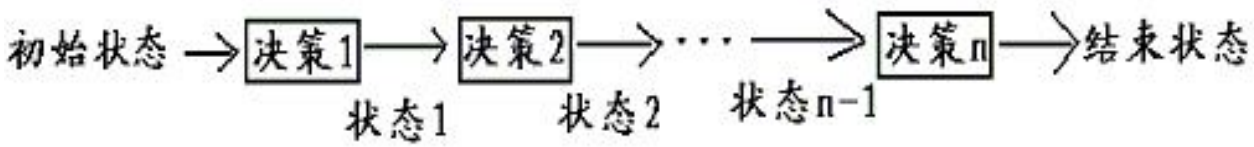
- 而动态规划算法对每个子子问题只求解一次，将其结果存在一张表中，从而避免后面每次遇到各个子子问题时重复去计算。
- 动态规划通常应用于最优化问题。此类问题可能有多个可行解，每个解有一个评价值，而我们希望找出一个最优解（评价值最优：最大或最小）。有时，还需要输出最优解的具体方案。

概念

- 状态：
  - 对问题的描述，如F[i]，F[i][j]等。
  - 当两个状态的出边完全相同时，他们可以被合并，我们称其为一类状态。此时某个状态可以完全取代另外一个状态。如果出现这样的情况，我们只需要找出该类中的最优状态，其他状态就可以被“抛弃”。
  - 在表示一类状态时，我们只需要记录最少的几个关键字，能够将其与其他状态完全区别开即可。
  - 所以，状态的维数要尽量少，保证状态总数尽量少。

概念

- 状态转移方程：
  - 描述状态之间的关系，类似于递归公式。
  - 一般形式为：F[状态A] = min/max {F[状态B] + w[B→A]}
  - 解释与理解：递归，或者假设已知所有F[B]，如何求F[A]？
  - 递归的边界：初始状态（初始值）。
  - 实现的方法：递推（顺推、倒推），目的是为了得到一个最优解。
  - 决策：根据转移方程做出一个合理选择，从一个最优状态转移到另一个最优状态。
  - 阶段性：动态规划处理的问题称为多阶段决策问题，一般根据时间、顺序、过程等划分阶段，决定了求值的顺序。



概念

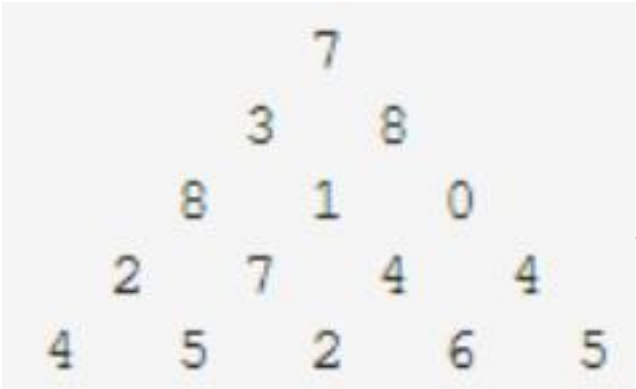
- 适用条件：
  - 最优化原理（最优子结构性质）：一个问题的最优解包含了子问题的最优解。
  - 无后效性：每个状态都是过去历史的一个完整总结。
- 其他：
  - 重叠子问题：DP的核心思想就是减少冗余计算。
  - 所求解的问题不能相互依赖（DAG，有向无环图）。

分类

- 动态规划算法十分繁多。
- 按照状态的维度，可分为线性DP，二维DP，多维DP…
- 按照状态的表示，可分为区间DP，树形DP，集合DP…
- 有时，还需要使用状态压缩、单调队列、四边形不等式原理等方法和工具优化DP。

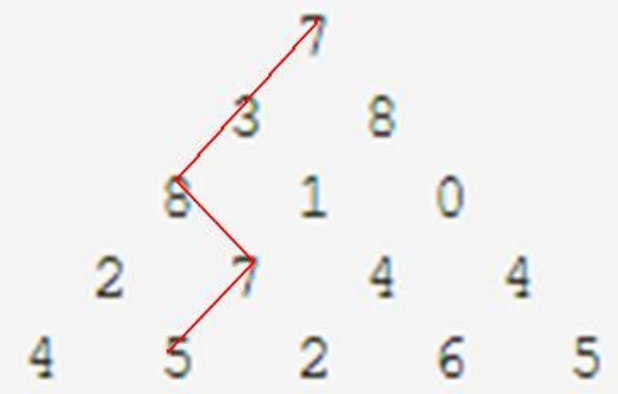
第一个简单又必须知道的例子——数字三角形（1O194）

- 问题描述：codevs1220
  - 下图给出了一个数字三角形。
  - 从三角形的顶部到底部有很多条不同的路径。对于每条路径，把路径上面的数加起来可以得到一个和，你的任务就是找到最大的和。
  - 注意：路径上的每一步只能从一个数走到下一层上和它最近的左边的那个数或者右边的那个数。
  - 每个数的范围[0..100]。
  - 行数不超过100。



