第5章 动态规划基础

在本章中，我们将学习记忆化搜索和背包问题等动态规划基础。动态规划（Dynamic Programming，DP）不是一种具体的算法，而是一种思想，是一种通过把原问题分解为相对简单的子问题的方法，针对满足特定条件的一类问题，对各状态维度进行分阶段、有顺序、无重复、决策性地遍历求解。

5.1 记忆化搜索

例题　[USACO1.5][IOI1994]数字三角形 Number Triangles（洛谷 P1216）

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。

```
          7
        3   8
      8   1   0
    2   7   4   4
  4   5   2   6   5
```

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

【输入格式】

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。

$1 \le r \le 1000$，所有输入在 $[0,100]$ 范围内。

【输出格式】

单独的一行,包含那个可能得到的最大的和。

【分析】

设 $dfs(x,y)$ 表示从第 x 行第 y 列出发，走到最后一行的所有方案的数字之和最大值。很容易得到递推式 $dfs(x,y) = \max\big(dfs(x+1,y), dfs(x+1,y+1)\big) + a[x][y]$。可以看到，原问题 $dfs(x,y)$ 分成了两个规模更小的子问题 $dfs(x+1,y)$ 和 $dfs(x+1,y+1)$，我们可以用同样的方法对这两个子问题进行求解，于是有如下代码。

```
1. const int M = (int)1e3;
2.
3. int n;
```

```
4. int a[M + 5][M + 5];
5.
6. int dfs(int x, int y) {
7.     if (x == n) return a[x][y];
8.     return a[x][y] + max(dfs(x + 1, y), dfs(x + 1, y + 1));
9. }
10.
11. void work() {
12.     scanf("%d", &n);
13.     for (int i = 1; i <= n; i++)
14.         for (int j = 1; j <= i; j++)
15.             scanf("%d", &a[i][j]);
16.     printf("%d\n", dfs(1, 1));
17. }
```

上述代码正确实现了递推式，但是仔细分析后我们可以发现其时间复杂度是指数级别的，这是因为在计算 $dfs(x-1,y-1)$ 时需要先计算出 $dfs(x,y)$，而在计算 $dfs(x-1,y)$ 时也需要先计算出 $dfs(x,y)$，导致了状态 $dfs(x,y)$ 的重复计算。

为解决 $dfs(x,y)$ 的重复计算问 x,y 题，我们可以使用数组 $f[x][y]$ 记录 $dfs(x,y)$ 的计算结果，当 $f[x][y]=-1$ 时表示 $dfs(x,y)$ 之前未计算过，否则 $f[x][y]$ 表示 $dfs(x,y)$ 的计算结果。当求解 $dfs(x,y)$ 时，若 $f[x][y]\neq -1$ 则直接返回 $f[x][y]$，否则按照递推式进行计算并保存结果到 $f[x][y]$。

由于每个状态只需要计算一次，因此时间复杂度为 $O(n^2)$。

```
1. const int M = (int)1e3;
2.
3. int n;
4. int a[M + 5][M + 5];
5. int f[M + 5][M + 5];
6.
7. int dfs(int x, int y) {
8.     if (~f[x][y]) return f[x][y];
9.     if (x == n) return f[x][y] = a[x][y];
10.     return f[x][y] = a[x][y] + max(dfs(x + 1, y), dfs(x + 1, y + 1));
11. }
12.
13. void work() {
14.     scanf("%d", &n);
15.     for (int i = 1; i <= n; i++)
16.         for (int j = 1; j <= i; j++)
17.             scanf("%d", &a[i][j]), f[i][j] = -1;
18.     printf("%d\n", dfs(1, 1));
```

```
19. }
```

这就是一道经典的记忆化搜索题目，在暴力搜索的基础上添加了记忆数组 $f[x][y]$，以此来避免重复计算。在动态规划中，我们习惯将 $dfs(x,y)$ 称为状态，将递推式称为状态转移方程。

例题　台阶问题（洛谷 P1192）

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。

【输入格式】

两个正整数 N,K。

$1\leq N\leq 10^5$，$1\leq K\leq 10^2$。

【输出格式】

一个正整数 $ans\ \%100003$，为到达第 N 级台阶的不同方式数。

【分析】

设状态 $dfs(u)$ 表示当前在第 u 级台阶，到达第 N 级台阶的方案数，则有状态转移方程 $dfs(u)=\sum_{1\leq i\leq k} dfs(u+i)$。总共有 N 个状态，计算每个状态的时间复杂度为 $O(k)$，因此总时间复杂度为 $O(NK)$。

```
1. const int M = (int)1e5;
2. const int mod = (int)100003;
3.
4. int n, k;
5. int f[M + 5];
6.
7. int dfs(int u) {
8.     if (~f[u]) return f[u];
9.     if (u == n) return f[u] = 1;
10.    f[u] = 0;
11.    for (int i = 1; i <= k && u + i <= n; i++) (f[u] += dfs(u + i)) %= mod;
12.    return f[u];
13. }
14.
15. void work() {
16.    scanf("%d %d", &n, &k);
17.    memset(f, -1, sizeof(int) * (n + 1));
18.    printf("%d\n", dfs(0));
19. }
```

例题　可爱の星空（牛客竞赛 2021 秋季算法入门班第七章习题：动态规划 1）

“当你看向她时，有细碎星辰落入你的眼睛，真好。”——小可爱

在一个繁星闪烁的夜晚，卿念和清宇一起躺在郊外的草地上，仰望星空。

星语心愿，他们，想把这片星空的星星，连成一棵漂亮的树，将这美好的景色记录下来。

现在，天上共有 n 颗星星，编号分别为$1,2,\cdots,n$，一开始任何两个点之间都没有边连接。

之后，他们两个想在 (u,v) 之间连无向边，需要付出 $|u$ 联通块大小$-v$ 联通块大小$|$ 的代价。

他们两个想用最少的代价来使这 n 个点联通，所以他们想知道最小代价是多少。

【输入格式】

第一行一个正整数，表示数据组数 T。

接下来 T 行每行一个正整数，表示询问的 n。

对于 60% 的数据有 $1 \le T \le 10^5, 1 \le n \le 10^5$；

对于另外 40% 的数据有 $T = 1, 1 \le n \le 10^{12}$。

【输出格式】

T 行，每行一个数表示答案。

【分析】

经过手动模拟数据后，可以直观感觉到两个连通块的大小越接近越好。设状态 $f[n]$ 表示 n 个点联通的最小代价，那么当 n 为偶数时可以分为两个 $n/2$ 大小的两个连通块，有状态转移方程 $f[n] = f[n/2] \times 2$；当 n 为奇数时可以分为 $n/2$ 和 $n/2+1$ 大小的两个连通块，合并这两个连通块的代价为 1，有状态转移方程 $f[n] = f[n/2] + f[n/2+1] + 1$。

由于每次子问题的规模减半，因此时间复杂度为 $O(Tlogn)$。

```
unordered_map<ll, ll> f;

ll dfs(ll u) {
    if (f.count(u)) return f[u];
    if (u == 0 || u == 1) return f[u] = 0;
    if (u & 1) return f[u] = dfs(u / 2) + dfs(u / 2 + 1) + 1;
    return f[u] = dfs(u / 2) * 2;
}

void work() {
    ll n; scanf("%lld", &n);
    printf("%lld\n", dfs(n));
}
```

例题　[SHOI2002] 滑雪（洛谷 P1434）

Michael 喜欢滑雪。这并不奇怪，因为滑雪的确很刺激。可是为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，你不得不再次走上坡或者等待升降机来载你。Michael 想知道在一个区域中最长的滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。下面是一个例子：

1 2 3 4 5

16 17 18 19 6

15 24 25 20 7

14 23 22 21 8

13 12 11 10 9

一个人可以从某个点滑向上下左右相邻四个点之一，当且仅当高度会减小。在上面的例子中，一条可行的滑坡为 $24 \to 17 \to 16 \to 1$（从 24 开始，在 1 结束）。当然 $25 \to 24 \to 23 \to \cdots \to 3 \to 2 \to 1$ 更长。事实上，这是最长的一条。

【输入格式】

输入的第一行为表示区域的二维数组的行数 R 和列数 C。下面是 R 行，每行有 C 个数，代表高度(两个数字之间用 1 个空格间隔)。

$1 \leq R, C \leq 100$。

【输出格式】

输出区域中最长滑坡的长度。

【分析】

设状态 $f[x][y]$ 表示从点 (x, y) 出发所能走的最大长度，则有状态转移方程 $\max_{\substack{a[x'][y']<a[x][y] \\ |x-x'|+|y-y'|=1}} f[x'][y']$。时间复杂度为 $O(RC)$。

```
const int M = (int)1e2;

int n, m;
int a[M + 5][M + 5];
int f[M + 5][M + 5];
int dx[] = {0, 0, 1, -1};
int dy[] = {1, -1, 0, 0};

int dfs(int x, int y) {
    if (~f[x][y]) return f[x][y];
    f[x][y] = 1;
     for (int i = 0; i < 4; i++) {
```

```
13.         int vx = x + dx[i], vy = y + dy[i];
14.         if (vx < 1 || vx > n || vy < 1 || vy > m || a[x][y] <= a[vx][vy]) continue;
15.         f[x][y] = max(f[x][y], dfs(vx, vy) + 1);
16.     }
17.     return f[x][y];
18. }
19.
20. void work() {
21.     scanf("%d %d", &n, &m);
22.     for (int i = 1; i <= n; i++)
23.         for (int j = 1; j <= m; j++)
24.             scanf("%d", &a[i][j]), f[i][j] = -1;
25.     int mx = 0;
26.     for (int i = 1; i <= n; i++)
27.         for (int j = 1; j <= m; j++)
28.             mx = max(mx, dfs(i, j));
29.     printf("%d\n", mx);
30. }
```

5.2 动态规划基础

通过记忆化搜索可以引出动态规划的基本思想。能用动态规划解决的问题，需要满足三个条件：最优子结构，无后效性和子问题重叠。

最优子结构

具有最优子结构也可能是适合用贪心的方法求解。

注意要确保我们考察了最优解中用到的所有子问题。

1. 证明问题最优解的第一个组成部分是做出一个选择；
2. 对于一个给定问题，在其可能的第一步选择中，假定你已经知道哪种选择才会得到最优解。你现在并不关心这种选择具体是如何得到的，只是假定已经知道了这种选择；
3. 给定可获得的最优解的选择后，确定这次选择会产生哪些子问题，以及如何最好地刻画子问题空间；
4. 证明作为构成原问题最优解的组成部分，每个子问题的解就是它本身的最优解。方法是反证法，考虑加入某个子问题的解不是其自身的最优解，那么就可以从原问题的解中用该子问题的最优解替换掉当前的非最优解，从而得到原问题的一个更优的解，从而与原问题最优解的假设矛盾。

