



动态规划专题

湖南师大附中 许力

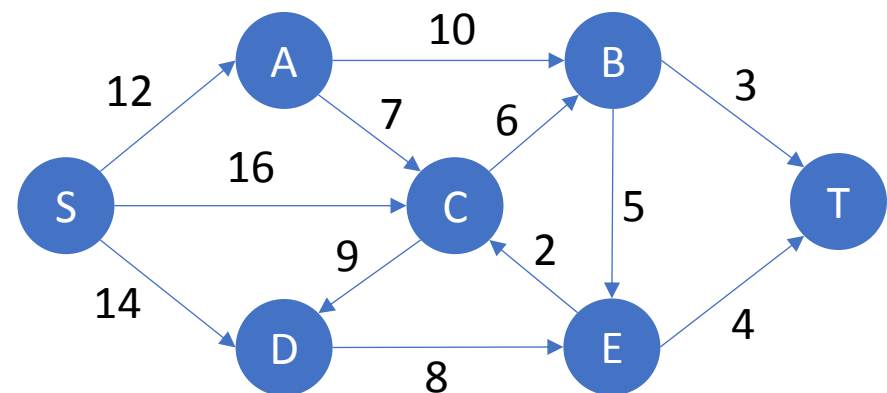


动态规划

- 动态规划思想的起源与核心：将问题 $f(n)$ 拆成几个子问题，分别求解这些子问题，即可得出 $f(n)$ 的解，这是解决问题的最优子结构特性
- 而子问题是如何求出来的，在求解 $f(n)$ 时不必关心，也不会影响 $f(n)$ 的求解。这就是无后效性

动态规划

- 求最短路
- 记 $f(T)$ 表示从 S 走到 T 的最短路
- $f(T) = \min(f(B)+3, f(E)+4)$
=
- $f(v) = \min(f(u)+w(u,v))$



怎么设计DP算法

- 首先把某一阶段的解状态表示为 x ，对于状态 x ，记我们要求出的答案为 $f(x)$
- 找出 $f(x)$ 与哪些状态有关（记为 y_1 、 y_2 、.....），写出一个式子（称为状态转移方程），通过 $f(y)$ 来推出 $f(x)$

尼克的任务

- 一个工作日从第 1 分钟开始到第 n 分钟结束。同一时刻有多个任务要完成，可以选择其中一个，而如果某个时刻只有一个任务，则必须选择。问如何决策才能得到最大闲暇时间

sample	
Input	Output
15 6	4
1 2	
1 6	
4 11	
8 5	
8 1	
11 5	

分析

- 要闲暇时间多，也就是工作时间尽量短
- 对于每一分钟，只有工作或者闲暇两种决策

分析

- 设 $f[i]$ 表示第 i 分钟开始直到最后能得到的最大闲暇，设 $t[j]$ 表示 i 时刻开始的任务所持续的时间

– $f[i] = f[i+1] + 1$ // 第 i 时刻无工作可做

– $f[i] = \max(f[i], f[i+t[j]])$ // 第 i 时刻有工作做

– 最后 $ans = f[1]$

分析

- 如果顺推，前面的决策会对后面造成影响，所以倒推比顺推易于理解
 - 顺推：设 $f[i]$ 表示第 1 分钟开始直到当前时间能得到的最大闲暇，
设 $t[j]$ 表示 i 时刻开始的任务所持续的时间
-
- $f[i+1] = f[i] + 1$ // 第 i 时刻无工作可做
 - $f[i+t[j]] = \max(f[i], f[i+t[j]])$ // 第 i 时刻有工作做
-
- 最后 $ans = f[n+1]$

小奇挖矿

- 现在有 $m+1$ 个星球，从左到右标号为0到 m ，小奇最初在0号星球。有 n 处矿体，第 i 处矿体有 a_i 单位原矿在第 b_i 个星球上。由于飞船使用的是老式的跳跃引擎，每次它只能从第 x 号星球移动到第 $x+4$ 号星球或 $x+7$ 号星球。每到一个星球，小奇会采走该星球上所有的原矿，求小奇能采到的最大原矿数量。注意，小奇不必最终到达 m 号星球

sample	
Input	Output
3 13 100 4 10 7 1 11	101

分析

- 设 $f[i]$ 表示到 i 号星球累计能采到的最大原矿数, $c[i]$ 为 i 号星球上的原矿数

$$- f[i] = \max(f[i-4], f[i-7]) + c[i]$$

线性动态规划的几类常见模型

- 最长不下降/不上升子序列（LIS）
- 最长公共子序列（LCS）
- 区间DP

最长不下降子序列

- 给定长度为 n 的序列 a_i ，求其中最长的不下降子序列

– 3 5 7 9 4 1 2

– 1 7 3 5 9 4 8

- 注意子序列和子串有区别，不一定连续

最长不下降子序列

- 典型模型
- 设 $f[i]$ 表示以 a_i 结尾的最长不下降子序列，则：
 - $f[1] = 1$
 - $f[i] = \max(f[j]+1, f[i])$ ($j < i$, 且 $a_j \leq a_i$)
- 时间复杂度 $O(n^2)$

参考代码

```
f[]初始化为 1;  
for (int i = 2; i <= n; ++ i)  
{  
    for (int j = 1; j <= i - 1; ++ j)  
        if (a[i] > a[j]) f[i] = max(f[j] + 1, f[i]);    //状态转移  
    if (f[i] > ans) ans = f[i];  
}
```

nlogn做法

- 有没有办法优化呢？当然是有的
- 思想来源于我们之前举的例子：1 7 3 5 9 4 8
- 有一个贪心的思想蕴含其中：如果当前已知的LIS结尾元素越小，后续元素就越有机会连接进来，当然LIS有更大可能会更长

nlogn做法

- 增设一个数组 $d[i]$ ，用于记录长度为 i 的最长上升子序列中最后那个数字
- 特别的，如果出现多个长度为 i 的最长上升子序列，则记录最小的那个数字

nlogn做法

- 只要当前考察的 $a[]$ 比 $d[]$ 数组中的最后一个数大，则一定能构成一个长度+1的合法序列，那么 $len++$
- 而一旦发现比较结果是小的，那我们就可以用 $a[]$ 替换掉对应位置上的 $d[]$ ，因为我们希望结尾的这个元素尽可能小，使得这个序列尽可能长

nlogn做法

• 例如原序列 $a[]$: 1 7 3 5 9 4 8, 则 $d[]$ 数组:

-1 0 0 0 0 0 0 len=1

-1 7 0 0 0 0 0 len=2

-1 3 0 0 0 0 0 len=2

-1 3 5 0 0 0 0 len=3

-1 3 5 9 0 0 0 len=4

-1 3 4 9 0 0 0 len=4

-1 3 4 8 0 0 0 len=4

nlogn做法

-1 3 5 9 0 0 0 len = 4

-1 3 4 9 0 0 0 len = 4

- 在这一步，要用4更新5，所以需要找到5的位置
- 因为 $d[]$ 数组中的元素严格维持**单调不降**的关系。而对于一个满足单调性的序列，二分查找就可以把搜一遍的复杂度从 $O(n)$ 降到 $O(\log n)$

nlogn做法

- 改进后的算法复杂度 $O(n\log n)$
- 这种利用决策单调性，优化转移复杂度的思想，在DP题中很常见

参考代码

```
len = 1; d[1] = a[1];
for (int i = 2; i <= n; i++)
    if (a[i] > d[len]) d[++len] = a[i];
    else {
        int pos = find(i); //二分查找
        d[pos] = a[i];
    }
```

lower_bound

- 也可以直接用STL里的 `lower_bound`，返回第一个比 `a[i]` 大的数的位置

最长公共子序列 (LCS)

- 给定 a_i 、 b_i 两个序列，求其最长公共子序列（可能不唯一）

– **ABACDB**

– **BADBCB**

- 暴力的复杂度惊人： $O(n \cdot 2^n)$

分析

- 典型模型:
- 设 $a[x]$ 表示 a 序列的前 x 项组成的子序列, $b[y]$ 表示 b 序列的前 y 项组成的子序列, $f[x][y]$ 表示 a 、 b 这两个序列的最长公共子序列:
 - $f[x][y] = f[x-1][y-1] + 1$ if ($a[x]==b[y]$)
 - $f[x][y] = \max(f[x-1][y], f[x][y-1])$ if ($a[x]!=b[y]$)

分析

- 样例数据对应的f[][]数组变化情况

```
ABACDB
BADBCB
0 1 1 1 1 1
1 1 1 2 2 2
1 2 2 2 2 2
1 2 2 2 3 3
1 2 3 3 3 3
1 2 3 4 4 4
4
```

参考代码

```
int lena = strlen(a), lenb = strlen(b);

for (int i = 1; i <= lena; ++ i)
    for (int j = 1; j <= lenb; ++ j)
        if (a[i-1] == b[j-1]) f[i][j] = f[i-1][j-1] + 1;
        else f[i][j] = max(f[i-1][j], f[i][j-1]);

printf("%d", f[lena][lenb]);
```

回文字串

- 任意给定一个字符串S，保持S的字符顺序不变，至少要加几个字符才能使S变成回文字串？
- 例如字符串S: abfcbfa，至少添加2个字符变成回文字串：
af**b**cbfcbfa

分析

- 首先，回文串就是正反读起来一样，那么我们可以把S串的反向也保存起来： **abfcbfa**，反向**afbcbfa**
- 那么很明显它们共同的部分，是可以正反读起来一样的，差别就在于哪些不同的部分

分析

- 求出原串和逆串的LCS: **abfba**
- 然后原串: abfcbfa和LCS: abfba的差值: cf即为答案, 如果还要求方案的话注意正反

石子归并

- 有 n 堆石子排成一行，每堆石子的重量为 $w[i]$ ，每次可以合并相邻的两堆石子，合并的代价为两堆石子的重量和： $w[i] + w[i+1]$
- 询问合并总代价最小和最大的方案

sample

Input

4

4 5 9 4

Output

43

54

分析

- 设 $f[i][j]$ 表示从第 i 堆石子合并到第 j 堆石子的最优方案
- 对于区间 $[i,j]$ ，假设中间某点为 k ，则 $[i,j]$ 必然是由两个子区间 $[i,k]$ 和 $[k+1,j]$ 合并而来，因此可以枚举中间点 k 。这个 k ，本质上是石子构成的区间之间的分割线

分析

$$f[i][j] = \min(f[i][j], f[i][k] + f[k+1][j] + (\text{sum}[j] - \text{sum}[i-1]))$$

- 最终答案就是： $f[1][n]$
- 这里的sum要做前缀和预处理

参考代码

```
for (i = n; i >= 1; -- i)    // 往回倒推
    for (j = i + 1; j <= n; ++ j)
        for (k = i; k < j; ++ k)
            f[i][j] = min(f[i][j], f[i][k] + f[k+1][j] + (sum[j] - sum[i-1]));

printf("%d", f[1][n]);
```

环形排列

- 如果石子是环形排列的话，就构成首尾相接的圆环
- 一般的做法是拆长度为 n 的环为长度为 $2*n$ 的链
- 不过枚举左右端点的时候，可能会出现右端点在左端点左边？

数字三角形

- 要求经过的数字和最大
- 从底层往上逆推

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

- **`sum[i][j] += max(a[i+1][j], a[i+1][j+1])`** // i 逆序


参考代码

```
for (i = n-1; i >= 1; -- i)
    for (j = 1; j <= i; ++ j)
        a[i][j] += max(a[i+1][j], a[i+1][j+1]);


printf("%d", a[1][1]);    //走到顶, 即为最终答案
```

