第5章 基本算法

基本算法没有复杂的逻辑和步骤，但是计算效率高、应用场景丰富，是算法竞赛必考的知识点。

在算法竞赛中有一些“通用”的算法，例如尺取法、二分法、倍增法、前缀和、差分、离散化、分治、贪心等。这些算法没有复杂的逻辑，代码也简短，但是它们的应用场合多，效率也高，广泛应用在编程和竞赛中。在蓝桥杯大赛中，这些算法几乎是必考的。本章介绍前缀和、差分、二分、贪心。

5.1 算法与算法复杂度

在前面几章讲解例题时有很多题分析了“算法复杂度”，使读者对算法分析有了一定的了解。进行算法分析是做竞赛题时的一个必要步骤，用于评估问题的难度和决定使用什么算法来求解。在蓝桥杯这种赛制中，一道题往往可以用多种算法求解，例如较差算法可以通过30%的测试，中等算法可以通过70%的测试，优秀算法可以通过100%的测试。

本节回答两个基本问题：什么是算法？如何评估算法？

5.1.1 算法的概念

“程序＝算法＋数据结构”。算法是解决问题的逻辑、方法、过程，数据结构是数据在计算机中的存储和访问方式，两者紧密结合解决复杂问题。

算法(Algorithm)是对特定问题求解步骤的一种描述，是指令的有限序列。它有5个特征：

(1) 输入。一个算法有零个或多个输入。程序可以没有输入，例如一个定时闹钟程序，它不需要输入，但是能够每隔一段时间就输出一个报警。

(2) 输出。一个算法有一个或多个输出。程序可以没有输入，但是一定要有输出。

(3) 有穷性。一个算法必须在执行有穷步之后结束，且每一步都在有穷时间内完成。

(4) 确定性。算法中的每一条指令必须有确切的含义，对于相同的输入只能得到相同的输出。

(5) 可行性。算法描述的操作可以通过已经实现的基本操作执行有限次来实现。

5.1.2 计算资源

计算机软件运行需要计算时间和存储空间两种资源。资源是有限的，一个算法对这两种资源的使用程度可以用来衡量该算法的优劣。

时间复杂度：代码运行需要的时间。

空间复杂度：代码运行需要的存储空间。

与这两个复杂度对应，程序设计题会给出对运行时间和空间的说明，例如“时间限制：1s，内存限制：256MB”，参赛队员提交到判题系统的代码需要在1s内运行结束，且使用的空间不能超过256MB。若有一个不满足，就判错。

这两个限制条件非常重要，是检验代码性能的参数，所以队员拿到题目后第一步需要分析代码运行需要的计算时间和存储空间。

如何衡量代码运行的时间？通过代码打印运行时间可以得到一个直观的认识。

下面的Python代码只有一个for语句，代码对k累加n次，最后打印运行时间。

```
from time import *
k = 0
n = 10**7                    # 一千万
s = process_time()           # 开始时间
for i in range(n): k += 5    # 循环 n 次
t = process_time()           # 终止时间
print(t - s)                 # 打印运行时间
```

在作者的笔记本式计算机上运行，循环 n＝一千万次，用时 1.29s。换了一台台式计算机，用时 0.234375s。Python 的 for 循环很耗时，比 C++的循环慢很多。Python 的编程题一般给 10s 的测试时间，此时代码只能做千万次左右的循环。

竞赛评测用的判题服务器（OJ 系统），性能可能比这个好一些，也可能差不多。对于 C++题目，如果题目要求“时间限制：1s”，那么内部的循环次数 n 应该在一亿次以内，Java 也在一亿次以内。对于同等规模的 Python 题目，时间限制一般是 5～10s。

由于代码的运行时间依赖于计算机的性能，不同的计算机结果不同，所以直接把运行时间作为判断标准并不准确，用代码执行的“计算次数”来衡量更加合理。例如上述代码循环了 n 次，把它的运行次数记为 O(n)，O 称为计算复杂度的“大 O 记号”。

5.1.3 算法复杂度

衡量算法性能的主要标准是时间复杂度。

时间复杂度比空间复杂度更重要。一个算法使用的空间很容易分析，而时间复杂度往往关系到算法的根本逻辑，更能说明一个程序的优劣。如果不特别说明，提到“复杂度”时一般指时间复杂度。

对于竞赛题，一般情况下做简单的时间复杂度分析即可，不用精确计算，常用“大 O 记号”做估计。

例如，在一个有 n 个数的无序数列中查找某个数 x，可能第一个数就是 x，也可能最后一个数才是 x，平均查找时间是 n/2 次，最差情况需要查找 n 次；把查找的时间复杂度记为最差情况下的 O(n)，而不是 O(n/2)。按最差情况，是因为判题系统可能故意用“很差”的数据来测试。再例如，冒泡排序算法的计算次数约等于 $n^2/2$ 次，但是仍记为 $O(n^2)$，而不是 $O(n^2/2)$，这里把 1/2 系数忽略了。

即使是同样的算法，不同的人写出的代码的效率也不一样。OJ 系统所判定的运行时间是整个代码运行所花的时间，而不是理论上算法所需要的时间。同一个算法，不同的人写出的程序，复杂度和运行时间可能差别很大，跟他使用的编程语言、逻辑结构、库函数等都有关系。所以竞赛队员需要进行训练，提高自己的编码能力，纠正自己不合理的写代码的习惯。

用“大 O 记号”表示的算法复杂度有以下分类。

(1) O(1)，常数时间。计算时间是一个常数，和问题的规模 n 无关。例如，用公式计算时，一次计算的复杂度就是 O(1)；哈希算法，用 hash 函数在常数时间内计算出存储位置；在矩阵 A[M][N]中查找第 i 行 j 列的元素，只需要访问一次 A[i][j]就够了。

(2) $O(\log_2 n)$，对数时间。计算时间是对数，通常是以 2 为底的对数，每一步计算后，问题的规模减小一半。例如在一个长度为 n 的有序数列中查找某个数，用折半查找的方法，只需要 $\log_2 n$ 次就能找到。$O(\log_2 n)$和 O(1)没有太大差别。例如 $n=10^7$ 时，$\log_2 n<24$。

(3) O(n),线性时间。计算时间随规模 n 线性增长。在很多情况下,这是算法可能达到的最优复杂度,因为对输入的 n 个数,程序一般需要处理所有的数,即计算 n 次。例如查找一个无序数列中的某个数,可能需要检查所有的数。再例如图问题,有 V 个点和 E 条边,大多数图的问题,都需要搜索所有的点和边,复杂度的最优上限是 O(V+E)。

(4) $O(n\log_2 n)$。这常是算法能达到的最优复杂度。例如分治法,一共 $O(\log_2 n)$个步骤,每个步骤对每个数操作一次,所以总复杂度是 $O(n\log_2 n)$。例如 $n=10^6$ 时,$n\log_2 n=2\times10^7$。

(5) $O(n^2)$。一个两重循环的算法,复杂度是 $O(n^2)$。类似的复杂度有 $O(n^3)$、$O(n^4)$等。

(6) $O(2^n)$。一般对应集合问题,例如一个集合中有 n 个数,这些数不分先后,子集共有 2^n 个。

(7) O(n!)。一般对应排列问题。如果集合中的数分先后,按顺序输出所有的排列,共有 O(n!)个。

把上面的复杂度分成两类:多项式复杂度,包括 O(1)、O(n)、$O(n\log_2 n)$、$O(n^k)$等,其中 k 是一个常数;指数复杂度,包括 $O(2^n)$、O(n!)等。

如果一个算法是多项式复杂度,称它为“高效”算法;如果是指数复杂度,则称它为“低效”算法。

对于竞赛题目的限制时间,C/C++一般为 1s,Java 一般为 3s,Python 一般为 5～10s。对应普通计算机的计算速度是几千万次每秒。那么,上述的时间复杂度可以换算出能解决问题的数据规模。例如,如果一个算法的复杂度是 O(n!),当 n=11 时,11!=39916800,这个算法只能解决 n≤11 的问题。问题规模和可用算法如表 5.1 所示。

表 5.1 问题规模和可用算法

问题规模 n	可用算法的时间复杂度					
	$O(\log_2 n)$	O(n)	$O(n\log_2 n)$	$O(n^2)$	$O(2^n)$	O(n!)
$n\leqslant11$	√	√	√	√	√	√
$n\leqslant25$	√	√	√	√	√	×
$n\leqslant5000$	√	√	√	√	×	×
$n\leqslant10^6$	√	√	√	×	×	×
$n\leqslant10^7$	√	√	×	×	×	×
$n<10^8$	√	×	×	×	×	×

在拿到题目后,大家一定要分析自己准备使用的算法的复杂度,评估自己的代码能通过多少测试。

5.2 前 缀 和

扫一扫 视频讲解

5.2.1 前缀和的概念

前缀和是一种操作简单但是非常有效的优化方法,能把计算复杂度为 O(n)的区间计

算优化为O(1)的端点计算。

前缀和是出题者喜欢考核的知识点，在算法竞赛中很常见，在蓝桥杯大赛中几乎必考。其原因有以下两点：

(1) 原理简单，方便在很多场景下应用，与其他考点结合。

(2) 可以考核不同层次的能力。前缀和的题目一般也能用暴力法求解，暴力法能通过30%的测试，用前缀和优化后能通过70%～100%的测试。

首先了解“前缀和”的概念。一个长度为n的数组a[1]～a[n]，前缀和sum[i]等于a[1]～a[i]的和：

$$sum[i]=a[1]+a[2]+\cdots+a[i]$$

使用递推，可以在O(n)时间内求得所有前缀和：

$$sum[i]=sum[i-1]+a[i]$$

如果预计算出前缀和，就能使用它快速计算出数组中任意一个区间a[i]～a[j]的和，即

$$a[i]+a[i+1]+\cdots+a[j-1]+a[j]=sum[j]-sum[i-1]$$

上式说明，复杂度为O(n)的区间求和计算优化为O(1)的前缀和计算。

5.2.2 例题

前缀和是一种很简单的优化技巧，应用场合很多，在竞赛中极为常见。如果大家对题目建模时发现有区间求和的操作，可以考虑使用前缀和优化。

第1章曾用例题“求和”介绍了前缀和的应用，下面再举几个例子。

例5.1 可获得的最小取值 lanqiaoOJ 3142

问题描述：有一个长度为n的数组a，需要进行k次操作来取出数组中的元素。每次操作必须选择以下两种操作之一：(1)取出数组中的最大元素；(2)取出数组中的最小元素和次小元素。要求在进行完k次操作后取出的数的和最小。

输入：第一行输入两个整数n和k，表示数组长度和操作次数。第二行输入n个整数，表示数组a。

数据范围：$3\leqslant n\leqslant 2\times 10^5$，$1\leqslant a_i\leqslant 10^9$，$1\leqslant k\leqslant 99999$，$2k<n$。

输出：输出一个整数，表示最小的和。

输入样例：	输出样例：
5 1 2 5 1 10 6	3

第一步肯定是排序，例如从小到大排序，然后再进行两种操作。操作(1)在a[]的尾部选一个数，操作(2)在a[]的头部选两个数。

大家容易想到一种简单方法：每次在操作(1)和操作(2)中选较小的值。这是贪心法的思路。用贪心法对吗？分析之后发现用贪心法是错误的，例如{3，1，1，1，1，1，1}，做k=3次操作，如果每次都按贪心法，就是做3次操作(2)，结果是6，但是正确答案是做3次操作(1)，结果是5。

