第 4 章 动态规划

动态规划可以说是整个算法基础篇中，最难的一章：

- 首先，入门难。刚开始接触动态规划，你会觉得这玩意有点晦涩 + 玄学，稀里糊涂的就把问题解决了。不用担心，等做过五六道题目之后，就能理解动态规划是如何解决问题的；--- 坚持学、重复学
- 其次，题型多。不仅算法基础篇会讲到动态规划，在算法提高篇还会再讲。动态规划细分的话，可以分出十几种类型。所以，学习的成本是比较高的；
- 最后，题目难。在竞赛中，如果遇到动态规划的问题，只要不是经典题型，那么大概率就是以压轴题的形式出现。

但是，即使很难，我们也要慢慢去学习。想取得好成绩，动态规划是避不开的。不过大家放心，往后讲解的时候，我会循序渐进的进行，相信大家都是可以听得懂的。

1. 入门：从记忆化搜索到动态规划

1. 记忆化搜索

在搜索的过程中，如果搜索树中有很多重复的结点，此时可以通过一个 "备忘录"，记录第一次搜索到的结果。当下一次搜索到这个结点时，直接在 "备忘录" 里面找结果。其中，搜索树中的一个一个结点，也称为一个一个状态。

比如经典的斐波那契数列问题：

```
int f[N]; // 备忘录

int fib(int n)
{
    // 搜索之前先往备忘录里面瞅瞅
    if(f[n] != -1) return f[n];
    if(n == 0 || n == 1) return f[n] = n;

    // 返回之前，把结果记录在备忘录中
    f[n] = fib(n - 1) + fib(n - 2);
    return f[n];
}
```

2. 递归改递推

在用记忆化搜索解决斐波那契问题时，如果关注 "备忘录" 的填写过程，会发现它是从左往右依次填写的。当 i 位置前面的格子填写完毕之后，就可以根据格子里面的值计算出 i 位置的值。所以，整个递归过程，我们也可以改写成循环的形式，也就是递推：

```
int f[N]; // f[i] 表示：第 i 个斐波那契数

int fib(int n)
{
    // 初始化前两个格子
    f[0] = 0; f[1] = 1;

    // 按照递推公式计算后面的值
    for(int i = 2; i <= n; i++)
    {
        f[i] = f[i - 1] + f[i - 2];
    }

    // 返回结果
    return f[n];
}
```

3. 动态规划

动态规划（Dynamic Programming，简称DP）是一种用于解决多阶段决策问题的算法思想。它通过将复杂问题分解为更小的子问题，并存储子问题的解（通常称为“状态”），从而避免重复计算，提高效率。因此，动态规划里，蕴含着分治与剪枝思想。

上述通过**记忆化搜索**以及**递推**解决斐波那契数列的方式，其实都是动态规划。

> 注意：
>
> - 动态规划中的相关概念其实远不止如此，还会有：重叠子问题、最优子结构、无后效性、有向无环图等等。
> - 这些概念没有一段时间的沉淀是不可能完全理解的。可以等学过一段时间之后，再去接触这些概念。不过，这些概念即使不懂，也不影响做题~

在递推形式的动态规划中，常用下面的专有名词来表述：

1. 状态表示：指 `f` 数组中，每一个格子代表的含义。其中，这个数组也会称为 `dp` 数组，或者 `dp` 表。
2. 状态转移方程：指 `f` 数组中，每一个格子是如何用其余的格子推导出来的。

3. 初始化：在填表之前，根据题目中的默认条件或者问题的默认初始状态，将 `f` 数组中若干格子先填上值。

其实递推形式的动态规划中的各种表述，是可以对应到递归形式的：

- 状态表示 <---> 递归函数的意义；
- 状态转移方程 <---> 递归函数的主函数体；
- 初始化 <---> 递归函数的递归出口。

4. **如何利用动态规划解决问题**

第一种方式当然就是记忆化搜索了：

- 先用递归的思想去解决问题；
- 如果有重复子问题，就改成记忆化搜索的形式。

第二种方式，直接使用递推形式的动态规划解决：

1. 定义状态表示：

   一般情况下根据经验+递归函数的意义，赋予 `dp` 数组相应的含义。（其实还可以去蒙一个，如果蒙的状态表示能解决问题，说明蒙对了。如果蒙错了，再换一个试~）

2. 推导状态转移方程：

   根据状态表示以及题意，在 `dp` 表中分析，当前格子如何通过其余格子推导出来。

3. 初始化：

   根据题意，先将显而易见的以及边界情况下的位置填上值。

4. 确定填表顺序：

   根据状态转移方程，确定按照什么顺序来填表。

5. 确定最终结果：

   根据题意，在表中找出最终结果。

1.1 下楼梯

> **题目来源：** 洛谷
>
> **题目链接：** P10250 [GESP样题 六级] 下楼梯
>
> **难度系数：** ★

【题目描述】

> 顽皮的小明发现，下楼梯时每步可以走 1 个台阶、 2 个台阶或 3 个台阶。现在一共有 N 个台阶，你能帮小明算算有多少种方案吗？

【输入描述】

> 输入一行，包含一个整数 N。
>
> 对全部的测试点，保证 $1 \leq N \leq 60$。

【输出描述】

> 输出一行一个整数表示答案。

【示例一】

> 输入：
>
> 4
>
> 输出：
>
> 7

【示例二】

> 输入：
>
> 10
>
> 输出：
>
> 274

【解法】

因为上楼和下楼是一个可逆的过程，因此我们可以把下楼问题转化成上到第 n 个台阶，一共有多少种方案。

解法：动态规划

1. 状态表示：

   $dp[i]$ 表示：走到第 i 个台阶的总方案数。

   那最终结果就是在 $dp[n]$ 处取到。

2. 状态转移方程：

   根据最后一步划分问题，走到第 i 个台阶的方式有三种：

   a. 从 $i-1$ 台阶向上走 1 个台阶，此时走到 i 台阶的方案就是 $dp[i-1]$；

   b. 从 $i-2$ 台阶向上走 2 个台阶，此时走到 i 台阶的方案就是 $dp[i-2]$；

   c. 从 $i-3$ 台阶向上走 3 个台阶，此时走到 i 台阶的方案就是 $dp[i-3]$；

   综上所述，$dp[i]=dp[i-1]+dp[i-2]+dp[i-3]$。

3. 初始化：

   填 i 位置的值时，至少需要前三个位置的值，因此需要初始化 $dp[0]=1,dp[1]=1,dp[2]=2$，然后从 $i=3$ 开始填。

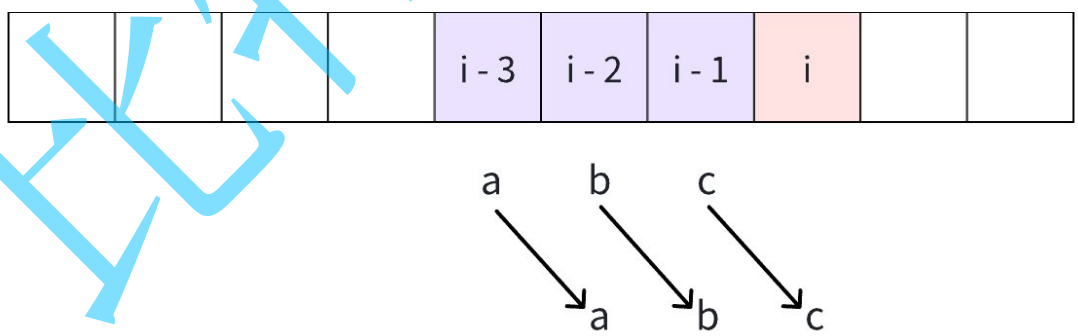
   或者初始化 $dp[1]=1,dp[2]=2,dp[3]=4$，然后从 $i=4$ 开始填。

4. 填表顺序：

   明显是从左往右。

动态规划的空间优化：

我们发现，在填写 $dp[i]$ 的值时，我们仅仅需要前三个格子的值，第 $i-4$ 个及其之前的格子的值已经毫无用处了。因此，可以用三个变量记录 i 位置之前三个格子的值，然后在填完 i 位置的值之后，滚动向后更新。



【参考代码】

```
#include <iostream>

using namespace std;

```

```cpp
typedef long long LL;

const int N = 65;

int n;
// LL f[N]; // f[i] 表示：有 i 个台阶的时候，一共有多少种方案

int main()
{
    cin >> n;

    // 版本一：用数组的形式
    // 初始化
    // f[0] = 1; f[1] = 1; f[2] = 2;

    // for(int i = 3; i <= n; i++)
    // {
    //     f[i] = f[i - 1] + f[i - 2] + f[i - 3];
    // }

    // cout << f[n] << endl;

    // 版本二：空间优化
    LL a = 1, b = 1, c = 2;

    for(int i = 3; i <= n; i++)
    {
        LL t = a + b + c;
        a = b;
        b = c;
        c = t;
    }

    // 小细节
    if(n == 1) cout << b << endl;
    else cout << c << endl;

    return 0;
}
```


1.2 数字三角形

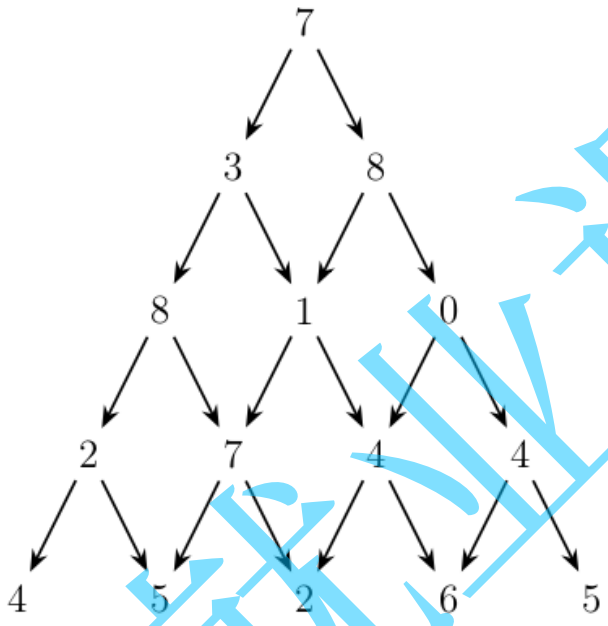
> **题目来源：** 洛谷
>
> **题目链接：** P1216 [USACO1.5] [IOI1994]数字三角形 Number Triangles

难度系数： ★

【题目描述】

观察下面的数字金字塔。

写一个程序来查找从最高点到底部任意处结束的路径，使路径经过数字的和最大。每一步可以走到左下方的点也可以到达右下方的点。



洛谷

在上面的样例中，从 $7 \to 3 \to 8 \to 7 \to 5$ 的路径产生了最大权值。

【输入描述】

第一个行一个正整数 r ,表示行的数目。

后面每行为这个数字金字塔特定行包含的整数。

对于 100% 的数据，$1 \leq r \leq 1000$ ，所有输入在 $[0, 100]$ 范围内。

【输出描述】

单独的一行,包含那个可能得到的最大的和。

【示例一】

输入：

```
5
7
3 8
8 1 0
```

> 2 7 4 4
>
> 4 5 2 6 5
>
> 输出：
>
> 30

【解法】

学习动态规划最经典的入门题。

解法：动态规划

1. 状态表示：

   $dp[i][j]$ 表示：走到 $[i,j]$ 位置的最大权值。

   那最终结果就是在 dp 表的第 n 行中，所有元素的最大值。

2. 状态转移方程：

   根据最后一步划分问题，走到 $[i,j]$ 位置的方式有两种：

   a. 从 $[i-1,j]$ 位置向下走一格，此时走到 $[i,j]$ 位置的最大权值就是 $dp[i-1][j]$；

   b. 从 $[i-1,j-1]$ 位置向右下走一格，此时走到 $[i,j]$ 位置的最大权值就是 $dp[i-1][j-1]$；

   综上所述，应该是两种情况的最大值再加上 $[i,j]$ 位置的权值：
   $dp[i][j] = max(dp[i-1][j], dp[i-1][j-1]) + a[i][j]$。

3. 初始化：

   因为 dp 表被 0 包围着，并不影响我们的最终结果，因此可以直接填表。

   思考，如果权值出现负数的话，需不需要初始化？

   - 此时可以全都初始化为 $-\infty$，负无穷大在取 max 之后，并不影响最终结果。

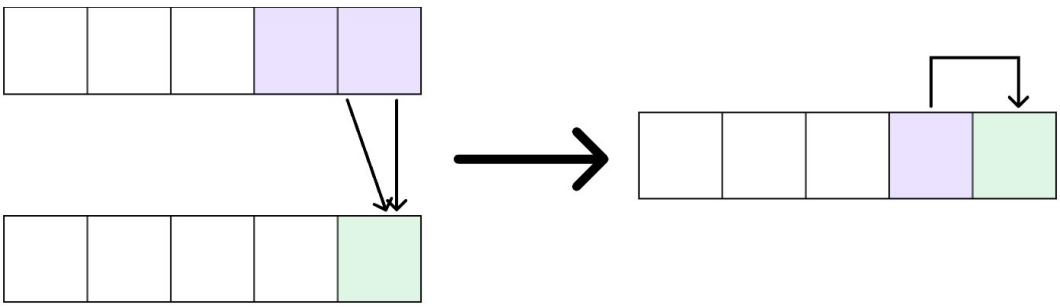
4. 填表顺序：

   从左往右填写每一行，每一行从左往右。

动态规划的空间优化：

我们发现，在填写第 i 行的值时，我们仅仅需要前一行的值，并不需要第 $i-2$ 以及之前行的值。

因此，我们可以只用一个一维数组来记录上一行的结果，然后在这个数组上更新当前行的值。



需要注意，当用因为我们当前这个位置的值需要左上角位置的值，因此滚动数组优化的时候，要改变第二维的遍历顺序。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 1010;

int n;
int a[N][N];
int f[N]; // f[i][j]: 从 [1, 1] 走到 [i, j] 时，所有方案下的最大权值

int main()
{
    cin >> n;

    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= i; j++)
            cin >> a[i][j];

    // 二维数组的版本
    // for(int i = 1; i <= n; i++)
    // {
    //     for(int j = 1; j <= i; j++)
    //     {
    //         f[i][j] = max(f[i - 1][j], f[i - 1][j - 1]) + a[i][j];
    //     }
    // }

    // 空间优化
    for(int i = 1; i <= n; i++)
    {
```

```
31            for(int j = i; j >= 1; j--) // 修改一下遍历顺序
32            {
33                f[j] = max(f[j], f[j - 1]) + a[i][j];
34            }
35        }
36
37        int ret = 0;
38        for(int j = 1; j <= n; j++)
39        {
40            ret = max(ret, f[j]);
41        }
42
43        cout << ret << endl;
44
45        return 0;
46    }
```

2. 线性 dp

线性dp 是动态规划问题中最基础、最常见的一类问题。它的特点是**状态转移只依赖于前一个或前几个状态**，状态之间的关系是线性的，通常可以用一维或者二维数组来存储状态。

我们在入门阶段解决的《下楼梯》以及《数字三角形》其实都是线性 dp，一个是一维的，另一个是二维的。

2.1 基础线性 dp

2.1.1 台阶问题

> **题目来源：** 洛谷
>
> **题目链接：** P1192 台阶问题
>
> **难度系数：** ★

【题目描述】

> 有 N 级台阶，你一开始在底部，每次可以向上迈 $1 \sim K$ 级台阶，问到达第 N 级台阶有多少种不同方式。

【输入描述】

> 两个正整数 N, K 。

> 对于 100% 的数据， $1 \leq N \leq 100000$，$1 \leq K \leq 100$。

【输出描述】

> 一个正整数 $ans(\mathrm{mod}100003)$，为到达第 N 级台阶的不同方式数。

【示例一】

> 输入：
>
> 5 2
>
> 输出：
>
> 8

【解法】

斐波那契数列模型

1. 状态表示：

   $dp[i]$ 表示：走到 i 位置的方案数。

   那么 $dp[n]$ 就是我们要的结果。

2. 状态转移方程：

   可以从 $i-k \leq j \leq i-1$ 区间内的台阶走到 i 位置，那么总方案数就是所有的 $dp[j]$ 累加在一起。

   注意 $i-k$ 不能小于 0。

3. 初始化：

   $dp[0]=1$，起始位置，为了让后续填表有意义。

4. 填表顺序：

   从左往右。

【参考代码】