第一个简单又必须知道的例子——数字三角形（1018）

- 样例输入：
  - 5
  - 7
  - 3 8
  - 8 1 0
  - 2 7 4 4
  - 4 5 2 6 5
- 样例输出：
  - 30



第一个简单又必须知道的例子——数字三角形（1O194）

- 一个常见的“深搜”式写法：dp1_1.cpp
- int a[101][101]，ans = –2147483647;
- void dfs(int x,int y,int sum){
- //表示从上往下，搜索到第x行、第y列得到的最大和sum
- sum += a[x][y];
- if(x == n){
- ans = max(ans,sum);//max函数实现打擂台
- return;
- }
- dfs(x+1,y,sum);//递归到正下方
- dfs(x+1,y+1,sum);//递归到右下
- }
- 主程序中调用dfs(1,1,0)，输出ans即可。

第一个简单又必须知道的例子——数字三角形（IOI94）

- 一个“描述问题”式的深搜写法：dp1_2.cpp
- int opt(int x,int y){
- //直接递归求解，由下至上直到a[1][1]
- if(x == n) return a[x][y];
- else return a[x][y] + max(opt(x+1,y),opt(x+1,y+1));
- }
- 答案为?
- opt(1,1)
- 离动态规划已经非常接近了！

第一个简单又必须知道的例子——数字三角形（1019A）

- 两个非常重要的概念：
  - opt(x,y)
  - 是问题的描述（状态）
  - opt(x,y)=a[x][y]+max(opt(x+1,y),opt(x+1,y+1))
  - 是问题之间的逻辑关系（状态转移方程）
- 但是，以上代码效率低下，为什么？
- 每个节点opt（x,y）会被调用2次，最多被调用2^n次。
- 冗余：重复计算同一个子问题。
- 简单修改一下，就可以得到下面一个高效的写法。

第一个简单又必须知道的例子——数字三角形（10194）

- 一个被称为“记忆化搜索”的版本：dp1_3.cpp
- int f[101][101];
- memset(f,–1,sizeof(f));
- int opt(int x,int y){
- if(f[x][y] != –1) return f[x][y];//记忆数组，以空间换时间
- else if(x == n) return a[x][y];//直接查表，不再冗余计算
- else {
- f[x][y] = a[x][y] + max(opt(x+1,y),opt(x+1,y+1));//记忆
- return f[x][y];
- }
- }
- 体现了动态规划非常重要的思想：减少冗余计算。
- 有观点认为：记忆化搜索 = 动态规划。

第一个简单又必须知道的例子——数字三角形（1O194）

- 更加“动态规划”式的写法：dp1_4.cpp
- int f[101][101];
- void doit(){
- for(int i = n; i >= 1; i--)
- for(int j = 1; j <= i; j++)
- if(i == n) f[i][j] = a[i][j];
- else f[i][j] = a[i][j] + max(f[i+1][j], f[i+1][j+1]);
- }
- 答案为?
- f[1][1]

第一个简单又必须知道的例子——数字三角形（10194）

- 如果需要打印具体的选择方案呢？dp1_5.cpp
- int f[101][101][2];//增加一维，记录最优值的来源
- void doit(){
- for(int i = n; i >= 1; i--)
- for(int j = 1; j <= i; j++)
- if(i == n) f[i][j][0] = a[i][j];
- else if(f[i+1][j][0] >= f[i+1][j+1][0]){
- f[i][j][0] = a[i][j] + f[i+1][j][0];
- f[i][j][1] = 0;//由正下方求得
- }else{
- f[i][j][0] = a[i][j] + f[i+1][j+1][0];
- f[i][j][1] = 1;//由右下方求得
- }
- }

第一个简单又必须知道的例子——数字三角形（10194）

- 本题小结：
  - 搜索与动态规划的相似点：状态、状态转移
  - 动态规划的优点：减少冗余计算
  - 动态规划的应用场合：求最优解及其具体方案（数）

- 以下学习的提示：
  - 应用动态规划的难点：状态的设计和转移
  - 本题：f[i][j]表示从第i行、第j列这个数到最后一行的路径上所有数之和的最大值。那么，状态转移方程为：f[i][j]=a[i][j]+max{f[i+1][j],f[i+1][j+1]}
  - 动态规划解题的思维：思考问题采用递归的思想，假设F[1]~F[i-1]已经知道了，如何求F[i]？编程实现采用递推的方法逐步推进。