重新考虑所有可能的情况。设操作(2)做 p 次,操作(1)做 k－p 次,求和:

$$\sum_{i=1}^{2p} a_i + \sum_{i=n+p-k+1}^{n} a_i$$

为了找最小的和,需要把所有的 p 都试一遍。如果直接按上面的公式计算,那么验证一个 p 的计算量是 O(n),验证所有的 p,1≤p≤k,总计算量为 O(kn),超时。

大家容易发现公式的两个部分就是前缀和,分别等于 sum[2p]、sum[n]－sum[n+p－k]。如果提前算出前缀和 sum[],那么验证一个 p 的计算量是 O(1),验证所有 p 的总计算量是 O(n)。

下面是 Python 代码。对于代码的计算复杂度,第 3 行 sort()是 $O(n\log_2 n)$,第 5、7 行的 for 循环都是 O(n),总复杂度为 $O(n\log_2 n)$。

```
1  n, k = map(int, input().split())
2  b = list(map(int, input().split()))
3  a = [0] + sorted(b)              # a[0]不用,从 a[1]开始
4  s = [0] * (n+1)                  # 前缀和
5  for i in range(1, n+1): s[i] = s[i-1] + a[i]
6  ans = 10**18
7  for p in range(1, k+1):
8      ans = min(s[n] - s[n+p-k] + s[2*p], ans)
9  print(ans)
```

再看一道简单题。

例 5.2　并行处理 http://oj.ecustacm.cn/problem.php?id=1811

问题描述:现在有 n 个任务需要在 GPU 上运行,但是只有两块 GPU,每块 GPU 一次只能运行一个任务,两块 GPU 可以并行处理。已知 n 个任务需要的时间,需要选择一个数字 i,将任务 1 到任务 i 放到第一块 GPU 上运行,将任务 i+1 到任务 n 放到第二块 GPU 上运行。请求出最短运行时间。

输入:输入的第一行为正整数 n,1≤n≤100000。第二行为 n 个整数 A_i,表示第 i 个任务需要的时间,$1\leqslant A_i\leqslant 10^9$。

输出:输出一个数字,表示答案。

输入样例:	输出样例:
3 4 2 3	5

题目的意思是把 n 个数划分为左右两部分,分别求和,其中一个较大、一个较小,问在所有可能的划分情况中较大的和最小的是多少?因为 n 比较大,需要一个复杂度约为 O(n)的算法。

这是一道很直接的前缀和题目。用前缀和在 O(n)的时间内预计算出所有的前缀和 sum[]。若在第 i 个位置划分,左部分的和是 sum[i],右部分的和是 sum[n]－sum[i]。在所有的划分中,较大的和的最小值是 min(ans, max(sum[i], sum[n]－sum[i]))。

```
1  #pypy
2  n = int(input())
3  a = list(map(int, input().split()))
4  sum = [0] * (n + 1)                    # 前缀和
5  for i in range(1, n + 1):
```

```
    sum[i] = sum[i - 1] + a[i - 1]      # 求前缀和
ans = sum[n]
for i in range(1, n + 1):               # 较大的和最小是多少
    ans = min(ans, max(sum[i], sum[n] - sum[i]))
print(ans)
```

下面的例题是前缀和在异或计算中的应用，这也是其常见的应用场景。

例 5.3　2023 年第十四届蓝桥杯省赛 C/C++大学 A 组试题 H：异或和之和 lanqiaoOJ 3507

时间限制：1s 内存限制：256MB 本题总分：20 分

问题描述：给定一个数组 A_i，分别求其每个子段的异或和，并求出它们的和。或者说，对于每组满足 $1 \leqslant L \leqslant R \leqslant n$ 的 L、R，求出数组中第 L 至第 R 个元素的异或和。然后输出每组 L、R 得到的结果加起来的值。

输入：输入的第一行包含一个整数 n。第二行包含 n 个整数 A_i，相邻整数之间使用一个空格分隔。

输出：输出一个整数，表示答案。

输入样例：	输出样例：
5 1 2 3 4 5	39

评测用例规模与约定：对于 30%的评测用例，$n \leqslant 300$；对于 60%的评测用例，$n \leqslant 5000$；对于所有评测用例，$1 \leqslant n \leqslant 10^5$，$0 \leqslant A_i \leqslant 2^{20}$。

$a_1 \sim a_n$ 的异或和是指 $a_1 \oplus a_2 \oplus \cdots \oplus a_n$。

下面给出 3 种方法，分别通过 30%、60%、100%的测试。

（1）通过 30%的测试。

本题的简单做法是直接按题意计算所有子段的异或和，然后加起来。

那么有多少个子段？

长度为 1 的子段异或和有 n 个：$a_1, a_2, \cdots, a_n$

长度为 2 的子段异或和有 n−1 个：$a_1 \oplus a_2, a_2 \oplus a_3, \cdots, a_{n-1} \oplus a_n$

……

长度为 n 的子段异或和有一个：$a_1 \oplus a_2 \oplus a_3 \oplus \cdots \oplus a_{n-1} \oplus a_n$

共 $n^2/2$ 个子段。

下面 Python 代码的第 4、5 行遍历所有的子段[L，R]，第 7 行求[L，R]的子段和，共 3 重 for 循环，计算复杂度为 $O(n^3)$，只能通过 30%的测试。

```
n = int(input())
a = list(map(int, input().split()))
ans = 0
for L in range(n):                          # 遍历所有区间[L,R]
    for R in range(L, n):
        sum = 0
        for i in range(L, R + 1):           # 子段和
            sum ^= a[i]
```

```
9           ans += sum                          # 累加所有子段和
10  print(ans)
```

(2) 通过 60%的测试。

本题可以用前缀和优化。

记异或和 $a_1 \oplus a_2 \oplus \cdots \oplus a_i$ 为

$$s_i = a_1 \oplus a_2 \oplus \cdots \oplus a_i$$

这里 s_i 是异或形式的前缀和，这样就把复杂度为 O(n)的子段异或和计算 $a_1 \oplus a_2 \oplus \cdots \oplus a_i$ 优化为 O(1)的求 s_i 的计算。

以包含 a_1 的子段为例，这些子段的异或和相加，等于

$$a_1 + a_1 \oplus a_2 + \cdots + a_1 \oplus \cdots \oplus a_i + \cdots + a_1 \oplus \cdots \oplus a_n = s_1 + s_2 + \cdots + s_i + \cdots + s_n$$

前缀和的计算用递推得到。普通前缀和的递推公式为 s[i]=s[i−1]+a[i]，异或形式的前缀和的递推公式为 s[i]=s[i−1] ^ a[i]，下面代码中用这个公式的简化形式求解了前缀和。

Python 代码的计算复杂度是多少？第 4 行和第 6 行用两重循环遍历所有的子段，同时计算前缀和，计算复杂度是 $O(n^2)$，可以通过 60%的测试。

```
1   n = int(input())
2   a = list(map(int, input().split()))
3   ans = 0
4   for L in range(n):
5       sum = 0                             # sum 是包含 a[L]的子段的前缀和
6       for R in range(L, n):
7           sum ^= a[R]                     # 用递推求前缀和 sum
8           ans += sum                      # 累加所有子段和
9   print(ans)
```

(3) 通过 100%的测试。

本题有没有进一步的优化方法？这就需要仔细地分析异或的性质了。根据异或的定义有 a⊕a=0、0⊕a=a、0⊕0=0。推导子段 $a_i \sim a_j$ 的异或和：

$$a_i \oplus a_{i+1} \oplus \cdots \oplus a_{j-1} \oplus a_j = (a_1 \oplus a_2 \oplus \cdots \oplus a_{i-1}) \oplus (a_1 \oplus a_2 \oplus \cdots \oplus a_j)$$

记 $s_i = a_1 \oplus a_2 \oplus \cdots \oplus a_i$，这是异或形式的前缀和。上式转化为

$$a_i \oplus a_{i+1} \oplus \cdots \oplus a_{j-1} \oplus a_j = s_{i-1} \oplus s_j$$

若 $s_{i-1} = s_j$，则 $s_{i-1} \oplus s_j = 0$；若 $s_{i-1} \neq s_j$，则 $s_{i-1} \oplus s_j = 1$。题目要求所有子段异或和相加的结果，这等于判断所有的$\{s_i, s_j\}$组合，若 $s_i \neq s_j$，则结果加 1。

如何判断两个 s 是否相等？可以用位操作的技巧，如果它们的第 k 位不同，则两个 s 肯定不等。下面以 $a_1=011$，$a_2=010$ 为例，分别计算第 k 位的异或和，并且相加：

k=0，第 0 位异或和，$s_1=1$，$s_2=1 \oplus 0=1$，$ans_0 = a_1 + a_2 + a_1 \oplus a_2 = s_1 + s_1 \oplus s_2 + s_2 = 1+0+1=2$。

k=1，第 1 位异或和，$s_1=1$，$s_2=1 \oplus 1=0$，$ans_1 = a_1 + a_2 + a_1 \oplus a_2 = s_1 + s_1 \oplus s_2 + s_2 = 1+1+0=2$。

k=2，第 2 位异或和，$s_1=0$，$s_2=0 \oplus 0=0$，$ans_2 = a_1 + a_2 + a_1 \oplus a_2 = s_1 + s_1 \oplus s_2 + s_2 = 0+0+0=0$。

最后计算答案：$ans = ans_0 \times 2^0 + ans_1 \times 2^1 + ans_2 \times 2^2 = 6$。

本题 $0 \leqslant A_i \leqslant 2^{20}$，所有的前缀和 s 都不超过 20 位。下面代码的第 4 行逐个计算 20 位的每一位，第 7 行的 for 循环计算 n 个前缀和，总计算量约为 $20 \times n$。

```
1  n = int(input())
2  a = [int(x) for x in input().split()]
3  ans = 0
4  for k in range(21):                  # 所有 a 都不超过 20 位
5      zero, one = 1, 0                 # 统计第 k 位的 0 和 1 的数量
6      cnt, sum = 0, 0                  # cnt 用于统计第 k 位有多少对 s_i⊕s_j = 1
7      for i in range(n):
8          v = (a[i] >> k) & 1          # 取 a[i]的第 k 位
9          sum ^= v                     # 对所有 a[i]的第 k 位做异或得到 sum，sum 等于 0 或者 1
10         if sum == 0:                 # 前缀和为 0
11             zero += 1                # 0 的数量加 1
12             cnt += one               # 这次 sum = 0，这个 sum 跟前面等于 1 的 sum 异或得 1
13         else:                        # 前缀异或为 1
14             one += 1                 # 1 的数量加 1
15             cnt += zero              # 这次 sum = 1，这个 sum 跟前面等于 0 的 sum 异或得 1
16     ans += cnt * (1 << k)            # 第 k 位的异或和相加
17 print(ans)
```

前面的例子都是一维数组上的前缀和，下面介绍二维数组上的前缀和。

例 5.4　领地选择 https://www.luogu.com.cn/problem/P2004

问题描述：作为在虚拟世界里统率千军万马的领袖，小 Z 认为天时、地利、人和三者是缺一不可的，所以谨慎地选择首都的位置对于小 Z 来说是非常重要的。首都被认为是一个占地 $c \times c$ 的正方形。小 Z 希望寻找到一个合适的位置，使得首都所占领的位置的土地价值和最高。