```
#include <iostream>

using namespace std;
```

```cpp

const int N = 1e5 + 10, MOD = 1e5 + 3;

int n, k;
int f[N];

int main()
{
    cin >> n >> k;

    f[0] = 1;
    for(int i = 1; i <= n; i++)
    {
        for(int j = 1; j <= k && i - j >= 0; j++)
        {
            f[i] = (f[i] + f[i - j]) % MOD;
        }
    }

    cout << f[n] << endl;

    return 0;
}
```

2.1.2 最大子段和

> **题目来源：** 洛谷
>
> **题目链接：** P1115 最大子段和
>
> **难度系数：** ★★

【题目描述】

> 给出一个长度为 n 的序列 a ，选出其中连续且非空的一段使得这段和最大。

【输入描述】

> 第一行是一个整数，表示序列的长度 n 。
>
> 第二行有 n 个整数，第 i 个整数表示序列的第 i 个数字 a_i 。
>
> - 对于 40% 的数据，保证 $n \leq 1 \times 10^3$ 。
> - 对于 100% 的数据，保证 $1 \leq n \leq 2 \times 10^5, -10^4 \leq a_i \leq 10^4$ 。

【输出描述】

> 输出一行一个整数表示答案。

【示例一】

> 输入：
>
> 7
>
> 2 -4 3 -1 2 -4 3
>
> 输出：
>
> 4

【解法】

又又又遇见这道题了~

1. 状态表示：

   $dp[i]$ 表示：以 i 位置元素为结尾的「所有子数组」中和的最大值。

   那我们的最终结果应该是 dp 表里面的最大值。

2. 状态转移方程：

   $dp[i]$ 的所有可能可以分为以下两种：

   a. 子数组的长度为 1 ：此时 $dp[i] = a[i]$ ；

   b. 子数组的长度大于 1 ：此时 $dp[i]$ 应该等于以 $i-1$ 为结尾的「所有子数组」中和的最大值再加上 $a[i]$ ，也就是 $dp[i-1] + a[i]$ 。

   应该是两种情况下的最大值，因此可得转移方程： $dp[i] = max(a[i], dp[i-1] + a[i])$ 。

3. 初始化：

   把第一个格子初始化为 0 ，往后填数的时候就不会影响最终结果。

4. 填表顺序：

   根据「状态转移方程」易得，填表顺序为「从左往右」。

【参考代码】

```
#include <iostream>

using namespace std;
```

```cpp

const int N = 2e5 + 10;

int n;
int f[N];

int main()
{
    cin >> n;

    int ret = -1e9;
    for(int i = 1; i <= n; i++)
    {
        int x; cin >> x;
        f[i] = max(f[i - 1] + x, x);
        ret = max(ret, f[i]);
    }

    cout << ret << endl;

    return 0;
}
```

2.1.3 传球游戏

> **题目来源：** 洛谷
>
> **题目链接：** P1057 [NOIP2008 普及组] 传球游戏
>
> **难度系数：** ★★

【题目描述】

> 上体育课的时候，小蛮的老师经常带着同学们一起做游戏。这次，老师带着同学们一起做传球游戏。
>
> 游戏规则是这样的：n 个同学站成一个圆圈，其中的一个同学手里拿着一个球，当老师吹哨子时开始传球，每个同学可以把球传给自己左右的两个同学中的一个（左右任意），当老师再次吹哨子时，传球停止，此时，拿着球没有传出去的那个同学就是败者，要给大家表演一个节目。
>
> 聪明的小蛮提出一个有趣的问题：有多少种不同的传球方法可以使得从小蛮手里开始传的球，传了 m 次以后，又回到小蛮手里。两种传球方法被视作不同的方法，当且仅当这两种方法中，接到球的同学按接球顺序组成的序列是不同的。比如有三个同学 1 号、2 号、3 号，并假设小蛮为 1 号，球传了 3 次回到小蛮手里的方式有 $1\to2\to3\to1$ 和 $1\to3\to2\to1$，共 2 种。

【输入描述】

> 一行，有两个用空格隔开的整数 $n,m(3\le n\le 30,1\le m\le 30)$。

> 对于 100% 的数据，满足：$3 \le n \le 30, 1 \le m \le 30$。

【输出描述】

> 1 个整数，表示符合题意的方法数。

【示例一】

> 输入：
>
> 3 3
>
> 输出：
>
> 2

【解法】

1. 状态表示：

   $f[i][j]$ 表示传了 i 次，落在第 j 个人手里时的总方案数。

   那么 $f[m][1]$ 就是我们想要的结果。

2. 状态转移方程：

   因为是一个环形结构，第一个位置和最后一个位置可以特殊处理：

   a. 当 $2 \le j \le n-1$ 时，可以从 $j-1$ 或者 $j+1$ 传到该位置，那么总方案数就是 $f[i-1][j-1] + f[i-1][j+1]$；

   b. 当 $j = 1$ 时，可以从 n 或者 2 传到该位置，那么总方案数就是 $f[i-1][n] + f[i-1][2]$；

   c. 当 $j = n$ 时，可以从 $n-1$ 或者 1 传到该位置，那么总方案数就是 $f[i-1][1] + f[i-1][n-1]$。

3. 初始化：

   刚开始的状态设置为 1，让后续填表是正确的，$f[0][1] = 1$。

4. 填表顺序：

   一定要先循环次数，再循环位置。因为我们更新状态是从低次数更新高次数，也就是第一行更新第二行。因此填表顺序应该是从上往下每一行，行的顺序无所谓。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 50;

int n, m;
int f[N][N]; // f[i][j] 表示：传递了 i 次之后，落在了第 j 号同学手里的方案数

int main()
{
    cin >> n >> m;

    f[0][1] = 1;
    for(int i = 1; i <= m; i++)
    {
        // 第一个人
        f[i][1] = f[i - 1][2] + f[i - 1][n];

        // 中间的同学
        for(int j = 2; j < n; j++)
        {
            f[i][j] = f[i - 1][j - 1] + f[i - 1][j + 1];
        }

        // 最后一位同学
        f[i][n] = f[i - 1][1] + f[i - 1][n - 1];
    }

    cout << f[m][1] << endl;

    return 0;
}
```


2.1.4 乌龟棋

> **题目来源：** 洛谷
>
> **题目链接：** P1541 [NOIP2010 提高组] 乌龟棋
>
> **难度系数：** ★★★

【题目描述】

> 小明过生日的时候，爸爸送给他一副乌龟棋当作礼物。

乌龟棋的棋盘是一行 N 个格子，每个格子上一个分数（非负整数）。棋盘第 1 格是唯一的起点，第 N 格是终点，游戏要求玩家控制一个乌龟棋子从起点出发走到终点。

乌龟棋中 M 张爬行卡片，分成 4 种不同的类型（M 张卡片中不一定包含所有 4 种类型的卡片，见样例），每种类型的卡片上分别标有 $1,2,3,4$ 四个数字之一，表示使用这种卡片后，乌龟棋子将向前爬行相应的格子数。游戏中，玩家每次需要从所有的爬行卡片中选择一张之前没有使用过的爬行卡片，控制乌龟棋子前进相应的格子数，每张卡片只能使用一次。

游戏中，乌龟棋子自动获得起点格子的分数，并且在后续的爬行中每到达一个格子，就得到该格子相应的分数。玩家最终游戏得分就是乌龟棋子从起点到终点过程中到过的所有格子的分数总和。

很明显，用不同的爬行卡片使用顺序会使得最终游戏的得分不同，小明想要找到一种卡片使用顺序使得最终游戏得分最多。

现在，告诉你棋盘上每个格子的分数和所有的爬行卡片，你能告诉小明，他最多能得到多少分吗？

【输入描述】

每行中两个数之间用一个空格隔开。

第 1 行 2 个正整数 N,M，分别表示棋盘格子数和爬行卡片数。

第 2 行 N 个非负整数，$a_1,a_2,\ldots,a_N$，其中 a_i 表示棋盘第 i 个格子上的分数。

第 3 行 M 个整数，$b_1,b_2,\ldots,b_M$，表示 M 张爬行卡片上的数字。

输入数据保证到达终点时刚好用光 M 张爬行卡片。

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

【输出描述】

一个整数，表示小明最多能得到的分数。

【示例一】

输入：

```
9 5
6 10 14 2 8 8 18 5 17
1 3 1 2 1
```

输出：

```
73
```

【解法】

1. 状态表示：

$f[i][a][b][c][d]$ 表示：走到 i 位置时，编号为 1234 的卡片分别用了 $abcd$ 张，此时的最大分数。

我们发现，当 1234 用的卡片数确定之后，走到的位置 i 可以计算出来，其中 $i = 1 + a + 2b + 3c + 4d$。

因此状态表示可以优化掉一维，变成 $f[a][b][c][d]$，表示：编号为 1234 的卡片分别用了 $abcd$ 张，此时的最大分数。

2. 状态转移方程：

设根据最后一次用的卡片种类，分情况讨论：

a. 如果 $a > 0$，并且最后一张用 1 卡片，最大分数为：$f[a-1][b][c][d] + nums[i]$；

b. 如果 $b > 0$，并且最后一张用 2 卡片，最大分数为：$f[a][b-1][c][d] + nums[i]$；

c. 如果 $c > 0$，并且最后一张用 3 卡片，最大分数为：$f[a][b][c-1][d] + nums[i]$；

d. 如果 $d > 0$，并且最后一张用 4 卡片，最大分数为：$f[a][b][c][d-1] + nums[i]$；

综上所述，取四种情况里面的最大值即可。

3. 初始化：

一张卡片也不用的情况下，可以获得第一个格子的分数，$f[0][0][0][0] = nums[1]$。

4. 填表顺序：

从小到大枚举每种卡片使用的张数即可。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 360, M = 50;

int n, m;
int x[N], cnt[5];
int f[M][M][M][M];

int main()
{
    cin >> n >> m;
```

```
14
15      for(int i = 1; i <= n; i++) cin >> x[i];
16      for(int i = 1; i <= m; i++)
17      {
18          int t; cin >> t;
19          cnt[t]++;
20      }
21
22      // 初始化
23      f[0][0][0][0] = x[1];
24      for(int a = 0; a <= cnt[1]; a++)
25          for(int b = 0; b <= cnt[2]; b++)
26              for(int c = 0; c <= cnt[3]; c++)
27                  for(int d = 0; d <= cnt[4]; d++)
28                  {
29                      int i = 1 + a + 2 * b + 3 * c + 4 * d;
30                      int& t = f[a][b][c][d];
31                      if(a) t = max(t, f[a - 1][b][c][d] + x[i]);
32                      if(b) t = max(t, f[a][b - 1][c][d] + x[i]);
33                      if(c) t = max(t, f[a][b][c - 1][d] + x[i]);
34                      if(d) t = max(t, f[a][b][c][d - 1] + x[i]);
35                  }
36
37      cout << f[cnt[1]][cnt[2]][cnt[3]][cnt[4]] << endl;
38
39      return 0;
40  }
```

2.2 路径类 dp

路径类 dp 是线性 dp 的一种，它是在一个 n × m 的矩阵中设置一个行走规则，研究从起点走到终点的方案数、最小路径和或者最大路径和等等的问题。

入门阶段的《数字三角形》其实就是路径类 dp。

2.2.1 矩阵的最小路径和

> **题目来源：** 牛客网
>
> **题目链接：** DP11 矩阵的最小路径和
>
> **难度系数：** ★

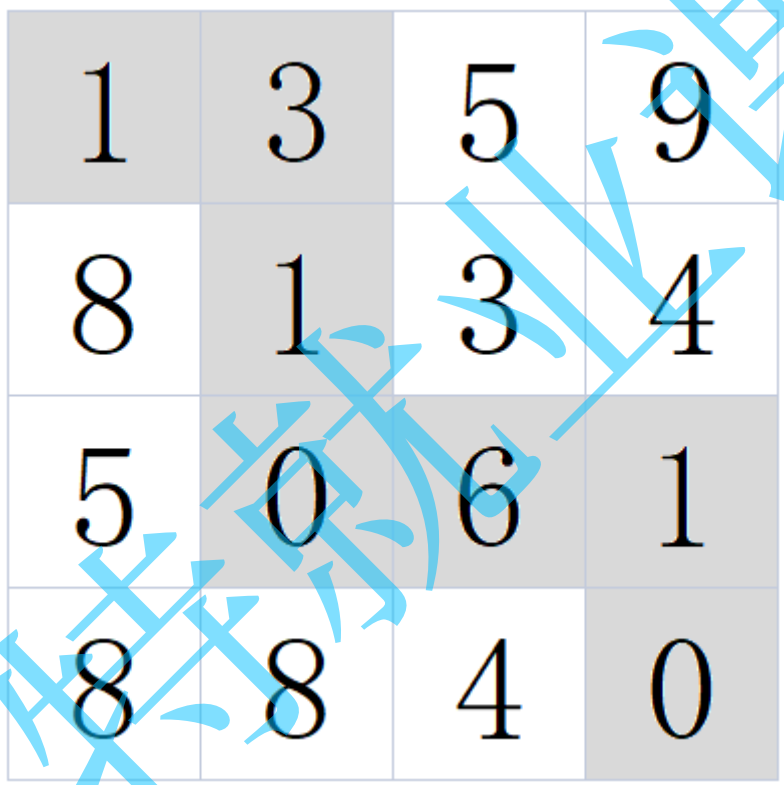
【题目描述】

给定一个 $n \times m$ 的矩阵 a ，从左上角开始每次只能向右或者向下走，最后到达右下角的位置，路径上所有的数字累加起来就是路径和，输出所有的路径中最小的路径和。

数据范围: $1 \leq n, m \leq 500$ ，矩阵中任意值都满足 $0 \leq a_{i,j} \leq 100$

要求：时间复杂度 $O(nm)$

例如：当输入 $[[1,3,5,9],[8,1,3,4],[5,0,6,1],[8,8,4,0]]$ 时，对应的返回值为 12 ，所选择的最小累加和路径如下图所示：



【输入描述】

第一行输入两个正整数 n 和 m 表示矩阵 a 的长宽

后续输入 n 行每行有 m 个数表示矩阵的每个元素

【输出描述】

输出从左上角到右下角的最小路径和

【示例一】

输入：

```
4 4
1 3 5 9
8 1 3 4
5 0 6 1
8 8 4 0
```

8 1 3 4

5 0 6 1

8 8 4 0

输出：

12

【示例二】

输入：

2 3

1 2 3

1 2 3

输出：

7

【解法】

1. 状态表示：

   $dp[i][j]$ 表示：到达 $[i,j]$ 位置处，最小路径和是多少。

   那我们的最终结果就是 $dp[n][m]$ 。

2. 状态转移：

   到达 $[i,j]$ 位置之前的一小步，有两种情况：

   i. 从 $[i-1,j]$ 向下走一步，转移到 $[i,j]$ 位置；

   ii. 从 $[i,j-1]$ 向右走一步，转移到 $[i,j]$ 位置。

   由于到 $[i,j]$ 位置两种情况，并且我们要找的是最小路径，因此只需要这两种情况下的最小值，再加上 $[i,j]$ 位置上本身的值即可： $dp[i][j]=min(dp[i-1][j],dp[i][j-1])+a[i][j]$ 。

3. 初始化：

   第一行和第一列是要初始化的，因为会越界访问。

   但是如果把整张表初始化为无穷大，然后把 $dp[0][1]$ 和 $dp[1][0]$ 的值设为 0 ，后续填表就是正确的。

4. 填表顺序：

根据「状态转移方程」的推导来看，填表的顺序就是「从上往下」填每一行，每一行「从左往后」。

【参考代码】

