ACM算法与程序设计
（四）
动态规划基础

杜育根

Ygdu@sei.ecnu.edu.cn

4. 动态规划基础

- 这节课主要讲解递推与动态规划的基础，通过学习本课程，你将能解决大部分基础的动态规划题目。

4.1. 递推

- 递推 也是经常被使用的一种简单算法。递推 是一种用若干步可重复的简单运算来描述复杂问题的方法。 递推的特点在于，每一项都和他前面的若干项有一定关联，这种关联一般可以通过 递推关系式 来表示，可以通过其前面若干项得出某项的数据。 对于递推问题的求解一般从初始的一个或若干个数据项出发，通过递推关系式逐步推进，从而得出想要的结果，这种求解问题的方法叫 递推法。其中，初始的若干数据项称为边界。

兔子问题

- 意大利人斐波那契(Leonardo Fibonacci），他描述兔子生长的数目问题： 有一对兔子，从出生后第3个月起每个月都生一对兔子，小兔子长到第三个月后每个月又生一 对兔子，假如兔子都不死，问每个月的兔子总数为多少?
- 设第i个月的兔子数量为F_i，第 i月的成年兔子数量为a_i，第i月的小兔子数量为b_i，那么$F_i = a_i + b_i$。第i月的成年兔子，由第i−1个月的小兔子长大与第 i−1月的成年兔子得到，也就是说，第i-1个月兔子的总数量，$a_i = a_{i-1} + b_{i-1} = F_{i-1}$。第 i 个月的小兔子，是由第 i−1 个月的成年兔子生的，也就是第 i−2 个月的成年兔子和小兔子，也就i−2个月兔子的总数量，$b_i = a_{i-1} = a_{i-2} + b_{i-2} = F_{i-2}$ 。这样，我们就找到了这个题目的递推关系式为：$F_i = F_{i-1} + F_{i-2}$。边界值，我们知道 F_1=1，F_2=1。那么，我们就可以写出这个题目的代码了。
- 1 f[1] = 1, f[2] = 1;
- 2 for(int i = 3;i <= n; ++i) {
- 3 f[i] = f[i-1] + f[i-2];
- 4 } //f[n] 即为所求。

信封装错

- 某人写了n封信，并有n个信封，如果所有的信都装错了信封。求所有的信都装错信封共有多少种不同情况。
- 这个题目，就没有办法像上一道题目一样不通过分析直接有答案。我们F_n来表示n 封信和 n个信封全部装错的方案数。
- 第一步，把第n封信放在第m个信封（m≠n），一共有 n−1 种方法。
- 第二步，放第 m封信的时候，有两种可能： 把第 m封信放在第 n个信封里，这样对于剩下的 n−2 封信全部放错方案数就是 F_{n-2}种。 不让第 m 封信放在第 n 个信封里。这样对于除第n封以外的n−1 封信，放到剩余的n−1 个信封且全部放错，就F_{n-1}种方案数。（不理解这一步的同学，可以想一下，这 n−1 封信里面，除第 m 封信以外每个信都不能放对应的正确信封，第m个信不能放第n 个信封，这样n−1 封信，每个信有且仅有一个对应的不能放置的信封，那么方案数就是F_{n-1} ）
- 这样，这一步共有$F_{n-2}+F_{n-1}$种方案数。 到这里，我们已经可以得出这个递推式了，根据乘法原理：$F_n=(n-1)*(F_{n-1}+F_{n-2})$ 。 我们再想一下，代码中的初始条件，也就是边界值。在只有一封信的时候，不可能装错，那么 F_1=0；有两封信的时候，装错的方案数为 F_2=1。现在我们初始条件和递推公式全有了，那么可以写代码了。
- 1 for(int i = 3; i <= n; ++i) {
- 2 f[i] = (i-1) * (F[i-1] + F[i-2]);
- 3 } // f[n]即为答案

习题：马拦过河卒

- 棋盘上A点有一个过河卒,需要走到目标B点.卒行走的规则：可以向下、或者向右.同时在棋盘上C点有一个对方的马,该马所在的点和所有跳跃一步可达的点称为对方马的控制点，因此称之为“马拦过河卒”。注：马走日字。
- 棋盘用坐标表示,A点（0,0）、B点（n,m）（其中：n,m为不超过20的正整数）,同样马的位置坐标是需要给出的.
- 任务：计算出卒从A点能够到达B点的路径的条数,假设马的位置是固定不动的,并不是卒走一步马走一步.
- 【输入格式】：输入文件中仅一行为四个整数,分别表示B点坐标和马的坐标.
- 【输出格式】：输出文件中仅一行为一个整数,表示所有的路径条数.
- 样例输入：
- 6 6 3 3
- 样例输出：
- 6

思路

- 根据题目的要求，卒只能向下或者向右走，那么想要到达棋盘上的一个点，有两种方式：从左边的格子过来，或者从上边的格子过来。所以，过河卒到达某点的路径数目等于到达与其相邻的左边点和上边点的路径数目和。我们用 S[i,j] 来表示到达点(i,j) 的路径数目。所以递推式为：S[i,j]=S[i−1,j]+S[i,j−1]。我们根据递推式发现，我们可以用逐行或逐列的递推方法求出从起点到终点的路径数目。我们来想一下边界条件，因为（0,0）是卒的起始位置，那么S[0,0]=1。

习题：墙壁涂色

- 小明觉得白色的墙面好单调，他决定给房间的墙面涂上颜色。他买了 3种颜料分别是红、黄、蓝，然后把房间的墙壁竖直地划分成 n个部分，小明希望每个相邻的部分颜色不能相同。他想知道一共有多少种给房间上色的方案。
- 例如，当 n = 5 时，下面就是一种合法方案。

蓝	红	黄	红	黄

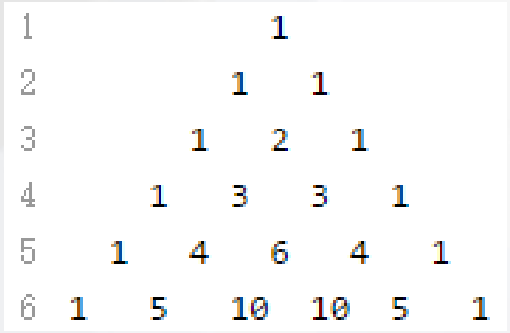
- 由于墙壁是一个环形，所以下面这个方案就是不合法的。

蓝	红	黄	红	蓝

- 输入格式: 一个整数 n，表示房间被划分成多少部分。 (1≤n≤50)
- 输出格式: 一个整数，表示给墙壁涂色的合法方案数。
- 样例输入
- 4
- 样例输出
- 18

习题：杨辉三角

- 杨辉三角是二项式系数在三角形中的一种几何排列。它的每个数等于它上方两数之和，每行数字左右对称，由1开始逐渐变大。



- 请求出杨辉三角的第 n行，第 m项的数字是什么。
- 输入格式：第一行输入两个整数 n，m代表行数和列数。 （1≤n,m≤50）
- 输出格式 输出一个整数，代表杨辉三角的第n 行，第 m 项的数字。
- 样例输入
- 6 3
- 样例输出
- 10