输入：第一行有 3 个整数 n、m、c，表示地图的宽和长以及首都的边长。接下来 n 行，每行 m 个整数，表示地图上每个地块的价值，价值可能为负数。

数据范围：对于 60%的数据，$n, m \leqslant 50$；对于 90%的数据，$n, m \leqslant 300$；对于 100%的数据，$1 \leqslant n, m \leqslant 10^3$，$1 \leqslant c \leqslant \min(n, m)$。

输出：一行两个整数 x、y，表示首都左上角的坐标。

输入样例：	输出样例：
3 4 2 1 2 3 1 −1 9 0 2 2 0 1 1	1 2

概括题意：在 $n \times m$ 的矩形中找一个边长为 c 的正方形，把正方形内所有坐标点的值相加，使价值最大。

简单的做法是枚举每个坐标，作为正方形的左上角，然后计算出边长 c 内所有地块的价值和，找到价值和最大的坐标。时间复杂度为 $O(nmc^2)$，能通过 60%的测试。请读者练习。

本题是二维前缀和的直接应用。

一维前缀和定义在一维数组 a[]上：sum[i]＝a[1]＋a[2]＋…＋a[i]。

把一维数组 a[]看成一条直线上的坐标，前缀和就是所有坐标值的和，如图 5.1 所示。

二维前缀和是一维前缀和的推广。设二维数组 a[][]有 1～n 行、1～m 列，二维前缀和如下：

图 5.1　一维数组 a[]在直线上

$$\begin{aligned}sum[i][j]=&a[1][1]+a[1][2]+a[1][3]+\cdots+a[1][j]+\\&a[2][1]+a[2][2]+a[2][3]+\cdots+a[2][j]+\cdots+\\&a[i][1]+a[i][2]+a[i][3]+\cdots+a[i][j]\end{aligned}$$

把 a[i][j]的(i,j)看成二维平面的坐标，那么 sum[i][j]就是左下角坐标(1,1)和右上角坐标(i,j)围成的方形中所有坐标点的和，如图 5.2 所示。

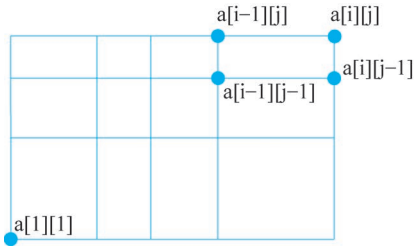

图 5.2　二维数组和二维平面

二维前缀和 sum[][]存在以下递推关系：

sum[i][j]＝sum[i－1][j]＋sum[i][j－1]－sum[i－1][j－1]＋a[i][j]

根据这个递推关系，用两种 for 循环可以计算出 sum[][]。

对照图 5.2 理解这个公式，sum[i－1][j]是坐标(1,1)～(i－1, j)内所有的点，sum[i][j－1]是(1,1)～(i, j－1)内所有的点，两者相加，其中 sum[i－1][j－1]被加了两次，所以要减去一次。

下面的 Python 代码中第 7 行让 sum[][]和 a[][]共用，从而节省一半空间，否则提交判题系统会超出空间限制。

```
1  n, m, c = map(int, input().split())
2  a = [[0] * (m + 1) for _ in range(n + 1)]
3  for i in range(1, n + 1):
4      row = list(map(int, input().split()))
5      for j in range(1, m + 1):
6          a[i][j] = row[j - 1]
7          a[i][j] = a[i-1][j] + a[i][j-1] - a[i-1][j-1] + a[i][j]
8  # 上一行等于 s[i][j] = s[i-1][j] + s[i][j-1] - s[i-1][j-1] + a[i][j]
9  Max = float('-inf')
10 x = 0
11 y = 0
12 for x1 in range(1, n - c + 2):
13     for y1 in range(1, m - c + 2):          # 枚举所有坐标点
14         x2 = x1 + c - 1
15         y2 = y1 + c - 1                     # 正方形的右下角坐标
16         ans = a[x2][y2] - a[x2][y1-1] - a[x1-1][y2] + a[x1-1][y1-1]
17         if ans > Max:
18             Max = ans
19             x = x1
20             y = y1
21 print(x, y)
```

5.3　差　分

扫一扫

视频讲解

前缀和的主要应用是差分，差分是前缀和的逆运算。

5.3.1 一维差分

与一维数组 a[]对应的差分数组 d[]的定义如下：

$$d[k]=a[k]-a[k-1]$$

即差分数组 d[]是原数组 a[]的相邻元素的差。根据 d[]的定义，可以反过来推出：

$$a[k]=d[1]+d[2]+\cdots+d[k]$$

即 a[]是 d[]的前缀和，所以“差分是前缀和的逆运算”。

为了方便理解，把每个 a[]看成直线上的坐标，把每个 d[]看成直线上的小线段，它的两端是相邻的 a[]，如图 5.3 所示。将这些小线段相加，就得到了从起点开始的长线段 a[]。

图 5.3 把每个 d[]看成小线段，把每个 a[]看成从 a[1]开始的小线段的和

差分是一种巧妙且简单的方法，它应用于区间的修改和查询问题。把给定的数据元素集 A 分成很多区间，对这些区间做很多次操作，每次操作是对某个区间内的所有元素做相同的加减操作，若一个一个地修改这个区间内的每个元素，复杂度是 O(n)，非常耗时。引入“差分数组”，当修改某个区间时，只需要修改这个区间的“端点”，就能记录整个区间的修改，而对端点的修改非常快，复杂度是 O(1)。当所有的修改操作结束后，再使用差分数组，计算出新的 A。

为什么使用差分数组能提升修改的效率？如何把 O(n)的区间操作优化为 O(1)的端点操作？

将区间[L，R]内的每个元素 a[]加上 v，只需要把对应的 d[]做以下操作。

(1) 把 d[L]加上 v：d[L] += v。

(2) 把 d[R+1]减去 v：d[R+1] -= v。

使用 d[]，能精确地实现只修改区间内元素的目的，而不会修改区间外的 a[]值。根据前缀和 a[x]=d[1]+d[2]+…+d[x]，有

(1) $1\leqslant x<L$，前缀和 a[x]不变。

(2) $L\leqslant x\leqslant R$，前缀和 a[x]增加了 v。

(3) $R<x\leqslant N$，前缀和 a[x]不变，因为被 d[R+1]中减去的 v 抵消了。

每次操作只需要修改区间[L，R]的两个端点的 d[]值，复杂度是 O(1)。经过这种操作后，原来直接在 a[]上做的复杂度为 O(n)的区间修改操作就变成了在 d[]上做的复杂度为 O(1)的端点操作。在完成区间修改并得到 d[]后，最后用 d[]计算 a[]，复杂度是 O(n)。m 次区间修改和一次查询，总复杂度为 O(m+n)，比暴力法的 O(mn)好多了。

数据 A 可以是一维的线性数组 a[]、二维矩阵 a[][]、三维立体 a[][][]。相应地，定义一维差分数组 D[]、二维差分数组 D[][]、三维差分数组 D[][][]。

例 5.5 重新排序 lanqiaoOJ 2128

问题描述：给定一个数组 A 和一些查询 L_i，R_i，求数组中第 L_i 至第 R_i 个元素的和。

小蓝觉得这个问题很无聊，于是他想重新排列一下数组，使最终每个查询结果的和尽可能的大。小蓝想知道相比原数组，所有查询结果的总和最多可以增加多少？

输入：输入的第一行包含一个整数 n。第二行包含 n 个整数 A_1、A_2、……、A_n，相邻两个整数之间用一个空格分隔。第三行包含一个整数 m，表示查询的数目。接下来的 m 行，每行包含两个整数 L_i、R_i，相邻两个整数之间用一个空格分隔。

输出：输出一行，包含一个整数，表示答案。

输入样例：	输出样例：
5 1 2 3 4 5 2 1 3 2 5	4

评测用例规模与约定：对于 30%的评测用例，n，m≤50；对于 50%的评测用例，n，m≤500；对于 70%的评测用例，n，m≤5000；对于所有评测用例，1≤n，m≤10^5，1≤A_i≤10^6，1≤L_i≤ R_i≤n。

本题的 m 个查询可以统一处理，在读入 m 个查询后，每个 a[i]被查询了多少次就知道了。用 cnt[i]记录 a[i]被查询的次数，cnt[i] * a[i]就是 a[i]对总和的贡献。

下面分别给出通过 70%和 100%测试的两种解法。

(1) 通过 70%的测试。

下面代码的第 11 行先计算出 cnt[]，第 13 行计算出原数组上的总和 ans1。

然后计算新数组上的总和。显然，把查询次数最多的数分给最大的数，对总和的贡献最大。对 a[]和 cnt[]排序，把最大的 a[n]与最大的 cnt[n]相乘、次大的 a[n－1]与次大的 cnt[n－1]相乘，等等。代码的第 17 行计算出新数组上的总和 ans2。

代码的主要计算是第 8 行和第 10 行的 for 循环，复杂度为 O(mn)，该代码只能通过 70%的测试。

```
1  N = 100003
2  a = [0] * N
3  cnt = [0] * N
4  n = int(input())
5  a[1:n+1] = map(int, input().split())          # a[0]不用，从 a[1]开始
6  m = int(input())
7  ans1,ans2 = 0,0                               # ans1：原区间和；ans2：新区间和
8  for _ in range(m):
9      L, R = map(int, input().split())
10     for i in range(L, R+1):
11         cnt[i] += 1                           # 第 i 个数被加了一次，累计加了多少次
12 for i in range(1, n+1):
13     ans1 += a[i] * cnt[i]                     # 在原数组上求区间和
14 a[1:n+1] = sorted(a[1:n+1])                   # a[0]不用，从 a[1]开始
15 cnt[1:n+1] = sorted(cnt[1:n+1])               # cnt[0]不用，从 cnt[1]开始
16 for i in range(1, n+1):
17     ans2 += a[i] * cnt[i]
18 print(ans2 - ans1)
```

（2）通过 100%的测试。

本题是差分优化的直接应用。

前面提到，通过 70%测试的代码效率低的原因是第 11 行计算 cnt[]。根据差分的应用场景，每次查询的[L，R]就是对 a[L]～a[R]中的所有数的累加次数加 1，也就是对 cnt[L]～cnt[R]中的所有 cnt[]加 1。那么对 cnt[]使用差分数组 d[]即可。

下面代码中第 11、12 行用差分数组 d[]记录 cnt[]的变化，第 15 行用 d[]恢复得到 cnt[]。其他部分和前面通过 70%测试的代码一样。

在下面的代码中，第 9 行 for 循环的计算复杂度只有 O(m)，最耗时的是第 18、19 行的排序，复杂度为 $O(n\log_2 n)$，该代码能通过 100%的测试。

```
1  N = 100003
2  a = [0] * N
3  d = [0] * N
4  cnt = [0] * N
5  n = int(input())
6  a[1:n+1] = map(int, input().split())
7  m = int(input())
8  ans1, ans2 = 0, 0
9  for _ in range(m):
10     L, R = map(int, input().split())
11     d[L] += 1
12     d[R+1] -= 1
13 cnt[0] = d[0]
14 for i in range(1, n+1):
15     cnt[i] = cnt[i-1] + d[i]          # 用差分数组 d[]求 cnt[]
16 for i in range(1, n+1):
17     ans1 += a[i] * cnt[i]
18 a[1:n+1] = sorted(a[1:n+1])
19 cnt[1:n+1] = sorted(cnt[1:n+1])
20 for i in range(1, n+1):
21     ans2 += a[i] * cnt[i]
22 print(ans2 - ans1)
```