```cpp
#include <iostream>
#include <cstring>

using namespace std;

const int N = 510;

int n, m;
int f[N][N];

int main()
{
    cin >> n >> m;

    // 初始化
    memset(f, 0x3f, sizeof f);
    f[0][1] = 0;

    for(int i = 1; i <= n; i++)
    {
        for(int j = 1; j <= m; j++)
        {
            int x; cin >> x;
            f[i][j] = min(f[i - 1][j], f[i][j - 1]) + x;
        }
    }

    cout << f[n][m] << endl;

    return 0;
}
```


2.2.2 「木」迷雾森林

> **题目来源：** 牛客网
>
> **题目链接：** 「木」迷雾森林

> **难度系数：** ★

【题目描述】

> 赛时提示：保证出发点和终点都是空地
>
> 帕秋莉掌握了一种木属性魔法
>
> 这种魔法可以生成一片森林（类似于迷阵），但一次实验时，帕秋莉不小心将自己困入了森林
>
> 帕秋莉处于地图的左下角，出口在地图右上角，她只能够向上或者向右行走
>
> 现在给你森林的地图，保证可以到达出口，请问有多少种不同的方案
>
> 答案对 2333 取模

【输入描述】

> 第一行两个整数 m, n 表示森林是 m 行 n 列
> 接下来 m 行，每行 n 个数，描述了地图
> 0 - 空地
> 1 - 树（无法通过）
>
> 对于 100% 的数据， $n, m \leq 3,000$

【输出描述】

> 一个整数表示答案

【示例一】

> 输入：
>
> ```
> 3 3
> 0 1 0
> 0 0 0
> 0 0 0
> ```
>
> 输出：
>
> ```
> 3
> ```


【解法】

1. 状态表示：

   $f[i][j]$ 表示：到达 $[i, j]$ 位置时，有多少种方案。

   那么 $f[1][m]$ 就是我们要的结果。

2. 状态转移方程：

a. 如果 $[i,j]$ 位置是空地，到达 $[i,j]$ 位置有两种方式：

- 从 $[i+1,j]$ 向上走一步，此时的方案数为 $f[i+1][j]$；
- 从 $[i,j-1]$ 向右走一步，此时的方案数为 $f[i][j-1]$。

两者总和就是到达 $[i,j]$ 位置的总方案数。

b. 如果 $[i,j]$ 位置是树，无法走到，$f[i][j]=0$。

3. 初始化：

可以在原始矩阵的规模上多加上一行和一列，把 $f[n+1][1]$ 或者 $f[n][0]$ 初始化为 1，这样后续填表就会有意义。

4. 填表顺序：

从下往上每一行，每一行从左往右。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 3010, MOD = 2333;

int n, m;
int a[N][N];
int f[N][N];

int main()
{
    scanf("%d%d", &n, &m);

    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++)
            scanf("%d", &a[i][j]);

    // 初始化
    f[n][0] = 1;
    for(int i = n; i >= 1; i--)
        for(int j = 1; j <= m; j++)
            if(a[i][j] == 0)
                f[i][j] = (f[i][j - 1] + f[i + 1][j]) % MOD;
```

```
25
26      cout << f[1][m] << endl;
27
28      return 0;
29  }
```

2.2.3 过河卒

题目来源： 洛谷

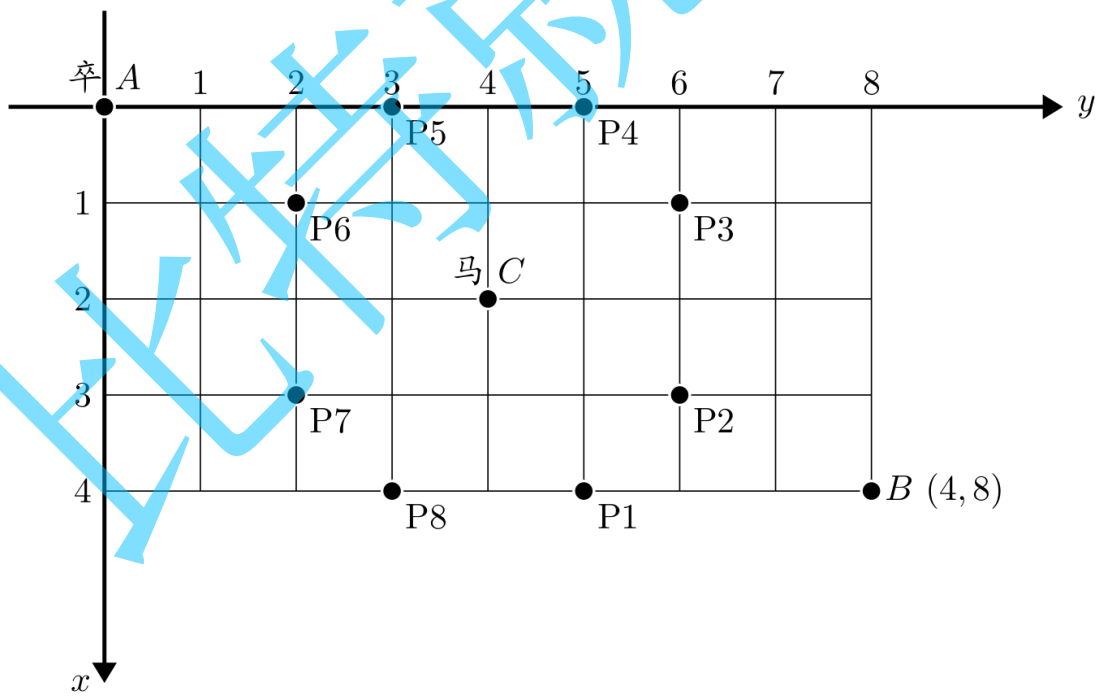
题目链接： P1002 [NOIP2002 普及组] 过河卒

难度系数： ★

【题目描述】

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”。

棋盘用坐标表示，A 点 $(0,0)$、B 点 (n,m)，同样马的位置坐标是需要给出的。



现在要求你计算出卒从 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。

【输入描述】

一行四个正整数，分别表示 B 点坐标和马的坐标。

> 对于 100% 的数据， $1 \le n, m \le 20$ ， $0 \le$ 马的坐标 ≤ 20 。

【输出描述】

> 一个整数，表示所有的路径条数。

【示例一】

> 输入：
>
> 6 6 3 3
>
> 输出：
>
> 6

【解法】

1. 状态表示：

   $f[i][j]$ 表示：到达 $[i, j]$ 位置的方案数。

   那么 $f[n][m]$ 就是我们要的结果。

2. 状态转移方程：

   a. 如果 $[i, j]$ 位置能走到，到达 $[i, j]$ 位置之前的一小步，有两种情况：

   - 从 $[i-1, j]$ 向下走一步，走到 $[i, j]$，此时的方案数为 $f[i-1][j]$；
   - 从 $[i, j-1]$ 向右走一步，走到 $[i, j]$，此时的方案数为 $f[i][j-1]$；

   那么总方案数 $f[i][j] = f[i-1][j] + f[i][j-1]$ 。

   b. 如果 $[i, j]$ 位置走不到，$f[i][j] = 0$ 。

3. 初始化：

   我们可以给原始的矩阵多加一行多加一列， n, m, x, y 全部 $+1$ ，这样填任何一个位置都不会越界。

   然后初始化 $f[1][0] = 1$ 或者 $f[0][1] = 1$ ，保证后续填表正确即可。

4. 填表顺序：

   从上往下每一行，每一行从左往右。

【参考代码】

```cpp
#include <iostream>

using namespace std;

typedef long long LL;

const int N = 25;

int n, m, a, b;
LL f[N][N];

// 判断是否是被马所拦截的点
bool check(int i, int j)
{
    return (i == a && j == b) || (i != a && j != b && abs(i - a) + abs(j - b) == 3);
}

int main()
{
    cin >> n >> m >> a >> b;
    n++; m++; a++; b++;

    // 初始化
    f[0][1] = 1;
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++)
        {
            if(check(i, j)) continue;

            f[i][j] = f[i - 1][j] + f[i][j - 1];
        }

    cout << f[n][m] << endl;

    return 0;
}
```


2.2.4 方格取数

> **题目来源：** 洛谷
>
> **题目链接：** P1004 [NOIP2000 提高组] 方格取数
>
> **难度系数：** ★★★

【题目描述】

设有 $N \times N$ 的方格图 $(N \leq 9)$，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 。如下图所示（见样例）：

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

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。

【输入描述】

输入的第一行为一个整数 N（表示 $N \times N$ 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。

数据范围：$1 \leq N \leq 9$。

【输出描述】

只需输出一个整数，表示 2 条路径上取得的最大的和。

【示例一】

输入：

```
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：
>
> 67

【解法】

贪心 + 两次 dp 是错误的，因为两次最优不等于全局最优，可以举出反例。正解应该是同时去走两条路，两者相互影响，所以放在一起考虑。

1. 状态表示：

   需要知道当前这两条路径走到什么位置，因此需要四维 $f[i_1][j_1][i_2][j_2]$ 来表示第一条路走到 $[i_1, j_1]$ 第二条路走到 $[i_2][j_2]$。

   但是我们发现，因为两者是同时出发的，所以横纵坐标之和是一个定值。也就是说，只要知道了横纵坐标之和，以及两者的横坐标，就可以计算出纵坐标，状态表示就可以优化掉一维。

   优化后的状态表示：$f[st][i_1][i_2]$ 表示：第一条路在 $[i_1, st - i_1]$，第二条路在 $[i_2, st - i_2]$ 时，两者的路径最大和。那我们的最终结果就是 $f[n \times 2][n][n]$。

2. 状态转移方程：

   第一条路可以从上 $[i_1 - 1, st - i_1]$ 或者左 $[i_1, st - i_1 - 1]$ 走到 $[i_1, st - i_1]$ 位置；第二条路可以从上 $[i_2 - 1, st - i_2]$ 或者左 $[i_2, st - i_2 - 1]$ 走到 $[i_2, st - i_2]$ 位置。排列组合一下一共 4 中情况，分别是：

   - 上 + 上，此时的最大和为：$f[st - 1][i_1 - 1][i_2 - 1]$；
   - 上 + 左，此时的最大和为：$f[st - 1][i_1 - 1][i_2]$；
   - 左 + 上，此时的最大和为：$f[st - 1][i_1][i_2 - 1]$；
   - 左 + 左，此时的最大和为：$f[st - 1][i_1][i_2]$；

   取上面四种情况的最大值，然后再加上 $a[i_1][j_1]$ 和 $a[i_2][j_2]$。但是要注意，如果两个路径当前在同一位置时，只用加上一个 $a[i_1][j_1]$ 即可。

3. 初始化：

   算的是路径和，0 不会影响最终结果，直接填表。

4. 填表顺序：

   先从小到大循环横纵坐标之和，然后依次从小到大循环两者的横坐标。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 15;

int n;
int a[N][N];
int f[N * 2][N][N];

int main()
{
    cin >> n;
    int x, y, w;
    while(cin >> x >> y >> w, x)
    {
        a[x][y] = w;
    }

    for(int s = 2; s <= n + n; s++)
    {
        for(int i1 = 1; i1 <= n; i1++)
        {
            for(int i2 = 1; i2 <= n; i2++)
            {
                int j1 = s - i1, j2 = s - i2;
                if(j1 <= 0 || j1 > n || j2 <= 0 || j2 > n) continue;

                int t = f[s - 1][i1][i2];
                t = max(t, f[s - 1][i1][i2 - 1]);
                t = max(t, f[s - 1][i1 - 1][i2]);
                t = max(t, f[s - 1][i1 - 1][i2 - 1]);

```

```
                if(i1 == i2)
                {
                    f[s][i1][i2] = t + a[i1][j1];
                }
                else
                {
                    f[s][i1][i2] = t + a[i1][j1] + a[i2][j2];
                }
            }
        }
    }

    cout << f[n + n][n][n] << endl;

    return 0;
}
```

2.3 经典线性 dp

经典线性 dp 问题有两个：最长上升子序列（简称：LIS）以及最长公共子序列（简称：LCS），这两道题目的很多方面都是可以作为经验，运用到别的题目中。比如：解题思路，定义状态表示的方式，推到状态转移方程的技巧等等。

因此，这两道经典问题是一定需要掌握的。

2.3.1 最长上升子序列（一）

> **题目来源：** 洛谷
>
> **题目链接：** B3637 最长上升子序列
>
> **难度系数：** ★

【题目描述】

> 这是一个简单的动规板子题。
>
> 给出一个由 $n(n \leq 5000)$ 个不超过 10^6 的正整数组成的序列。请输出这个序列的最长上升子序列的长度。
>
> 最长上升子序列是指，从原序列中按顺序取出一些数字排在一起，这些数字是逐渐增大的。

【输入描述】

> 第一行，一个整数 n，表示序列长度。
>
> 第二行有 n 个整数，表示这个序列。

【输出描述】

> 一个整数表示答案。

【示例一】

> 输入：
>
> 6
>
> 1 2 4 1 3 4
>
> 输出：
>
> 4

【解法】

1. 状态表示

   $dp[i]$ 表示：以 i 位置元素为结尾的「所有子序列」中，最长递增子序列的长度。

   最终结果就是整张 dp 表里面的最大值。

2. 状态转移方程：

   对于 $dp[i]$，我们可以根据「子序列的构成方式」，进行分类讨论：

   - 子序列长度为 1：只能自己玩了，此时 $dp[i] = 1$；
   - 子序列长度大于 1：$a[i]$ 可以跟在前面某些数后面形成子序列。设前面的某一个数的下标为 j，其中 $1 \le j < i - 1$。只要 $a[j] < a[i]$，i 位置元素跟在 j 元素后面就可以形成递增序列，长度为 $dp[j] + 1$。

     因此，我们仅需找到满足要求的最大的 $dp[j] + 1$ 即可。

   综上，$dp[i] = max(dp[j] + 1, dp[i])$，其中 $1 \le j < i \ \&\& \ nums[j] < nums[i]$。

3. 初始化：

   不用单独初始化，每次填表的时候，先把这个位置的数改成 1 即可。

4. 填表顺序：

   显而易见，填表顺序「从左往右」。

【参考代码】

```
1  #include <iostream>
2
```

```cpp
using namespace std;

const int N = 5010;

int n;
int a[N];
int f[N];

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];

    int ret = 0;
    for(int i = 1; i <= n; i++)
    {
        f[i] = 1; // 长度为 1 的子序列
        for(int j = 1; j < i; j++)
        {
            if(a[j] < a[i])
            {
                f[i] = max(f[i], f[j] + 1);
            }
        }
        ret = max(ret, f[i]);
    }

    cout << ret << endl;

    return 0;
}
```


