

动态规划应用浅谈

李琛

Lic.tiny@gmail.com

- 上节课我们讲到了
- 状态的表示和状态的转移
- 是动态规划的重点和难点
- 今天，我们将就这两个方面进行更深入的探讨。

什么是动态规划

- 动态规划算法与分治法类似，其基本思想也是将待求解问题分解成**若干个子问题**
- 但是经分解得到的子问题往往不是互相独立的。不同子问题的数目常常只有多项式量级。在用分治法求解时，有些子问题被**重复计算**了许多次。
- 如果能够**保存已解决的子问题的答案**，而在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法。

由此不难得出结论：
动态规划实质就是

记忆化搜索

数塔问题的记忆化搜索写法

- `function dp(i,j:longint):longint;`
- `begin`
- `if (j=0) then exit(0);`
- `if (f[i,j]<>0) then exit(f[i,j]);`
- `f[i,j]:=max(dp(i-1,j-1),dp(i-1,j))+a[i,j];`
- `dp:=f[i,j];`
- `end;`

合并石子

- 问题描述:

有 N 堆石子排成一行,现要将石子有次序地合并成一堆.规定每次只能选相邻的2堆合并成新的一堆,并将新的一堆的石子数,记为该次合并的得分。

试设计出1个算法,计算出将 N 堆石子合并成1堆的最小得分和最大得分.

- 如果把归并1~4堆看成整个问题，则这个问题可以分解为三个归并方案，每个归并方案包含1~2个子问题：
 - ①先归并1~3；再与4归并
 - ②归并1~2，归并3~4；再归并
 - ③归并2~4；再与1归并
- (2) 如果我们继续分析增加更多堆数进行归并的问题，就会发现 n 个数归并时，会分解为 $n-1$ 种类型：
 - Case1: 归并第1堆，归并2~ n 堆；再最后归并
 - Case2: 归并1~2堆，归并3~ n 堆；再最后归并
 -
 - Case $n-1$: 归并1~ $n-1$ 堆，归并第 n 堆；再最后归并

- 归并的代价
- (1) 归并左右两堆的最小代价之和
- (2) 归并区间所有石子的权值之和
- 设 $F[i, j]$ 表示归并 $[I, J]$ 区间的最小代价
- 满足无后效性
- 满足最优子结构
- $F[i, j] = \min(F[i, k] + F[k+1, j]) + \text{sum}[i, j]$
- $(i \leq k < j)$
- $F[i, i] = \text{sum}[i, i]$
- 如何求出F数组?

一、枚举区间开头与结尾

- for i:=1 to n do
- for j:=i to n do
- F[i,j]:=10000000;
- for i:=1 to n do F[i,i]:=a[i];
- sum[0]:=0;
- for i:=1 to n do sum[i]:=sum[i-1]+a[i];
- for j:=2 to n do{枚举步长, 保证小区间先计算}
- for i:=1 to n-j+1 do
- for k:=i to i+j-2 do
- F[i,i+j-1]:=min(F[i,i+j-1],
- F[i,k]+F[k+1,i+j-1]+sum[i+j-1]-sum[i-1]);
- Writeln(F[1,n]);

二、记忆化搜索

- `function dp(i,j:longint):longint;`
- `var k,s:longint;`
- `begin`
- `s:=sum[j]-sum[i-1];`
- `if(i=j) then exit(s);`
- `if(f[i,j]<>0) then exit(f[i,j]);`
- `f[i,j]:=maxlongint;`
- `for k:=i to j-1 do`
- `f[i,j]:=min(f[i,j],dp(i,k)+dp(k+1,j)+s);`
- `dp:=f[i,j];`
- `end;`

动态规划的基本步骤

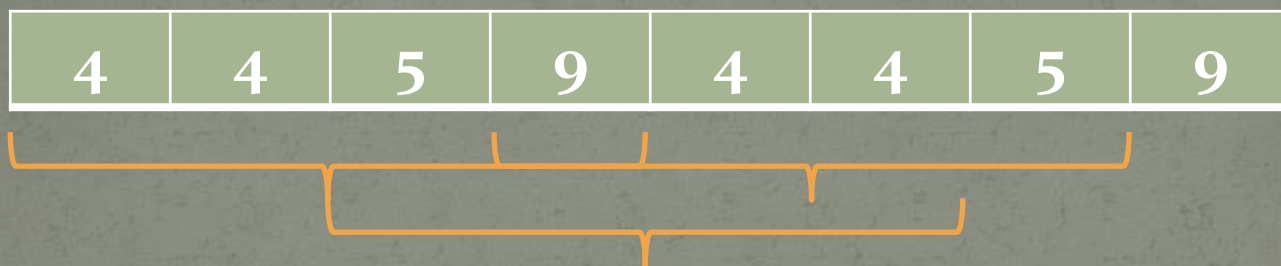
- 1) 描述最优解的结构
- 2) 递归定义最优解的值
- 3) 按自底向上的方式计算最优解的值
- ~~4) 由计算出的结果构造一个最优解~~