4.2. 动态规划入门

- 动态规划 是编程解题的一种重要手段。1951 年美国数学家 R.Bellman 等人，根据一类多阶段问题的特点，把多阶段决策问题变换为一系列互相联系的单阶段问题，然后逐个加以解决。与此同时，他提出了解决这类问题的“最优化原理”，从而创建了解决最优化问题的一种新方法：动态规划。
- 动态规划算法通常用于求解具有某种最优性质的问题。在这类问题中，可能会有许多可行解。每一个解都对应于一个值，我们希望找到具有最优值的解。 我们可以用一个表来记录所有已解的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。具体的动态规划算法多种多样，但它们具有相同的填表格式。

动态规划的基本概念

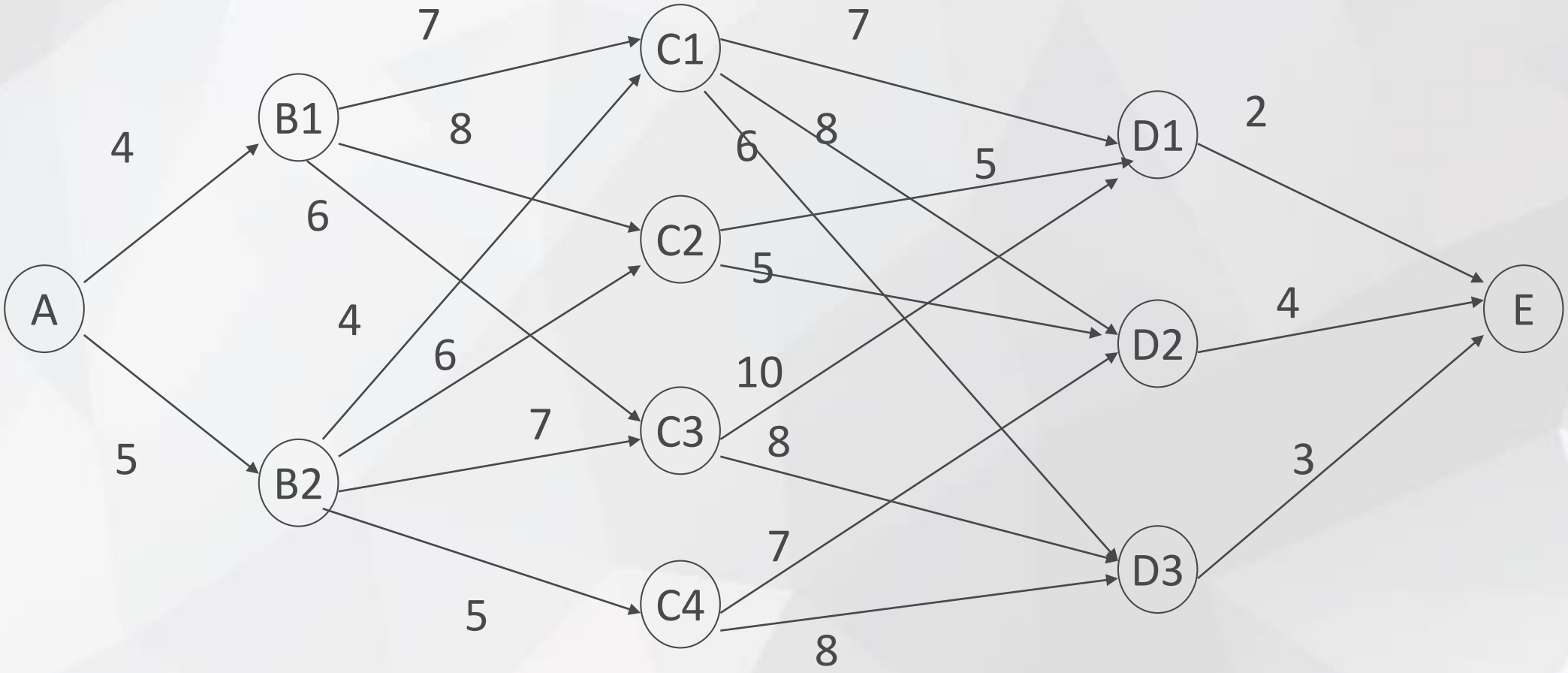
- 阶段：把所给问题的求解过程恰当地分成若干个相互联系的阶段，以便于求解。过程不同，阶段数就可能不同。描述阶段的变量称为阶段变量，常用 k表示。阶段的划分，一般是根据时间和空间的自然特征来划分，但要便于把问题的过程转化为多阶段决策的过程。
- 状态：状态表示每个阶段开始面临的自然状况或客观条件，它不以人们的主观意志为转移，也称为不可控因素。通常一个阶段有若干个状态，状态通常可以用一个或一组数来描述，称为状态变量。
- 决策：表示当过程处于某一阶段的某个状态时，可以做出不同的决定，从而确定下一阶段的状态，这种决定称为决策。不同的决策对应着不同的数值，描述决策的变量称决策变量。
- 状态转移方程：动态规划中本阶段的状态往往是上一阶段的状态和上一阶段的决策的结果，由第i段的状态 f(i) ,和决策 u(i)来确定第 i+1段的状态。状态转移表示为F(i+1)=T(f(i),u(i))，称为状态转移方程。
- 策略：各个阶段决策确定后，整个问题的决策序列就构成了一个策略，对每个实际问题，可供选择的策略有一定范围，称为允许策略集合。允许策略集合中达到最优效果的策略称为最优策略。

动态规划必须满足最优化原理与无后效性

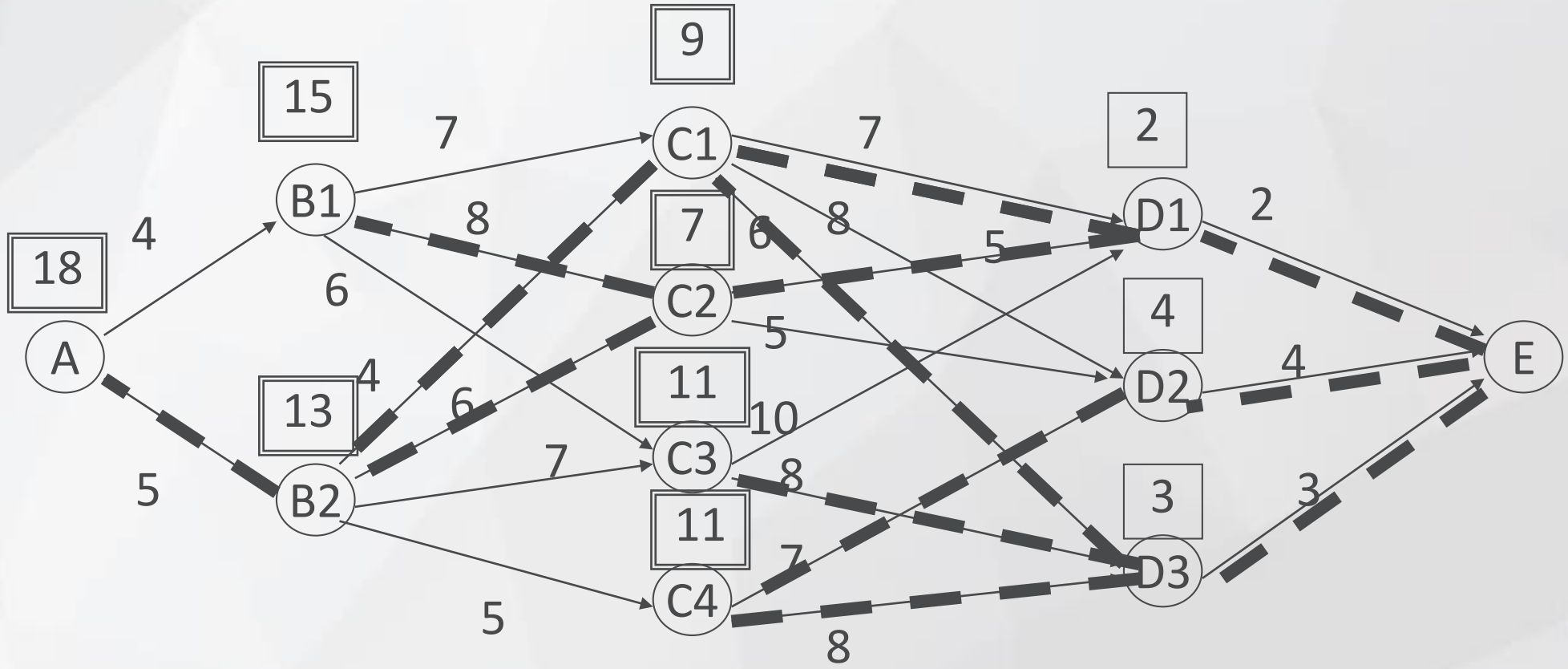
- 最优化原理：“一个过程的最优决策具有这样的性质：即无论其初始状态和初始决策如何，其今后诸策略对以第一个决策所形成的状态作为初始状态的过程而言，必须构成最优策略”。也就是说一个最优策略的子策略，也是最优的。
- 无后效性：如果某阶段状态给定后，则在这个阶段以后过程的发展不受这个阶段以前各个状态的影响。