2.3.2 最长上升子序列（二）

> **题目来源：** 牛客网
>
> **题目链接：** 【模板】最长上升子序列
>
> **难度系数：** ★★

【题目描述】

> 给你一个长度为 n 的数组，求最长的严格上升子序列的长度。

【输入描述】

> 第一行一个整数 n ，表示数组长度。

> 第二行 n 个整数，表示数组中的元素。
>
> $1 \leq n \leq 100000$

【输出描述】

> 输出一行，表示答案。

【示例一】

> 输入：
>
> 5
>
> 1 2 2 2 3
>
> 输出：
>
> 3

【解法】

利用贪心 + 二分优化动态规划：

- 我们在考虑最长递增子序列的长度的时候，其实并不关心这个序列长什么样子，我们只是关心最后一个元素是谁。这样新来一个元素之后，我们就可以判断是否可以拼接到它的后面。
- 因此，我们可以创建一个数组，统计长度为 `x` 的递增子序列中，最后一个元素是谁。为了尽可能的让这个序列更长，我们仅需统计长度为 `x` 的所有递增序列中最后一个元素的「最小值」。
- 统计的过程中发现，数组中的数呈现「递增」趋势，因此可以使用「二分」来查找插入位置。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 1e5 + 10;

int n;
int a[N];
int f[N], len; // 注意与上一个动态规划数组的含义是不一样的

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];

```

```
    for(int i = 1; i <= n; i++)
    {
        // 处理边界情况
        if(len == 0 || a[i] > f[len]) f[++len] = a[i];
        else
        {
            // 二分插入位置
            int l = 1, r = len;
            while(l < r)
            {
                int mid = (l + r) / 2;
                if(f[mid] >= a[i]) r = mid;
                else l = mid + 1;
            }
            f[l] = a[i];
        }
    }

    cout << len << endl;

    return 0;
}
```

2.3.3 合唱队形

> **题目来源：** 洛谷
>
> **题目链接：** P1091 [NOIP2004 提高组] 合唱队形
>
> **难度系数：** ★★

【题目描述】

> n 位同学站成一排，音乐老师邀请其中的 $n-k$ 位同学出列，使得剩下的 k 位同学排成合唱队形。
>
> 合唱队形是指这样的一种队形：设 k 位同学从左到右依次编号为 $1,2,\ldots,k$，他们的身高分别为 $t_1,t_2,\ldots,t_k$，则他们的身高满足 $t_1<\cdots<t_i>t_{i+1}>ti+1>\ldots>t_k(1\le i\le k)$。
>
> 你的任务是，已知所有 n 位同学的身高，计算最少需要几位同学出列，可以使得剩下的同学排成合唱队形。

【输入描述】

> 共二行。
>
> 第一行是一个整数 n（$2\le n\le 100$），表示同学的总数。

> 第二行有 n 个整数，用空格分隔，第 i 个整数 $t_i(130 \leq t_i \leq 230)$ 是第 i 位同学的身高（厘米）。
>
> 对于全部的数据，保证有 $n \leq 100$。

【输出描述】

> 一个整数，最少需要几位同学出列。

【示例一】

> 输入：
>
> 8
>
> 186 186 150 200 160 130 197 220
>
> 输出：
>
> 4

【解法】

对于每一个位置 i ，计算：

- 从左往右看：以 i 为结尾的最长上升子序列 $f[i]$；
- 从右往左看：以 i 为结尾的最长上升子序列 $g[i]$；

最终结果就是所有 $f[i]+g[i]-1$ 里面的最大值。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 110;

int n;
int a[N];
int f[N], g[N];

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];

    // 从左往右
    for(int i = 1; i <= n; i++)
```

```
    {
        f[i] = 1;
        for(int j = 1; j < i; j++)
        {
            if(a[j] < a[i])
            {
                f[i] = max(f[i], f[j] + 1);
            }
        }
    }

    // 从右往左
    for(int i = n; i >= 1; i--)
    {
        g[i] = 1;
        for(int j = n; j > i; j--)
        {
            if(a[j] < a[i])
            {
                g[i] = max(g[i], g[j] + 1);
            }
        }
    }

    int ret = 0;
    for(int i = 1; i <= n; i++)
    {
        ret = max(ret, f[i] + g[i] - 1);
    }

    cout << n - ret << endl;

    return 0;
}
```


2.3.4 牛可乐和最长公共子序列

> **题目来源：** 牛客网
>
> **题目链接：** 牛可乐和最长公共子序列
>
> **难度系数：** ★★

【题目描述】

> 牛可乐得到了两个字符串 s 和 t ,牛可乐想请聪明的你帮他计算出来，两个字符串的最长公共子序列长度是多少。
>
> 最长公共子序列的定义是，子序列中的每个字符都能在两个原串中找到，而且每个字符的先后顺序和原串中的先后顺序一致。

【输入描述】

> 输入包含多组数据，请读至文件末尾。
>
> 每行包含两个字符串 s,t ，两个字符串用一个空格字符间隔，单个字符串长度不超过 5000 。
>
> 数据保证所有数据的字符串 s 长度之和与字符串 t 长度之和均不超过 5000 。

【输出描述】

> 对于每组数据，输出一个整数，代表最长公共子序列的长度。

【示例一】

> 输入：
>
> abccde bcee
>
> 输出：
>
> 3

【解法】

1. 状态表示：

   $dp[i][j]$ 表示： $s1$ 的 $[1,i]$ 区间以及 $s2$ 的 $[1,j]$ 区间内的所有的子序列中，最长公共子序列的长度。

   那么 $dp[n][m]$ 就是我们要的结果。

2. 状态转移方程：

   对于 $dp[i][j]$ ，我们可以根据 $s1[i]$ 与 $s2[j]$ 的字符分情况讨论：

   a. 两个字符相同 $s1[i]=s2[j]$ ：那么最长公共子序列就在 $s1$ 的 $[1,i-1]$ 以及 $s2$ 的 $[1,j-1]$ 区间上找到一个最长的，然后再加上 $s1[i]$ 即可。因此 $dp[i][j]=dp[i-1][j-1]+1$ ；

   b. 两个字符不同 $s1[i]\neq s2[j]$ ：那么最长公共子序列一定不会同时以 $s1[i]$ 和 $s2[j]$ 结尾。那么我们找最长公共子序列时，有下面三种策略：

   - 去 $s1$ 的 $[1,i-1]$ 以及 $s2$ 的 $[1,j]$ 区间内找：此时最大长度为 $dp[i-1][j]$ ；
   - 去 $s1$ 的 $[1,i]$ 以及 $s2$ 的 $[1,j-1]$ 区间内找：此时最大长度为 $dp[i][j-1]$ ；
   - 去 $s1$ 的 $[1,i-1]$ 以及 $s2$ 的 $[1,j-1]$ 区间内找：此时最大长度为 $dp[i-1][j-1]$ 。

我们要三者的最大值即可。但是我们仔细观察会发现，第三种包含在第一种和第二种情况里面，但是我们求的是最大值，并不影响最终结果。因此只需求前两种情况下的最大值即可。

综上，状态转移方程为：

$if(s1[i] = s2[j])\ dp[i][j] = dp[i-1][j-1] + 1$;

$if(s1[i] \neq s2[j])\ dp[i][j] = max(dp[i-1][j], dp[i][j-1])$。

3. 初始化：

   直接填表即可。

4. 填表顺序：

   根据「状态转移方程」得：从上往下填写每一行，每一行从左往右。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 5010;

string s, t;
int f[N][N];

int main()
{
    while(cin >> s >> t)
    {
        int n = s.size(), m = t.size();
        // s = " " + s, t = " " + t;

        for(int i = 1; i <= n; i++)
        {
            for(int j = 1; j <= m; j++)
            {
                if(s[i - 1] == t[j - 1]) f[i][j] = f[i - 1][j - 1] + 1;
                else f[i][j] = max(f[i - 1][j], f[i][j - 1]);
            }
        }

        cout << f[n][m] << endl;
```

```
27        }
28
29        return 0;
30    }
```

2.3.5 编辑距离

> **题目来源：** 洛谷
>
> **题目链接：** P2758 编辑距离
>
> **难度系数：** ★★★

【题目描述】

> 设 A 和 B 是两个字符串。我们要用最少的字符操作次数，将字符串 A 转换为字符串 B。这里所说的字符操作共有三种：
>
> 1. 删除一个字符；
> 2. 插入一个字符；
> 3. 将一个字符改为另一个字符。
>
> A, B 均只包含小写字母。

【输入描述】

> 第一行为字符串 A；第二行为字符串 B；字符串 A, B 的长度均小于 2000。

【输出描述】

> 只有一个正整数，为最少字符操作次数。

【示例一】

> 输入：
>
> sfdqxbw
>
> gfdgw
>
> 输出：
>
> 4

【解法】

两个字符串之间的 dp 问题，与最长公共子序列的分析方式类似。

1. 状态表示：

   $dp[i][j]$ 表示：字符串 A 中 $[1, i]$ 区间与字符串 B 中 $[1, j]$ 区间内的编辑距离。

   那么 $dp[n][m]$ 就是我们要的结果。

2. 状态转移方程：

   对于 $dp[i][j]$，我们可以根据 $A[i]$ 与 $B[j]$ 的字符分情况讨论：

   a. 两个字符相同 $A[i] = B[j]$：那么 $dp[i][j]$ 就是 A 的 $[1, i-1]$ 以及 B 的 $[1, j-1]$ 区间内编辑距离，因此 $dp[i][j] = dp[i-1][j-1]$；

   b. 两个字符不同 $A[i] \neq B[j]$：那么对于 A 字符串，我们可以进行下面三种操作：

   - 删掉 $A[i]$：此时 $dp[i][j]$ 就是 A 的 $[1, i-1]$ 以及 B 的 $[1, j]$ 区间内的编辑距离，因此 $dp[i][j] = dp[i-1][j] + 1$；
   - 插入一个字符：在字符串 A 的后面插入一个 $B[j]$，此时的 $dp[i][j]$ 就是 A 的 $[1, i]$ 以及 B 的 $[1, j-1]$ 区间内的编辑距离，因此 $dp[i][j] = dp[i][j-1] + 1$；
   - 将 $A[i]$ 替换成 $B[j]$：此时的 $dp[i][j]$ 就是 A 的 $[1, i-1]$ 以及 B 的 $[1, j-1]$ 区间内的编辑距离，因此 $dp[i][j] = dp[i-1][j-1] + 1$。

   我们要三者的最小值即可。

3. 初始化：

   需要注意，当 i, j 等于 0 的时候，这些状态也是有意义的。我们可以全部删除，或者全部插入让两者相同。

   因此需要初始化第一行 $dp[0][j] = j\ (1 \le j \le m)$，第一列 $dp[i][0] = i\ (1 \le i \le n)$。

4. 填表顺序：

   初始化完之后，从 $[1, 1]$ 位置开始从上往下每一行，每一行从左往右填表即可。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 2010;

string a, b;
int n, m;
int f[N][N];

int main()
{
```

```
13      cin >> a >> b;
14      n = a.size(); m = b.size();
15      a = " " + a; b = " " + b;
16
17      // 初始化
18      for(int i = 1; i <= n; i++) f[i][0] = i;
19      for(int j = 1; j <= m; j++) f[0][j] = j;
20
21      for(int i = 1; i <= n; i++)
22          for(int j = 1; j <= m; j++)
23          {
24              if(a[i] == b[j]) f[i][j] = f[i - 1][j - 1];
25              else f[i][j] = min(min(f[i - 1][j], f[i - 1][j - 1]), f[i][j -
    1]) + 1;
26          }
27
28      cout << f[n][m] << endl;
29
30      return 0;
31  }
```

3. 背包问题

背包问题是动态规划中最经典的问题，很多题目或多或少都有背包问题的影子。它的基本形式是：给定一组物品，每个物品有体积和价值，在不超过背包容量的情况下，选择物品使得总价值最大。

背包问题有多种变体，主要包括：

1. **01 背包问题**：每种物品只能选或不选（选 0 次或 1 次）。
2. **完全背包问题**：每种物品可以选择无限次。
3. **多重背包问题**：每种物品有数量限制。
4. **分组背包问题**：物品被分为若干组，每组只能选一个物品。
5. **混合背包**：以上四种背包问题混在一起。
6. **多维费用的背包问题**：限定条件不止有体积，还会有其他因素（比如重量）。

除了经典的总价值最大问题，还会有：

1. **方案总数。**
2. **最优方案。**
3. **方案可行性。**

4. **输出具体方案。**

因此，背包问题种类非常繁多，题型非常丰富。但是，尽管背包有很多变形，都是从 01 背包问题演化过来的。所以，一定要把 01 背包问题学好。

3.1 01 背包

3.1.1 01 背包

> **题目来源：** 牛客网
>
> **题目链接：** 【模板】01背包
>
> **难度系数：** ★★

【题目描述】

> 你有一个背包，最多能容纳的体积是 V 。
>
> 现在有 n 个物品，第 i 个物品的体积为 v_i ,价值为 w_i 。
>
> 1. 求这个背包至多能装多大价值的物品?
> 2. 若背包恰好装满，求至多能装多大价值的物品?

【输入描述】

> 第一行两个整数 n 和 V ，表示物品个数和背包体积。
>
> 接下来 n 行，每行两个数 v_i 和 w_i ，表示第 i 个物品的体积和价值。
>
> $1 \leq n, V, v_i, w_i \leq 1000$

【输出描述】

> 输出有两行，第一行输出第一问的答案，第二行输出第二问的答案，如果无解请输出 0 。

【示例一】

> 输入：
>
> ```
> 3 5
> 2 10
> 4 5
> 1 4
> ```
>
> 输出：
>
> ```
> 14
> ```

> 9

【示例二】

> 输入：
>
> 3 8
>
> 12 6
>
> 11 8
>
> 6 8
>
> 输出：
>
> 8
>
> 0

【解法】

我们先解决第一问：

1. 状态表示：

   $dp[i][j]$ 表示：从前 i 个物品中挑选，总体积「不超过」 j ，所有的选法中，能挑选出来的最大价值。

   那么 $dp[n][v]$ 就是我们要的结果。

2. 状态转移方程：

   线性 dp 状态转移方程分析方式，一般都是根据「最后一步」的状况，来分情况讨论：

   a. 不选第 i 个物品：相当于就是去前 $i-1$ 个物品中挑选，并且总体积不超过 j 。此时 $dp[i][j]=dp[i-1][j]$ ；

   b. 选择第 i 个物品：那么我就只能去前 $i-1$ 个物品中，挑选总体积不超过 $j-v[i]$ 的物品。此时 $dp[i][j]=dp[i-1][j-v[i]]+w[i]$ 。但是这种状态不一定存在，因此需要特判一下。

   综上，状态转移方程为： $dp[i][j]=max(dp[i-1][j],dp[i-1][j-v[i]]+w[i])$ 。

3. 初始化：

   直接填表，第一行的 0 不影响结果。

4. 填表顺序：

   根据「状态转移方程」，我们仅需「从上往下」填表即可。

接下来解决第二问：

第二问仅需修改一下初始化以及最终结果即可。

1. 初始化：

   因为有可能凑不齐 j 体积的物品，因此我们把不合法的状态设置为负无穷。这样在取最大值的时候，就不会考虑到这个位置的值。负无穷一般设置为 $-0x3f3f3f3f$ 即可。

