

动态规划入门

Dynamic Programming Book for Beginners

Mingqi_H

version = 3.0.8α

Build Time: 2018-02-03 21:56:22+08:00

To Hatsune Miku

目录

0 自我介绍 && 前言	1
0.1 作者介绍	1
0.2 参与者	1
0.3 前言	1
1 记忆化搜索	3
1.1 我们为什么需要记忆化搜索?	3
1.2 如何实现记忆化搜索?	5
2 对动态规划的初步了解	7
2.1 一些定义	7
2.2 动态规划实例	7
2.2.1 最优子结构	9
2.2.2 无后效性	9
2.3 动态规划的应用及其局限性	10
3 动态规划基础——序列型 DP	14
3.1 最长上升子序列问题—LIS	14
3.1.1 介绍与实现	14
3.1.2 LIS 的打印	16
3.1.3 例题—导弹拦截	16
3.2 最长公共子序列—LCS	18
3.2.1 介绍与实现	18
3.2.2 LCS 的打印	20
3.2.3 LCS 的优化	20
4 动态规划初步——背包问题	22
4.1 背包问题是什么?	22
4.2 01 背包问题	23
4.2.1 简介	23
4.2.2 搜索	24
4.2.3 二维 DP	25
4.2.4 一维 DP	26

4.3	完全背包问题、多重背包问题与混合背包问题	28
4.3.1	完全背包问题	28
4.3.2	多重背包问题	28
4.3.3	混合背包问题	29
4.4	二维背包问题与分组背包问题	31
4.4.1	二维背包问题	31
4.4.2	分组背包问题	31
4.5	有依赖的背包问题与泛化物品（选学）	33
4.5.1	有依赖的背包问题	33
4.5.2	泛化物品	35
5	动态规划提高——棋盘型 DP 与区间型 DP	38
5.1	棋盘型 DP	38
5.2	区间型 DP	39
6	动态规划 Professional——树形 DP、状压 DP 与数位 DP	42
6.1	树形 DP	42
6.2	状压 DP	47
6.2.1	定义与特点	47
6.2.2	有关位运算的知识	47
6.2.3	基础例题—愤怒的小鸟	48
6.2.4	提高例题—宝藏	52
6.2.4.1	随机化暴力	55
6.2.4.2	模拟退火	56
6.2.4.3	最小生成树	61
6.2.4.4	状压 DP	62
6.3	数位 DP	65
6.3.1	基础知识	65
6.3.2	入门例题—不要 62	67
6.3.3	常数优化	69
6.3.4	减法的艺术—F(x)	70
7	典型题目与难题选讲	73
7.1	典型题目	73

7.2 难题选讲	75
7.2.1 组合数问题—数论/数学, DP, 前缀和优化	75
7.2.2 换教室—期望 DP	78

0 自我介绍 && 前言

0.1 作者介绍

本人黄铭祺，GRYZ 三校区 60 级蒟蒻 *OIer* × 1.

参加过 NOIp 2017 提高组复赛.

我不会 DP，但是既然老师要我讲，那也就现学现卖地来讲一点.

另外我不会讲课，很有可能讲不好，如果有错误还请指出。

0.2 参与者

参与文档改进的是同一学校的另一位 wcz 同学. 他对本文档作出了大幅修改.wcz 本人与作者是好朋友.

0.3 前言

这篇文章主要介绍动态规划，当然动态规划是一种思想，题目种类也非常多，不可能在这么短的时间内讲完，所以这节课主要是对动态规划的初步介绍，会做最基础的题目，更重要的是理解动态规划的思想，这就足够了。

当然，需要有一定的前置知识，即语言部分与图论，搜索，这里默认读者们拥有一定的编程经验和对基本算法有一定的了解。

这节课要讲的东西可能有些多了。大家先看看目录讨论一下重点要听什么内容。序列型 DP、背包问题是所有 DP 的基础，棋盘型 DP、区间型 DP 什么的也要涉及一点，剩下的类型就看讲课的时间和大家的理解程度吧。

这节课覆盖的内容有几乎所有常见种类的 DP，以及 DP 优化的一些技巧，是为了让各位对 DP 的各种类型有一个初步的了解。各个例题应该都属于非常基础的类型，如果搞不懂的话可以上网查相关资料（不要来问我，我也可能讲不明白）。



1 记忆化搜索

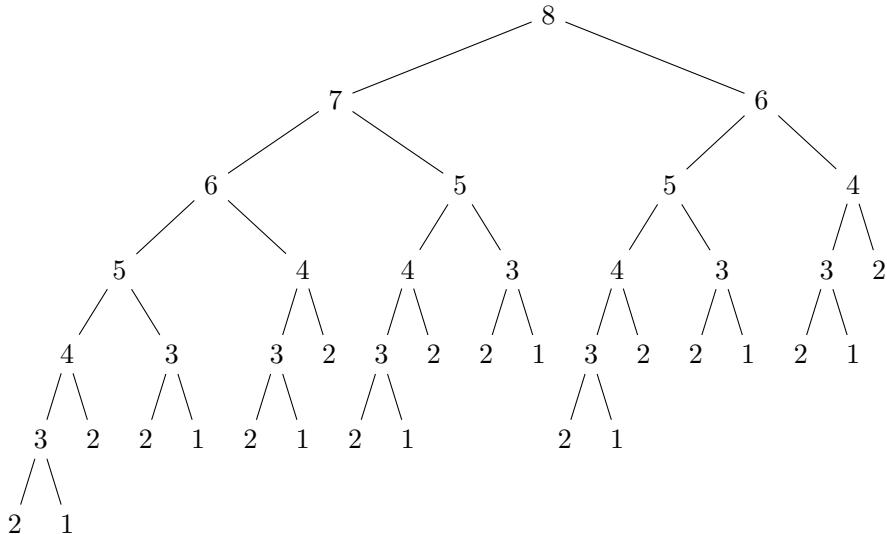
1.1 我们为什么需要记忆化搜索？

这一章我们讲记忆化搜索，记忆化搜索这个东西其实和普通的搜索没有太大的区别，是一个所谓“以空间换时间”的操作。

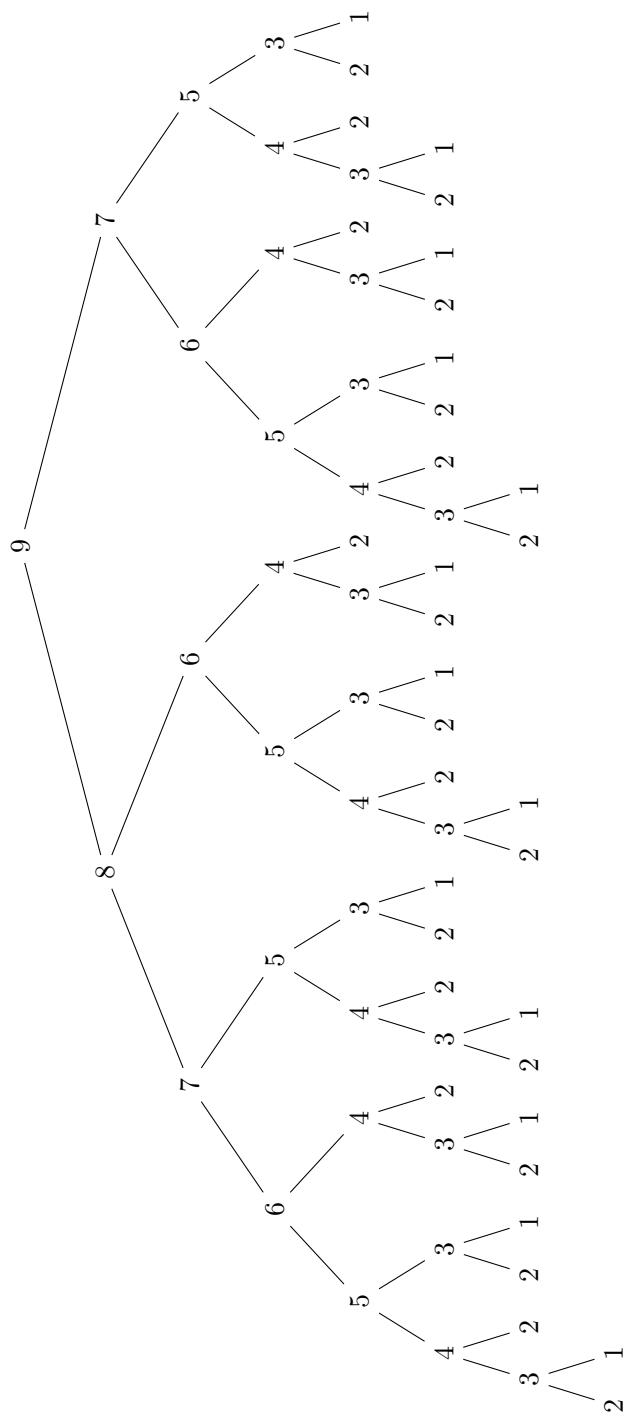
举个例子，我们要计算斐波那契数列的第 n 项。普通的递归可能会这么写：

```
int fib(int x)
{
    if(x==1 || x==2) return 1;
    return fib(x-1)+fib(x-2);
}
```

下图是当 $n = 8$ 时递归树的情况，可以看到，在调用过程中出现了大量的重叠子问题，而此时我们的程序不得不进行重复递归调用，这消耗了大量的时间。



再来更加极端一点的例子，下页图显示了当 n 的值扩大 1 时，问题的规模就几乎扩大了一倍。这几乎是不可接受的。



$n = 9$ 时的递归树，可以看到重叠子问题更多了，问题规模几乎扩大了一倍。

n 的取值	递归调用次数
.....
40	204668309
41	331160281
42	535828591
43	866988873
44	1402817465
45	2269806339
.....

上表展示了当 n 的取值较大时递归调用次数的变化情况，经模拟测算可以发现增长率几乎是指数级别的增长，事实证明也如此。

这样的时间复杂度几乎是不可接受的。所以我们需要使用记忆化搜索这一手段来优化。记忆化搜索，又称为带备忘的搜索，顾名思义，就是把已经搜索过的结果记录下来，作为一个“备忘”，当程序需要再次调用这个搜索过程时就直接调用这个结果即可，不需要重复搜索了。

1.2 如何实现记忆化搜索？

还是以刚才的斐波那契数列为例。我们可以开一个数组 $f[]$ ，用于记录程序已经计算过的斐波那契数列的项的值。然后需要用的时候直接判断一下 f 数组里有没有这个元素（是否为初始值），如果有就直接返回这个值，没有再递归调用。代码如下：

```

int f[1000]; //初始值为 0，因为斐波那契数列中不可能出现 0，所以 0 是非法的。
int fib(int x)
{
    if(x==1||x==2) return 1; //递归边界条件
    if(f[x]!=0) return f[x]; //如果已经计算过了就返回结果。
    return f[x]=fib(x-1)+fib(x-2); //如果还没有计算过就递归计算并保存结果。
}

```

下表展示了记忆化搜索递归调用的次数：

n 的取值	递归调用次数
.....
40	3
41	3
42	3
43	3
44	3
45	3
.....

可见，记忆化搜索能够帮我们减少递归的调用，提高代码的效率，以上代码即使在 n 非常大的情况下仍能较快的给出结果。

同时，不可忽略的一点是记忆化搜索需要有一个与状态等大的数组（当然你也可以用 map 或者 set 之类的东西，只不过稍慢一些），空间占用很成问题。这也是记忆化搜索（包括 DP）的弊端所在。当然由于 DP 在某些情况下可以优化空间复杂度（压缩之类的），所以存在一类题目，可以使得 DP 通过而记忆化搜索 MLE 或 TLE。

记忆化搜索通常适用于对于 DFS 的优化，对于 BFS 就不是那么实用了。当你在考场上写不出来 DP 但又非常确定这道题目是个 DP 的时候，不妨先打个暴力，然后转成记忆化搜索，可以帮助你拿到很多分数，还是非常实用的一种技巧。

下面给出普通搜索转记忆化搜索的模板：

```

int f[1000][1000]; //与搜索的参数个数相同
void dfs(int a,int b)
{
    if(找到了这个问题的解) return ans;
    if(f[a][b]!=nil) return f[a][b]; //这里的 nil 表示一个无效的值，可以认为是在正常情况下
    ↳ 不可能出现的一个值
    //状态转移过程被省略一部分
    return f[a][b]=dfs(...,...); //这一句原来是 return dfs(...,...);
}

```

课堂笔记

Note on the text

2 对动态规划的初步了解

动态规划通常是用来处理一些最优化问题，需要问题具有可以划分阶段的特性以及其抽象模型所需要满足的一些条件。可以将对问题一定阶段的解法抽象成一个状态，从这一阶段的状态得到下一阶段的状态的过程被称为状态转移，我们的目标就是求出完整问题的最优状态。

2.1 一些定义

定义 2.1.1 (动态规划 (dynamic programming)) 是运筹学的一个分支，是求解决策过程 (*decision process*) 最优化的数学方法。20世纪 50 年代初美国数学家 R.E.Bellman 等人在研究多阶段决策过程 (*multistep decision process*) 的优化问题时，提出了著名的最优化原理 (*principle of optimality*)，把多阶段过程转化为一系列单阶段问题，利用各阶段之间的关系，逐个求解，创立了解决这类过程优化问题的新方法——动态规划。

定义 2.1.2 (状态) 状态是指解决问题过程中的每个步骤或可能，类似搜索中的状态（还记得搜索题常见的 *cur* 和 *nxt* 这两个变量吗？就是这样的东西）。

定义 2.1.3 (状态转移方程) 状态转移方程是指解决问题过程中，由步骤 *A* 到步骤 *B* 所需要进行的操作以及状态发生的变化，类似搜索中状态的扩展（还记得八皇后问题中 *dx* 和 *dy* 这两个数组吗？就是类似这样的东西，实际上就是从上一步到这一步所需要进行的操作或者什么东西的改变）。

2.2 动态规划实例

我们先来看一道题目：

例题 2.2.1 由正实数构成的数字三角形排列如图所示，第一行的数为 a_{11} ；第二行的数从右到左依次为 a_{21}, a_{22}, \dots 第 n 行的数为 $a_{n1}, a_{n2}, \dots, a_{nn}$ 。从 a_{11} 开始，每一行的数 a_{ij} 只有两条边可以分别通向下一行的两个数 $a_{(i+1)j}$ 和 $a_{(i+1)(j+1)}$ 。用动态规划算法找出一条从 a_{11} 向下通到 $a_{n1}, a_{n2}, \dots, a_{nn}$ 中某个数的路径，使得该路径上的数之和达到最大。

令 $f(i, j)$ 是从 a_{11} 到 a_{ij} 路径上的数的最大和，并且 $f(i, 0) = f(0, j) = 0$ ，则 $f(i, j) = (\quad)$ 。

$$\begin{array}{cccc}
 & a_{11} & & \\
 a_{21} & & a_{22} & \\
 a_{31} & a_{32} & a_{33} & \\
 \vdots & \vdots & \vdots & \\
 a_{n1} & a_{n2} & \cdots & a_{nn}
 \end{array}$$

- A. $\max\{f(i-1, j-1), f(i-1, j)\} + a_{ij}$
- B. $f(i-1, j-1) + f(i-1, j)$
- C. $\max\{f(i-1, j-1), f(i-1, j)\}$
- D. $\max\{f(i, j-1), f(i-1, j)\} + a_{ij}$

题目中涉及的 $f(i, j)$ 其实就是一个可行的状态设计, 状态设计所必须满足的条件是它包含着所有需要的信息, 对例题来说只需要知道此到达位置的最大收益而无需知道具体路径.