例 最短路径问题

现有一张地图，各结点代表城市，两结点间连线代表道路，线上数字表示城市间的距离。如图1所示，试找出从结点A到结点E的最短距离。



最短路求解



最短路径:A—B2—C1—D1—E　　最短路距离:18

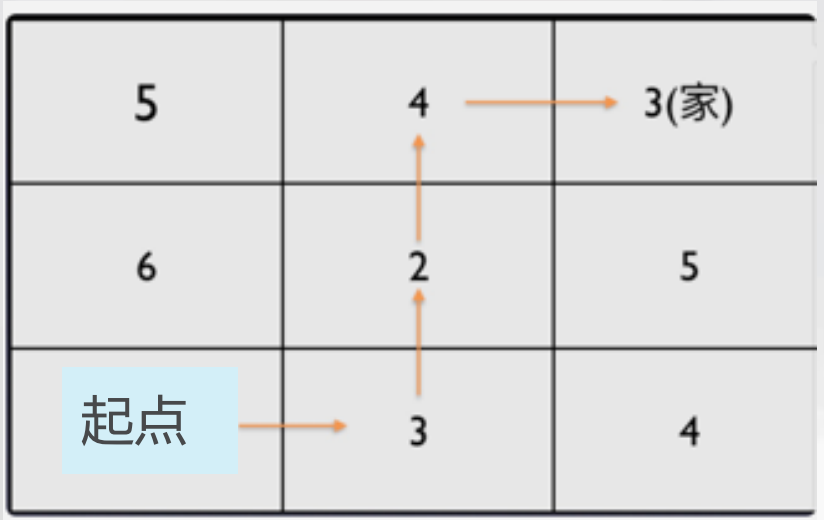
A—B2—C1—D3—E

A—B2—C2—D1—E

回家

- 小明要回家，已知小明在(1,1) 位置，家在(n,n) 坐标处。小明走到一个点 (i,j) 会花费一定的体力aij，而且小明只会往家的方向走，也就是只能往上，或者往右走。小明想知道他回到家需要花费的最少体力是多少。如右下图所示，格子中的数字代表走到该格子花费的体力：　　对于该图来说，最优策略已在图上标出，共花费体力为：3+2+4+3=12。

把走到一个点看做一个状态，只有两种方式，一个是从下往上面走到该点，一种是从左边走到该点。那么点(i,j) 要么是从(i−1,j) 走到 (i,j)，要么是从点(i,j−1) 走到(i,j)。所以从哪个点走到(i,j) 就是一个决策，我们用dp(i,j) 来代表走到点(i,j) 一共花费的最少体力。 我们需要花费最少力气走到家，所以可以得到状态转移方程：$dp(i,j)=\min(dp(i-1,j),dp(i,j-1))+aij$ 。根据转移方程我们可以推出走到每个点花费的最少体力。 对于图中的边界点，要在转移前加上判断是否为边界，如：点(1,3) 只能从点(1,2) 走过来，点(3,1) 只能从点(2,1) 走过来。



代码

- 动态规划的题目的核心是写出状态转移方程，写出转移方程那么代码实现就变得简单多了。大部分的动态规划题目，在计算出转移方程后，可以用类似于递推的循环嵌套，来写出。

```
int a[100][100]; // a数组代表走到点(i,j)花费的体力
int dp[100][100]; // dp数组代表走到点(i,j)一共花费的最少体力
dp[1][1] = 0;
for (int i = 1; i <= n; ++i) {
    for (int j = 1;j <= n; ++j) {
        if (i == 1 && j == 1) {
            continue;
        } else if (i == 1) { //边界点
            dp[i][j] = dp[i][j-1] + a[i][j];
        } else if (j == 1) { //边界点
            dp[i][j] = dp[i-1][j] + a[i][j];
        } else {
            dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + a[i][j]; //转移方程
        }
    }
}
```

过河

- 在一个夜黑风高的晚上，有 n个小朋友在桥的这边，现在他们需要过桥，但是由于桥很窄，每次只允许不超过两人通过，他们只有一个手电筒，所以每次过桥后，需要有人把手电筒带回来，第 i号小朋友过桥的时间为a[i]，两个人过桥的总时间为二者中时间长者。问所有小朋友过桥的总时间最短是多少。
- 我们先将所有人按花费时间递增进行排序，假设前i个人过河花费的最少时间为opt[i]，那么考虑前i−1个人已经过河的情况，即河这边还有 1个人，河那边有i−1 个人，并且这时候手电筒肯定在对岸，所以opt[i]=opt[i−1]+a[1]+a[i]（让花费时间最少的人把手电筒送过来，然后和第 i个人一起过河）。 如果河这边还有两个人，一个是第 i号，另外一个无关，河那边有i−2 个人，并且手电筒肯定在对岸，所以 opt[i] = opt[i-2] +a[1]+a[i]+2×a[2]（让花费时间最少的人把电筒送过来，然后第 i个人和另外一个人一起过河，由于花费时间最少的人在这边，所以下一次送手电筒过来的一定是花费次少的，送过来后花费最少的和花费次少的一起过河，解决问题），所以 opt[i] = min(opt[i-1]+ a[1] + a[i] ,opt[i-2] + a[1]+ a[i] + 2×a[2])。

习题：捡水果

- 小明在玩一款游戏，他在一个山顶，现在他要下山，山上有许多水果，小明每下一个高度就可以捡起一个水果，并且获得水果的能量。山的形状是数字堆成的三角型，数字代表水果的能量。每次下一个高度，小明需要选择是往左下走，还是往右下走。现在，小明希望你能帮他计算出下山能获得的最大能量。
- 输入格式：第一行输入一个 n，代表山的高度。（1<n<=1000）接下来 n 行，第 i+1 行有i 个数字，代表水果的能量，水果能量为正整数且不大于1000。
- 输出格式：输出一个数字，代表下山一共获得的最大能量，占一行。
- 样例输入
- 4
- 3
- 1 2
- 6 2 3
- 3 5 4 1
- 样例输出
- 15