   然后把 $dp[0][0] = 0$ 修改成 0 ，因为这是一个合法的状态，最大价值是 0 ，也让后续填表是正确的。

2. 返回值：

   在最后拿结果的时候，也要判断一下最后一个位置是不是小于 0 ，因为有可能凑不齐。

   不能判断是否等于 $-0x3f3f3f3f$ ，因为这个位置的值会被更新，只不过之前的值太小，导致更新后还是小于 0 的。

【参考代码】（后面有空间优化的版本）

```cpp
// 原始版本
#include <iostream>
#include <cstring>

using namespace std;

const int N = 1010;

int n, m;
int v[N], w[N];
int f[N][N];

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> v[i] >> w[i];

    // 第一问
    for(int i = 1; i <= n; i++)
    {
        for(int j = 0; j <= m; j++)
        {
            f[i][j] = f[i - 1][j];
            if(j >= v[i])
            {
```

```
                f[i][j] = max(f[i][j], f[i - 1][j - v[i]] + w[i]);
            }
        }
    }

    cout << f[n][m] << endl;

    // 第二问
    // 初始化
    memset(f, -0x3f, sizeof f);
    f[0][0] = 0;
    for(int i = 1; i <= n; i++)
    {
        for(int j = 0; j <= m; j++)
        {
            f[i][j] = f[i - 1][j];
            if(j >= v[i])
            {
                f[i][j] = max(f[i][j], f[i - 1][j - v[i]] + w[i]);
            }
        }
    }

    if(f[n][m] < 0) cout << 0 << endl;
    else cout << f[n][m] << endl;

    return 0;
}

// 空间优化的 01背包
#include <iostream>
#include <cstring>

using namespace std;

const int N = 1010;

int n, m;
int v[N], w[N];
int f[N];

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> v[i] >> w[i];

    // 第一问
```

```cpp
    for(int i = 1; i <= n; i++)
    {
        for(int j = m; j >= v[i]; j--) // 修改遍历顺序
        {
            f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
    }

    cout << f[m] << endl;

    // 第二问
    // 初始化
    memset(f, -0x3f, sizeof f);
    f[0] = 0;
    for(int i = 1; i <= n; i++)
    {
        for(int j = m; j >= v[i]; j--) // 修改遍历顺序
        {
            f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
    }

    if(f[m] < 0) cout << 0 << endl;
    else cout << f[m] << endl;

    return 0;
}
```

3.1.2 采药

> **题目来源：** 洛谷
>
> **题目链接：** P1048 [NOIP2005 普及组] 采药
>
> **难度系数：** ★

【题目描述】

> 辰辰是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。”
>
> 如果你是辰辰，你能完成这个任务吗？

【输入描述】

> 第一行有 2 个整数 $T(1 \leq T \leq 1000)$ 和 $M(1 \leq M \leq 100)$，用一个空格隔开，T 代表总共能够用来采药的时间，M 代表山洞里的草药的数目。
>
> 接下来的 M 行每行包括两个在 1 到 100 之间（包括 1 和 100）的整数，分别表示采摘某株草药的时间和这株草药的价值。

【输出描述】

> 输出在规定的时间内可以采到的草药的最大总价值。

【示例一】

> 输入：
>
> 70 3
>
> 71 100
>
> 69 1
>
> 1 2
>
> 输出：
>
> 3

【解法】

基本 01 背包问题，将时间看成体积，就是标准的不放满的 01 背包问题。不再赘述~

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 1010;

int n, m;
int t[N], w[N];
int f[N];

int main()
{
    cin >> m >> n;
    for(int i = 1; i <= n; i++) cin >> t[i] >> w[i];

    for(int i = 1; i <= n; i++)
        for(int j = m; j >= t[i]; j--) // 修改遍历顺序
        {
            f[j] = max(f[j], f[j - t[i]] + w[i]);
```

```
            }

        cout << f[m] << endl;

        return 0;
    }
```

3.1.3 小 A 点菜

题目来源： 洛谷

题目链接： P1164 小A点菜

难度系数： ★★

【题目描述】

uim 神犇拿到了 uoi 的 ra（镭牌）后，立刻拉着基友小 A 到了一家……餐馆，很低端的那种。

uim 指着墙上的价目表（太低级了没有菜单），说：“随便点”。

不过 uim 由于买了一些书，口袋里只剩 M 元 $(M \leq 10000)$。

餐馆虽低端，但是菜品种类不少，有 N 种 $(N \leq 100)$，第 i 种卖 a_i 元 $(a_i \leq 1000)$。由于是很低端的餐馆，所以每种菜只有一份。

小 A 奉行“不把钱吃光不罢休”的原则，所以他点单一定刚好把 uim 身上所有钱花完。他想知道有多少种点菜方法。

由于小 A 肚子太饿，所以最多只能等待 1 秒。

【输入描述】

第一行是两个数字，表示 N 和 M。

第二行起 N 个正数 a_i（可以有相同的数字，每个数字均在 1000 以内）。

【输出描述】

一个正整数，表示点菜方案数，保证答案的范围在 int 之内。

【示例一】

输入：

```
4 4
1 1 2 2
```

输出：

```
3
```

【解法】

背包问题求方案数，稍微修改一个状态表示，然后根据具体问题分析状态转移方程和初始化即可。

1. 状态表示：

   $dp[i][j]$ 表示：从前 i 个菜中挑选，总价钱恰好等于 j ，此时的总方案数。

2. 状态转移方程：

   针对 $a[i]$ 选或者不选，分两种情况讨论：

   a. 如果不选 $a[i]$ ：相当于去前 $i-1$ 个菜中挑选，总价钱恰好为 j 的方案数，此时的方案数就是 $dp[i-1][j]$ ；

   b. 如果选 $a[i]$ ：那么应该去前 $i-1$ 个菜中挑选，总价值恰好为 $j-a[i]$ ，此时的方案数就是 $dp[i-1][j-a[i]]$ ；

   因为我们要的是总方案数，于是 $dp[i][j]=dp[i-1][j]+dp[i-1][j-a[i]]$ 。

   注意第二个状态可能不存在，要注意判断一下 $j \geq a[i]$ 。

3. 初始化：

   $dp[0][0]=1$ ，如果没有物品，想凑成总体积为 0 是可行的，啥也不选就是一种方案。当然，这个状态也是为了让后面的值是正确的。

   其余位置的值是 0 就不影响填表的正确性。

4. 填表顺序：

   从上往下每一行，每一行从左往右。

   空间优化版本：每一行从右往左。

【参考代码】

```
#include <iostream>

using namespace std;

const int N = 110, M = 10010;

int n, m;
int a[N];
int f[N][M];

int main()
```

```cpp
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> a[i];

    f[0][0] = 1;
    for(int i = 1; i <= n; i++)
    {
        for(int j = 0; j <= m; j++)
        {
            f[i][j] = f[i - 1][j];
            if(j >= a[i]) f[i][j] += f[i - 1][j - a[i]];
        }
    }

    cout << f[n][m] << endl;

    return 0;
}
```

3.1.4 Cow Frisbee Team

> **题目来源：** 洛谷
>
> **题目链接：** P2946 [USACO09MAR] Cow Frisbee Team S
>
> **难度系数：** ★★★

【题目描述】

> 老唐最近迷上了飞盘，约翰想和他一起玩，于是打算从他家的 N 头奶牛中选出一支队伍。
>
> 每只奶牛的能力为整数，第 i 头奶牛的能力为 R_i。飞盘队的队员数量不能少于 1、大于 N。一支队伍的总能力就是所有队员能力的总和。
>
> 约翰比较迷信，他的幸运数字是 F，所以他要求队伍的总能力必须是 F 的倍数。请帮他算一下，符合这个要求的队伍组合有多少？由于这个数字很大，只要输出答案对 10^8 取模的值。

【输入描述】

> 第一行：两个用空格分开的整数：N 和 F。
>
> 第二行到 $N+1$ 行：第 $i+1$ 行有一个整数 R_i，表示第 i 头奶牛的能力。
>
> 对于 100% 的数据，$1 \le N \le 2000$，$1 \le F \le 1000$，$1 \le R_i \le 10^5$。

【输出描述】

> 第一行：单个整数，表示方案数对 10^8 取模的值。

【示例一】

> 输入：
>
> 4 5
>
> 1
>
> 2
>
> 8
>
> 2
>
> 输出：
>
> 3

【解法】

01 背包问题变形。

1. 状态表示：

   $dp[i][j]$ 表示：从前 i 头奶牛中挑选，总和模 f 之后为 j 时，一共有多少种组合。

   那么 $dp[n][0]-1$ 就是最终结果。（因为动态规划会把全都不选这种情况也考虑进去，所以要剪掉）

2. 状态转移方程：

   对于第 i 头奶牛选或者不选，可以分为两种情况讨论：

   a. 如果不选 $a[i]$：此时的总方案数就是去 $[1,i-1]$ 里面凑余数正好是 j ，也就是 $dp[i-1][j]$ ；

   b. 如果选 $a[i]$：此时已经有一个余数为 $a[i]\ \%\ f$ ，只用再去前面凑一个 $j-a[i]\ \%\ f$ 即可。但是直接减可能会减出来一个负数，我们要把它补正，最终凑的数为 $(j-a[i]\ \%\ f+f)\ \%\ f$ 。那么总方案数就是 $dp[i-1][(j-a[i]\ \%\ f+f)\ \%\ f]$ 。

   因为要的总方案数，所以状态转移方程就是上面两种情况的总和。

3. 初始化：

   $dp[0][0]=1$：什么也不选的时候，总和是 0 ，余数也是 0 ，属于一种方案，也是为了后续填表是正确的。

4. 填表顺序：

   从上往下每一行，每一行从左往右。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 2010, M = 1010, MOD = 1e8;

int n, m;
int a[N];
int f[N][M];

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> a[i];

    f[0][0] = 1;
    for(int i = 1; i <= n; i++)
    {
        for(int j = 0; j < m; j++)
        {
            f[i][j] = (f[i - 1][j] + f[i - 1][((j - a[i] % m) % m + m) % m]) % MOD;
        }
    }

    cout << f[n][0] - 1 << endl;

    return 0;
}
```


3.2 完全背包

3.2.1 完全背包

> **题目来源：** 牛客网
>
> **题目链接：** 【模板】完全背包
>
> **难度系数：** ★★

【题目描述】

> 你有一个背包，最多能容纳的体积是 V 。

> 现在有 n 种物品，每种物品有任意多个，第 i 种物品的体积为 v_i ,价值为 w_i 。
>
> 1. 求这个背包至多能装多大价值的物品?
> 2. 若背包恰好装满，求至多能装多大价值的物品?

【输入描述】

> 第一行两个整数 n 和 V ，表示物品个数和背包体积。
>
> 接下来 n 行，每行两个数 v_i 和 w_i ，表示第 i 种物品的体积和价值。
>
> $1 \leq n, V \leq 1000$

【输出描述】

> 输出有两行，第一行输出第一问的答案，第二行输出第二问的答案，如果无解请输出 0 。

【示例一】

> 输入：
>
> 2 6
>
> 5 10
>
> 3 1
>
> 输出：
>
> 10
>
> 2

【解法】

先解决第一问：

1. 状态表示：

   $dp[i][j]$ 表示：从前 i 个物品中挑选，总体积不超过 j ，所有的选法中，能挑选出来的最大价值。（这里是和 01 背包一样哒）

   那我们的最终结果就是 $dp[n][V]$ 。

2. 状态转移方程：

   线性 dp 状态转移方程分析方式，一般都是根据最后一步的状况，来分情况讨论。但是最后一个物品能选很多个，因此我们的需要分很多情况：

   a. 选 0 个第 i 个物品：此时相当于就是去前 $i-1$ 个物品中挑选，总体积不超过 j 。此时最大价值为 $dp[i-1][j]$ ；

   b. 选 1 个第 i 个物品：此时相当于就是去前 $i-1$ 个物品中挑选，总体积不超过 $j-v[i]$ 。因为挑选了一个 i 物品，此时最大价值为 $dp[i-1][j-v[i]]+w[i]$ ；

c. 选 2 个第 i 个物品：此时相当于就是去前 $i-1$ 个物品中挑选，总体积不超过 $j-2\times v[i]$ 。因为挑选了两个 i 物品，此时最大价值为 $dp[i-1][j-2\times v[i]]+2\times w[i]$ ；

d. .....

综上，状态转移方程为：

$$dp[i][j]=max(dp[i-1][j],dp[i-1][j-v[i]]+w[i],dp[i-1][j-2\times v[i]]+2\times w[i]...$$

当计算一个状态的时候，需要一个循环才能搞定的时候，就要想到去优化。优化的方向就是用一个或者两个状态来表示这一堆的状态，通常就是用数学的方式做一下等价替换。

观察发现第二维是有规律的变化的，因此去看看 $dp[i][j-v[i]]$ 这个状态：

$dp[i][j-v[i]]=max(dp[i-1][j-v[i]],dp[i-1][j-2\times v[i]]+w[i],dp[i-1][j-3\times v[i]]+2\times w[i]...)$

我们发现，把 $dp[i][j-v[i]]$ 加上 $w[i]$ 正好和 $dp[i][j]$ 中除了第一项以外的全部一致，因们可以修改状态转移方程为： $dp[i][j]=max(dp[i-1][j],dp[i][j-v[i]]+w[i])$

3. 初始化：

我们多加一行，方便我们的初始化，此时仅需将第一行初始化为 0 即可。因为什么也不选，也能满足体积不小于 j 的情况，此时的价值为 0 。

4. 填表顺序：

根据状态转移方程，我们仅需从上往下填表即可。

接下来解决第二问：

第二问仅需修改一下初始化以及最终结果即可。

1. 初始化：

因为有可能凑不齐 j 体积的物品，因此我们把不合法的状态设置为负无穷。这样在取最大值的时候，就不会考虑到这个位置的值。负无穷一般设置为 $-0x3f3f3f3f$ 即可。

然后把 $dp[0][0]=0$ 修改成 0 ，因为这是一个合法的状态，最大价值是 0 ，也让后续填表是正确的。

2. 返回值：

在最后拿结果的时候，也要判断一下最后一个位置是不是小于 0 ，因为有可能凑不齐。

不能判断是否等于 $-0x3f3f3f3f$，因为这个位置的值会被更新，只不过之前的值太小，导致更新后还是小于 0 的。

【参考代码】（后续还有空间优化的版本）

```cpp
// 原始版本
#include <iostream>
#include <cstring>

using namespace std;

const int N = 1010;

int n, m;
int v[N], w[N];
int f[N][N];

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> v[i] >> w[i];

    // 第一问
    for(int i = 1; i <= n; i++)
    {
        for(int j = 0; j <= m; j++)
        {
            f[i][j] = f[i - 1][j];
            if(j >= v[i]) f[i][j] = max(f[i][j], f[i][j - v[i]] + w[i]);
        }
    }

    cout << f[n][m] << endl;

    // 第二问
    memset(f, -0x3f, sizeof f);
    f[0][0] = 0;
    for(int i = 1; i <= n; i++)
    {
        for(int j = 0; j <= m; j++)
        {
            f[i][j] = f[i - 1][j];
            if(j >= v[i]) f[i][j] = max(f[i][j], f[i][j - v[i]] + w[i]);
        }
    }
```

```cpp

    if(f[n][m] < 0) cout << 0 << endl;
    else cout << f[n][m] << endl;

    return 0;
}

// 空间优化的版本
#include <iostream>
#include <cstring>

using namespace std;

const int N = 1010;

int n, m;
int v[N], w[N];
int f[N];

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> v[i] >> w[i];

    // 第一问
    for(int i = 1; i <= n; i++)
    {
        for(int j = v[i]; j <= m; j++) // 从小到大循环
        {
            f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
    }

    cout << f[m] << endl;

    // 第二问
    memset(f, -0x3f, sizeof f);
    f[0] = 0;
    for(int i = 1; i <= n; i++)
    {
        for(int j = v[i]; j <= m; j++) // 从小到大循环
        {
            f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
    }

    if(f[m] < 0) cout << 0 << endl;
```

```
88        else cout << f[m] << endl;
89
90        return 0;
91    }
```

3.2.2 疯狂的采药

题目来源： 洛谷

题目链接： P1616 疯狂的采药

难度系数： ★

【题目描述】

LiYuxiang 是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同种类的草药，采每一种都需要一些时间，每一种也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。”

如果你是 LiYuxiang，你能完成这个任务吗？

此题和原题的不同点：

1. 每种草药可以无限制地疯狂采摘。
2. 药的种类眼花缭乱，采药时间好长好长啊！师傅等得菊花都谢了！

【输入描述】

输入第一行有两个整数，分别代表总共能够用来采药的时间 t 和代表山洞里的草药的数目 m。

第 2 到第 $(m+1)$ 行，每行两个整数，第 $(i+1)$ 行的整数 a_i, b_i 分别表示采摘第 i 种草药的时间和该草药的价值。

对于 100% 的数据，保证 $1 \le m \le 10^4, 1 \le t \le 10^7$，且 $1 \le m \times t \le 10^7, 1 \le a_i, b_i \le 10^4$。

【输出描述】

输出一行，这一行只包含一个整数，表示在规定的时间内，可以采到的草药的最大总价值。

【示例一】

输入：

```
70 3
71 100
69 1
```

```
1 2
```

输出：

```
140
```

【解法】

完全背包模版题。

1. 状态表示：

   $dp[i][j]$ 表示：从前 i 个药材中挑选，总时间不超过 j ，此时能采摘到的最大价值。

   那么 $dp[n][m]$ 就是结果。

2. 状态转移方程：

   对于 i 位置的药材，可以选择采 $0, 1, 2, 3...$ 个：

   a. 选 0 个：最大价值为 $dp[i-1][j]$ ；

   b. 选 1 个：最大价值为 $dp[i-1][j-t[i]]+v[i]$ ；

   c. 选 2 个：最大价值为 $dp[i-1][j-2\times t[i]]+2\times v[i]$ ；

   d. ...

