dp进阶之路

——我对于dp的一点浅薄的思考和总结

邓丝雨

2015 年 8 月 12 日

很久很久很久以前，我是看着dp就害怕的弱渣，直到某一天我竞赛班一个同学写了一篇《dp进行曲》，让我知道了学习dp就是刷题+总结方程，除了要见更多的模型，更重要的就是要多思考，就是想出状态后很自然的转移再外加找一个合理的边界……

于是，在那年noip之前，刷了几十道dp之后，就有所领悟了……虽然不敢说我真的会动态规划，但我很喜欢这种思考，也不再畏惧……

于是，我也想写一篇类似于《dp进行曲》的东西，与大家共勉!

说明：

由于大部分dp的难度都在于想出状态和方程，本报告绝大部分题目没有附上标程!

由于个人的一些原因，本报告里的例题有很大的一部分oi题目(都是让我脑洞大开的好题)。

而且作为题解，同等情况下我肯定还是希望写汉语题的，这样不用翻译题意哈哈!!（你究竟是有多懒？）

虽然本人希望本题解可以同时面向没有任何dp基础的同学和在暑假集训中学过dp基础的同学，但本人能力时间都有限，不可能做到面面俱到，所以我选择将dp基础部分的内容简单带过，没有dp基础，或者基础不好的同学可自行百度或者参考暑假集训资料!

dp绝对不止是我暑假进行讲课时分成几类那么简单，灵活的状态，多变的方程也许才是dp真正的精髓，虽然我所见所感只是皮毛，但是“运用之妙，存乎一心”是本弱学习dp的目标!

学习dp，就是要多思考和总结，正如《背包九讲》里说的**“‘思考’是一个oier最重要的品质”**

acmer 和oier 我感觉是一种生物，所以嘛……

PS：

对于最终把一份dp进阶指南写成了类似题目汇总归纳我表示万分惭愧……表示要是有希望通过这份总结来进一步学习dp的同学，千万别抱太多的希望，我现在自己都对自己很无语……泪流满面中……

PPS：

在上次听了羊大师讲dp之后，有了一些新的想法——羊大师一直在强调“子问题”这个概念，并且表示“子问题”可以用于确定状态，仔细想一想我深以为然，只要弄清楚了目标问题的子问题是什么，并且确立子问题之间的关系，即可解决题目，当然有一些问题并不是直接的子问题，需要转换其他问题的子问题!

目录

1 动态规划基本思想

1.1 基本思想

“动态规划是针对于一类求最优解问题的算法，其核心是**将一个问题分解为若干子问题**（类似于分治思想），通过每一次的最优决策，来得出一个最优解。” —from 南理工罗睿

在我看来，dp的核心原理是**分类加法原理**和**分步乘法原理**，通过这两个原理（加法原理用得多一些）,在当状态的上一个或几个状态中找出最优解求得当前状态，而对于前一个或者几个状态有采用同样的方法直到到达可以直接求出的边界状态（偶尔边界状态会很恶心……）

我高中的老师曾经说过：所有的题都是搜索。dp和搜索在思想上的异曲同工之妙还要好好体会啊!

（所以，在学会了搜索以后，记忆化搜索可能会是大部分人dp入门最简单的理解方式！但是到了后面大家可能都会越来越喜欢递推……）

1.2 适用条件

满足一下三点：

1、具有相同子问题：首先，我们必须要保证这个问题能够分解出几个子问题，并且能够通过这些子问题来解决这个问题。

2、满足最优化原理（最优子结构）:一个最优决策的子决策也是最优的。

3、具有无后效性: 这是动态规划中极为重要的一点（也是有可能被考察的一点），它要求每一个问题的决策，不能够对解决其它未来的问题产生影响，如果产生影响，就无法保证决策的最优性，这就是无后效性。往往这需要我们找到一个合适状态!（在后文中我们将看到一个例子用来展示巧妙消除无后效性!）

1.3 个人最常用的解题步骤

第一步：确定问题的子问题[1] 要点：注意分析哪些量随着问题规模的变小是会变小的，哪些变量与问题无关。

第二步：确定状态， 要点：结合子问题，敢想敢试，不要轻易否定一个状态，多思考，不要希望每个题都能一蹴而就!

第三步：推出状态转移方程 要点：注意验证适用条件是否满足，注意不要漏掉条件。

第四步：确定边界条件 要点：先根据状态含义确定，确定后验证第一层是否正确，如果不正确或者无法按含义确定，则也可以采用第一层的值反推的方式确定边界。

第五步：确定实现方式 要点：根据拓扑序是否明显和个人习惯确定!

第六步：如果需要的话，确定优化方法! 要点：注意优先考虑能否降维，不要局限于单调队列，四边形不等式这些标准的用于dp优化的东西，扩宽思维，避免定式!

[1]我之前一直都没有注意到这一步，但在大师讲过以后我觉得这样很有道理，尤其对于初学的同学以及一新的题目

1.4 资料提到的几种解题方法

注：从一个资料找到的方法，感觉还很有道理，可以借鉴!!

（1）模型匹配法：

最先考虑的就是这个方法了。挖掘问题的本质，如果发现问题是自己熟悉的某个基本的模型，就直接套用，但要小心其中的一些小的变动，现在考题办都是基本模型的变形套用时要小心条件，三思而后行。[2]

（2）三要素法

仔细分析问题尝试着确定动态规划的三要素，不同问题的却定方向不同：

先确定阶段的问题：数塔问题

先确定状态的问题：大多数都先确定状态的。[3]

先确定决策的问题：背包问题

一般都是先从比较明显的地方入手，至于怎么知道哪个明显就是经验问题了，多做题就会发现。

（3）寻找规律法：

这个方法很简单，耐心推几组数据后，看他们的规律，总结规律间的共性，有点贪心的意思。[4]

（4）边界条件法

找到问题的边界条件，然后考虑边界条件与它的领接状态之间的关系。这个方法也很起效。

（5）放宽约束和增加约束

这个思想是在陈启锋的论文里看到的，具体内容就是给问题增加一些条件或删除一些条件使问题变的清晰。

——from 《动态规划经典》

2 基础动规的一些模型

2.1 线性dp

线性dp的经典例题有：最长上升子序列（LIS），最长公共子序列（LCS），最大子序列和等等（默认经典例题大家已经会了……不会的请去看资料）

2.1.1 线性dp例题1：文件排版

题目大意：

写电子邮件是有趣的，但不幸的是经常写不好看，主要是因为所有的行不一样长，你的上司想要发排版精美的电子邮件，你的任务是为他编写一个电子邮件排版程序。完成这个任务最简单的办法是在太短的行中的单词之间插入空格，但这并不是最好的方法，考虑如下例子：

This is the exampleyou are

actuallyconsidering.

假设我们想将第二行变得和第一行一样长，靠简单地插入空格则我们将得到如下结果：

[2]本方法适用于经典模型的变形题，但是总是这样想会导致思维定式，并且在遇到一个没有见到的题目时可能会影响分析了……

[3]个人觉得这是一般都常用的方法

[4]递推式比较简单时适用，注意第一次推出来的不一定是最终的，有可能可以直接降维，eg：原方程$f[i][j]=\sum_{k=1}^{j}f[i-1][k]$ 可变为$f[i][j]=f[i-1][j]+f[i][j-1]$

This is the exampleyou are
actually considering.
但这太难看了，因为在第二行中有一个非常大的空白，如果将第一行的单词“are”移到下一行我们将得到较好的结果：
This is the example you
are actually considering.

当然，这必须对难看程度进行量化。因此我们必须给出单词之间的空格的难看程度，一个包含N个空格符的空白段，其难看程度值为$(n-1)^2$，程序的目的是使难看程度的总和最小化。例如，第一个例子的难看程度是1+7×7=50，而第二个例子的难看程度仅为1+1+1+4+1+4=12。

输出时，每一行的开头和结尾处都必须是一个单词，即每行开头和结尾处不能有空白。唯一例外的是该行仅有一个单词组成的情况，对于这种情况你可将单词放在该行开头处输出，此时如果该单词比该行应有的长度短则我们指定它的最坏程度为500，当然在这种情况下，该行的实际长度即为该单词的长度。

输入：

输入文件第一行是一个整数N，表示该段要求达到的宽度，$1 \le N \le 80$。该段文章由一个或多个单词组成，单词由ASCII码值为33到126（包含33 和126）的字符组成，单词与单词之间用空格隔开（可能超过一个）。单词长度不会超过段落要求达到的宽度。一段文字所有单词的总长度不会超过10000个字符，任何一行都不会超过100个字符，任何一个单词都在同一行内。

输出：

对于每个段落，找出使其难看程度最小的排版形式并输出句子：“Minimalbadness is B.”,B是指按可能的最好排版形式会发生的难看程度值。注意排版后文本行数任意，多余的空格也可删除。

FORMAT.IN

28
This is the exampleyou are
actually considering.

FORMAT.OUT

Minimal badness is 12.

f[i]表示第i个单词放在当前行的最后的最小难看程度，

f[i]=min(f[j] + cost(j，i))

(cost(j，i)表示j+1到i在同一行的最小难看程度，
我们显然可以知道，空格平均分配的难看程度最小)

不过要注意的是：题目中说：唯一例外的是该行仅有一个单词组成的情况，对于这种情况你可将单词放在该行开头处输出，此时如果该单词比该行应有的长度短则我们指定它的最坏程度为500————这句话的意思：

```
1         if (length(i)==n)  return 0;
2           else return 500;
```

思考：

这个题比较简单，不过我最开始想的状态就是二维的————第i行j个单词截止……然后实际上由于没有行数的限制，第几行这个条件根本不需要。（观察方程可以知道这一维不是必须的，但是最好是能够一开始就想明白!）

首先要说明的是省掉第几行这一维并不是就地滚动，这一维是根本不需要的枚举量!!! 我们只需要知道，前一行末尾是哪个就可以了……

其次，本来行就是个和这题没关系的因素，考虑到它完全是因为定式思维……

状态的设计和转移时所需的量是有对应关系的，我们总是说状态决定转移，但是转移又何尝不会影响了状态的选择呢？

2.1.2 线性dp例题2：三个串的最长公共子序列

题目大意：给你三个串，求其最长公共子序列，输出最长公共子序列的方案!（每个串长度不超过300）

思路：

这显然是一个最长公共子序列的衍生题，首先我们否定掉了先求两个串的最长公共子序列然后用这个去和剩下一个求的方法（可以想到这样不一定是最优解），（构造反例aaaabbbbcefg 和aabbefcg 和aabbc)

当状态不够的时候就加一维是最直观有效的方法之一，由于三个串对等的地位，所以我们可以猜想三个串的状态是对称的。

那么结合最长公共子序列的做法：状态可以设计为——f[i][j][k] 表示三个串前i，j，k个字符所能组成的最长公共子序列的长度

方程是显而易见的：

f[i][j][k] =max (f[i-1][j][k],f[i][j-1][k],f[i][j][k-1]);

如果s1[i]，s2[j]和s3[k]一样，那么：

f[i][j][k]=max (f[i][j][k], f[i-1][j-1][k-1]+1);

但是本题求的是**方案!!!!**

在这里顺便就说了，dp最常用的记录方案（或者方案数目）的办法是开一个数组，和dp的数组同一个结构（我在本题的描述中设dp的数组为f，方案的数组为g），在f数组发生转移的时候同时用g来记录当前状态是从哪里转移过来（或者是到当前状态的方案数目），那么，只需要在求出最后结果后回推回去就可以求出方案了（方案数就直接得出）

不过对于这个题来说这样做是有点麻烦的了，为什么这么说？因为每个状态是有三个参数的（那么g最好是个struct类型，记录三个值），然后转移又是四种情况，对应生成答案的方法还不一样，总之不是很好写的。

(dp的记录方案题比较少，虽然方法固定原理简单，但是写起来是有比较多的细节要注意的，所以如果能有更好的办法……)

那么这个题有没有其他方式呢？没有的话我说这么多不方便什么都得干嘛？

直接把f[i][j][k]搞成一个字符串……然后状态直接对应方案……

至于判断从哪儿转移的时候要用到最长公共串的长度不就可以通过这个串求得么？

说得再详细一点就没意思了……这个问题到此为止，欢迎讨论！[5]

小结：

Q:为什么在一般的dp记录方案中g不直接表示方案？

A:因为随着状态的转移转移方案往往越来越复杂（越来越长，包含的信息太多），在大部分情况下实在是太不实用[6]

Q:为什么这个题可以？

A：1.方案是一个字符串，可以很容易储存

2.而且这个字符串又可以表示全部用于转移的信息（长度），所以直接存就可以了!!

2.1.3 线性dp例题3：顺序对齐

题目大意：

考虑两个字符串右对齐的最佳解法。例如，有一个右对齐方案中字符串是aaddefgghc和adcdegh。

第一行:aad defgghc

第二行: adcde gh

每一个数值匹配的位置值2分，一段连续的空格值-1分。所以总分是匹配点的2倍减去连续空格的段数，在上述给定的例子中，6个位置(A,D,D,E,G,H) 匹配，三段空格，所以得分2*6+(-1)*3=9，注意，我们并不处罚左边的不匹配位置。若匹配的位置是两个不同的字符，则既不得分也不失分。

请你写个程序找出最佳右对齐方案。

输入

包含两行，每行一个字符串，最长50个字符。字符全部是大字字母。

输出

一行，为最佳对齐的得分。

input

AADDEFGGHC

ADCDEGH

output

9

思路：

状态设计为：f[i][j]表示两个字符串的**后i，j个**字符对应的最大值

转移分成三种情况：1.直接放在对应位置，两个字母不相同

2.放在对应位置，相同

3.有空格

那么转移方程为：

没空格：f[i][j] = f[i + 1][j + 1];

如果s1[i] == s2[j - 1] ： f[i][j] = max(f[i + 1][j + 1] + 2, f[i][j]);

如果s1前面有连续空格：f[i][j] = max(f[k][j] - 1, f[i][j]);

[5] 不要固守于一个现成的方法，多思考能不能有其他的办法更高效的解决这个问题!

[6]不要轻易的说一个方法不能，但也不要因为能就一定要用它，多思考，综合比较，具体情况要具体分析!!

如果s2前面有连续空格：f[i][j] = max(f[i][k] - 1, f[i][j]);

注意，由于前面的空格不算，所以答案是所有f[i][j]里最大的一个……

（假设ij之前都不等，最开头加空格）

思考：

前两天看算导，书上说递归是分治的思想，其实dp也是分治的思想，通过相同子结构把问题简化并缩小范围……

2.1.4 线性dp例题4：Number String

题目大意：

由数字1到n组成的所有排列中，问满足题目所给的n-1个字符的排列有多少个，如果第i字符是‘I’表示排列中的第i-1个数是小于第i个数的。如果是‘D’，则反之。

example：“ID”对应132，231 ； “I?”对应123 132 231

思路：

状态是一个很常见的状态——**f[i][j] 表示长度为i，末尾为j的序列的个数。**

由于求的是1..n的排列（不能有重复的元素），所以f[i][j]还包含一个条件————**每个元素小于等于i**

但是这样也就等于我每次都只能加i？可是最后一位是j啊？？

把(1,i-1)中大于等于j的变成j+1，这样既不破坏增减性又不影响结果数目————实质是满足了无后效性……

所以转移方程如下：

如果s[i - 1]等于I dp[i][j] = summ[i - 1][j - 1];

如果s[i - 1]等于D dp[i][j] = summ[i - 1][i] - summ[i - 1][j - 1];

如果s[i - 1]等于? dp[i][j] = summ[i - 1][i];

summ[i][j]表示长度为i，末尾小于等于j的所有序列之和。(边求解即可边维护前缀和)

思考：

关于排列的dp如何去除重复的问题————首先，在状态中往往包含最后一位的情况，然后状态转移的时候还有可能有状态的转化！如这个题中对于**i位串中的数都小于i**的限制，当i变成i+1的时候范围扩大，之前的限制发生了变化，所以要对状态进行转化！

dp的**相同子问题**和**最优子结构**不仅是判断动规的条件，也是要在动规的方程中体现出来的，将问题范围缩小的过程中（即寻找相同子问题的过程中），什么量是要一块缩小的，哪些量又是可以不缩小的，是值得思考的问题!!