再看一道例题。

例 5.6　推箱子 http://oj.ecustacm.cn/problem.php?id=1819

问题描述：在一个高度为 H 的箱子的前方有一个长和高为 N 的障碍物。障碍物的每一列存在一个连续的缺口，第 i 列的缺口从第 l 个单位到第 h 个单位（从底部由 0 开始数）。现在请清理出一条高度为 H 的通道，使得箱子可以直接推出去。请输出最少需要清理的障碍物面积。图 5.4 所示为样例中的障碍物，长度和高度均为 5，箱子的高度为 2，不需要考虑箱子会掉入某些坑中，最少需要移除两个单位的障碍物可以造出一条高度为 2 的通道。

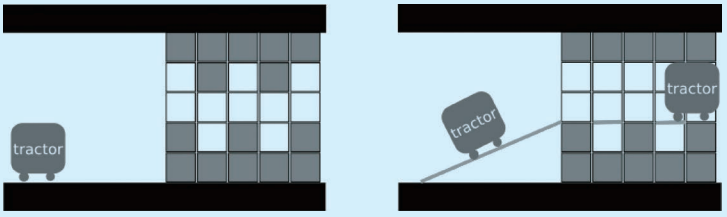

图 5.4　推箱子示例

输入：输入的第一行为两个正整数 N 和 H，表示障碍物的尺寸和箱子的高度，$1 \leqslant H \leqslant N \leqslant 1000000$。接下来 N 行，每行包含两个整数 l_i 和 h_i，表示第 i 列缺口的范围，$0 \leqslant l_i \leqslant h_i < N$。

输出：输出一个数字，表示答案。

输入样例：	输出样例：
5 2 2 3 1 2 2 3 1 2 2 3	2

箱子的高度为 H，检查障碍物中的连续 H 行，看哪 H 行需要清理的障碍物最少，或者哪 H 行中的空白最多。在输入样例中，障碍物共 5 行，这 5 行中的空白数量从底部开始往上数分别是(0，2，5，3，0)，其中(5，3)这两行的空白最多，是 5+3=8，需要移除的障碍物的数量是 N×H−8=5×2−8=2。

用数组 a[]表示障碍物，a[i]是障碍物中第 i 行的空白数量。把题目抽象为：a[]是 N 个整数，从 a[]中找出连续的 H 个整数，要求它们的和最大。

先考虑用暴力法求解。

(1) 如果用暴力法从左到右依次对 a[]中的 H 个整数求和，找到最大的和，总计算量是 O(NH)，超时。

(2) 需要注意输入的问题。题目按列给出空白数量，需要转换为行的空白数量。如果简单地转换，计算量太大。例如样例第 1 列的空白位置是(2，3)，需要赋值 a[2]++、a[3]++。一列有 H 个空白，a[]数组需要赋值 H 次，N 列的总计算量是 O(NH)，超时。

本题用差分与前缀和来优化。

(1) 用差分处理输入。下面代码中第 6 行读一个列的起点位置 l_i 和终点位置 h_i，第 7 行和第 8 行输入 d[]，d[]是 a[]的差分，计算量仅为 O(N)。

(2) 用前缀和求区间和。第 10 行用 d[]求 a[]；第 12 行计算 a[]的前缀和 sum[]；第 15 行找到最大的区间和。其计算量仅为 O(N)。

上述题解的实现见下面的代码。代码可以进行空间优化，d[]、a[]、sum[]都用 sum[]存储，见第 7、8、10、12 行的注释，请读者思考为什么可以这样做。当然，在一般情况下题目给的存储空间够大，不需要做这个优化。

```
1  import sys
2  input = sys.stdin.readline                    # 加这行后读入会快一些
3  n, h = map(int, input().split())
4  d, a, sum = [0] * (n+10), [0] * (n+10), [0] * (n+10)
5  for i in range(1, n+1):
6      li, hi = map(int, input().split())
7      d[li] += 1                                # 可替换为 sum[li]++;
8      d[hi+1] -= 1                              # 可替换为 sum[hi+1]--;
9  for i in range(1, n+1):                       # 用差分数组计算原数组
10     a[i] = a[i-1] + d[i-1]                    # 可替换为 sum[i] += sum[i-1];
```

```
for i in range(1, n + 1):                    # 用原数组计算前缀和数组
    sum[i] = sum[i - 1] + a[i]               # 可替换为 sum[i] += sum[i - 1];
ans = sum[h - 1]
for left in range(1, n - h + 2):
    ans = max(ans, sum[left + h - 1] - sum[left - 1])
print(n * h - ans)
```

5.3.2 二维差分

从一维差分容易扩展到二维差分。一维是线性数组，一个区间[L，R]有两个端点；二维是矩阵，一个区间由 4 个端点围成。设矩阵是 n 行 n 列的。

对比一维差分和二维差分的效率：一维差分的一次修改是 O(1)的，二维差分的修改也是 O(1)的，设有 m 次修改；一维差分的一次查询是 O(n)的，二维差分是 $O(n^2)$的，所以二维差分的总复杂度是 $O(m+n^2)$。由于计算一次二维矩阵的值需要 $O(n^2)$次计算，所以二维差分已经达到了最好的复杂度。

下面从一维差分推广到二维差分。由于差分是前缀和的逆运算，首先需要从一维前缀和推广到二维前缀和，然后从一维差分推广到二维差分。

在一维差分中，数组 a[]是从第 1 个 D[1]开始的差分数组 D[]的前缀和：

$$a[k]=D[1]+D[2]+\cdots+D[k]$$

在二维差分中，a[][]是差分数组 D[][]的前缀和。在由原点坐标(1，1)和坐标(i，j)围成的矩阵中，所有的 D[][]相加等于 a[i][j]。

可以把每个 D[][]看成一个小格；在坐标(1，1)和(i，j)所围成的范围内，所有小格子加起来的总面积等于 a[i][j]。在图 5.5 中，每个格子的面积是一个 D[][]，例如阴影格子是 D[i][j]，它由(i−1，j)、(i，j)、(i−1，j−1)、(i，j−1)4 个坐标点定义。坐标点(i，j)的值是 a[i][j]，它等于坐标(1，1)和(i，j)所围成的所有格子的总面积。图 5.5 演示了这些关系。

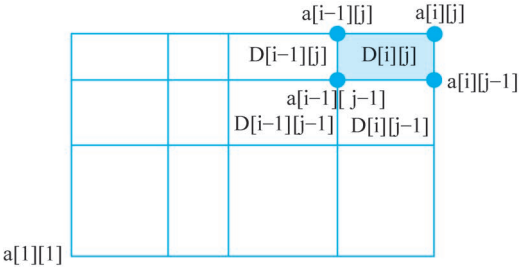

图 5.5　把每个 a[][]看成总面积，把每个 D[][]看成小格子的面积

在一维情况下，差分是 D[i]=a[i]−a[i−1]。在二维情况下，差分变成了相邻的 a[][]的“面积差”，计算公式是：

$$D[i][j]=a[i][j]-a[i-1][j]-a[i][j-1]+a[i-1][j-1]$$

这个公式可以通过图 5.5 来观察。阴影方格表示 D[i][j]的值，它的面积这样求：大面积 a[i][j]减去两个小面积 a[i−1][j]、a[i][j−1]，由于两个小面积的公共面积 a[i−1][j−1]被减了两次，所以需要加回来一次。

差分最关键的操作是区间修改。在一维情况下，做区间修改只需要修改区间的两个端

点的 D[]值。在二维情况下，一个区间是一个小矩阵，有 4 个端点，需要修改这 4 个端点的 D[][]值。例如坐标点(x_1，y_1)～(x_2，y_2)定义的区间，对应 4 个端点的 D[][]：

```
1  D[x1][y1] += d;          //二维区间的起点
2  D[x1][y2 + 1] -= d;      //把 x 看成常数，y 从 y1 到 y2 + 1
3  D[x2 + 1][y1] -= d;      //把 y 看成常数，x 从 x1 到 x2 + 1
4  D[x2 + 1][y2 + 1] += d;  //由于前两式把 d 减了两次，多减了一次，这里加一次回来
```

图 5.6 演示了区间修改。两个黑色点围成的矩形是题目给出的区间修改范围，只需要改变 4 个 D[][]值，就改变了图中 4 个阴影块的面积。

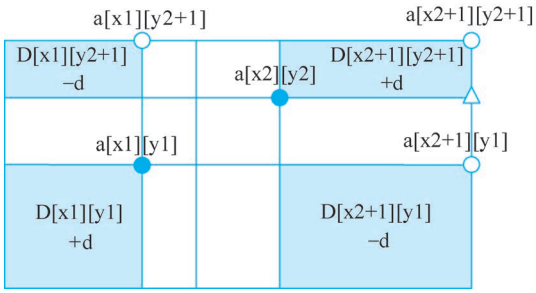

图 5.6　二维差分的区间修改

读者可以用这个图观察每个坐标点的 a[][]值的变化情况。例如符号“△”标记的坐标(x2+1,y2)，它在修改的区间之外；a[x2+1][y2]的值是从(1,1)到(x2+1,y2)的总面积，在这个范围内，D[x1][y1]+d，D[x2+1][y1]−d，两个 d 抵消，a[x2+1][y2]保持不变。

下面的例题是二维差分的直接应用。

例 5.7　2023 年第十四届蓝桥杯省赛 Python 大学 C 组试题 F：棋盘 lanqiaoOJ 3533

时间限制：10s 内存限制：512MB 本题总分：15 分

问题描述：小蓝有一个 n×n 大小的棋盘，一开始棋盘上都是白子。小蓝进行了 m 次操作，每次操作会将棋盘上某个范围内的所有棋子的颜色取反(也就是白色棋子变为黑色，黑色棋子变为白色)。请输出所有操作做完后棋盘上每个棋子的颜色。

输入：输入的第一行包含两个整数 n、m，用一个空格分隔，表示棋盘的大小与操作数。接下来 m 行，每行包含 4 个整数 x_1、y_1、x_2、y_2，相邻整数之间使用一个空格分隔，表示将在 x_1 至 x_2 行和 y_1 至 y_2 列中的棋子的颜色取反。

输出：输出 n 行，每行 n 个 0 或 1，表示该位置上棋子的颜色。如果是白色，输出 0，否则输出 1。

输入样例：	输出样例：
3 3 1 1 2 2 2 2 3 3 1 1 3 3	001 010 100

评测用例规模与约定：对于 30%的评测用例，n,m≤500；对于所有评测用例，1≤n,m≤2000，1≤x_1≤x_2≤n，1≤y_1≤y_2≤m。

下面用两种方法求解，分别通过30%和100%的测试。

（1）模拟。

按题目的描述编码实现。第3、5、6行有三重for循环，计算复杂度为$O(mn^2)$，只能通过30%的测试。

```
n, m = map(int, input().split())
a = [[0] * 2100 for _ in range(2100)]
for _ in range(m):
    x1, y1, x2, y2 = map(int, input().split())
    for i in range(x1, x2 + 1):
        for j in range(y1, y2 + 1):
            a[i][j] ^= 1
for i in range(1, n + 1):
    for j in range(1, n + 1):
        print(a[i][j], end = "")
    print()
```