第一个简单又必须知道的例子——数字三角形（1O194）

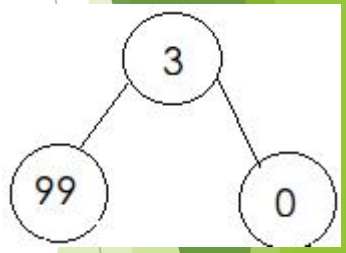
- 举一反三：codevs2193
  - 上面题目加上一个限制条件：要求路径必须经过第n/2行、第n/2列。

- 思考分析：
  - 还是一样的状态表示，假设f[i][j]表示从第i行、第j列这个数到最后一行的路径上所有数之和的最大值。那么，状态转移方程为：

    f[i][j]=a[i][j]+max{f[i+1][j],f[i+1][j+1]}
  - 只是在编程实现时，当计算到第n/2行时，把所有非f[n/2][n/2]的状态值都设置为负无穷大。

第一个简单又必须知道的例子——数字三角形（1O194）

- 举一反三：codevs2189
  - 还是上面的题目，但是要求路径上的所有数之和％100的最大值。

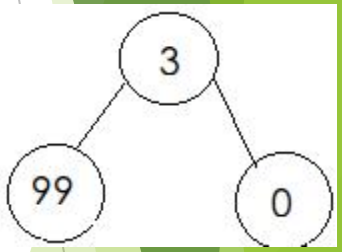


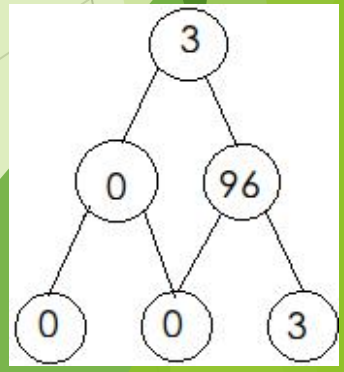
- 思考分析：
  - 如果我们还是沿用上面的方法，设计状态、进行状态转移：

    f[i][j] = (a[i][j] + max{f[i+1][j], f[i+1][j+1]}) % 100
  - 对不对呢？如果不对，你能举出反例吗？
  - 按照这个转移，会取(99+3) % 100 = 2，而实际答案显然应该是取(0+3) % 100 =3。那么如何修改？

第一个简单又必须知道的例子——数字三角形（1018）

- 思考分析：
  - 把% 100放到里面去，即：f[i][j] = max{(a[i][j]+f[i+1][j]) % 100, (a[i][j]+f[i+1][j+1]) % 100}
  - 这样对吗？
  - 再给一个例子：
  - 按照以上转移，就会取：0→0→3这条路径上的数，答案为3。而如果取：0→96→3这条路径上的数，答案为99。
  - 所以，还是不对。
  - 其实，这样定义状态根本就不存在“最优子结构”，也就说“当前（最终）的最优状态不一定由之前得到的最优状态转移而来”。

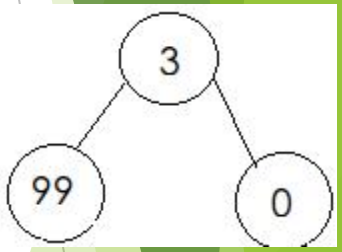


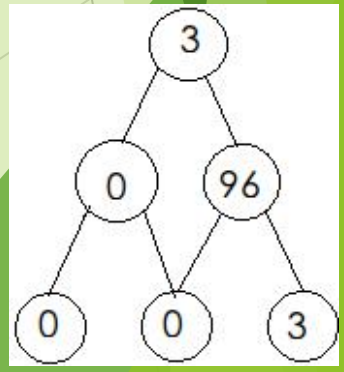


第一个简单又必须知道的例子——数字三角形（10194）

- 思考分析：
  - 那么，本题就不能使用动态规划了吗？其实，还是可以的，我们需要重新设计状态表示和转移方程。
  - 设f[i][j][k]表示有没有一条从以第i行、第j个这个数到最后一行的路径上数之和％100=k。f[i][j][k]为布尔型数组。
  - 转移方程如下：f[i][j][(k+a[i,j]) % 100] =

    f[i+1][j][k] || f[i+1][j+1][k]　　其中：i<n,j<=i,0<=k<100。
  - 表示：如果左边或右边有一条％100 = k的到最后一行的线路，那么就可以把它转移到这个点的决策。
  - 边界条件是f[n][i][a[n,i] % 100]为true，1<=i<=n。
  - 最后，统计并输出最大的满足f[i][j][k]为true的k。