本题还有一个注意点是：通过**前缀和**的维护将转移的复杂度降了一维，这提醒我们————即使是转移过程中会改变的量，如果满足前缀和的性质也是可以一边转移一边求和然后用的哦!!!

2.2 区间dp

区间dp其实也是**线性dp的一种**，只是由于其实在太规律，所以分成一类以方便学习!!

要点：状态必然包含区间是哪个[i,j]，通过**枚举区间分界点进行转移**。

也就是说一个大区间是由两个子区间合并来的或者是两个子区间加上中间元素合并来的！在合并的时候自然是要满足最优化原理和无后效性原则……

不能多说，到题目中去体会！

这类问题经常会遇到环，**环的处理办法**通常有两种：

1）**加倍**——将数据复制加倍，就可以保证最后一个与第一个相连；

2）**取余**——在调用数组时对n取余。

有可能需要提前处理合并区间的费用，如何处理视情况而定，不要忘记预处理和前缀和的办法！

经典例题有：石子合并，数链剖分，括号匹配，田忌赛马, 凸多边形三角划分等

2.2.1 区间dp例题1：凸多边形三角划分

题目大意：

给定一个具有N($N \leq 50$)个顶点(从1 到N 编号)的凸多边形，每个顶点的权均已知。问如何把这个凸多边形划分成N-2 个互不相交的三角形，使得这些三角形顶点的权的乘积之和最小

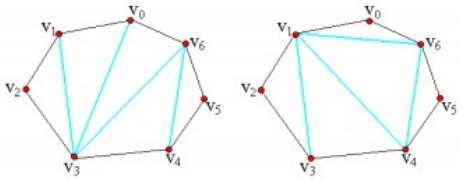
输入格式: 第一行为顶点数N；第二行为N 个顶点(从1 到N)的权值。

输出格式: 最小的和的值。

输入样例: 5 121 122 123 245 231

输出样例: 12214884

凸多边形三角剖分如下图所示：



思路：

这是一个比较基础的区间dp……但是貌似初学者看到这个题总是很无力的样子……设**F[I,J]**($I < J$)**表示从顶点I 到顶点J 的凸多边形三角剖分后所得到的最大乘积**，我们可以得到下面的动态转移方程：

f[I,J]=Minf[I,K]+f[K,J]+s[I]×s[J]×s[K](i<k<j)

显然，目标状态为：f[1,n]

思考：

区间dp是动态规划里**分治思想**体现的最明显的一个类型，通过枚举区间分割点将大区间分成两部分（也有可能是三部分，后面有题目会提到）

具体对于本题来说，是把一个大的多边形分割成两个小的多边形，然后合并，只要我们知道问题是满足dp的性质的就只需要着眼于当前区间的操作，就可以保证转移是正确的了

其实这个题和石子合并没多大区别，只是换了一种描述可能就会让人觉迷茫……由于有一个几何模型的存在，要看出其实就是石子合并模型就貌似略有困难了……但是实际上问题是怎么转化来的呢？

给你一个n个数序列，它首尾相接围成一个环，每次拿走一个数，所付出的代价值是他乘以他左右两边的数，求拿走n-2个数所付出的最大代价和……

在去掉几何模型后问题有没有变得直观一点呢？？？也许对于这个题来说并非如，但是有了这个模型，下一个题也许就容易许多了！

2.2.2 区间dp例题2：HDU 5115 Dire Wolf

题目大意：

有n头狼排成一排，每只狼都对相邻的狼的攻击力有加成作用，每杀死一只狼所受到的伤害为当前狼的攻击力（算上加成的部分），被杀死之后的狼对相邻的狼的攻击力的加成会被取消，同时，原先与被杀死的狼相邻的两头狼会变成相邻的狼。要求使得受到的伤害值最小，求出最小值。

思路：

是不是跟上一个题的抽象模型差不多？这不就变成区间dp水题了么……

dp[i][j]表示从第i头狼到第j头狼全部被杀死所受到的最小伤害。

a[i]表示第i头狼的初始攻击力，b[i]表示第i头狼对相邻狼的加成值。

dp[i][j]=min(dp[i][k-1]+a[k]+dp[k+1][j])+b[i-1]+b[j+1]; (i < k < j 且j > i)

对于区间[i,j]：**先打k的左右两边再打k** [7]

dp[i][i]=a[i]+b[i-1]+b[i+1]; (边界)

思考：

有人问我这样先打区间左边那个再打剩下的那个不就没包含进去？好好想想是这样么？它到底是包含在哪种情况里面呢？

高中老师说过，dp就是：**枚举最后一次决策!** 只要**最后一次决策涵盖到了所有转移过来的情况**，那么就涵盖了所有情况了……

2.2.3 区间dp例题3：ZOJ 3469 Food Delivery

题目大意：

一个人他外卖员要去送外卖，他的店在X位置，他的速度为V，然后现在有N个人要外卖，这N个人的坐标分别为Xi，第 i 个人每等一分钟不满意度会增加Bi，他希望送完所有人让总的不满意度最少，求最少的不满意度是多少……（语死早，总是描述不清楚题目在说什么）

思路：

看起来确实像个dp……（废话，不是dp放这里干嘛）可是可是怎么dp呢？

首先我们可以知道，他肯定是从店铺所在位置向两边送（至于每次两边送多远就是需要dp算的了）

那么，用**f[i][j][0]表示送完[i,j]这个区间且送货员停在i 位置,的最小不满值,f[i][j][1]表示送完[i,j]这个区间且送货员停在j位置,的最小不满值**

f[i][j][0] = min(f[i][j][0], f[i + 1][j][0] + (a[i + 1].x - a[i].x)×(add + a[i].v)) * v;

f[i][j][1] = min(f[i][j][1], f[i][j - 1][1] + (a[j].x - a[j - 1].x)×(add + a[j].v)) * v ;

f[i][j][0] = min(f[i][j][0], f[i + 1][j][1] + (a[j].x - a[i].x)×(add + a[i].v)) * v;

f[i][j][1] = min(f[i][j][1], f[i][j - 1][0] + (a[j].x - a[i].x)×(add + a[j].v)) * v;

其中add 是所有不在[i,j]区间里的每个客户的单位时间不满意度增长量之和(可用前缀和O(1)求出)

[7]自己定义一个打的方案，或者区间处理顺序并没有什么不可以的，只有满足动态规划的性质！

因为**add + a[i].v 为每个客户的单位时间不满意度增长量之和**，而(a[i + 1].x - a[i].x)等为等待的路程，另外，设V是速度的倒数……

所以等待的时间是(a[i + 1].x - a[i].x)×V

但是，如果把V直接在转移的时候乘进去的话中间结果会爆int，得用long long,所以可以一直不乘V，到输出的时候再乘上!

思考：

这个道题告诉我们在区间dp中当在区间左界和右界情况不一样的时候还可以，用**加一维01状态**的方式区分……

用01区分两种情况(或者01234……区分多种情况)其实是个很常见的思路……

关于add的求法多说一句————请不要忘记前缀和这个很有效的工具!

2.3 背包问题

要点：背包转移都是统一的思路——枚举当前物品装还是不装（装几个，怎么装）（难怪《动态规划经典》那份资料说：背包问题是**先确定转移再确定的状态**! 他的转移真的是很规律啊）

（我一个高中同学说背包问题算法都是从01背包上衍生出来的，想想也还是有一定的道理!）

由于背包九讲对于包问题讲的相当详细，所以此时，我就挑几个问题来说!

要想学通背包问题，背包九讲至少要看三遍吧!!!

2.3.1 各种问题向01背包的转化

首先，01背包（枚举取还是不取）的方程为：f[i,j]=maxf[i-1,j],f[i-1,j-w[i]]+v[i]，利用就地滚动可以写为**f[j] = max(F[j],f[jC[i]] + W[i]) (j从大到小枚举)**

完全背包（枚举取多少个）F[j] = max(F[j],f[j-C[i]] + W[i]) (j 从小到大枚举) （完全背包和01背包只是枚举顺序不同……但是却实现了不同的功能!）

多重背包（枚举取多少个）方法是：“将第i 种物品分成若干件01 背包中的物品，其中每件物品有一个系数。这件物品的费用和价值均是原来的费用和价值乘以这个系数。这些系数分别为$1, 2, 2^2...2^k`1, Mi - (2^k) + 1$,且k 是满足$Mi` 2^k + 1 > 0$ 的最大整数”——《背包问题九讲》

多重背包转化的思路其实是这样的：把第i种物品换成若干件物品，使得原问题中第i种物品可取的每种策略——取0..n[i]件——均能等价于取若干件代换以后的物品。另外，取超过n[i]件的策略必不能出现。

将n[i]拆成$1, 2, 4, ..., 2^(k - 1), n[i] - (2^k) + 1$，（k是满足$n[i] - 2^k + 1 > 0$ 的最大整数）道理何在？(还记得暑假集训ppt上是怎么讲的么？去观察一个数的**二进制拆分**!)

二维费用的背包问题——加一维就好了[8]

分组的背包问题(枚举还是不取，取哪一个) ——f[k][v]=maxf[k-1][v],f[k-1][v-c[i]]+w[i] 物品i属于组k，（注意循环顺序，枚举k在最外层）

[8]当你发现一个新问题是在你熟悉的dp的基础上添加了限制条件而来时，往往给状态加一维是一种很有效的方法

有依赖的背包问题——转化成物品组（取主件，取主件和某几个附件的物品组）

2.3.2 背包问题例题1：机器分配

题目大意：

某总公司拥有高效生产设备M台，准备分给下属的N 个分公司。各分公司若获得这些设备，可以为总公司提供一定的盈利。问：如何分配这M 台设备才能使国家得到的盈利最大？求出最大盈利值。

分配原则：每个公司有权获得任意数目的设备，但总台数不得超过总设备数M。其中$M \leq 100$, $N \leq 100$。

输入数据：

第一行为两个整数M，N。接下来是一个$N \times M$的矩阵，其中矩阵的第i行的第j列的数Aij表明第i个公司分配j台机器的盈利。所有数据之间用一个空格分隔。

输出数据：

只有一个数据，为总公司分配这M台设备所获得的最大盈利。

input

3 2

1 2 3

2 3 4

output

4

.

思路：

转移是：枚举这台设备让哪一个公司用

所以状态是：f[i,j]表示前i个公司分配j台机器所能获得的最大值。

f[i,j]=max(f[i-1,j],f[i-1,j-k]+cost[i,k]) (cost[i,k]表示第i个公司分配k台机器的获利)

思考：

这算是个比较简单的题……枚举每个物品放到哪一个背包里面，虽然不属于《背包九讲》介绍到的经典类型，但是他依然是使用了背包的思路

《背包问题九讲》中提到了一个泛化物品的概念——

“考虑这样一种物品，它并没有固定的费用和价值，而是它的价值随着你分配给它的费用而变化。这就是泛化物品的概念。

更严格的定义之。在背包容量为V 的背包问题中，泛化物品是一个定义域为0...V 中的整数的函数h，当分配给它的费用为v 时，能得到的价值就是h(v)。

这个定义有一点点抽象，另一种理解是一个泛化物品就是一个数组h[0...V]，给它费用v，可得到价值h[v]。”

如果利用泛化物品思想来想这个题，是不是就觉得有新的感触？

——于是问题转化成：背包容量为M，有N个物品，这些物品的体积和价值的对应关系给出。求装满背包能获得的最大价值。

此时：g[i][j]表示前N个物品，装满体积为M的背包所获得的最大价值……方程和上面一样orz……

不过思维方式是有所改变的哦!

我在见到这个题之前，一直觉得泛化物品只是一个可有可无的概念，但是看过这个题以后，觉得这种思维方式也是很有意思的……然后我就发现各类背包都可以化归为泛化物品……虽然这样做时间复杂度有所提高，但是很有趣，大家也可以自己分析一下!

2.3.3 背包问题例题2：多人背包

题目大意：

DD 和好朋友们要去爬山啦！他们一共有K 个人，每个人都会背一个包。这些包的容量是相同的，都是V。可以装进背包里的一共有N 种物品，每种物品都有给定的体积和价值。在DD 看来，合理的背包安排方案是这样的：每个人背包里装的物品的总体积恰等于包的容量。每个包里的每种物品最多只有一件，但两个不同的包中可以存在相同的物品。任意两个人，他们包里的物品清单不能完全相同。在满足以上要求的前提下，所有包里的所有物品的总价值最大是多少呢？

输入格式

第一行有三个整数：K、V、N。($k \leq 50 v \leq 5000 n \leq 200$)

第二行开始的N 行，每行有两个整数，分别代表这件物品的体积和价值。

输出格式

只需输出一个整数，即在满足以上要求的前提下所有物品的总价值的最大值。

样例输入

2 10 5

3 12

7 20

2 4

5 6

1 1

样例输出

57

这是一个很简单的典型例题，**背包问题第K优解**，在背包九讲中也有讲解[9] ，使用队（shu）列（zu）就可以实现，方程改变为f[i] = merge(f[i - v[j-1]] + c[j]) 其中**f是一个单调的队列**，merge 是**队列的合并操作!**

关于怎么把两个有序的队列合并到一起仍保持有序，参考归并排序的写法……

2.3.4 背包问题例题3：poj1015

题目大意：(本题目描述来自网上)

[9]似乎最新版已经把这一块删掉了，可以去看之前的版本

在遥远的国家佛罗布尼亚，嫌犯是否有罪，须由陪审团决定。陪审团是由法官从公众中挑选的。先随机挑选n个人作为陪审团的候选人，然后再从这n个人中选m人组成陪审团。选m人的办法是：

控方和辩方会根据对候选人的喜欢程度，给所有候选人打分，分值从0到20。为了公平起见，法官选出陪审团的原则是：选出的m个人，必须满足辩方总分和控方总分的差的绝对值最小。如果有多种选择方案的辩方总分和控方总分的之差的绝对值相同，那么选辩控双方总分之和最大的方案即可。

输入:

输入包含多组数据。每组数据的第一行是两个整数n和m，n是候选人数目，m是陪审团人数。注意:$1 \le n \le 200, 1 \le m \le 20$ 而且$m \le n$。接下来的n行，每行表示一个候选人的信息，它包含2个整数，先后是控方和辩方对该候选人的打分。候选人按出现的先后从1开始编号。两组有效数据之间以空行分隔。最后一组数据$n = m = 0$

输出

对每组数据，先输出一行，表示答案所属的组号,如'Jury ♯ 1' 等。接下来的一行要象例子那样输出陪审团的控方总分和辩方总分。再下来一行要以升序输出陪审团里每个成员的编号，两个成员编号之间用空格分隔。每组输出数据须以一个空行结束。