合并石子原题 (noi1995)

- 在一个圆形操场摆放 N 堆石子, 现要将石子有次序地合并成一堆. 规定每次只能选相邻的2堆合并成新的一堆, 并将新的一堆的石子数, 记为该次合并的得分。
试设计出1个算法, 计算出将 N 堆石子合并成1堆的最小得分和最大得分. $1 \leq N \leq 100$

- 链会做了，怎么做环？
- ①枚举断点
- 观察到必然有两堆相邻的石子只在最后一次合并。
- 于是我们枚举这个断点，将环断开成链。
- 枚举断点 $O(N)$ ，每次DP复杂度 $O(N^3)$ ，共 $O(N^4)$

- ②将链复制一遍



- 每一个长度为 N 的区间对应一种断点方案。
- 可以直接对 $[1, 2N]$ 这个区间整体进行动态规划。
- 每个长度为 N 的区间的答案都是合法的。
- 最优解在 $[1, N], [2, N+1] \dots [N+1, 2N]$ 中。
- 时间复杂度: $O(N^3)$

- 对于环上的动态规划常见方法：
- ①枚举断点
- ②将链复制成两倍

接苹果(apples)

- 农场的夏季是收获的好季节。在John的农场，他们用一种特别的方式来收苹果：Bessie摇苹果树，苹果落下，然后John尽力接到尽可能多的苹果。作为一个有经验的农夫，John将这个过程的坐标化。他清楚地知道什么时候($1 \leq t \leq 1,000,000$)什么位置(用二维坐标表示, $-1000 \leq x, y \leq 1000$)会有苹果落下。他只有提前到达那个位置，才能接到那个位置掉下的苹果。一个单位时间，John能走 s ($1 \leq s \leq 1000$)个单位。假设他开始时($t=0$)站在 $(0,0)$ 点，他最多能接到多少个苹果？

- 输入:
- 第一行是两个整数 N (苹果个数, $N \leq 5000$)和 S (速度);
- 第 $2 \dots N+1$ 行: 每行3个整数 X_i, Y_i, T_i , 表示每个苹果掉下的位置和落下的时间。
- 输出: 仅一行, 一个数, 表示最多能接到几个苹果。

[样例]

apples.in

5 3

0 0 1

0 3 2

-5 12 6

-1 0 3

-1 1 2

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1			2									
-1	3	2											
-2													
-3													
-4													
-5													6

apples.out

3

说明：John可以接到第1,5,4个苹果。

分析

- 首先划分阶段，很明显，按照苹果掉落的时间先后顺序来划分阶段，所以有必要按时间从小到大给各个苹果排个序，并按顺序标上1..n的编号。
- 假如John现在正站在某个位置上接苹果，为了使他到当前为止接到的苹果数最大，我们想要知道的是他前一步在哪个位置接苹果，并且要知道到那个位置为止接到的苹果最多是多少。
- 假设 $\text{dis}(i, j)$ 表示第 i 个苹果与第 j 个苹果之间的直线距离。 $\text{time}(i)$ 表示第 i 个苹果掉落的时刻。 $F(i)$ 表示John当前站在第 i 个苹果的位置上最多能接到的苹果总数（包括他以前接的）。

- $F(i) = \max \{ F(j) + 1 \}$
- 其中 $0 \leq j \leq i-1$, 且 $\text{dis}(i, j) \leq (\text{time}(i) - \text{time}(j)) * S$
- 初始条件: $F(0) = 0$
- 表示John站在出发点 $(0, 0)$ 时一个苹果也没接到。

Noip2010 乌龟棋

- 小明过生日的时候，爸爸送给他一副乌龟棋当作礼物。
- 乌龟棋的棋盘是一行 N 个格子，每个格子上一个分数（非负整数）。棋盘第1格是唯一的起点，第 N 格是终点，游戏要求玩家控制一个乌龟棋子从起点出发走到终点。
- 乌龟棋中 M 张爬行卡片，分成4种不同的类型（ M 张卡片中不一定包含所有4种类型的卡片，见样例），每种类型的卡片上分别标有1、2、3、4四个数字之一，表示使用这种卡片后，乌龟棋子将向前爬行相应的格子数。游戏中，玩家每次需要从所有的爬行卡片中选择一张之前没有使用过的爬行卡片，控制乌龟棋子前进相应的格子数，每张卡片只能使用一次。
- 游戏中，乌龟棋子自动获得起点格子的分数，并且在后续的爬行中每到达一个格子，就得到该格子相应的分数。玩家最终游戏得分就是乌龟棋子从起点到终点过程中到过的所有格子的分数总和。
- 很明显，用不同的爬行卡片使用顺序会使得最终游戏的得分不同，小明想要找到一种卡片使用顺序使得最终游戏得分最多。
- 现在，告诉你棋盘上每个格子的分数和所有的爬行卡片，你能告诉小明，他最多能得到多少分吗？