方格取数

- 有 $n \times n$ ($n \leq 9$) 的棋盘，某些格子中存放了正整数，其他的格子则存放了0。某人从左上角走到右下角，他只能向右走或向下走，然后他可以取走所经过方格中的数（取走后方格中变为0）。问取到最大和的方案



0	0	0	0	0	0	0	0
0	0	13	0	0	6	0	0
0	0	0	0	7	0	0	0
0	0	0	14	0	0	0	0
0	21	0	0	0	4	0	0
0	0	15	0	0	0	0	0
0	14	0	0	0	0	0	0
0	0	0	0	0	0	0	0



分析

- 二维DP，和数字三角形类似，就不多说了

分析

- 如果是两个人同时走，就变成四维DP
- 设 $f[i][j]$ 表示第一个人走到 (i,j) 取得的最大和， $f[k][l]$ 表示第二个人走到 (k,l) 取得的最大和：

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

分析

- 能不能降维呢？
 - 如果我们记录当前走过的步数，因为两个人同时走，所以步数是相同的
 - 可以用 $f[s][i][j]$ 表示走了 s 步之后，此时第一个人走到了 $(i, s-i)$ 的位置，第二个人走到了 $(j, s-j)$ 的位置取得的最大和
-
- 记得逆序，这样可以节省空间开销

背包类DP

- 0-1背包
- 完全背包
- 多重背包
- 多维费用背包
- 分组背包

0-1背包DP

- 有 n 个重量为 $w[i]$ ，价值为 $c[i]$ 的物品，挑选一些放入一个最大承重为 W 的背包，使得背包中的物品价值总和最大

分析

- 和背包问题的最大区别是：**0-1**背包有可能放不满，故贪心可能取不到最优解

分析

- 典型模型：
- 设 $f[i][j]$ 表示取了前 i 个物品，消耗了 j 单位重量所得到的最大价值总和
 - 如果第 i 件物品不取： $f[i][j] = f[i-1][j]$
 - 如果第 i 件物品要取： $f[i][j] = f[i-1][j-w[i]] + c[i]$

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

分析

- 样例数据对应的f[]数组变化情况（正序输出）

```
5 10
2 6
2 3
6 5
5 4
4 6
6 6 6 6 6 6 6 6 6
6 6 9 9 9 9 9 9 9
9 9 11 11 14
4 9 9 11 11 14
6 6 9 9 11 11 15
15
-----
```

0-1背包DP空间优化

- 递推式

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

中的 $f[i][j]$ 一定要使用二维数组吗？

0-1背包DP空间优化

- $f[i-1]$ 本来就会直接更新为 $f[i]$ 的，所以完全可以得到：

$$f[j] = \max(f[j], f[j-w[i]]+c[i])$$

0-1背包DP细节讨论

- 两种不同的问法：恰好将背包装满，和未要求恰好将背包装满，有区别吗？

集合划分

- 现有一个由 $1 \dots n$ ($n \leq 500$) 所有自然数组成的集合 Q 。需要你将它划分为 A 、 B 两个子集，使得两个子集的元素之和相同。要求： $A \cup B = Q$ ； $A \cap B = \Phi$ 。求有多少种划分方法

分析

- 这个问题可以看成是0-1背包问题，对于分解成的一个集合，每个数要么选，要么不选
- 背包的最大容量为 $\text{sum}[i]/2$ ，而且sum必须为偶数（小优化）

完全背包DP

- 有 n 种重量为 $w[i]$ ，价值为 $c[i]$ 的物品，每种物品都有无限多件可供挑选，挑选一些放入一个最大承重为 W 的背包，使得背包中的物品价值总和最大

分析

- 实际上就是每种物品可以无限制地重复选

分析

- 依然还是可以用0-1背包的思路，即把可无限选的物品看成独立的

$$f[j] = \max(f[j], f[j-w[i]]+c[i])$$

- 不过因为可以无限制选，所以应该从空载状态开始顺推，才能把最优的物品重复选入

完全背包DP剪枝

- 完全背包问题有一个明显的剪枝：

if ($w[i] \leq w[j]$ && $c[i] \geq c[j]$) 则可直接跳过当前j物品

多重背包DP

- 有 n 种重量为 $w[i]$ ，价值为 $c[i]$ 的物品，每种物品都有 $t[i]$ 件可供挑选，挑选一些放入一个最大承重为 W 的背包，使得背包中的物品价值总和最大

分析

- 在完全背包DP的基础上做了限定：每种物品可选择放入的数量为 $0 \sim t[i]$ 件

分析

- 因此从 $0 \sim t[i]$ 中枚举放入了 k 个该种物品

$$f[j] = \max(f[j], f[j - k * w[i]] + k * c[i])$$

- 其中 k 的枚举范围应该是 $0 \sim k * w[i] \leq W$ ，因此也可以理解为把一件物品拆成 k 个，整体取出或放入

多重背包DP优化

- 借助于二进制的思想
- 令每件物品在被选取时附带一个系数，这个系数是 2^0 、 2^1 、 2^2 、 2^3 、.....、 2^{i-1} ， $k-2^i+1$ ，其中 $k-2^i+1>0$
- 比如某物品最终选取了7件，那我们枚举的顺序是1件、2件、4件；若是最终选取了13件，那么我们枚举的顺序是选取1件、2件、4件、6件，远比 $O(n)$ 的枚举效率要高

多维费用背包DP

- 有 n 个重量为 $w[i]$ ，体积为 $v[i]$ ，价值为 $c[i]$ 的物品，挑选一些放入一个最大承重为 W ，最大容积为 V 的背包，使得背包中的物品价值总和最大