样例输入

4 2

1 2

2 3

4 1

6 2

0 0

样例输出

Jury ♯ 1

Best jury has value 6 for prosecution and value 4 for defence:

2 3

思路：

在面对一个dp的时候，如果已知了他是一个背包问题，我们最重要的是分析出什么？

——我觉得有三点：这个背包的体积是什么？物品的价值是什么？物品的费用又是什么？

想清楚了这三个问题，事情就解决了。

对于本题来说，这是一个二维费用的背包问题——一个费用是控辩差，这个费用比较特殊，有正有负，而我们最后取的是最接近0的那一个；另一个费用是1，这是为了保证最终选了m个人；而物品的价值是控辩和。这样不就解决问题了么？

那么，怎样判断一个问题是背包问题呢？

正如之前分析背包问题的做法的时候所说的，做背包的第一步是分析转移——取还是不取，取几个，那么当我们看到题目是要决定选还是不选选几个的时候，就可以往背包问题的方向想了！

2.4 树形dp

要点：树形dp也比较明显的体现了分治的思想，先第一个子树，再第二个子树……然后合并……

此类dp虽然方程变化多端，但是都以一个共同点：**每一个父亲的值都是其由各个儿子决定，采取记忆化搜索的形式来实现**。换句话说，树形dp就是一个**后序遍历!!**

这一类的题往往给出的是一个**无根树**，往往你就可以给他们安一个根的，关于树的储存问题，一般是用链表（邻接表）+遍历染色实现的，实际上是不需要真的去构建一棵树，只需要记录父子关系，手中无树，但**心中有“树”!**

还有关于大家都知道的“左儿子右兄弟”的多叉转二叉，表示其实在树形dp里不是必须的，可能有的题来说变成二叉树会方便一点（不过我没遇到过），但是我在此要说明的是多叉转二叉也可以通过心中有树的方法来实现……

所以我一直觉得这类dp是最好入门不过的，但是，有难度的题也是很难的。。。

网上的一个blog里将树形dp又分成了三类：常规树形DP，树形背包问题，删点或者删边类树形DP，虽然好像不是太全但是感觉还是有道理的。

（blog地址：http://blog.csdn.net/woshi250hua/article/details/7644959）

另外，树分治也可以说是树形dp，于是……

2.4.1 树形dp例题1：HDU 2196 Computer

题目大意：

给一棵树，每条树边都有权值ci，问从每个顶点出发，经过的路径权值之和最大为多少？$n \leq 10000$,$ci \leq 10^9$

输入样例：

5
1 1
2 1
3 1
1 1

输出样例：

3
2
3
4
4

思路：

当我们把这个树转化成一个有根树以后，最长的路径是两种情况——**从当前点到它的子树**，或者**从当前点经过他的根……**

所以**两次dfs**就可以了，第一次维护一个点到他的子树的最长路，第二次用来求结果，但是这个时候有一个问题：

判断一个点的从父节点过来的最大值，那么如果他的父节点存的最大值正好是从该点过来的，那么就失去了从父节点过来的状态了!

怎么办？

把第二大值也记下来！同时**记录当前的最大值和次大值是从哪一个点转移过来的，**转移的时候判断一下就可以了**!!**

代码：

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <string>
#include <algorithm>
#include <cmath>
#include <vector>
#include <string>
using namespace std;

const int N = 10010;
struct ty
{
  int to, next, len;
}edge[N * 2];
int head[N];
int n, tot;
int maxn[N], maxid[N];
int tmp[N], tmpid[N];

void AddEdge(int x, int y, int z)
{
  edge[tot].to = y;
  edge[tot].len = z;
  edge[tot].next = head[x];
  head[x] = tot++;

  edge[tot].to = x;
  edge[tot].len = z;
  edge[tot].next = head[y];
  head[y] = tot++;
}

void Dfs1(int x, int fa)
{
  maxn[x] = tmp[x] = 0;
  for (int i = head[x]; i != -1; i = edge[i].next)
  {
    int y = edge[i].to;
    if (y != fa)
    {
      Dfs1(y, x);
      if (tmp[x] < maxn[y] + edge[i].len)
      {
        tmp[x] = maxn[y] + edge[i].len;
        tmpid[x] = y;
        if(tmp[x] > maxn[x])
        {
          swap(tmp[x], maxn[x]);
```

```
51                  swap(tmpid[x], maxid[x]);
52                }
53              }
54            }
55          }
56        }
57
58        void Dfs2(int x, int fa)
59        {
60          for(int i = head[x]; i != -1; i=edge[i].next)
61          {
62            int y = edge[i].to;
63            if(y == fa)continue;
64            if(y == maxid[x])
65            {
66              if(tmp[y] < edge[i].len + tmp[x] )
67              {
68                tmp[y] = edge[i].len + tmp[x];
69                tmpid[y] = x;
70                if(tmp[y] > maxn[y])
71                {
72                  swap(tmp[y], maxn[y]);
73                  swap(tmpid[y], maxid[y]);
74                }
75              }
76            }
77            else
78            {
79              if(tmp[y] < edge[i].len + maxn[x])
80              {
81                tmp[y] = edge[i].len + maxn[x];
82                tmpid[y] = x;
83                if(tmp[y] > maxn[y])
84                {
85                  swap(tmp[y], maxn[y]);
86                  swap(tmpid[y], maxid[y]);
87                }
88              }
89            }
90            Dfs2(y, x);
91          }
92        }
93        int main()
94        {
95          while(scanf("%d",&n) != EOF)
96          {
97            memset(head, -1, sizeof(head));
98            tot = 0;
99            for(int i = 2; i <= n; i++)
100           {
101             int x, y;
102             scanf("%d%d", &x, &y);
103             AddEdge(i, x, y);
104           }
105           Dfs1(1, -1);
```

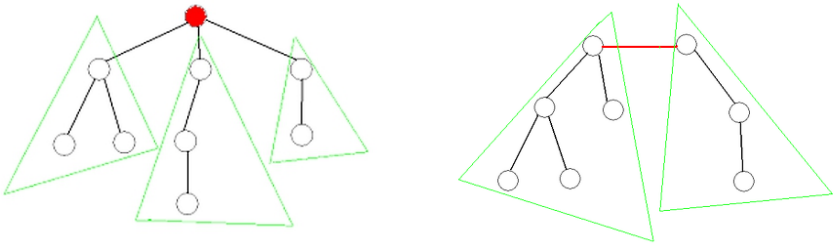
```
106             Dfs2(1, -1);
107             for(int i = 1; i <= n; i++)
108               printf("%d\n", maxn[i]);
109           }
110           return 0;
111         }
```

2.4.2 树分治的简要理论介绍

树分治大体分为两类：**点分治**和**边分治**

点分治是指：首先选取一个点将无根树转为有根树，再递归处理每一颗以根结点的儿子为根的子树（这和树形dp有异曲同工之妙啊）

边分治是指：在树中选取一条边，将原树分成两棵不相交的树，递归处理。



为了在效率上比较好，我们选取的根应该是树的**重心点**，选取的边应该是树的**中心边**。所谓“重心”指的是：将其删去后，结点最多的树的结点个数最小。而中心边则是可以使得所分离出来的两棵子树的结点个数尽量平均的边。而对于求重心和中心边这两个问题，都可以使用在树形动态规划来解决，时间复杂度均为O(N)，其中N为树的结点总数。

给出国家集训队论文的链接：http://wenku.baidu.com/view/e087065f804d2b160b4ec0b5.html###

树分治究竟怎么做？让我们通过例题来学习!!

2.4.3 树形dp（树分治）例题：POJ 1741 Tree

题目描述：给定一棵N ($1 \leq N \leq 100001$)个结点的带权树，定义dist(u,v)为v,u两点间的最短路径长度，路径的长度定义为路径上所有边的权和。再给定一个K($1 \leq k \leq 10^9$)，如果对于不同的两个结点a, b，如果满足$dist(a,b) \leq k$则称a, b为合法点对。求合法点对个数。

分析：

普通的dfs，复杂度是O(n^2)的，所以必须想办法!

显然，我们很清楚的知道一条路径要么过根结点，要么在一棵子树中，根据树分治的思想：路径在子树中的情况只需递归处理即可，路径过根结点的情况则需要处理。

那么过根结点怎么处理呢？?

如果我们能找到以某点为根的子孙节点到根的距离，然后从这些距离里面找出两个距离之和小等于k的方案数是不是就可以了？

否！如果这两个点在同一个子树，那么也会被计算进去，而它之前已经被计算过了……所以说如果用depth(i)表示i到根结点的路径长度，belong(i)表示i点所属于子树的根。那么所求就是满足depth(i)+depth(j)≤ k且belong(i) ！= belong(j)　的个数

即求：满足depth(i) + Depth(j)≤ k的i, j个数–满足depth(i) + Depth(j)≤ k且belong(i) = belong(j)的i, j个数

depth 和belong 的求法是最基础的树状dp……而把距离都算出来之后，要怎么快速找到方案数呢？O(N^2)，想着O(n)解决，先对距离序列排序，然后找头尾两个数，如果符合情况，算中间的个数，然后从头的下一个开始算，如果不符合情况说明太大，要从尾的前一个开始计算……然后就ok了！

提醒一点，对于子树的计算，由于本题要求关于子树的情况和哪一个点为根无关，对于每一个子树也是要以重心为根的!!

代码：

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <string>
#include <algorithm>
using namespace std;
const int N  = 21000;

struct tynode
{
  int t, len;
  int sum, maxn;
  tynode *next;
} tree[N], *head[N], tp[N];
int n, ptr, ans, root;
int tot, k, dist[N], vis[N], size[N], sign[N];
void AddEdge(int a, int b, int c)
{

  tree[ptr].t = b;
  tree[ptr].len = c;
  tree[ptr].next = head[a];
  head[a] = &tree[ptr++];
}
void Dfs(int x, int past)
{
  tp[x].sum = tp[x].maxn = 0;

  tynode *p = head[x];
  while (p != NULL)
  {
    if (p->t != past && vis[p->t] == 0)
    {
      Dfs(p->t, x);
      tp[x].sum += tp[p->t].sum;  //sum: the sum of son points
      tp[x].maxn = max(tp[x].maxn, tp[p->t].sum);  //maxn: the biggest size of the sons
    }
    p = p->next;
  }

  tp[x].sum++;   //add the root
  sign[tot] = x;  // hash
  size[tot++] = tp[x].maxn;  //record  the max size
```

```
45          }
46          int GetRoot(int x)  // find the specific root of the tree
47          {
48            tot = 0;
49            Dfs(x, 0);  // dfs to calculate the data we need
50
51            int maxn = 0x7f7f7f7f, maxsign = 0, cnt = tp[x].sum;
52
53            for (int i = 0; i < tot; ++i)  // find all the point
54            {
55              size[i] = max(size[i], cnt - size[i]);
56              if (size[i] < maxn)
57              {
58                maxn = size[i];
59                maxsign = sign[i];  // get the smallest
60              }
61            }
62            return maxsign;
63          }
64
65          void CalcDist(int s, int past, int dis) // to calculate every  'points dist to the root
66           {
67            tynode *p = head[s];
68            dist[tot++] = dis;
69
70            while (p != NULL)
71            {
72              if (p->t != past && vis[p->t] == 0 && dis + p->len <= k)
73                CalcDist(p->t, s, dis + p->len);
74              p = p->next;
75            }
76          }
77
78          void CountSum(int x)// calculate the sum of depth(i) + depth(j) >= k
79          {
80            sort(dist, dist + tot);
81            int left = 0, right = tot - 1;
82
83            while (left < right)
84            {
85              if (dist[left] + dist[right] <= k)
86                ans += right - left, left++;
87              else right--;
88            }
89          }
90          void CountDel(int x)  // calculate the sum of depth(i) + depth(j) >= k && belong(i) ==
                belong(j)
91          {
92            vis[x] = 1;
93            tynode *p = head[x];
94
95            while (p != NULL)
96            {
97              if (vis[p->t] == 0)
98              {
```

```
99              tot = 0, CalcDist(p->t, x, p->len); //to search the point in the same subtree (the
                    belong[i] of these points are the same)
100
101             sort(dist, dist + tot);
102             int left = 0, right = tot - 1;
103             while (left < right)
104             {
105               if (dist[left] + dist[right] <= k)
106                 ans -= right - left, left++;
107               else right--;
108             }
109           }
110           p = p->next;
111         }
112       }
113       void Deal(int s, int past) // s: the root of the present tree  //past: father of s
114       {
115         root = GetRoot(s);    // get the  '' heart of the tree
116         tot = 0;
117         CalcDist(root, 0, 0); // dp to calculate the dist to the root
118         CountSum(root);       // calculate the sum of depth(i) + depth(j) >= k
119         CountDel(root);      // calculate the sum of depth(i) + depth(j) >= k && belong(i) ==
                belong(j)
120
121         tynode *p = head[root]; //deal with the sons of the root
122         while (p != NULL)
123         {
124           if (p->t != past && vis[p->t] == 0)   Deal(p->t, root);
125           p = p->next;
126         }
127       }
128       int main()
129       {
130         while (scanf("%d%d", &n, &k) != EOF  && n + k != 0)
131         {
132           ans = ptr = 0;
133           for (int i = 0; i < N; ++i)
134             vis[i] = 0, head[i] = NULL;
135
136           for (int i = 1; i < n; ++i)
137           {
138             int a, b, c;
139             scanf("%d%d%d", &a, &b, &c);
140             AddEdge(a, b, c);
141             AddEdge(b, a, c);
142           }
143
144           Deal(1, 0);
145           printf("%d\n", ans);
146         }
147         return 0;
148       }
```

表示自己写了一遍死活过不到样例再看标程发现自己写得好挫……然后又照着标程来了一遍……orz啊……

思考：树分治和树形dp的区别和联系在哪里？？

首先说联系：都是分别处理子树然后合并……也就是说都是利用了分治的思想！而且通过这个题我们可以发现树分治的很多处理都是用的树形dp!!!

关于区别：树形dp对于子树来说，根是一定的，而树分治则是可以**将每一个子树完全独立，单独求一个重心作为根……**

表示树分治我也只做了这么一个题……理解不透彻是必然的，群巨有什么见解请一定指教!!

2.5 状压dp

状压dp不是那么神秘，他的转移方程一般都是普普通通，只不过其中的一维或多维是“压缩的”，即把一个状态(一个方案、一个集合等)压缩成一个整数。

这很明显是一个Hash的过程，所以状压DP又被称为**Hash DP或集合DP**。

除去这一点区别，它和普通递推DP没有什么差别，都是从已有的状态推知新的状态，所做的决策都没有后效性……

以状压dp比较简单的例题《互不侵犯》（暑假讲课ppt状压例二）为例来说两句——

这个题为什么用状压dp？因为普通的dp不行！普通的dp的状态都是一个一个格子转移的，而我现在需要一行一行的转移，也就是说我所处在的状态是一个集合!!! 又因为这个集合可以压缩成一个数值，用状压dp就再好不过了!!

如果还没明白，那么我们再用tsp问题为例——

tsp问题的关键在于走过了哪些点和现在在那个点，而如果我们先让tsp问题的解法满足无后效性，就必须将哪些点走过这一个信息带在状态里，所以以状压的方式保存这一信息!