习题：逃生

小明在玩一款逃生的游戏。在一个n×m的矩形地图上，小明位于其中一个点。地图上每个格子有加血的药剂，和掉血的火焰，药剂的药效不同，火焰的大小也不同，每个格子上有一个数字，如果格子上的数字是正数说明是一个药剂代表增加的生命值，如果是负数说明是火焰代表失去的生命值。小明初始化有v点血量，他的血量上限是c，任何时刻他的生命值都不能大于血量上限，如果血量为 0 则会死亡，不能继续游戏。矩形地图上的四个角(1,1)，(1,m)，(n,1)，(n,m)为游戏的出口。游戏中只要选定了一个出口，就必须朝着这个方向走。例如，选择了左下的出口，就只能往左和下两个方向前进，选择了右上的出口，就只能往右和上两个方向前进，左上和右下方向的出口同理。

如果成功逃生，那么剩余生命值越高，则游戏分数越高。为了能拿到最高分，请你帮忙计算如果成功逃生最多能剩余多少血量，如果不能逃生输出 −1。

输入格式：第一行依次输入整数 n，m，x，y，v，c（1<n,m≤1000，1≤x≤n，1≤y≤m，1≤v≤c≤10000），其中 n,m 代表地图大小，(x,y) 代表小明的初始位置，v 代表小明的初始化血量，c 代表小明的生命值上限。接下来 n行，每行有 m 个数字，代表地图信息。（每个数字的绝对值不大于100，地图中小明的初始位置的值一定为 0）

输出格式: 一行输出一个数字，代表成功逃生最多剩余的血量，如果失败输出 −1。

样例输入	样例输出
4 4 3 2 5 10 1 2 3 4 -1 -2 -3 -4 4 0 2 1 -4 -3 -2 -1	10

习题：小明的新游戏

- 工作空闲之余，小明经常带着同事们做游戏，最近小明发明了一个好玩的新游戏：n 位同事围成一个圈，同事 A 手里拿着一个兔妮妮的娃娃。小明喊游戏开始，每位手里拿着娃娃的同事可以选择将娃娃传给左边或者右边的同学，当小明喊游戏结束时，停止传娃娃。此时手里拿着娃娃的同事即是败者。
- 玩了几轮之后，小明想到一个问题：有多少种不同的方法，使得从同事 A 开始传娃娃，传了 m 次之后又回到了同事 A 手里。两种方法，如果接娃娃的同事不同，或者接娃娃的顺序不同均视为不同的方法。例如1−>2−>3−>1 和 1->3->2->1 是两种不同的方法。
- 输入格式: 输入一行，输入两个整数n,m(3≤n≤30,1≤m≤30)，表示一共有 n 位同事一起游戏，一共传 m 次娃娃。
- 输出格式: 输出一行，输出一个整数，表示一共有多少种不同的传娃娃方法。
- 样例输入
- 3 3
- 样例输出
- 2

习题：小明的城堡之旅

- 蒜国地域是一个 n 行 m 列的矩阵，下标均从 1 开始。蒜国有个美丽的城堡，在坐标 (n,m) 上，小明在坐标（1,1）的位置上。小明打算出发去城堡游玩，游玩结束后返回到起点。在出发去城堡的路上，小明只会选择往下或者往右走，而在返回的路上，小明只会选择往上或者往左走，每次只能走一格。已知每个格子上都有一定数量的蒜味可乐，每个格子至多经过一次。
- 现在小明请你来帮他计算一下，如何计划来回行程，可以收集到最多的蒜味可乐。
- 输入格式: 第一行输入两个整数n,m(1≤n,m≤50)，表示蒜国是一个 n 行 m列的矩阵。
- 接下来输入 n行，每行输入 m 个整数，代表一个 n×m 的矩阵，每个整数代表对应位置上的蒜味可乐数量，每行的每两个整数之间用一个空格隔开。其中小明的位置和城堡的位置上没有蒜味可乐，用 0 表示，其余位置上的整数范围在 [1,100] 内。
- 输出格式：输出一行，输出一个整数，表示小明在来回路上能收集到的蒜味可乐的最大值。
- 样例输入
- 3 3
- 0 2 9
- 4 8 6
- 2 7 0
- 样例输出
- 36

4.3. 01 背包

- 背包问题 给定一组物品，每种物品都有自己的重量（体积）和价值，现有一个背包，在受限制的重量（或者容积）下，取若干物品，使得总价值最大。这一类问题，被称为背包问题。
- 题目描述
- 当前有 N件物品和一个承重量为 V的背包。已知第 i件物品的重量是 w[i]，价值是 v[i]。 由于每种物品有且仅有一件，因此只能选择放或不放，我们称之为 01 背包问题。
- 现在你需要选出若干件物品，在它们的重量之和不超过 C的条件下，使得价值总和尽可能大。 对于每个物品是否要装入背包，我们自然可以进行暴力枚举或搜索，但是如果要暴力地去做，那么时间复杂度会非常的高，这时候需要一种更加优化的算法——动态规划。

01背包问题解析

- 对于 01 背包，先确定这个问题的状态。共有 N个物品，背包总承重为 V，那么可以根据物品和容量来确定一个状态。前 i个物品，放在背包里，总重量不超过 j的前提下，所获得的最大价值为dp[i][j]。 是否将第 i个物品装入背包中，就是决策。为了使价值最大化，如果第 i个物品放入背包后，总重量不超过限制且总价值比之前要大，那么就将第 i个物品放入背包。根据这个逻辑写出转移方程：
- $\forall\ j<w[i], dp[i][j]=dp[i-1][j]$
- $\forall\ w[i]\leq j\leq C, dp[i][j]=max(dp[i-1][j], dp[i-1][j-w[i]]+v[i])$

- 为了更好地理解这个转移方程，看一个具体的例子。有 5个物品，编号1–5，重量分别是[2,2,6,5,4]，价值分别是[6,3,5,4,6]，背包的总承重为10。 列出本问题中所有的状态变量，也就是转移方程中的dp 变量，且初始化为 0。当第一个物品放入背包，因为第一个物品重量为2，价值为6，只要背包容量大于等于 2，都可以放第一个物品，根据我们的转移方程
- dp[i][j]=max(dp[i−1][j],dp[i−1][j−w[i]]+v[i])，那么：

重量(w[i])	价值v[i]	编号(i)	0	1	2	3	4	5	6	7	8	9	10
		0	0	0	0	0	0	0	0	0	0	0	0
2	6	1	0	0	6	6	6	6	6	6	6	6	6
2	3	2											
6	5	3											
5	4	4											
4	6	5											

根据我们的转移方程继续放第二个物品，容量为2，3 的时候：$dp[i-1][j-w[i]]+v[i] < dp[i-1][j]$，则 $dp[i][j] = dp[i-1][j]$。

容量大于 3 的时候：$dp[i-1][j-w[i]]+v[i] > dp[i-1][j]$，则$dp[i][j] = dp[i-1][j-w[i]]+v[i]$。$dp$ 变量被更新为：