   由于要的是最大价值，应该是上述所有情况的最大值。其中第二个往后的状态可以用 $dp[i][j-t[i]]+v[i]$ 替代，因此状态转移方程为 $dp[i][j]=max(dp[i-1][j], dp[i][j-t[i]]+v[i])$ 。

3. 初始化：

   全部初始化 0 ，不影响后续填表的正确性。

4. 填表顺序：

   从上往下每一行，每一行从左往右。

   空间优化版本也是如此。

【参考代码】

```
1  #include <iostream>
2
3  using namespace std;
4
5  typedef long long LL;
```

```cpp

const int N = 1e4 + 10, M = 1e7 + 10;

int n, m;
int t[N], w[N];
LL f[M];

int main()
{
    cin >> m >> n;
    for(int i = 1; i <= n; i++) cin >> t[i] >> w[i];

    for(int i = 1; i <= n; i++)
    {
        for(int j = t[i]; j <= m; j++)
        {
            f[j] = max(f[j], f[j - t[i]] + w[i]);
        }
    }

    cout << f[m] << endl;

    return 0;
}
```

3.2.3 Buying Hay

> **题目来源：** 洛谷
>
> **题目链接：** P2918 [USACO08NOV] Buying Hay S
>
> **难度系数：** ★★

【题目描述】

> 约翰的干草库存已经告罄，他打算为奶牛们采购 $H(1 \leq H \leq 50000)$ 磅干草。
>
> 他知道 $N(1 \leq N \leq 100)$ 个干草公司，现在用 1 到 N 给它们编号。第 i 公司卖的干草包重量为 $P_i(1 \leq P_i \leq 5,000)$ 磅，需要的开销为 $C_i(1 \leq C_i \leq 5,000)$ 美元。每个干草公司的货源都十分充足， 可以卖出无限多的干草包。
>
> 帮助约翰找到最小的开销来满足需要，即采购到至少 H 磅干草。

【输入描述】

> 第一行：两个整数，分别是 N 和 H
>
> 接下来 N 行：每行两个整数，分别表示 P_i 和 C_i

【输出描述】

> 一个整数，表示约翰至少采购 H 磅干草所用的最小开销。

【示例一】

> 输入：
>
> 2 15
>
> 3 2
>
> 5 3
>
> 输出：
>
> 9

【解法】

完全背包简单变形。

1. 状态表示：

   $dp[i][j]$ 表示：从前 i 个干草公司中挑选，总重量至少为 j 磅，此时的最小开销。

   注意注意注意！这是我们遇到的第三类限制情况。

   之前的限制条件为不超过 j ，或者是恰好等于 j 。这道题的限定条件是至少为 j ，也就是说可以超过 j 。这会对我们分析状态转移方程的时候造成影响。

   根据状态表示， $dp[n][m]$ 就是结果。

2. 状态转移方程：

   对于 i 位置的公司，可以选择买 $0, 1, 2, 3...$ 个：

   a. 选 0 个：开销为 $dp[i-1][j]$ ；

   b. 选 1 个：开销为 $dp[i-1][j-p[i]]+c[i]$ 。问题来了，状态表示里面是至少为 j ，也就是说 $j-p[i]$ 小于 0 也是合法的。因为公司提供了 $p[i]$ 的重量，大于 j ，是符合要求的。但是 dp 表的下标不能是负数，处理这种情况的方式就是对 $j-p[i]$ 与 0 取一个最大值 $max(j-p[i], 0)$ 。当重量很大的时，只用去前面凑重量为 0 的就足够了，这样就符合我们的状态表示了。因此，最终开销为 $dp[i-1][max(0, j-p[i])]+c[i]$

   c. 选 2 个：开销为 $dp[i-1][max(0, j-2\times p[i])]+2\times c[i]$ ；

   d. ...

   由于要的是最小开销，应该是上述所有情况的最小值。

   其中第二个往后的状态可以用 $dp[i][max(0, j-p[i])]+c[i]$ 替代，因此状态转移方程为 $dp[i][j]=max(dp[i-1][j], dp[i][max(0, j-p[i])]+c[i])$ 。

3. 初始化：

   全部初始化正无穷大 $0x3f3f3f3f$，然后 $dp[0][0]=0$，不影响后续填表的正确性。

4. 填表顺序：

   从上往下每一行，每一行从左往右。

   空间优化版本也是如此。

【参考代码】

```cpp
#include <iostream>
#include <cstring>

using namespace std;

const int N = 110, M = 50010;

int n, m;
int p[N], c[N];
int f[M];

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> p[i] >> c[i];

    memset(f, 0x3f, sizeof f);
    f[0] = 0;
    for(int i = 1; i <= n; i++)
    {
        for(int j = 0; j <= m; j++)
        {
            f[j] = min(f[j], f[max(0, j - p[i])] + c[i]);
        }
    }

    cout << f[m] << endl;

    return 0;
}
```

3.2.4 纪念品

题目来源： 洛谷

题目链接： P5662 [CSP-J2019] 纪念品

难度系数： ★★★

【题目描述】

小伟突然获得一种超能力，他知道未来 T 天 N 种纪念品每天的价格。某个纪念品的价格是指购买一个该纪念品所需的金币数量，以及卖出一个该纪念品换回的金币数量。

每天，小伟可以进行以下两种交易无限次：

1. 任选一个纪念品，若手上有足够金币，以当日价格购买该纪念品；
2. 卖出持有的任意一个纪念品，以当日价格换回金币。

每天卖出纪念品换回的金币可以立即用于购买纪念品，当日购买的纪念品也可以当日卖出换回金币。当然，一直持有纪念品也是可以的。

T 天之后，小伟的超能力消失。因此他一定会在第 T 天卖出所有纪念品换回金币。

小伟现在有 M 枚金币，他想要在超能力消失后拥有尽可能多的金币。

【输入描述】

第一行包含三个正整数 T, N, M，相邻两数之间以一个空格分开，分别代表未来天数 T，纪念品数量 N，小伟现在拥有的金币数量 M。

接下来 T 行，每行包含 N 个正整数，相邻两数之间以一个空格分隔。

第 i 行的 N 个正整数分别为 $P_{i,1}, P_{i,2}, ..., P_{i,N}$，其中 $P_{i,j}$ 表示第 i 天第 j 种纪念品的价格。

【输出描述】

输出仅一行，包含一个正整数，表示小伟在超能力消失后最多能拥有的金币数量。

【示例一】

输入：

```
6 1 100
50
20
25
20
25
50
```

输出：

> 305

【示例二】

> 输入：
>
> 3 3 100
>
> 10 20 15
>
> 15 17 13
>
> 15 25 16
>
> 输出：
>
> 217

【解法】

总策略：贪心。从前往后，一天一天的考虑如何最大金币。

因为纪念品可以在当天买，当天卖。因此所有的交易情况，就可以转换成"某天买隔天卖"的情况。那么，我们就可以贪心的将每一天能拿到的最大利润全都拿到手：

- 从第一天开始，把第一天的金币看成限制条件，第二天的金币看成价值，求出：在不超过 m 的情况下，能获得的最大价值 m_1；
- 然后时间来到第二天，把这一天的金币看成限制条件，第三天的金币看成价值，求出：在不超 m_1 的情况下，能获得的最大价值 m_2；
- 以此类推，直到把第 $t-1$ 天的情况计算出来，能获得最大价值就是结果。

接下来就处理，拿到第 i 行以及 $i+1$ 行数据，在最大金币数量为 m 的前提下，获得的最大利润是多少？

- 因为每一个纪念品都可以无限次购买；
- 把前一行看成限制，后一行减去前一行的值看成价值，就变成了标准的完全背包问题。

那我们的解决方法就是一行一行的跑完全背包，跑一行拿到最大价值，然后放到下一行继续跑，直到跑完倒数第二行。

完全背包的逻辑：

1. 状态表示：

   $dp[i][j]$ 表示从前 i 个纪念品中挑选，总花费不超过 j 的情况下，最大的利润。

   那么，$dp[n][m]+m$ 就是能得到的最大金币数量。

2. 状态转移方程：

   根据最后一个纪念品选的数量，分成如下情况：

   a. 如果选 0 个：能获得的最大金币数量为 $dp[i-1][j]$；

   b. 如果选 1 个：能获得的最大金币数量为 $dp[i-1][j-w[i]]+v[i]-w[i]$；

   c. 如果选 2 个：能获得的最大金币数量为 $dp[i-1][j-2\times w[i]]+2\times(v[i]-w[i])$；

   d. ...

   其中除了第一个状态外的所有状态都可以用 $dp[i][j-w[i]]+v[i]-w[i]$ 来表示，又因为要的是最大值，所以状态转移方程就是所有情况的最大值。

3. 初始化

   全为 0 即可。

【参考代码】

```cpp
#include <iostream>
#include <cstring>

using namespace std;

const int N = 110, M = 1e4 + 10;

int t, n, m;
int p[N][N];
int f[M];

// 完全背包
int solve(int v[], int w[], int m)
{
    // 清空数据
    memset(f, 0, sizeof f);
    for(int i = 1; i <= n; i++)
    {
        for(int j = v[i]; j <= m; j++)
        {
            f[j] = max(f[j], f[j - v[i]] + w[i] - v[i]);
        }
    }

    return m + f[m];
}
```

```
27
28  int main()
29  {
30      cin >> t >> n >> m;
31      for(int i = 1; i <= t; i++)
32          for(int j = 1; j <= n; j++)
33              cin >> p[i][j];
34
35      // 贪心
36      for(int i = 1; i < t; i++)
37      {
38          m = solve(p[i], p[i + 1], m);
39      }
40
41      cout << m << endl;
42
43      return 0;
44  }
```

3.3 多重背包

3.3.1 多重背包

> **题目来源：** 牛客网
>
> **题目链接：** 多重背包
>
> **难度系数：** ★★

【题目描述】

> 有 n 种物品，第 i 种物品有 x_i 个，每一个物品重量为 w_i ，价值为 v_i ，现有一个承重能力为 T 的背包，在不超过承重能力的情况下，背包种最多能装多少价值的物品。

【输入描述】

> 第一行输入两个整数 $n, T(1 \leq n, T \leq 100)$ ，代表物品种数和背包承重能力
>
> 接下来 n 行，每行三个整数 $x_i, w_i, v_i(1 \leq x_i, w_i \leq 20, 1 \leq v_i \leq 200)$ 描述一个物品，分别代表物品的个数、物品的重量、物品的价值。

【输出描述】

> 输出一行一个整数，表示在不超过承重能力的情况下，背包物品的最大价值。

【示例一】

> 输入：

> 2 8
>
> 4 2 100
>
> 2 4 100
>
> 输出：
>
> 400

【解法】

多重背包问题有两种解法：

1. 按照背包问题的常规分析方式，仿照完全背包，第三维枚举使用的个数；
2. 利用二进制可以表示一定范围内整数的性质，转化成 01 背包问题。

小建议：并不是所有的多重背包问题都能用二进制优化，而且优化版本的代码很长。因此，如果时间复杂度允许的情况下，能不优化就不优化~~~

解法一：常规分析

1. 状态表示：

   $dp[i][j]$ 表示：从前 i 个物品中挑选，总重量不超过 j 的情况下，最大的价值。

   $dp[n][m]$ 就是最终结果。

2. 状态转移方程：

   根据第 i 个物品选的个数，可以分 $x[i]+1$ 种情况：

   a. 选 0 个：价值为 $dp[i-1][j]$；

   b. 选 1 个：价值为 $dp[i-1][j-w[i]]+v[i]$；

   c. 选 2 个：价值为 $dp[i-1][j-2\times w[i]]+2\times v[i]$；

   d. ...

   e. 选 $x[i]$ 个：价值为 $dp[i-1][j-x[i]\times w[i]]+x[i]\times v[i]$。

   因为要的是最大价值，所以 $dp[i][j]$ 等于上述所有情况的最大值。但是要注意 $j-k*w[i]$ 要大于等于 0，并且不能按照完全背包的方式优化。（大家可以试着写一写，是不能由之前的状态表示的）

3. 初始化：

   全部为 0 就不影响最终结果。

4. 填表顺序：

   从上往下每一行，每一行从左往右。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 110;

int n, m;
int x[N], w[N], v[N];
int f[N];

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> x[i] >> w[i] >> v[i];

    for(int i = 1; i <= n; i++)
        for(int j = m; j >= 0; j--)
            for(int k = 0; k <= x[i] && k * w[i] <= j; k++)
                f[j] = max(f[j], f[j - k * w[i]] + k * v[i]);

    cout << f[m] << endl;

    return 0;
}
```


解法二：转化成 01 背包问题

优化方式：用二进制将 $x[i]$ 个物品分组。

连续的二进制数有一个性质，就是 $2^0 \sim 2^k$ 能够表示区间 $[1, 2^{k+1} - 1]$ 里面所有的整数。比如：

- $1, 2, 4, 8$ 可以表示 $[1, 15]$ 内所有的整数。具体原因可以参考整数的二进制表示，$1, 2, 4, 8$ 正好对应二进制表示中每一位的权值，所以排列组合起来就可以表示 $[1, 15]$ 内所有的整数。
- 同理 $1, 2, 4$ 就可以表示 $[1, 7]$ 内所有的整数。

根据这样一个性质，我们就可以把 $x[i]$ 拆成一些二进制数再加上多出来的数，这样的一组数就可以表示 $[1, x[i]]$ 内所有的整数，问题就变成了 01 背包。

比如 $x[i]=9$，$w[i]=2, v[i]=3$：

- $9=1+2+4+2$；
- 分成 4 组，每组的重量和价值分别为 $(2,3)$、$(4,6)$、$(8,12)$、$(4,6)$ 。

```cpp
#include <iostream>

using namespace std;

const int N = 110 * 5;

int n, m;
int w[N], v[N], pos;
int f[N];

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++)
    {
        int x, y, z; cin >> x >> y >> z;
        // 按照二进制拆分
        int t = 1;
        while(x >= t)
        {
            pos++;
            w[pos] = t * y;
            v[pos] = t * z;
            x -= t;
            t *= 2;
        }
        if(x) // 处理剩余
        {
            pos++;
            w[pos] = x * y;
            v[pos] = x * z;
        }
    }