要保持子问题空间尽量简单，只在必要时扩展。

最优子结构的不同体现在两个方面：

1. 原问题的最优解中涉及多少个子问题；
2. 确定最优解使用哪些子问题时，需要考察多少种选择。

子问题图中每个定点对应一个子问题，而需要考察的选择对应关联至子问题顶点的边。

无后效性

已经求解的子问题，不会再受到后续决策的影响。

子问题重叠

如果有大量的重叠子问题，我们可以用空间将这些子问题的解存储下来，避免重复求解相同的子问题，从而提升效率。

基本思路

对于一个能用动态规划解决的问题，一般采用如下思路解决：

1. 将原问题划分为若干阶段，每个阶段对应若干个子问题，提取这些子问题的特征（称之为 状态）；
2. 寻找每一个状态的可能决策，或者说是各状态间的相互转移方式（用数学的语言描述就是状态转移方程）。
3. 按顺序求解每一个阶段的问题。

如果用图论的思想理解，我们建立一个有向无环图，每个状态对应图上一个节点，决策对应节点间的连边。这样问题就转变为了一个在 DAG 上寻找最长（短）路的问题。

例题　　方块与收纳盒（牛客竞赛 2021 秋季算法入门班第七章习题：动态规划 1）

现在有一个大小 $n \times 1$ 的收纳盒，我们手里有无数个大小为 1×1 和 2×1 的小方块，我们需要用这些方块填满收纳盒，请问我们有多少种不同的方法填满这个收纳盒。

【输入格式】

第一行是样例数 T。

第 2 到 $2+T-1$ 行每行有一个整数 n，描述每个样例中的 n。

$1 \leq T < 80, 1 \leq n \leq 80$。

【输出格式】

对于每个样例输出对应的方法数。

【分析】

设状态 $f[i]$ 表示大小为 $i \times 1$ 的收纳盒的方案数，则状态转移方程为 $f[i] = f[i-1] + f[i-2]$，表示 $i \times 1$ 的收纳盒的方案数可以分解为两个子问题：

1. 在 $(i-1)\times 1$ 的收纳盒的基础上添加一个大小为 1×1 的小方块；
2. 在 $(i-2)\times 1$ 的收纳盒的基础上添加一个大小为 2×1 的小方块。

我们可以预处理出 f 数组，每次查询 $O(1)$ 回答，时间复杂度为 $O(N+T)$。

```
const int M = (int)80;

ll f[M + 5];

void init() {
    f[1] = 1, f[2] = 2;
    for (int i = 3; i <= M; i++) f[i] = f[i - 1] + f[i - 2];
}

void work() {
    int n; scanf("%d", &n);
    printf("%lld\n", f[n]);
}
```

例题　舔狗舔到最后一无所有（牛客竞赛 2021 秋季算法入门班第七章习题：动态规划 1）

作为队伍的核心，forever97 很受另外两个队友的尊敬。

Trote_w 每天都要请 forever97 吃外卖，但很不幸的是宇宙中心 forever97 所在的学校周围只有 3 家 forever97 爱吃的外卖。

如果 Trote_w 给 forever97 买了别家的外卖，forever97 就会大喊“我不吃我不吃”。但是 forever97 又不喜欢连续三天吃一种外卖。

如果 Trote_w 哪天忘了这件事并且三天给他买了同一家外卖，那么 forever97 就会把 Trote_w 的头摁进手机屏幕里。

作为 Trote_w 的好朋友，你能告诉他连续请 forever97 吃 n 天饭，有多少不同的购买方法吗？

【输入格式】

多组样例第一行一个整数 T 代表测试样例数。

接下来 T 行每行一个整数 n，代表 Trote_w 要请 forever97 吃 n 天饭。

$1\le T\le 20, 1\le n\le 10^5$。

【输出格式】

输出 T 个整数代表方案数，由于答案太大，你只需要输出 mod 10^9+7 后的答案即可。

【分析】

解法一

设状态 $f[i][j][k]$ 表示考虑了前 i 天，第 $i-1$ 天吃 j，第 i 天吃 k 的方案数，则有状态转移方程 $f[i][j][k]=\sum_{l\neq j\ or\ l\neq k} f[i-1][l][j]$，也就是说对于 $f[i][j][k]$ 的计算，我们只需要枚举第 $i-2$ 天吃的饭 l，使得最近三天的饭不完全相同。时间复杂度为 $O(n+T)$。

```
const int M = (int)1e5;
const int mod = (int)1e9 + 7;

int f[M + 5][3][3];

void init() {
    for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) f[2][i][j] = 1;
    for (int i = 3; i <= M; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 3; k++) {
                for (int l = 0; l < 3; l++) if (!(l == j && j == k)) {
                    (f[i][j][k] += f[i - 1][l][j]) %= mod;
                }
            }
        }
    }
}

void work() {
    int n; scanf("%d", &n);
    if (n == 1) return printf("3\n"), void();
    ll ans = 0;
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++) (ans += f[n][i][j]) %= mod;
    printf("%lld\n", ans);
}
```


解法二

设状态 $f[i]$ 表示考虑了前 i 天且第 i 天吃 1 的方案数，根据轮换对称性不难看出“考虑了前 i 天且第 i 天吃 1 的方案数”，“考虑了前 i 天且第 i 天吃 2 的方案数”与“考虑了前 i 天且第 i 天吃 3 的方案数”三者是相同的。则有状态转移方程 $f[i]=2f[i-1]+2f[i-2]$，表示可以前 i 天且第 i 天吃 1 的方案数可以分解为两个子问题：

1. 第 $i-1$ 天吃 2 或 3，对 $f[i]$ 的贡献为 $2f[i-1]$。
2. 第 $i-1$ 天吃 1，那么第 $i-2$ 天必须吃 2 或者 3，对 $f[i]$ 的贡献为 $2f[i-2]$。

时间复杂度为 $O(n+T)$。

```
const int M = (int)1e5;
```

```
const int mod = (int)1e9 + 7;

int f[M + 5];

void init() {
    f[1] = 3, f[2] = 9;
    for (int i = 3; i <= M; i++) f[i] = 2ll * (f[i - 1] + f[i - 2]) % mod;
}

void work() {
    int n; scanf("%d", &n);
    printf("%d\n", f[n]);
}
```

可以看到同一个问题可以有不同的状态划分，不同的状态有各自的状态转移方程。解法一与解法二的时间复杂度相同，但是可以明显看到二者的常数不同，解法二的常数更小一点。通常，我们只需要找出符合题目时空限制的状态和状态转移方程，在此基础上可以按照时空复杂度和编码复杂度对解法进行选择。

例题　Longest Ordered Subsequence（POJ2533）

定义一个数列 $a_{1\sim n}$ 上升当且仅当 $a_1 < a_2 < \cdots < a_n$。

给定数列 $a_{1\sim n}$，找出其所有的上升子序列中长度的最大值，即最长上升子序列的长度。

【输入格式】

第一行输入一个整数 n 表述数列长度。

第二行输入 n 个整数 $a_{1\sim n}$。

$1 \le n \le 10^3, 0 \le a_i \le 10^4$。

【输出格式】

输出一个整数表示数列 a 的最长上升子序列的长度。

【分析】

解法一

设状态 $f[i]$ 表示考虑了 $a_{1\sim i}$ 且选择了 a_i 的上升子序列长度的最大值，则有状态转移方程 $f[i] = \sum_{\substack{1\le j<i \\ a_j<a_i}} f[j] + 1$，答案为 $\max\limits_{1\le i\le n} f[i]$。时间复杂度 $O(n^2)$。

```
const int M = (int)1e3;

int n, a[M + 5];
int f[M + 5];

int lis() {
```

```
    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);
    }
    return *max_element(f + 1, f + n + 1);
}

void work() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
    printf("%d\n", lis());
}
```

解法二

设 $f[i]$ 表示考虑了已扫描过的序列中，所有长度为 i 的上升子序列的末尾元素最小值。基于贪心中的决策包容性，可以保证对于同样长度的上升子序列，我们记录的是末尾元素最小值，那么这样在扫描后续序列时，就更有可能构造出更长的上升子序列。

可以证明 f 数组单调递增。使用反证法，假设 $f[i]=f[j]\ (i<j)$，那么 $f[j]$ 对于的长度为 j 的上升子序列中，其倒数第二个元素一定小于 $f[j]$，那么这个倒数第二个元素可以作为 $f[i]$，则有 $f[i]<f[j]$，与假设矛盾。同理可以反证 $f[i]>f[j]\ (i<j)$ 的情形。

假设考虑了前 $i-1$ 个元素时，f 数组的长度为 m。当 $a[i]>f[m]$ 时，那么上升子序列就可以扩展一个长度，即 $f[++m]=a[i]$；否则我们可以使用 $a[i]$ 更新 $f[1\sim m]$ 中的某个值，更新哪个值呢？根据贪心当然是更新 $f[1\sim m]$ 中第一个大于等于 $a[i]$ 的元素，即 $f[lower\_bound(f+1,f+m+1,a[i])-f]=a[i]$。

考虑完所有数之后，答案即为 m。时间复杂度为 $O(nlogn)$。

```
const int M = (int)1e3;

int n, a[M + 5];
int f[M + 5];

int lis() {
    int m = 0;
    f[0] = -1;
    for (int i = 1; i <= n; i++) {
        if (a[i] > f[m]) f[++m] = a[i];
        else f[lower_bound(f + 1, f + m + 1, a[i]) - f] = a[i];
    }
    return m;
```

```
14. }
15.
16. void work() {
17.     scanf("%d", &n);
18.     for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
19.     printf("%d\n", lis());
20. }
```

这就是著名的最长上升子序列（Longest Increasing Subsequence，LIS）算法。解法一基于动态规划，解法二基于贪心，虽然解法二的时间复杂度比解法一的低，但在统计 LIS 方案数时解法一更为方便和灵活。

例题　[NOIP1999 普及组] 导弹拦截（洛谷 P1020）

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，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。