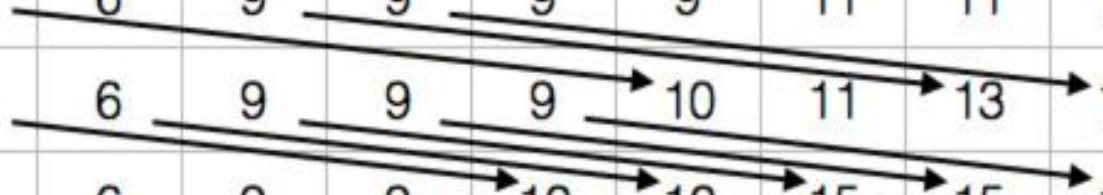
重量(w[i])	价值v[i]	编号(i)	0	1	2	3	4	5	6	7	8	9	10
		0	0	0	0	0	0	0	0	0	0	0	0
2	6	1	0	0	6	6	6	6	6	6	6	6	6
2	3	2	0	0	6	6	9	9	9	9	9	9	9
6	5	3											
5	4	4											
4	6	5											

当放第 3 个物品的时候，第三个物品的重量 $w[i]=6$，所以在背包容量 $j<w[i]$ 的时候，$dp[i][j]=dp[i-1][j]$。在背包容量 $j \geq v[i]$ 的时候 $dp[i][j]=max(dp[i-1][j], dp[i-1][j-w[i]]+v[i])$。

重量(w[i])	价值v[i]	编号(i)	0	1	2	3	4	5	6	7	8	9	10
		0	0	0	0	0	0	0	0	0	0	0	0
2	6	1	0	0	6	6	6	6	6	6	6	6	6
2	3	2	0	0	6	6	9	9	9	9	9	9	9
6	5	3	0	0	6	6	9	9	9	9	11	11	14
5	4	4											
4	6	5											

以此类推，把剩余的物品全部根据转移方程放入背包后，dp 变量为：

重量(w[i])	价值v[i]	编号(i)	0	1	2	3	4	5	6	7	8	9	10
		0	0	0	0	0	0	0	0	0	0	0	0
2	6	1	0	0	6	6	6	6	6	6	6	6	6
2	3	2	0	0	6	6	9	9	9	9	9	9	9
6	5	3	0	0	6	6	9	9	9	9	11	11	14
5	4	4	0	0	6	6	9	9	9	10	11	13	14
4	6	5	0	0	6	6	9	9	12	12	15	15	15



通过上面的表格，可以知道当这 5 个物品放入容量为 10 的背包中，最大的价值是 15，也就是 $dp[5][10] = 15$。

代码

```
1   for (int i = 1; i <= N; ++i) {
2       for (int j = 0; j <= V; ++j) {
3           if(j >= w[i]) {
4               dp[i][j] = max(dp[i - 1][j - w[i]] + v[i], dp[i - 1][j]);
5           }
6           else {
7               dp[i][j] = dp[i-1][j];
8           }
9       }
10  }
```

时间上是两重循环，时间复杂度为 $\mathcal{O}(NV)$。

空间是二维的，空间复杂度也为 $\mathcal{O}(NV)$。

优化空间复杂度

空间复杂度还可以优化到 $\mathcal{O}(V)$，但时间复杂度已经很难再优化了。

当前计算第 i 件物品，总重量为 v 的最大价值。我们注意到，它需要的状态是 $dp[i-1][0\ldots v]$。

如果从大到小，也就是从 V 到 0 枚举，则当前引用的 $dp[v]$ 和 $dp[v-w[i]]$，仍然是计算第 $i-1$ 件物品的结果，即 $dp[i-1][v], dp[i-1][v-w[i]]$，那么这一维的状态是冗余的。

故我们可以化简转移方程：$dp[j] = max(dp[j-w[i]]+v[i], dp[j])$。

```
for (int i = 1; i <= n; ++i)
    for (int j = v; j >= w[i]; --j)
        dp[j] = max(dp[j - w[i]] + v[i], dp[j]);
```

这个做法的空间复杂度为 $\mathcal{O}(V)$。

习题：小明的购物袋 1

- 小明去超市购物，他有一只容量为 V 的购物袋，同时他买了 n 件物品，已知每件物品的体积 vi 。小明想知道，挑选哪些物品放入购物袋中，可以使袋子剩余的空间最小。
- 输入格式：第一行输入一个整数 V（1≤V≤20,000），表示购物袋的容量。
- 第二行输入一个整数 n（1≤n≤30），表示小明购买的 n 件物品。
- 接下来输入 n 行，每行输入一个整数 vi（1≤vi≤10,000），表示第 i 件物品的体积。
- 输出格式：输出一行，输出一个整数，表示购物袋最小的剩余空间。
- 样例输入
- 20
- 5
- 7
- 5
- 7
- 3
- 7
- 样例输出
- 1

习题：小明的购物袋 2

- 小明去超市购物，他有一只容量为 V 的购物袋，同时他想买 n 件物品，已知每件物品的体积 vi和重要度 pi。小明想知道，挑选哪些物品放入购物袋中，可以使得买到的物品重要度之和最大，且物品体积和不超过购物袋的容量。
- 输入格式
- 第一行输入两个整数 V（1≤V≤1000）和 n（1≤n≤100）。代表购物袋的总体积为 V，小明一共想买 n 件物品。接下来输入 n 行，每行输入两个整数 vi和 pi（1≤vi,pi≤100），分别表示每件物品的体积和重要度。
- 输出格式
- 输出一行，输出一个整数，表示小明能买到物品的最大重要度之和。
- 样例输入
- 50 4
- 1 5
- 60 99
- 49 8
- 33 7
- 样例输出
- 13

4.4. 完全背包问题

当前有 N 种物品，第 i 种物品的体积是 c_i，价值是 w_i。

每种物品的数量都是无限的，可以任意选择若干件，我们称之为完全背包问题。

现有容量为 V 的背包，请你放入若干物品，使总体积不超过 V，并且总价值尽可能大。

解析

每种物品可以取 0 件，1 件，2 件，3 件，甚至更多。我们仍然可以写出状态转移方程：

$$dp[i][v] = max(dp[i][v - c_i] + w_i, dp[i - 1][v])$$

更简单的：

$$dp[v] = max(dp[v - c_i] + w_i, dp[v])$$

这与 01 背包问题几乎是一样的。而这道题目规定物品的数量是无限的，这个细节怎样处理？

注意到 $dp[i][v] = max(dp[i - 1][v], dp[i - 1][v - c_i] + w_i, dp[i - 1][v - c_i \times 2] + w_i \times 2 \ldots)$，

而 $dp[i][v - c[i]] = max(dp[i - 1][v - c_i], dp[i - 1][v - c_i \times 2] + w_i, dp[i - 1][v - c_i \times 3] + w_i \times 2 \ldots)$，也就是说，我们完全可以用 $dp[i][v - c[i]]$ 的信息去更新 $dp[i][v]$。

```
for (int i = 1; i <= n; ++ i)
    for (int j = c[i]; j <= v; ++ j)
        dp[j] = max(dp[j - c[i]] + w[i], dp[j]);
```

不难发现，与 01 背包相比，完全背包只是第二重循环的顺序发生了改变。

我们保证用较小容量的最大价值去更新较大容量的最大价值，不重不漏。

该算法时间复杂度 $\mathcal{O}(NV)$，空间复杂度 $\mathcal{O}(V)$。

习题：小明的购物袋 3

- 小明去超市购物，他有一只容量为 V 的购物袋，同时他想买 n种物品，已知每种物品的体积 vi和重要度 pi 。小明想知道，怎么挑选物品放入购物袋中，可以使得买到的物品重要度之和最大，且物品体积和不超过购物袋的容量。注意超市中每种物品的数量无限多。
- 输入格式：第一行输入两个整数 n, V (1≤n≤1,000,1≤V≤10,000)。
- 接下来输入 n 行，每行输入两个整数 vi和pi(1≤vi,pi≤10,000)，分别表示第 i 种物品的体积和重要度。
- 输出格式：输出一行，输出一个整数，表示能买到物品的最大重要度之和。
- 样例输入
- 4 20
- 3 7
- 2 5
- 4 6
- 5 9
- 样例输出
- 50