- 输入格式
- 输入文件的每行中两个数之间用一个空格隔开。
- 第1行2个正整数 N 和 M ，分别表示棋盘格子数和爬行卡片数。
- 第2行 N 个非负整数， $a_1a_2\ldots a_N$ ，其中 a_i 表示棋盘第 i 个格子上的分数。
- 第3行 M 个整数， $b_1b_2\ldots b_M$ ，表示 M 张爬行卡片上的数字。
- 输入数据保证到达终点时刚好用光 M 张爬行卡片。
- 输出格式
- 输出只有1行，1个整数，表示小明最多能得到的分数。

- 输入样例

- 9 5

- 6 10 14 2 8 8 18 5 17

- 1 3 1 2 1

- 输出样例

- 73

- 提示

- 小明使用爬行卡片顺序为1, 1, 3, 1, 2, 得到的分数为 $6+10+14+8+18+17=73$ 。注意, 由于起点是1, 所以自动获得第1格的分数6。

- 对于30%的数据有 $1 \leq N \leq 30$, $1 \leq M \leq 12$ 。

- 对于50%的数据有 $1 \leq N \leq 120$, $1 \leq M \leq 50$, 且4种爬行卡片, 每种卡片的张数不会超过20。

- 对于100%的数据有 $1 \leq N \leq 350$, $1 \leq M \leq 120$, 且4种爬行卡片, 每种卡片的张数不会超过40; $0 \leq a_i \leq 100$, $1 \leq i \leq N$; $1 \leq b_i \leq 4$, $1 \leq i \leq M$ 。

- 动态规划的特性比较明显：一个状态的确定只需要知道当前走了多少步以及四种卡片的使用情况。
- 很容易得出这样的状态表示：
- $F[n,a,b,c,d]$ 表示当前在 n 格，四种卡片分别用了 a,b,c,d 张时的最大得分。转移只需要枚举上一张用了什么，转移的复杂度 $O(1)$ 。
- 边界： $F[1,0,0,0]=a[1]$
- 最后答案为 $F[n,A,B,C,D]$ ，其中 A,B,C,D 为各个卡片的总张数。
- 时间复杂度 $O(A*B*C*D)=2560000$

- 问题解决了么?
- 空间复杂度 $O(N * A * B * C * D) \approx \text{九亿} \rightarrow 3\text{G}$
- 爆空间! (' □ ') ' ———
- 如何优化?

- 考虑当前位置和已用卡片的关系?
- 当前的位置可以根据已用的卡片唯一确定!
- $N = a + b * 2 + c * 3 + d * 4$
- 于是N这一维可以省掉了
- 方程: $F[a, b, c, d]$ 表示四种牌用了 a, b, c, d 张时所得的最大得分
- 边界: $F[0, 0, 0, 0] = a[1]$
- 这样空间复杂度也只有 $2560000 \approx 9M$ 了。

核心代码

- function solve(x,a,b,c,d:longint):longint;
- begin
- if f[a,b,c,d]<>-1 then exit(f[a,b,c,d]);
- if a>0 then f[a,b,c,d]:=max(f[a,b,c,d],solve(x-1,a-1,b,c,d)+s[x]);
- if b>0 then f[a,b,c,d]:=max(f[a,b,c,d],solve(x-2,a,b-1,c,d)+s[x]);
- if c>0 then f[a,b,c,d]:=max(f[a,b,c,d],solve(x-3,a,b,c-1,d)+s[x]);
- if d>0 then f[a,b,c,d]:=max(f[a,b,c,d],solve(x-4,a,b,c,d-1)+s[x]);
- exit(f[a,b,c,d]);
- end;
- 初始f[]全部为-1, f[0,0,0,0]=a[1]。

URAL 鹰蛋

- 有一堆共 M 个鹰蛋，一位教授想研究这些鹰蛋的坚硬度 E 。他是通过不断从一幢 N 层的楼上向下扔鹰蛋来确定 E 的。当鹰蛋从第 E 层楼及以下楼层落下时是不会碎的，但从第 $(E+1)$ 层楼及以上楼层向下落时会摔碎。如果鹰蛋未摔碎，还可以继续使用；但如果鹰蛋全碎了却仍未确定 E ，这显然是一个失败的实验。教授希望实验是成功的。