这道题就是 NOIp 2017 提高组初赛试题第 11 题, 一个经典的数字三角形问题。但是已经涉及到了动态规划的本质了。动态规划最重要的两步就是**设计状态**和**考虑状态转移方程**。以上内容涉及到两个非常重要的概念, 状态和状态转移方程。我们先不学习 DP, 而是先温习一下我们已经**具备的知识**。

同学们大概是会做基本的递推和搜索了.

大部分动态规划都可以被表示成一种递推的形式.

对于搜索, DP 就可以看成是一种搜索。本质与搜索并没有任何不同, 那么为什么搜索的时间复杂度会达到 $O(2^n)$, 而 DP 可以做到 $O(n^k)$ 的时间复杂度呢?

大家应该知道, 搜索是利用状态转移来完成问题的整个步骤的, 动态规划也是这么做的, 我认为它与搜索的最本质区别是它是利用上一步的最优状态来进行这一步的转移. 而搜索是利用上一步的所有状态来进行本步的转移并且在其中选择最优的那个来更新答案.

这就可以看出, 动态规划能做的事情搜索都是能做的. 但是有搜索能做但是动态规划不能做的.

什么时候动态规划能解决这个问题呢? 问题关键是“最优”两个字. 这里会涉及到做动态规划的两个前提: **最优子结构** 和 **无后效性**.

2.2.1 最优子结构

尝试从直观上去体会它的含义.

对于一个可以划分为若干阶段的问题. 如果能利用它的一个子问题的最优解得到整个问题的最优解. 那么这个问题是具有最优子结构的.

对于这个问题, 从起点出发到达终点的最优解是整个问题的解, 而从起点到达其中不是终点的某个点的最优解是整个问题一个子问题的解.

考虑这个例题具不具有最优子结构. 我们可以发现, 如果最终的最优解出现在 $f(u, v)$, 也就是从起点以某条路径走到 $f(u, v)$ 会获得最大收益.

考虑到达 (u, v) 的路径, 我们知道只有两个点能到达 (u, v) , 也就是 $(u - 1, v - 1)$ 和 $(u - 1, v)$, 那么容易知道问题的最优解一定是从这两个点的最优解转移过来的. 不然一定会有更优的解. 说明这道例题是具有最优子结构的. 所以通过求出 $f(u, v - 1)$ 和 $f(u - 1, v - 1)$ 就可以得到 $f(u, v)$.

2.2.2 无后效性

无后效性指的是问题的一个阶段的状态已知, 那么这个阶段此后的阶段只与此阶段有关, 而与之前发生过的所有阶段的状态无关, 也不会影响到未来的任何阶段. 对于例题, 从起点到达某个点 (u, v) 的最大收益需要通过 $f(u - 1, v)$ 和 $f(u - 1, v - 1)$ 来求解, 而不需要知道 $f(u - 2, v - 1)$ 或者再之前的状态. 状态 $f(u, v)$ 也不会影响到 $f(u - 2, v - 1)$ 等未来状态. 我们知道例题同样是满足无后效性的.

通过对问题的进一步学习我们会知道对于一个问题的无后效性取决于状态设计.

解法 1 通过对问题的进一步分析容易得到问题的解法. 其状态转移方程可以这么写:

$$f(i, j) = \max\{f(i - 1, j), f(i - 1, j - 1)\} + a_{ij}$$

根据在递推方面的经验我们可以如果需要求出最终结果 $f(u, v)$. 我们首先需要求出 $f(u - 1, v - 1)$, $f(u - 1, v)$. 如果一步一步逆推的话, 大概需要递归来解决. 我们大概需要在程序中设计一个函数 $f(i, j)$.

```
f(i, j)
return max{f(i-1, j), f(i-1, j-1)}+a(i, j)

ans()
return f(u, v)
```

而对于这个问题而言正推和逆推并没有本质区别.

按照递推的设计思路, 这个程序应该这样设计.

```
for(int i=1; i<=u; ++i)
for(int j=1; j<=v; ++j)
f[i][j]=max(f[i-1][j]+f[i-1][j-1])+a[i][j];
```

2.3 动态规划的应用及其局限性

大家应该提高对于动态规划的重视. 在 NOIp 中, 动态规划通常会出 1~2 个题目, 当然并不是所有人都能全部做出来的, 说它没用, 是因为如果不能想出可靠的转移, 反而浪费了大量的时间, 还不如写最简单的算法, 并且 NOIp 通常情况下会给不少的部分分, 这部分部分分通常用不到 60 行代码就能拿到, 这意味着我们学好暴力, 每道 DP 题就可以拿到 20~60 分。

动态规划当然不是万能的, 有很大的**局限性**, 这一点在前面有所介绍.

- 所解决的问题必须是满足**最优子结构**和**无后效性**的可分阶段的问题.
- 某些能用动态规划解决的问题可能会有其他更好更优的算法.
- 其本身的局限性, n^k 的时间复杂度以及同样规模的空间占用.
- 设计出足够优秀的状态这有时是很难的, 见多识广很重要.

例题 2.3.1 最大正方形

题目描述

在一个 $n \times m$ 的只包含 0 和 1 的矩阵里找出一个不包含 0 的最大正方形，输出边长。

输入格式

输入文件第一行为两个整数 $n, m(1 \leq n, m \leq 100)$ ，接下来 n 行，每行 m 个数字，用空格隔开，0 或 1.

输出格式

一个整数，最大正方形的边长

输入样例

```
4 4
0 1 1 1
1 1 1 0
0 1 1 0
1 1 0 1
```

输出样例

```
2
```

这个题目和上一个题目是比较像的，在状态设计上和转移上。只需要对题目进行合适的建模。

考虑如下的状态设计， $f(i, j)$ 表示以 (i, j) 为右下角元素的最大正方形的大小。容易发现它满足最优子结构和无后效性。

$$\begin{array}{ccccccccc} \cdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots \\ \cdots & 1 & 1 & 1 & \cdots & 1 & \cdots \\ \cdots & 1 & 1 & 1 & \cdots & 1 & \cdots \\ \cdots & \vdots & \vdots & \vdots & \ddots & \vdots & \cdots \\ \cdots & 1 & 1 & 1 & \cdots & 1 & \cdots \\ \cdots & 1 & 1 & 1 & \cdots & 1 & \cdots \\ \cdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

我们容易发现如果对于一个位置 (i, j) ，以它为右下角元素存在一个边长大于 2 的正方形，那么 $(i - 1, j - 1), (i - 1, j), (i, j - 1)$ 都必定存在以其为右下角元素的正方形。考虑状态 $f(i, j)$ 由哪些状态转移而来，容易发现，

如果 $f(i, j) = k$, 那么 $f(i, j - 1), f(i - 1, j - 1), f(i - 1, j) \geq k - 1$



课堂笔记

Note on the text

3 动态规划基础——序列型 DP

应某位 dalao 要求，在讲今天的主要内容之前，先讲一下最基础的序列型 DP，即最长子序列问题。

3.1 最长上升子序列问题—LIS

3.1.1 介绍与实现

定义 3.1.1 (最长上升子序列) 给出一个序列 a_1, a_2, \dots, a_n , 求它的一个子序列（设为 s_1, s_2, \dots, s_n ），使得这个子序列满足这样的性质： $s_1 < s_2 < s_3 < \dots < s_n$ 并且这个子序列的长度最长。输出这个最长的长度。

以上是最长上升子序列的定义。其他最长子序列问题的定义也类似。

思考一下。尽量使用动态规划的思想（虽然还没讲）。

考虑状态 $dp[i]$ 表示当前枚举到了第 i 个数，当前最长上升子序列长度为 $dp[i]$ 。

用 $arr[]$ 表示原数组， $dp[]$ 表示最长上升子序列的长度。我们枚举每一个数，每一个数都枚举其前面所有的数。如果当前枚举到的数大于之前枚举的数，那么最长上升子序列的长度一定至少比这个大 1（为什么？）。用一个 ans 变量暂存一下，然后枚举结束后用 ans 更新一下当前的最长上升子序列的长度即可。

```
int ans;
dp[1] = 1;
for(int i = 2; i <= n; ++i)
{
    ans = dp[i];
    for(int j = 1; j < i; ++j)
        if(arr[i] > arr[j] && dp[j] > ans)
            ans = max(ans, dp[j]);
    dp[i] = ans + 1;
}
```

时间复杂度 $O(n^2)$ 。

当然这不是最优化的方法，考虑一下如何优化？

明显的，我们可以看到第二层枚举不是必需的。第二层枚举的作用是找到最接近且比当前枚举的数。所以可以使用二分查找优化掉这一层循环。

增加一个 b 数组， $b[i]$ 用以表示长度为 i 最长子序列的最后一个数最小可以是多少， k 表示当前 b 数组的长度，则：

$$dp[i] = \begin{cases} b[k+1] = arr[i] & , arr[i] > b[k] \\ b[binary_search(arr[i], 1, k)] & , arr[i] < b[k] \end{cases}$$

容易发现， b 数组始终是单调递增的。所以我们可以二分查找 $arr[i]$ 在 b 数组出现的位置，从而把时间复杂度降至 $O(n \log n)$ 。

二分应该都会吧？如果不会二分的话就看看代码：

```
int binarySearch(const int *Array, int start, int end, int key)
//Array: 待查找的数组, start: 起始点下标, end: 终止点下标, key: 待查找的数。
{
    int left, right;
    int mid;
    left = start;
    right = end;
    while (left <= right)
    {
        mid = (left + right) / 2;
        if (key == Array[mid]) return mid;
        else if (key < Array[mid]) right = mid - 1;
        else if (key > Array[mid]) left = mid + 1;
    }
    return -1; //没有找到
}
```

以上二分代码适用于一般情况，更常用的写法：

```
int binarySearch(int num, int l, int r)
//num 为待查找的数。
{
    while (l <= r)
    {
        int mid = (l+r)/2;
        if (num >= b[mid])
            l = mid + 1;
        else
            r = mid - 1;
    }
    return l;
}
```

当然如果愿意用 STL 的话，lower_bound() 函数也很好用：

```
#include<algorithm>
std::lower_bound(first,last,val); //在区间 [first, last) 中二分查找元素 val。
//如果找到相应的 val，则返回一个指向元素位置的指针；
//否则返回一个指向最接近 val 且小于 val 的元素位置的指针。
//另外一种用法：
std::lower_bound(first,last,val,comp);
//以自定义 comp 作为比较器进行比较（类似 sort 的用法），返回值同上。
//同理还有 upper_bound() 函数。
```

3.1.2 LIS 的打印

在状态转移的时候记录一波转移方向，保存在 pre[] 数组中，然后需要输出的时候就可以根据 pre[] 数组逆序构造出 LIS，使用一个栈保存起来就可以正向输出了。

代码实现起来比较容易，自己写一下，也不那么常用。

3.1.3 例题—导弹拦截

例题 3.1.1 导弹拦截

题目描述

某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭。由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹。

输入导弹依次飞来的高度（雷达给出的高度数据是不大于 50000 的正整数），计算这套系统最多能拦截多少导弹，如果要拦截所有导弹最少要配备多少套这种导弹拦截系统。

输入输出格式

输入格式：

一行，若干个整数（个数少于 100000）

输出格式：

2 行，每行一个整数，第一个数字表示这套系统最多能拦截多少导弹，第二个数字表示如果要拦截所有导弹最少要配备多少套这种导弹拦截系统。

输入输出样例

输入样例:

```
389 207 155 300 299 170 158 65
```

输出样例:

```
6  
2
```

导弹拦截这道题目出自 NOIp 1999，容易看出第一问是一个裸的 LIS 问题（最长不升子序列就是 LIS 的一个变形而已）。可以直接套上 LIS 的板子看看能不能拿到 100 分。



课堂笔记

Note on the text

3.2 最长公共子序列—LCS

3.2.1 介绍与实现

定义 3.2.1 (最长公共子序列) 一个数列，如果分别是两个或多个已知数列的子序列，且是所有符合此条件序列中最长的，则称为已知序列的最长公共子序列。在计算机科学中，最长递增子序列是指，在一个给定的数值序列中，找到一个子序列，使得这个子序列元素的数值依次递增，并且这个子序列的长度尽可能地大。

以上是对最长公共子序列问题的定义。理解一下。

考虑一下这种题目如何打暴力。

非常容易，对吧，就是暴力枚举第一个序列的每一个子序列，对每一个子序列判断它是否为第二个序列的子序列。然而这样的时间复杂度是 $O(2^{\min(n,m)})$ 的，其中 n, m 为字符串的长度。

思考这样一个性质：如果一个 LCS 是另一个 LCS 的前缀，那么一定存在一个 LCS，使得它的长度为这两个 LCS 长度之和。

定理 3.2.1 (LCS 的最优子结构) 令 $X = \langle x_1, x_2, \dots, x_m \rangle$ 和 $Y = \langle y_1, y_2, \dots, y_n \rangle$ 为两个序列， $Z = \langle z_1, z_2, \dots, z_k \rangle$ 为 X 和 Y 的任意 LCS。

1. 如果 $x_m = y_n$ ，则 $z_k = x_m = y_n$ 。且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个 LCS。
2. 如果 $x_m \neq y_n$ ，那么 $x_m \neq y_n$ 意味着 Z 是 X_{m-1} 和 Y 的一个 LCS。
3. 如果 $x_m \neq y_n$ ，那么 $z_k \neq y_n$ 意味着 Z 是 X 和 Y_{n-1} 的一个 LCS。

举个例子：

	j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A	
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

上图是求 $X = \langle A, B, C, D, B, D, A, B \rangle$ 和 $B = \langle B, D, C, A, B, A \rangle$ 的 LCS 的过程。

证明 (1) 如果 $z_k \neq x_m$, 那么可以将 $x_m = y_n$ 追加到 Z 的末尾, 得到 X 和 Y 的一个长度为 $k+1$ 的公共子序列。与 Z 是 X 和 Y 的最长公共子序列的假设矛盾。因此, 必然有 $z_k = x_m = y_n$ 。这样, 前缀 Z_{k-1} 是 X_{m-1} 和 T_{n-1} 的一个长度为 $k-1$ 的公共子序列。我们希望证明它是一个 LCS。利用反证法, 假设存在 X_{m-1} 和 Y_{n-1} 的一个长度大于 $k-1$ 的公共子序列 W , 则将 $x_m = y_n$ 追加到 W 的末尾会得到 X 和 Y 的一个长度大于 k 的公共子序列, 矛盾。

(2) 如果 $z_k \neq x_m$, 那么 Z 是 X_{m-1} 和 Y 的一个公共子序列。如果存在 X_{m-1} 和 Y 的一个长度大于 k 的公共子序列 W , 那么 W 也是 X_m 和 Y 的公共子序列, 与 Z 是 X 和 Y 的最长公共子序列的假设矛盾。

(3) 与情况 (2) 对称。 ■

于是我们 get 到了 LCS 的一个很重要的性质, 没有后效性, 具有最优子结构的性质。这是 LCS 问题可以 DP 的基础。

很容易看出, 最初我们写的暴力求解算法中, 有大量重叠的子问题 (考虑求子序列 $X = \langle x_a, x_{a+1}, \dots, x_b \rangle$ 的 LCS, 必定求这个子序列的子序列 $Y = \langle x_{a+m}, x_{a+m+1}, \dots, x_{a+n} \rangle$ 这个子序列的 LCS), 所以 LCS 问题也有重叠子问题性质。

由上述定理我们就可以写出一个递归解。刚才的暴力中我们是枚举每一个子序列, 然后进行判断, 现在我们考虑能否直接构造出这样的一个序列。

明显的, 状态转移有以下两种:

- 如果两位相等, 那么新的 LCS 可以连接到旧的 LCS 之后, 构成一个新的较长的 LCS。
- 如果两位不相等, 那么新的 LCS 不变, 还是前一位的最长的 LCS。

设状态 $f[i][j]$ 表示到字符串 X 的第 i 位与字符串 Y 的第 j 位的 LCS 长度, 则:

$$f[i][j] = \begin{cases} f[i-1][j-1] + 1 & , X[i] = Y[j] \\ \max(f[i][j-1], f[i-1][j]) & , X[i] \neq Y[j]. \end{cases}$$

根据上面的状态转移方程写出代码即可, 时间复杂度 $O(nm)$.