明白我要说什么了么？

dp的状态究竟包含了什么？

个人觉得，**dp的状态其实就是决定答案的核心变量**，如果核心变量是一个两个或者几个单独的量那么正常的dp就行，如果核心变量中包含着一个集合，或者说**一系列相关的状态**（比如互不侵犯里每一行放国王的方式就是互相相关的）那么就会考虑集合dp，而如果方便的话，就会对集合的表示进行二进制（或者四进制等）的压缩!

所以，总而言之，状压dp中，状压其实仅仅是一种存储信息的手法!

当你通过分析本题结果的核心变量有哪些的方法去找状态的时候，你往往就有可能看到状态里哪些是可以优化是可以省掉，然后你有可能就会降维或者是常数优化了……

当然，这是做了足够题后的美好的愿望……（反正本弱现在还没有达到……）

2.5.1 状压dp例题1：混乱的队伍

题目大意：

LongDD的N($4 \leq N \leq 16$)个员工每人都有一个唯一的编号Si($1 \leq Si \leq 25,000$). 员工们为他们的编号感到骄傲,所以每个人都把他的编号刻在一个金牌上, 并且把金牌挂在他们的脖子上.

员工们对在吃饭的时候被排成一支“混乱”的队伍非常反感. 如果一个队伍里任意两相邻的人的编号相差超过K ($1 \leq K \leq 3400$), 它就被称为是混乱的. 比如说，当N = 6,K = 1 时,1, 3, 5, 2, 6, 4 就是一支“混乱”的队伍, 但是1, 3, 6, 5, 2, 4 不是(因为5和6只相差1). 那么, 有多少种能够使员工们排成“混乱”的队伍的方案呢?

输入格式:

第1 行:用空格隔开的两个整数N 和K

第2..N+1行: 第i+1行包含了一个用来表示第i个员工的编号的整数: Si

输出格式

第1 行:只有一个整数, 表示有多少种能够使员工们排成“混乱”的队伍的方案. 答案保证是一个在64位范围内的整数.

输入样例

4 1

3

4

2

1

输出样例

2

思路：

T表示状态：T的第i位为1则第i个人在队列中。

f[T，j]表示状态为T，最后一个人的编号为j时的方案数。

那么前一个人的编号与j相差必然大于k

f[T，j]= σ **f[T'，p]** ，其中$abs(s[j]-s[p])>k$且T’ +(1 <<(j-1))=T

思考：

记得这个题去年我写题解的时候把他归结到了“状压只会tsp里面”，为什么呢？

仔细看这个方程是不是跟tsp问题一模一样？

本题可以转化成有n个点，当点的标号差大于k的时候就连一条边，求TSP回路的个数!

于是这个题就牵涉到dp和图论之间的转化了……

2.5.2 状压dp例题2：POJ 2411 Mondriaan'sDream

题目大意：

一个矩阵，只能放1×2的木块，问将这个矩阵完全覆盖的不同放法有多少种。

输入样例(n M : 矩阵规格N×M)

1 2

1 3

1 4

2 2

2 3
2 4
2 11
4 11
0 0

输出样例

1
0
1
2
3
5
144
51205

思路：

如果是**横着的就定义11**，如果**竖着的定义为01**，那么状态能否转移的判断是：当前AND行上一行，是不是全为1（不可能出现竖着的00啊），而某一个状态是否合法的判断是：检查当前行里有没有横放的，但为奇数的1——取反就是当前行竖放的状态，那么当前行状态-上一行取反就是当前行的人横放，然后判断这个数中所有连续的1是不是都是偶数个就比较简单了!

状态: f[T][i]第i行状态为T时候的方案数

f[T][i] = f[T][i] + f[T'][i-1] T'为i-1行的，不与i行状态T冲突的状态

第一行：符合条件的状态记为1 即f[T][0] = 1

思考：

状压dp的一个个重点就是定义状态，什么为0什么为1是很重要的，既要 **方便表示**，更要**方便判断**，如果状态之间能否转移及状态是否合法不能使用位运算判断的话会可能就会比较痛苦。

对于这个题来说，还有很重要的一点——**竖向方块本来对于每一行来说有两种情况：覆盖本行和上一行，覆盖本行和下一行**。由于这两种情况的存在，如果这样考虑将会非常麻烦，但是如果我们**把竖着放当成从下往上放**就简单多了，而这样想就会得出上述的状态设计!!!

而且这样就把相邻了两行的状态有效的衔接在了一起(注意千万别乱想当前行和上一行是这样链接的，我们如果考虑的是i到i+1，就应该考虑i-1到i而不是i到i-1 递推方向始终相同不能发生变化，这样才能保证是**相同子问题**！)

2.5.3 状压dp例题3：HDU 5305 Friends

题目大意：

给出n个点m条边($n \leq 8$)，每个边可能是黑色也可能是白色，求每个点相连的边黑边和白边的数量都相等的图有几种。

思路：

这个题大家都是用搜索+剪枝做的，但是知道了这个题的dp做法以后真的是让人感叹啊！

首先，若某个点的度是奇数，就直接不可能了，然后再考虑dp的状态！

我们按照边的顺序一条一条染色（默认所有边原来是白色的），f[i][T] 表示染完（染黑或者染白）第i条边，状态为T的时候的方法数。

T由n个两位的二进制数组成，每两位表示一个点所连的黑边的数目，转移什么的都是显而易见的事情，不再多说！

思考：

之所以把这个本来是搜索的题的状压dp的做法放在这里，主要是因为其体现了状压只是一个手段这一点，而且从这个题来看转移的手段是非常灵活的，比如在插头dp里常见的四进制，和这个题里面的两位二进制，其实有异曲同工之处！

当然，这种情况下，对位运算的灵活运用就是蛮重要的了，其实呢，个人觉得只要心中把这个数当成是二进制数想清楚该怎样使用位运算还是比较容易的！

代码：

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <cmath>
#include <algorithm>
#include <queue>
#include <map>
using namespace std;
int n, m;
int edge[100][3], inc[100];
int f[40][1<<17];
int main()
{
  int t;
  scanf("%d", &t);
  while (t--)
  {
    memset(edge, 0, sizeof(edge));
    memset(inc, 0, sizeof(inc));
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; i++)
    {
      int x, y;
      scanf("%d%d", &x, &y);
      x--,y--;
      edge[i][0] = x;
      edge[i][1] = y;

      inc[x]++;
      inc[y]++;
    }
//--------------------------
    //特判 某些点入度是奇数的情况
    bool boo = 1;
    for (int i = 0; i < n; i++)
```

```
37            {
38              if (inc[i] & 1)
39              {
40                boo = 0;
41                break;
42              }
43            }
44            if (boo == 0)
45            {
46              printf("0\n");
47              continue;
48            }
49          //-------------------------
50            memset(f, 0 ,sizeof(f));
51            f[0][0] = 1;
52            for (int i = 1; i <= m; i++)
53            {
54              for (int t = 0; t < (1 << (2 * n)); t++)
55              {
56                int tmp = f[i - 1][t];
57                f[i][t] += tmp; // 第条边维持白色i
58
59                int u = edge[i][0], v = edge[i][1]; // ，是第条边连着的两个点uvi
60                int tu = (t >> (u * 2)) & 3, tv = (t >> (v * 2)) & 3; // tv 和 tu 是，两点的状态uv
61
62                if (tu + 1 > inc[u] / 2) continue;
63                if (tv + 1 > inc[v] / 2) continue; // 如果染色以后会过半，不用染黑了
64
65                int su = tu + 1, sv = tv + 1;
66
67                int s = t ^ (tu << (u * 2)) ^ (tv << (v * 2));  //先把和的状态变成uv0
68                s = s | (su << (u * 2)) ^ (sv << (v * 2));  //把和的状态变成和uvsusv
69
70                f[i][s] += tmp;
71              }
72            }
73            int s = 0;
74            for (int i = n-1; i >= 0; i--)
75            {
76              s <<= 2;
77              s |= inc[i] / 2;
78            }
79
80            printf("%d\n", f[m][s]);
81          }
82          return 0;
83        }
```

2.6 数位dp

数位dp写成记忆化搜索之后就是几乎模板化的了。。f[i][……]表示i位数balabala的情况下的个数……然后记忆化搜索的写法就可以解决转移和统计的问题……[10]

[10]数位dp由于有模板……理解模板以后套用就是了，我理解模板的方式是阅读代码和给代码写注释

2.6.1 数位dp例题1；HDU 3555 Bomb

题目大意：

求1到n中含有49的数有多少

思路：

状态：f[i][0] i位的数字没有49；

f[i][1] i位的数字没有49 但最高位是4

f[i][2] i位的数字有49

转移方程：

f[i][0] = 9×(f[i - 1][0]) - f[i - 1][1];

f[i][1] = f[i - 1][0];

f[i][2] = f[i - 1][2]×10+ f[i - 1][1] + f[i - 1][0];

边界：

f[0][0] = 1;

以前我写题解的时候说：数位dp的难度不在于转移，要算出一个整百整千整万的数包含多少个是很容易的（转移很简单），但是我们给的数都是形如“abcdef”这种数……零头的计算往往包含很多细节，而记忆化搜索很好的解决这个问题!!

代码：

```
#include <iostream>
#include <string>
#include <cstdio>
#include <cstring>
using namespace std;
int bit[40];
long long f[40][4];
long long dp(int pos, int st, bool flag)
//表示所在位；pos 为状态，st: st=0 没有49, st=1 前一位为4 ,st=2 已经出现过49；表示高位是否是与原数不同的flag
{
  if (pos == 0) return st == 2;
  //所有位都已扩展完，如果st==2 返回1，否则返回0
  if (flag && f[pos][st] != - 1) return f[pos][st];
  //如果高位与原数不同且已经算过，直接返回
  long long ans = 0;
  int x = flag ? 9 : bit[pos];
  //维护当前位最多是多少
  for (int i = 0; i <= x; i++)
  {
    if ((st == 2) || (st == 1 && i == 9))
    //如果已经出现49 或者上一位为4，当前位为9
      ans += dp(pos - 1, 2, flag || i < x);
      //算从当前状态扩展过去的下一位
      else if (i == 4) ans += dp(pos - 1, 1, flag || i <x);
      //当前位是，将向下扩展的4改成st1
      else ans += dp(pos - 1, 0, flag || i < x);
      //维持st0
  }
  if (flag) f[pos][st] = ans;
  //记忆化
```

```
31          return ans;
32        }
33        long long calc(long long x)
34        {
35          int len = 0;
36          while (x)
37          {
38            bit[++len] = x % 10;
39            x = x / 10;
40          }
41          //拆位
42          dp(len, 0, 0);
43          //记忆化搜索进行计算
44        }
45        int main()
46        {
47          int t;
48          scanf("%d", &t);
49          memset(f, -1, sizeof(f));
50          for (int i = 1; i <= t; i++)
51          {
52
53            long long  n;
54            cin >> n;
55            cout << calc(n) << endl;
56            //算小于的符合条件的数n
57          }
58          return 0;
59        }
```

2.6.2 数位dp例题2；HDU 3652 B-number

题目大意：

求1到n中含有13且能够被13整除的数有多少个……

思路：

关于一个数含不含13，方法与上一个题类似，但是可以整除13怎么办？利用同余把除以13的余数也计入状态，int dp(int pos, int mod, int st, int flag) 其中mod就是当前数除以13的余数（数组也相应的多了一维）

记忆化搜索的代码如下：

```
1         int dp(int pos, int mod, int st, int flag)
2         {
3           if (pos == 0) return (st == 2 && mod == 0);
4           if (flag && f[pos][mod][st] != -1) return f[pos][mod][st];
5           int ans = 0;
6           int x = flag ? 9 : bit[pos];
7           for (int i = 0; i <= x; i++)
8           {
9             int tmp = (mod * 10 + i) % 13;
10            if ((st == 2) || (st == 1 && i == 3))
11              ans += dp(pos - 1, tmp, 2, flag || i < x);
```

```
            else if (i == 1) ans += dp(pos - 1, tmp, 1, flag || i <x);
            else ans += dp(pos - 1, tmp, 0, flag || i < x);
        }
        if (flag) f[pos][mod][st] = ans;
        return ans;
    }
```

2.6.3 数位dp例题3；HDU 4389 X mod f(x)

题目大意：

计算区间内一个数字各位之和能整除该数字的个数

思路：

跟上一个题还是很像……由于f（x）最大就是81，所以可以算对于1 - 81每一个数都求一下就可以了，f[pos][mod][x][sum] 表示前pos位数，除以x的余数是mod，各个位数和为sum的数的个数

f[pos][mod][x][sum] += f[pos - 1], [(mod×10 + i) % x], [x], [sum + i];

记忆化搜索的代码如下：

```
    int dp(int pos, int mod, int x, int sum, bool flag)
    {
        if (pos == 0) return (x == sum && mod % sum == 0);
        if(flag && f[pos][mod][x][sum] != -1)return f[pos][mod][x][sum];
        int re = 0;
        int d = flag ? 9 : bit[pos];

        for (int i = 0; i <= d; i++)
        {
            int tmp = (mod * 10 + i) % x;
            re += dp(pos - 1, tmp , x, sum + i ,flag || i < d);
        }
        if (flag) f[pos][mod][x][sum] = re;
        return re;
    }
```

2.6.4 数位dp例题4；HDU 3709 Balanced Number

题目大意：

找出区间内平衡数的个数，所谓的平衡数，就是以这个数字的某一位为支点，另外两边的数字大小乘以力矩之和相等，即为平衡数

思路：

枚举支点，算力矩, dp[pos][x][st] 表示pos位的数，支点是第x位，目前的的力矩为st的数的个数

转移为：dp[pos][x][st] += dp[pos - 1][x][st + i×(pos - x)]

代码如下：

```
    #include <iostream>
    #include <cstdio>
    #include <cstring>
    using namespace std;
```