URAL 鹰蛋

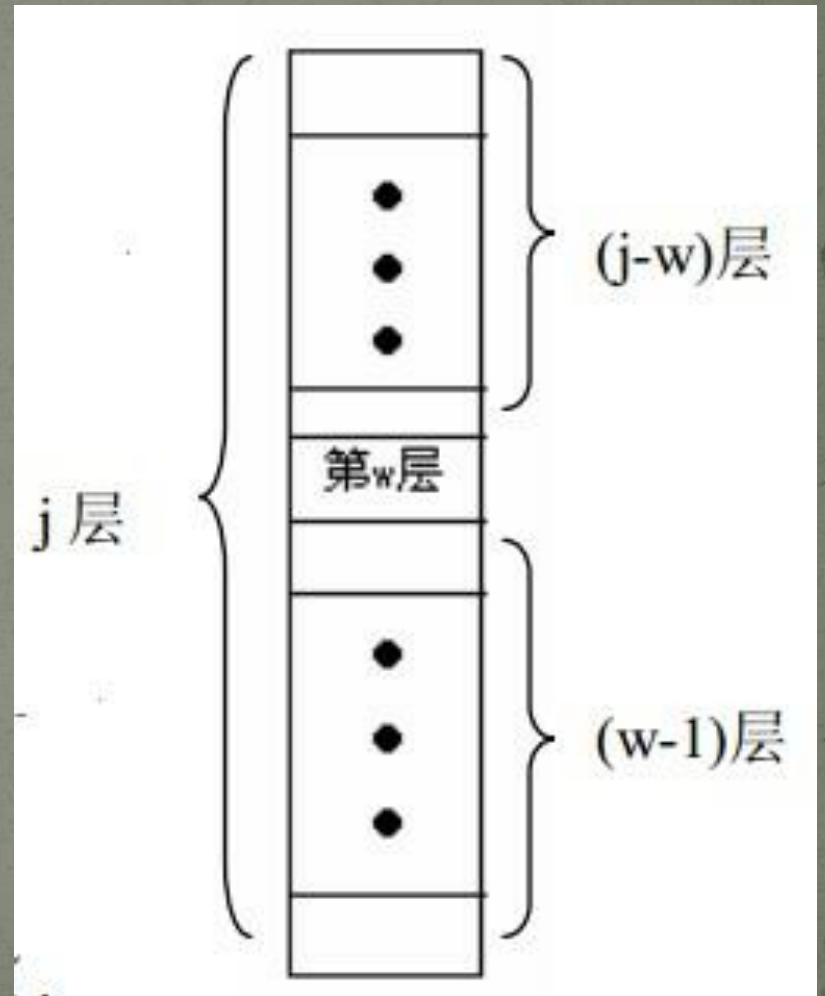
- 例如：若鹰蛋从第 1 层楼落下即摔碎， $E=0$ ；若鹰蛋从第 N 层楼落下仍未碎， $E=N$ 。
- 这里假设所有的鹰蛋都具有相同的坚硬度。给定鹰蛋个数 M 与楼层数 N 。
- 要求最坏情况下确定 E 所需要的最少次数。
- $M, N \leq 1000$

- 样例：
- $M=1$, $N=10$
- $ANS=10$
- 样例解释：为了不使实验失败，只能将这个鹰蛋按照从一楼到十楼的顺序依次扔下。一旦在第 $(E+1)$ 层楼摔碎， E 便确定了。（假设在第 $(N+1)$ 层摔鹰蛋会碎）