3.2.2 LCS 的打印

同 LIS 的打印，记录一下转移的方向，然后就可以了，时间复杂度为 $O(nm)$ 。

3.2.3 LCS 的优化

考虑一下是不是有什么地方还可以优化。

考虑上一节说的 LCS 的打印的过程，也可以不记录下转移的方向，因为每个状态转移的方向都是固定的，所以可以在 $O(1)$ 的时间内判断出一个状态是由三个状态中的哪一个转移而来的，因此不再需要 $\text{pre}[]$ 数组，只需要在输出过程中逆序转移，判断一下转移方向，把每一个数丢到栈里，再依次输出即可。

对于不需要输出 LCS 的问题，还可以使用滚动数组优化一下 f 数组的空间占用。因为每次转移只用到了 f 数组当前的一行以及前一行，所以只需要保留这两行数据即可。但是如果需要计算 LCS 的元素，就不能使用滚动数组优化（思考一下为什么）。

例题的话，可以参考一下洛谷 P1439 的前 50%，至于后面的优化，建议参考一下题解，因为严格意义上这个优化意义不是很大，常数也很大，不是那么实用。



4 动态规划初步——背包问题

这一部分，我们将开始对背包问题的研究。

准备好了吗？

限于篇幅以及我的姿势水平，在这一部分里我们仅研究最基础的动态规划问题。本人的目的不是想让你们通过这节课精通 DP，而是想让你们对于 DP 有一个初步的了解，会做基本的 DP 题，并懂得举一反三，更重要的是理解 DP 的思想。

4.1 背包问题是什么？

定义 4.1.1 (背包问题 (Knapsack problem)) 是一种组合优化的 NP 完全问题。问题可以描述为：给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。

以上就是百度百科对背包问题的定义，所谓“NP 完全问题”是指这个问题具有多项式解法，即时间复杂度可以写成多项式形式 $O(n^k)$, $n \in \mathbb{N}_+$ 且 k 为已知常数)。背包问题本质上就是一种最优化组合问题，符合 DP 的要求，所以 DP 可以完成对背包问题的求解。时间复杂度是 $O(n^2)$ ，空间复杂度是 $O(n^2)$ ，优化后可以做到 $O(n)$ 。

背包问题可以分为以下 8 类：

- 01 背包问题
- 完全背包问题
- 多重背包问题
- 混合背包问题
- 二维背包问题
- 分组背包问题
- 有依赖的背包问题
- 泛化物品

每种问题都有类似的做法，同时也有它们各自的不同之处。下面我们将从最基础的 01 背包开始讲起，它非常重要，是后面所有背包问题的基础，一定要掌握。

4.2 01 背包问题

4.2.1 简介

定义 4.2.1 (01 背包问题) 一个背包总容量为 V , 现在有 N 个物品, 第 i 个物品体积为 $weight[i]$, 价值为 $value[i]$, 现在往背包里面装东西, 怎么装能使背包的内物品价值最大?

以上就是 01 背包的定义了。

看到此类问题, 我们的第一反应大概就是贪心了。然而贪心是肯定不行的。

为什么不行? 举一个例子:

例题 4.2.1 最少硬币找零问题: 给予不同面值的硬币若干种 (每种硬币个数无限多), 用若干种硬币组合为某种面额的钱, 使硬币的个数最少。

在现实生活中, 我们往往使用的是贪心算法, 比如找零时需要 13 元, 我们先找 10 元, 再找 2 元, 再找 1 元。这是因为现实生活中的硬币 (纸币) 种类特殊。如果我们的零钱可用的有 1、2、5、9、10。我们找零 18 元时, 贪心算法的策略是: $10+5+2+1$, 四种, 但是明明可以用两个 9 元的啊。所以肯定不能用贪心。

那么我们该怎么办? dalao 们思考两分钟, 不然我们先想一想搜索的方法吧 (如果秒掉了的话就看看下面的图)。



例题 4.2.2 01 背包问题：有 n 个重量和价值分别为 $w[i]$ 和 $v[i]$ 的物品。从这些物品中挑出总重量不超过 W 的物品，求所有挑选方案中价值总和的最大值。

4.2.2 搜索

相信 dalao 们都能一眼秒掉这个问题。利用搜索，我们可以很快解决这个问题。代码如下：

1. `void dfs(int index,int sumw,int sumv)//index: 数组下标, sumw: 当前情况的物品体积,
→ sumv: 当前情况的物品价值
{
 if(index>n)
 {
 if(sumw<=T&&sumv>maxvalue)
 maxvalue=sumv;
 return;
 }
 dfs(index+1,sumw+w[index],sumv+v[index]);//选
 dfs(index+1,sumw,sumv); //不选`

2. `int W, n;
int w[MAXN], v[MAXN];
int dfs(int i, int j)
{
 int res;
 if(i == n) res = 0;
 else if(j < w[i]) res = dfs(i+1, j);
 else res = max(dfs(i+1, j), dfs(i+1, j-w[i])+v[i]);
 return res;
}`

然而以上代码有一个问题。时间复杂度太高， $O(2^n)$ 显然不能满足我们的要求，不能只过 $n \leq 15$ 的这部分数据啊。

考虑优化一下。记忆化搜索？

```
int W, n;  
int w[MAXN], v[MAXN];  
int dp[MAXN][MAXN];  
int dfs(int i, int j)  
{
```

```

if(dp[i][j] >= 0) return dp[i][j];
int res;
if(i == n) res = 0;
else if(j < w[i]) res = dfs(i+1, j);
else res = max(dfs(i+1, j), dfs(i+1, j-w[i])+v[i]);
return dp[i][j] = res;
}

```

开一个二维数组记录一下每次搜索到的结果，这样时间复杂度就降低到了 $O(nW)$ ，似乎不错。

4.2.3 二维 DP

不然把上面的那个递归转化一下，转化成递推？简化一下代码？

设状态 $dp[i+1][j]$ 表示从前 i 个物品挑选出总重量超过 j 的物品时，背包中的最大价值，那么根据题意，我们有以下递推式（就是状态转移方程）：

$$dp[i+1][j] = \begin{cases} dp[i][j] & , j < w[i] \\ \max\{dp[i+1][j], dp[j - w[i]] + v[i]\} & , \text{其他情况.} \end{cases}$$

我来解释一下上面的状态转移方程。首先，要明确状态以及状态之间的转移过程。容易看到，在 DFS 中状态有两个参数 i, j ，分别代表 当前是第几个物品 以及 当前背包的剩余容量。所以 DP 也应该有两个状态，含义相同。如果背包容量不足了，那么就不取，价值不变；否则有两种决策：

- 取当前物品，背包剩余容量减当前物品的体积，当前所获得价值加当前物品的价值；
- 不取当前物品，考虑下一个物品。

在这两种决策中取一个最优解即可。DP 同理，只是将决策写成递推式的形式，本质和搜索是相同的。

所以我们得到以下代码：

```

for(int i = 0; i < N; ++i)
    for(int j = 0; j <= W; ++j)
        if(j < w[i])
            dp[i][j] = dp[i + 1][j];
        else
            dp[i][j] = max(dp[i + 1][j], dp[i + 1][j - w[i]] + v[i]);

```

重要提示: 在 DP 中有一点非常重要, 那就是 对状态数组的初始化, 万万不可忽略! 在本题中, 如果我们将数组开成全局变量, 那么就不需要(而不是可以忽略!)对状态数组的初始化, 因为初始状态什么都没有放, 价值都为 0, 同时全局变量默认的初始值都是 0。这只是一个个例! 很多题目都是需要对状态数组进行初始化的。在做 DP 题时, 一定要先对初始状态进行初始化, 否则你极有可能获得 0 分, 即使你的状态转移方程没有推错!

4.2.4 一维 DP

考虑将前面的二维 DP 优化一下。

好像时间复杂度已经是最优化了, 但是空间复杂度好像还有待提升。

看看刚才的状态转移方程或者代码。转移前后的状态有什么关系?

共同之处在于: $dp[i][j] = dp[i + 1][Something]$, 通过这个性质, 我们可以去掉这个方程的第一维, 并且不取新的物品与背包没有空间时的情况相同, 所以我们可以得到以下状态转移方程:

$$dp[i] = \max\{dp[i], dp[i - w[j]] + v[j]\}, i \geq w[j]$$

时间复杂度: $O(n^2)$, 空间复杂度 $O(n)$ 。

这个状态转移方程非常重要, 在以后的学习中还会多次用到。

有什么问题随时问我就好, 我会很乐意帮忙的。有不会的必须弄明白, 因为 01 背包问题是所有其他背包问题的基础。

重要提示: 请特别注意此题的枚举顺序, 由于 $dp[i]$ 是由 $dp[i - w[j]]$ 转移来的, 为了避免后效性需要倒序枚举背包的容量, 这也是压维优化 DP 的特征(但不是所有的压维优化 DP 都需要倒序枚举)。

课堂笔记
Note on the text

4.3 完全背包问题、多重背包问题与混合背包问题

这一节内容看起来会比较多，但是因为有了前面一节的基础，相信大家也能够很容易的掌握。

定义 4.3.1 (完全背包问题) 一个背包总容量为 V ，现在有 N 个物品，第 i 个物品体积为 $weight[i]$ ，价值为 $value[i]$ ，每个物品都有无限多件，现在往背包里面装东西，怎么装能使背包的内物品价值最大？

定义 4.3.2 (多重背包问题) 一个背包总容量为 V ，现在有 N 个物品，第 i 个物品体积为 $weight[i]$ ，价值为 $value[i]$ ，每个物品有 $num[i]$ ($num[i] \geq 1$) 件，现在往背包里面装东西，怎么装能使背包的内物品价值最大？

定义 4.3.3 (混合背包问题) 一个背包总容量为 V ，现在有 N 个物品，第 i 个物品体积为 $weight[i]$ ，价值为 $value[i]$ ，有的物品只有一件，有的物品有 $num[i]$ ($num[i] \geq 1$) 件，有的物品有无限多件，现在往背包里面装东西，怎么装能使背包的内物品价值最大？

以上是对这三种背包问题的定义，希望大家能够理解。可以发现这三种问题非常类似，也有不同。下面我就逐一来讲解。

4.3.1 完全背包问题

还记得上一节讲的 01 背包的一维状态转移方程吗？当时我曾经说过“为了避免后效性需要倒序枚举”，完全背包问题因为每个物品都有无限多件，所以可以利用这个后效性，正序枚举即可。

完全背包问题就讲完了。

4.3.2 多重背包问题

对于多重背包问题，我们可以考虑每一件物品都拆成 01 背包来做， N 种物品，分别枚举每种物品的每一件即可。

就是加一层循环，枚举每一种物品中的每一件的状态。代码如下：

```
for(int i=0;i<N;i++)
    for(int j=0;j<num[i];j++)
        for(int k=V;k>=v[i];k--)
            dp[k]=max(dp[k],dp[k-v[i]]+w[i]);
```

这里稍微加了一点常数优化，注意最内层的循环，没有从 V 枚举到 0，因为背包容量小于物品体积时对答案没有任何贡献（本来就放不下，当然不能硬塞）。

4.3.3 混合背包问题

这种问题就是把前面三种背包问题混合起来，所以我们只需要把前面三种背包都写一遍（实际上只要写完全背包和多重背包即可），然后加个判断就行了。

以下代码假设物品数量已经读入，第 i 个物品的数量用 $\text{num}[i]$ 表示，且当 $\text{num}[i]=-1$ 时，表示这个物品有无限多件：

```
for(int i=0;i<N;i++)
{
    if(num[i]!=-1)
    {
        for(int j=0;j<num[i];j++)
            for(int k=V;k>=v[i];k--)
                dp[k]=max(dp[k],dp[k-v[i]]+w[i]);
    }
    else
    {
        for(int j=v[i];j<=V;j++)
            dp[j]=max(dp[j],dp[j-v[i]]+w[i]);
    }
}
```

这三种问题是不是很简单呢？有不会的问题可以提问。

课堂笔记 Note on the text

4.4 二维背包问题与分组背包问题

4.4.1 二维背包问题

定义 4.4.1 (二维背包问题) 对于每件物品，具有两种不同的费用；选择这件物品必须同时付出这两种代价；对于每种代价都有一个可付出的最大值（背包容量）。问怎样选择物品可以得到最大的价值。

定义比较长，但是事实上就是背包问题的再次综合。状态加一维表示第二种代价，然后套用适当的背包模板即可。注意最好使用一维的 DP 进行状态转移，否则可能会爆空间。

设第 i 件物品所需的两种代价分别为 $a[i]$ 和 $b[i]$ 。两种代价可付出的最大值（两种背包容量）分别为 u 和 v 。第 i 件物品的价值为 $value[i]$ 。

三维状态转移方程： $f[i][u][v] = \max(f[i-1][u][v], value[i] + f[i-1][u-a[i]][v-b[i]])$

二维状态转移方程（注意需要倒序枚举）： $f[i][j] = \max(f[i][j], value[k] + f[u-a[k]][v-b[k]])$

时间复杂度： $O(n^3)$ ，空间复杂度 $O(n^2)$ 。

4.4.2 分组背包问题

定义 4.4.2 (分组背包问题) 有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用是 $w[i]$ ，价值是 $v[i]$ 。这些物品被划分为 k 组，每组中的物品互相冲突，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，且价值总和最大。

这种问题看起来与前面的问题不同。但实质是相同的。只不过枚举的对象由每一个物品变成每一类物品，然后对每一类物品分别枚举其中的每一个物品即可。

设枚举对象 x 为第 $1 \sim k$ 类物品， i 为第 x 类物品中的第 i 件物品：

状态转移方程： $f[i] = \max\{f[i], f[i - w[i]] + v[i]\}, i \in k_x$. 有问题随时间。

课堂笔记

Note on the text

4.5 有依赖的背包问题与泛化物品（选学）

以下内容为选学，掌握与否都不很重要。讲解以经典例题为主，不能光讲基础的东西不讲题啊。

4.5.1 有依赖的背包问题

例题 4.5.1 金明的预算方案

题目描述

金明今天很开心，家里购置的新房就要领钥匙了，新房里有一间金明自己专用的很宽敞的房间。更让他高兴的是，妈妈昨天对他说：“你的房间需要购买哪些物品，怎么布置，你说了算，只要不超过 N 元钱就行”。今天一早，金明就开始做预算了，他把想买的物品分为两类：主件与附件，附件是从属于某个主件的，下表就是一些主件与附件的例子：

主件	附件
电脑	打印机、扫描仪
书柜	图书
书桌	台灯、文具
工作椅	无

如果要买归类为附件的物品，必须先买该附件所属的主件。每个主件可以有 0 个、1 个或 2 个附件。附件不再有从属于自己的附件。金明想买的东西很多，肯定会超过妈妈限定的 N 元。于是，他把每件物品规定了一个重要度，分为 5 等：用整数 1~5 表示，第 5 等最重要。他还从因特网上查到了每件物品的价格（都是 10 元的整数倍）。他希望在不超过 N 元（可以等于 N 元）的前提下，使每件物品的价格与重要度的乘积的总和最大。

设第 j 件物品的价格为 $v[j]$ ，重要度为 $w[j]$ ，共选中了 k 件物品，编号依次为 j_1, j_2, j_k ，则所求的总和为：

$$v[j_1] \times w[j_1] + v[j_2] \times w[j_2] + \cdots + v[j_k] \times w[j_k]$$

请你帮助金明设计一个满足要求的购物单。

输入输出格式

输入格式

输入的第 1 行，为两个正整数，用一个空格隔开：

N m (其中 N ($N < 32000$) 表示总钱数, m ($m < 60$) 为希望购买物品的个数。)