4.5. 多重背包问题

有 N 种物品，第 i 种物品的体积是 c_i，价值是 w_i，每种物品的数量都是有限的，为 n_i。

现有容量为 V 的背包，请你放入若干物品，在总体积不超过 V 的条件下，使总价值尽可能大。

朴素算法

我们可以把 n_i 个物品逐个拆分，则得到 $\sum n_i$ 个物品。原问题转化为 01 背包求解，时间复杂度是 $\mathcal{O}(V \times \sum n)$。

我们也可以在转移的过程中枚举 k，代表第 i 个物品选取的数量。

$$dp[i][v] = max(dp[i-1][v - k * c_i] + k * w_i), 0 \leq k \leq n_i$$

这样的时间复杂度也是 $\mathcal{O}(V \times \sum n)$。

优化

我们可以考虑二进制的思想，将第 i 种物品分成若干件物品，可以有 $(c_i, w_i), (c_i \times 2, w_i \times 2), (c_i \times 4, w_i \times 4)$，等等。每件物品有一个系数，分别为 $1, 2, 4, \cdots, 2^{k-1}, n - 2^k + 1$，$k$ 是满足 $n - 2^k + 1 > 0$ 的最大整数。

例如，n 为 13，拆分成 $1, 2, 4, 6$ 四件物品。又有：$8 = 2 + 6$，$11 = 1 + 4 + 6$，根据二进制的性质，$0 \ldots 13$ 都可以由这四件物品组合得到。

于是，n 件物品，就拆分成至多 $\log n$ 件物品。

再转化为 01 背包问题求解，原问题复杂度降低到 $\mathcal{O}(V \sum logn)$。

补充

多重背包问题也有基于朴素的状态转移方程的 $\mathcal{O}(VN)$ 的算法，借助单调队列可以使每个状态的值以 $\mathcal{O}(1)$ 的时间复杂度求解出来，但由于要使用单调队列这一数据结构优化 DP，我们就不再进行详细的解释了。

习题：平分娃娃

- 小明酷爱收集萌萌的娃娃。小明收集了 6 种不同的娃娃，第 i 种娃娃的萌值为 i（1≤i≤6）。现在已知每种娃娃的数量 mi，小明想知道，能不能把娃娃分成两组，使得每组的娃娃萌值之和相同。
- 输入格式
- 输入一行，输入 6 个整数，代表每种娃娃的数量 mi（0≤mi≤20,000）。
- 输出格式
- 输出一行。如果能把所有娃娃分成萌值之和相同的两组，请输出Can be divided.，否则输出Can’t be divided.。
- 样例输入
- 2 0 1 1 2 1
- 样例输出
- Can’t be divided.

4.6. LIS 与 LCS

LIS（最长上升子序列）

LIS（Longest Increasing Subsequence）是动态规划的一个经典应用。LIS 问题有两种算法，两种方法的时间复杂度分别是 $\mathcal{O}(n^2)$ 和 $\mathcal{O}(n \log n)$。我们这里介绍时间复杂度为 $\mathcal{O}(n^2)$ 的算法，这种算法易于理解且代码简单。

在原序列取任意多项，不改变他们在原来数列的先后次序，得到的序列称为原序列的 **子序列**。最长上升子序列，就是给定序列的一个最长的、数值从低到高排列的子序列，最长子序列不一定是唯一的。例如，序列 $2, 1, 5, 3, 6, 4, 6, 3$ 的最长上升子序列为 $1, 3, 4, 6$ 和 $2, 3, 4, 6$，长度均为 4。

先确定动态规划的状态，这个问题可以用序列某一项作为结尾来作为一个状态。用 $dp[i]$ 表示一定以第 i 项为结尾的最长上升子序列。用 $a[i]$ 表示第 i 项的值，如果有 $j < i$ 且 $a[j] < a[i]$，那么把第 i 项接在第 j 项后面构成的子序列长度为：$dp[i] = dp[j] + 1$。

要使 $dp[i]$ 为以 i 结尾的最长上升子序列，需要枚举所有满足条件的 j。所以转移方程是：

$$\forall\, 1 \le j < i \;\&\&\; a[j] < a[i], dp[i] = max(dp[i], dp[j] + 1)$$

根据转移方程的到如下表

i	1	2	3	4	5	6	7	8
$a[i]$	2	1	5	3	6	4	6	3
$dp[i]$	1	1	2	2	3	3	4	2

最后，dp 数组里面的最大值就是最长上升子序列的长度了。

C++代码如下

```cpp
int dp[MAX_N], a[MAX_N], n;
int ans = 0;  // 保存最大值

for (int i = 1; i <= n; ++i) {
    dp[i] = 1;
    for (int j = 1; j < i; ++j) {
        if (a[j] < a[i]) {
            dp[i] = max(dp[i], dp[j] + 1);
        }
    }
    ans = max(ans, dp[i]);
}

cout << ans << endl;  // ans 就是最终结果
```

LIS改进版

i	1	2	3	4	5	6	7	8
$a[i]$	2	1	5	3	6	4	6	3
$dp[i]$	1	1	2	2	3	3	4	2

在 dp 值相同的情况下，保留较小的数显然更好。因为后面的数若能跟较大的数构成上升子序列，也一定能能较小的数构成上升子序列，反之则不一定。例如 $a_3 = 5$ 与 $a_4 = 3$ 的 dp 均为 2，但 $a_6 = 4$ 不能与 $a_3 = 5$ 构成上升子序列，而可以和 $a_4 = 3$ 构成上升子序列。

因此，不同的 dp 值只需要存一个对应的最小值，将这个最小值顺序排列，他们一定是升序的。

于是，借助二分查找的方式，就可以很快查到更新的值，整体时间复杂度 $\mathcal{O}(nlogn)$。

优化C++代码

```
int ans[MAX_N], a[MAX_N], dp[MAX_N], n;  // ans 用来保存每个 dp 值
                                         //对应的最小值，a 是原数组
int len; // LIS 最大值
ans[1] = a[1];
len = 1;
for (int i = 2; i <= n; ++i) {
    if (a[i] > ans[len]) {
        ans[++len] = a[i];
    } else {
        int pos = lower_bound(ans, ans + len, a[i]) - ans;
                                  //二分查找目前ans数组中大于等于a[i]位置
        ans[pos] = a[i];
    }
}
cout << len << endl;  // len 就是最终结果
```

LCS最长公共子序列

LCS：给定两个序列 S_1 和 S_2，求二者公共子序列 S_3 的最长的长度。

有了前面的基础，可以发现这个问题仍然可以按照序列的长度来划分状态，也就是 S_1 的前 i 个字符和 S_2 的前 j 个字符的最长公共子序列长度，记为 $lcs[i][j]$。