【输入格式】

一行，若干个整数，中间由空格隔开。

导弹的个数不超过 10^5 个，弹的高度为正整数，且不超过 5×10^4。

【输出格式】

两行，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。

【分析】

容易看出最多拦截导弹数为原序列的最长非递增子序列的长度，关于导弹拦截系统最少数量的计算有两种方法。

解法一

基于贪心，我们维护一个 set 记录每套导弹拦截系统上一次拦截导弹的高度。依次扫描导弹，若当前导弹高度 $a[i]$ 大于 set 中的最大值，那么说明我们需要新加一套导弹拦截系统；否则，我们可以选择已有的一套导弹拦截系统去拦截当前导弹，选择那一套导弹拦截系统呢？当然是上一次拦截导弹的高度在数值上第一个大于等于 $a[i]$ 的那一套。

扫描完成后，答案即为 set 的大小。时间复杂度为 $O(nlogn)$。

```cpp
const int M = (int)1e5;
const int inf = 0x3f3f3f3f;

int f[M + 5];

int lis1(const vector<int>& v) {
    int n = 0;
    f[0] = inf;
    for (const int &x: v) {
        if (x <= f[n]) f[++n] = x;
        else {
            int l = 1, r = n, mid;
            while (l < r) {
                mid = (l + r) >> 1;
                if (f[mid] < x) r = mid;
                else l = mid + 1;
            }
            f[r] = x;
        }
    }
    return n;
}

int lis2(const vector<int>& v) {
    set<int> st;
    st.insert(v[0]);
    for (size_t i = 1; i < v.size(); i++) {
        if (v[i] > *--st.end()) st.insert(v[i]);
        else st.erase(st.lower_bound(v[i])), st.insert(v[i]);
    }
    return st.size();
}

void work() {
    vector<int> v;
    for (int x; ~scanf("%d", &x);) v.push_back(x);
    printf("%d\n", lis1(v));
    printf("%d\n", lis2(v));
}
```


解法二

在分析之前，我们先以比较容易理解的方式介绍一下代数关系基本概念和狄尔沃斯定理。

偏序集

定义非空集合 A 中的一个二元关系 $\leq$，譬如对于 (a_1,b_1) 和 (a_2,b_2) 两个元素，我们可以定义 $(a_1,b_1)\leq(a_2,b_2)$ 当且仅当 $a_1\leq a_2$ 且 $b_1\leq b_2$，但当 $a_1>a_2$ 且 $b_1\leq b_2$ 时这两个元素不可比。

在满足以下三个条件时，$(A,\leq)$ 称为一个偏序集：

1. 自反性：$\forall a\in A,a\leq a$。
2. 反对称性：$\forall a,b\in A$，若 $a\leq b,b\leq a$，则 $a=b$。
3. 传递性：$\forall a,b,c\in A$，若 $a\leq b,b\leq c$，则 $a\leq c$。

全序集

设 $\leq$ 为非空集合 A 上的一个偏序关系，若满足 $\forall a,b\in A$ 都有 $a\leq b$ 或 $b\leq a$，则 $(A,\leq)$ 是一个全序集。

反链

若偏序集 $(A,\leq)$ 中的元素两两不可比，则称 A 为反链。

狄尔沃斯（Dilworth）定理

狄尔沃斯定理亦称偏序集分解定理，是关于偏序集的极大极小的定理，该定理断言：对于任意有限偏序集，其最大反链中元素的数目必等于最小链划分中链的数目。此定理的对偶形式亦真，它断言：对于任意有限偏序集，其最长链中元素的数目必等于其最小反链划分中反链的数目。

设第 i 枚导弹的高度为 $a[i]$，我们可以在集合 $A=\{(i,a[i])|1\leq i\leq n\}$ 上定义二元关系 $(A,\leq)$ 为 $(i,a[i])\leq(j,a[j])$ 当且仅当 $i<j$ 且 $a[i]\geq a[j]$，那么导弹拦截系统个数为该二元关系上最小链划分中链的数目，根据狄尔沃斯定理可知其等于该二元关系上的最大反链中元素的数目，即满足 $i<j$ 且 $a[i]<a[j]$ 的最大元素个数，也就是原序列的最长上升子序列。

```
1. const int M = (int)1e5;
2. const int inf = 0x3f3f3f3f;
3.
4. int f[M + 5];
5.
6. int lis1(const vector<int>& v) {
7.     int n = 0;
8.     f[0] = inf;
9.     for (const int &x: v) {
10.        if (x <= f[n]) f[++n] = x;
11.        else {
```

```
            int l = 1, r = n, mid;
            while (l < r) {
                mid = (l + r) >> 1;
                if (f[mid] < x) r = mid;
                else l = mid + 1;
            }
            f[r] = x;
        }
    }
    return n;
}

int lis2(const vector<int>& v) {
    int n = 0;
    f[0] = 0;
    for (const int &x: v) {
        if (x > f[n]) f[++n] = x;
        else f[lower_bound(f + 1, f + n + 1, x) - f] = x;
    }
    return n;
}

void work() {
    vector<int> v;
    for (int x; ~scanf("%d", &x);) v.push_back(x);
    printf("%d\n", lis1(v));
    printf("%d\n", lis2(v));
}
```


解法三

在解法二中，我们使用最长非递增子序列计算拦截导弹的最大数目，狄尔沃斯定理计算导弹拦截系统的最少数目。仔细思考之后可以发现可以使用狄尔沃斯定理的对偶形式计算拦截导弹的最大数目。

设第 i 枚导弹的高度为 $a[i]$，我们可以在集合 $A=\{(i,a[i])|1\le i\le n\}$ 上定义二元关系 $(A,\le)$ 为 $(i,a[i])\le(j,a[j])$ 当且仅当 $i<j$ 且 $a[i]\ge a[j]$，那么导弹拦截最大数量为该二元关系上最长链中元素的数目，根据狄尔沃斯定理的对偶形式可知其等于该二元关系上的最小反链划分中链的数目，即对关系 $a[i]<a[j]\ (i<j)$ 的最小划分的链的数目，同样可以使用 set 贪心地进行维护。

```
int lis1(const vector<int>& v) {
    multiset<int> st;
    st.insert(v[0]);
```

```
    for (size_t i = 1; i < v.size(); i++) {
        if (v[i] <= *st.begin()) st.insert(v[i]);
        else st.erase(--st.lower_bound(v[i])), st.insert(v[i]);
    }
    return st.size();
}

int lis2(const vector<int>& v) {
    set<int> st;
    st.insert(v[0]);
    for (size_t i = 1; i < v.size(); i++) {
        if (v[i] > *--st.end()) st.insert(v[i]);
        else st.erase(st.lower_bound(v[i])), st.insert(v[i]);
    }
    return st.size();
}

void work() {
    vector<int> v;
    for (int x; ~scanf("%d", &x);) v.push_back(x);
    printf("%d\n", lis1(v));
    printf("%d\n", lis2(v));
}
```

例题　Common Subsequence（POJ1458）

给定两个字符串 s 和 t，求二者的最长公共子序列。

【输入格式】

输入包含多组测试数据，每组测试数据输入两个字符串 s 和 t。

$1 \le |s|, |t| \le 10^3$。

【输出格式】

对于每组测试数据输出一个整数，表示二者的最长公共子序列。

【分析】

设状态 $f[i][j]$ 表示考虑了字符串 s 的前 i 个字符和字符串 t 的前 j 个字符，所得到的最长公共子序列的长度，那么当 $s[i] \neq t[j]$ 时有 $f[i][j] = \max(f[i-1][j], f[i][j-1])$，表示；当 $s[i] = t[j]$ 时则有状态转移方程 $f[i][j] = \max(f[i-1][j], f[i][j-1], f[i-1][j-1]+1)$。时间复杂度为 $O(|s| \times |t|)$。

```
const int M = (int)1e3;

char s[M + 5], t[M + 5];
int f[M + 5][M + 5];
```

```
5.
6. void work() {
7.     int n = strlen(s + 1), m = strlen(t + 1);
8.     for (int i = 1; i <= n; i++) {
9.         for (int j = 1; j <= m; j++) {
10.            f[i][j] = max(f[i - 1][j], f[i][j - 1]);
11.            if (s[i] == t[j]) f[i][j] = max(f[i][j], f[i - 1][j - 1] + 1);
12.        }
13.    }
14.    printf("%d\n", f[n][m]);
15. }
```

这就是著名的最长公共子序列（Longest Common Subsubquence，LCS）问题，朴素的动态规划解法的时间复杂度为 $O(n^2)$。

例题　【模板】最长公共子序列（洛谷 P1439）

给出 $1,2,\cdots,n$ 的两个排列 P_1 和 P_2，求它们的最长公共子序列。

【输入格式】

一行是一个数 n。

接下来两行，每行为 n 个数，为自然数 $1,2,\cdots,n$ 的一个排列。

$1 \le n \le 10^5$。

【输出格式】

输出两个排列的最长公共子序列的长度。

【分析】

如果使用朴素的动态规划求解 LCS，时间复杂度为 $O(n^2)$，不可接受。由于给出的是两个排列，记 $idx[i]$ 表示数字 i 在排列 P_1 中的下标，那么我们可以构造出序列 $a = idx[P_2[i]]$，即排列 P_1 中每个元素在排列 P_2 中的位置。于是两个排列的 LCS 的长度等于序列 a 的 LIS 的长度。

举例来说，假设 $P_1 = [3,2,1,4,5], P_2 = [1,2,3,4,5]$，那么我们一定可以构造一个数值映射使得 $P_1 = [a,b,c,d,e], P_2 = [c,b,a,d,e]$，由于只是数值相对大小改变，所以 LCS 的长度不会变。可以观察到如此映射之后，排列 P_1 单调递增，又因为 LCS 一定是 P_1 的子序列，所以我们只需要找到 P_2 的 LIS。

时间复杂度为 $O(nlogn)$。

```
1. int n;
2. int id[M + 5], a[M + 5];
3. int f[M + 5];
4.
```

```
int lis() {
    int m = 0;
    for (int i = 1; i <= n; i++) {
        if (f[m] < a[i]) f[++m] = a[i];
        else f[upper_bound(f + 1, f + m + 1, a[i]) - f] = a[i];
    }
    return m;
}

void work() {
    scanf("%d", &n);
    for (int i = 1, x; i <= n; i++) scanf("%d", &x), id[x] = i;
    for (int i = 1, x; i <= n; i++) scanf("%d", &x), a[i] = id[x];
    printf("%d\n", lis());
}
```

例题　Mr. Young's Picture Permutations（POJ2279）

有 n 个学生合影，站成左端对齐的 k 排，每排分别有 $n_1, n_2, \cdots, n_k$ 个人，第 1 排站在最后边，第 k 排站在最前边。学生身高互不相同，把他们从高到低依次标记为 $1,2,\cdots,n$。在合影时要求每一排从左到右身高递减，每一列从后到前身高也递减，问一共有多少种安排合影位置的方案？