（2）差分。通过100%的测试。

这是一道很直白的二维差分题。第5～8行是二维差分计算，其实改成注释中的写法也行，请读者思考。

```
n, m = map(int, input().split())
a = [[0] * (n + 2) for _ in range(n + 2)]
d = [[0] * (n + 2) for _ in range(n + 2)]
def insert(x1, y1, x2, y2):
    d[x1][y1] += 1          # d[x1][y1] ^= 1
    d[x1][y2 + 1] -= 1      # d[x1][y2 + 1] ^= 1
    d[x2 + 1][y1] -= 1      # d[x2 + 1][y1] ^= 1
    d[x2 + 1][y2 + 1] += 1  # d[x2 + 1][y2 + 1] ^= 1
for _ in range(m):
    x1, y1, x2, y2 = map(int, input().split())
    insert(x1, y1, x2, y2)
for i in range(1, n + 1):
    for j in range(1, n + 1):
        a[i][j] = d[i][j] + a[i - 1][j] + a[i][j - 1] - a[i - 1][j - 1]
        print(a[i][j] & 1, end = "")
    print()
```

【练习题】

lanqiaoOJ：灵能传输196、和与乘积1595、冒险岛3224、统计子矩阵2109、无线网络发射器选址374。

洛谷：求区间和B3612、光骓者的荣耀P5638、dx分计算P10233、领地选择P2004、地毯P3397、最大加权矩形P1719、卡牌游戏P6625。

扫一扫 视频讲解

5.4 二　分

二分法是一种思路简单、编码容易、效率极高、应用广泛的算法。在任何算法竞赛中，二分法都是最常见的知识点之一。二分法的应用场景比前缀和更广泛。

二分法的思想很简单，每次把搜索范围缩小一半，直到找到答案为止。设初始范围是[L，R]，常见的二分法代码这样写：

```
while L < R:                        # 一直二分,直到区间[L,R]缩小到 L = R
    mid = (L + R) // 2              # mid 是 L、R 的中间值
    if check(mid):R = mid           # 答案在左半部分[L,mid],更新 R = mid
    else:L = mid + 1                # 答案在右半部分[mid + 1,R],更新 L = mid + 1
```

答案在区间[L, R]中,通过中间值 mid 缩小搜索范围。这样搜索答案:

(1) 如果答案在左半部分,把范围缩小到[L, mid],更新 R=mid,然后继续。

(2) 如果答案在右半部分,把范围缩小到[mid+1, R],更新 L=mid+1,然后继续。

经过多次二分,最后范围缩小到 L=R,这就是答案。

二分的效率极高,例如在 $2^{30}=10$ 亿个数字中查找某个数,前提是这些数已经排序,然后用二分法来找,最多只需要找 $\log_2(2^{30})=30$ 次。二分法的计算复杂度是 $O(\log_2 n)$。

以猜数字游戏为例,一个[1, 100]内的数字,猜 7 次就能猜出来。例如猜数字 68,如表 5.2 所示。

表 5.2 猜数字 68

次 数	原[L,R]	a[mid]	check()判断	更新[L,R]
1	[1,100]	50	小于或等于 50 吗? --不	[51,100]
2	[51,100]	75	小于或等于 75 吗? --是	[51,75]
3	[51,75]	63	小于或等于 63 吗? --不	[64,75]
4	[64,75]	69	小于或等于 69 吗? --是	[64,69]
5	[64,69]	66	小于或等于 66 吗? --不	[67,69]
6	[67,69]	68	小于或等于 68 吗? --是	[67,68]
7	[67,68]	67	小于或等于 67 吗? --不	[68,68]

下面是代码实现。二分函数 bin_search()操作 3 个变量:区间左端点 L、右端点 R、二分的中位数 mid。每次把区间缩小一半,把 L 或 R 更新为 mid,直到 L= R 为止,即找到了答案所处的位置。

```
def check(x, mid): return x <= a[mid]
def bin_search(n, x):
    L, R = 1, n
    while L < R:
        mid = (L+R) // 2
        if check(x, mid):R = mid
        else:L = mid + 1
    return a[L]
n = 100
a = [0] * 105
for i in range(1, n + 1):a[i] = i
x = 68
print("x = ", bin_search(n, x))
```

二分法把长度为 n 的有序序列上 O(n)的查找时间优化到了 $O(\log_2 n)$。

注意,二分法的应用前提是序列是单调有序的,从小到大或从大到小。在无序的序列上无法二分,如果是乱序的,应该先排序再二分。

如果在乱序序列上只搜一次,不需要用二分法。如果用二分法,需要先排序,排序的复杂度为 $O(n\log_2 n)$,再二分是 $O(\log_2 n)$,排序加二分的总复杂度为 $O(n\log_2 n)$。如果用暴力法直接在乱序的 n 个数中查找,复杂度是 O(n),比排序加二分快。

如果不是搜一个数,而是搜 m 个数,那么先排序再做 m 次二分的计算复杂度是

$O(n\log_2 n+m\log_2 n)$，而暴力法是 $O(mn)$，当 m 很大时，二分法远远好于暴力法。

在做二分法题目时，需要建模出一个有序的序列，并且答案在这个序列中。在编程时，根据题目要求确定区间[L，R]范围，并写一个 check()函数来更新 L 和 R。

[L，R]的中间值 mid 有多种计算方法：

```
mid = (L+R)/2
mid = (L+R)>>1
mid = L+(R-L)/2
```

5.4.1 二分法的经典应用

二分法的经典应用场景是“最小值最大化（最小值尽量大）”和“最大值最小化（最大值尽量小）”。

1. 最小值最大化

以“牛棚问题”为例。有 n 个牛棚，分布在一条直线上，有 k 头牛，给每头牛安排一个牛棚住，$k<n$。由于牛的脾气很大，所以希望让牛之间尽量住得远一些。这个问题简化为：在一条直线上有 n 个点，选 k 个点，其中某两点之间的距离是所有距离中最小的，求解目标是让这个最小距离尽量大。

这就是“最小值（两点间的最小距离）最大化”。

“牛棚问题”的求解可以用猜的方法。猜最小距离是 D，看能不能在 n 个点中选 k 个，使得任意两点之间的距离≥D。如果可以，说明 D 是一个合法的距离。然后猜新的 D，直到找到那个最大的合法的 D。

具体操作是：从第 1 个点出发，逐个检查后面的点，第一个距离≥D 的点就是选中的第 2 点；然后从第 2 点出发，再选后面第一个距离第 2 点≥D 的点，这是选中的第 3 点；…继续下去，直到检查完 n 个点。这一轮猜测的计算复杂度是 O(n)。

检查完 n 个点，如果选中的点的数量≥k，说明 D 猜小了，下次猜大一点；如果选中的点的数量<k，说明 D 猜大了，下次猜小一点。

如何猜 D？简单的办法是从小到大一个一个试，但是计算量太大了。

用二分法可以加速猜 D 的过程。设 D 的初值是一个极大的数，例如就是所有 n 个点的总长度 L。接下来的二分操作和前面的“猜数字游戏”一样，经过 $O(\log_2 L)$次，就能确定 D。

总计算量：一共进行 $O(\log_2 L)$轮猜测，每一轮的计算量为 O(n)，总计算量为 $O(n\log_2 L)$。

2. 最大值最小化

经典的例子是“序列划分”问题。有一个包含 n 个正整数的序列，把它划分成 k 个子序列，每个子序列是原序列的一个连续部分，第 i 个子序列的和为 S_i。在所有 s 中有一个最大值，问如何划分才能使最大的 s 最小？

这就是“最大值（所有子序列和的最大值）最小化”。

例如序列{2，2，3，4，5，1}，将其划分成 k=3 个连续的子序列。下面举两种分法：{(2，2，3)，(4，5)，(1)}，子序列和分别是 7、9、1，最大值是 9；{(2，2，3)，(4)，(5，1)}，子序列和是 7、4、6，最大值是 7。第 2 种分法比第 1 种分法好。

仍然用猜的方法。在一次划分中猜一个 x，对任意的 S_i 都有 $S_i\leqslant x$，也就是说 x 是所有

S_i 中的最大值。

如何找到这个 x？简单的办法是枚举每一个 x，用贪心法每次从左向右尽量多划分元素，S_i 不能超过 x，划分的子序列的个数为 k 个，但是枚举所有的 x 太耗时了。

用二分法可以加速猜 x 的过程。用二分法在[max，sum]范围内查找满足条件的 x，其中 max 是序列中最大元素的值，sum 是所有元素的和。

5.4.2 例题

二分法的题目非常多，请读者大量练习二分法的题目。下面举几个例子。

例 5.8 求阶乘 lanqiaoOJ 2145

问题描述：满足 n!的末尾恰好有 k 个 0 的最小的 n 是多少？如果这样的 n 不存在，输出－1。

输入：输入一个整数 k。

输出：输出一个整数，表示答案。

输入样例：	输出样例：
2	10

评测用例规模与约定：对于 30%的评测用例，$1 \leqslant k \leqslant 10^6$；对于 100%的评测用例，$1 \leqslant k \leqslant 10^{18}$。

尾零是由 2×5 相乘得到的，所以只需要计算 n!中 2 和 5 的因子数量。又因为 n!中 2 的因子数量远大于 5 的因子数量，所以只需要计算 5 的因子数量。

例如 25!＝25×…×20×…×15×…×10×…×5×…，其中的 25、20、15、10、5 分别有 2、1、1、1、1 共 6 个因子 5，所以尾零有 6 个。

(1) 通过 30%的测试。

简单的方法是检查每个 n，计算 n!的尾零数量，尾零数量等于 k 的 n 就是答案。下面代码中第 3 行的 while 循环 n/5 次，对于 100%的数据，$1 \leqslant k \leqslant 10^{18}$，由于 n 比 k 还大，代码超时。

check(n)函数返回 n!的尾零数量，也就是计算 n!有多少个因子 5，读者可以按自己的思路实现这个函数。下面的 check()用了巧妙的方法。以 25!为例，对 5 有贡献的是 5、10、15、20、25，即 5×1、5×2、5×3、5×4、5×5，共有 6 个 5，其中 5 个 5 是由 25/5 得到的，即 5×i 中的 5；还有一个 5 是循环 5 次后多的一个 5，即 i×5 的 5。再例如 100!，尾零的数量包括两部分：100/5＝20、20/5＝4。

```
1  def check(n):
2      cnt = 0
3      while n:
4          cnt += n // 5
5          n //= 5
6      return cnt
7  k = int(input())
8  for n in range(5, 10 ** 18, 5):
9      cnt = check(n)
10     if cnt == k:
11         print(n)
```

```
            break
        if cnt > k:
            print(-1)
            break
```

（2）通过100%的测试。

本题可以用二分优化，也就是用二分来猜n。因为n递增时，尾零的数量也是单调递增的，符合二分法的应用条件。

下面讨论代码的计算复杂度。第11行的二分是$O(\log_2 E)$，这里$E=10^{19}$。第1行做一次check()差不多是$O(1)$。总计算量约为$\log_2 E=\log_2 10^{19}<70$次。

```
def check(n):                           # 计算n!的末尾有多少个0
    cnt = 0
    while n:
        cnt += n // 5
        n //= 5
    return cnt
k = int(input())
L = 0
R = 10 ** 19                            # R的初值为一个极大的数
while L < R:
    mid = (L + R) // 2
    if check(mid) >= k:R = mid          # mid!的尾零数量超过了k,说明mid大了
    else:L = mid + 1                    # mid小了
if check(R) == k:print(R)
else:print(-1)
```