分析

- 相比0-1背包，选取物品时需要考虑的因素不止一个

分析

- 设 $f[i][j][k]$ 表示取了前 i 个物品，消耗了 j 单位重量、 k 单位体积所得到的最大价值总和

直接空间优化后的转移方程：

$$f[j][k] = \max(f[j][k], f[j-w[i]][k-v[i]] + c[i])$$

- 其中 j 的循环范围是 $W \sim w[i]$ ， k 的循环范围是 $V \sim v[i]$

分组背包DP

- 有 n 个重量为 $w[i]$ ，价值为 $c[i]$ 的物品，分成若干组，每个物品有一个 $p[i]$ 值代表所属组号。同组物品只能选取一件，挑选一些放入一个最大承重为 W 的背包，使得背包中的物品价值总和最大

分析

- 决策变为：在一组物品中选一件，或者一件都不选

分析

- 怎样保证每一分组内的物品不重复出现？
 - 可以将这一分组内的物品当成状态来枚举，也就是每一分组内的物品用一个最内层**for**循环的变量来枚举，这样可以保证同一分组内的物品分配的背包已使用空间是相同的

金明的预算方案

- 要购买主件和附件两类物品，要买附件需先买主件，每个主件可以有0~2个附件。每件物品有价格 v ($v < 10000$)，重要度 w (1~5)。现在要求在总价格不超过 n 元的前提下，使每件物品的价格与重要度的乘积之和最大

sample	
Input	Output
1000 5	2200
800 2 0	
400 5 1	
300 5 1	
400 3 0	
500 2 0	

分析

- 带附件的分组背包DP
- 怎么分组？我们可以预处理一下，主件单独成组，附件与主件进行组合，形成物品组
- 这样就变成分组背包DP：本组内的物品要么都不选，要么选一件（主件）

分析

- 当然因为最多只会有2个附件，也可以变成5种情况的0-1背包来做
 1. 主件附件都不选
 2. 只选主件
 3. 选主件和第一个附件
 4. 选主件和第二个附件
 5. 选主件和两个附件

树

- 树是图的特例，也可以被叫做“无向无环连通图”
- 任意两点间有且仅有一条路径
- 一个节点只有一个父亲节点，但可以有多个儿子节点
- 树具有天然的层次性和递归性

树形DP

- 线性DP、区间DP的状态转移是在数轴上完成的
- 树形DP的状态转移是在树上完成的
 1. 根节点→叶节点：按DFS序方向转移
 2. 叶节点→根节点：到根节点相交时判断max/min

树形DP

- 比如这样一个问题：在一棵树上，要求选取最多的节点，使得选中的这些节点之间，都不存在直接的父子关系

分析

- 很明显的，一个节点只有选或者不选两种状态
- 而且，如果某个节点被选取，那么它的儿子必定不能入选；如果不选取，它的儿子可以选也可以不选
- 依此可以列出转移方程

没有上司的舞会

- 某大学有 n ($n \leq 6000$) 个职员，编号为 $1 \sim n$ 。他们之间有从属关系，也就是说他们的关系就像一棵以校长为根的树，父节点就是子节点的直接上司
- 现在有个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数 R_i ，但是呢，如果某个职员的上司来参加舞会了，那么这个职员就无论如何也不肯来参加舞会了。求邀请哪些职员可以使快乐指数最大，及最大的快乐指数是多少

分析

- 因公司的人际关系为一棵树，我们可以用链表维护树，把树保存在数组里
- 假设编号为`root`的节点是树的根，则问题可以描述成`f[root]`表示求以`root`为根的树上能得到的最大气氛和。我们希望用`root`的儿子们的对应信息来推出`f[root]`
- 就是从根节点向下搜索，在最底层将信息`f[]`更新上来

分析

- 由于节点root有选和不选两种可能，所以当选择root节点时，需要它的子节点不选时的最大值，当root节点不选时，它的子节点选和不选都可以
- 所以需要加一维来表示树的根节点是否选取才能从子树推算出父亲的最大值

分析

- 定义状态 $f[i][j]$ 表示以 i 为根的树能得到的最大的气氛和， $j=0$ 时表示根节点不选， $j=1$ 时表示根节点要选。状态转移方程为：

$f[i][0] += \max(f[\text{son}][1], f[\text{son}][0])$ //son为 i 的子节点

$f[i][1] += f[\text{son}][0] + g[i]$ //g[i]表示节点 i 的气氛值

- 答案是： $\max(f[\text{root}][0], f[\text{root}][1])$ //root为整棵树的根
- 就是最后在根节点处比较根节点选与不选的最大值

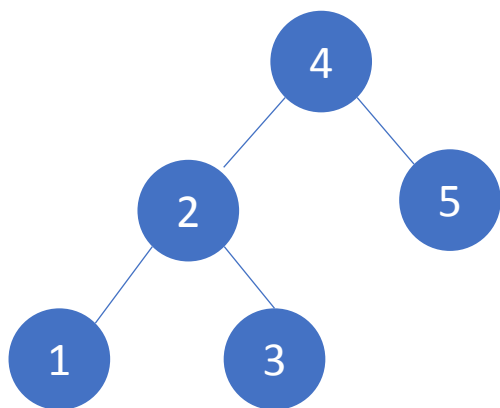
加分二叉树

- 设有 n 个节点的二叉树 $tree$ 的中序遍历为 $(1,2,3,\dots,n)$ ，其中数字 $1,2,3,\dots,n$ 为节点编号。每个节点都有一个分数（均为正整数），记第 i 个节点的分数为 d_i ， $tree$ 及它的每个子树都有一个加分，任一棵子树 $subtree$ （也包含 $tree$ 本身）的加分计算方法如下：
 $subtree$ 的左子树加分 $\times subtree$ 的右子树加分 $+ subtree$ 的根节点分数
- 若某个子树为空，规定其加分为1，叶子的加分就是叶节点本身的分数。不考虑它的空子树
- 试求一棵符合中序遍历为 $(1,2,3,\dots,n)$ 且加分最高的二叉树 $tree$ 。要求：
 1. $tree$ 的最高加分
 2. $tree$ 的前序遍历