```
5         long long a, b;
6         long long f[30][30][2005];
7         int bit[30];
8         long long dp(int pos, int x, int st, int flag)   //是力矩st
9         {
10          if (pos == 0) return st == 0;
11          if (st < 0) return 0;
12          if (flag && f[pos][x][st] != -1) return f[pos][x][st];
13          int d = flag ? 9 : bit[pos];
14          long long ans = 0;
15          for (int i = 0; i <= d; i++)
16          {
17            int tmp = st + i * (pos - x);
18            ans += dp(pos - 1, x, tmp, flag || i != d);
19          }
20          if (flag) f[pos][x][st] = ans;
21          return ans;
22        }
23        long long solve(long long n)
24        {
25          if (n < 0) return 0;
26          int len = 0;
27          while (n > 0)
28          {
29            bit[++len] = n % 10;
30            n /= 10;
31          }
32          long long res = 0;
33          for (int i = 1; i <= len; i++)
34          {
35            res += dp(len, i, 0, 0);
36          }
37          return res - len + 1;
38        }
39        int main()
40        {
41          int t;
42          scanf("%d", &t);
43          memset(f, -1, sizeof(f));
44          while (t--)
45          {
46            cin >> a >> b;
47            cout << solve(b) - solve(a - 1) << endl;
48          }
49          return 0;
50        }
```

掌握了记忆化搜索的写法数位dp是不是就变得简单了呢？

一边转移一边统计简直是很美好啊!! O(∩_∩)O

3 dp优化

常见的优化主要作用于几个方面：

1.**方程上的优化**（例如去除冗余维度，进行降维）

2.**状态总数上的优化**（相当于搜索的各种剪枝，去除无效状态，比如各种基于贪心的DP （在下一章节详述），利用map、链表之类的数据结构优化，）

3.**每个状态转移的状态数的优化**（依靠决策的单调性优化【斜率优化】、数学优化【四边形不等式优化】）

4.**每次状态转移时间的优化**（减少递推时间【预处理】、减少枚举时间【Hash表、单调队列、线段树】、针对线性递推的优化【矩乘+快速幂】）

表示本章节的大部分东西本弱都不会orz……学会了我会补上的……

3.1 降维

dp能够降维当然就是最棒的优化了，将几个相关的状态合并，改变状态设计，或者预处理一些量都是可以取到降维的效果，但涵盖太广本弱没有能力详细分析，所以就写两个例题一起来感受一下……

3.1.1 dp降维例题1：hdu1024 多个不相交子段和问题

题目大意：

给定一个长度为n的数组，求将其分成m个不相交子段和最大值的问题

输入文件

每组数据先n和m，然后是n个数($n \leq 1000000$)

输出文件

最大和

输入样例

1 3 1 2 3

2 6 -1 4 -2 3 -2 3

输出样例

6

8

思路：

显而易见的，f[i][j]表示前i个数(第i个数必须取)组成j个不相交子段所能取得的最大和，同样显而易见的，$f[i][j] = max(f[i-1][j] + a[i], f[k][j-1] + a[i]) \quad 0 < k < i$

由于n和m的范围，首先我们想到的是得用滚动数组，然而本转移方程的复杂度是$O(n^2 \times m)$，所以必须得优化。

首先，我们来考虑本题的拓扑序——一般而言，我是采用外层按照i从小到大，内层按照j从小到大的顺序来的，但是事实上，外层j内层i也是可以的!

如果我们把j作为外层循环再来看这个方程：f[k][j-1]的最大值不是可以在上一层循环的时候顺便预处理出来么？

这样的话时间复杂度不就变成$O(N^2)$了!

代码:

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <cmath>
#include <algorithm>
#include <queue>
using namespace std;
#define N 1000100
int f[N], maxn[N], a[N];
int n, m;
int main()
{
  while (scanf("%d%d", &m, &n) != EOF)
  {
    for (int i = 1; i <= n; i++)
    {
      scanf("%d", &a[i]);
    }
    memset(f, 0, sizeof(f));
    memset(maxn, 0, sizeof(maxn));
    int tmp;
    for (int j = 1; j <= m; j++)
    {
      tmp = -0x7f7f7f7f;
      for (int i = j; i <= n; i++)
      {
        f[i] = max(f[i - 1] + a[i], maxn[i - 1] + a[i]);
        maxn[i - 1] = tmp;
        tmp = max(tmp, f[i]);
      }
    }
    printf("%d\n", tmp);
  }
  return 0;
}
```

3.1.2 dp降维例题2:方格取数

题目:

设有N*N的方格图($N \leq 10$,我们将其中的某些方格中填入正整数,而其他的方格中则放入数字0。

某人从图的左上角的A 点出发,可以向下行走,也可以向右走,直到到达右下角的B点。在走过的路上,他可以取走方格中的数(取走后的方格中将变为数字0)。

此人从A点到B 点共走两次,试找出2条这样的路径,使得取得的数之和为最大。

输入文件

输入的第一行为一个整数N(表示N×N的方格图),接下来的每行有三个整数,前两个表示位置,第三个数为该位置上所放的数。一行单独的0表示输入结束。

输出文件

只需输出一个整数，表示2条路径上取得的最大的和。

输入样例

8
2 3 13
2 6 6
3 5 7
4 4 14
5 2 21
5 6 4
6 3 15
7 2 14
0 0 0

输出样例

67

思路:

这道题是典型的**双线程动态规划**，简单地说就是将**两条路同时算进状态里**

F[i][j][k][l]表示两个人分别走到i行j列和k行l列时所能取得最大值

如果((i == k) 且(j ==l))

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];

否则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];

但是如果我们注意到这一点——两**条路的长度得一样!** 所以可以少一维——

f[i][x][y]第i个阶段，两条路分别走到横坐标为x和y的点的最大值

每步走一格，所以可以通过i求出纵坐标!!，然后就少一维啰!!!

3.1.3 dp降维例题3：看樱花

题目：

“妹妹背着洋娃娃，走到花园看樱花”。

已知：这是一个$1 \times N$ 的花园（虽然比较奇怪），被分成了N个格子，每个格子里有一种神奇的樱花（我也不知道为什么神奇，反正洋娃娃看着高兴），看到第i个格子上的花洋娃娃会得到不同的满足度Ci（每个花的满足度只被计算一次）。现在妹妹会背着洋娃娃从任意格子走进花园，当然从第i个格子进去会消耗Di个单位的满足度，然后游历花园，在一个格子向右走需要耗费R个单位的满足度，向左走需要耗费L个单位的满足度，最后从第i个格子出花园又要耗费Fi个单位的满足度。

接下来，我们需要设计一套游历方案，使得最终获得的总满足度最高（太低的话洋娃娃会……）

输入格式

第一行依次给出三个正整数N，L，R。

第二行有N个非负整数，第i个数为Di。

第三行有N个非负整数，第i个数为Fi。

第四行有N个非负整数，第i个数为Ci。

输出格式

仅需要输出一行包括一个整数，表示最大获得的满足度为多少。

数据规模

对于20分的数据，N小于等于10。

对于40分的数据，N小于等于100。

对于80分的数据，N小于等于1000。

对于100分的数据，N小于等于100000。输入数据里的所有数在int范围内。

输入样例

5 1 1

1 1 1 1 1

1 1 1 1 1

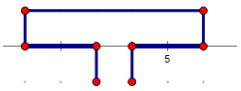
1 1 3 1 1

输出样例

1

思路：

对于所有有可能的行走方法，可以概括成一种——设进入点在出口点的左边，一定是先向左走，然后回头向右走，经过出口点再走了一段后，再回头向左走走到出口点。进入点在右边的情况类似。如图(从左向右或者反过来)：(图丑勿喷……)



只要出入口确定，那么左右需要到达的范围其实也确定了——对于每个点都可以**预处理**出l[i],r[i]代表向左最多的得分和向右最多的得分，于是很明显：

dp[i][j]表示出口和入口为i，j（顺序不限制，保证i小于j）

dp[i][j]=max(l[i]+r[j]+(sum[j-1]-sum[i])-(j-i)×rr-d[i]-f[j])（$i \le j$）**左进右出**

dp[i][j]=max(l[i]+r[j]+(sum[j-1]-sum[i])-(j-i)×ll-d[j]-f[i])（$i \le j$）**右进左出**

但是这样是是O(n^2)的过不完，于是我们看看上面的式子，可以化简成这样：(以左进右出为例)

ans = max（l[i]-sum[i]+i×rr-d[i]）+max(r[j]+sum[j-1]-j×rr-f[j])

于是问题成功变为O（n）的有木有!!

如果你没有看懂，我就多说两句!

用g1[i]表示的是max（l[k]-sum[K]+K×rr-d[K]）（$k \le i$）

用g2[i]表示的是max（r[j]+sum[j-1]-j×rr-f[j]）（$k \le i$）

然后就可以在所有g1[i] + g2[i]中找一个最大的作为答案就可以了……

思考：

这个题降维的切入点在哪？？

f[i][j]的转移方程中，同i和j相关各个**变量可以完全分离**，且**形式相似**……而分开求最大值以后，可以发现，跟那个求两个最大区间和的是不是很相似？于是不用同时枚举入口和出口，只枚举左边的一个就可以了……

多想一想世界就大为不同了是吧？

3.2 线段树优化dp

3.2.1 线段树优化dp例题：HDU 4719 Oh My Holy FFF

题目大意：

给你一个长度为n的序列A = ai，需要将其分解成若干个子序列，要求每个序列长度不超过L，且各个序列最后一个元素单调递增，对于每一个划分方案，如果我们把每一个子序列的最后一个数记为b[i]，那么这个划分方案的得分是$\Sigma(b[i]^2 - b[i-1])$，求对于一个序列的合法划分所能得到的最大的得分值

Sample Input

2

5 2

1 4 3 2 5

5 1

5 4 3 2 1

Sample Output

Case #1: 31

Case #2: No solution

思路：

其实这个题普通的转移是很简单的——f[i]表示第i个人作为当前子序列的结尾所能得到的最大得分，显而易见的$f[i] = max(f[j] + a[i]^2 - a[j])$, 其中$a[j] < a[j]$ 且$j < i$

但是这样是$O(n^2)$的时间复杂度，要跪的，需要优化!

——通过观察方程，我们可以**分离变量**从而注意到f[i] 取决于两个值——$a[i]^2$ 和$f[j] - a[j]$其中$a[i]^2$由i本身决定，无法改变，而f[j] - a[j] 的值则应该取尽量大的……这样似乎是可以用线段树……

但是显而易见，我们直接在i前面长度为L的区间里找最大的f[j] - a[j] 是不行的，因为有可能找到的那个a[j]不小于a[i]……为了保证这一点，我们可以这样——**按照a值从小到大dp**，求出f值以后将f[i]-a[i]放入线段树内（没求出的初值赋为-1等无意义的特殊值），然后每次在i的前了L长去找线段树的最大值就可以了。

在一个区间取最值我们都能很愉快的想到线段树，但是当除了最值还有其他限制条件的时候怎么办？可以用这个限制条件去约束拓扑序，从而使不能转移来的点后计算，从而使得转移直接符合了约束条件（想办法把判断去掉……）

3.2.2 由上题引发的对于动态规划拓扑序的一点思考

之前从来没有考虑过这个问题，一直认为拓扑序只是对于选择dp的写法有影响（递推还是记忆化搜索），然而事实上不是这样的，这个题简直是给人当头一棒!

求最长上升子序列的拓扑序究竟是什么？

确切的说，其实不是期末一般认为的从左到右的，而是从满足$i < j$ 且$a[i] < a[j]$ 的i转移了j,从左到右事实上只是我们**递推的顺序**而已，而这个递推顺序是可以改变的……

从而我似乎得出了最长上升子序列的线段树优化的做法，orz……（虽然并没有什么用……）

总而言之：动态规划的拓扑序，跟拓扑排序的结果一样，其实是多种多样的，先保证满足哪一个条件，判断哪一个条件，其实是很灵活的事情，当寻找不到好的方法的时候，换个角度想问题往往更有效。

最后，再说明刚刚那个例题为什么要这么做——有两个条件需要满足:$j-L \le i < j$ 且$a[i] < a[j]$，在算法过程中，其实线段树的存在就已经足够保证$j-L \le i < j$，那么$a[i] < a[j]$ 用计算的时候的顺序保证就好了（如果我们依然按照从左到右的顺序，岂不是计算的顺序就显得很多余么……）

3.3 指针优化dp

首先要说明的是，这里的“指针”是一种广义的概念，可以用数组啊之类的东西代替，用以起到指向某一处的作用，那么这种指针怎么优化dp呢？我们从今年我校区域赛的网赛里的一道dp说起

3.3.1 指针优化dp例题：HDU 5009 Paint Pearls

题目大意：

给定一系列的颜色，可以划分为任意多个任意大小的区间，每个区间的花费为区间颜色数的平方，问你总花费最小是多少？

Sample Input

3

1 3 3

10

3 4 2 4 4 2 4 3 2 2

Sample Output

2

7

思路：

网赛的时候我高中同学拿常数优化把它过了，简直orz这数据啊……这个题比较靠谱的办法是用双向链表（其实就是指针）优化。

既然是优化嘛，肯定先说朴素的方法……

f[i]表示涂完前i个所化的最小代价，显然有f[i] = mindp[j]+$num(j+1,i)^2$, 其中$1 \le j$ ¡ i, num(j+1,i)表示区间[j+1,i]的颜色个数。

这个是一个O(n^2)的方法，显然是要超时的啊!

我们来观察一下第二组样例：3 4 2 4 4 2 4 3 2 2，假设现在f[1]到f[8]已经算好了，现在要计算f[9]，可以看到a[9]为2，按照方程f[i]=mindp[j]+$num(j+1,i)^2$, 枚举j=8，j=7……到了j=6的时候我们发现，可以直接变成j = 0，为什么我这么说？

因为a[6], a[7],a[8]已经包含了3个数，而之前只包含这三个数了，那么把这三个数放在一起肯定不会比把从头开始的所有数放在一起差，也就是说，**j是可以跳着枚举的**，如果我们能够把**怎么跳**处理出来，不就大大降低了复杂度了么？

换句话说如果当前计算第i位，那么即可以将a[i]加进来，而之前如果有和a[i]值相等的点就必须删掉（不用再遍历这个点（用指针跳过他），因为他不会再影响不同的数的个数），使用双向链表（其实是两个数组）维护，保证任意时刻，**每种颜色只会保存一次**，复杂度就降下来了。

但仍然可以给出卡这个方的数据——每一个数都不一样，这种时候复杂度仍为O（n^2），于是继续优化。我们知道如果一个一个涂，那么需要花费n。所以**最优方案不可能大于n**，也就是不能连着$\sqrt{n}$个不同的颜色一起涂，否则代价大于n了，这里进行剪枝[11]。于是复杂度降为O（n $\sqrt{n}$）,一个两个这样的数据还是可以接受的。

你问我怎么写？让我们看代码!!