如果 S_1 的第 i 项，和 S_2 的第 j 项相同，那么 $S_1[i]$ 与 $S_2[j]$ 作为公共子序列的末尾，则 $lcs[i][j] = lcs[i-1][j-1] + 1$, 也可以不让 $S_1[i]$ 与 $S_2[j]$ 作为公共子序列的末尾，则 $lcs[i][j] = max(lcs[i][j-1], lcs[i-1][j])$。不难证明 $max(lcs[i][j-1], lcs[i-1][j]) \leq lcs[i-1][j-1] + 1$。**那么转移方程是：**

$$\forall\ S_1[i] = S_2[j], lcs[i][j] = lcs[i-1][j-1] + 1。$$

如果 S_1 的第 i 项，和 S_2 的第 j 项不相同，那么 $S_1[i]$ 与 $S_2[j]$ 必然不是公共子序列的末尾。**那么转移方程是：**

$$\forall\ S_1[i] \neq S_2[j], lcs[i][j] = max(lcs[i][j-1], lcs[i-1][j])。$$

举例

举一个例子，两个序列 $S_1 = abcfbc$ 和 $S_2 = abfcab$，根据转移方程可以得出 $lcs[i][j]$

lcs	0	$1(a)$	$2(b)$	$3(c)$	$4(f)$	$5(b)$	$6(c)$
0	0	0	0	0	0	0	0
$1(a)$	0	1	1	1	1	1	1
$2(b)$	0	1	2	2	2	2	2
$3(f)$	0	1	2	2	3	3	3
$4(c)$	0	1	2	3	3	3	4
$5(a)$	0	1	2	3	3	3	4
$6(b)$	0	1	2	3	3	4	4

最后的 $lcs[6][6]$，就是 `"abcfbc"` 和 `"abfcab"` 的最长公共子序列的长度。

习题：小明跳木桩

- 小明面前有一排 n 个木桩，木桩的高度分别是h1,h2,h3...hn。小明第一步可以跳到任意一个木桩，接下来的每一步小明不能往回跳只能往前跳，并且跳下一个木桩的高度不大于 当前木桩。小明希望能踩到尽量多的木桩，请你帮小明计算，最多能踩到多少个木桩。
- 输入格式
- 第一行输入一个整数 n 代表木桩个数。第二行输入 n 个整数h1,h2,h3..hn，分别代表 n 个木桩的高度。(1≤n≤1000,1≤hi≤100000)
- 输出格式
- 输出一个整数，代表最多能踩到的木桩个数，占一行。
- 样例输入
- 6
- 3 6 4 1 4 2
- 样例输出
- 4

习题：删除最少的元素

- 给定有 n 个数的 A 序列：$A1, A2, A3 \ldots An$ 。对于这个序列，我们想得到一个子序列 $A_{p1}, A_{p2}, \ldots\ldots, A_{pi}, \ldots\ldots, A_{pm}(1 \le p1 < p2 < \cdots pi < \cdots < pm \le n)$，满足 $A_{p1} \ge A_{p2} \ge \ldots\ldots \ge A_{pi} \le \ldots\ldots \le A_{pm}$。从 A 序列最少删除多少元素，可以得到我们想要的子序列。
- 输入格式
- 第一行输入一个整数 n，代表 A 序列中数字的个数。接下来输入 n 个整数，代表 A1,A2 ,A3 ...An。(1≤n≤1000，1≤Ai≤10000)
- 输出格式
- 输出需要删除的元素个数，占一行。
- 样例输入
- 7
- 3 2 4 1 2 5 3
- 样例输出
- 2

习题：最长公共子序列

- 这次我们的问题非常简单，小明有两个字符串 a 和 b，小明想知道两个字符串的最长公共子序列的长度。
- 输入格式
- 第一行输入一个字符串。第二行输入一个字符串。（字符串只包含字母，每个字符串长度不超过 1000）
- 输出格式
- 输出二者的最长公共子序列的长度，占一行。
- 样例输入
- computer
- education
- 样例输出
- 2

习题：回文串

- 一个字符串如果从左往右读和从右往左读都一样，那么这个字符串是一个回文串。例如：”abcba”，”abccba”。
- 小明想通过添加字符把一个非回文字符串变成回文串。例如：”trit”，可以添加一个’i’ 变成回文串”tirit”。请你用程序计算出，对于一个给定的字符串，最少需要添加几个字符，才能变成回文串。
- 输入格式
- 输入一个长度为n(1≤n≤3000) 的字符串。 (字符串只包含字母)
- 输出格式
- 输出最少需要添加的字符个数，占一行。
- 样例输入
- trit
- 样例输出
- 1

4.7. 状态压缩动态规划

若元素数量比较小（不超过 20）时，想要存储每个元素取或不取的状态时，可以借助位运算将状态压缩。

需要借助状态压缩过程的动态规划就是状态压缩 DP（很多地方会简称为 **状压 DP**）。

取若干元素，也就是对应的位置记为 1，其余位置记为 0。例如，一共有 5 个元素 a, b, c, d, e，我们分别用 $1, 2, 4, 8, 16$ 表示这五个元素，则集合 $\{a, c, e\}$ 可以用 $(10101)_2 = 21$ 来表示，而集合 $\{b, c, d\}$ 可以用 $(01110)_2 = 14$ 表示。

对于元素个数为 n 的情况，其空间复杂度为 $\mathcal{O}(2^n)$。

例题

- 给定一个n×m 的矩阵，行数和列数都不超过 20，其中有些格子可以选，有些格子不能选。现在你需要从中选出尽可能多的格子，且保证选出的所有格子之间不相邻（没有公共边）。
- 例如下面这个矩阵（2×3 的矩阵，可选的位置标记为1，不可选的位置标记为0）：
- 1 1 1
- 0 1 0
- 最多可选 3 个互不相邻的格子，方案如下（选中的位置标记为x）：
- x 1 x
- 0 x 0

例题解析

- 我们可以自上而下，一行行地选择格子。在一行内选择格子的时候，只和上一行的选择方案有关，我们就可以将“当前放到第几行、当前行的选择方案”作为状态进行动态规划。
- 这里，我们就要用到刚刚提到的状态压缩：一行里选择格子的方案实际上是一个集合，我们要将这个集合压缩为一个整数。比如，对于一个 3列的矩阵，如果当前行的状态是$(5)_{10}=(101)_2$ ，那么就意味着当前行选择了第一个和第三个格子；类似地，如果当前行的状态是$(6)_{10}=(110)_2$，那么就意味着当前行选择了第一个和第二个格子。
- 如果上一行的状态是now，下一行的状态是prev，那么我们只需要确保上下两行的选择方案里没有重复的元素，也就是(now & prev) == 0就可以了。 此外，我们还需要判断当前行的状态是否合法，因为读入的矩阵中并不是每个格子都可以选择的，如果我们将矩阵中每行的值也用状态压缩来存储，不妨记为flag，那么当前行选择的格子集合一定包含于当前行合法格子的集合，也就是说，(now | flag) == flag必须成立。 这样，我们就可以通过枚举上一行的所有状态，来更新当前行、当前状态的最优解了。直到算完最后一行，统计一下所有状态的最大值即可。

C++代码

```
const int MAX_N = 20;
const int MAX_M = 20;
int state[MAX_N];
int dp[MAX_N + 1][1 << MAX_M];
bool not_intersect(int now, int prev) {
    return (now & prev) == 0;
}
bool fit(int now, int flag) {
    return (now | flag) == flag;
}

int count(int now) {
    int s = 0;  // 统计 now 的二进制形式中有多少个 1
    while (now) {
        s += (now & 1);
            // 判断 now 二进制的最后一位是否为 1，如果是则累加
        now >>= 1;  // now 右移一位
    }
    return s;
}
int main() {
    // 初始化所有数组
    for (int i = 1; i <= n; ++i) {
        for (int j = 0; j < m; ++j) {
            int flag;
            cin >> flag;
            state[i] |= (1 << j) * flag;   // 将 (i,j) 格子的状态放入 state[i]
                        //中，state[i] 表示第 i 行的可选格子组成的集合
        }
    }
```