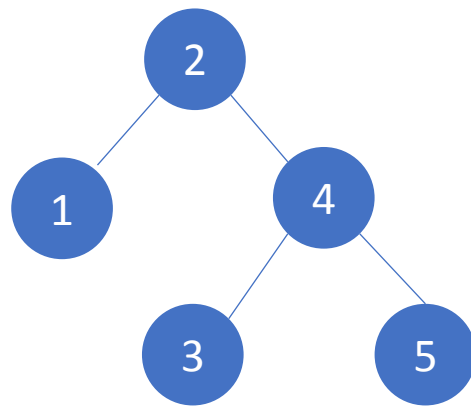
分析

- 样例分析

sample	
Input	Output
5	145
5 7 1 2 10	3 1 2 4 5

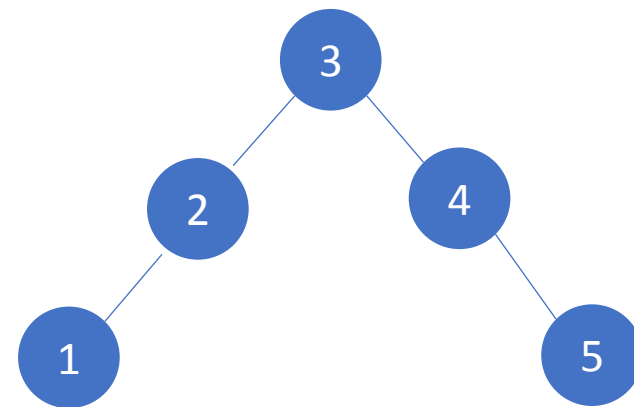


$$(5 \times 1 + 7) \times 10 + 2 = 132$$



$$(1 \times 10 + 2) \times 5 + 7 = 67$$

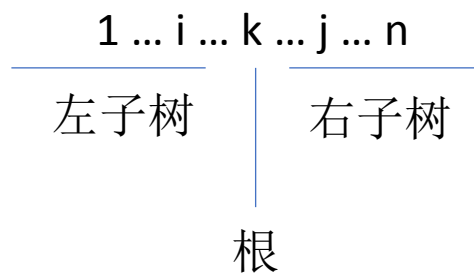
.....



$$(5 \times 1 + 7) \times (10 \times 1 + 2) + 1 = 145$$

分析

- 如果要整棵树的权值最大，必然有左子树的权值最大，右子树的权值也最大，符合最优子结构



分析

- 设 $f[i][j]$ 表示区间 $[i,j]$ 的最大权值

状态转移方程:

$$f[i][j] = \max(d[k] + f[i][k-1] * f[k+1][j]) \quad (i \leq k \leq j)$$

初始化时: $f[i][i] = d[i]$

答案: $f[1][n]$

分析

- 于是可以写出一个DP算法：
 - 按照状态 $f[i][j]$ 中 i 与 j 的距离来划分阶段，先处理掉边界情况(空树和只含有一个节点的树)
 - 然后就可以按照阶段自底向上进行动态规划了，最后再递归地输出。因为最终还要输出前序遍历，所以要保存每次的方案
- 时间复杂度 $O(n^3)$

重建道路

- 有 n ($n \leq 150$) 个节点的一棵树，求切断最少的边，恰好使一棵大小为 p 的子树被分离

sample	
Input	Output
11 6	2
1 2	
1 3	
1 4	
1 5	
2 6	
2 7	
2 8	
4 9	
4 10	
4 11	

分析

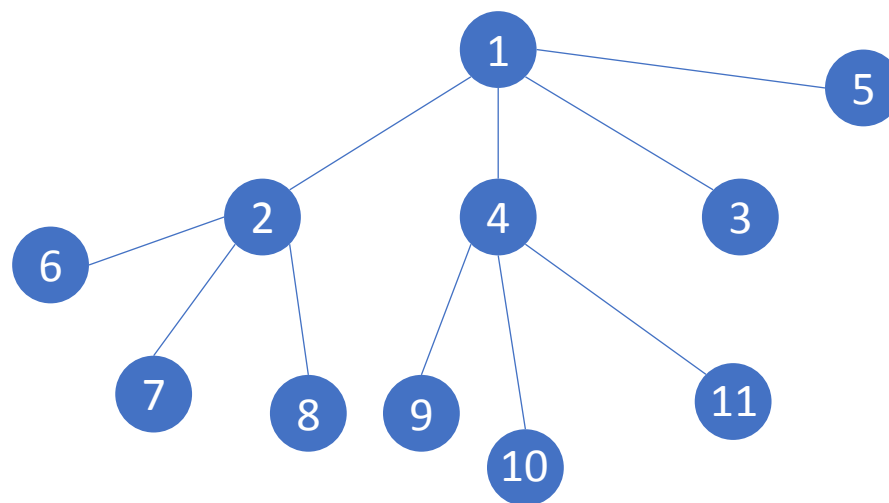
- 最小生成树？
- 没有合适的切入点

分析

- 题中要求每次分出的是一棵子树
- 那么可以根据子树的状态找到： 如果要分出来的是某个节点，那么最少要切掉和它相邻的 **tot** 条边，其中 **tot** 为当前点连出的边数，然后其它的节点答案可以由子树获得