    // 针对拆分之后的物品，做一次 01背包即可
    for(int i = 1; i <= pos; i++)
        for(int j = m; j >= w[i]; j--)
            f[j] = max(f[j], f[j - w[i]] + v[i]);

```

```
40        cout << f[m] << endl;
41
42        return 0;
43    }
```

3.3.2 摆花

> **题目来源：** 洛谷
>
> **题目链接：** P1077 [NOIP2012 普及组] 摆花
>
> **难度系数：** ★★

【题目描述】

> 小明的花店新开张，为了吸引顾客，他想在花店的门口摆上一排花，共 m 盆。通过调查顾客的喜好，小明列出了顾客最喜欢的 n 种花，从 1 到 n 标号。为了在门口展出更多种花，规定第 i 种花不能超过 a_i 盆，摆花时同一种花放在一起，且不同种类的花需按标号的从小到大的顺序依次摆列。
>
> 试编程计算，一共有多少种不同的摆花方案。

【输入描述】

> 第一行包含两个正整数 n 和 m，中间用一个空格隔开。
>
> 第二行有 n 个整数，每两个整数之间用一个空格隔开，依次表示 $a_1, a_2, \cdots, a_n$。
>
> 对于 100% 数据，有 $0 < n \leq 100, 0 < m \leq 100, 0 \leq a_i \leq 100$。

【输出描述】

> 一个整数，表示有多少种方案。注意：因为方案数可能很多，请输出方案数对 $10^6 + 7$ 取模的结果。

【示例一】

> 输入：
>
> 2 4
>
> 3 2
>
> 输出：
>
> 2

【解法】

题意：每一种花可以选 $[0, a[i]]$ 个，在总数恰好等于 m 时的总方案数。

正好是多重背包求方案数的模型，我们可以用多重背包的思考方式来解决这道题。

1. 状态表示：

   $dp[i][j]$ 表示：从前 i 个花中挑选，正好摆放 j 个花盆时的方案数。

2. 状态转移方程：

   根据第 i 种花选的个数 $k(0 \le k \le min(j, a[i]))$ 分情况讨论：

   - 如果当前花选了 k 盆，之前的花要去凑够 $j-k$ 盆，总的方案数就是 $dp[i-1][j-k]$；
   - 因为要的是总方案数，所以最终结果应该是 k 的变化过程中的状态的总和。

   $dp[i][j] = dp[i][j] + dp[i-1][j-k]$

3. 初始化：

   $dp[0][0] = 1$，相当于起始状态，为了让后续的填表有意义，不然全都是 0。

4. 填表顺序：

   从上往下每一行，每一行从左往右。

这道题就不能用二进制优化，因为这道题的背包，限定条件和价值是一一对应的，并且求的是方案数。如果用二进制优化会统计多余的情况，比如：

- 有两个物品，个数分别是 $3, 2$，要凑成总和为 4。
- 拆分之后为 $(1, 2)$、$(1, 1)$，跑一遍 01 背包之后，结果是 3，但是实际情况应该是 2。
- 原因是 $1, 2, 1$ 被统计了 2 次。但是在实际情况里，第一个物品全选，第二个物品选 1 个，只属于 1 种情况，而 01 背包的逻辑会统计两次。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 110, MOD = 1e6 + 7;

int n, m;
int a[N];
int f[N];
```

```cpp

int main()
{
    cin >> n >> m;
    for(int i = 1; i <= n; i++) cin >> a[i];

    f[0] = 1;
    for(int i = 1; i <= n; i++)
    {
        for(int j = m; j >= 0; j--)
        {
            for(int k = 1; k <= j && k <= a[i]; k++)
            {
                f[j] = (f[j] + f[j - k]) % MOD;
            }
        }
    }

    cout << f[m] << endl;

    return 0;
}
```

3.4 分组背包

3.4.1 通天之分组背包

> **题目来源：** 洛谷
>
> **题目链接：** P1757 通天之分组背包
>
> **难度系数：** ★★

【题目描述】

> 自 01 背包问世之后，小 A 对此深感兴趣。一天，小 A 去远游，却发现他的背包不同于 01 背包，他的物品大致可分为 k 组，每组中的物品相互冲突，现在，他想知道最大的利用价值是多少。

【输入描述】

> 两个数 m,n，表示一共有 n 件物品，总重量为 m。
>
> 接下来 n 行，每行 3 个数 a_i,b_i,c_i，表示物品的重量，利用价值，所属组数。
>
> $0 \le m \le 1000$，$1 \le n \le 1000$，$1 \le k \le 100$，a_i,b_i,c_i 在 int 范围内。

【输出描述】

> 一个数，最大的利用价值。

【示例一】

> 输入：
>
> 45 3
>
> 10 10 1
>
> 10 5 1
>
> 50 400 2
>
> 输出：
>
> 10

【解法】

跟之前的分析方式基本一致，相信你们自己就能把它做出来。

因为一个组里面最多只能挑一个元素，所以我们就以一个组为单位。

1. 状态表示：

   $dp[i][j]$ 表示从前 i 组中挑选物品，总重量不超过 j 的情况下，最大的价值。

   那么 $dp[n][m]$ 就是最终结果。

2. 状态转移方程：

   根据第 i 组选什么物品，可以分若干情况讨论。设选择的物品重量为 a ，价值为 b ，此时的最大价值就是 $dp[i-1][j-a]+b$ 。

   因为要的是最大值，所以考虑所有物品之后，取所有情况的最大值就是 $dp[i][j]$ 。

3. 初始化：

   全是 0 即可。

【参考代码】

```
#include <iostream>
#include <vector>

using namespace std;

typedef pair<int, int> PII;

```

```cpp
const int N = 1010;

int n, m, cnt;
vector<PII> g[N]; // g[i]
int f[N];

int main()
{
    cin >> m >> n;
    for(int i = 1; i <= n; i++)
    {
        int a, b, c; cin >> a >> b >> c;
        cnt = max(c, cnt);
        g[c].push_back({a, b});
    }

    // 动态规划
    for(int i = 1; i <= cnt; i++)
    {
        for(int j = m; j >= 0; j--)
        {
            // 在这一组中选择物品
            for(auto& t : g[i])
            {
                int a = t.first, b = t.second;
                if(j >= a) f[j] = max(f[j], f[j - a] + b);
            }
        }
    }

    cout << f[m] << endl;

    return 0;
}
```


3.4.2 排兵布阵

> **题目来源：** 洛谷
>
> **题目链接：** P5322 [BJOI2019] 排兵布阵
>
> **难度系数：** ★★★

【题目描述】

小 C 正在玩一款排兵布阵的游戏。在游戏中有 n 座城堡，每局对战由两名玩家来争夺这些城堡。每名玩家有 m 名士兵，可以向第 i 座城堡派遣 a_i 名士兵去争夺这个城堡，使得总士兵数不超过 m 。

如果一名玩家向第 i 座城堡派遣的士兵数严格大于对手派遣士兵数的两倍，那么这名玩家就占领了这座城堡，获得 i 分。

现在小 C 即将和其他 s 名玩家两两对战，这 s 场对决的派遣士兵方案必须相同。小 C 通过某些途径得知了其他 s 名玩家即将使用的策略，他想知道他应该使用什么策略来最大化自己的总分。

由于答案可能不唯一，你只需要输出小 C 总分的最大值。

【输入描述】

输入第一行包含三个正整数 s, n, m ，分别表示除了小 C 以外的玩家人数、城堡数和每名玩家拥有的士兵数。
接下来 s 行，每行 n 个非负整数，表示一名玩家的策略，其中第 i 个数 a_i 表示这名玩家向第 i 座城堡派遣的士兵数。

对于 100% 的数据：
$1 \leq s \leq 1001 \leq n \leq 1001 \leq m \leq 20000$
对于每名玩家 $a_i \geq 0$，$\sum_{i=1}^{n} a_i \leq m$

【输出描述】

输出一行一个非负整数，表示小 C 获得的最大得分。

【示例一】

输入：

```
1 3 10
2 2 6
```

输出：

```
3
```

【示例二】

输入：

```
2 3 10
2 2 6
0 0 0
```

输出：

```
8
```

【解法】

一个城堡一个城堡分析，对于第 i 个城堡，考虑派遣的人数应该在所有玩家对这个城堡派遣人数中考虑。比如示例二的第三个城堡，我们考虑派遣的人数就是 1 和 13（因为要严格大于两倍，大一点就是最好的）。

因此，把每一个城堡看成一个小组，所有玩家在这个城堡派遣的人数看成一个一个物品，要求的就是在派遣人数不超过 m 的情况下的最大得分，符合分组背包。

小优化：对每个城堡中玩家的派遣人数从小到大排序，这样在选择第 k 个人数的时候，总得分就是 $k \times i$。

1. 状态表示：

   $dp[i][j]$ 表示：分配前 i 个城堡，在总人数不超过 j 的情况下，最大的得分。

   那么 $dp[n][m]$ 就是最终结果。

2. 状态转移方程：

   根据第 i 个城堡分配的人数，分情况讨论。假设分配的是排序后的第 k 个元素，那么分配人数为 $a[i][k]$，此时的最大得分为 $dp[i-1][j-a[i][k]]+i\times k$。

   由于要的是最大值，状态转移方程就是上述所有合法的 k 里面的最大值。

3. 初始化：

   全部为 0 即可。

【参考代码】

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 110, M = 2e4 + 10;

int s, n, m;
int a[N][N];
int f[N][M];

int main()
{
```

```
    cin >> s >> n >> m;
    for(int i = 1; i <= s; i++)
    {
        for(int j = 1; j <= n; j++)
        {
            cin >> a[j][i]; // 翻转一下
            // 小贪心
            a[j][i] = a[j][i] * 2 + 1;
        }
    }

    // 把每一行排序
    for(int i = 1; i <= n; i++)
    {
        sort(a[i] + 1, a[i] + 1 + s);
    }

    // 分组背包
    for(int i = 1; i <= n; i++)
    {
        for(int j = 0; j <= m; j++)
        {
            f[i][j] = f[i - 1][j];
            for(int k = 1; k <= s && a[i][k] <= j; k++)
            {
                f[i][j] = max(f[i][j], f[i - 1][j - a[i][k]] + k * i);
            }
        }
    }

    cout << f[n][m] << endl;

    return 0;
}
```


3.5 混合背包

3.5.1 樱花

> **题目来源：** 洛谷
>
> **题目链接：** P1833 樱花
>
> **难度系数：** ★★

【题目描述】

> 爱与愁大神后院里种了 n 棵樱花树，每棵都有美学值 $C_i(0 \le C_i \le 200)$。爱与愁大神在每天上学前都会来赏花。爱与愁大神是生物学霸，他懂得如何赏花：一种樱花树看一遍过，一种樱花树最多看 $P_i(0 \le P_i \le 100)$ 遍，一种樱花树可以看无数遍。但是看每棵樱花树都有一定的时间 $T_i(0 \le T_i \le 100)$。爱与愁大神离去上学的时间只剩下一小会儿了。求解看哪几棵樱花树能使美学值最高且爱与愁大神能准时（或提早）去上学。

【输入描述】

> 共 $n+1$ 行：
>
> 第 1 行：现在时间 T_s（几时：几分），去上学的时间 T_e（几时：几分），爱与愁大神院子里有几棵樱花树 n。这里的 T_s, T_e，T_e 格式为：`hh:mm`，其中 $0 \le hh \le 23$，$0 \le mm \le 59$，且 hh, mm, n 均为正整数。
>
> 第 2 行到第 $n+1$ 行，每行三个正整数：看完第 i 棵树的耗费时间 T_i，第 i 棵树的美学值 C_i，看第 i 棵树的次数 P_i（$P_i = 0$ 表示无数次，P_i 是其他数字表示最多可看的次数 P_i）。
>
> 100% 数据：$T_e - T_s \le 1000$（即开始时间距离结束时间不超过 1000 分钟），$n \le 10000$。保证 T_e, T_s 为同一天内的时间。

【输出描述】

> 只有一个整数，表示最大美学值。

【示例一】

> 输入：
>
> 6:50 7:00 3
>
> 2 1 0
>
> 3 3 1
>
> 4 5 4
>
> 输出：
>
> 11

【解法】

分类讨论即可。

【参考代码】

```cpp
#include <iostream>

using namespace std;

```

```cpp
const int N = 1e4 + 10, M = 1010;

int n, m;
int t[N], c[N], p[N];
int f[M];

int main()
{
    int t1, t2, t3, t4; char ch;
    cin >> t1 >> ch >> t2 >> t3 >> ch >> t4 >> n;
    m = t3 * 60 + t4 - (t1 * 60 + t2);

    for(int i = 1; i <= n; i++) cin >> t[i] >> c[i] >> p[i];

    for(int i = 1; i <= n; i++)
    {
        if(p[i] == 0) // 完全背包
        {
            for(int j = t[i]; j <= m; j++)
            {
                f[j] = max(f[j], f[j - t[i]] + c[i]);
            }
        }
        else // 多重背包 或 01 背包
        {
            for(int j = m; j >= t[i]; j--)
            {
                for(int k = 1; k <= p[i] && k * t[i] <= j; k++)
                {
                    f[j] = max(f[j], f[j - t[i] * k] + c[i] * k);
                }
            }
        }
    }

    cout << f[m] << endl;

    return 0;
}
```


3.6 多维费用的背包问题

3.6.1 L 国的战斗之间谍

> **题目来源：** 洛谷
>
> **题目链接：** P1910 L 国的战斗之间谍
>
> **难度系数：** ★★

【题目描述】

> 俗话说的好：“知己知彼，百战不殆”。L 国的指挥官想派出间谍前往I国，于是，选人工作就落到了你身上。
>
> 你现在有 N 个人选，每个人都有这样一些数据： A （能得到多少资料）、 B （伪装能力有多差）、 C （要多少工资）。已知敌人的探查间谍能力为 M （即去的所有人 B 的和要小于等于 M ）和手头有 X 元钱，请问能拿到多少资料？

【输入描述】

> 第一行三个整数 N, M, X 代表总人数，敌国侦察能力和总钱数。
>
> 第二行至第 $N+1$ 行，每行三个整数 A_i, B_i, C_i 分别表示第 i 个人能得到的资料，他的伪装能力有多差和他要的工资。
>
> 数据范围： $1 \le n \le 100, 1 \le m \le 1000, 1 \le x \le 1000$。

【输出描述】

> 一行一个整数表示能得到的资料总数。

【示例一】

> 输入：
>
> ```
> 3 10 12
> 10 1 11
> 1 9 1
> 7 10 12
> ```
>
> 输出：
>
> ```
> 11
> ```

【解法】

无非就是在 01 背包的基础上多加了一维，那我们就把状态表示也加上一维即可。

1. 状态表示：

   $dp[i][j][k]$ 表示：从前 i 个人中挑选，伪装能力之和不超过 j ，总工资不超过 k ，此时能获取到的最多资料总数。

   那么 $dp[n][m][x]$ 就是结果。

2. 状态转移方程：

   根据第 i 个人选或者不选分两种情况讨论：

   a. 不选：此时的最多资料为 $dp[i-1][j][k]$；

   b. 选：那就要去前 $i-1$ 各种，凑伪装能力之和不超过 $j-b[i]$，总工资不超过 $k-c[i]$ 时的最多工资，再加上第 i 个人的工资。也就是 $dp[i-1][j-b[i]][k-c[i]]+a[i]$。

   取上述两种情况的最大值即可。注意第二种情况要特判一下。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 110, M = 1010;

int n, m, x;
int a[N], b[N], c[N];
int f[M][M];

int main()
{
    cin >> n >> m >> x;
    for(int i = 1; i <= n; i++) cin >> a[i] >> b[i] >> c[i];

    for(int i = 1; i <= n; i++)
        for(int j = m; j >= b[i]; j--)
            for(int k = x; k >= c[i]; k--)
                f[j][k] = max(f[j][k], f[j - b[i]][k - c[i]] + a[i]);

    cout << f[m][x] << endl;


    return 0;
}
```