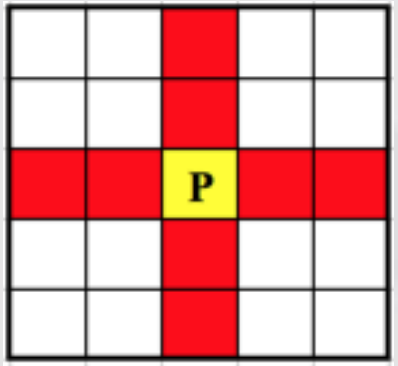
```
for (int i = 1; i <= n; ++i) {
        for (int j = 0; j < (1 << MAX_M); ++j) {  // 枚举当前行的状态
            if (!fit(j, state[i])) {
                // 如果当前行状态不合法则不执行后面的枚举
                continue;
            }
            int cnt = count(j);  // 统计当前行一共选了多少个格子
            for (int k = 0; k < (1 << MAX_M); ++k) {
                if (!fit(k, state[i - 1]) && !not_intersect(j, k)) {
                    // 判断前一行是否合法和当前行和前一行的方案是否冲突
                    dp[i][j] = max(dp[i][j], dp[i - 1][k] + cnt);
                    // 更新当前行、当前状态的最优解
                }
            }
        }
    }
    int ans = 0;  // 保存最终答案
    for (int i = 0; i < (1 << MAX_M); ++i) {
        ans = max(ans, dp[n - 1][i]);  // 枚举所有状态，更新最大值
    }
    cout << ans << endl;
    return 0;
}
```

例题 2

- 给定一个长度不超过16 的字符串 s，如果 s 中的一个子序列是回文，那么我们就可以从 s 中移除这个子序列，求最少经过多少步我们可以移除整个字符串 s。如：我们可以从“dqewfretd”中移除 “defed”，剩下的字符串即：“qwrt”。 例题解析 使用状态压缩 DP 求解，当某状态对应的子序列是回文时，答案为 1；否则是其两个互补的子集操作步骤数之和的最小值。 在更新时我们需要枚举要求解的状态的所有的子集，可以仿照如下的代码进行求解。时间复杂度是 O(3^n + n*2^n)。
- 1 for (int t = 1; t < (1 << n); t++) { // 枚举当前状态
- 2 dp[t] = IsPalindrome(t) ? 1 : inf;
- // 判断当前状态是否是回文，如果是回文则步骤数为 1
- 3 for(int i = t; i; i = (i - 1) & t) { // 枚举 t 的所有子集
- 4 dp[t] = min(dp[t], dp[i] + dp[t ^ i]); // 更新当前状态的解的最小值
- 5 }
- 6 }
- 7 printf("%d\n", dp[(1 << n) - 1]); // 输出最终答案

习题：灌溉机器人

- 农田灌溉是一项十分费体力的农活，特别是大型的农田。小明想为农民伯伯们减轻农作负担，最近在研究一款高科技——灌溉机器人。它可以在远程电脑控制下，给农田里的作物进行灌溉。
- 现在有一片N行M列的农田。农田的土壤有两种类型：类型H和类型 P，每一个格子上的土
- 壤类型相同。其中类型P的土壤硬度较大，可以用来布置灌溉机器人，但是一个格子上只能
- 布置一台。类型 H 的土壤不能布置灌溉机器人。一台灌溉机器人的灌溉区域如右图所示：
- 黄色表示灌溉机器人布置的格子，红色表示其灌溉区域，即四个方向上各外扩展两个格子。



- 小明想在农田上尽可能多布置一些灌溉机器人，但是任意一台机器人不能在任意一台机器人的
- 灌溉区域里，否则机器容易进水出故障。现在已知农田每个格子的土壤类型，请你来帮小明计算一下，小明最多能布置多少台灌溉机器人。
- 输入格式: 输入第一行输入两个正整数N,M(N≤100,M≤10)，表示农田的行和列。接下来输入 N 行，每行输入连续的 M 个字符（P或者H），中间没有空格。表示农田每个格子上的土壤类型。
- 输出格式: 输出一行，输出一个整数，表示最多能摆放的灌溉机器人的数量。
- 样例输入
- 3 4
- PHPP
- PHPP
- PHHP
- 样例输出
- 3

习题：小明的积木

- 小明酷爱搭积木，他用积木搭了 n 辆重量为 wi的小车和一艘最大载重量为 W 的小船，他想用这艘小船将 n 辆小车运输过河。每次小船运载的小车重量不能超过 W。另外，小船在运载小车时，每辆小车会对小船有一个损坏值si，当多辆小车一起运载时，该趟运载对小船的损坏值为船上所有小车的最大损坏值。
- 现在小明想知道，如何用小船运载 n 辆小车，可以使得对小船造成的总损坏值最小。
- 输入格式：第一行输入两个数 W 和 n（100≤w≤400，1≤n≤16），分别表示小船的最大载重量和小车总数。
- 接下来输入 n 行，每行输入两个整数si和 wi（1≤si ≤50，10≤wi≤100），分别表示每辆小车对小船的损坏值和每辆小车的重量。
- 输出格式：输出一行，输出一个整数，表示用小船运载 nn 辆小车，最小的总损坏值。
- 样例输入
- 90 4
- 32 50
- 15 20
- 40 50
- 13 40
- 样例输出
- 72

习题：消除字符串

- 小明喜欢中心对称的字符串，即回文字符串。现在小明手里有一个字符串 S，小明每次都会进行这样的操作：从 S中挑选一个回文的子序列，将其从字符串S中去除，剩下的字符重组成新的字符串 S。 小明想知道，最少可以进行多少次操作，可以消除整个字符串。
- 输入格式：输入一行。输入一个字符串 S（1≤length(S)≤16），字符串均由小写字母组成。
- 输出格式：输出一行，输出一个整数，表示消除整个字符串需要的最少操作次数。
- 样例输入
- abaccba
- 样例输出
- 2

习题：小明的蜡笔

- 小明收到了一个生日礼物——一盒精美的蜡笔，这可把他高兴坏了。小明在完成一道图论的题目后，拿着蜡笔想给题目上的一个无向图进行填色。无向图上一共有 n个点，编号从 0到n−1，那么该图就会有 2^n-1个非空子图。 小明想给每 i个子图进行填色，使得任意一条边连接的两个点的颜色不同，现在他想知道给第 i个子图填色，最少需要多少种不同的颜色，记为 si 。小明想请你帮他计算一下以下式子的结果：

$$\sum_{i=1}^{2^n-1} s_i \times 233^i.$$

- 输入格式：第一行输入一个整数n（1≤n≤16），表示无向图有n个点。 接下来输入一个n×n 的 01 矩阵，表示无向图边的情况。如果 a[i][j]=1，则说明 i 与 j之间有边相连，如果a[i][j]=0，则表示i与j之间没有边相连。
- 输出格式：输出一行，输出上述式子的结果，结果可能会很大，输出结果对 2^{32}取余的结果即可。
- 样例输入
  - 4
  - 0111
  - 1011
  - 1101
  - 1110
- 样例输出
  - 1595912448