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <map>
using namespace std;
#define N 50010
map <int, int> mp;
int n, a[N], pre[N], next[N], f[N];
int main()
{
  while(scanf("%d", &n) != EOF)
  {
    for (int i = 1; i <= n; i++)
    {
      scanf("%d", &a[i]);
      pre[i] = i - 1;    //初始化：前驱链表指向前一个
      next[i] = i + 1;   //初始化：后继链表指向后一个
    }
    mp.clear();            //由于元素的值很大，所以映射用以离散化map
    memset(f, 127, sizeof(f));
    f[0] = 0;
    pre[0] = -1;
    for(int i = 1; i <= n; i++)
    {
      if(mp.count(a[i]) == 0) mp[a[i]] = i;  //当前这个数没出息过， 把它的位置记录下来就行
      else{                                  //当前数之前就出现过
        int tmp = mp[a[i]];                 //记下这个数上一次出现的位置
        next[pre[tmp]] = next[tmp]; //的前驱的后继（原来是指向的）指向的后继（把跳过了）
            tmptmptmptmp
        pre[next[tmp]] = pre[tmp];  //的后继的前驱（原来是指向的）指向的前驱（把跳过了）
            tmptmptmptmp
        mp[a[i]] = i;               //把a[i出现的位置改为当前的位置]
      }
      int cnt = 0;
      for(int j = pre[i]; j != -1;j = pre[j]) //往前面扫描
      {
        cnt++;
        f[i] = min(f[i], f[j] + cnt * cnt); //转移dp
        if(cnt * cnt > i) break;            // 最多sqrt(n)的那个优化
      }
    }
    printf("%d\n",f[n]);
  }

  return 0;
}
```

[11]这里其实是一个常数优化

我们写这个题的时候要怎么想？个人觉得，首先我们必须明确指针的指向是用来干嘛的，我们为什么要使用指针链表？于本题而言，是为了**跳过后面出现过了的的数**。然后就问怎么跳过？？让他的前驱指向他的后继……所以我们维护了一个前驱指针和一个后继指针……于是我们需要每一次把最近的a[i]的位置记下来，以便于a[i]出现的时候把前一个跳过去!!

网赛的时候一直在想如何维护出任意一个区间的不同元素的个数，但是实际上这个维护时是做了不少的无用功的——每次转移的时候只需要知道**当前点往前有多少个不同元素就是了**，而且很明显的，从i往前的元素个数随着区间长度的增长而增加，而且出现的是类似于1222333334的情况，这种情况下，**每个个数只有最远的那个是有效的**，所以只要能够从1的尾部跳到2的尾部就成功了，而这个东西正好是可以通过上一次的最后一个点的结果转移过来，这才是关键所在啊。

3.4 map优化dp

用map来优化dp，一般是用于**表面上看起来状态很多但是实际上真正的状态并不多，或者状态不是连续的而是离散的**的情况……这个时候用map把状态映射离散化就可以解决tle或者mle的问题了……

3.4.1 map优化dp例题1：HDU 4028 The time of a day

题目大意：

给出1-N这N个数，问有多少个子集，集合各个数的里的lcm是大于等于m的

Sample Input

3

5 5 (N = 5, M = 5)

10 1

10 128

Sample Output

Case #1: 22

Case #2: 1023

Case #3: 586

思路:

设f[i][j],表示1 i的数组成的最小公倍数为j的个数，

转移：**f[i + 1][LCM(j, i+1)] = f[i][LCM(j, i+1)] + f[i][j]**

由于j实在是太大了，而且**j不是连续的**，所以用了map……first 存j的值，second存f值（个数）……

状态转移部分的代码：

```
map<long long,long long> f[45];
f[1][1] =1;
map<long long, long long>::iterator it;
for(int i = 2; i <= 40;i++)
{
  f[i] = f[i - 1];
  f[i][i]++;
  for(it = f[i - 1].begin(); it != f[i - 1].end(); it++)
  {
    f[i][lcm(it->first,i)] += it->second;
  }
```

```
12          }
```

3.4.2 map优化dp例题2: Codeforces 512B Fox And Jumping

题目大意:

给出N个不同的数，取第i个数的代价为c[i]，求取出若干个数使得其最大公约数为1的最小代价

input

3

100 99 9900

1 1 1

7

15015 10010 6006 4290 2730 2310 1

1 1 1 1 1 1 10

8

4264 4921 6321 6984 2316 8432 6120 1026

4264 4921 6321 6984 2316 8432 6120 1026

output

2

6

7237

思路:

转移方程显而易见：**f[gcd(a,b)] = min(f[gcd(a,b)],f[a]+cost(b))**

还是那个问题，原题数据范围很大而gcd的值不连续……

所以gcd拿map存啰……

部分代码:

```
1           for(int i = 0; i < n; i++)
2           {
3             map<int,int>::iterator it;
4             for(it = dp.begin(); it != dp.end(); it++)
5             {
6               int t = gcd(l[i], it->first);
7               if(dp.count(t)!= 0) dp[t]  = min(dp[t], it->second + c[i]);
8               else dp[t] = it->second + c[i];
9             }
10          }
```

这个题是老板教我的，orz，做了这个题以后就会用map优化dp了，真开心!!

map对于dp的优化实质上是将连续的状态**离散化!**

3.5 常数优化dp

表示常数优化dp这种事情是oi比较容易遇到，但是我最近还真的碰到常数优化的题，严格的来讲，其实不能说是常数优化，而是通过分析得出**某一个量只能是在某一个范围内的。**

让我们来看题目!!!

3.5.1 常数优化例题1：Codeforces 506A Mr. Kitayuta, the Treasure Hunter

题目大意：

一共有30001个岛屿，下标为0到30000，有些岛屿藏有宝藏。你从0往下标大的方向跳，第一步跳的距离为d。如果上一步跳的距离为l，这一步就可以跳l-1或l或l+1（距离必须大于0）。问最多拿到多少宝藏。

Input

4 10

10

21

27

27

Output

3

思路：

设f[i][j]表示到达第i个岛屿，跳的距离为j 得到的最大宝藏数，那么转移方程为：

f[i+j][j] = max(f[i+j][j], f[i][j] + c[i+j])

f[i+j+1][j+1] = max(f[i+j+1][j+1], f[i][j] + c[i+j+1])

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

如果开一个f[30000][30000]的数组的话会MLE(也会tle貌似)，这个时候我们需要多想一想。

由于**最大步长是1+2+3+...a+250 ¿ 30000**。所以**第二维最多在第一次的d的基础上上下浮动250（其实没那么多）**，于是**第二维开500**就够了。第二维的值x=250表示和d相等，(x - 250) + d 才是当前跳跃的距离orz!!

代码：

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <string>
#include <algorithm>
#include <map>
using namespace std;
int a[30010];
int f[30010][510];
int main()
{
  int n, d;
  scanf("%d%d", &n, &d);
  for (int i = 0; i < n; i++)
  {
    int t;
    scanf("%d", &t);
    a[t]++;
  }
```

```
21            memset(f, - 1, sizeof(f));
22            f[d][250] = a[d];
23            int re = 0;
24            for (int i = d; i <= 30000; i++)
25            {
26              for (int j = 0 ;j <= 510; j++)
27              {
28                re = max(re, f[i][j]);
29                if (f[i][j] == -1) continue;
30                int len = d - 250 + j;
31                if (i + len <= 30000)
32                  f[i + len][j] = max(f[i + len][j] , f[i][j] + a[i + len]);
33                if (i + len + 1 <= 30000)
34                  f[i + len + 1][j + 1] = max(f[i + len + 1][j + 1] , f[i][j] + a[i + len + 1]);
35                if (i + len - 1 <= 30000 && len - 1 > 0)
36                  f[i + len - 1][j - 1] = max(f[i + len - 1][j - 1] , f[i][j] + a[i + len - 1]);
37
38              }
39            }
40            printf("%d\n", re);
41
42            return 0;
43          }
```

3.5.2 常数优化例题2：NOIP 2005 过河

题目：

在河上有一座独木桥，一只青蛙想沿着独木桥从河的一侧跳到另一侧。在桥上有一些石子，青蛙很讨厌踩在这些石子上。由于桥的长度和青蛙一次跳过的距离都是正整数，我们可以把独木桥上青蛙可能到达的点看成数轴上的一串整点：0，1，……，L（其中L是桥的长度）。坐标为0的点表示桥的起点，坐标为L的点表示桥的终点。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。

题目给出独木桥的长度L，青蛙跳跃的距离范围S,T，桥上石子的位置。你的任务是确定青蛙要想过河，最少需要踩到的石子数。

输入文件

输入的第一行有一个正整数L（$1 \leq L \leq 10^9$），表示独木桥的长度。第二行有三个正整数S，T，M，分别表示青蛙一次跳跃的最小距离，最大距离，及桥上石子的个数，其中$1 \leq S \leq T \leq 10$，$1 \leq M \leq 100$。第三行有M个不同的正整数分别表示这M个石子在数轴上的位置（数据保证桥的起点和终点处没有石子）。所有相邻的整数之间用一个空格隔开。

输出文件

输出只包括一个整数，表示青蛙过河最少需要踩到的石子数。

样例输入

10

2 3 5

2 3 5 6 7

样例输出

2

数据规模

对于30%的数据，$L \leq 10000$；

对于全部的数据，$L \leq 10^9$。

思路：

显然，对于L很小的数据来说，不难写出动态规划的状态转移方程

f[i]表示走到位置i时踩到的最少石子数

a[i]表示位置i的石头数

f[i]=minf[i-j] + a[i] ($0 \leq j$ 且j属于[i-t,i-s])

由于数据较大，直接动规只能过三组，所以需要优化

单独处理s == t的情况（不要问我怎么处理，太简单不想说）

由于石子数实在是很少（和桥的长度比起来），所以两个石子之间的距离可能会很大很大，显而易见的，**当两个石头之间的距离超过一定大小时，那个点就一定能到达的**，那么我们就可以把大于这个距离的压缩成这个距离了!

但是这个距离是多少呢？

显然如果s == 1那么1就可以了，也就是说，如果跳跃的距离比较小，那么肯定这个长度也比较小，换而言之，当跳跃距离只能是9和10的时候这个最短的长度最大。这个长度是多了呢？？

90!!

90 到99 是不是都可以组成的啊？那么大于99的肯定也都可以啦……

于是，当两个石子之间的距离大于90，就把期距离改成90就行了……

桥的长度大大减小，问题顺利解决!

思考：

表示作为这两个题来说，所谓的常数优化，实际上都是**通过分析直接缩小了枚举的范围，提前将不可能出现的状态排除在外**，当然，例1也可以用map来优化……都是So easy 不是么？

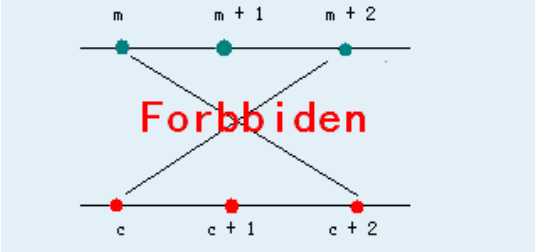
3.6 二分优化dp

表示以前一直记得最长上升子序列是单调队列优化的dp，结果做了才发现不是这样的orz，真是too young……

3.6.1 二分优化dp例题：HDU 1025 Constructing Roads In JGShining's Kingdom

题目大意：

有两条平行线，每条线上有n（$n \leq 500000$）个点，两条线上的点一一对应，将对应的点相连，问最多能选多少个点对使得连线不想交!



样例输入：

2

1 2

2 1

3

1 2

2 3

3 1

样例输出：

Case 1:

My king, at most 1 road can be built.

Case 2:

My king, at most 2 roads can be built.

思路：

表示按照第一个量排序以后对第二个量求最长上升子序列就是答案!!!

但是这个题的问题在于，n 最大为500000，O(n^2)的求法会TLE，咋办呢？

让**f[i]表示长度为i的最长上升子序列最小的最后一位数是a[i]**，我们可以保证f[i]是递增的，然后求哪一个是a[x]的前一个的时候就可以**二分**了……

表示一直以为最长上升子序列是单调队列优化的……真是弱得无语……

代码：

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <string>
#include <algorithm>
#include <cmath>
#include <vector>
#include <queue>
using namespace std;
const int N = 500010;
int a[N];
int f[N];
int n;
int main()
{
  int cas = 0;
  while(scanf("%d", &n) != EOF)
```

```
19              {
20                for(int i = 1; i <= n; i++)
21                {
22                  int x, y;
23                  scanf("%d%d", &x, &y);
24                  a[x] = y;
25                }
26                f[1] = a[1];
27                int tot = 1;
28                for(int i = 2; i <= n; i++)
29                {
30                  int l = 1, r = tot;
31                  while(l <= r)
32                  {
33                    int mid = (l + r) / 2;
34                    if(f[mid] > a[i]) r = mid - 1;
35                    else l = mid + 1;
36                  }
37                  f[l] = a[i];
38                  if(l > tot) tot++;
39                }
40                if(tot == 1)
41                  printf("Case %d:\nMy king, at most %d road can be built.\n\n", ++cas, tot);
42                else printf("Case %d:\nMy king, at most %d roads can be built.\n\n",++cas, tot);
43              }
44
45            }
```

关于dp优化，有一个超级经典的题——URAL 1223 鹰蛋……于本弱这篇题解的篇幅已经很恶心了而这个题的论述实在是很长很长，也由于我无论怎么写也不如专门论述这个题的国家集训队论文，在本章就不讨论这个题，只是给出国家集训队论文《优化，再优化!——从《鹰蛋》一题浅析对动态规划算法的优化》的网址链接，强烈推荐大家去看!! 这个题涵盖了好几种dp优化的方法!!

论文链接：http://wenku.baidu.com/view/e3935d0216fc700abb68fcac.html

论文配套ppt链接：http://wenku.baidu.com/view/7d57940ef12d2af90242e6ac.html

4 一些dp趣题

4.1 求反方面的dp

通过dp求目标的**反面**，然后用总数一减就好了，orzzzzz!!!

4.1.1 求反方面dp例题1：无聊的数列

题目描述：

有一个不下降序列$a1 \leq a2 \leq \& \leq an$。bi定义为数列中有多少个数严格小于ai，ci定义为数列中有多少个数严格大于ai。

你的秘书把n个数对(bi, ci) — $1 \leq i \leq n$打乱顺序抄写在一张纸上。由于她很粗心，致使不少数对被抄错了。

你的任务是：改动最少的数对，使得存在与这些数对对应的原数列ai —$1 \leq i \leq n$。

输入格式

第一行n($n \leq 1000$),表示n个数对。

接下来n行，每行两个数b,c。意思和题目描述一样。

输出格式

一行，表示要改动的最少数列。

输入样例

5

0 2

0 3

2 1

1 2

4 0

输出样例

2

思路：

这其实是个比较简单的题……这个题直接求需要改动的有多少比较麻烦，那么我们可以去求他的反面——区间内符合要求的有多少个!

根据题目可知，如果在数列中严格小于某数的有b个，严格大于它的有c个，那么，这个数的信息如果没有问题，它必然在(b,n-c]这个区间内。

于是我们可以预处理出一个a[i][j]，表示区间(i,j]内的数有多少个。当然，a[i][j] 的值可能大于j-i，这时应将其控制为j-i。

然后用f[i]表示前i个数中符合的数对共有多少个。我们枚举j，并设[j+1,i]这个区间里的所有数都是相等的，那么这个区间里符合规定的数共有f[j]+min(a[j][i],j-i)。

最后非法的有n-f[n]个。

4.2 需要消除后效性的dp

无后效性是能够使用dp解题的必要性质，如果状态不满足无后效性就不能使用动态规划，因此，找到一个满足无后效性的方程式很重要的，如何消除方程无后效性也就成为的一个需要好好总结的点!

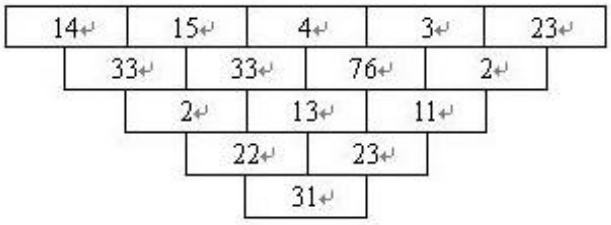
首先，**加一维**状态是最容易想到的；其次，**设计一个状压的状态**也有可能能够消除转移的后效性；然后，通过**预处理**，多重dp也是可以消除无后效性……当然还有很多很多的技巧还是我不会的，在此只对我知道的好题进行一个分析!

4.2.1 思维转个圈，消除后效性例题：hnoi 打砖块

题目描述：

一个凹槽中放置了n层砖块，最上面的一层有n块砖，第二层有n-1块，……最下面一层仅有一块砖。第i层的砖块从左至右编号为1，2，……i，第i层的第j块砖有一个价值a[i][j]（$a[i][j] \leq 50$）。下面是一个有5层砖块的例子：如果你要敲掉第i层的第j块砖的话，若i=1，你可以直接敲掉它，若$i > 1$，则你必须

先敲掉第i-1层的第j和第j+1块砖。



你的任务是从一个有n（$n \leq 50$）层的砖块堆中，敲掉($m \leq 500$)块砖，使得被敲掉的这些砖块的价值总和最大。

输入格式

你将从文件中读入数据，数据的第一行为两个正整数，分别表示n,m，接下来的第i每行有n-i+1个数据，分别表示a[i][1],a[i][2]……a[i][n –i +1]。

输出格式

第1行：表示被敲掉砖块的最大值总和

输入样例

4 5

2 2 3 4

8 2 7

2 3

49

输出样例

19

数据规模

对于20 %的数据，满足$1 \leq N \leq 10$,$1 \leq m \leq 30$;

对于100 %的数据，满足$1 \leq N \leq 50$,$1 \leq m \leq 500$;

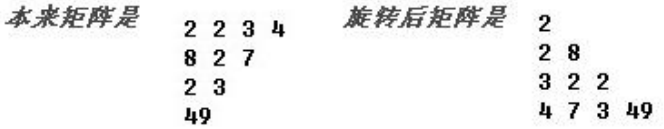
思路：

看起来和数字三角形有异曲同工之妙！那么我们先设计个状态来看：f[i][j][q]为取第i行第j个共选了q个的最大值，那么f[i][j][q] = f[i-1,j,q]+f[i-1,j+1,p-q-1]+a[i,j],f[i,j,p]，可是这个真的对吗？？？？

仔细看图你发现这样的转移是不满足无后效性的。。。换句话说它并不能一行一行的转移……

让我们来认真看看样例！从下往上打如果行不通换一个方向呢？

来看图：



这样就是**从左向右连续的**打了!!而此时只要保证每行所打到的最后的斜前方（i-1, j-1）已经被打过了就可以满足题目的要求!

所以：f[i][j][k]表示打到第i行,总共打了j个方块,其中第i行打了k个方块，则f[i][j][k]=max(f[i-1][j-k][p]+sum[i][k]) (Sum[i][k]表示第i行前k个数的和)

小小的提醒边界的处理——$k == 0$时直接转移会re，所以要单独处理的……

随便说说：

我记得当年比赛的时候做出这个题的同学说他是因为画着这个三角形的纸转了90度，然后就脑动大开了……

纸张旋转90度很容易，但是让思维跟着旋转呢？

4.3 与贪心结合的dp

有一篇国家集训队论文叫做《贪婪的动态规划》讲解了贪心思想和动态规划结合灵活运用在有些题目上会收到的奇效。在《贪婪的动态规划》一文中讲到动态规划有两种情况会借用贪心思想：一是**在确定状态的时候利用贪心的思想会使得问题变得更加容易**，这种情况的典型题是“田忌赛马”，本题需要通过贪心分析出“田忌出马时不是出最强的，就是出最弱的”这一点才能想出状态和转移方程（由于这是区间dp典型例题，在此就不再赘述, 不会的自行阅读暑假集训ppt或者求组度娘）。二则是**在状态非常庞大的时候利用贪心来优化算法!**

关于使用贪心优化算法：一些题目虽然容易确立出状态以及轻松的写出状态转移方程，但是直观上的算法往往效率不高，而贪心历来是与“高效”一词密不可分的，而运用好贪心思想有可能就能够使原来效率低下的算法得到重生……

4.3.1 贪心+dp例题1：Codeforces 459E Pashmak and Graph

题目大意：

题意：给你一个图，求一条边数最多的一条路径，并且这个路径的边的权值严格递增，输出这个边数的最大值。

输入数据：