从第 2 行到第 $m+1$ 行, 第 j 行给出了编号为 $j-1$ 的物品的基本数据, 每行有 3 个非负整数

v p q (其中 v 表示该物品的价格 ($v < 10000$), p 表示该物品的重要度 (1~5), q 表示该物品是主件还是附件。如果 $q=0$, 表示该物品为主件, 如果 $q>0$, 表示该物品为附件, q 是所属主件的编号)

输出格式

输出只有一个正整数, 为不超过总钱数的物品的价格与重要度乘积的总和的最大值 (不超过 200000)。

输入输出样例

输入样例

```
1000 5
800 2 0
400 5 1
300 5 1
400 3 0
500 2 0
```

输出样例

```
2200
```

这道例题是 NOIp2006 的提高组第二题。结合之前的讲解, dalao 们有思路吗? 先思考 2 分钟。

其实很简单, 考虑状态:

- 主件和两个附件都不选
- 只选择主件, 不选择任何一个附件
- 选择主件并选择附件 1
- 选择主件并选择附件 2
- 选择主件和两个附件

共计 5 个状态，所以大概需要写 5 个状态转移方程。

思路和正常向的 01 背包一致。

设每一件物品的属性：

- m：主件体积
- a：附件 1 体积
- b：附件 2 体积
- x：主件价值
- y：附件 1 价值
- z：附件 2 价值

则我们可以得到：

$$\begin{aligned}f[j] &= \max(f[j], f[j - m[i]] + x[i]) \\f[j] &= \max(f[j], f[j - m[i] - a[i]] + x[i] + y[i]) \\f[j] &= \max(f[j], f[j - m[i] - b[i]] + x[i] + z[i]) \\f[j] &= \max(f[j], f[j - m[i] - a[i] - b[i]] + x[i] + y[i] + z[i])\end{aligned}$$

唯一需要注意的是边界条件（不能出现访问越界的错误）。

4.5.2 泛化物品

例题 4.5.2 最佳课题选择

描述

*Matrix67*要在下个月交给老师 n 篇论文，论文的内容可以从 m 个课题中选择。由于课题数有限，*Matrix67*不得不重复选择一些课题。完成不同课题的论文所花的时间不同。具体地说，对于某个课题 i ，若 *Matrix67*计划一共写 x 篇论文，则完成该课题的论文总共需要花费 $A_i * x^{B_i}$ 个单位时间（系数 A_i 和指数 B_i 均为正整数）。给定与每一个课题相对应的 A_i 和 B_i 的值，请帮助 *Matrix67*计算出如何选择论文的课题使得他可以花费最少的时间完成这 n 篇论文。

格式

输入格式

第一行有两个用空格隔开的正整数 n 和 m , 分别代表需要完成的论文数和可供选择的课题数。

以下 m 行每行有两个用空格隔开的正整数。其中, 第 i 行的两个数分别代表与第 i 个课题相对应的时间系数 A_i 和指数 B_i 。

输出格式

输出完成 n 篇论文所需要耗费的最少时间。

样例

样例输入

```
10 3
2 1
1 2
2 1
```

样例输出

```
19
```

数据范围

对于 30% 的数据, $n \leq 10, m \leq 5$;

对于 100% 的数据, $n \leq 200, m \leq 20, A_i \leq 100, B_i \leq 5$ 。

这题非常经典, 是泛化物品问题的一道好题。

首先给出泛化物品的定义:

定义 4.5.1 (泛化物品) 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 $h(v)$ 。求能获得的最大价值。

这题实际上不是很难, 因为每件物品的数量都只能取整数, 即函数 $h(x)$ 的定义域为 $x \in \mathbb{N}_+$ 。所以我们只需要先预处理出 $h(x)$ 在适当范围内的取值, 然后套用适当的背包模板即可。

注意状态转移方程中加的部分是预处理出来的数组中对应的函数值。

课堂笔记

Note on the text

5 动态规划提高——棋盘型 DP 与区间型 DP

让我们再来看看稍微难一些的 DP，这一部分先初步了解一下就行了，只要会了 DP 的思想，几乎所有的 DP 问题都能解决。

5.1 棋盘型 DP

例题 5.1.1 *Likecloud-吃、吃、吃*

题目背景

问世间，青春期为何物？

答曰：“甲亢，甲亢，再甲亢；挨饿，挨饿，再挨饿！”

题目描述

正处在某一特定时期之中的李大水牛由于消化系统比较发达，最近一直处在饥饿的状态中。某日上课，正当他饿得头昏眼花之时，眼前突然闪现出了一个 $n \times m (nm \leq 200)$ 的矩型的巨型大餐桌，而自己正处在这个大餐桌的一侧的中点下边。餐桌被划分为了 $n*m$ 个小方格，每一个方格中都有一个圆形的巨型大餐盘，上面盛满了令李大水牛朝思暮想的食物。李大水牛已将餐桌上所有的食物按其所能提供的能量打了分（有些是负的，因为吃了要拉肚子），他决定从自己所处的位置吃到餐桌的另一侧，但他吃东西有一个习惯——只吃自己前方或左前方或右前方的盘中的食物。

由于李大水牛已饿得不想动脑了，而他又想获得最大的能量，因此，他将这个问题交给了你。

每组数据的出发点都是最后一行的中间位置的下方！

输入输出格式

输入格式：

第一行为 $m\ n.$ (n 为奇数)，李大水牛一开始在最后一行的中间的下方，接下来为 $m \times n$ 的数字矩阵。

共有 m 行，每行 n 个数字，数字间用空格隔开，代表该格子上的盘中的食物所能提供的能量。数字全是整数。

输出格式：

一个数，为你所找出的最大能量值。

输入输出样例

输入样例

6 7
16 4 3 12 6 0 3
4 -5 6 7 0 0 2
6 0 -1 -2 3 6 8
5 3 4 0 0 -2 7
-1 7 4 0 7 -5 6
0 -1 3 4 12 4 2

输出样例

41

棋盘型 DP 大概就是给你一个棋盘，有一个人从某一个点出发，要求走到某一个点的什么东西。

针对这个题目，只需要设计出状态和方程就可以了。这个题目非常容易？设 $dp[i][j]$ 表示当前吃到的格子能获得的最大能量，每一个格子都是由其左上方、正上方和右上方的格子转移而来。然后就可以轻易的得到状态转移方程：

$$f[i][j] = \max\{\max\{f[i-1][j-1], f[i-1][j]\}, f[i-1][j+1]\} + map[i][j]$$

这就完了。

5.2 区间型 DP

例题 5.2.1 [USACO06FEB] Treats for the Cows 奶牛零食

题目描述

FJ has purchased N ($1 \leq N \leq 2000$) yummy treats for the cows who get money for giving vast amounts of milk. FJ sells one treat per day and wants to maximize the money he receives over a given period time.

The treats are interesting for many reasons: The treats are numbered $1..N$ and stored sequentially in single file in a long box that is open at both ends. On any day, FJ can retrieve one treat from either end of his stash of treats. Like fine wines and delicious cheeses, the treats improve with age and command greater prices. The treats are not uniform: some are better and have higher intrinsic value. Treat i has value $v(i)$ ($1 \leq v(i) \leq 1000$). Cows

*pay more for treats that have aged longer: a cow will pay $v(i)*a$ for a treat of age a . Given the values $v(i)$ of each of the treats lined up in order of the index i in their box, what is the greatest value FJ can receive for them if he orders their sale optimally?*

The first treat is sold on day 1 and has age $a=1$. Each subsequent day increases the age by 1.

约翰经常给产奶量高的奶牛发特殊津贴，于是很快奶牛们拥有了大笔不知该怎么花的钱。为此，约翰购置了 $N(1 \leq N \leq 2000)$ 份美味的零食来卖给奶牛们。每天约翰售出一份零食。当然约翰希望这些零食全部售出后能得到最大的收益。这些零食有以下这些有趣的特性：

- 零食按照 $1 \dots N$ 编号，它们被排成一列放在一个很长的盒子里。盒子的两端都有开口，约翰每天可以从盒子的任一端取出最外面的一个。
- 与美酒与好吃的奶酪相似，这些零食储存得越久就越好吃。当然，这样约翰就可以把它们卖出更高的价钱。
- 每份零食的初始价值不一定相同。约翰进货时，第 i 份零食的初始价值为 $V_i (1 \leq V_i \leq 1000)$ 。
- 第 i 份零食如果在被买进后的第 a 天出售，则它的售价是 $v_i \times a$ 。

V_i 的是从盒子顶端往下的第 i 份零食的初始价值。约翰告诉了你所有零食的初始价值，并希望你能帮他计算一下，在这些零食全被卖出后，他最多能得到多少钱。

输入输出格式

输入格式：

第 1 行：一个整数 N 。

第 $2 \sim N+1$ 行：第 $i+1$ 行包含零食 i 的价值 v_i 。

输出格式：

一行一个数表示 FJ 在这些零食全被卖出后最多能得到的钱数。

输入输出样例

输入样例：

5
1
3

1

5

2

输出样例：

43

说明

样例解释：五个零食，在第一天 FJ 可以卖出 1 号零食（价值为 1）或者 5 号零食（价值为 2）。FJ 按 1,5,2,3,4 顺序出售零食，获得的收益为 $1 \times 1 + 2 \times 2 + 3 \times 3 + 4 \times 1 + 5 \times 5 = 43$ 。

这题是个很经典的区间 DP，区间 DP 的一般套路为设状态 $dp[l][r]$ 为左端点为 l，右端点为 r 区间 [l,r] 的值。

初始状态为： $f[i][i]$ 为自己第一天被取也就是 $v[i]*n$ ，尝试往左右拓展即可。边界条件为取完所有元素（即序列为空的状态）。

状态转移方程： $f[l][r] = \max\{f[l][r-1] + v[r] * (n - i + 1), f[l+1][r] + v[l] * (n - i + 1)\}$.

于是这两种 DP 就都讲完了，相信大家对 DP 有了更深一步的了解。下面我们将学习更复杂的 DP，可能是省选难度的 DP 了（不存在的，NOIp 2017 就考了一道状压 DP）。



课堂笔记

Note on the text

6 动态规划 Professional——树形 DP、状压 DP 与数位 DP

这一部分让我们看看非常水的丧心病狂的 DP 类型，在 NOIp 2017 中首次出现了以前只出现在省选等高水平比赛中的 DP 类型——状压 DP，这种类型的 DP 本质上是枚举状态（和搜索一样），但是又具有 DP 的某些特性。所以比较困难。树形 DP 则是另外一种不是很常见的 DP 类型，就是将 DP 的位置由矩阵上转移到树上来做。最后是最少见的数位 DP。其实也不是非常难。

6.1 树形 DP

定义 6.1.1 (树形 DP) 就是把普通的 DP 转移到树上来做，转移方向就是由普通的在数组转移变为在树上借助边转移。其他的内容与普通 DP 无异。唯一需要注意的是先 **DP 儿子节点**。

以上是对树形 DP 的定义，可以看出来它与普通的 DP 没什么区别。

为什么要先对儿子进行 DP 呢？其实非常简单，树的结构是由下而上越来越小的，如果先 DP 父亲节点，那么由于儿子节点还没有被更新，仍然保持初始值 0，自然无法更新父亲节点。所以要先对较小规模的问题求解，即先处理依赖关系较少的节点（状态转移的基本性质，未知量由已知量转移而来）。

下面通过一道题目来说明一下树形 DP 的基本过程。

例题 6.1.1 选课

题目描述

在大学里每个学生，为了达到一定的学分，必须从很多课程里选择一些课程来学习，在课程里有些课程必须在某些课程之前学习，如高等数学总是在其它课程之前学习。现在有 N 门功课，每门课有个学分，每门课有一门或没有直接先修课（若课程 a 是课程 b 的先修课即只有学完了课程 a ，才能学习课程 b ）。一个学生要从这些课程里选择 M 门课程学习，问他能获得的最大学分是多少？

输入输出格式

输入格式：

第一行有两个整数 N, M 用空格隔开。 $(1 \leq N \leq 300, 1 \leq M \leq 300)$

接下来的 N 行，第 $I+1$ 行包含两个整数 ki 和 si , ki 表示第 I 门课的直接先修课, si 表示第 I 门课的学分。若 $ki=0$ 表示没有直接先修课 ($1 \leq ki \leq N$, $1 \leq si \leq 20$)。

输出格式：

只有一行，选 M 门课程的最大得分。

输入输出样例

输入样例：

```
7 4
2 2
0 1
0 4
2 1
7 1
7 6
2 2
```

输出样例：

```
13
```

题目大意是给定一个有依赖关系的课程安排表，一个课程可能依赖于多个课程，也可能被多个课程依赖，每学一个课程都能获得一些学分，最多学习 M 个课程，求能获得最大学分。

很明显，这是一个 DAG (有向无环图)，也就是一棵树。依赖关系可能不连续，也就是说给定的是一个森林。

对于森林的问题，我们可以通过虚拟一个总根节点，森林中所有的根节点都向这个总根节点连一条无向边，然后这个虚拟节点上的答案就是问题的答案。

因为树形数据结构是递归定义的（父节点与儿子节点，递归起点可以看做是根节点或某一个叶子节点，都可以遍历完整棵树），所以树形 DP 的基本思路应该是对树的每一个节点递推地计算该点的最优解，需要使用到 DFS。

这道题的状态转移方程不难想到，设状态 $f[i][j]$ 表示第 i 个节点取 j 个子节点（不包括自己）所取得的最大（最小）收益。然后标准状态转移模板就是 $f[i][j] = \max(f[i][j], f[i][j - k] + f[\text{儿子节点编号}][k])$

现在考虑一下一棵子树的根节点可能会有多个儿子的情况。传统上有

两种解决方案，第一种就是直接 DFS，DP 需要用到子树的时候就 DFS 这棵子树，不去考虑儿子的个数的问题。

这种做法的代码如下：

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;
int n,m,f[2005][2005];
int head[2005],next[2005],w[2005];
int dfs(int x){
    if (head[x]==-1) return 0;
    int sum=0;
    for (int i=head[x];i!=-1;i=next[i])
    {
        int t=dfs(i);
        sum+=t+1;
        for (int j=sum;j>=0;j--)
        {
            for (int k=0;k<=t;k++)
                if (j-k-1>=0) f[x][j]=max(f[x][j],f[x][j-k-1]+f[i][k]);
        }
    }
    return sum;
}
int main()
{
    scanf("%d%d",&n,&m);
    memset(f,0,sizeof(f));
    memset(head,-1,sizeof(head));
    for (int i=1;i<=n;i++)
    {
        int a;
        scanf("%d%d",&a,&w[i]);
        next[i]=head[a];
        head[a]=i;
    }
    for (int i=1;i<=n;i++) f[i][0]=w[i];
    f[0][0]=0;
    dfs(0);
    printf("%d",f[0][m]);
    return 0;
}
```

另外一种做法是多叉树转为二叉树来做。多叉树转二叉树在树这一节内容 lpy 学姐已经给你们讲过了，就是左儿子右兄弟表示法，我就不讲了，

直接说思路。

用 $f[root][k]$ 表示第 $root$ 节课为根且还剩 k 个自由选课数时，学分的最大值；

- 不选此课，它与它的后续值皆为零；
- 选此课，最终求 \max 【它的值+兄弟与左孩子瓜分剩余选课数时最优解，原状态，不选它（但它兄弟的状态要与它合并，因为左孩子只有一个】】

树形 DP 本质上是对这棵树的 DFS 操作。需要写个记忆化搜索优化一下。

于是就可以写出代码：