经典问题1
——最长不下降子序列（LIS）

- 题目描述：
  - 给一个数组$a_1, a_2 \ldots a_n$，找到最长的不下降子序列$a_{b1}<=a_{b2}<= \ldots <=a_{bk}$，其中$b_1<b_2<\ldots<b_k$。
  - 输出最长的不下降子序列长度。
- 样例输入：
  - 5
  - 9 3 6 2 7
- 样例输出：
  - 3
- 样例说明：3 6 7

经典问题1
——最长不下降子序列（LIS）

- 状态的设计：
  - f[i]：表示前i个元素的最长不下降子序列长度？
  - 无法描述问题之间的关系！
  - 正确的状态表示：前i个元素的最长不下降子序列长度，且第i个元素已经被选。
- 状态的转移：
  - f[i]=max(f[j])+1
  - 其中，要满足：a[j]<=a[i]，1<=j<i
  - 边界（初始化）：f[i]=1，1<=i<=n
- 意思就是在a[i]之前找一个比a[i]小的a[j]，将a[i]放在以a[j]结尾的LIS序列之后，得到一个新的LIS序列。

经典问题1
——最长不下降子序列（LIS）

- 参考代码：时间复杂度:O（n^2），参考程序：lis1.cpp
- for(int i = 1; i <= n; i++){ //跟踪样例体会一维DP的求解过程
- f[i] = 1; //可以把f[i]放在外面初始化为1，然后i从2到n求解
- for(int j = 1; j <= i–1; j++)
- if(a[j] <= a[i]) f[i] = max(f[i],f[j]+1);
- }

- 答案是不是f[n]，为什么?
- int ans = f[1];
- for(int i = 2; i <= n; i++)
- if(f[i] > ans) ans = f[i];
- printf("%d\n",ans);

经典问题1
——最长不下降子序列（LIS）

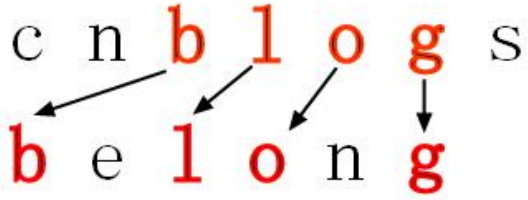
- 如果要输出方案呢?
- ```
  for(int i = 1; i <= n; i++){
        f[i] = 1;s[i] = 0;
        for(int j = 1; j <= i–1; j++)
           if(a[j] <= a[i])
                if(f[j]+1 > f[i]) {
                  f[i] = f[j] + 1;
                  s[i] = j; //记录当前的值是由哪个前驱结点求得
                }
  }
  ```
- 参考程序：lis2.cpp，打擂台输出长度后，再根据这个位置，倒过来找到这个序列是有前面那些元素组成，把这些元素压栈，最后倒序输出栈中的这些数即可。
  ```

经典问题1
——最长不下降子序列（LIS）

- 应用举例
  - 拦截导弹（mis）：第1问、第2问分开讨论。
  - 低价购买（buylow）
  - 航线（ship）：既要输出最优解，还要输出达到最优解的不同方案数。
  - ……

经典问题2
——最长公共子序列（LCS）

- 题目描述：
  - 给定两个序列X、Y，长度不超过5000。
  - 请求出两个序列的最长公共子序列。
  - 子序列不是字串（不要求连续）。
  - 两个字符串cnblogs和belong的公共子序列为blog。
  - 可以发现最长公共子序列是不唯一的，但是长度一定是唯一的，比如这里的最长公共子序列的长度为4。
- 样例输入：
  - cnblogs
  - belong
- 样例输出：
  - 4



经典问题2
——最长公共子序列（LCS）

- 状态设计：

  F[i][j]表示X序列前i个元素和Y序列前j个元素的LCS。

- 状态转移方程：

$$F[i][j]=\begin{cases}0 & \text{若 } i=0 \text{或} j=0\\ F[i-1][j-1]+1 & \text{若 } i,j>0, x_i=y_j\\ \max\{F[i][j-1],F[i-1][j]\} & \text{若 } i,j>0, x_i\neq y_j\end{cases}$$

- 参考程序：lcs1.cpp
  - int len1 = 1, len2 = 1;
  - while((s1[len1] = getchar()) != '\n') len1++;
  - len1--;
  - while((s2[len2] = getchar()) != '\n') len2++;
  - len2--;

经典问题2
——最长公共子序列（LCS）

$$F[i][j]=\begin{cases}0 & \text{若 } i=0 \text{或} j=0\\ F[i-1][j-1]+1 & \text{若 } i,j>0, x_i=y_j\\ \max\{F[i][j-1],F[i-1][j]\} & \text{若 } i,j>0, x_i\neq y_j\end{cases}$$

- 参考程序：lcs1.cpp。
  - for(int i = 0; i <= len1; i++) c[i][0] = 0;
  - for(int j = 0; j <= len2; j++) c[0][j] = 0;
  - for(int i = 1; i <= len1; i++)
  - for(int j = 1; j <= len2; j++)
  - if(s1[i] == s2[j]) c[i][j] = c[i–1][j–1] + 1;
  - else c[i][j] = max(c[i–1][j],c[i][j–1]);
  - printf("%d\n",c[len1][len2]);
- 跟踪样例，体会二维DP的求解过程。

经典问题2
——最长公共子序列（LCS）

$$F[i][j]=\begin{cases}0 & \text{若 } i=0 \text{或} j=0\\ F[i-1][j-1]+1 & \text{若 } i,j>0, x_i=y_j\\ \max\{F[i][j-1],F[i-1][j]\} & \text{若 } i,j>0, x_i\neq y_j\end{cases}$$

- 以上算法时间复杂度为O(len1*len2)，已经比较优了。空间复杂度可不可以优化呢？
- 类似于杨辉三角形的做法
  - 滚动数组：2个一维数组分别存放当前行和上一行，迭代。
  - 一个数组：从后往前计算。

经典问题2
——最长公共子序列（LCS）

- 参考程序：lcs2.cpp