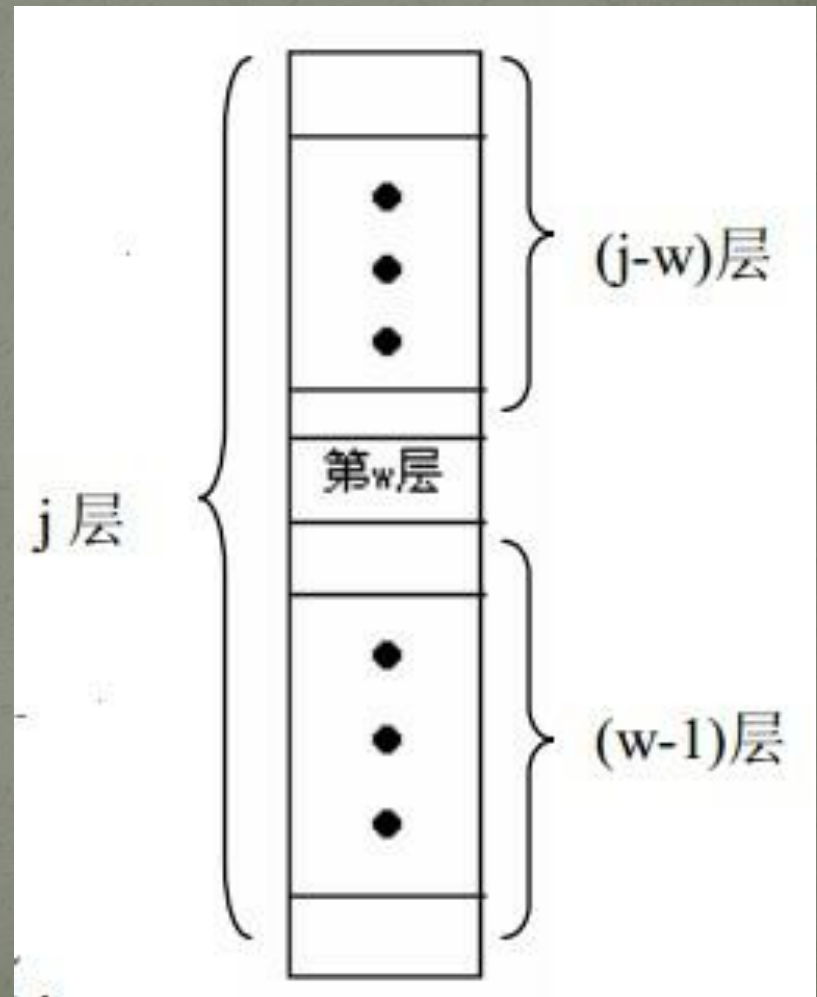
- 样例：
- $M=2, N=5$
- $ANS=3$
- 样例解释： 首先从3楼扔个蛋下去
- ①.碎了.E可能是1或2, 所以从1开始往上扔, 共2次
- ②.没碎。E可能是3或4或5.同①。

- 首先考虑简化版：如果蛋有无数个？
- 问题变成了在 $[1, n]$ 中猜数字。
- 每次告诉你大了还是小了。
- 这是就可以运用二分查找。
- 最坏情况需要 $\text{trunc}(\log_2(n)) + 1$ 次
- 如果蛋小于 $\text{trunc}(\log_2(n)) + 1$ 个呢？

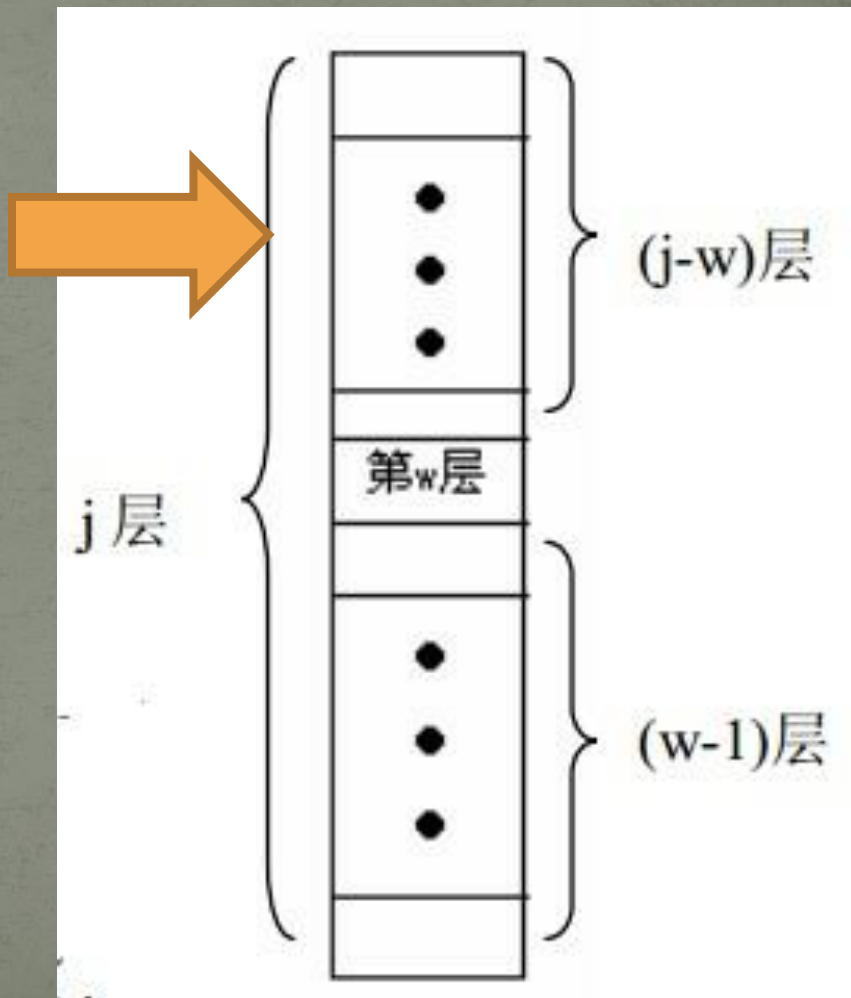
- 动态规划!
- 令 $F[i,j]$ 表示确定
- i 个蛋
- j 层楼
- 的最小扔蛋次数
- 如果现在从第 w 层扔了个蛋?