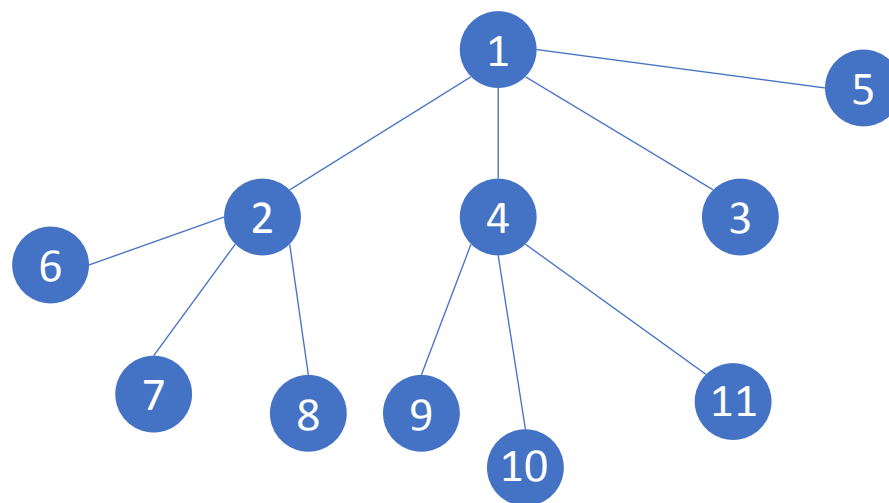
分析

- 于是我们每剪掉一个节点以及下面的子树的时候首先要确定减掉哪些边，例如样例：
- 我们怎么剪掉一棵大小为6的子树？



分析

- 于是我们发现：
 - 以当前点 u 以及下面的子树作为剪出来的节点剪出 i 个点，可以换成：下面的儿子 v 剪出 j 个点，加上当前点剪出剩余 $i-j$ 个点的答案减掉多剪的边两次



分析

- 可以列出转移方程:

$$\mathbf{dp[u][i] = \min(dp[u][i], dp[v][j] + dp[u][i-j] - 2)}$$

- 然后初始情况为 $dp[u][1]$ =出边，其余根据儿子节点更新答案
- 最后答案为: $\min\{dp[u][p]\}$

数位DP

- 经典的数位DP是要求统计符合限制的数字的个数，可以通过记录决定了前多少位以及大小关系来DP，或者DP处理不同位数的数的个数，然后DP统计
- 往往和数学中的统计、概率、方案数、进制等有关

Gray Code

- 给你一个不完整的 01 串($\text{len} \leq 200,000$), 让你填空缺的部分然后使它的格雷码的权值最大
- 权值定义为给定一个 $a[]$, 如果格雷码中的第 i 位为 1, 那么获得权值 $a[i]$

分析

- 格雷码的定义：在一组数的编码中，若任意两个相邻的代码只有一位二进制数不同，则称这种编码为格雷码
- $G[i]$ 只与 $B[i]$ 和 $B[i+1]$ 有关，如果确定数字则直接计算

分析

- 设 $dp[i][0/1]$ 表示第 i 位为 0/1 时第 i 位以及第 $i-1$ 位之前能获得的最大权值是多少:

$$dp[i][0/1] = \max(dp[i-1][0/1], dp[i-1][1/0] + a[i])$$

- 注意如果第 i 位是一个固定数 0/1, 那么第 i 位的 $dp[i][0/1]$ 不应该作为 $dp[i-1][0/1]$ 转移

其他技巧

- 有时候状态数过多，需要离散化、缩点等各种小技巧

过河

- 河上有一座独木桥，一只青蛙想沿着独木桥从河的一侧跳到另一侧。在桥上有 $M(1 \leq M \leq 100)$ 个石子，青蛙需要尽量少踩在这些石子上。青蛙从桥的起点开始，不停的向终点方向跳跃。一次跳跃的距离是 S 到 T 之间的任意正整数（包括 S, T ）。当青蛙跳到或跳过坐标为 L 的点时，就算青蛙已经跳出了独木桥
- 给出独木桥的长度 $L(L \leq 10^9)$ ，青蛙跳跃的距离范围 S, T ，桥上石子的位置，求最少需要踩到的石子数

sample	
Input	Output
10	2
2 3 5	
2 3 5 6 7	

分析

- 不能往回跳，符合无后效性
- 设 $f[i]$ 表示跳到数轴上的第 i 个点需要踩到的最少石子数，动态规划的状态转移方程如下：

$$f[0]=0$$

$$f[i]=\min(f[i-k], f[i-k]+1) \quad (S \leq k \leq T) \quad // \text{第} i \text{个点有没有石子}$$

最后的答案： $\text{ans}=\min(f[L], f[L+1], \dots, f[L+T-1])$

分析

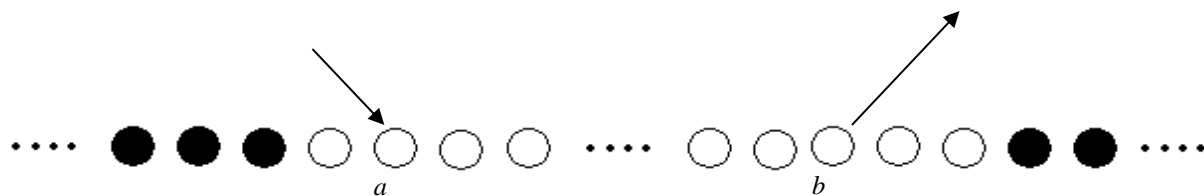
- 这个转移的时间复杂度是 $O(L \times (T-S))$ ，但本题的 L 高达 10^9 ，根本无法承受

分析

- L 可以达到 10^9 ，而 M 最大只有100，也就是说石子的分布是非常稀疏的
- 我们来考虑这样一个问题：长度为 k 的一段没有石子的独木桥，判断是否存在一种跳法从一端正好跳到另一端

分析

- 如下图所示，白色点表示连续的一长段的空地，黑色点表示石子
- 假设在原先的最优解中，白色段的第一个被跳到的位置a，最后一个被跳到的位置是b，则在做压缩处理之后，如果可以存在一种跳法从a正好跳到b，那么就可以把这段压缩掉，而且不经过任何石子



分析

- 假设[S,T]: [4,5]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
•				•	•			•	•	•		•	•	•	•	•	•	•	•	•