```
3 3
1 2 1
2 3 1
3 1 1
3 3
1 2 1
2 3 2
3 1 3
6 7
1 2 1
3 2 5
2 4 2
2 5 2
2 6 9
5 4 3
```

4 3 4

输出数据：

1

3

6

思路：

这是个想通了就很简单很简单的题。。。

显而易见的贪心——小的边排前面……所以，先按照边长排序，然后dp，以当前边的尾节点为路径的结尾，然后一个边的首段点为u，尾端点为v,裸地转移就是dp[v]=max(dp[u]+1,dp[v])，注意边相等要单独处理，然后就ok了……

4.3.2 贪心+dp例题2：群巨要过河

题目大意：

清明节群巨打完月赛一起去春游，一共n个人，走到一条河的岸边，想要过河到另一边岸，但是却没有桥。于是水院的同学自告奋勇造了一艘船，但是这个船太小了，每一次过河一次只能坐两个人。

就像大家切题的手速不一样，群巨摆渡渡河的时间也各不相同t，船划到对岸的时间等于船上渡河时间较长的人所用的时间。

现在已知每一个巨巨的渡河时间t，未来ceo大人天巨想知道最少要花费多少时间，才能使所有人都过河。

输入格式

多组数据，以文件结尾结束，每组数据第一行为人数n，一下有n行，每行一个数。第i+1 行的数为第i个人渡河时间。

输出格式

对于每组数据输出一个数，表示所有人渡过河最少渡河时间。

样例输入

4

6

7

10

15

样例输出

42

HINT

初始：这边1,2,3,4，对岸

第一次：这边3,4，对岸1,2，时间7

第二次：这边1,3,4，对岸2，时间6

第三次：这边1，对岸2,3,4，时间15

第四次：这边1,2，对岸3,4，时间7

第五次：，对岸1,2,3,4，时间7

所以总时间为42，没有比这更优的方案了。

思路：

直接想状态貌似不可行的样子……

显而易见的：花费时间较长的2个人坐同一条船可以节省时间

那么我们就可以考虑**让1或2来回运船**

于是就很容易想到让1,2两个人来回奔波，其代价是a[2](1,2一起过)+a[1](1回来)+a[n] (最费时间两个人一起过)+a[2](2回来)，样例就是这样的，但这个方法并不全面……

因为还有一种方法是**2不上场，让1一个人来回送那两位**。

eg:5 7 8 9这组数据:

1,2来回接:7+5+9+7=28

1一个人接:9+8+5+5=27

所以最优值应该在这二者中取最小的。

于是它由贪心变为了DP……

设**f[i]为前i个人过河时所用的时间**

a[i] 从小到大排序!

最快的那个人与任意一个过河，然后回来:f[i] = f[i-1] + a[1] + a[i]。

最快那个与次快那个过河，然后次快那个回来，任意两人过河，最后最快那个回来: f[i] = f[i-2] + a[1] + a[i] + a[2] ×2

所以得出动态转移方程：**f[i]=min(f[i-1]+a[1]+a[i],f[i-2]+a[1]+a[i]+a[2]×2)**

思考：

这个题是利用了贪心确定了转移的方法，然后就可以求出转移方程了……

4.3.3 贪心+dp例题3：HDU 4976 A simple greedy problem

题目大意：

有n个小兵，编号1 n，血量为ai，均为整数。现在两人轮流打小兵，主角先选最多一个兵造成一点伤害（也就是可以不选），然后对手会对所有兵造成1点伤害，循环。求主角最多能杀掉多少个兵。$n \leq 1000$，$ai \leq 1000$。

输入数据：

2

5

1 2 3 4 5

5

5 5 5 5 5

输出数据：

Case #1: 5

Case #2: 2

思路：

很明显，如果所有小兵血量都不一样，那么主角可以打死所有的小兵，如果没有就得先把相等的小兵的血变成不等的……而变成不等的是用贪心——用尽量小的次数让所有数都不等，尽量减小得少……代码是这样：

```
1         for (int i = 0; i < n; i++)
2         for (int j = a[i]; j > 0; j--)
3           {
4             if (b[j] == 0)
5             {
6               b[j] = a[i];
7               break;
8             }
9           }
```

然后就可以在预处理的基础上dp了,用f[i][j]，表示进行了i组攻击（主角攻击一次，对手攻击一次，称为一组），主角留有j次攻击，最多能杀死的小兵数目(还可以就地滚动哦) f[j] = max(f[j], f[j - b[i] + i - 1] + 1);（b[i]是用最少的次数变得每一个数都不一样以后的数值……）

思考：

先贪心确定方法（目标状态），然后再dp计算能让多少个小兵达到要求的情况，这也是一种很有趣的办法哈哈!

4.4 高维dp

对于高维的dp我只有一句话：敢想，敢写!!

4.4.1 高维dp例题1：最小密度路径

题目：

给出了一张有N个点M条边的加权，接下来有Q个询问，每个询问包括2个节点X和Y，要求算出从X到Y的一条路径，使得密度最小（密度的定义为，路径上边的权值和除以边的数量）。

输入格式

第一行包括2个整数N和M。

以下M行，每行三个数字A、B、W，表示从A到B有一条权值为W的有向边。

再下一行有一个整数Q。

以下Q行，每行一个询问X和Y，如题意所诉。

输出格式

对于每个询问输出一行，表示该询问的最小密度路径的密度（保留3位小数），如果不存在这么一条路径输出“OMG!”（不含引号）。

输入样例

3 3

1 3 5

2 1 6

2 3 6

2

1 3

2 3

输出样例

5.000

5.500

数据规模

对于60%的数据，有$1 \leq N \leq 10$，$1 \leq M \leq 00$，$1 \leq W \leq 1000$，$1 \leq Q \leq 1000$；

对于100%的数据，有$1 \leq N \leq 50$，$1 \leq M \leq 1000$，$1 \leq W \leq 100000$，$1 \leq Q \leq 100000$。

思路：

看起来和floyd很像啊啊啊啊，但是，问题来了，a到b之间的最小密度路径加上b到c之间的最小密度路径真的的a到c的最小密度路径么？no no no！这是不满足最优子结构和无后效性的啊啊啊……

不满足无后效性咋个办？最简单的办法是**加一维**试试!

f[i][j][k][p]表示i到j经过k点做中转且有p条边，然后枚举一下边数不就ok了么？是不是挺简单哈哈!

4.4.2 高维dp例题2：SCOI 2008 着色方案

题目：

有n个木块排成一行，从左到右依次编号为1 n。你有k种颜色的油漆，其中第i种颜色的油漆足够涂ci个木块。所有油漆刚好足够涂满所有木块，即c1+c2+...+ck=n。相邻两个木块涂相同色显得很难看，所以你希望统计任意两个相邻木块颜色不同的着色方案。

输入

第一行为一个正整数k，第二行包含k个整数c1,c2, ... ,ck。

输出

输出一个整数，即方案总数模1,000,000,007的结果。

样例输入

3

1 2 3

5

2 2 2 2 2

10

1 1 2 2 3 3 4 4 5 5

样例输出

10

39480

85937576

数据规模

50%的数据满足：$1 \leq k \leq 5, 1 \leq ci \leq 3$

100%的数据满足：$1 \leq k \leq 15, 1 \leq ci \leq 5$

思路：

作为一个dp，想到一个合适的状态是一件很重要的事情，对于这一个题来说，这确实也是最难的一点……对于百分之五十的数据，我们当然可以最简单的用f[i][j][k][l][m]表示每一种种颜色分别剩i,j,k,l,m个的方案数，但是还有百分之五十数据就没办法了……ACM又不是oi，可以骗分……这真是一个悲伤的故事……

这个时候，如果你善于观察，你就会发现ci是小于5的,如果你对于数据范围很敏感，你就大概能够想到这样一个状态了：若每种颜色刷的木板数记为Ci 那么 **f[a][b][c][d][e][last]表示C值为1的颜料有a种，C值为2的颜料有a种……最后一个格子颜色的C值是last!**

怎么转移呢？？？ 动态规划的转移就是枚举最后一次决策嘛……

所以：

f[a][b][c][d][e][1] = f[a-1][b][c][d][e][1] * a + f[a+1][b-1][c][d][e][2] * b + f[a][b+1][c-1][d][e][3] * c
+ f[a][b][c+1][d-1][e][4] * d + f[a][b][c][d+1][e-1][5] * e

f[a][b][c][d][e][2] = f[a-1][b][c][d][e][1] * (a - 1) + f[a+1][b-1][c][d][e][2] * b + f[a][b+1][c-1][d][e][3] * c
+ f[a][b][c+1][d-1][e][4] * d + f[a][b][c][d+1][e-1][5] * e

f[a][b][c][d][e][3] = f[a-1][b][c][d][e][1] * a + f[a+1][b-1][c][d][e][2] * (b-1) + f[a][b+1][c-1][d][e][3] * c
+ f[a][b][c+1][d-1][e][4] * d + f[a][b][c][d+1][e-1][5] * e

……

代码：

这个题记忆化搜索肯定是要好写一些，虽然状态转移方程看起来很恐怖的样子，但是只要代码写得有点技巧，还是很简洁的!!（表示我自己最开始写那个，简直是不忍直视）

```
1
2         #include<cstdio>
3         #include<cstdlib>
4         #include<cstring>
5         #include<iostream>
6         using namespace std;
7         #define N 17
8         #define L 7
9         #define mod 1000000007
10        int n,k;
11        int s[L];
12        long long f[N][N][N][N][N][L];
13        bool v[N][N][N][N][N][L];
14        long long Dfs(int a, int b, int c, int d, int e, int x)
15        {
16          if (a + b + c + d + e == 0) return 1;
17          long long tmp = 0;
18          if (v[a][b][c][d][e][x]) return f[a][b][c][d][e][x];
19          if (a) tmp += (a - (x==2)) * Dfs(a-1,b,c,d,e,1), tmp %= mod;
20          if (b) tmp += (b - (x==3)) * Dfs(a+1,b-1,c,d,e,2), tmp %= mod;
21          if (c) tmp += (c - (x==4)) * Dfs(a,b+1,c-1,d,e,3), tmp %= mod;
22          if (d) tmp += (d - (x==5)) * Dfs(a,b,c+1,d-1,e,4), tmp %= mod;
23          if (e) tmp += e * Dfs(a,b,c,d+1,e-1,5), tmp %= mod;
24          v[a][b][c][d][e][x] = 1;
25          f[a][b][c][d][e][x] = tmp;
26          return f[a][b][c][d][e][x];
27        }
28        int main()
29        {
```