- ①蛋碎了
- 此时必有 $E < w$, 我们便只能用 $(i-1)$ 个蛋在下面的 $(w-1)$ 层确定 E , 并且要求最坏情况下次数最少。
- 这是一个子问题, 答案为 $f[i-1, w-1]$.
- 总次数便为 $f[i-1, w-1] + 1$;



- ②蛋没碎
- 此时必有 $E \geq w$.
- 用 i 个蛋在这 $(j-w)$ 层确定 E .
- 注意, 这里的实验与在第 $1 \sim (j-w)$ 层确定 E 所需次数是一样的, 因为它们的实验方法与步骤都是相同的, 只不过这 $(j-w)$ 层在上面罢了。完全可以把它看成是对第 $1 \sim (j-w)$ 层进行的操作。
- 答案为 $f[i, j-w]$, 总次数便为
- $f[i, j-w] + 1$ 。



- 扔这个蛋下去的时候，两种情况都要考虑到。
- 所以如果在w层扔蛋的最少次数是
- $\max\{f[i-1, w-1], f[i, j-w]\} + 1$.
- 由于可以在 $[1, j]$ 内任意w层扔蛋
- 所以要穷举w，并取最小值。
- 状态转移方程：
- $f[i, j] = \min\{\max\{f[i-1, w-1], f[i, j-w]\} + 1\}, 1 \leq w \leq j$
- 最后答案为 $f[m, n]$
- 时间复杂度为 $O(N^3)$ ，可能要超时？

- 猜数字的结论废了吗？
- 若 $m > \text{trunc}(\log_2(n)) + 1$ ，直接输出！
- 于是第一维的复杂度降到了 $\log_2 n$ 级别
- 时间复杂度 $O(N^2 \log_2 N) \approx$ 一千万。
- 时间复杂度可以接受，问题得到解决。

核心代码

- `if m>=trunc(ln(n)/ln(2))+1 then`
- `begin`
- `writeln(trunc(ln(n)/ln(2))+1);`
- `halt;`
- `end;`
- `f` 数组初始为无穷大
- `for i:=0 to m do f[i,0]:=0;`
- `for i:=1 to n do f[1,i]:=i;`
- `for i:=2 to m do`
- `for j:=1 to n do`
- `for w:=1 to j do`
- `f[i,j]:=min(f[i,j],max(f[i-1,w-1],f[i,j-w])+1);`
- `writeln(f[m,n]);`

- 这道题通过考察问题自身性质，减少了状态总数，从而使问题得到了解决。
- 实际上，鹰蛋这道题是一道经典题，可以有許多优化方式。其最低的复杂度可以达到 $O(\sqrt{N})$ ，有兴趣的同学可以课后与我讨论。

- 在动态规划的过程中，常常会遇到超时的问题。
- 一般有两种改进方向：
- ① 减少状态总数
- ② 减少转移时间
- 这两方面的优化是常常需要考虑的，可以通过做题来体会。

最小总代价

- n 个人在做传递物品的游戏,编号为 $1-n$ 。
- 游戏规则是这样的:开始时物品可以在任意一人手上,他可把物品传递给其他人中的任意一位;下一个人可以传递给未接过物品的任意一人。
- 即物品只能经过同一个人一次,而且每次传递过程都有一个代价;不同的人传给不同的人的代价值之间没有联系;
求当物品经过所有 n 个人后,整个过程的总代价是多少。

- 输入格式
- 第一行为n,表示共有n个人 ($16 \geq n \geq 2$) ;
以下为 $n \times n$ 的矩阵, 第i+1行、第j列表示物品从编号为i的人传递到编号为j的人所花费的代价, 特别的有第i+1行、第i列为-1 (因为物品不能自己传给自己), 其他数据均为正整数(≤ 10000)。
- 输出格式
- 一个数, 为最小的代价总和。

样例输入

2

-1 9794

2724 -1

样例输出

2724

- 搜索?
- 每次搜一个没有拿过球的人, 时间复杂度 $O(N!)$
- 在 $N \geq 12$ 时会超时
- 考虑使用动态规划
- 单纯使用 $F[i]$ 表示前 i 个人的最小代价有后效性。
- 如何设计状态?