- 从数据中我们可以看到，12以后的点全都是可以到达的了
- 如果S、T的区间是[4,5]，在一段100000的距离中没有石子，可以做怎样的平移？

分析

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
•				•	•			•	•	•		•	•	•	•	•	•	•	•	•

- 首先， $T*(T-1)$ 之后的部分，一定是可以全部访问到的。也就是一定会存在满足条件的方案
- 所以，超过 $T*(T-1)$ 的长度也可以按 $T*(T-1)$ 处理
- 因为 $T \leq 10$ ，这样复杂度可以降低到 $10*(10-1)*(M+1)$

分析

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
•				•	•			•	•	•		•	•	•	•	•	•	•	•	•

- 其实还可以更进一步：我们观察到实际上12以后的点都可以被访问到了
- 考察每段连续访问的区间，可以得出： $S * \lceil (S-1)/(T-S) \rceil$ 后的部分就可以全部访问到了
- 或者也可以推测得出： $S * (S-1)$

分析

- 比如：区间是 $[3,5]$ ？
- 最“坏”的区间是？

分析

$S=T$ 时:

- 这时候由于每一步只能按固定步长跳，所以若第 i 个位置上有石子并且 $i \% S=0$ ，那么这个石子就一定要被踩到。这时我们只需要统计石子的位置中哪些是 S 的倍数即可

分析

$S < T$ 时:

- 首先我们作如下处理: 若存在某两个相邻石子之间的空白区域长度, 我们就将这段区域按照之前的推论缩短。压缩处理之后的最优值和原先的最优值相同

动态规划的缺陷

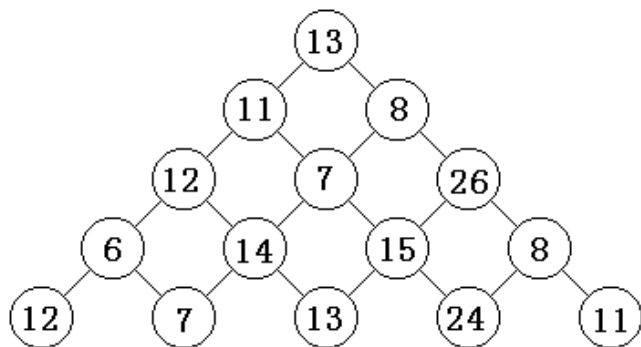
- 由之前的一些例子可以看到，动态规划将指数级别时间复杂度的搜索改进成了多项式时间的算法。其中的关键在于解决冗余，这是动态规划算法的根本目的
- 动态规划实质上是一种以空间换时间的方法，它在实现的过程中，不得不存储转移过程中的各种状态，所以它的空间消耗是偏大的

记忆化搜索

- 怎么解决这个问题呢？
- 考虑这样的空间开销是怎样产生的：产生了重叠子问题
- 观察数字三角形问题，可以发现随着层数增加，越来越多的节点被重复遍历（状态被重复存储），于是想到自顶向下遍历，将遍历过的状态保存起来