【输入格式】

输入包含多组数据，每组数组第一行输入一个整数 k，第二行输入 k 个数表示 $n_1, n_2, \cdots, n_k$。

$1 \le k \le 5, 1 \le n \le 30$。

【输出格式】

输出一个整数表示方案数。

【分析】

因为在合法的合影方案中每行、每列的身高都是单调的，所以我们可以从高到低依次考虑标记为 $1,2,\cdots,n$ 的学生站的位置。这样一来，在任意时刻，已经安排好位置的学生在每一行中占据的一定是从左端开始的连续若干个位置，用一个 k 元组 $(r_1, r_2, r_3, r_4, r_5)$ 表示每一行已安排的学生人数，即可描绘出已经处理部分的轮廓。

当安排一名新的学生时，我们考虑所有满足如下条件的行号 i：

1. $r_i \le n_i$
2. $i = 1$ 或 $r_{i-1} > r_i$。

只要该学生站在这样一行中，每列学生的身高单调性也就得以满足。换言之，我们不需

要考虑已经站好的 $\sum_{1\le i\le k} r_i$ 名学生的具体方案。k 元组 $(r_1, r_2, \cdots, r_k)$ 描绘的轮廓内的合影方案总数就足以构成一个子问题。

设状态 $f[i][r1][r2][r3][r4][r5]$ 表示考虑了前 i 高的人，第 i 排有 r_i 个人的方案数，根据均值不等式可知 $\prod_{1\le i\le 5} r_i \le \left(\frac{\sum_{1\le i\le 5} r_i}{5}\right)^5$，空间复杂度为 $O(n^6)$。事实上，可以发现 $i = \sum_{1\le i\le 5} r_i$，因此我们可以省略第一维。于是有新状态 $f[r_1][r_2][r_3][r_4][r_5]$ 表示考虑了前 $\sum_{1\le i\le 5} r_i$ 高的人，第 i 排有 r_i 个人的方案数。于是有状态转移方程 $f[r_1][r_2][r_3][r_4][r_5] = f[r_1-1][r_2][r_3][r_4][r_5] + f[r_1][r_2-1][r_3][r_4][r_5] + \cdots + f[r_1][r_2][r_3][r_4][r_5-1]$。

已知状态和状态转移方程后，动态规划在枚举过程一般有两种处理方法：填表法和刷表法。

填表法

枚举 $f[r_1][r_2][r_3][r_4][r_5]$，然后用 $f[r_1-1][r_2][r_3][r_4][r_5], f[r_1][r_2-1][r_3][r_4][r_5], \cdots, f[r_1][r_2][r_3][r_4][r_5-1]$去更新它。

```
int k, a[6];

void work() {
    memset(a, 0, sizeof a);
    for (int i = 1; i <= k; i++) scanf("%d", &a[i]);
    int n = accumulate(a + 1, a + k + 1, 0);
    vector<vector<vector<vector<vector<ll>>>>> f =
        vector<vector<vector<vector<vector<ll>>>>>(a[1] + 1,
        vector<vector<vector<vector<ll>>>>(a[2] + 1,
        vector<vector<vector<ll>>>(a[3] + 1,
        vector<vector<ll>>(a[4] + 1,
        vector<ll>(a[5] + 1, 0)))));
    f[0][0][0][0][0] = 1;
    for (int r1 = 0; r1 <= a[1]; r1++)
    for (int r2 = 0; r2 <= min(r1, a[2]); r2++)
    for (int r3 = 0; r3 <= min(r2, a[3]); r3++)
    for (int r4 = 0; r4 <= min(r3, a[4]); r4++)
    for (int r5 = 0; r5 <= min(r4, a[5]); r5++) if (r1 || r2 || r3 || r4 || r5) {
        if (r1) f[r1][r2][r3][r4][r5] += f[r1 - 1][r2][r3][r4][r5];
        if (r2) f[r1][r2][r3][r4][r5] += f[r1][r2 - 1][r3][r4][r5];
        if (r3) f[r1][r2][r3][r4][r5] += f[r1][r2][r3 - 1][r4][r5];
        if (r4) f[r1][r2][r3][r4][r5] += f[r1][r2][r3][r4 - 1][r5];
        if (r5) f[r1][r2][r3][r4][r5] += f[r1][r2][r3][r4][r5 - 1];
    }
    printf("%lld\n", f[a[1]][a[2]][a[3]][a[4]][a[5]]);
}
```

刷表法

使用 $f[r_1][r_2][r_3][r_4][r_5]$ 分别去更新 $f[r_1+1][r_2][r_3][r_4][r_5], f[r_1][r_2+1][r_3][r_4][r_5], \cdots, f[r_1][r_2][r_3][r_4][r_5+1]$。

```
int k, a[6];

void work() {
    memset(a, 0, sizeof a);
    for (int i = 1; i <= k; i++) scanf("%d", &a[i]);
    int n = accumulate(a + 1, a + k + 1, 0);
    vector<vector<vector<vector<vector<ll>>>>> f =
        vector<vector<vector<vector<vector<ll>>>>>(a[1] + 1,
        vector<vector<vector<vector<ll>>>>(a[2] + 1,
        vector<vector<vector<ll>>>(a[3] + 1,
        vector<vector<ll>>(a[4] + 1,
        vector<ll>(a[5] + 1, 0))))));
    f[0][0][0][0][0] = 1;
    for (int r1 = 0; r1 <= a[1]; r1++)
    for (int r2 = 0; r2 <= min(r1, a[2]); r2++)
    for (int r3 = 0; r3 <= min(r2, a[3]); r3++)
    for (int r4 = 0; r4 <= min(r3, a[4]); r4++)
    for (int r5 = 0; r5 <= min(r4, a[5]); r5++) {
        if (r1 < a[1]) f[r1 + 1][r2][r3][r4][r5] += f[r1][r2][r3][r4][r5];
        if (r2 < a[2]) f[r1][r2 + 1][r3][r4][r5] += f[r1][r2][r3][r4][r5];
        if (r3 < a[3]) f[r1][r2][r3 + 1][r4][r5] += f[r1][r2][r3][r4][r5];
        if (r4 < a[4]) f[r1][r2][r3][r4 + 1][r5] += f[r1][r2][r3][r4][r5];
        if (r5 < a[5]) f[r1][r2][r3][r4][r5 + 1] += f[r1][r2][r3][r4][r5];
    }
    printf("%lld\n", f[a[1]][a[2]][a[3]][a[4]][a[5]]);
}
```


在本题中，填表法和刷表法看起来没有什么区别，但在有些题目中，其中一种方法求解起来更加自然和简便。

例题　最长公共上升子序列（AcWing272）

给定两个长度为 n 的序列 A 和 B，求二者的最长公共上升子序列。

【输入格式】

第一行输入一个整数 n 表示序列长度。

第二行输入 n 个整数表示序列 A。

第三行输入 n 个整数表示序列 B。

$1 \le n \le 3 \times 10^3, 1 \le A_i, B_i \le 2^{31}-1$。

【输出格式】

输出两个序列的最长公共上升子序列的长度。

【分析】

这道题目是 LIS 和 LCS 的综合，因此我们可以考虑结合二者的状态。设状态 $f[i][j]$ 表示考虑了序列 A 的前 i 个数，序列 B 的前 j 个数，且以 B_j 结尾的 LCIS 长度。枚举 A_i 和 B_j，$f[i-1][j]$ 在 $f[i][j]$ 的预选决策集合中，此外若 $A_i = B_j$，则 $f[i][j]$ 的预选决策集合还包括 $f[i-1][k]$，即 $f[i][j] = \max\limits_{\substack{0 \le k < j \\ a[i] < b[k]}} f[i-1][k] + 1)$。

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        f[i][j] = f[i - 1][j];
        if (a[i] == b[j]) {
            for (int k = 0; k < j; k++)
                if (b[k] < a[i]) f[i][j] = max(f[i][j], f[i - 1][k] + 1);
        }
    }
}
```

注意到，当 j 遍历 $1 \sim m$ 时，i 是一个定值，这使得条件 $b_k < a_i$ 是固定的。因此，当变量 j 增加 1 时，k 的取值范围从 $0 \le k < j$ 变为 $0 \le k < j+1$，即整数 j 可能会进入新的决策集合中。也就是说，我们只需要 $O(1)$ 地检查条件 $B_j < A_i$ 是否满足，更新最大值，已经在决策集合中的数一定不会被去除。

时间复杂度为 $O(n^2)$。

```
const int M = (int)3e3;
const int inf = 0x3f3f3f3f;

int n, a[M + 5], b[M + 5];
int f[M + 5][M + 5];

void work() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
    for (int i = 1; i <= n; i++) scanf("%d", &b[i]);
    for (int i = 1; i <= n; i++) {
        int mx = 0;
        for (int j = 1; j <= n; j++) {
            f[i][j] = f[i - 1][j];
            if (a[i] == b[j]) f[i][j] = mx + 1;
            if (a[i] > b[j]) mx = max(mx, f[i - 1][j]);
```

```
17.         }
18.     }
19.     printf("%d\n", *max_element(f[n] + 1, f[n] + n + 1));
20. }
```

这道题转移部分的优化告诉我们，在实现状态转移方程时，要注意观察决策集合的范围随着状态的变化。对于决策集合中的元素只增多不减少的情景，就可以像本题一样维护一个变量来记录决策集合的当前信息，避免重复扫描，把转移的复杂度降低一个量级。

5.3 背包问题

背包是线性 DP 中一类重要而特殊的模型。在本节中讲述了 0/1 背包、完全背包、分组背包、混合背包和多维背包问题。此外，还有一类树上的背包问题，称为树上背包，将在树形 DP 章节内进行讲述。

5.3.1 0/1 背包

例题　　01 背包问题（AcWing2）

有 n 件物品和一个容量是 V 的背包。每件物品只能使用一次。

第 i 件物品的体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。输出最大价值。

【输入格式】

第一行两个整数 n,V，用空格隔开，分别表示物品数量和背包容积。

接下来有 n 行，每行两个整数 v_i,w_i，用空格隔开，分别表示第 i 件物品的体积和价值。

$1 \le n,V,v_i,w_i \le 10^3$。

【输出格式】

输出一个整数，表示最大价值。

【分析】

解法一

设状态 $f[i][j]$ 表示考虑了前 i 件物品，在容量为 j 的背包中所能获得的最大价值。可以直观的感受到，当 i 固定时，数组 $f[i][j]$ 是非严格单调递增的。假设不取第 i 件物品，则有状态转移方程 $f[i][j]=\max\limits_{0\le k\le j} f[i-1][k]=f[i-1][j]$；若取第 i 件物品，则需要消耗 v_i 的体积，同时带来 w_i 的价值，于是有状态转移方程 $f[i][j]=\max\limits_{0\le k\le j-v_i} f[i-1][k]+$

$w_i = f[i-1][j-v_i] + w_i)$。数组 f 的初始值为全 0，答案为 $f[n][V]$。时间复杂度为 $O(nV)$，空间复杂度为 $O(nV)$。