例5.9　2022年第十三届蓝桥杯省赛Python大学C组试题I：青蛙过河　lanqiaoOJ 2097

时间限制：1s 内存限制：512MB 本题总分：25分

问题描述：小青蛙住在一条河的河边，它想到河对岸的学校去学习。小青蛙打算经过河里的石头跳到对岸。河里的石头排成了一条直线，小青蛙每次跳跃必须落在一块石头或者岸上。每块石头有一个高度，每次小青蛙从一块石头起跳，这块石头的高度就会下降1，当石头的高度下降到0时小青蛙不能再跳到这块石头上（某次跳跃后使石头的高度下降到0是允许的）。小青蛙一共需要去学校上x天课，所以它需要往返2x次。当小青蛙具有一个跳跃能力y时，它能跳不超过y的距离。请问小青蛙的跳跃能力至少是多少才能用这些石头上完x次课。

输入：输入的第一行包含两个整数n、x，分别表示河的宽度和小青蛙需要去学校的天数。注意2x才是小青蛙实际过河的次数。第二行包含n−1个非负整数H_1、H_2、……、H_{n-1}，其中$H_i>0$表示在河中与小青蛙的家相距i的地方有一块高度为H_i的石头，$H_i=0$表示这个位置没有石头。

输出：输出一行，包含一个整数，表示小青蛙需要的最小跳跃能力。

输入样例：	输出样例：
5 1 1 0 1 0	4

评测用例规模与约定：对于30%的评测用例，$n\leqslant 100$；对于60%的评测用例，$n\leqslant 1000$；对于所有评测用例，$1\leqslant n\leqslant 10^5$，$1\leqslant x\leqslant 10^9$，$1\leqslant H_i\leqslant 10^4$。

往返累计 2x 次相当于单向走 2x 次。跳跃能力越大，越能保证可以通过 2x 次。用二分法找到一个最小的满足条件的跳跃能力。设跳跃能力为 mid，每次能跳多远就跳多远，用二分法检查 mid 是否合法。

```
def check(sum, mid, x):
    for i in range(1, len(sum) - mid + 1):      # 每个区间的高度之和都要大于或等于 2x
        if sum[i + mid - 1] - sum[i - 1] < 2 * x:
            return False
    return True
n, x = map(int, input().split())
h = [0] * n
h[1:] = list(map(int, input().split()))         # h[0]不用，从 h[1]开始
sum = [0] * n
sum[0] = 0
for i in range(1, n):
    sum[i] = sum[i - 1] + h[i]                  # 跳跃区间的高度之和
L, R = 1, n
while L < R:
    mid = (L + R) // 2
    if check(sum, mid, x):R = mid
    else:L = mid + 1
print(L)
```

例 5.10　2023 年第十四届蓝桥杯省赛 Python 大学 B 组试题 D：管道　lanqiaoOJ 3544

时间限制：10s 内存限制：512MB 本题总分：10 分

问题描述：有一根长度为 len 的横向管道，该管道按照单位长度分为 len 段。每一段的中央有一个可开关的阀门和一个检测水流的传感器。一开始管道是空的，位于 L_i 的阀门会在 S_i 时刻打开，并不断地让水流入管道。对于位于 L_i 的阀门，它流入的水在 T_i（$T_i \geqslant S_i$）时刻会使从第 $L_i-(T_i-S_i)$ 段到第 $L_i+(T_i-S_i)$ 段的传感器检测到水流。求管道中每一段中间的传感器都检测到有水流的最早时间。

输入：输入的第一行包含两个整数 n、len，用一个空格分隔，分别表示会打开的阀门数和管道长度。接下来 n 行，每行包含两个整数 L_i、S_i，用一个空格分隔，表示位于第 L_i 段管道中央的阀门会在 S_i 时刻打开。

输出：输出一个整数，表示答案。

输入样例：	输出样例：
3 10 1 1 6 5 10 2	5

评测用例规模与约定：对于 30%的评测用例，$n \leqslant 200$，S_i，$len \leqslant 3000$；对于 70%的评测用例，$n \leqslant 5000$，S_i，$len \leqslant 10^5$；对于 100%的评测用例，$1 \leqslant n \leqslant 10^5$，$1 \leqslant S_i$，$len \leqslant 10^9$，$1 \leqslant L_i \leqslant len$，$L_{i-1} < L_i$。

按题目的设定，管道内是贯通的，每个阀门都连着一个进水管，打开阀门后会有水从这个进水管进入管道，并逐渐流到管道内的所有地方。

先解释样例。设长度 L 的单位是米，水流的速度是米/秒。L=1 处的阀门在第 S=1 秒打开，T=5 秒时，覆盖范围 L-(T-S)=1-(5-1)=-3，L+(T-S)=1+(5-1)=5；L=6 处的阀门在 S=5 秒打开，T=5 秒时，只覆盖了 L=6；L=10 处的阀门在 L=2 秒打开，T=5 秒时，覆盖范围 L-(T-S)=10-(5-2)=7，L+(T-S)=10+(5-2)=13。所以这 3 个阀门在 T=5 时覆盖了[-3，5]、6、[7,13]，管道中的所有传感器都检测到了水流。

读者可能会立刻想到可以用二分法猜时间 T。先猜一个 T，然后判断在 T 时刻是否整个管道有水。如何判断？位于 L_i 的阀门，它影响到的小区间是[$L_i-(T_i-S_i)$，$L_i+(T_i-S_i)$]，n 个阀门对应了 n 个小区间。那么问题转化为：给出 n 个小区间，是否能覆盖整个大区间。这是贪心法中的“区间覆盖问题”，请大家阅读 5.5 节中对“区间覆盖问题”的说明。

本题还可以再简单一点。题目给的评测用例指出 $L_{i-1}<L_i$，即已经按左端点排序，可以省去排序的步骤。

在 check(t)函数中，定义 last_L 为当前覆盖到的最左端，last_R 为最右端。然后逐个遍历所有的小区间，看它对扩展 last_L、last_R 有无贡献。在所有小区间处理完毕后，如果[last_L,last_R]能覆盖整个[1，len]区间，则这个时刻 t 就是可行的。

```
def check(t):                    # 检查 t 时刻,管道内是否都有水
    cnt = 0
    last_L = 2
    last_R = 1
    for i in range(n):
        if t >= S[i]:
            cnt += 1             # 特判 t 是否够大
            left = L[i] - (t - S[i])
            right = L[i] + (t - S[i])
            if left < last_L:
                last_L = left
                last_R = max(last_R, right)
            elif left <= last_R + 1:
                last_R = max(last_R, right)
    if cnt == 0:return False
    if last_L <= 1 and last_R >= length:
        return True
    else: return False

n, length = map(int, input().split())
L = []
S = []
for _ in range(n):
    l, s = map(int, input().split())
    L.append(l)
    S.append(s)
LL,R = 0, int(2e9)               # LL 避免和 L[]重名
ans = -1
while LL <= R:                   # 二分
    mid = (R + LL) >> 1
    if check(mid):
        ans = mid
        R = mid - 1
    else:LL = mid + 1
print(ans)
```

下面是一道思维有难度的二分法题目。

例 5.11　2022 年第十三届蓝桥杯省赛 Python 大学 B 组试题 H：技能升级 lanqiaoOJ 2129

时间限制：1s 内存限制：512MB 本题总分：20 分

问题描述：小蓝最近正在玩一款 RPG 游戏。他的角色一共有 N 个可以加攻击力的技能。其中第 i 个技能首次升级可以提升 A_i 点攻击力，以后每次升级增加的点数都会减少 B_i。$\lceil A_i/B_i \rceil$（向上取整）次之后，再升级该技能将不会改变攻击力。现在小蓝可以总计升级 M 次技能，他可以任意选择升级的技能和次数。请计算小蓝最多可以提高多少点攻击力？

输入：输入的第一行包含两个整数 N 和 M。以下 N 行，每行包含两个整数 A_i 和 B_i。

输出：输出一行，包含一个整数，表示答案。

输入样例：	输出样例：
3 6 10 5 9 2 8 1	47

评测用例规模与约定：对于 40%的评测用例，$1 \leqslant N, M \leqslant 1000$；对于 60%的评测用例，$1 \leqslant N \leqslant 10^4$，$1 \leqslant M \leqslant 10^7$；对于所有评测用例，$1 \leqslant N \leqslant 10^5$，$1 \leqslant M \leqslant 2 \times 10^9$，$1 \leqslant A_i, B_i \leqslant 10^6$。

下面讲解 3 种方法，它们分别能通过 40%、60%、100%的测试。

（1）暴力法。

先试一试暴力法，直接模拟题意，升级 m 次，每次升级时选用攻击力最高的技能，然后更新它的攻击力。

下面代码的复杂度是多少？第 12 行升级 m 次，是 O(m)的；第 13 行选用最大攻击，使用 Python 的 max()函数，是 O(n)的。总复杂度为 O(mn)，只能通过 40%的测试。

```
1  import math
2  n,m = map(int,input().split())
3  a = []                                  # 存 ai
4  b = []                                  # 存 bi
5  c = []                                  # ai/bi
6  for i in range(n):
7      a_,b_ = map(int,input().split())
8      a.append(a_)
9      b.append(b_)
10     c.append(math.ceil(a_/b_))          # 向上取整
11 ans = 0
12 for i in range(m):                      # 一共升级 m 次
13     max_num = max(a)                    # 每次升级时使用最大的攻击
14     index = a.index(max_num)            # 最大攻击对应的序号
15     a[index] -= b[index]                # 更新攻击力
16     if c[index]>0: ans += max_num       # 累加攻击力
17     c[index] -= 1
18 print(ans)
```

（2）暴力法＋优先队列。

上面的代码可以稍做改进。在 n 个技能中选用最高攻击力，可以使用优先队列，一次操作的复杂度为 $O(\log_2 n)$。m 次升级，总复杂度为 $O(m\log_2 n)$，能通过 60％的测试。

下面是 Python 的“暴力法＋优先队列”实现。优先队列用堆实现，heapq 默认是小根堆，第 7 行把 a 取负，将最大值变成了最小值。

```
1  from heapq import *
2  n,m = map(int,input().split())
3  q = []
4  ans = 0
5  for i in range(n):
6      a,b = map(int,input().split())
7      heappush(q,(-a,b))
8  for i in range(m):                    # 升级 m 次
9      p = heappop(q)                    # 每次升级时使用最大的攻击。读队列最大值并删除
10     a,b = -p[0],p[1]
11     ans += a                          # 累加攻击力
12     a = max(a - b, 0)                 # 更新攻击力
13     if a != 0: heappush(q,(-a,b))     # 重新放进队列
14 print(ans)
```

（3）二分法。

本题的正解是二分法，能通过 100％的测试。

本题 $m\leqslant 2\times 10^9$ 太大，若逐一升级 m 次必定会超时。但是不能直接对 m 进行二分，因为需要知道每个技能升级多少次，而这与 m 无关。