```
memset(f,0,sizeof(f));
memset(pf,0,sizeof(pf));
//f存放的是二维数组的当前行，相当于二维中的f[i,j]
//pf存放的是二维数组的上一行，相当于二维中的f[i-1,j]
for(int i = 1; i <= len1; i++){
   for(int j = 1; j <= len2; j++){//固定i的情况下，计算f[j]，相当于f[i,j]
      f[j] = max(f[j-1],pf[j]);
      //s1[i] != s2[j]的情况转移到f[j-1]和pf[j]
      //即s1[i-1]与s2[j]的LCS和s1[i]与s2[j-1]的LCS
      if((s1[i] == s2[j]) && (pf[j-1] + 1 > f[j])) f[j] = pf[j-1] + 1;
      //s1[i]==s2[j]的情况转移到s1[i]与s2[j]的LCS再加1
   }
   for(int i = 0; i <= 1000; i++) pf[i] = f[i];//滚动
}
printf("%d\n",f[len2]);
```

合唱队形（CHORUS,NOIP2004）

- 题目描述：
  - N位同学站成一排，音乐老师要请其中的(N-K)位同学出列，使得剩下的K位同学排成合唱队形。
  - 合唱队形是指这样的一种队形：设K位同学从左到右依次编号为1，2…K，他们的身高分别为T_1，T_2…T_K，则他们的身高满足$T_1 < \ldots < T_i > T_{i+1} > \cdots > T_K$ (1<=i<=K)。
  - 你的任务是，已知所有N位同学的身高，计算最少需要几位同学出列，可以使得剩下的同学排成合唱队形。
  - n <= 1000。
  - 题目缺陷：$T_1 < \ldots < T_n$ 或递减的也算合唱队形。

合唱队形（CHORUS,NOIP2004）

- 样例输入：
  - 8
  - 186 186 150 200 160 130 197 220
- 样例输出：
  - 4

- 讨论分析：
  - 最后的合唱队形一定是什么情况？
  - 一个递增序列(DP1)＋一个递减序列(DP2)！
  - 那么，中间那个最高的是哪一位同学呢？
  - 穷举中间这位同学i。

合唱队形（CHORUS,NOIP2004）

- 但是，如果先枚举中间最高的那个人，接着对它的左边求最长上升序列，对右边求最长下降序列，那么，总的时间复杂度为O(n^3)，n=1000还是比较危险的。
- 分析发现：我们只需要先求好最长上升DP1[1..n]和下降序列DP2[1..n]，然后再枚举中间最高的同学，每一种合唱队形的长度为DP1[i]+DP2[i]-1，打擂台取最大值ans,答案为n-ans。这样的时间复杂度为O(n^2)。
- 如果本题的n<=10^6呢？
- 进一步优化：利用单调队列优化到$O(n*\log_2 n)$，略。

上机题

- 数字三角形（5个程序，自测）
- 最长不下降子序列（LIS，自测）
- 最长公共子序列（LCS，自测）
- 合唱队形（chorus，noip2004）
- 低价购买（buylow）
- 拦截导弹（mis）
- 航线（ship）

谢谢大家！