```
const int M = (int)1e3;

int f[M + 5][M + 5];

void work() {
    int n, V; scanf("%d %d", &n, &V);
    for (int i = 1, v, w; i <= n; i++) {
        scanf("%d %d", &v, &w);
        for (int j = 0; j <= V; j++) {
            f[i][j] = f[i - 1][j];
            if (j >= v) f[i][j] = max(f[i][j], f[i - 1][j - v] + w);
        }
    }
    printf("%d\n", f[n][V]);
}
```

解法二

设状态 $f[i][j]$ 表示考虑了前 i 件物品，恰好使用了 j 的体积的所有方案中所能获得的最大价值。假设不取第 i 件物品，则有状态转移方程 $f[i][j] = f[i-1][j]$；若取第 i 件物品，则需要消耗 v_i 的体积，同时带来 w_i 的价值，于是有状态转移方程 $f[i][j] = f[i-1][j-v_i] + w_i)$。数组初始值设置为 $f[0][0] = 0$，其余项全为 $-inf$ 表示非法状态或者未计算状态，答案为 $\max\limits_{0 \le j \le V} f[n][j]$。时间复杂度为 $O(nV)$，空间复杂度为 $O(nV)$。

```
const int M = (int)1e3;
const int inf = 0x3f3f3f3f;

int f[M + 5][M + 5];

void work() {
    int n, V; scanf("%d %d", &n, &V);
    memset(f, -inf, sizeof f);
    f[0][0] = 0;
    for (int i = 1, v, w; i <= n; i++) {
        scanf("%d %d", &v, &w);
        for (int j = 0; j <= V; j++) {
            f[i][j] = f[i - 1][j];
            if (j >= v) f[i][j] = max(f[i][j], f[i - 1][j - v] + w);
        }
    }
```

```
    printf("%d\n", *max_element(f[n], f[n] + V + 1));
}
```

解法三

在解法二状态转移方程中，可以看到 $f[i][]$ 的计算只与 $f[i-1][]$ 有关。在这种情况下，我们可以使用滚动优化的优化方法，降低空间开销。方法就是令 $u = i\&1$，把 i 替换为 u，把 $i-1$ 替换为 $u\text{^}1$。

时间复杂度为 $O(nV)$，空间复杂度为 $O(V)$。

```
const int M = (int)1e3;
const int inf = 0x3f3f3f3f;

int f[2][M + 5];

void work() {
    int n, V; scanf("%d %d", &n, &V);
    memset(f, -inf, sizeof f);
    f[0][0] = 0;
    for (int i = 1, u, v, w; i <= n; i++) {
        u = i & 1;
        scanf("%d %d", &v, &w);
        for (int j = 0; j <= V; j++) {
            f[u][j] = f[u ^ 1][j];
            if (j >= v) f[u][j] = max(f[u][j], f[u ^ 1][j - v] + w);
        }
    }
    printf("%d\n", *max_element(f[n & 1], f[n & 1] + V + 1));
}
```

解法四

进一步分析解法三中代码，容易发现，在每个阶段开始时，实际上执行了依一次从 $f[i-1][]$ 到 $f[i][]$ 的拷贝操作，这提示我们可以省略掉 $f[][]$ 的第一维。

时间复杂度为 $O(nV)$，空间复杂度为 $O(V)$。

```
const int M = (int)1e3;
const int inf = 0x3f3f3f3f;

int f[M + 5];

void work() {
    int n, V; scanf("%d %d", &n, &V);
    memset(f, -inf, sizeof f);
    f[0] = 0;
```

```
    for (int i = 1, v, w; i <= n; i++) {
        scanf("%d %d", &v, &w);
        for (int j = V; j >= v; j--) f[j] = max(f[j], f[j - v] + w);
    }
    printf("%d\n", *max_element(f, f + V + 1));
}
```

注意上面的代码，我们使用了倒序枚举，这是因为0/1背包问题中要求物品最多取一件。如果我们对 j 循环使用正序枚举，$f[0]$ 会更新 $f[v]$，而更新后的 $f[v]$ 又会更新 $f[2v]$，这就会产生同一件物品取多次的情况。反观如果我们对 j 循环使用倒序枚举，先用上一物品阶段计算出的 $f[v]$ 更新现阶段的 $f[2v]$，再用上一物品阶段计算出的 $f[0]$ 更新现阶段的 $f[v]$，就不会出现同一件物品取多次情况。

解法五

前四种的 dp 计算顺序是填表法，面对物品体积都为偶数的情况会浪费至少一半的计算开销。为此我们可以使用刷表法，只有当 $f[j] \geq 0$ 表示其为有效状态时，才使用 $f[j]$ 对 $f[j+v]$ 更新。

时间复杂度为 $O(nV)$，空间复杂度为 $O(V)$。

```
template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

const int M = (int)1e3;
const int inf = 0x3f3f3f3f;

int f[M + 5];

void work() {
    int n, V; scanf("%d %d", &n, &V);
    memset(f, -inf, sizeof f);
    f[0] = 0;
    for (int i = 0, v, w; i < n; i++) {
        scanf("%d %d", &v, &w);
        for (int j = V - v; j >= 0; j--) if (f[j] >= 0) cmax(f[j + v], f[j] + w);
    }
    printf("%d\n", *max_element(f, f + V + 1));
}
```

例题　数字组合（AcWing278）

给定 N 个正整数 $A_1, A_2 \cdots, A_N$，从中选出若干个数，使它们的和为 M，求有多少种选择方案。

【输入格式】

第一行包含两个整数 N 和 M。 第二行包含 N 个整数，表示 $A_1, A_2 \cdots, A_N$。

$1 \le N \le 10^2, 1 \le M \le 10^4, 1 \le A_i \le 10^3$。

【输出格式】

包含一个整数，表示可选方案数。

【分析】

不难发现，这道题目是求0/1背包的方案数。设状态 $f[i]$ 表示凑出和为 i 的方案个数，则有状态转移方程 $f[i+A] += f[i]$。初始值为 $f[0]=1$，答案为 $f[M]$。时间复杂度为 $O(NM)$，空间复杂度为 $O(M)$。

```
1. const int M = int(1e4);
2.
3. int f[M + 5];
4.
5. void work() {
6.     int n, m; scanf("%d %d", &n, &m);
7.     f[0] = 1;
8.     for (int i = 1, a; i <= n; i++) {
9.         scanf("%d", &a);
10.        for (int j = m - a; j >= 0; j--) if (f[j]) f[j + a] += f[j];
11.    }
12.    printf("%d\n", f[m]);
13. }
```


5.3.2 完全背包

例题　完全背包问题（AcWing3）

有 n 种物品和一个容量是 V 的背包，每种物品都有无限件可用。

第 i 种物品的体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。 输出最大价值。

【输入格式】

第一行两个整数 n, V，用空格隔开，分别表示物品数量和背包容积。

接下来有 n 行，每行两个整数 v_i, w_i，用空格隔开，分别表示第 i 件物品的体积和价值。

$1 \le n, V, v_i, w_i \le 10^3$。

【输出格式】

输出一个整数，表示最大价值。

【分析】

解法一

设状态 $f[i][j]$ 表示考虑了前 i 件物品，恰好使用了 j 体积的最大价值。若第 i 件物品不取，则有状态转移方程 $f[i][j]=f[i-1][j]$；否则有状态转移方程 $f[i][j]=f[i][j-v]+w$。时间复杂度为 $O(nV)$，空间复杂度为 $O(nV)$。

```
const int M = (int)1e3;
const int inf = 0x3f3f3f3f;

int f[M + 5][M + 5];

void work() {
    int n, V; scanf("%d %d", &n, &V);
    memset(f, -inf, sizeof f);
    f[0][0] = 0;
    for (int i = 1, v, w; i <= n; i++) {
        scanf("%d %d", &v, &w);
        for (int j = 0; j <= V; j++) {
            f[i][j] = f[i - 1][j];
            if (j >= v) f[i][j] = max(f[i][j], f[i][j - v] + w);
        }
    }
    printf("%d\n", *max_element(f[n], f[n] + V + 1));
}
```

解法二

类似于 0/1 背包问题的解法五，完全背包也可以使用滚动优化和刷表法。

时间复杂度为 $O(nV)$，空间复杂度为 $O(V)$。

```
template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

const int M = (int)1e3;
const int inf = 0x3f3f3f3f;

int f[M + 5];

void work() {
    int n, V; scanf("%d %d", &n, &V);
    memset(f, -inf, sizeof f);
    f[0] = 0;
    for (int i = 0, v, w; i < n; i++) {
        scanf("%d %d", &v, &w);
```

```
        for (int j = 0; j <= V - v; j++) if (f[j] >= 0) cmax(f[j + v], f[j] + w);
    }
    printf("%d\n", *max_element(f, f + V + 1));
}
```

与 0/1 背包问题唯一不同的是第二层循环的枚举顺序，在完全背包中使用了正序枚举，之前已经分析过，正序枚举会导致物品的重复计算，而这恰好是完全背包所需要的。

例题　自然数拆分（AcWing279）

给定一个自然数 N，要求把 N 拆分成若干个正整数相加的形式，参与加法运算的数可以重复。

注意:

1. 拆分方案不考虑顺序；
2. 至少拆分成 2 个数的和。

求拆分的方案数 $mod\ 2147483648$ 的结果。

【输入格式】

一个自然数 N。

$1 \le N \le 4 \times 10^3$。

【输出格式】

输入一个整数，表示结果。

【分析】

我们可以把 $1 \sim N$ 中的每个数看作一种物品，第 i 件物品的体积为 i。于是原问题可以看作是求完全背包的方案数。

设状态 $f[i]$ 为体积恰好为 i 的方案数，则有状态转移方程 $f[i+v]+=f[i]$。

时间复杂度为 $O(N^2)$，空间复杂度为 $O(N)$。

```
const int M = int(4e3);
const ll mod = 2147483648ll;

ll f[M + 5];

void work() {
    int n; scanf("%d", &n);
    f[0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= n - i; j++) if (f[j]) (f[j + i] += f[j]) %= mod;
    }
    printf("%lld\n", f[n] - 1);
```