思考升级技能的过程，是每次找攻击力最高的技能。对某个技能，最后一次升级的攻击力肯定比之前升级的攻击力小，也就是前面的升级都更大。可以设最后一次升级提升的攻击力是 mid，对每个技能，若它最后一次能升级 mid，那么它前面的升级都更大。所有这样最后能达到 mid 的技能，它们前面的升级都应该使用。用二分法找到这个 mid，另外，升级技能减少的攻击力的过程是一个等差数列，用 O(1)次计算即可知道每个技能升级了几次。知道了每个技能升级的次数，就可以计算一共提升了多少攻击力，这就是题目的答案。

下面给出 Python 代码。check(mid)函数找这个 mid。第 6 行，若所有技能升级的总次数大于或等于 m 次，说明 mid 设小了，在第 16 行让 L 增大，即增加了 mid。第 8 行，若所有技能升级的总次数小于 m 次，说明 mid 设大了，在第 17 行让 R 减小，即缩小了 mid。

分析代码的复杂度。第 14～17 行，二分 $O(\log_2 A)$次，这里 A 表示 $1\leqslant A_i\leqslant 10^6$；每次 check()是 O(n)，二分的总复杂度是 $O(n\log_2 A)$。第 20 行的 for 循环是 O(n)。代码的总复杂度是 $O(n\log_2 A)+O(n)$，能通过 100％的测试。

```
1  def check(mid):                       # 最后一次技能升级，最多能不能到 mid
2      cnt = 0
3      for a,b in skills:
4          if a < mid:continue           # 第 i 个技能的初值还不够 mid，不用这个技能
5          cnt += (a - mid) // b + 1     # 第 i 个技能用掉的次数
6          if cnt >= m:                  # 所有技能升级的总次数大于或等于 m 次，说明 mid 设小了
7              return True
8      return False                      # 所有技能升级的总次数小于 m 次，说明 mid 设大了
9
10 n, m = map(int, input().split())
11 skills = [list(map(int, input().split())) for _ in range(n)]
```

```
12
13  L,R = 1,1000000                    # 二分枚举最后一次攻击力最大能加多少
14  while(L <= R):
15      mid = (L + R) // 2
16      if check(mid): L = mid + 1     # 增加 mid
17      else:R = mid - 1               # 减小 mid
18  attack = 0
19  cnt = m
20  for a,b in skills:
21      if a < R:continue
22      t = (a - L) // b + 1           # 第 i 个技能升级的次数
23      if a - b * (t - 1) == R:
24          t -= 1                     # 这个技能每次升级刚好等于 R,其他技能更好
25      attack += (a * 2 - (t - 1) * b) * t / 2
26      cnt -= t
27  print(int(attack) + cnt * R)
```

【练习题】

二分的题目非常多，每个 OJ 网站都能用“二分”搜出很多二分题目。

lanqiaoOJ：分巧克力 99、跳石头 364、课凑成的最大花束数 3344、最大通过数 3346、蓝桥 A 梦做铜锣烧 3151、肖恩的苹果林 3683、求函数零点 4496、妮妮的月饼工厂 3990、解立方根 1217、一元三次方程求解 764、二分查找数组元素 1389。

5.5 贪　心

扫一扫 视频讲解

贪心(Greedy)是容易理解的算法思想：把整个问题分解成多个步骤，在每个步骤都选取当前步骤的最优方案，直到所有步骤结束；在每一步都不考虑对后续步骤的影响，在后续步骤中也不能回头改变前面的选择①。

贪心策略在人们的生活中经常用到。例如下象棋时，初级水平的棋手只会“走一步看一步”，就是贪心法；而水平高的棋手能“走一步看三步”，轻松击败初级棋手，可以看成是动态规划。

贪心这种“只顾当下，不管未来”的解题策略让人疑惑：在完成所有局部最优操作后得到的解不一定是全局最优，那么该如何判断能不能用贪心呢？

有时很容易判断：一步一步在局部选择最优，最后结束时能达到全局最优。例如吃自助餐，怎么吃才能“吃回票价”？它的数学模型是一类背包问题，称为“部分背包问题”：有一个容量为 C 的背包，有 m 种物品，第 i 种物品 w_i 千克，单价为 v_i，且每种物品是可以分割的，例如大米、面粉等，问如何选择物品使得装满背包时总价值最大。此时显然可以用贪心法，只要在当前物品中选最贵的放进背包即可：先选最贵的物品 A，A 放完之后再选剩下的最贵的物品 B，……，直到背包放满。

有时看起来能用贪心，但实际上贪心的结果不是最优解。例如最少硬币支付问题：有多种面值的硬币，数量不限，需要支付 M 元，问怎么支付才能使硬币数量最少？

① 作者拟过两句赠言：“贪心说，我从不后悔我走过的路”，“贪心说，其实我有一点后悔，但是我回不了头”。大多数读者选前一句。

最少硬币支付问题[①]是否能用贪心求最优解和硬币的面值有关。

如果硬币的面值为 1 元、2 元、5 元，用贪心是对的。贪心策略是当前选择可用的最大面值的硬币。例如支付 M＝18 元，第一步选面值最大的 5 元硬币，用掉 3 个硬币，还剩 3 元；第二步选面值第二大的 2 元硬币，用掉 1 个硬币，还剩 1 元；最后选面值最小的 1 元硬币，用掉 1 个，这样共用 5 个硬币。在这个解决方案中，硬币总数是最少的，贪心法的结果是全局最优的。

如果是其他面值的硬币，用贪心法不一定能得到全局最优解。例如，硬币的面值很奇怪，分别是 1 元、2 元、4 元、5 元、6 元。支付 M＝9 元，如果用贪心法，每次选择当前面值最大的硬币，那么答案是 6＋2＋1，需要 3 个硬币，而最优解是 5＋4，只需要两个硬币。

概括地说，判断一个题目是不是能用贪心，需要满足以下特征：

（1）最优子结构性质。当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质，也称此问题满足最优性原理。

（2）贪心选择性质。问题的整体最优解可以通过一系列局部最优的选择来得到。也就是说，通过一步一步局部最优能最终得到全局最优。

最后讨论贪心法的效率，贪心法的计算量是多少？贪心法由于每一步都在局部做计算，且只选取当前最优的步骤做计算，不管其他可能的计算方案，所以计算量很小。在很多情况下，贪心法可以说是计算量最少的算法了。与此相对，暴力法一般是计算复杂度最差的，因为暴力法计算了全局的所有可能的方案。

由于贪心的效率高，所以如果一个问题确定可用贪心法得到最优解，那么应该使用贪心。如果用其他算法，大概率会超时。

在算法竞赛中，贪心法几乎是必考点，有的题考验思维能力，有的题结合了贪心和其他算法。虽然贪心策略很容易理解，但贪心题可能很难。

贪心也是蓝桥杯大赛的常见题型。不论是省赛还是国赛，贪心出现的概率都非常大。

虽然用贪心法不一定能得到最优解，但是它解题步骤简单、编程容易、计算量小，得到的解“虽然不是最好的，但是还不错！”像蓝桥杯这种赛制，一道题有多个测试点，用贪心也许能通过 10％～30％的测试，若别无他算法，可以一试。

5.5.1 经典贪心问题

1. 部分背包问题

前文介绍了用贪心求解部分背包问题，下面是例题。

例 5.12 部分背包问题 https://www.luogu.com.cn/problem/P2240

问题描述：有 N(N≤100)堆金币，第 i 堆金币的总重量和总价值分别是 m_i、v_i(1≤m_i，v_i≤100)。有一个承重量为 C(C≤1000) 的背包，要求装走尽可能多价值的金币。所有金币都可以随意分割，分割完的金币的重量价值比(也就是单位价格)不变。请问最多可

① 任意面值的最少硬币支付问题，正解是动态规划。参考《算法竞赛入门到进阶》，清华大学出版社，罗勇军著，7.1.1 节给出了各种硬币问题的动态规划解法。

以拿走多少价值的金币？

输入：第一行两个整数 N、C。接下来 N 行，每行两个整数 m_i、v_i。

输出：输出一个实数表示答案，保留两位小数。

输入样例：	输出样例：
4 50 10 60 20 100 30 120 15 45	240.00

按单位价格排序，最贵的先拿，便宜的后拿。

```
n, c = map(int, input().split())
a = []
for i in range(n):
    w, v = map(int, input().split())
    p = v / w
    a.append((w, v, p))
a.sort(key = lambda x: x[2], reverse = True)
sum = 0.0
for i in range(n):
    if c >= a[i][0]:
        c -= a[i][0]
        sum += a[i][1]
    else:
        sum += c * a[i][2]
        break
print("%.2f" %sum)
```

2. 不相交区间问题

不相交区间问题或者称为区间调度问题、活动安排问题。

给定一些区间（活动），每个区间有左端点和右端点（开始时间和终止时间），要求找到最多的不相交区间（活动）。

以下按“活动安排问题”来解释。

这个问题的目的是求最多活动数量，所以持续时间长的活动不受欢迎，受欢迎的是尽快结束的、持续时间短的活动。

考虑以下 3 种贪心策略：

（1）最早开始时间。先选最早开始的活动 a，当 a 结束后，再选下一个最早开始的活动。这种策略不好，因为它没有考虑活动的持续时间。假如 a 一直不结束，那么其他活动就不能开始。

（2）最早结束时间。先选最早结束的活动 a，a 结束后，再选下一个最早结束的活动。这种策略是合理的。越早结束的活动，越能腾出后续时间容纳更多的活动。

（3）用时最少。先选时间最短的活动 a，再选不冲突的下一个最短活动。这个策略似乎也可行，但是很容易找到反例，证明这个策略不正确。

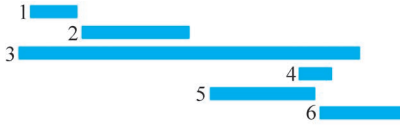

图 5.7　活动安排

如图 5.7 所示的例子，用“策略（1）最早开始时间”，选 3；用“策略（2）最早结束时间”，选 1、2、5、6；

用“策略(3)用时最少”，选 4、1、2。其中策略(2)的结果是最好的。

总结活动安排问题的贪心策略：先按活动的结束时间(区间的右端点)排序，然后每次选结束最早的活动，并保证选择的活动不重叠。

例 5.13　线段覆盖　https://www.luogu.com.cn/problem/P1803

问题描述：有 n 个比赛，每个比赛的开始、结束的时间点已知。yyy 想知道他最多能参加几个比赛。yyy 要参加一个比赛必须善始善终，而且不能同时参加两个及以上的比赛。

输入：第一行是一个整数 n，接下来 n 行，每行是两个整数 L_i、R_i($L_i<R_i$)，表示比赛开始、结束的时间。其中，$1\leqslant n\leqslant 10^6$，$1\leqslant L_i<R_i\leqslant 10^6$。

输出：输出一个整数，表示最多参加的比赛数目。

输入样例：	输出样例：
3 0 2 2 4 1 3	2

按策略(2)编码。

```
n = int(input())
a = []
for _ in range(n):
    L, R = map(int, input().split())
    a.append((L, R))
a.sort(key = lambda x: x[1])        # 按照结束时间排序。请与下一个例题代码比较
ans = 0
lastend = -1
for i in range(n):
    if a[i][0] >= lastend:
        ans += 1
        lastend = a[i][1]
print(ans)
```