```
#include<cstdio>
#include<cstring>
using namespace std;
const int N=320;
int n,m;
int f[N][N],b[N],c[N],s[N];
void dp(int root,int k)
{
    if(f[root][k]>=0) return;
    if(root==0||k==0){f[root][k]=0;return;}
    dp(b[root],k); //兄弟与左孩子享有同等权利;
    for(int i=0;i<k;i++)
    {
        dp(c[root],k-i-1); //选第 root 门课;
        dp(b[root],i); //因为根节点给下一代指标时，只会给左孩子，所以下一代的左孩子要分给
        // 兄弟指标;
        f[root][k]=max(f[root][k], max(f[b[root]][k], f[b[root]][i] + f[c[root]][k - i
        // - 1] + s[root])));
        //求三个状态最大值: 选此状态(原封不动), 不选, 选此状态(更新, 此节点+瓜分指标后的兄
        // 弟与左孩子的最优解)
    }
}
int main()
{
    cin>>n>>m;
    int fa;
    for(int i=1;i<=n;i++)
    {
        cin>>fa>>s[i];
        if(fa==0)fa=n+1;
    }
}
```

```
b[i]=c[fa];//c[i] 记录的是长子，长子把‘长子’的位子让给 i, 原长子成为了新长子兄弟,  
c[fa]=i;//父亲直辖原长子;  
}  
memset(f,-1,sizeof(f));  
dp(c[n+1],m);  
cout<<f[c[n+1]] [m];//无先修课的的最优值会集中于 c[n+1];  
return 0;  
}
```



课堂笔记

Note on the text

6.2 状压 DP

6.2.1 定义与特点

定义 6.2.1 (状压 DP) 和普通的 DP一样，同样需要设计状态与状态转移方程，保存一些状态来相互转移的解题过程，与普通 DP 不同的是，状压 DP 的状态不是简单的状态，而是不同的方案等等。状态压缩中的状态常用二进制来表示，转移过程常用位运算来完成。

以上是我给状压 DP 的简单定义。或许不是非常准确，但是基本能反映出状压 DP 的含义了。如果不是特别理解的话，还是联想 DP 最初的定义，就是一个无限优化的搜索。只不过这个优化有点特殊，表示的状态与常见的 DP 有些不同。

状压 DP 的特点：**数据范围非常小，一般有 $n \leq 24$** ，因为状压 DP 需要枚举每一个子集。

学习状压 DP 前首先需要了解位运算的相关知识。让我们先来看一下。

6.2.2 有关位运算的知识

设有一个集合 S ，用一个二进制数表示集合中每一个元素的选择情况（第 i 个二进制位为 0 表示不选集合 S 中的第 i 个元素，当然这个集合不具有数学上集合的无序性，要求是按编号有序排列的），则这个二进制数表示的是集合 S 的一个子集。

状压 DP 就是枚举子集，转移常常涉及对并集，交集、补集的操作。可以使用位运算来进行加速对集合的运算。

设有两个集合 $S_1 \subset S, S_2 \subset S$ ，则我们有以下集合运算与位运算的关系：

集合运算	与之等价的位运算
$S_1 \cap S_2$	$S_1 \& S_2$
$S_1 \cup S_2$	$S_1 \mid S_2$
$\complement_{S_1} S_2$	$S_1 \sim S_2$

状压的一些技巧：

- 若 S 是 U 的子集，则 S 关于 U 的补集为： $S \sim U$
- 判断点 k 是否在集合 S 中（即 S 的第 $k - 1$ 位是否为 1）：
 $S \& (1 << (k-1)) != 0 ? "Yes" : "No";$

- 枚举 S 的子集: `for(int i=S;i;i=(i-1)&S){...}`
- 设置一个大小为 n 的全集: `n=1<<(n-1)`

下面给出两道例题，再细致地谈一下状压 DP 的内容和实现细节。

6.2.3 基础例题—愤怒的小鸟

例题 6.2.1 愤怒的小鸟

问题描述

Kiana 最近沉迷于一款神奇的游戏无法自拔。

简单来说，这款游戏是在一个平面上进行的。

有一架弹弓位于 $(0, 0)$ 处，每次 *Kiana* 可以用它向第一象限发射一只红色的小鸟，小鸟们的飞行轨迹均为形如 $y = ax^2 + bx$ 的曲线，其中 a, b 是 *Kiana* 指定的参数，且必须满足 $a \leq 0$ 。

当小鸟落回地面（即 x 轴）时，它就会瞬间消失。

在游戏的某个关卡里，平面的第一象限中有 n 只绿色的小猪，其中第 i 只小猪所在的坐标为 (xi, yi) 。

如果某只小鸟的飞行轨迹经过了 (xi, yi) ，那么第 i 只小猪就会被消灭掉，同时小鸟将会沿着原先的轨迹继续飞行；

如果一只小鸟的飞行轨迹没有经过 (xi, yi) ，那么这只小鸟飞行的全过程就不会对第 i 只小猪产生任何影响。

例如，若两只小猪分别位于 $(1, 3)$ 和 $(3, 3)$ ，*Kiana* 可以选择发射一只飞行轨迹为 $y = -x^2 + 4x$ 的小鸟，这样两只小猪就会被这只小鸟一起消灭。

而这个游戏的目的，就是通过发射小鸟消灭所有的小猪。

这款神奇游戏的每个关卡对 *Kiana* 来说都很难，所以 *Kiana* 还输入了一些神秘的指令，使得自己能更轻松地完成这个游戏。这些指令将在【输入格式】中详述。

假设这款游戏一共有 T 个关卡，现在 *Kiana* 想知道，对于每一个关卡，至少需要发射多少只小鸟才能消灭所有的小猪。由于她不会算，所以希望由你告诉她。

【输入格式】

第一行包含一个正整数 T ，表示游戏的关卡总数。

下面依次输入这 T 个关卡的信息。每个关卡第一行包含两个非负整数 n, m ，分别表示该关卡中的小猪数量和 *Kiana* 输入的神秘指令类型。接下来

的 n 行中，第 i 行包含两个正实数 (xi, yi) ，表示第 i 只小猪坐标为 (xi, yi) 。
数据保证同一个关卡中不存在两只坐标完全相同的小猪。

如果 $m=0$ ，表示 Kiana 输入了一个没有任何作用的指令。

如果 $m=1$ ，则这个关卡将会满足：至多用 $\lceil \frac{n}{3} + 1 \rceil$ 只小鸟即可消灭所有小猪。

如果 $m=2$ ，则这个关卡将会满足：一定存在一种最优解，其中有一只小鸟消灭了至少 $\lfloor \frac{n}{3} \rfloor$ 只小猪。

保证 $1 \leq n \leq 18, 0 \leq m \leq 2, 0 \leq xi, yi \leq 10$ ，输入中的实数均保留到小数点后两位。

上文中，符号 $\lceil x \rceil$ 和 $\lfloor x \rfloor$ 分别表示对 c 向上取整和向下取整。

【输出格式】

对每个关卡依次输出一行答案。

输出的每一行包含一个正整数，表示相应的关卡中，消灭所有小猪最少需要的小鸟数量。

【输入样例 1】

```
2
2 0
1.00 3.00
3.00 3.00
5 2
1.00 5.00
2.00 8.00
3.00 9.00
4.00 8.00
5.00 5.00
```

【输出样例 1】

```
1
1
```

【输入样例 2】

```
3
2 0
1.41 2.00
1.73 3.00
3 0
1.11 1.41
2.34 1.79
```

```
2.98 1.49
5 0
2.72 2.72
2.72 3.14
3.14 2.72
3.14 3.14
5.00 5.00
```

【输出样例 2】

```
2
2
3
```

【输入样例 3】

```
1
10 0
7.16 6.28
2.02 0.38
8.33 7.78
7.68 2.09
7.46 7.86
5.77 7.44
8.24 6.72
4.42 5.11
5.42 7.79
8.15 4.99
```

输出样例 3

```
6
```

【说明】

【样例解释 1】

这组数据中一共有两个关卡。

第一个关卡与【问题描述】中的情形相同，2只小猪分别位于 $(1.00, 3.00)$ 和 $(3.00, 3.00)$ ，只需发射一只飞行轨迹为 $y = -x^2 + 4x$ 的小鸟即可消灭它们。

第二个关卡中有5只小猪，但经过观察我们可以发现它们的坐标都在抛物线 $y = -x^2 + 6x$ 上，故 Kiana 只需要发射一只小鸟即可消灭所有小猪。

【数据范围】

测试点编号	n	m	t
1	≤ 2	$= 0$	≤ 10
2			≤ 30
3	≤ 3		≤ 30
4			≤ 30
5	≤ 4		≤ 10
6			≤ 30
7	≤ 5		≤ 10
8	≤ 6		
9	≤ 7		≤ 10
10	≤ 8		
11	≤ 9		
12	≤ 10		≤ 30
13	≤ 12	$= 1$	≤ 30
14		$= 2$	
15		$= 0$	≤ 15
16	≤ 15	$= 1$	
17		$= 2$	
18		$= 0$	≤ 5
19	≤ 18	$= 1$	
20		$= 2$	

考慮一下，有思路嗎？

其实这个数据范围，暴力也可以过。大家不妨先写个暴力玩玩。

因为状态的数目不是很多，所以我们考虑用二进制数来表示抛物线打掉的猪的集合。用 0 和 1 表示是否选中这一只猪，若二进制的第 i 位为表示选第 i 只猪，为 1 表示不选。举个例子，若使用 32 位有符号整数，即 `int` 类型表示一个状态，则二进制数 `111111111111111111111111111100B` 表示的状态为打掉第一只猪和第二只猪所需抛物线的最少条数；而二进制数 `11111111111111000000000000000000B` 表示的状态为打掉第 1 到第 18 只猪所需抛物线的最少条数（注意有符号整数最高位是符号位，所以实际可用于表示集合的位数是 31，如果开了 `short` 类型就对应题目中前 17 个数据）。

设状态 $dp[S]$ 表示打掉集合 S (用二进制表示) 中所有的猪需要的最少小鸟数。状态的数目不是很多，可以考虑枚举每一条抛物线。一个贪心的

思路是使每条抛物线都至少打掉两只猪，可以证明这样做是最优的（明显的，一次打两只猪要优于一次打只个猪）。枚举两只猪，如果它们不在同一列，代入这两只猪的坐标，解得 a 和 b 的值（可以使用加减消元法）。如果这组 a 和 b 没有出现过，那么就用这组 a 和 b 代入每一只猪看能否打到。于是我们就预处理出了 m 条抛物线能够打到的猪的编号的集合 S ，存入 $a[]$ 数组。

我们的目标状态是打完所有的猪。也就是说答案是状态 $dp[0]$ 的值。边界条件是 $dp[2^n - 1] = 0$ 表示一头猪都不打的情况。然后由刚才预处理的 $a[]$ 数组，我们可以得到状态转移方程：

$$dp[i] = \min_{j=1}^m \{dp[i|a[j]], dp[i]\} + 1, i|a[j] < dp[i].$$

6.2.4 提高例题—宝藏

例题 6.2.2 宝藏

【问题描述】

参与考古挖掘的小明得到了一份藏宝图，藏宝图上标出了 n 个深埋在地下的宝藏屋，也给出了这 n 个宝藏屋之间可供开发的 m 条道路和它们的长度。

小明决心亲自前往挖掘所有宝藏屋中的宝藏。但是，每个宝藏屋距离地面都很远，也就是说，从地面打通一条到某个宝藏屋的道路是很困难的，而开发宝藏屋之间的道路则相对容易很多。

小明的决心感动了考古挖掘的赞助商，赞助商决定免费赞助他打通一条从地面到某个宝藏屋的通道，通往哪个宝藏屋则由小明来决定。

在此基础上，小明还需要考虑如何开凿宝藏屋之间的道路。已经开凿出的道路可以任意通行不消耗代价。每开凿出一条新道路，小明就会与考古队一起挖掘出由该条道路所能到达的宝藏屋的宝藏。另外，小明不想开发无用道路，即两个已经被挖掘过的宝藏屋之间的道路无需再开发。

新开发一条道路的代价是：

$$L \times K$$

L 代表这条道路的长度， K 代表从赞助商帮你打通的宝藏屋到这条道路起点的宝藏屋所经过的宝藏屋的数量（包括赞助商帮你打通的宝藏屋和这条道路起点的宝藏屋）。

请你编写程序为小明选定由赞助商打通的宝藏屋和之后开凿的道路，使得工程总代价最小，并输出这个最小值。

【输入格式】

第一行两个用空格分离的正整数 n 和 m ，代表宝藏屋的个数和道路数。

接下来 m 行，每行三个用空格分离的正整数，分别是由一条道路连接的两个宝藏屋的编号（编号为 $1 \sim n$ ），和这条道路的长度 v 。

【输出格式】

输出共一行，一个正整数，表示最小的总代价。

【输入输出样例】

【输入样例 1】

```
4 5
1 2 1
1 3 3
1 4 1
2 3 4
3 4 1
```

【输出样例 1】

```
4
```

【输入样例 2】

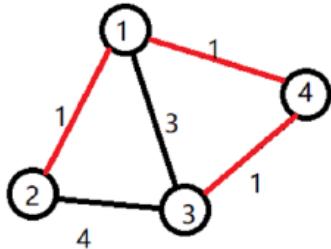
```
4 5
1 2 1
1 3 3
1 4 1
2 3 4
3 4 2
```

【输出样例 2】

```
5
```

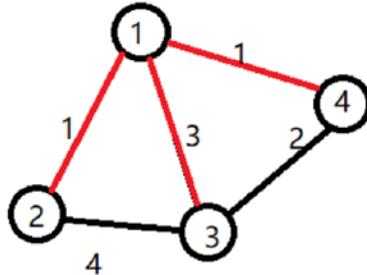
【说明】

【样例解释 1】



小明选定让赞助商打通了 1 号宝藏屋。小明开发了道路 $1 \rightarrow 2$, 挖掘了 2 号宝藏。开发了道路 $1 \rightarrow 4$, 挖掘了 4 号宝藏。还开发了道路 $4 \rightarrow 3$, 挖掘了 3 号宝藏。工程总代价为: $1 \times 1 + 1 \times 1 + 1 \times 2 = 4$

【样例解释 2】



小明选定让赞助商打通了 1 号宝藏屋。小明开发了道路 $1 \rightarrow 2$, 挖掘了 2 号宝藏。开发了道路 $1 \rightarrow 3$, 挖掘了 3 号宝藏。还开发了道路 $1 \rightarrow 4$, 挖掘了 4 号宝藏。工程总代价为: $1 \times 1 + 3 \times 1 + 1 \times 1 = 5$

【数据规模与约定】

对于 20% 的数据: 保证输入是一棵树, $1 \leq n \leq 8$, $v \leq 5000$ 且所有的 v 都相等。

对于 40% 的数据: $1 \leq n \leq 8$, $0 \leq m \leq 1000$, $v \leq 5000$ 且所有的 v 都相等。

对于 70% 的数据: $1 \leq n \leq 8$, $0 \leq m \leq 1000$, $v \leq 5000$ 。

对于 100% 的数据: $1 \leq n \leq 12$, $0 \leq m \leq 1000$, $v \leq 500000$ 。

一句话题意: 给定一个有重边, 边有权值的无向图。从某一个点出发, 求到达所有的点需要的最少费用, 并且限制两点之间只有一条路径。

费用的计算公式为: 所有边的费用之和。而边 $(x \rightarrow y)$ 的费用就为: y 到初始点的距离 * 边权。

此题为 NOIp 2017 Day2 T2。是一道比较基础的状压 DP。有多种方法可以 AC 这道题目，正解有状压 DP，模拟退火，遗传算法等等。

会讲多种算法的。

6.2.4.1 随机化暴力

dalao 们应该都学习过搜索。搜索本质上就是个暴力。数据范围这么小，搜索是可以通过大部分数据的。是很划算的。

考虑随机一个点序列（在 DFS 时不按照输入顺序从 1 到 n 依次遍历整张图，容易被卡），然后指定一个定点为起点进行 DFS 计算出当前的花费（当然不一定要对整张图进行一次 DFS，稍微剪一下枝），符合题意时更新一下答案就行了。当然这种算法在 RP 不好的时候会输出最大值，但这就是小概率事件了，况且可以多次对这个图进行随机化 DFS。

考虑 DFS 的终止条件，由于我们希望图是一棵树（此时边数最小并且任意两点都可达），所以终止条件应该是找到了一棵树，即当前已经 DFS 过的点数等于 $n + 1$ （联系最小生成树理解一下？）。

代码：