```
}
```

5.3.3 多重背包

例题　多重背包问题 I（AcWing4）

有 n 种物品和一个容量是 V 的背包。

第 i 种物品最多有 s_i 件，每件体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。

输出最大价值。

【输入格式】

第一行两个整数 n,V，用空格隔开，分别表示物品数量和背包容积。

接下来有 n 行，每行两个整数 v_i,w_i,s_i，用空格隔开，分别表示第 i 件物品的体积、价值和数量。

$1 \le n,V,v_i,w_i,s_i \le 10^2$。

【输出格式】

输出一个整数，表示最大价值。

【分析】

我们可以把第 i 种物品看作独立的 s_i 个物品，转化共有 $\sum_{1 \le i \le n} s_i$ 件物品的 0/1 背包问题。时间复杂度为 $O(nsV)$，空间复杂度为 $O(V)$。

```
template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

const int M = (int)1e2;
const int inf = 0x3f3f3f3f;

int f[M + 5];

void work() {
    int n, V; scanf("%d %d", &n, &V);
    memset(f, -inf, sizeof f);
    f[0] = 0;
    for (int i = 0, v, w, s; i < n; i++) {
        scanf("%d %d %d", &v, &w, &s);
        while (s-- > 0) {
            for (int j = V - v; j >= 0; j--)
                if (f[j] >= 0) cmax(f[j + v], f[j] + w);
        }
    }
```

```
20.     printf("%d\n", *max_element(f, f + V + 1));
21. }
```

例题　多重背包问题 II（AcWing5）

有 n 种物品和一个容量是 V 的背包。

第 i 种物品最多有 s_i 件，每件体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。

输出最大价值。

【输入格式】

第一行两个整数 n,V，用空格隔开，分别表示物品数量和背包容积。

接下来有 n 行，每行两个整数 v_i,w_i,s_i，用空格隔开，分别表示第 i 件物品的体积、价值和数量。

$1 \le n \le 10^3, 1 \le V, v_i, w_i, s_i \le 2 \times 10^3$。

【输出格式】

输出一个整数，表示最大价值。

【分析】

倘若使用多重背包问题 I 的解法，简单地将第 i 种物品拆分为独立的 s_i 件物品，时间复杂度为 $O(nsV)$，在本题的数据范围中，计算规模约为 4×10^9，显然无法接受，我们考虑如何优化。

众所周知，从 $2^0, 2^1, \cdots, 2^{k-1}$ 这 k 个 2 的整数次幂中选出若干个相加，可以表示出 $0 \sim 2^k - 1$ 之间的任何整数。于是，我们可以求出满足 $\sum_{0 \le j \le p} 2^j \le s_i$ 的最大整数 p，设 $r_i = s_i - \sum_{0 \le j \le p} 2^j = s_i - (2^{p+1} - 1)$。那么，

1. 由于 p 的最大性，有 $\sum_{0 \le j \le p+1} 2^j > s_i$，可推出 $2^{p+1} > r_i$。因此从 $2^0, 2^1, \cdots, 2^p$ 中选出若干个相加可以表示出 $0 \sim r_i$ 之间的任何整数。
2. 从 $2^0, 2^1, \cdots, 2^p$ 以及 r_i 中选出若干个相加，可以表示出 $r_i \sim r_i + 2^{p+1} - 1$ 之间的任何整数，即 $r_i \sim s_i$ 之间的任何整数。

综上所述，我们可以把数量为 s_i 的第 i 种物品拆成 $p+2$ 件物品，它们的（体积，价值）分别为：

$$(2^0 v_i, 2^0 w_i), (2^1 v_i, 2^1 w_i), \cdots, (2^p v_i, 2^p w_i), (r_i v_i, r_i w_i)$$

这 $p+2$ 件物品的体积可以凑成 $0 \sim s_i v_i$ 之间所有能被 v_i 整除的数，并且不能凑成大于 $s_i v_i$ 的数，即等价于原来的 s_i 件物品。

这就是多重背包的二进制拆分法，将 s_i 件物品拆分成等价的 $\log(s_i)$ 件物品。时间复杂度为 $O(nlog(s)V)$，空间复杂度为 $O(V)$。

```
template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

const int M = (int)2e3;
const int inf = 0x3f3f3f3f;

int f[M + 5];

void work() {
    int n, V; scanf("%d %d", &n, &V);
    memset(f, -inf, sizeof f);
    f[0] = 0;
    for (int i = 0, v, w, s; i < n; i++) {
        scanf("%d %d %d", &v, &w, &s);
        vector<int> cnt;
        for (int i = 1; i <= s; i <<= 1) cnt.push_back(i), s -= i;
        if (s) cnt.push_back(s);
        for (const int &x: cnt) {
            for (int j = V - x * v; j >= 0; j--)
                if (f[j] >= 0) cmax(f[j + x * v], f[j] + x * w);
        }
    }
    printf("%d\n", *max_element(f, f + V + 1));
}
```

例题　多重背包问题 III（AcWing6）

有 n 种物品和一个容量是 V 的背包。

第 i 种物品最多有 s_i 件，每件体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。

输出最大价值。

【输入格式】

第一行两个整数 n,V，用空格隔开，分别表示物品数量和背包容积。

接下来有 n 行，每行两个整数 v_i,w_i,s_i，用空格隔开，分别表示第 i 件物品的体积、价值和数量。

$1 \le n \le 10^3, 1 \le V, v_i, w_i, s_i \le 2 \times 10^4$。

【输出格式】

输出一个整数，表示最大价值。

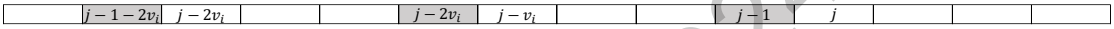
【分析】

倘若使用多重背包问题 II 的解法，将第 i 种物品拆分为独立的 $\log(s)$ 件物品，时间复杂度为 $O(nlog(s)V)$，在本题的数据范围中，计算规模约为 2×10^8，有些无法接受，我们考虑如何优化。

当外层循环进行到 i 时，$f[j]$ 表示从前 i 种物品中选出若干个放入背包，体积之和为 j时，价值之和最大值。倒序循环 j，在状态转移时，考虑选取第 i 个物品的个数为 cnt，则有 $f[j]=\max\limits_{1\le cnt\le s_i}(f[j-cnt\times v_i]+cnt\times w_i)$。

画出能够转移到状态 j 的决策集合 $\{j-cnt\times v_i|1\le cnt\le s_i\}$：

当循环变量 j 减小 1 时：



可以发现，相邻两个状态 j 和 $j-1$ 对应的决策候选集合没有重叠，很难快速地从 $j-1$ 对应的决策集合得到 j 对应的集合。

但是，我们试着考虑一下状态 j 和 $j-v_i$。

这两者对应的决策候选集合之间的关系，与单调队列维护的滑动窗口非常相似，只有一个新决策加入候选集合、一个已有决策被排除。所以，我们应该把状态 j 按照除以 v_i 的余数分组，对每一组分别进行计算，不同组之间的状态在阶段 i 不会相互转移。

余数为 0 —— $0,v_i,2v_i,\cdots$

余数为 1 —— $1,1+v_i,1+2v_i,\cdots$

…

余数为 v_i-1 —— $(v_i-1),(v_i-1)+v_i,(v_i-1)+2v_i,\cdots$

把倒序循环 j 的过程，改为对每个余数 $u\in[0,v_i-1]$，倒序循环 $p=\lfloor(V-u)-v_i\rfloor\sim0$，对应的状态就是 $j=u+p\times v_i$。第 i 种物品只有 s_i 个，故能转移到 $j=u+p\times v_i$ 的决策候选集合就是 $\{u+k\times v_i|p-s_i\le k\le p-1\}$。写出新的状态转移方程：

$$f[u+p\times v_i]=\max_{p-s_i\le k\le p-1}\{f[u+k\times v_i]+(p-k)\times w_i\}$$

把外层循环 i 和 u 看作定值，当内层循环变量 p 减小 1 时，决策 k 的取值范围 $[p-s_i,p-1]$ 的上下界均单调减小。状态转移方程等号右侧的式子仍然分为两部分，仅包含变量 p 的 $p\times w_i$ 和仅包含变量 k 的 $f[u+k\times v_i]-k\times w_i$。综上所述，我们可以建立

一个决策点 k 单调递减，数值 $f[u+k\times v_i]-k\times w_i$ 单调递减的队列，用于维护候选集合。对于每个 p，执行单调队列的三个惯例操作：

1. 检查队头合法性，把大于 $p-1$ 的决策点出队。
2. 取队头为最优决策，更新 $f[u+p\times v_i]$。
3. 把新决策 $k=p-s_i-1$ 插入队尾，入队前检查队尾单调性，排除无用决策。

整个算法的时间复杂度为 $O(nV)$，空间复杂度为 $O(V)$。

```cpp
template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

const int M = int(2e4);
const int inf = 0x3f3f3f3f;

int f[M + 5];
int q[M + 5], l, r;

void work() {
    int n, V; scanf("%d %d", &n, &V);
    memset(f, -inf, sizeof f);
    f[0] = 0;
    for (int i = 1, v, w, s; i <= n; i++) {
        scanf("%d %d %d", &v, &w, &s);
        for (int j = 0; j < v; j++) {
            l = 1, r = 0;
            int mx = (V - j) / v;
            for (int k = mx - 1; k >= max(0, mx - s); k--) {
                while (l <= r && f[j + q[r] * v] - q[r] * w
                    <= f[j + k * v] - k * w) r--;
                q[++r] = k;
            }
            for (int k = mx; k >= 1; k--) {
                if (l <= r && q[l] == k) l++;
                cmax(f[j + k * v], f[j + q[l] * v] + (k - q[l]) * w);
                if (k - s - 1 >= 0) {
                    while (l <= r && f[j + q[r] * v] - q[r] * w
                        <= f[j + (k - s - 1) * v] - (k - s - 1) * w) r--;
                    q[++r] = k - s - 1;
                }
            }
        }
    }
    printf("%d\n", *max_element(f, f + V + 1));
```