- 考虑一个集合 $\{S\}$ ， S 中的元素代表了当前传过球的人。
- 设 $F[\{S\},i]$ 表示达到 $\{S\}$ 状态，当前球在 i 的最小代价。
- 可以发现它满足最优化原理和无后效性。
- 如何转移？

- 如何转移?
- 转移时我们可以枚举一个不在 S 里的元素 j , 将 j 添加到 S 中变为 S' , 得到 $F[S',j]=\max(F[S',j], F[S,i]+\text{map}[i,j])$
- 代表当前 i 将球传给了 j .
- S 是一个集合, 如何表示?

- 考虑到每个人只有传过和没传过两个状态。
- 而且人数 ≤ 16 。

- 考虑到每个人只有传过和没传过两个状态。
- 而且人数 ≤ 16 。

● 使用二进制！

- 用一个处于 $[0, 65535]$ 之间的二进制数mask表示S。
- 复杂度 $O(2^N * N^2) = 65536 * 16 * 16 \approx 1700$ 万，可以承受。
- 如何在mask中添加/查询人？

位运算

- 什么是位运算？
- 程序中的所有数在计算机内存中都是以二进制的形式储存的。位运算说穿了，就是直接对整数在内存中的二进制位进行操作。比如，and 运算本来是一个逻辑运算符，但整数与整数之间也可以进行 and 运算。举个例子，6 的二进制是 110，11 的二进制是 1011，那么 6 and 11 的结果就是 2，它是二进制对应位进行逻辑运算的结果（0 表示 False，1 表示 True，空位都当 0 处理）： $110 \text{ and } 1011 = 0010 \Rightarrow 2$

Pascal 中的位运算符

- 下面的 a 和 b 都是整数类型，则：

位运算	Pascal 语言
与 (and)	A and B
或 (or)	A or B
异或 (xor)	A xor B
取反 (not)	not a
左移 (shl)	A shl B
右移 (shr)	A shr B

位运算的使用

- 1. and 运算
- and 运算通常用于二进制取位操作，可以用来判断某一位是否是1.
- Mask and 2^k 如果非0，代表mask第k位为1
- 例如一个数 and 1 的结果就是取二进制的最末位。
- 这可以用来判断一个整数的奇偶，二进制的最末位为0 表示该数为偶数，最末位为1 表示该数为奇数.

位运算的使用

- 2. or 运算
- or 运算通常用于二进制特定位上的无条件赋值。
- $\text{Mask or } 2^k$ 表示将mask的第k位强行变为1。
- 例如一个数 or 1 的结果就是把二进制最末位强行变成1。

位运算的使用

- 3. XOR 运算
- XOR 运算通常用于对二进制的特定一位进行取反操作，因为异或可以这样定义：
- 0 和 1 异或 0 都不变，异或 1 则取反。
- 异或法则可以简单记为不进位加法。
- 异或有许多有趣的性质，在此不再赘述。

位运算的使用

- 4. not 运算
- not 运算的定义是把内存中的 0 和 1 全部取反。

位运算的使用

- 5. shl 运算
- $a \text{ shl } b$ 就表示把 a 转为二进制后左移 b 位（在后面添 b 个 0）。
- 例如 100 的二进制为 1100100，而 110010000 转成十进制是 400，那么 $100 \text{ shl } 2 = 400$ 。
- 可以看出， $a \text{ shl } b$ 的值实际上就是 a 乘以 2 的 b 次方，因为在二进制数后添一个 0 就相当于该数乘以 2。
- 通常认为 $a \text{ shl } 1$ 比 $a * 2$ 更快，因为前者是更底层一些的操作。
- 通常情况下，可以用 $1 \text{ shl } k$ 表示 2^k

位运算的使用

- 6. shr 运算
- 和 shl 相似， $a \text{ shr } b$ 表示二进制右移 b 位（去掉末 b 位），相当于 a 除以 2 的 b 次方（取整）。我们也经常
- 常用 $\text{shr } 1$ 来代替 $\text{div } 2$ ，因为位运算相对速度较快。

- 有了位运算。这道题就迎刃而解了。
- $\text{Mask and } 2^k$ 判断第k个人在不在集合S中。
- $\text{Mask} := \text{mask or } 2^k$ 将第k个人添加到集合S中。