```
#include<cctype>
#include<cstdio>
#include<algorithm>
using namespace std;
#define N 20
#define fr(_a,_b,_c) for(int _a=_b;_a<= _c;_a++)
inline int read()
{
    char ch=getchar();int ret=0,flag=1;
    while(!isdigit(ch)&&ch!='-')ch=getchar();
    if(ch=='-')flag=-1;
    while(isdigit(ch))ret=ret*10+ch-'0',ch=getchar();
    return flag*ret;
}
int n,m,d[N][N],ans,t[N],h[N];
void dfs(int x,int w)//x 表示当前 DFS 过的点的个数，w 表示当前状态下的花费。
{
    if(ans<=w)//剪枝：如果已经找到的答案比当前状态下的花费小，回溯。
        return;
    if(x==n+1)//更新解，如果 DFS 到了一棵树并且需要更新答案，更新答案。
    {
        ans=w;
        return;
    }
}
```

```

    }
    for(int i=1;i<=x-1;i++)
        if(d[t[x]][t[i]]+1)
    {
        h[t[x]]=h[t[i]]+1;
        dfs(x+1,w+d[t[x]][t[i]]*h[t[x]]);
    }
}
int main()
{
    n=read();
    m=read();
    fr(i,1,n)
        fr(j,1,n)
            d[i][j]=-1;
    fr(i,1,m)
    {
        int u=read(),v=read(),w=read();
        if(d[u][v]==-1)
            d[u][v]=d[v][u]=w;
        else
            d[u][v]=d[v][u]=min(d[u][v],w);
    }
    ans=(1<<20);
    fr(i,1,n)
        t[i]=i;
    srand((unsigned long long)new char);
    fr(i,1,5040)
    {
        fr(i,1,n*n)
            swap(t[rand()%n+1],t[rand()%n+1]);
        fr(i,1,n)
            h[i]=0;
        dfs(2,0);
    }
    printf("%d\n",ans);
    return 0;
}

```

时间复杂度 $O(5040 \times 3^n)$, 期望得分 0~100.

6.2.4.2 模拟退火

在介绍模拟退火算法之前先介绍一下 TSP 问题和 TSP 问题所属的 NP 问题和 NPC 问题。

定义 6.2.2 (NP 问题) 是指存在多项式算法能够解决的非决定性问题，而其中 NP 完全问题又是最有可能不是 P 问题的问题类型。所有的 NP 问题都可以用多项式时间划归到他们中的一个。所以显然 NP 完全的问题具有如下性质：它可以在多项式时间内求解，当且仅当所有的其他的 NP 完全问题也可以在多项式时间内求解。

以上是 NP 问题的定义，所谓多项式算法就是时间复杂度可以写作 $O(n^a)$ ， $a \in \mathbf{R}$ 的形式的算法。

定义 6.2.3 (NPC 问题) NP 中的某些问题的复杂性与整个类的复杂性相关联。这些问题中任何一个如果存在多项式时间的算法，那么所有 NP 问题都是多项式时间可解的。这些问题被称为 NP 完全问题 (NPC 问题)。

以上是对 NPC 问题的定义，换句话说就是不容易有有效算法的问题（需要搜索、枚举等等很暴力的算法来解决的问题）。

定义 6.2.4 (TSP 问题) 又译为旅行推销员问题、货郎担问题，是数学领域中著名问题之一。问题的一般形式类似于“假设有一个旅行商人要拜访 n 个城市，他必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。”

TSP 问题可以被证明是一个 NPC 问题。

以上是对 TSP 问题的定义，要注意到这是一个 NPC 问题。

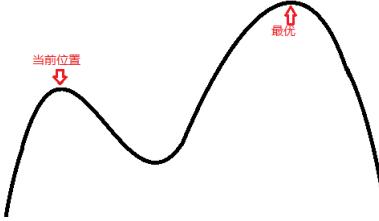
定义 6.2.5 (模拟退火算法) 是一种通用概率演算法，用来在一个大的搜寻空间内找寻命题的最优解。模拟退火算法是解决 TSP (旅行商问题) 的有效方法之一。

模拟退火的出发点是基于物理中固体物质的退火过程与一般组合优化问题之间的相似性。模拟退火算法是一种通用的优化算法，其物理退火过程由加温过程、等温过程、冷却过程这三部分组成。

可以看到，模拟退火是解决需要搜索的问题的一种有效算法（但是只针对于部分问题，不具有普适性）。

转回来看这个题，由于答案具有比较强的连续性（可能与随机数据等等有关），所以模拟退火算法适用于这道题目。

考虑下图所示的函数图像：



如果坚决不接受较差的解，那么我们就无法到达最优解所在的位置。这使我们必须接受存在较差的解（但不能全部接受，否则就与搜索一样了）。我们可以写一个接受函数来决定是否接受这个解。假设答案越小越优，我们可以得到以下接受函数：

```

bool accept(double delta, double temper)//delta表示函数的变化量, temper表示当前“温度”，
→ “温度”随时间推移缓慢减小，从而实现“退火”的过程。
{
    if(delta <= 0) return true;//如果找到了更优的解，就接受。
    return rand() <= exp((-delta) / temper) * RAND_MAX;//随机接受较差的解。
}

```

下面解释一下第二行接受较差的解的代码：

我们将 $\text{rand}() \leq \exp\left(\frac{-\Delta}{\text{temper}}\right) \times \text{RAND_MAX}$ 移项，可得： $\frac{\text{rand}()}{\text{RAND_MAX}} \leq \exp\left(\frac{-\Delta}{\text{temper}}\right)$ ，根据定义 temper 始终为正，而 Δ 为正数，所以 $\exp()$ 函数的自变量为负数。由指数函数的定义，返回值应该在区间 $(0,1)$ 之间，定为接受的概率。而左边随机生成了一个实数，如果比概率小就接受，否则就不接受。

简单来说，就是随机化接受较差的解，从而达到到达最优解的过程。由于“温度”随时间推移缓慢减小，所以跳往较差解的概率也下降。

模拟退火的原理是，通过赋予搜索过程一种时变且最终趋于零的概率突跳性，从而可有效避免陷入局部极小并最终趋于全局最优的串行结构的优化算法。由于这个原理的限制，模拟退火算法能找到的最优解可能不是最优解，但是与最优解已经非常相近了。这个特性比较适合做较高水平的比赛中的提交答案题，可能能~~拿~~掉标程，获得更高的分数。但是对于传统型题目，就很有可能 WA 掉。

结合这题的代码理解一下模拟退火的流程：

```

#include<bits/stdc++.h>
using namespace std;

```

```

const int maxn=110;
const int maxm=2010;
int mp[maxn][maxn],n,m,i,j,k,x,y,z,ans,s;
int a[maxn],b[maxn],c[maxn],ha[maxn],oldf,newf,vis[maxn];
double T;//模拟退火中的温度
int read()
{
    int tot=0,fh=1;
    char c=getchar();
    while((c-'0'<0)|| (c-'0'>9)){if(c=='-')fh=-1;c=getchar();}
    while((c-'0'>=0)&&(c-'0'<=9)){tot=tot*10+c-'0';c=getchar();}
    return tot*fh;
}
int dfs(int x,int dep)//遍历这张图, x表示当前点的编号, dep表示当前遍历过的点的个数
{
    if (dep>13) return 0;//如果已经不是一棵树了, 不符合题中“小明不想开发无用道路, 即两个
    ↪ 已经被挖掘过的宝藏屋之间的道路无需再开发”这一条件, 进行剪枝。
    if (x!=s){vis[x]=dfs(c[x],dep+1);return vis[x];}//若还未到达终点就继续进行 DFS.
    else return 1;//若到达终点表示这是一组可行解。
}
int check()
{
    int i;
    memset(vis,0,sizeof(vis));
    for (i=1;i<=n;i++)//以每一个点为起点进行 DFS
    {
        if (vis[i]==0) vis[i]=dfs(i,1);//vis[i] 表示以 i 号点为起点是否存在可行解。
        if (vis[i]==0) return 0;//不存在可行解
    }
    return 1;
}
int dfs2(int x)//用于计算  $\sum L_i \times K_i$  新值的搜索, x表示当前点的编号。
{
    if ((s==x)|| (vis[x]!=0)) return vis[x];//如果是起点或者已经被访问过了就返回 x 点的
    ↪ 深度。
    int t1=dfs2(c[x])+1;//记录深度, 即题目中的 K
    vis[x]=t1;//记录当前点的深度
    newf=newf+t1*mp[c[x]][x];//更新  $\sum L_i \times K_i$  的值
    return t1;
}
void getnewf()//对 dfs2() 的封装, 用于计算  $\sum L_i \times K_i$  的值
{
    int i;
    newf=0;
    memset(vis,0,sizeof(vis));
    for (i=1;i<=n;i++)//以每个点为起点进行 DFS

```

```

{
    if (vis[i]==0) dfs2(i); //如果没被访问过，即当前点对答案的贡献还没有计算进去，就计算一下从当前点可以增广到的点的距离。
}
}

void bfs(int x)//寻找近似解
{
    int l=1,r=1,i; a[l]=x; oldf=0; b[x]=0;
    memset(ha,-1,sizeof(ha)); ha[x]=0;
    while (l<=r)
    {
        for (i=1;i<=n;i++)
        {
            if ((ha[i]==-1)&&(mp[a[l]][i]!=1e9))//如果没有被增广过并且 l 与 i 之间有边
            {
                ha[i]=ha[a[l]]+1;//记录一下深度
                r++; a[r]=i;//入队
                b[i]=a[l];//记录一下父亲节点
                oldf=oldf+mp[a[l]][i]*ha[i];//计算原有的 f 函数
            }
        }
        l++;//旧节点出队
    }
    ans=min(ans,oldf);//两者取一个较小值。
}

int main()
{
    n=read(); m=read();
    for (i=1;i<=n;i++)
    {
        for (j=1;j<=n;j++)
        {
            if (i==j) mp[i][j]=0;
            else mp[i][j]=1e9;
        }
    }
    for (i=1;i<=m;i++)
    {
        x=read(); y=read(); z=read();
        mp[x][y]=min(mp[x][y],z);
        mp[y][x]=min(mp[y][x],z);
    }
    ans=1e9;
    srand(19260817);
    for (i=1;i<=n;i++)
    {

```

```

s=i; bfs(i); //以每个点为起点进行一次 BFS 计算深度。
if (n<=2) continue; //如果只有两个点，就不需要退火。
for (T=10000;T>=0.00001;T=T*0.9999)//模拟退火过程，T 表示温度。
{
    x=(rand()%n)+1; if (x>=i) x++; //随机指定起点和终点。
    y=(rand()%n)+1;
    if (y>x) y++;
    if (mp[x][y]==1e9) continue; //不连通
    for (j=1;j<=n;j++) c[j]=b[j]; //复制一下 b 数组
    c[x]=y; //起点的父亲节点是终点。
    if (check()==0) continue; //如果不存在可行解跳出循环
    getnewf(); //计算新的代价
    if ((newf<=oldf)||((exp((oldf-newf)/T))>=((rand()%1000000)/1000000.0))) //接受
    ↳ 受函数，解释见上。
    {
        ans=min(ans,newf); //更新一下答案
        for (j=1;j<=n;j++) b[j]=c[j]; //复制一下 c 数组
        oldf=newf; //刚刚计算出的函数值变成旧的函数值
    }
}
printf("%d\n",ans);
return 0;
}

```

时间复杂度 $O(207223n^3)$.

6.2.4.3 最小生成树

这是考场上很多人想到的非完美解法，当时在考场上好像是可以过 1 号样例，但是 2 号样例过不了。我在考场上也想到了这种方法，然而并没有写完，所以 gg。

考虑一下最小生成树的问题所在。Kruskal 算法需要贪心地选择边权较小的边，所以如果枚举的点不在最小生成树上，贪心的结果就是错误的。

就是说，如果到某一个点的最优路径不在最小生成树上（因为不仅仅是最小生成树上的点，如果一条边在最小生成树上，但是他的深度非常深，那么它对答案的贡献，即 $L \times K$ 就会非常大），答案就不对了。有一组数据可以卡掉 Prim 算法（当然据说 Prim 可以过大数据，yyf dalao 写过 Kruskal 好像是没有分）。

至于卡掉最小生成树的数据的话，rqy dalao 出了一组：

7	7
---	---

```

1 2 5000
2 3 1
3 4 1
4 5 1
5 6 2
2 7 9
6 7 3

```

6.2.4.4 状压 DP

考虑一个事实，给定一个加点序列，上一节提出的贪心的策略就是正确的了（计算答案的过程转化成类似最短路的东西）。

所以枚举一下 1~12 的全排列，get 到加点的序列，然后贪心就行了。

但是这样时间复杂度就是 $O(n!2^n)$ ，只能过一部分数据（可以拿到 80 分）。

考虑题目要求构造一棵生成树，使得所有边的费用之和最小。问题转换为树上问题。

设所有的点构成全集 U ，定义状态 $f_{i,j}$ 表示考虑到树的第 i 层，前 i 层已选的点的集合为 S （使用二进制压缩状态）的最小代价。

状态的转移可以枚举所有不在 S 中的集合（令集合 $S' \subseteq (\complement_U S)$ ，枚举集合 S' ）作为树的第 $i + 1$ 层。

状态转移：

$$dp[i][S] \rightarrow dp[i+1][S|S'] + (i+1) \times shortestPath[S'][S]$$

其中 $shortestPath[S'][S]$ 表示从集合 S' 到集合 S 的最短距离（需要提前用 Floyd 算法预处理两点间的最短路，然后处理出来），如果理解不了就看一下上面对集合间最短距离的定义，就是枚举集合 S 与集合 S' 所有点对 (a,b) 所在集合 $P = \{p(a,b) \mid a \in S, b \in S', S \cap S' = \emptyset\}$ ，点对 $p(a,b)$ 的最短路长度之和，程序中使用 $pval$ 数组保存了每个点对之间的最短路，然后再转化成集合之间的距离，存储到 $sval$ 数组里面。

DP 的顺序：由 DP 方程，由大到小枚举层数，然后枚举集合就行了。

边界条件：枚举根节点，则 $dp[0][1 \ll (root - 1)] = 0$ 。

代码：