```
}
```

例题　Coins（POJ1742）

有 n 种硬币，第 i 种硬币的面值为 $a[i]$，数量为 $c[i]$。问使用这些硬币可以凑出 $1\sim m$ 中的多少种金额。

【输入格式】

输入包含多组测试数据，每组测试数据第一行输入两个整数 n 和 m，第二行输入 n 个整数表示序列 a，第三行输入 n 个整数表示序列 c。

$1\le n\le 10^2, 1\le c_i\le 10^3, 1\le a_i, m\le 10^4$。

【输出格式】

输出一个整数，表示使用这些硬币可以凑出 $1\sim m$ 中的多少种金额。

【分析】

很明显，这是一个多重背包问题，可以使用二进制拆分或者单调队列优化解决。不过，本题关注的是可行性而不是最优性，这是一个特殊之处。

事实上，我们可以使用贪心解决该问题。设 $f[j]$ 表示考虑了前 i 种硬币，是否能够凑出 j，若能凑出 j 则 $f[j]=true$；否则为 $false$。此外，使用数组 $cnt[j]$ 表示在考虑了前 $i-1$ 种硬币之后，凑出 j 使用的第 i 种硬币的最少数量。若 $f[j]=false, f[j-a_i]=true$ 且 $cnt[j-a_i]<c_i$，那么我们就可以使用一枚第 i 种硬币凑出 j，并更新 $cnt[j]$ 为 $cnt[j-a_i]+1$。

```
const int M = int(1e5);

int n, m;
bool f[M + 5];
int cnt[M + 5];
int a[M + 5], c[M + 5];

void work() {
    memset(f, 0, sizeof(bool) * (m + 1));
    for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
    for (int i = 1; i <= n; i++) scanf("%d", &c[i]);
    f[0] = true;
    for (int i = 1; i <= n; i++) {
        memset(cnt, 0, sizeof(int) * (m + 1));
        for (int j = a[i]; j <= m; j++)
            if (!f[j] && f[j - a[i]] && cnt[j - a[i]] < c[i])
                f[j] = true, cnt[j] = cnt[j - a[i]] + 1;
    }
```

```
19.     printf("%d\n", accumulate(f + 1, f + m + 1, 0));
20. }
```

5.3.4 分组背包

例题　　分组背包问题（AcWing9）

有 n 组物品和一个容量是 V 的背包。

每组物品有若干个，同一组内的物品最多只能选一个。

每件物品的体积是 v_{ij}，价值是 w_{ij}，其中 i 是组号，j 是组内编号。

求解将哪些物品装入背包，可使物品总体积不超过背包容量，且总价值最大。

输出最大价值。

【输入格式】

第一行有两个整数 n,V ，用空格隔开，分别表示物品组数和背包容量。

接下来有 n 组数据：

每组数据第一行有一个整数 s_i，表示第 i 个物品组的物品数量；

每组数据接下来有 s_i 行，每行有两个整数 v_{ij},w_{ij}，用空格隔开，分别表示第 i 个物品组的第 j 个物品的体积和价值。

$1 \le n,V,s_i,v_{ij},w_{ij} \le 10^2$。

【输出格式】

输出一个整数，表示最大价值。

【分析】

当第一层循环到 i 时，设状态 $f[j]$ 表示考虑了前 i 组物品，体积之和恰好为 j 的最大价值，对于二层循环的每个 j 再枚举第 i 组的物品 k 对 $f[j]$ 进行更新，如此就可以保证每组至多选择一个物品。

时间复杂度为 $O(nsV)$，空间复杂度为 $O(V)$。

```
1. template <typename T>
2. inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}
3.
4. const int M = int(1e2);
5. const int inf = 0x3f3f3f3f;
6.
7. int f[M + 5];
8. int v[M + 5], w[M + 5];
9.
10. void work() {
11.     int n, V; scanf("%d %d", &n, &V);
```

```
12.     memset(f, -inf, sizeof f);
13.     f[0] = 0;
14.     for (int i = 1, s; i <= n; i++) {
15.         scanf("%d", &s);
16.         for (int j = 1; j <= s; j++) scanf("%d %d", &v[j], &w[j]);
17.         for (int j = V; j >= 1; j--) {
18.             for (int k = 1; k <= s; k++)
19.                 if (j >= v[k]) cmax(f[j], f[j - v[k]] + w[k]);
20.         }
21.     }
22.     printf("%d\n", *max_element(f, f + V + 1));
23. }
```

5.3.5 混合背包

例题　混合背包问题（AcWing7）

有 n 种物品和一个容量是 V 的背包。

物品一共有三类：

1. 第一类物品只能用 1 次；
2. 第二类物品可以用无限次；
3. 第三类物品最多只能用 s_i 次。

每种体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。 输出最大价值。

【输入格式】

第一行两个整数 n,V，用空格隔开，分别表示物品种数和背包容积。

接下来有 n 行，每行三个整数 v_i,w_i,s_i，用空格隔开，分别表示第 i 种物品的体积、价值和数量。

$s_i=-1$ 表示第 i 种物品只能用 1 次；

$s_i=0$ 表示第 i 种物品可以用无限次；

$s_i>0$ 表示第 i 种物品可以使用 si 次。

$1\le n,V,v_i,w_i\le 10^3,-1\le s_i\le 10^3$。

【输出格式】

输出一个整数，表示最大价值。

【分析】

只需要把 0/1 背包、完全背包和多重背包的代码合起来。

时间复杂度为 $O(nV)$，空间复杂度为 $O(V)$。

```cpp
template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

const int M = int(2e4);
const int inf = 0x3f3f3f3f;

int n, V;
int f[M + 5];
int q[M + 5], l, r;

void zeroOne(int v, int w) {
    for (int j = V - v; j >= 0; j--) if (f[j] >= 0) cmax(f[j + v], f[j] + w);
}

void complete(int v, int w) {
    for (int j = 0; j <= V - v; j++) if (f[j] >= 0) cmax(f[j + v], f[j] + w);
}

void multiple(int v, int w, int s) {
    for (int j = 0; j < v; j++) {
        l = 1, r = 0;
        int mx = (V - j) / v;
        for (int k = mx - 1; k >= max(0, mx - s); k--) {
            while (l <= r && f[j + q[r] * v] - q[r] * w
                <= f[j + k * v] - k * w) r--;
            q[++r] = k;
        }
        for (int k = mx; k >= 1; k--) {
            if (l <= r && q[l] == k) l++;
            cmax(f[j + k * v], f[j + q[l] * v] + (k - q[l]) * w);
            if (k - s - 1 >= 0) {
                while (l <= r && f[j + q[r] * v] - q[r] * w
                    <= f[j + (k - s - 1) * v] - (k - s - 1) * w) r--;
                q[++r] = k - s - 1;
            }
        }
    }
}

void work() {
    scanf("%d %d", &n, &V);
    memset(f, -inf, sizeof f);
    f[0] = 0;
```

```
    for (int i = 1, v, w, s; i <= n; i++) {
        scanf("%d %d %d", &v, &w, &s);
        if (s == -1) zeroOne(v, w);
        else if (s == 0) complete(v, w);
        else multiple(v, w, s);
    }
    printf("%d\n", *max_element(f, f + V + 1));
}
```

例题　I love exam（HDU 6968）

小 Z 有 n 门课需要考试，但他由于没上课所以每门课的初试成绩为 0。不过幸好他有 m 个复习材料，第 i 个复习材料最多使用一次，需要花费 y 的时间，可以使名为 s 的课程分数提高 x。单门课程满分为 100 分，60 分以下是不及格。小 Z 想知道在至多 t 的时间内且至多有 p 门课程不及格的条件下，自己所有课程的得分之和最大是多少。

【输入格式】

第一行输入一个整数 T 表示测试数据个数。

每组测试数据第一行输入一个整数 n 表示课程数量，第二行输入 n 个字符串 s 表示课程名，第三行输入一个整数 m 表示复习材料个数，接下来 m 行，每行输入课程名 s、提高的分数 x 和花费的时间 y，接下来一行输入两个整数 t 和 p，分别表示最多的复习时间和最多能接受的不及格课程数量。

$1 \le T \le 10, 1 \le n \le 50, 1 \le m \le 15000, 1 \le x, y \le 10, 1 \le t \le 500, 0 \le p \le 3, 1 \le |s| \le 15$。

【输出格式】

对于每组数据，输出一个整数，在至多 t 的时间内且至多有 p 门课程不及格的条件下，自己所有课程的得分之和最大值。

【分析】

解法一

设状态 $f[i][j]$ 表示第 i 门课程取得 j 分所需要的最少时间，可以用 0/1 背包在 $O(100m)$ 内处理出来。

设状态 $g[i][j][k]$ 表示考虑了前 i 门课程，花费了 j 时间，有 k 门课程不及格的得分之和最大值。可以把得分看作物品价值，得分所需的时间 f 看作物品体积，做一遍分组背包即可计算出数组 g。时间复杂度为 $O(100ntp)$。

综上所述，总的时间复杂度为 $O\big(T(100m + 100ntp)\big)$，在本题中计算规模约为 9×10^7，

空间复杂度为 $O(100n + tp)$。

```cpp
template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

template <typename T>
inline bool cmin(T &a, const T &b) {return a > b ? a = b, true : false;}

const int M = 15000;
const int N = 50;
const int O = 500;
const int inf = 0x3f3f3f3f;

int n, m, t, p;
map<string, int> id;
int c[M + 5], x[M + 5], y[M + 5];
int f[N + 5][105];
int g[2][O + 5][4];

void zeroOne() {
    memset(f, inf, sizeof f);
    for (int i = 1; i <= n; i++) f[i][0] = 0;
    for (int i = 1; i <= m; i++)
        for (int j = 100; j >= 0; j--)
            cmin(f[c[i]][min(100, j + x[i])], f[c[i]][j] + y[i]);
}

void group() {
    memset(g, -inf, sizeof g);
    g[0][0][0] = 0;
    for (int i = 1, u; i <= n; i++) {
        u = i & 1;
        memset(g[u], -inf, sizeof(g[u]));
        for (int j = t; j >= 0; j--) {
            for (int k = 0; k <= p; k++) {
                for (int l = 0; l <= 100; l++) if (j >= f[i][l]) {
                    if (l < 60 && k > 0)
                        cmax(g[u][j][k], g[u ^ 1][j - f[i][l]][k - 1] + l);
                    if (l >= 60) cmax(g[u][j][k], g[u ^ 1][j - f[i][l]][k] + l);
                }
            }
        }
    }
}

```