核心代码

- for mask:=0 to all do
- for j:=1 to n do
- if (two[j] and mask) <> 0 then {枚举在S中的人j}
- for k:=1 to n do
- if ((two[k] and mask)=0) then {枚举不在S中的人k}
- f[mask or two[k], k] := {更新S'}
- min(f[mask or two[k], k], f[mask, j] + map[j, k]);

初始化

- `two[1]:=1;`
- `for i:=2 to 17 do`
- `two[i]:=two[i-1]*2; {定义two[i]为 2^{i-1} }`
- `all:=two[n+1]-1; {定义状态总数}`
- `for i:=1 to all do`
- `for j:=1 to n do`
- `f[i,j]:=1000000;`
- `for i:=1 to n do`
- `f[two[i],i]:=0;`

- 状态压缩是一种非常暴力的动态规划，特征也非常明显，一般适用于数据规模较小的情况。
- 状态压缩动态规划实际上也是按常规处理的一种动态规划的做法，只不过由于每个阶段状态数比较多，所以状态通常采用压缩存储的方式，以利于运算和转移（比如用二进制或三进制串表示），本质还是对状态的压缩，减少状态数。
- 应用特点：数据整体规模较小，或某一维较小。

DP 注意事项

- 一定要符合最优化原理，即满足最优子结构
- 可按照某一顺序，从小到大求解
- 从大到小求解可用记忆化搜索
- 注意边界条件
- 转移方程一定要清晰，不要漏情况
- 状态的意义一定要清晰

DP 的测试

- 一般来说动态规划程序短，易于查错。
- 但正是由于动态规划的程序短，导致任何细节写错都会出问题。
- 容易出错的地方：方程、初值、边界 etc。
- 所以放心不下的话，需要自己设计数据检查。

测试数据的设计

- 在我们的OI竞赛中，一道题目的正确性很大程度上决定了我们一次考试的成败，因此自测数据的设计显得十分重要，那我们在设计数据的过程中要注意些什么呢？
- 测试数据分很多种。一般而言可以把它们分成了小数据、随机数据和极限数据三种。

小数据的设计

- 样例无疑是所有Oier的最爱。大家学编程的时候就知道写完程序第一件事就是过样例，样例就是一个典型的小数据。小数据有两大优点：
- **易于调试**。对于静态查错能力弱而调试能力相对较强的同学而言，小数据很重要，绝大部分的错误都是靠小数据的跟踪调试找到的。
- **易于设计**。由于数据量小，我们往往可以手工设计质量更高的数据，同时对于数据本身也有直观的了解。与此同时，很多的题都会有所谓的“变态数据”，这和极限数据有着一些不同，它虽然数据量不大，但是剑走偏锋，由于这样那样的原因，我们就栽了跟头。为了使得自己的程序更加强壮，我们需要预先测试自己的程序是否能够通过这样的数据。

随机数据的设计

- 随机数据是属于那种数据量比小数据大，同时可以使用较弱的替代算法得到结果的数据。
- 一般的操作流程是这样的：
- 先写一个随机化的制作数据的程序；
- 暴力程序会写吧？
- 然后写一个针对题目的效率较差但是正确性能够保证的使用替代算法的程序；
- 最后使用一个批处理文件，进行多次对比测试，即生成一个数据，然后再比较两个程序的结果。（对拍）
- 一定要注意这三个程序的文件输入输出和批处理的实现，这些地方很容易出错。

- 随机数据具有以下特征：
- 数据量不同。数据量变大之后，对程序是一个新的挑战。一些更加难以发现的问题可能会显露出来。
- 可以随机化。与小数据不同，由于随机数据的数据个数过多，不能够穷举完成，因此推荐使用随机化。而随机化显然比穷举要容易编写得多，因此随机数据的实现更加方便。而随机化的缺点是，变态数据未必能够随机到。
- 而与极限数据相比，随机数据的优点是可以使用替代算法（暴力）。在极限数据的情况下，暴力程序往往会超时。

极限数据的设计

- 在测试完小数据和随机数据后，可以考虑测试极限数据。极限数据的目的在于检查正确性，因为这应该是小数据和随机数据的工作。
- 极限数据作为测试的最后一步，其数据可以针对性不强，但是它必须体现“极限”这一特点。
- 具体来说，它主要测试程序在极限情况下是否超时，或者数组是否越界，或者是否超出longint范围等等。

- 随机数据的bat不会写？
- 让我来演示一下。

- 对拍是检验程序正确性的利器，千万要熟练掌握！
- 需要强调的是，对拍成功的前提是暴力没有写错！

Thanks for listening!

Questions are welcome.