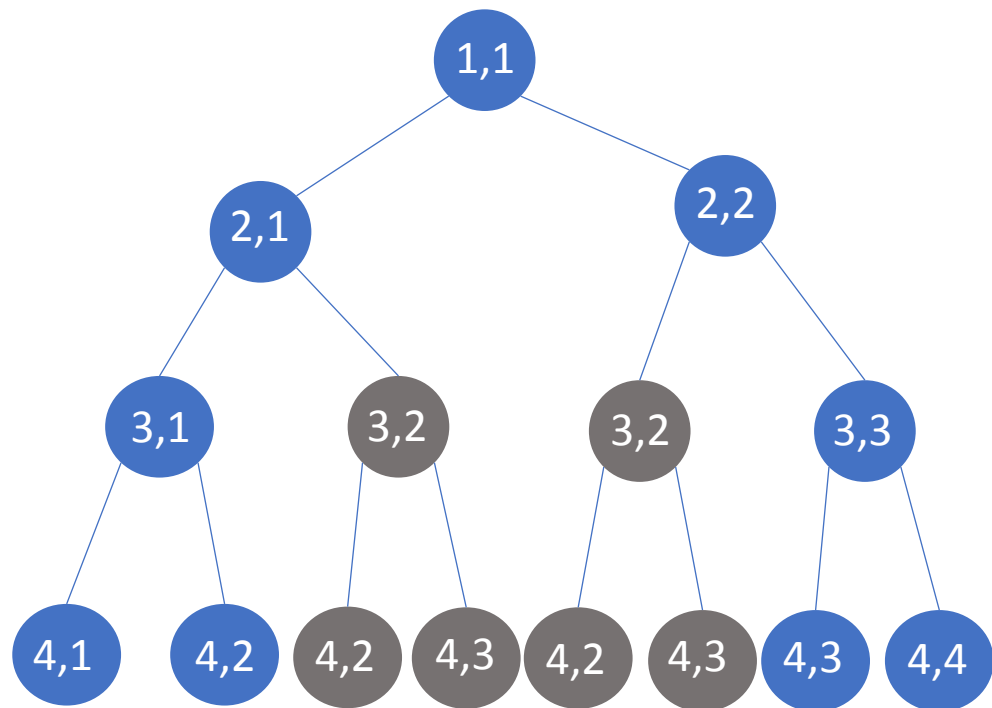
记忆化搜索

- 这种带备忘的搜索就是记忆化搜索
- 要储存的状态是 $n(n+1)/2$ 个，于是其空间复杂度为 $O(n^2)$

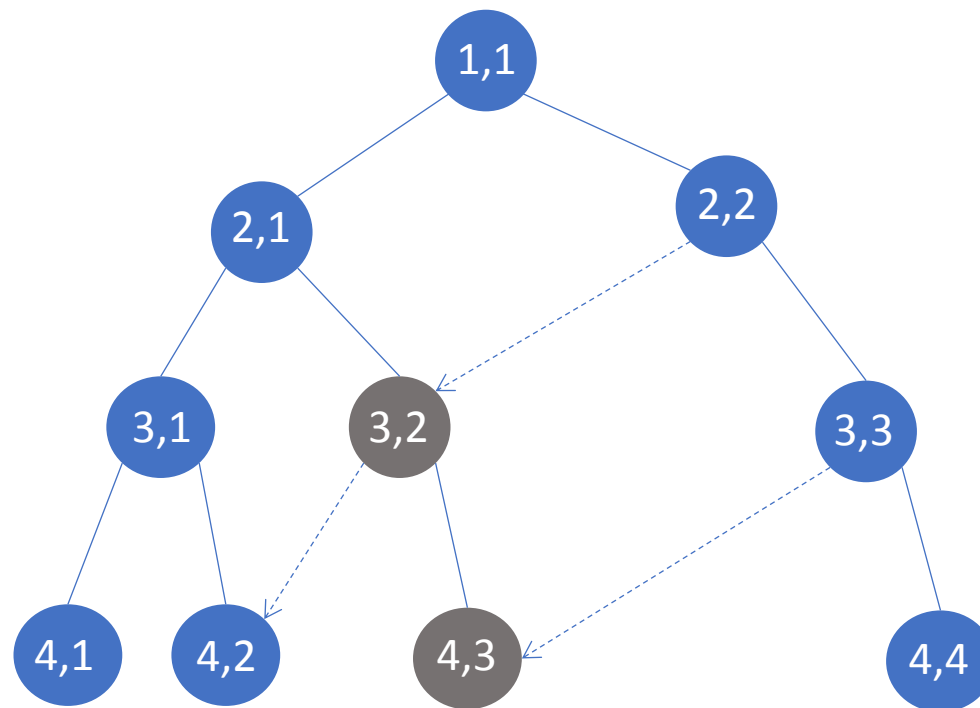


记忆化搜索

- 这是某被称为函数调用关系树的两棵树，可以清晰比较两种方法的空间消耗



重叠子问题



记忆化搜索

DP小结： 与其他算法的关系

1. 记忆化与递推

- 可以说所有用递推实现的动态规划均可以利用记忆化搜索实现
- 而某些题目之所以可以用递推求解，是因为这类题目同一阶段的状态表示上有很大的相似性

DP小结：与其他算法的关系

1. 记忆化与递推

- 利用递推实现动态规划时，单位操作时间小，可以方便地使用一些优化；而利用记忆化搜索实现动态规划时，单位操作时间大，但可以避免计算一些不必要的状态
- 总的来说，相比于用状态空间搜索的方法解决这类存在重复子问题的题目，使用动态规划增大了空间的开销，但提高了时间效率

DP小结：与其他算法的关系

2. 贪心

- 如果我们把每一步贪心作为一个阶段，那么可以认为每个阶段只有一个状态，再把贪心策略看作状态转移方程，那么这样是否也是一种“动态规划”呢？

DP小结： 与其他算法的关系

2. 贪心

- 它们二者之间的差别就在数据的维护上
- 有很多贪心问题在采取一次贪心策略之后要对数据进行全盘的维护，而动态规划一般没有这个步骤

DP小结： 与其他算法的关系

2. 贪心

- 而这个维护的目的是什么呢？
- 是为了保证正确性，和动态规划中的最优子结构的目的的一样，那么就说明目前的状态划分不具有最优子结构
- 是否能够通过改变状态的划分方法来代替这个维护并使新划分的状态具有最优子结构呢？

DP小结： 与其他算法的关系

2. 贪心

- 对于某些分步讨论的贪心决策问题，我们可以通过扩展状态使之能够用动态规划解决。
- 扩展出的状态的作用就是为了保证正确性，也就是保证最优子结构

DP小结：辅助

1. 排序

- 排序是一件利器，经常被使用在加入贪心思想的动态规划问题当中，有时也会出现在避免出现重复的问题当中
- 事实上就接近贪心了

DP小结：辅助

2. 预处理

- 预处理就是在 **DP** 之前对状态转移方程中的一些数据进行一些预先的计算，这样就方便了直接调用，并且同时还方便写优化
- 最典型的例子是石子归并中的 **sum** 数组，先将环状结构拆成链式结构，再利用前缀和快速求出某区间石子的总数

DP小结：优化

1. 改进状态表示/降维

- 状态的规模与状态表示的方法密切相关，通过改进状态表示减小状态总数是应用较为普遍的一种方法
- 最常见的方法就是降维，比如：0-1背包DP

DP小结：优化

1. 改进状态表示/降维

- 如果原方程是基于 n 维数组，且前 m 维每次只变化 1（取决于方程），后 $n-m$ 维的变化，由于循环的次序而不会被 $DP[i][j]$ 影响或覆盖，且最后求最大或最小值时只需要前 n 维的特定位置（一般是 $DP[i][j]$ 中 i =最大或 0），那么此时前 n 维就显得不是那么必要，那么我们就可以直接省去，节约空间

DP小结： 优化

2. 选择恰当的DP方向

- 这里说的方向是在动态规划方法实现中的方向，即正向或逆向
- 比如：完全背包DP

DP小结：优化

3. 滚动数组等小技巧

- 节省空间的优化做法
- 写DP经常需要开高维数组，空间开销很吓人，而由于无后效性，状态转移之后，没有必要把之前的全部结果都保存下来
- 最易理解的，比如斐波拉契数列： $f[i\%3]=f[(i-1)\%3]+f[(i-2)\%3]$
- 再比如某状态转移： $f[i\%2][j]=f[(i-1)\%2][j]+f[i\%2][j-1]$

推荐题单

- 跳房子
- 子矩阵
- 逛公园
- 宝藏
- 换教室
- 愤怒的小鸟
- 子串
- 飞扬的小鸟