```
void work() {
    id.clear();
    cin >> n;
    string name;
    for (int i = 1; i <= n; i++) {
        cin >> name;
        id[name] = i;
    }
    cin >> m;
    for (int i = 1; i <= m; i++) {
        cin >> name >> x[i] >> y[i];
        c[i] = id[name];
    }
    cin >> t >> p;
    zeroOne();
    group();
    int mx = -1;
    for (int i = 0; i <= t; i++)
        for (int j = 0; j <= p; j++) mx = max(mx, g[n & 1][i][j]);
    cout << mx << "\n";
}
```

解法二

设状态 $f[i][j]$ 表示第 i 门课程花费 j 的时间可以获得的最大分数。注意到 $1 \le x,y \le 10$，因此每门课程最多有 100 种复习材料。因此我们可以按照复习材料的（课程、提升分数、花费时间）分类，计算每类的个数。于是可以用单调队列优化的多重背包求解数组 f，时间复杂度为 $O(10^4 n)$。

设状态 $g[i][j][k]$ 表示考虑了前 i 门课程，花费了 j 时间，有 k 门课程不及格的得分之和最大值。可以把得分 f 看作物品价值，得分所需的时间看作物品体积，做一遍分组背包即可计算出数组 g。时间复杂度为 $O(100ntp)$。

综上所述，总的时间复杂度为 $O\big(T(10^4 n + 100ntp)\big)$，在本题中计算规模约为 8×10^7，空间复杂度为 $O(100n + tp)$。

```
template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

template <typename T>
inline bool cmin(T &a, const T &b) {return a > b ? a = b, true : false;}

const int M = 15000;
const int N = 50;
```

```
const int O = 500;
const int inf = 0x3f3f3f3f;

int n, m, t, p;
map<string, int> id;
int c[N + 5][11][11];
int f[N + 5][105];
int q[105];
int g[2][O + 5][4];

void multiple() {
    auto solve = [&](int *f, int V, int v, int w, int s) {
        for (int j = 0; j < v; j++) {
            int l = 1, r = 0, mx = (V - j) / v;
            for (int k = mx - 1; k >= max(0, mx - s); k--) {
                while (l <= r && f[j + q[r] * v] - q[r] * w
                    <= f[j + k * v] - k * w) r--;
                q[++r] = k;
            }
            for (int k = mx; k >= 1; k--) {
                if (l <= r && q[l] == k) l++;
                cmax(f[j + k * v], f[j + q[l] * v] + (k - q[l]) * w);
                if (k - s - 1 >= 0) {
                    while (l <= r && f[j + q[r] * v] - q[r] * w
                        <= f[j + (k - s - 1) * v] - (k - s - 1) * w) r--;
                    q[++r] = k - s - 1;
                }
            }
        }
    };

    memset(f, -inf, sizeof f);
    for (int i = 1; i <= n; i++) {
        f[i][0] = 0;
        for (int x = 1; x <= 10; x++) {
            for (int y = 1; y <= 10; y++) if (c[i][x][y]) {
                solve(f[i], 100, y, x, c[i][x][y]);
            }
        }
        for (int j = 0; j <= 100; j++) cmin(f[i][j], 100);
    }
}

void group() {
```

```
53.     memset(g, -inf, sizeof g);
54.     g[0][0][0] = 0;
55.     for (int i = 1, u; i <= n; i++) {
56.         u = i & 1;
57.         memset(g[u], -inf, sizeof(g[u]));
58.         for (int j = t; j >= 0; j--) {
59.             for (int k = 0; k <= p; k++) {
60.                 for (int l = 0; l <= j; l++) {
61.                     if (k > 0 && f[i][l] < 60)
62.                         cmax(g[u][j][k], g[u ^ 1][j - l][k - 1] + f[i][l]);
63.                     if (f[i][l] >= 60)
64.                         cmax(g[u][j][k], g[u ^ 1][j - l][k] + f[i][l]);
65.                 }
66.             }
67.         }
68.     }
69. }
70.
71. void work() {
72.     id.clear();
73.     memset(c, 0, sizeof c);
74.     cin >> n;
75.     string name;
76.     for (int i = 1; i <= n; i++) {
77.         cin >> name;
78.         id[name] = i;
79.     }
80.     cin >> m;
81.     for (int i = 1, x, y; i <= m; i++) {
82.         cin >> name >> x >> y;
83.         c[id[name]][x][y]++;
84.     }
85.     cin >> t >> p;
86.     multiple();
87.     group();
88.     int mx = -1;
89.     for (int i = 0; i <= t; i++)
90.         for (int j = 0; j <= p; j++) mx = max(mx, g[n & 1][i][j]);
91.     cout << mx << "\n";
92. }
```


5.3.6 多维背包

例题　二维费用的背包问题（AcWing8）

有 n 件物品和一个容量是 V 的背包，背包能承受的最大重量是 M。

每件物品只能用一次。体积是 v_i，重量是 m_i，价值是 w_i。

求解将哪些物品装入背包，可使物品总体积不超过背包容量，总重量不超过背包可承受的最大重量，且价值总和最大。

输出最大价值。

【输入格式】

第一行三个整数，n,V,M，用空格隔开，分别表示物品件数、背包容积和背包可承受的最大重量。

接下来有 n 行，每行三个整数 v_i,m_i,w_i，用空格隔开，分别表示第 i 件物品的体积、重量和价值。

【输出格式】

输出一个整数，表示最大价值。

【分析】

本题相较于普通的 0/1 背包问题增加重量这一维，但本质上仍然是 0/1 背包问题，我们可以使用相似的方法求解。

设 $f[j][k]$ 表示考虑了前 i 件物品，使用了恰好 j 的体积和恰好 k 的重量，所能得到的最大价值，则有状态转移方程 $f[j][k]=f[j-v][k-m]+w$。

时间复杂度为 $O(nVM)$，空间复杂度为 $O(VM)$。

```
template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

const int M = int(1e2);
const int inf = 0x3f3f3f3f;

int f[M + 5][M + 5];

void work() {
    int n, V, M; scanf("%d %d %d", &n, &V, &M);
    memset(f, -inf, sizeof f);
    f[0][0] = 0;
    for (int i = 1, v, m, w; i <= n; i++) {
        scanf("%d %d %d", &v, &m, &w);
        for (int j = V - v; j >= 0; j--)
            for (int k = M - m; k >= 0; k--)
                if (f[j][k] >= 0) cmax(f[j + v][k + m], f[j][k] + w);
    }
    int mx = -inf;
```

```
20.     for (int i = 0; i <= V; i++) cmax(mx, *max_element(f[i], f[i] + M + 1));
21.     printf("%d\n", mx);
22. }
```

例题　Jury Compromise（POJ1015）

在一个遥远的国家，一名嫌疑犯是否有罪需要由陪审团来决定。

陪审团是由法官从公民中挑选的。

法官先随机挑选 n 个人（编号 $1,2,\cdots,n$）作为陪审团的候选人，然后再从这 n 个人中按照下列方法选出 n 人组成陪审团。

首先，参与诉讼的控方和辩方会给所有候选人打分，分值在 0 到 20 之间。

第 i 个人的得分分别记为 $p[i]$ 和 $d[i]$。

为了公平起见，法官选出的 m 个人必须满足：辩方总分 D 和控方总分 P 的差的绝对值 $|D-P|$ 最小。

如果选择方法不唯一，那么再从中选择辨控双方总分之和 $D+P$ 最大的方案。

求最终的陪审团获得的辩方总分 D、控方总分 P，以及陪审团人选的编号。

若陪审团的人选方案不唯一，则任意输出一组合法方案即可。

【输入格式】

输入包含多组测试数据。

每组测试数据第一行包含两个整数 n 和 m。

接下来 n 行，每行包含两个整数 $p[i]$ 和 $d[i]$。

每组测试数据之间隔一个空行。

当输入数据 $n=0,m=0$ 时，表示结束输入，该数据无需处理。

$1\le n\le 200,1\le m\le 20,0\le p[i],d[i]\le 20$。

【输出格式】

对于每组数据，第一行输出 Jury #C，C 为数据编号，从 1 开始。

第二行输出 Best jury has value P for prosecution and value D for defence:，P 为控方总分，D 为辩方总分。

第三行输出按升序排列的陪审人选编号，每个编号前输出一个空格。

每组数据输出完后，输出一个空行。

【分析】

很明显，这是一个二维的 0/1 背包问题。设状态 $f[i][j][k]$ 表示考虑了前 i 个人，选择了 j 个人，$P-D=k$ 的 $P+D$ 的最大值。有状态转移方程 $f[i][j][k]=\max(f[i-$

$1][j][k], f[i-1][j-1]\big[k-p[i]+d[i]\big]+p[i]+d[i])$。初始值 $f[0][0][0]=0$。由于 $P-D$ 涉及到负数，因此我们可以给第三维 k 统一加一个偏移量将其映射到非负整数。

此外，题目还要求输出方案。当我们求解出最优状态 $f[i][j][k]$ 后，只需要比较 $f[i-1][j][k]$ 与其是否相等，若相等则说明第 i 个不选并更新状态为 $f[i-1][j][k]$，否则选择第 i 个人并更新状态为 $f[i-1][j-1][k-p[i]+d[i]]$。

时间复杂度为 $O(nm(P-D))$。

```
template <typename T>
inline bool cmax(T &a, const T &b) {return a < b ? a = b, true : false;}

const int M = int(2e2);
const int N = 20;
const int O = N * N;
const int inf = 0x3f3f3f3f;

int n, m, ca;
int p[M + 5], d[M + 5];
int f[M + 5][N + 5][O * 2 + 5];

void work() {
    for (int i = 1; i <= n; i++) scanf("%d %d", &p[i], &d[i]);
    memset(f, -inf, sizeof f);
    f[0][0][O] = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= min(i, m); j++) {
            for (int k = 0; k <= O * 2; k++) {
                f[i][j][k] = f[i - 1][j][k];
                if (j > 0 && k - p[i] + d[i] >= 0 && k - p[i] + d[i] <= 2 * O)
                    cmax(f[i][j][k], f[i - 1][j - 1][k - p[i] + d[i]] + p[i] + d[i]);
            }
        }
    }
    int mx = -inf, diff = -1;
    for (int i = 0; mx < 0; i++) {
        if (cmax(mx, f[n][m][O - i])) diff = O - i;
        if (cmax(mx, f[n][m][O + i])) diff = O + i;
    }
    vector<int> v;
    int sp = 0, sd = 0;
    int i = n, j = m, k = diff;
    while (j) {
        if (f[i][j][k] != f[i - 1][j][k])
```

```
36.             j--,
37.             k -= p[i] - d[i],
38.             sp += p[i],
39.             sd += d[i],
40.             v.push_back(i);
41.         --i;
42.     }
43.     printf("Jury #%d\n", ++ca);
44.     printf("Best jury has value %d for prosecution and value %d for defence:\n", sp,
sd);
45.     for (int i = int(v.size()) - 1; i >= 0; i--) printf(" %d", v[i]);
46.     printf("\n\n");
47. }
```