4. 区间 dp

区间 dp 也是线性 dp 的一种，它用区间的左右端点来描述状态，通过小区间的解来推导出大区间的解。因此，区间 DP 的核心思想是将大区间划分为小区间，它的状态转移方程通常依赖于区间的划分点。

常用的划分点的方式有两个：

- 基于区间的左右端点，分情况讨论；
- 基于区间上某一点，划分成左右区间讨论。

4.1 回文字串

> **题目来源：** 洛谷
>
> **题目链接：** P1435 [IOI2000] 回文字串
>
> **难度系数：** ★★

【题目描述】

> 回文词是一种对称的字符串。任意给定一个字符串，通过插入若干字符，都可以变成回文词。此题的任务是，求出将给定字符串变成回文词所需要插入的最少字符数。
>
> 比如 $Ab3bd$ 插入 2 个字符后可以变成回文词 $dAb3bAd$ 或 $Adb3bdA$，但是插入少于 2 个的字符无法变成回文词。
>
> **注意**：此问题区分大小写。

【输入描述】

> 输入共一行，一个字符串。
>
> 对于全部数据，$0 < l \leq 1000$。

【输出描述】

> 有且只有一个整数，即最少插入字符数。

【示例一】

> 输入：
>
> Ab3bd
>
> 输出：
>
> 2

【解法】

先找重复子问题定义状态表示：

- 大问题是让整个字符串 $[1,n]$ 变成回文串的最小插入次数；
- 当我们发现这个字符串左右元素一样的时候，那就去看看 $[2,n-1]$ 变成回文的最小插入次数；
- 如果左右不相同，那么我们会在左边补上一个字符，或者右边补上一个字符，然后看看剩下区间的最小插入次数。

因此，重复的子问题就是看看某个区间变成回文串的最小插入次数。

1. 状态表示：

   $dp[i][j]$ 表示：字符串 $[i,j]$ 区间，变成回文串的最小插入次数。

   那么 $dp[1][n]$ 就是我们要的结果。

2. 状态转移方程：

   根据区间的左右端点，分情况讨论：

   a. 如果 $s[i]=s[j]$：那我们就去看看 $[i+1,j-1]$ 区间的最小插入次数，即 $dp[i+1][j-1]$；

   b. 如果 $s[i]\neq s[j]$：

   - 要么去左边补一个 $s[j]$，此时的最小插入次数为 $dp[i][j-1]+1$；
   - 要么去右边补一个 $s[i]$，此时的最小插入次数为 $dp[i+1][j]+1$。

   因为要的是最小值，所以状态转移方程为 $min(dp[i+1][j],dp[i][j-1])+1$。

3. 初始化以及填表顺序：

我们看 dp 表：

加载失败

可以发现如下性质：

- 白色部分是用不到的非法区域，因为这个区域中的左端点大于右端点，不符合区间的定义；
- 每一个格子填表的时候，需要左边的格子以及下边的格子；
- 当 $i = j$ 的时候，填格子会用到非法区域，并且 $i = 1$ 以及 $i = n$ 的时候会越界，需要特殊处理。

综上所述：

- 对于初始化：我们需要初始化对角线位置的值。因为对角线表示长度为 1 的字符串，本身就是回文串，里面的值是 0 即可。
- 对于填表顺序，我们有两种策略：
  a. 从下往上填写每一行，每一行从左往右。这样就能保证在填写 $[i, j]$ 位置时，$[i + 1, j]$ 以及 $[i, j - 1]$ 已经被更新过了；
  b. 第一维循环：小到大枚举区间长度 $len(2 \le len \le n)$；第二维循环：枚举区间左端点 i ；然后计算出区间右端点 $j = i + len - 1$。这样我们填表的时候，就是一个对角线一个对角线的填，不会产生越界访问的问题。

对于区间 dp 的填表顺序，我们一般选取第二种，会让我们的代码看着很清晰，也比较符合区间 dp 的推导过程，从小区间递推到大区间。

【参考代码】

```
#include <iostream>

using namespace std;

const int N = 1010;

int f[N][N];

int main()
{
    string s; cin >> s;
    int n = s.size();
    s = " " + s;

    for(int len = 1; len <= n; len++) // 枚举长度
        for(int i = 1; i + len - 1 <= n; i++) // 枚举左端点
        {
            int j = i + len - 1;
```

```
19
20              if(s[i] == s[j]) f[i][j] = f[i + 1][j - 1];
21              else f[i][j] = min(f[i + 1][j], f[i][j - 1]) + 1;
22          }
23
24      cout << f[1][n] << endl;
25
26      return 0;
27  }
```

4.2 Treats for the Cows

题目来源： 洛谷

题目链接： P2858 [USACO06FEB] Treats for the Cows G/S

难度系数： ★★

【题目描述】

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 $1, \ldots, N$ 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 $V_i(1 \leq V \leq 1000)$ 。
- 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。

【输入描述】

第一行：整数 N ；

接下来 N 行，分别表示 V_i 。

【输出描述】

一个整数，表示能得到的最大价钱。

【示例一】

> 输入：
>
> 5
>
> 1
>
> 3
>
> 1
>
> 5
>
> 2
>
> 输出：
>
> 43

【解法】

贪心：每次都拿两边最小的。反例：$4,1,5,3$。

- 贪心解：$3\times1+4\times2+1\times3+5\times4=34$
- 正解：$4\times1+1\times2+3\times3+5\times4=35$

原因是，鼠目寸光。看似当前把最小的拿走了，但是如果先拿走一个较大的，可能会把更小的暴露出来。

正解还是老老实实的区间 dp：

1. 状态表示：

   $dp[i][j]$ 表示：把区间 $[i,j]$ 的零食全部拿走，最多能得到多少钱。

2. 状态转移方程：

   根据先拿左边还是先拿右边，能分成两种情况讨论：

   a. 先拿左边，然后去 $[i+1,j]$ 区间获得最多的钱，即 $a[i]\times(n-len+1)+dp[i+1][j]$；

   b. 先拿右边，然后去 $[i,j-1]$ 区间获得最多的钱，即 $a[j]\times(n-len+1)+dp[i][j-1]$；

   因为要的是最多的钱，所以应该是上面两种情况的最大值。

3. 初始化：

   当区间长度为 1 时：$dp[i][i]=n\times a[i]$。要注意，长度为 1，那就是第 n 次拿。

4. 填表顺序：

   先枚举区间长度，再枚举左端点，右端点通过计算。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 2010;

int n;
int a[N];
int f[N][N];

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];

    for(int len = 1; len <= n; len++)
        for(int i = 1; i + len - 1 <= n; i++)
        {
            int j = i + len - 1;
            int cnt = n - len + 1;

            f[i][j] = max(f[i + 1][j] + a[i] * cnt, f[i][j - 1] + a[j] * cnt);
        }

    cout << f[1][n] << endl;

    return 0;
}
```


4.3 石子合并（弱化版）

> **题目来源：** 洛谷
>
> **题目链接：** P1775 石子合并（弱化版）
>
> **难度系数：** ★★

【题目描述】

> 设有 $N(N \leq 300)$ 堆石子排成一排，其编号为 $1, 2, 3, \cdots, N$。每堆石子有一定的质量 $m_i(m_i \leq 1000)$。现在要将这 N 堆石子合并成为一堆。每次只能合并相邻的两堆，合并的代价为

> 这两堆石子的质量之和，合并后与这两堆石子相邻的石子将和新堆相邻。合并时由于选择的顺序不同，合并的总代价也不相同。试找出一种合理的方法，使总的代价最小，并输出最小代价。

【输入描述】

> 第一行，一个整数 N。
>
> 第二行， N 个整数 m_i。

【输出描述】

> 输出文件仅一个整数，也就是最小代价。

【示例一】

> 输入：
>
> 4
>
> 2 5 3 1
>
> 输出：
>
> 22

【解法】

1. 状态表示：

   $dp[i][j]$ 表示：合并区间 $[i,j]$ 石子，最小的代价。

   那么 $dp[1][n]$ 就是结果

2. 状态转移方程：

   根据最后一步合并的情况，可以分成 $j-i$ 种情况。设最后一步合并的时候，两个区间的分割点为 k ，也就是区间被分成 $[i,k]$ 和 $[k+1,j]$ ，此时的最小代价为合并左边区间的最小代码 + 合并右边区间的最小代价 + 合并两个区间的代价，即
   $dp[i][k]+dp[k+1][j]+sum[i,k]+sum[k+1,j]$。

   其中 $sum[i,k]+sum[k+1,j]$ 其实就是整个区间的和，可以用前缀和数组预处理一下，就可快速求出来。

   因为要的是最小代价，所以状态转移方程就是所有 k 变化范围内的最小值。

3. 初始化：

   区间长度为 1 的时候，不需要合并，代价为 0 。

4. 填表顺序：

   先枚举区间长度，再枚举左端点，右端点通过计算。

【参考代码】

```cpp
#include <iostream>
#include <cstring>

using namespace std;

const int N = 310;

int n;
int f[N][N];
int sum[N]; // 前缀和数组

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++)
    {
        int x; cin >> x;
        sum[i] = sum[i - 1] + x;
    }

    // 初始化
    memset(f, 0x3f, sizeof f);
    for(int i = 0; i <= n; i++) f[i][i] = 0;

    for(int len = 2; len <= n; len++)
    {
        for(int i = 1; i + len - 1 <= n; i++)
        {
            int j = i + len - 1;
            int t = sum[j] - sum[i - 1];

            // 枚举分割点
            for(int k = i; k < j; k++)
            {
                // [i, k] [k + 1, j]
                f[i][j] = min(f[i][j], f[i][k] + f[k + 1][j] + t);
            }
        }
    }

    cout << f[1][n] << endl;

    return 0;
```

```
44    }
```

4.4 石子合并

> **题目来源：** 洛谷
>
> **题目链接：** P1880 [NOI1995] 石子合并
>
> **难度系数：** ★★★

【题目描述】

> 在一个圆形操场的四周摆放 N 堆石子，现要将石子有次序地合并成一堆，规定每次只能选相邻的 2 堆合并成新的一堆，并将新的一堆的石子数，记为该次合并的得分。
>
> 试设计出一个算法,计算出将 N 堆石子合并成 1 堆的最小得分和最大得分。

【输入描述】

> 数据的第 1 行是正整数 N，表示有 N 堆石子。
>
> 第 2 行有 N 个整数，第 i 个整数 a_i 表示第 i 堆石子的个数。
>
> $1 \leq N \leq 100, 0 \leq a_i \leq 20$。

【输出描述】

> 输出共 2 行，第 1 行为最小得分，第 2 行为最大得分。

【示例一】

> 输入：
>
> ```
> 4
> 4 5 9 4
> ```
>
> 输出：
>
> ```
> 43
> 54
> ```

【解法】

处理环形问题的技巧：倍增。

在数组后面，将原始数组复写一遍，然后在倍增之后的数组上做一次石子合并（弱化版），就能得到以所有位置为起点并且长度为 len 的最小合并代价。

【参考代码】

```cpp
#include <iostream>
#include <cstring>

using namespace std;

const int N = 210;

int n, m;
int s[N];
int f[N][N]; // 最小得分
int g[N][N]; // 最大得分

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++)
    {
        cin >> s[i];
        // 倍增
        s[i + n] = s[i];
    }
    m = n + n;

    // 前缀和
    for(int i = 1; i <= m; i++)
    {
        s[i] += s[i - 1];
    }

    // 初始化
    memset(f, 0x3f, sizeof f);
    memset(g, -0x3f, sizeof g);
    for(int i = 1; i <= m; i++)
    {
        f[i][i] = g[i][i] = 0;
    }

    for(int len = 1; len <= n; len++)
    {
        for(int i = 1; i + len - 1 <= m; i++)
        {
            int j = i + len - 1;
            int t = s[j] - s[i - 1];

            // 枚举分割点
            for(int k = i; k < j; k++)
```

```
            {
                // [i, k] [k + 1][j]
                f[i][j] = min(f[i][j], f[i][k] + f[k + 1][j] + t);
                g[i][j] = max(g[i][j], g[i][k] + g[k + 1][j] + t);
            }
        }
    }

    // 更新结果
    int ret1 = 0x3f3f3f3f, ret2 = -0x3f3f3f3f;
    for(int i = 1; i <= n; i++)
    {
        ret1 = min(ret1, f[i][i + n - 1]);
        ret2 = max(ret2, g[i][i + n - 1]);
    }

    cout << ret1 << endl << ret2 << endl;

    return 0;
}
```

4.5 248

> **题目来源：** 洛谷
>
> **题目链接：** P3146 [USACO16OPEN] 248 G
>
> **难度系数：** ★★★

【题目描述】

> 贝西喜欢在手机上下载游戏来玩，尽管她确实觉得对于自己巨大的蹄子来说，小小的触摸屏用起来相当笨拙。
>
> 她对当前正在玩的这个游戏特别感兴趣。游戏开始时给定一个包含 N 个正整数的序列（ $2 \leq N \leq 248$ ），每个数的范围在 $1 \ldots 40$ 之间。在一次操作中，贝西可以选择两个相邻且相等的数，将它们替换为一个比原数大 1 的数（例如，她可以将两个相邻的 7 替换为一个 8 ）。游戏的目标是最大化最终序列中的最大数值。请帮助贝西获得尽可能高的分数！

【输入描述】

> 第一行包含整数 N ，接下来 N 行表示给定的游戏序列。

【输出描述】

> 输出能生成的最大整数。

【示例一】

输入：

4

1

1

1

2

输出：

3

【解法】

1. 状态表示：

   $dp[i][j]$ 表示：将区间 $[i,j]$ 合并的只剩下一个元素后，能得到的最大值。

   至于为什么要定义合并剩一个元素，因为如果不这样定义，相邻两个区间最大值虽然一样，但是不一定能合并。

   那么 dp 表里面的最大值就是结果，因为有些区间可能无法合并。

2. 状态转移方程：

   跟石子合并的讨论方式一样，根据最后一次合并的情况，可以把区间分成 $[i,k]$ 和 $[k+1,j]$，要想能够合并，需要满足下面条件：

   a. 两者合并后的最大值一致，才能合并：$dp[i][k]=dp[k+1][j]$；

   b. 合并后的最大值不能是 0。如果是 0，说明根本就不能合并：$dp[i][k]\neq 0$。

   如果能合并，合并后的最大值就是 $dp[i][k]+1$。那么状态转移方程就是所有符合要求的 k 里面的最大值。

3. 初始化：

   所有长度为 1 的区间，合并后的最大值应该是自己。所以初始化所有的 $dp[i][i]=a[i]$，也就是对角线。

4. 填表顺序：

   先枚举区间长度，再枚举左端点，右端点通过计算。

【参考代码】

```cpp
#include <iostream>

using namespace std;

const int N = 255;

int n;
int a[N];
int f[N][N];

int main()
{
    cin >> n;
    int ret = 0;
    for(int i = 1; i <= n; i++)
    {
        cin >> a[i];
        f[i][i] = a[i];
        ret = max(ret, a[i]);
    }

    for(int len = 2; len <= n; len++)
    {
        for(int i = 1; i + len - 1 <= n; i++)
        {
            int j = i + len - 1;

            // 枚举分割点
            for(int k = i; k < j; k++)
            {
                // [i, k] [k + 1, j]
                if(f[i][k] && f[i][k] == f[k + 1][j])
                {
                    f[i][j] = max(f[i][j], f[i][k] + 1);
                }
            }
            // 更新结果
            ret = max(ret, f[i][j]);
        }
    }

    cout << ret << endl;

    return 0;
}
```