```

#include<cstdio>
#include<algorithm>

```

```

using namespace std;
typedef long long LL;
const int INF = 1e7;
const int N = 13;
int n, m, g[N][N], u, v, p, U;
LL dp[N][1<<N], ans = 1e14, sval[1<<N][1<<N], pval[N][1<<N];
void init(int root)//初始化 DP 数组, 建树, 将边界条件设置好
{
    for(int i = 0; i <= n; i++)
        for(int j = 0; j <= U; j++)
            dp[i][j] = INF;
    dp[0][1<<(root-1)] = 0;
}
int main()
{
    scanf("%d%d", &n, &m);
    U = (1 << n) - 1;
    for(int i = 1; i <= n; i++)//初始化图、DP 数组、最短路数组
        for(int j = 1; j <= n; j++)
            if(i ^ j)
                g[i][j] = INF;
    for(int i = 1; i <= n; i++)
        for(int j = 0; j <= U; j++)
            pval[i][j] = INF;
    for(int i = 0; i <= U; i++)
        for(int j = 0; j <= U; j++)
            sval[i][j] = INF;
    while(m--)//邻接矩阵加边, 注意有重边的情况取较小值。
    {
        scanf("%d%d%d", &u, &v, &p);
        g[u][v] = min(g[u][v], p);
        g[v][u] = min(g[v][u], p);
    }
    for(int i = 1; i <= n; i++)//预处理所有满足条件的点对 (a, b) 之间的最短路, 使用 Floyd
    ↳ 算法
        for(int j = 0; j <= U; j++)
            for(int k = 1; k <= n; k++)
                if(j & (1 << (k - 1)))
                    pval[i][j] = min(pval[i][j], 1ll*g[i][k]);
    for(int i = 0; i <= U; i++)//预处理集合之间的距离
    {
        int C = i ^ U;//求出 C ∪ S
        for(int s = C; s; s = (s - 1) & C)
        {
            LL temp = 0;
            for(int j = 1; j <= n; j++)

```

```

        if(s & (1 << (j - 1)))
            temp += pval[j][i];
        sval[s][i] = temp >= INF ? INF : temp;
    }
}

for(int root = 1; root <= n; root++)//枚举每一个根
{
    init(root);
    for(int i = 0; i < n; i++)//枚举每一个点
        for(int S = 0; S <= U; S++)//枚举每一个集合
            if(dp[i][S] != INF)
            {
                int C = S ^ U;//求出 C_U S
                for(int s = C; s; s = (s - 1) & C)
                    dp[i+1][S|s] = min(dp[i+1][S|s], dp[i][S] + (i + 1) *
                        sval[s][S]);//这里的'/'运算是对集合做的运算，相当于求集合
                        的并。
            }
    for(int i = 0; i < n; i++) ans = min(ans, dp[i][U]);
}
printf("%lld", ans);
return 0;
}

```

时间复杂度 $O(n^2 \times 2^n) = 84934656 < 10^8$, 可以通过所有数据。

这样的话，状压 DP 就讲完了。还是有些难理解的，我讲的可能不是很清楚，课下好好理解一下吧。

(其实状压 DP 一般可以用启发式搜索或者是记忆化搜索水过去，就像前面讲的随机化暴力和模拟退火算法一样)

课堂笔记

Note on the text

6.3 数位 DP

6.3.1 基础知识

定义 6.3.1 (数位 DP) 数位 dp 是一种计数用的 dp, 一般就是要统计一个区间 $[l_e, r_i]$ 内满足一些条件数的个数。

以上是对于数位 DP 的定义。

做关于数位的题目首先考虑一下枚举方式:

```
1. for(int i=le;i<=ri;i++)
    if(right(i)) ans++;
```

无法记忆化搜索。gg。

2. 控制上界, 按位从高到低枚举。注意枚举时从高位开始, 前导零也要枚举。需要注意的是枚举是从“00001”这样开始枚举, 一直枚举到小于右边界。

基本的动态模板:

```
typedef long long ll;
int a[20];
ll dp[20][state];//不同题目状态不同
ll dfs(int pos,/*state 变量*/,bool lead/*前导零*/,bool limit/*数位上界变量*/)//不是每个
→ 题都要判断前导零
{
    //递归边界, 既然是按位枚举, 最低位是 0, 那么 pos== -1 说明这个数我枚举完了
    if(pos== -1) return 1; /*这里一般返回 1, 表示你枚举的这个数是合法的, 那么这里就需要你在
    → 枚举时必须每一位都要满足题目条件, 也就是说当前枚举到 pos 位, 一定要保证前面已经枚举
    → 的数位是合法的。不过具体题目不同或者写法不同的话不一定返回 1 */
    //第二个就是记忆化(在此前可能不同题目还能有一些剪枝)
    if(!limit && !lead && dp[pos][state] != -1) return dp[pos][state];
    /*常规写法都是在没有限制的条件记忆化, 这里与下面记录状态是对应, 具体为什么是有条件的记
    → 忆化后面会讲*/
    int up=limit?a[pos]:9;//根据 limit 判断枚举的上界 up; 这个的例子前面用 213 讲过了
    ll ans=0;
    //开始计数
    for(int i=0;i<=up;i++)//枚举, 然后把不同情况的个数加到 ans 就可以了
    {
        if(...)
        else if(...)
        ans+=dfs(pos-1,/*状态转移*/,lead && i==0,limit && i==a[pos]) //最后两个变量传参
        → 都是这样写的
    }
```

```

/*这里还算比较灵活，不过做几个题就觉得这里也是套路了
大概就是说，我当前数位枚举的数是 i，然后根据题目的约束条件分类讨论
去计算不同情况下的个数，还有要根据 state 变量来保证 i 的合法性，比如题目
要求数位上不能有 62 连续出现，那么就是 state 就是要保存前一位 pre，然后分类，
前一位如果是 6 那么这意味就不能是 2，这里一定要保存枚举的这个数是合法*/
}

//计算完，记录状态
if(!limit && !lead) dp[pos][state]=ans;
/*这里对应上面的记忆化，在一定条件下时记录，保证一致性，当然如果约束条件不需要考虑 lead,
→ 这里就是 lead 就完全不用考虑了*/
return ans;
}

ll solve(ll x)
{
    int pos=0;
    while(x)//把数位都分解出来
    {
        a[pos++]=x%10;//个人老是喜欢编号为 [0, pos)，看不惯的就按自己习惯来，反正注意数位边界就行
        x/=10;
    }
    return dfs(pos-1/*从最高位开始枚举*/,/*一系列状态 */ ,true,true); //刚开始最高位都是有限制并且有前导零的，显然比最高位还要高的一位视为 0 嘛
}
int main()
{
    ll le,ri;
    while(~scanf("%lld%lld",&le,&ri))
    {
        //初始化 dp 数组为 -1，这里还有更加优美的优化，后面讲
        printf("%lld\n",solve(ri)-solve(le-1));
    }
}

```

为了做到无遗漏的枚举，代码中的 *limit* 变量就是用于判断枚举位置。最重要的问题在于“枚举上界”。如果不能理解的话，举个例子：枚举上界为 219。假设已经枚举最高位为 1，因为 $0 \leq 2$ ，所以十位上怎么枚举也不会超过上界，*limit* 应该为 false，表示对这一位的枚举没有上界限制；假设枚举最高位为 2，此时对十位的枚举就应该有限制，枚举上界为 1；（对高位的枚举对低位有影响。）当然对位数小于 *ri* 的数来说，怎么枚举也没有问题，需要枚举上。想想判 *limit* 与不判 *limit* 记忆化搜索的不同。假设没判 *limit*，那么假设我们第一次枚举了百位是 0，显然后面的枚举 *limit*=false，也就是数位上 0 到 9 的枚举，然后当我十位枚举了 1，此时考虑 *dp*[0][1]，就

是枚举到个位，前一位是 1 的个数，显然 $dp[0][1]=9$; (个位只有是 1 的时候是不满足的)，这个状态记录下来，继续 dfs，一直到百位枚举了 2，十位枚举了 1，显然此时递归到了 $pos=0, pre=1$ 的层，而 $dp[0][1]$ 的状态已经有了即 $dp[pos][pre]!=-1$; 此时程序直接 return $dp[0][1]$ 了，然而显然是错的，因为此时是有 limit 的个位只能枚举 0，根本没有 9 个数，这就是状态冲突了。有 lead 的时候可能出现冲突，这只是两个最基本的不同的题目可能还要加限制，反正宗旨都是让 dp 状态唯一。有两点需要说明：

1. limit 为 true 的情况并不会非常多，逐个进行判断也不会特别浪费时间，所以需要记录一下，以解决子问题重叠的问题。
2. DP 状态可以改为 $dp[pos][status][limit]$ ，给 limit 也记忆化，记录不同的 limit 的个数，这种方法一般是对的。有些题目需要这样优化。

还涉及另一个问题。因为仅仅限制了枚举的上界，求出来的值不一定是原来题目要求的区间内的答案（实际上是区间 $[0, ri]$ 的答案），所以要对答案进行一些处理。

```
int main()
{
    long long le,ri;
    while(~scanf("%lld%lld",&le,&ri))
        printf("%lld\n",solve(ri)-solve(le-1));
}
```

上面的代码事实上就是对区间 $[0, ri]$ 与 $[li, ri]$ 取一个交集的过程，由高中数学的集合运算法则 $A \cap B = A(A \subseteq B)$ 而来。

DP 的话，同记忆化搜索。状态一般是与前面已经枚举的数位有关，并且往往是对连续 k 位有要求，就保存前面 $k - 1$ 位的状态，然后当前枚举第 k 位，和前 $k - 1$ 位恰好拼凑出完整的状态。当然如果状态是数位的和，就直接保存数位的前缀和。

6.3.2 入门例题—不要 62

我们来看一道入门难度的例题。

例题 6.3.1 HDU2089 不要 62

题目描述

杭州人称那些傻乎乎粘嗒嗒的人为 62 (音: *laoer*)。

杭州交通管理局经常会扩充一些的士车牌照，新近出来一个好消息，以后上牌照，不再含有不吉利的数字了，这样一来，就可以消除个别的士司机和乘客的心理障碍，更安全地服务大众。

不吉利的数字为所有含有 4 或 62 的号码。例如: 62315 73418 88914 都属于不吉利号码。但是, 61152 虽然含有 6 和 2, 但不是 62 连号, 所以不属于不吉利数字之列。

你的任务是, 对于每次给出的一个牌照区间号, 推断出交管局今次又要实际上给多少辆新的士车上牌照了。

输入格式

输入的都是整数对 n, m ($0 \leq n \leq m \leq 1000000$), 如果遇到都是 0 的整数对, 则输入结束。

输出格式

对于每个整数对, 输出一个不含有不吉利数字的统计个数, 该数值占一行位置。

样例输入

```
1 100
0 0
```

样例输出

```
80
```

就是数位上不能有 4 也不能有连续的 62, 没有 4 的话在枚举的时候判断一下, 不枚举 4 就可以保证状态合法了, 所以这个约束没有记忆化的必要, 而对于 62 的话, 涉及到两位, 当前一位是 6 或者不是 6 这两种不同情况我计数是不相同的, 所以要用状态来记录不同的方案数。

$dp[pos][sta]$ 表示当前第 pos 位, 前一位是否是 6 的状态, 这里 sta 只需要去 0 和 1 两种状态就可以了, 不是 6 的情况可视为同种, 不会影响计数。

代码:

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

```

typedef long long ll;
int a[20];
int dp[20][2];
int dfs(int pos,int pre,int sta,bool limit)
{
    if(pos== -1) return 1;
    if(!limit && dp[pos][sta] != -1) return dp[pos][sta];
    int up=limit ? a[pos] : 9;
    int tmp=0;
    for(int i=0;i<=up;i++)
    {
        if(pre==6 && i==2) continue;
        if(i==4) continue;//都是保证枚举合法性
        tmp+=dfs(pos-1,i,i==6,limit && i==a[pos]);
    }
    if(!limit) dp[pos][sta]=tmp;
    return tmp;
}
int solve(int x)
{
    int pos=0;
    while(x)
    {
        a[pos++]=x%10;
        x/=10;
    }
    return dfs(pos-1,-1,0,true);
}
int main()
{
    int le,ri;
    //memset(dp,-1,sizeof dp); 可优化
    while(~scanf("%d%d",&le,&ri) && le+ri)
    {
        memset(dp,-1,sizeof dp);
        printf("%d\n",solve(ri)-solve(le-1));
    }
    return 0;
}

```

其实还是记忆化搜索。

6.3.3 常数优化

讲一个常数优化，放在 ACM 赛制中比较好用。

把 memset 放在多组数据以外。由于数位 DP 的特点，DP 状态与数本身有关，而不是与数据的组数有关，所以之前的状态还可以继续使用。

求数位和是 10 的倍数的个数，这里简化为数位 sum%10 这个状态，即 dp[pos][sum] 这里 10 是与多组无关的，所以可以 memset 优化，不过注意如果题目的模数是输入的话那就不能这样了。

6.3.4 减法的艺术—F(x)

先来看一道例题。

例题 6.3.2 HDU4734 F(x)

题目描述

对于一个有 n 位的十进制数 $x(A_nA_{n-1}A_{n-2}\cdots A_2A_1)$ ，我们定义它的权值为：

$$F(x) = A_n \times 2^{n-1} + A_{n-1} \times 2^{n-2} + A_{n-2} \times 2^{n-3} + \cdots + A_2 \times 2 + A_1 \times 1.$$

给定两个数 A 和 B ，请计算在区间 $[0, B]$ 内有多少个数权值不超过 $F(A)$ 。

输入格式

第一行包含一个数 $T(T \leq 10000)$ ，表示测试数据组数；

对于每一组测试数据，都有两个数 A 和 $B(0 \leq A, B \leq 10^9)$ 。

输出格式

对于每一组数据，你应该先输出“Case #t: ”（不包含引号）， t 是测试数据编号，从 1 开始。然后输出答案。

样例输入

```
3
0 100
1 10
5 100
```

样例输出

```
Case #1: 1
Case #2: 2
Case #3: 13
```

常规想：这个 $f(x)$ 计算就和数位计算是一样的，就是加了权值，所以 $dp[pos][sum]$ ，这状态是基本的。 a 是题目给定的， $f(a)$ 是变化的不过 $f(a)$ 最大好像是 4600 的样子。如果要 memset 优化就要加一维存 $f(a)$ 的不同取值，那就是 $dp[10][4600][4600]$ ，这显然不合法。

这个时候就要用减法了， $dp[pos][sum]$ ， sum 不是存当前枚举的数的前缀和（加权的），而是枚举到当前 pos 位，后面还需要凑 sum 的权值和的个数，也就是说初始的是时候 sum 是 $f(a)$ ，枚举一位就减去这一位在计算 $f(i)$ 的权值，那么最后枚举完所有位 $sum \leq 0$ 时就是满足的，后面的位数凑足 sum 位就可以了。

仔细想想这个状态是与 $f(a)$ 无关的，一个状态只有在 $sum \leq 0$ 时才满足，如果我们按常规的思想求 $f(i)$ 的话，那么最后 $sum \leq f(a)$ 才是满足的条件。

这道题目展示了前缀和在数位 DP 中的重要作用。希望大家理解一下。
以下是代码：

```
#include<cstdio>
#include<cstring>
#include<iostream>
#include<string>

using namespace std;
const int N=1e4+5;
int dp[12][N];
int f(int x)
{
    if(x==0) return 0;
    int ans=f(x/10);
    return ans*2+(x%10);
}
int all;
int a[12];
int dfs(int pos,int sum,bool limit)
{
    if(pos==-1) {return sum<=all;}
    if(sum>all) return 0;
    if(!limit && dp[pos][all-sum]!=-1) return dp[pos][all-sum];
    int up=limit ? a[pos] : 9;
    int ans=0;
    for(int i=0;i<=up;i++)
    {
        ans+=dfs(pos-1,sum+i*(1<<pos),limit && i==a[pos]);
    }
    if(!limit) dp[pos][all-sum]=ans;
}
```