```
30          int t;
31          scanf("%d", &k);
32          for (int i = 1; i <= k; i++)
33          {
34            scanf("%d" ,&t);
35            s[t]++;
36            n += t;
37          }
38          printf("%lld", Dfs(s[1], s[2], s[3], s[4], s[5], 0));
39          return 0;
40        }
```

本题还有一种利用组合数学的解法,详细就讲解以后补上……

4.5 多重dp

个人觉得多重dp主要的思维方式是两种：**分解**和**转化**……分解是说将一个问题分解成比较简单一点的两步（或多步），而转化则是转化通过一次dp将问题成一个比较简单的问题……而这两种想法结合就解决多重dp了。

4.5.1 多重dp例题1 ： Min酱要旅行

题目：

从前有个富帅叫做Min酱，他很喜欢出门旅行，每次出门旅行，他会准备很大一个包裹以及一大堆东西，然后尝试各种方案去塞满它。

然而每次出门前，Min酱都会有个小小的烦恼。众所周知，富帅是很讨妹子喜欢的，所以Min酱也是有大把大把的妹子，每次出门都会有一只妹子随行。然而这些妹子总是会非常排斥Min酱准备的众多东西中的一件（也许是因为这件东西是其它妹子送给Min酱的），这件东西Min酱是万万不敢带上的，否则的话……嘿嘿嘿。另外，妹子们嫌Min酱的包裹太丑了，会自带一个包裹去换掉Min酱的包裹。

Min酱是个控制欲很强的人，然而这样一来，Min酱就不知道可以用多少种方案去填充包裹了，所以Min酱很郁闷。

于是Min酱找到了聪明的你，希望你能帮助他解决这些问题。

另外，Min酱是个典型的懒人，他不希望每次带不同的妹子出去都麻烦你，所以他希望你能给出有K1..Kn件物品，第i件不能带并且包裹大小为1..M的所有方案数。

Input

可能有多组数据。对于每一组数据：

第一行，两个整数n,m，分别表示物品数量和妹子带的包裹的最大容积。

第二行，n个整数，分别表示物品ki的体积。

$1 \leq n$, $m \leq 2300$ ， $ki \leq 1000$

Output

对于每一组数据，输出一个n×m的矩阵，第i行j列表示包裹容积为j，不能带i号物品时，装满包裹的方案总数。

为了美观起见，我们只保留方案数的个位。

Sample Input

3 2

1 1 2

Sample Output

11

11

21

思路：

这个题上次写dp三个题题解的时候写过，而这一次是站在多重dp的角度来看这个题，还是很有特色的!

第一次dp是f[i]表示背包体积为i的时候的方法数,01背包怎么做我就不说了f[i] = f[i - v[j]] + c[j] ……然后我们需要利用f[i]来求答案……

此时g[i]表示背包体积为i当前物品不用的方法数,那么g[i] = f[i] - 体积为i且当前物品一定要用的方法数! 当前物品一定要用怎么算？

——其实就是g[i - 当前物品体积] 嘛!!

所以：g[i] = f[i] - g[i - v[j]]

思考：

通过g[i - v[j]]表示一定用第j个物品的时候的方案数，这种转化也是很常见的。

当一个问题求解困难的时候，将其转化成整体减去他的反面，而他的反面正好通过之前的量就可以算出来……

4.5.2 多重dp例题2 ： SCOI 2009 粉刷匠

题目：

windy 有N条木板需要被粉刷。每条木板被分为M个格子。每个格子要被刷成红色或蓝色。windy每次粉刷，只能选择一条木板上一段连续的格子，然后涂上一种颜色。每个格子最多只能被粉刷一次。如果windy只能粉刷T次，他最多能正确粉刷多少格子？一个格子如果未被粉刷或者被粉刷错颜色，就算错误粉刷。

输入：

输入文件paint.in第一行包含三个整数，N M T。接下来有N行，每行一个长度为M 的字符串，'0'表示红色，'1'表示蓝色。

输入：

输出文件paint.out包含一个整数，最多能正确粉刷的格子数。

输入样例：

3 6 3

111111

000000

001100

输出样例：

16

数据范围：

30 % 的数据，满足$1 \le N, M \le 10, 0 \le T \le 100$

100 %的数据，满足$1 \le N, M \le 50, 0 \le T \le 2500$

思路：

表示如果只有一个木板……几乎是很好解决的——我们可以求出该木板粉刷x次，能够正确粉刷的格子数!

用**f[k][i][j]表示第k 条木板，粉刷i 次，前j 个格子能够正确粉刷的格子** 数，那么方程是显然的：**f[k][i][j] = max f[k][i −1][p] + w[p + 1,][j]**

($0 \le p \le i$, w[l][r]表示l,r段一次粉刷能够正确粉刷的格子数)

然后我们可以进行第二次动态规划：利用上一次的结果，求出所有木板粉刷y次，能够正确粉刷的格子数————

实际是一个**分组背包**问题——将每条木板看成一个物品组，则物品组中每个物品的费用c[i]和价值w[i]分别为x和f[k][x][M]。所求为从每组中最多选一件，求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量T，且价值（粉刷正确的格子数）总和最大.

4.5.3 多重dp例题3 ： HDU4455 Substring

题目大意:

给你一个序列，给出Q个查询，问对于每一个长度W的序列的不同元素的个数和。

for example:

序列1 1 2 3 4 4 5

w = 3 时，substring为:(1,1,2),(1,2,3),(2,3,4),(3,4,4),(4,4,5)

不同元素个数分别为: 2 3 3 2 2

所以w = 3时，答案是：12

Sample Input

7

1 1 2 3 4 4 5

3

1

2

3

0

Sample Output

7

10

思路:

状态很直观：f[i]表示子串长度为1的时候的答案,显而易见f[1] = n

那么怎么由f[n - 1] 推出f[n]呢？？

假设我们已经算出了f[3] 要来算f[4]

(112)3(445)

1 1 2 会变成1 1 2 3 ，1 2 3 会变成1 2 3 4……

4 4 5 会没有

所以：对于每一个位置大于等于4的元素:

如果他和前面三个都不一样（和他一样的跟他的距离大于3），最终的结果就会加一

并且，要把最后3个数的不同的元素个数减掉

于是，转移方程为：

f[i] = f[i - 1] + delta[i] - different[n - 1]

然后就是delta 和different 怎么算的问题了……

different 从后往前扫描一遍即可,O（n）不多说

delta[]直接求有困难，能不能再转化一下？？

通过刚刚的分析我们知道，——在i= k的时候要增加的个数实际等于(自己)与(前一个和自己相同的数)的距离大于k的数的个数

那么，首先我们用O(n)的扫描维护出每个数前一个与他相同的数的位置

然后再扫描一遍维护出每个数到他前一个数的距离。

接着再次扫描维护出离前一个距离为i的数的个数。

最后一个前缀和就可以求出与前一个距离小于i的数的个数,

用的时候，用总数减去它就是大于的啦啦啦啦!!!

每一步都是O(n)的，所以总时间复杂度也是O(n)的!!

思考:

本题的思路是这样的：先找出一个**最直观的方程**，然后发现方程里面的有没法求出来的东西，这个时候就开始考虑**转化那些求不出来的量**，一步一步转化到能求为止就额可以解决问题了。

这种转化其实不少见，只是在我们做题的时候往往方程里面有求不出来的东西或者是直接求解时间复杂度很高的东西就放弃了，其实这些要求的量是可以通过一次次递推来转化的……

4.6 有趣的dp状态设计

4.6.1 有趣状态dp例题1：RQNOJ 595 教主泡嫦娥

题目：

2012年12月21日下午3点14分35秒，全世界各国的总统以1及领导人都已经汇聚在中国的方舟上。但也有很多百姓平民想搭乘方舟，毕竟他们不想就这么离开世界，所以他们决定要么登上方舟，要么毁掉方舟。

LHX教主听说了这件事之后，果断扔掉了手中的船票。在地球即将毁灭的那一霎那，教主自制了一个小型火箭，奔向了月球……

教主登上月球之后才发现，他的女朋友忘记带到月球了，为此他哭了一个月。

但细心的教主立马想起了小学学过的一篇课文，叫做《嫦娥奔月》，于是教主决定，让嫦娥做自己的新任女友。

教主拿出他最新研制的LHX(Let's be Happy Xixi)卫星定位系统，轻松地定位到了广寒宫的位置。

见到嫦娥之后，教主用温柔而犀利的目光瞬间迷倒了嫦娥，但嫦娥也想考验一下教主。

嫦娥对教主说：“看到那边的环形山了么？你从上面那个环走一圈我就答应你～”

教主用LHX卫星定位系统查看了环形山的地形，环形山上一共有N个可以识别的落脚点，以顺时针1 N编号。每个落脚点都有一个海拔，相邻的落脚点海拔不同（第1个和第N个相邻）。

教主可以选择从任意一个落脚点开始，顺时针或者逆时针走，每次走到一个相邻的落脚点，并且最后回到这个落脚点。

教主在任意时刻，都会有“上升”、“下降”两种状态的其中一种。

当教主从第i个落脚点，走到第j个落脚点的时候（i和j相邻）

j的海拔高于i的海拔：如果教主处于上升状态，教主需要耗费两段高度差的绝对值的体力；否则耗费高度差平方的体力。

j的海拔低于i的海拔：如果教主处于下降状态，教主需要耗费两段高度差的绝对值的体力；否则耗费高度差平方的体力。

当然，教主可以在到达一个落脚点的时候，选择切换自己的状态（上升到下降，下降到上升），每次切换需要耗费M点的体力。在起点的时候，教主可以自行选择状态并且不算切换状态，也就是说刚开始教主可以选择任意状态并且不耗费体力。

教主希望花费最少的体力，让嫦娥成为自己的女朋友。

输入格式

输入的第一行为两个正整数N与M，即落脚点的个数与切换状态所消耗的体力。

接下来一行包含空格隔开的N个正整数，表示了每个落脚点的高度，题目保证了相邻落脚点高度不相同。

输出格式

输出仅包含一个正整数，即教主走一圈所需消耗的最小体力值。

注意：C++选手建议使用cout输出long long类型整数。

样例输入

6 7

4 2 6 2 5 6

样例输出

27

样例说明

从第3个落脚点开始以下降状态向前走，并在第4个落脚点时切换为上升状态。

这样共耗费$4+(7)+3+1+2^2+2^2+4=27$点体力。

数据规模

对于10%的数据，$N \leq 10$；

对于30%的数据，$N \leq 100$，$a[i] \leq 1000$；

对于50%的数据，$N \leq 1000$，$a[i] \leq 100000$；

对于100%的数据，$N \leq 10000$，$a[i] \leq 1000000$，$M \leq 1000000000$；

思路：

对于这个题目的小数据dp是显而易见的。。。f[i][j]表示在第i格，状态为j的体力。转移方程也很好写(在此不再赘述)。但是其不足是要枚举起点（起点不同会导致体力不同，这个不像有的人想的那样随便从哪个地方出发体力一样），于是复杂度是O(n^2)

必须要想办法……如果优化后还是dp的话，显而易见我们很想要把“枚举起点”这一步去掉……

来一个新的状态：我们用f[i][0..1][0..1]表示在第i个点，状态为0或1（上升或下降），是否改变过状态（0或1）。

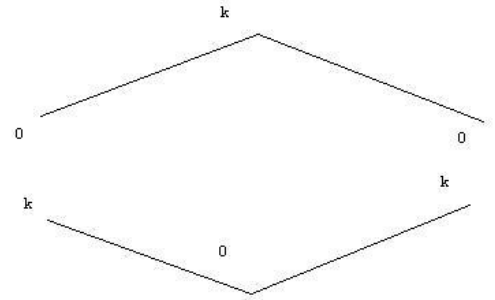
对于跳跃过程，可以分成三种情况：

一：**中途没有换过状态；**

二：**换了奇数次状态；**

三：**换了偶数次状态。**

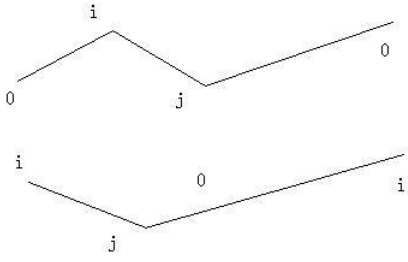
第一种情况：min(f[n][1][0],f[n][0][0])。



奇数次，就以1为例。路径是一个环，所以不论从何处出发，消耗在跳跃上的体力是一定的；所不同的只是转换所耗的体力。

容易想到，如果是在0处或是K处出发，那么只要转换一次状态；如果是从其他地方出发，就要转换两次，所以，最优解就是从0出发再回到0所耗的体力。也就是min(f[n][0][1],f[n][1][1])。

容易想到，这也可以推广到一般奇数次转换的情况



偶数次，就以2为例，首先如果最开始是由上升的，那么最后回来0时也是上升的，所以在推的时候要保证所有状态是从一个初始状态转移的。

同理，花在跳跃上的体力是一个定值，所不同的是转换所耗的体力。而转换N次总可以变成转换N-1次，并且转换偶数次后，会就成和初始状态一样，所以，如果最开始是从f[0][0][0]转移的，那么答案就应是f[n][0][1]-m，如果答案是从f[0][1][0]转移的，那么答案就应是f[n][1][1]-m。为了保证答案是从一个确定初始状态转移过来，我们可以把其他三个初始状态都赋一个极大值。

这也可以推广到一般的偶数次的情况。

第二种情况：min(f[n][1][1],f[n][0][1])。

第三种情况：首先要保证所以的状态从一个初始状态转移过来，然后，如果是从f[0][0][0]转移过来的，那么答案就是f[n][0][1]-m；如果是从f[0][1][0]转移过来的，答案就是f[n][1][1]-m。

随便说说：

偷懒直接用了高中同学题解的讲解……

（Q:你个懒人为什么不自己写一遍? 本弱:写了一遍还没有高中同学写得清楚所以就放弃了。。 Q:你为何这么弱！ 本弱：呜呜呜）

在此给出来源的网址，http://blog.sina.com.cn/s/blog_9aa2786a0101700v.html 特此膜拜兽哥……高中题解写得够详细orz!!

5 等待学习和写的部分

6 概率dp

6.1 基于联通性状态压缩的动态规划

6.2 组合数dp

6.3 dp优化

6.3.1 单调队列优化dp

6.3.2 斜率优化dp

6.3.3 四边形不等式优化dp

6.3.4 矩阵快速幂优化dp

6.4 dp和其他算法和数据结构

6.4.1 dp 和字符串

6.4.2 dp 和博弈论

6.4.3 dp 和图论

7 更新说明

2015.2.12：完成1.0初始版
2015.2.14：修改【树状dp】的部分描述
2015.2.14：增加【二分优化的dp】
2015.3.22: 修正了一些小错误
2015.5.14：增加【线段树优化的dp】以及【关于dp拓扑序的思考】
2015.5.21：修正了一些描述，加入了一些新的理解！并谁关键部分进行了加粗!
2015.8.10：在【背包问题】,【dp降维】和【状压dp】中各添加了一个例题!

8 后记

写完这个总结，感觉整个人收萌萌哒的了，哎……真是恐怖啊，居然一写就停不下来，写了这么多……

当年开脑洞的题一道的翻出来看，然后发现自己又不会了的感觉简直是醉醉哒，一点一点的复习，看自己当年留在空间里的题解，然后发现自己的题解简直写得很挫很挫，自己都看不懂了呵呵呵，不过那个时候至少还每个题写了题解，让我现在有东西可以翻……

顿时对于从去年暑假以后csdn的博客就没有更新了感到很是惭愧啊啊啊啊，一直说写，总是不写是个什么毛病？拖延症颓废症晚期么？？

对于动态规划，个人真的是很喜欢，但是一直学得不好，真是悲伤，个人觉得，dp就是需要多见题目的类型，然后多反思，多思考——为什么这个状态我想不到？为什么这个优化可以这样？这个方法要用在什么样的地方？怎样可以把这个思路扩展出去？

dp的题是讲不完了，只有通过多这样想，才能有举一反三的能力!!

但是我必须强调一点：反三的前提是举一，如果见都没见过这种类型的题和这个方法，要想想一想就会做了谈何容易?

所以多积累真的非常非常非常重要!!!!! 不仅是dp，其他类型的题也是一样!!

等你dp做多了，你就会觉得看到题的时候状态呼之欲出……当然本弱现在的水平，也就是看着水水哒题能秒出方程……然后再难一点就只能干瞪眼了!

写完题解我整个人都已经萌萌哒的了，所以肯定会有错，群巨发现的话请立即指正……

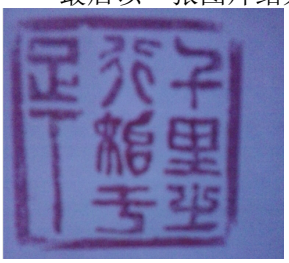
待写七八个小部分，等我学会了再更新……

但愿有生之年能够把这份总结更新完成吧……

通过写这次的题解，还是想通了很多的问题，真的很有收获!!

不过对于把一份dp进阶指南，写成了类似题目汇总归纳我表示万分惭愧和无语……表示要是有希望通过这份总结来进一步学习dp的同学，千万别抱太多的希望，我现在自己都对自己很无语……

最后以一张图片结束这篇总结……



路漫漫其修远兮，吾将上下而求索!