```
        return ans;
    }
    int solve(int x)
    {
        int pos=0;
        while(x)
        {
            a[pos++]=x%10;
            x/=10;
        }
        return dfs(pos-1,0,true);
    }
    int main()
    {
        int a,ri;
        int T;
        int kase=1;
        scanf("%d",&T);
        memset(dp,-1,sizeof dp);
        while(T--)
        {
            scanf("%d%d",&a,&ri);
            all=f(a);
            printf("Case # %d: %d\n",kase++,solve(ri));
        }
        return 0;
    }

```

课堂笔记

Note on the text

7 典型题目与难题选讲

7.1 典型题目

这里给出几道基础的 DP 题目，大家尽情 AK 吧！

- [USACO1.5] 数字三角形 Number Triangles—例题
- Likecloud-吃、吃、吃—数字三角形
- 金明的预算方案—例题
- 过河卒—模拟式 DP
- 最大子段和—DP 或者贪心
- 合唱队形—最长上升子序列+最长下降子序列
- 友好城市—最长上升子序列
- 导弹拦截—最长不下降子序列
- 低价购买—最长下降子序列+统计
- [HNOI2004] 打鼹鼠—最长上升子序列
- [USACO07DEC] 手链 Charm Bracelet—01 背包模板题
- 采药—01 背包模板题
- 开心的金明—01 背包
- [USACO09OCT]Bessie 的体重问题 Bessie's We...—01 背包
- 小书童——刷题大军—01 背包
- 装箱问题—01 背包
- 疯狂的采药—完全背包
- 通天之分组背包—分组背包
- NASA 的食物计划—二维背包
- 小 A 点菜—背包问题的变体

- 垃圾陷阱—类背包 DP
- 尼克的任务—考虑枚举方向
- 书本整理—转化
- 琪露诺—DP 的优化
- 组合数问题—杨辉三角+前缀和

题目有些多，但是还是最好一一刷完。最后 4 道题以及每个类型的最后一道题稍微有些难度。

想要提升能力的话，可以尝试一下下面这些题目：

- 方格取数—双线程 DP
- 传纸条—双线程 DP
- 乌龟棋—较高维度的 DP
- 小 a 和 uim 之大逃离—较复杂的 DP

以上差不多有 30 道题的样子了。希望大家能抽时间做一下。



7.2 难题选讲

7.2.1 组合数问题—数论/数学, DP, 前缀和优化

例题 7.2.1 组合数问题

【问题描述】

组合数 C_n^m 表示的是从 n 个物品中选出 m 个物品的方案数。举个例子，从 $(1, 2, 3)$ 三个物品中选择两个物品可以有 $(1, 2), (1, 3), (2, 3)$ 这三种选择方法。根据组合数的定义，我们可以给出计算组合数的一般公式：

$$C_n^m = \frac{n!}{m!(n-m)!}$$

其中 $n! = 1 \times 2 \times \dots \times n$ 。

小葱想知道如果给定 n, m 和 k ，对于所有的 $0 \leq i \leq n, 0 \leq j \leq \min(i, m)$ 有多少对 (i, j) 满足 C_i^j 是 k 的倍数。

【输入输出格式】

【输入格式】

从文件 *problem.in* 中读入数据。

第一行有两个整数 t, k ，其中 t 代表该测试点总共有多少组测试数据， k 的意义见 【问题描述】。

接下来 t 行每行两个整数 n, m ，其中 n, m 的意义见 【问题描述】。

【输出格式】

输出到文件 *problem.out* 中。

t 行，每行一个整数代表答案。

【输入输出样例】

【输入样例 1】

```
1 2
3 3
```

【输出样例 1】

```
1
```

【输入样例 2】

```
2 5
4 5
6 7
```

【输出样例 2】

```
0  
7
```

【说明】

【样例 1 说明】

在所有可能的情况下，只有 $C_2^1 = 2$ 是 2 的倍数。

【子任务】

子任务会给出部分测试数据的特点。如果你在解决题目中遇到了困难，可以尝试只解决一部分测试数据。

每个测试点的数据规模及特点如下表：

测试点	n	m	k	t
1	≤ 3	≤ 3	$= 2$	$= 1$
2			$= 3$	$\leq 10^4$
3	≤ 7	≤ 7	$= 4$	$= 1$
4			$= 5$	$\leq 10^4$
5	≤ 10	≤ 10	$= 6$	$= 1$
6			$= 7$	$\leq 10^4$
7	≤ 20	≤ 100	$= 8$	$= 1$
8			$= 9$	$\leq 10^4$
9	≤ 25	≤ 2000	$= 10$	$= 1$
10			$= 11$	$\leq 10^4$
11	≤ 60	≤ 20	$= 12$	$= 1$
12			$= 13$	$\leq 10^4$
13	≤ 100	≤ 25	$= 14$	$= 1$
14			$= 15$	$\leq 10^4$
15		≤ 60	$= 16$	$= 1$
16			$= 17$	$\leq 10^4$
17	≤ 2000	≤ 100	$= 18$	$= 1$
18			$= 19$	$\leq 10^4$
19		≤ 2000	$= 20$	$= 1$
20			$= 21$	$\leq 10^4$

这是一道数论题，暴力分也很多（50 分）。

我们可以先打个表（毕竟数论上来先打表），看看前若干项的值。

手推一下样例二：

$$\begin{array}{ccccccc} C_0^0 & & & & & & \\ C_1^0 & C_1^1 & & & & & \\ C_2^0 & C_2^1 & C_2^2 & & & & \\ C_3^0 & C_3^1 & C_3^2 & C_3^3 & & & \\ C_4^0 & C_4^1 & C_4^2 & C_4^3 & C_4^4 & & \\ C_5^0 & C_5^1 & C_5^2 & C_5^3 & C_5^4 & C_5^5 \end{array}$$

套套公式，看看在数组中的存储（不符合组合数的定义处设为 -1）：

$$\begin{array}{ccccccc} -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & 1 & 0 & 0 & 0 & 0 \\ -1 & 3 & 3 & 1 & 0 & 0 & 0 \\ -1 & 4 & 6 & 4 & 1 & 0 & 0 \\ -1 & 5 & 10 & 10 & 5 & 1 & 0 \\ -1 & 6 & 15 & 20 & 15 & 6 & 1 \end{array}$$

容易看出这是一个杨辉三角。

建立状态转移方程。 $O(n^2)$ 求出杨辉三角。然后数组存储杨辉三角的对应项对 k 取模后的值即可。

再开个数组，统计一下数组中有多少个 0 就行了。

然后写一个前缀和优化一下，问题解决。

时间复杂度 $O(n^2)$ 。

课堂笔记

Note on the text

7.2.2 换教室—期望 DP

期望 DP 是最近几年新加入 NOIp 考纲的内容，于 NOIp 2016 试水，以后几年可能会比较常考。在这里只介绍最基础的一道题，来源是 NOIp 2016 Day1 T3.

例题 7.2.2 换教室

题目描述

对于刚上大学的牛牛来说，他面临的第一个问题是根据实际情况申请合适的课程。

在可以选择的课程中，有 $2n$ 节课程安排在 n 个时间段上。在第 i ($1 \leq i \leq n$) 个时间段上，两节内容相同的课程同时在不同的地点进行，其中，牛牛预先被安排在教室 c_i 上课，而另一节课在教室 d_i 进行。

在不提交任何申请的情况下，学生们需要按时间段的顺序依次完成所有的 n 节安排好的课程。如果学生想更换第 i 节课程的教室，则需要提出申请。若申请通过，学生就可以在第 i 个时间段去教室 d_i 上课，否则仍然在教室 c_i 上课。

由于更换教室的需求太多，申请不一定能获得通过。通过计算，牛牛发现申请更换第 i 节课程的教室时，申请被通过的概率是一个已知的实数 k_i ，并且对于不同课程的申请，被通过的概率是互相独立的。

学校规定，所有的申请只能在学期开始前一次性提交，并且每个人只能选择至多 m 节课程进行申请。这意味着牛牛必须一次性决定是否申请更换每节课的教室，而不能根据某些课程的申请结果来决定其他课程是否申请；牛牛可以申请自己最希望更换教室的 m 门课程，也可以不用完这 m 个申请的机会，甚至可以一门课程都不申请。

因为不同的课程可能会被安排在不同的教室进行，所以牛牛需要利用课间时间从一间教室赶到另一间教室。

牛牛所在的大学有 v 个教室，有 e 条道路。每条道路连接两间教室，并且是可以双向通行的。由于道路的长度和拥堵程度不同，通过不同的道路耗费的体力可能会有所不同。当第 i ($1 \leq i \leq n-1$) 节课结束后，牛牛就会从这节课的教室出发，选择一条耗费体力最少的路径前往下一节课的教室。

现在牛牛想知道，申请哪几门课程可以使他在教室间移动耗费的体力值的总和的期望值最小，请你帮他求出这个最小值。

输入输出格式

输入格式

第一行四个整数 n, m, v, e 。 n 表示这个学期内的时间段的数量； m 表示牛牛最多可以申请更换多少节课程的教室； v 表示牛牛学校里教室的数量； e 表示牛牛的学校里道路的数量。

第二行 n 个正整数，第 i ($1 \leq i \leq n$) 个正整数表示 c_i ，即第 i 个时间段牛牛被安排上课的教室；保证 $1 \leq c_i \leq v$ 。

第三行 n 个正整数，第 i ($1 \leq i \leq n$) 个正整数表示 d_i ，即第 i 个时间段另一间上同样课程的教室；保证 $1 \leq d_i \leq v$ 。

第四行 n 个实数，第 i ($1 \leq i \leq n$) 个实数表示 k_i ，即牛牛申请在第 i 个时间段更换教室获得通过的概率。保证 $0 \leq k_i \leq 1$ 。

接下来 e 行，每行三个正整数 a_j, b_j, w_j ，表示有一条双向道路连接教室 a_j, b_j ，通过这条道路需要耗费的体力值是 w_j ；保证 $1 \leq a_j, b_j \leq v$ ， $1 \leq w_j \leq 100$ 。

保证 $1 \leq n \leq 2000$, $0 \leq m \leq 2000$, $1 \leq v \leq 300$, $0 \leq e \leq 90000$ 。

保证通过学校里的道路，从任何一间教室出发，都能到达其他所有的教室。

保证输入的实数最多包含 3 位小数。

输出格式

输出一行，包含一个实数，四舍五入精确到小数点后恰好 2 位，表示答案。你的输出必须和标准输出完全一样才算正确。

测试数据保证四舍五入后的答案和准确答案的差的绝对值不大于 4×10^{-3} 。（如果你不知道什么是浮点误差，这段话可以理解为：对于大多数的算法，你可以正常地使用浮点数类型而不用对它进行特殊的处理）

输入输出样例

输入样例

```
3 2 3 3
2 1 2
1 2 1
0.8 0.2 0.5
1 2 5
1 3 3
2 3 1
```

输出样例

```
2.80
```

说明

【样例说明】

所有可行的申请方案和期望收益如下表:

申请通过的时间段	出现的概率	耗费的体力值	耗费的体力值的期望
无	1.0	8	8.0
1	0.8	4	4.8
无	0.2	8	
2	0.2	0	6.4
无	0.8	8	
3	0.5	4	6.0
无	0.5	8	
1、2	0.16	4	4.48
1	0.64	4	
2	0.04	0	
无	0.16	8	
1、3	0.4	0	2.8
1	0.4	4	
3	0.1	4	
无	0.1	8	
2、3	0.1	4	5.2
2	0.1	0	
3	0.4	4	
无	0.4	8	

【提示】

1. 道路中可能会有多条双向道路连接相同的两间教室。 也有可能有道路两端连接的是同一间教室。

2. 请注意区分 n, m, v, e 的意义, n 不是教室的数量, m 不是道路的数量。

特殊性质 1:图上任意两点 $a_i, b_i, a_i \neq b_i$ 间, 存存在一条耗费体力最少的路径只包含一条道路。

特殊性质 2:对于所有的 $1 \leq i \leq n, k_i = 1$ 。

【子任务】

测试点	n	m	v	特殊性质 1	特殊性质 2
1	≤ 1	≤ 1	≤ 300		
2	≤ 2	≤ 0	≤ 20	×	×
3		≤ 1	≤ 100		
4		≤ 2	≤ 300		
5	≤ 3	≤ 0	≤ 20	√	√
6		≤ 1	≤ 100		×
7		≤ 2	≤ 300		
8	≤ 10	≤ 0	≤ 20	√	√
9		≤ 1	≤ 20		×
10		≤ 2	≤ 100	×	×
11		≤ 10	≤ 300		√
12		≤ 0	≤ 20	√	
13	≤ 20	≤ 1	≤ 100	×	×
14		≤ 2	≤ 300	√	
15		≤ 20			√
16		≤ 0	≤ 20	×	×
17		≤ 1	≤ 100		
18	≤ 300	≤ 2	≤ 300	√	√
19		≤ 300			
20		≤ 0	≤ 20	×	
21		≤ 1			
22		≤ 2	≤ 100		
23		≤ 2000	≤ 100		
24			≤ 300		
25					

题意：给出一幅 v 个点的无向图，表示教室及其连边。有 n 个时刻，每个时刻正常要到教室 $c[i]$ 上课，如果该时刻有申请更换，则到教室 $d[i]$ 上课。你只能在一切开始之前提交申请，且最多申请换 m 个时刻。第 i 个时刻申请成功的概率为 $k[i]$ 。求移动路程的期望最小值。 $n, m \leq 2000, v \leq 300$ 。

首先您得知道期望是什么，否则这个题期望得分就只有特殊性质 2 的部分分 35 分了。

定义 7.2.1 (期望) 离散型随机变量的一切可能的取值 x_i 与对应的概率 $P_i (=x_i)$ 之积的和称为该离散型随机变量的数学期望 (设级数绝对收敛), 记为 $E(x)$.

所以可以得到期望的计算公式:

$$E\xi = \sum_{i=1}^n x_i p_i$$

那么该怎么做呢? 思路显而易见, 就是先求出点与点之间的最短路, 然后套一下公式求出期望, 至于方案什么的就是很基础的 DP 了, 直接写状态转移方程就行了。第一眼先看看数据范围, $v \leq 300$? Floyd 多源最短路? 然后这张图就废了。就不写 SPFA, 辣鸡 SPFA, 毁我青春, 耗我时间, 还会 TLE (最差 $O(300 \times v^3 = v^4 = TLE = gg)$)。

然后就是推状态转移方程了。设状态 $f[i][j]$ 表示在第 i 个时间段申请换 j 次教室所耗费的体力值总和的期望的最小值, 那么这一个状态具有两个子状态: 申请成功和申请失败, 这两个子状态可以加上一维, 原来所求的值就是 $f[i][j][0/1]$ 中的较小值。根据期望的计算公式容易得到以下状态转移方程:

$$\begin{aligned} f[i][j][0] &= \min(f[i-1][j][0] + a[c[i-1]][c[i]], f[i-1][j][1] + a[c[i-1]][c[i]] * (1.0 - k[i-1]) + a[d[i-1]][c[i]] * k[i-1]); \\ f[i][j][1] &= \min(f[i-1][j-1][0] + a[c[i-1]][c[i]] * (1.0 - k[i]) + a[c[i-1]][d[i]] * k[i], f[i-1][j-1][1] + a[c[i-1]][c[i]] * (1.0 - k[i]) * (1.0 - k[i-1]) + a[d[i-1]][c[i]] * k[i-1] * (1.0 - k[i]) + a[c[i-1]][d[i]] * (1.0 - k[i-1]) * (k[i]) + a[d[i-1]][d[i]] * k[i-1] * k[i]); \end{aligned}$$

然后在 $f[n][0...m]$ 中取一个最小值作为答案即可。

那么, 这节课就到这里了。谢谢大家。

事实上, 动态规划还有很多其他的类型, 限于时间与我的姿势水平, 今天就不展开讲解了。那么以后再说?



2017年10月 第一版

2017年11月 第二版

2017年11月15日 修订版

2017年11月29日 增订版

2017年12月10日 状压DP

2017年12月13日 树形DP

2018年1月 序列型DP，修正语法错误

2018年1月 数位DP

TODO: add more questions.