3. 区间合并问题

给定若干个区间，合并所有重叠的区间，并返回不重叠的区间的个数。

以图 5.8 为例，1、2、3、5 合并，4、6 合并，新区间是 1′、4′。

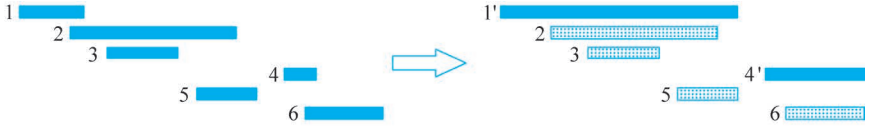

图 5.8　区间合并

贪心策略：按区间的左端点排序，然后逐一枚举每个区间，合并相交的区间。

定义不重叠的区间的个数(答案)为 ans。设当前正在合并的区间的最右端点为 end，当枚举到第 i 个区间[L_i, R_i]时：

若 $L_i\leqslant$end，说明与第 i 个区间相交，需要合并，ans 不变，更新 end=max(end, R_i)。

若 L_i>end，说明与第 i 个区间不相交，ans 加 1，更新 end=max(end，R_i)。

请读者用图 5.8 所示的例子模拟合并过程。

4. 区间覆盖问题

给定一个目标大区间和一些小区间，问最少选择多少小区间可以覆盖大区间。

贪心策略：尽量找出右端点更远的小区间。

操作步骤：先对小区间的左端点排序，然后依次枚举每个小区间，在所有能覆盖当前目标区间右端点的区间中选择右端点最大的区间。

在图 5.9 中，求最少用几个小区间能覆盖整个区间。先按左端点排序。设当前覆盖到了位置 R，选择的小区间的数量为 cnt。

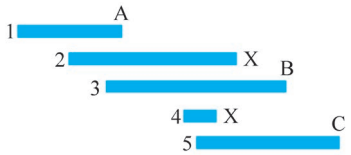

图 5.9 区间覆盖

从区间 1 开始，R 的值是区间 1 的右端点 A，R=A。cnt=1。

找到能覆盖 R=A 的区间 2、3，在区间 2、3 中选右端点更远的 3，更新 R 为区间 3 的右端点 B，R=B。cnt=2。

区间 4 不能覆盖 R=B，跳过。

找到能覆盖 R=B 的区间 5，更新 R=C。cnt=3。结束。

例 5.14 区间覆盖（加强版） https://www.luogu.com.cn/problem/P2082

问题描述：已知有 n 个区间，每个区间的范围是[L_i，R_i]，请求出区间覆盖后的总长。

输入：第一行是一个整数 n，接下来 n 行，每行是两个整数 L_i、R_i（L_i<R_i）。其中，$1 \leqslant n \leqslant 10^5$，$1 \leqslant L_i < R_i \leqslant 10^{17}$。

输出：输出共一行，包含一个正整数，为覆盖后的区间总长。

输入样例：	输出样例：
3 1 100000 200001 1000000 100000000 100000001	900002

按上述贪心策略写出代码。

```
class Data:
    def __init__(self, L, R):
        self.L = L
        self.R = R

n = int(input())
a = []
for _ in range(n):

    L, R = map(int, input().split())
    a.append(Data(L, R))
a.sort(key = lambda x: x.L)
lastend = -1
ans = 0
```

```
for i in range(n):
    if a[i].R >= lastend:
        ans += a[i].R - max(lastend, a[i].L) + 1
        lastend = a[i].R + 1
print(ans)
```

5.5.2 例题

贪心题在算法竞赛中也是必考点，由于贪心法可以灵活地嵌入题目当中与其他算法结合，题目可难、可易。

例 5.15　2023 年第十四届蓝桥杯省赛 Python 大学 C 组试题 E：填充 lanqiaoOJ 3519

时间限制：10s 内存限制：512MB 本题总分：15 分

问题描述：有一个长度为 n 的 01 串 s，其中有一些位置标记为“?”，在这些位置上可以任意填充 0 或者 1。请问如何填充这些位置可以使这个 01 串中出现的互不重叠的 00 和 11 子串最多，输出子串的个数。

输入：输入一行，包含一个字符串。

输出：输出一行，包含一个整数，表示答案。

输入样例：	输出样例：
1110? 0	2

样例说明：如果在问号处填 0，则最多出现一个 00 和一个 11，即 111000。对于所有评测用例，1≤n≤1000000。

本题有贪心、DP 两种解法。DP 解法见第 8 章，这里用贪心求解。

题目要求 00、11 尽可能多，所以目的是尽可能多配对，配对只在相邻 s[i]和 s[i+1]之间发生。从 s[0]、s[1]开始，每次观察相邻的两个字符 s[i]、s[i+1]，讨论以下情况。

(1) s[i]=s[i+1]，这两个字符可能是“00”“11”“??”，都配对。下一次观察 s[i+2]、s[i+3]。

(2) s[i]或 s[i+1]有一个是“?”，那么可以匹配。例如“1?”“?1”“0?”“?0”，都是匹配的。下一次观察 s[i+2]、s[i+3]。

(3) s[i]='0'，s[i+1]='1'，不配对。下一次观察 s[i+1]、s[i+2]。

(4) s[i]='1'，s[i+1]='0'，不配对。下一次观察 s[i+1]、s[i+2]。

下面是代码，只需要计算(1)、(2)即可。在代码中只有一个 while 循环，计算复杂度为 O(n)。

```
s = input()
ans = 0
i = 0
while i < len(s) - 1:
    if s[i] == s[i+1]:
        ans += 1
        i += 2
    elif s[i] == '?' or s[i+1] == '?':
        ans += 1
```

```
        i += 2
    else: i += 1
print(ans)
```

例 5.16　买二赠一 https://www.lanqiao.cn/problems/3539/learning/

问题描述：某商场有 N 件商品，其中第 i 件的价格是 A_i。现在该商场正在进行“买二赠一”的优惠活动，具体规则是每购买两件商品，假设其中较便宜的商品的价格是 P（如果两件商品的价格一样，则 P 等于其中一件商品的价格），就可以从剩余商品中任选一件价格不超过 P/2（向下取整）的商品，免费获得这一件商品。可以通过反复购买两件商品来获得多件免费商品，但是每件商品只能被购买或免费获得一次。小明想知道如果要拿下所有商品（包含购买和免费获得），至少要花费多少钱？

输入：第一行包含一个整数 N。第二行包含 N 个整数，代表 $A_1, A_2, A_3, \cdots, A_N$。

输出：输出一行，包含一个整数，表示答案。

评测用例规模与约定：对于 30% 的数据，$1 \leqslant N \leqslant 20$；对于 100% 的数据，$1 \leqslant N \leqslant 5 \times 10^5$，$1 \leqslant A_i \leqslant 10^9$。

输入样例：	输出样例：
7 1 4 2 8 5 7 1	25

样例说明：小明可以先购买价格为 4 和 8 的商品，免费获得一件价格为 1 的商品；然后买价格为 5 和 7 的商品，免费获得价格为 2 的商品；最后单独购买剩下的一件价格为 1 的商品。总计花费 4+8+5+7+1=25。不存在花费更低的方案。

最贵的商品显然不能免单，买了两个不能免单的最贵的商品后获得一个免单机会，那么这个免单机会给谁呢？给能免单的最贵的那个商品。这个贪心思路显然是对的。

以样例为例，排序得{8 7 5 4 2 1 1}。先购买最贵的 8、7，然后可以免单的最贵的商品是 2。再购买剩下的最贵的商品 5、4，免单 1。最后单独买 1。总价是 25。

需要查找价格为 P/2 的商品，由于价格已经排序，可以用二分法加快查找的时间。Python 有一个**自带的二分函数 bisect_left()**，请读者自行了解 **bisect_left()** 函数并熟悉它的应用。

```
import bisect
n = int(input())
a = list(map(int, input().split()))
a.sort()
ans = 0
cnt = 0
last = -1                                   # 购买的两件商品中便宜的那一件
last_id = n - 1                             # 能免单的位置
vis = [False] * n                           # vis[i] = True 表示已经免单了
for i in range(n - 1, -1, -1):
    if not vis[i]:
        cnt += 1
        ans += a[i]                         # last 是买的第 2 件
        last = a[i]
    if cnt == 2:                            # 买了两件
        cnt = 0
```

```
        x = bisect.bisect_left(a, last // 2, 0, last_id)
# 用二分找能免单的商品 a[x]
        if x > last_id or a[x] > last // 2:x -= 1    # 向下取整
        if x >= 0:
            vis[x] = True                           # x 免单了
            last_id = x - 1                         # 后面能免单的区间范围是[0,last_id]
print(ans)
```

例 5.17　购物　https://www.luogu.com.cn/problem/P1658

问题描述：某人要去购物，现在手上有 n 种不同面值的硬币，每种硬币有无限多个。为了方便购物，他希望带尽量少的硬币，但要能组合出 1 到 x 之间的任意值。

输入：第一行两个数 x、n，第二行 n 个数，表示每种硬币的面值。

输出：输出最少需要携带的硬币的个数，如果无解，输出－1。

评测用例规模与约定：对于 30％的评测用例，$n \leqslant 3$，$x \leqslant 20$；对于 100％的评测用例，$n \leqslant 10$，$x \leqslant 10^3$。

输入样例：	输出样例：
20 4 1 2 5 10	5

为了方便处理，把硬币的面值从小到大排序。

无解是什么情况？如果没有面值为 1 的硬币，组合不到 1，无解。如果有面值为 1 的硬币，那么所有的 x 都能满足，有解。所以，无解的充要条件是没有面值为 1 的硬币。

组合出 1～x 的任意值需要的硬币多吗？学过二进制的人都知道，1、2、4、8、……、2^{n-1} 这 n 个值可以组合出 1～2^n-1 的所有数。这说明只需要很少的硬币就能组合出很大的 x。

设已经组合出 1～s 的面值，即已经得到数字 1、2、3、……、s，下一步扩展到 s＋1。当然，如果能顺便扩展到 s＋2、s＋3、……扩展得越大越好，这样就能用尽量少的硬币扩展出更大的面值。

如何扩展到 s＋1？就是在数字 1、2、3、……、s 的基础上添加一个面值为 v 的硬币，得到 s＋1。v 可以选 1、2、……、s＋1，例如 v＝1，s＋1＝s＋v；v＝2，s＋1＝s－1＋v；……；v＝s＋1，s＋1＝v。如果 v＝s＋2，就不能组合到 s＋1 了。

v 的取值范围是[1，s＋1]，为了最大扩展，选 v 为[1，s＋1]内的最大硬币，此时 s 扩展到 s＋v。这就是贪心策略。

以本题的输入样例为例说明计算过程。设答案为 ans。

先选硬币 1，得到 s＝1。ans＝1。

再选[1，s＋1]＝[1，2]内的最大硬币 2，扩展 s＝1 为 s＝1＋v＝3。ans＝2。

再选[1，s＋1]＝[1，4]内的最大硬币 2，得到 s＝5。ans＝3。

再选[1，s＋1]＝[1，6]内的最大硬币 5，得到 s＝10。ans＝4。

再选[1，s＋1]＝[1，11]内的最大硬币 10，得到 s＝20。ans＝5。此时 s≥x，结束。

所以仅需要 5 个面值为 1、2、2、5、10 的硬币就可以组合得到 1、2、3、4、……、20。

```
x, n = map(int, input().split())
a = list(map(int, input().split()))
a.sort()
```

```
if a[0] != 1: print(-1); exit()
s, ans = 0, 0
while s < x:
    for v in range(n - 1, -1, -1):
        if a[v] <= s + 1:
            s += a[v]
            ans += 1
            break
print(ans)
```

例 5.18　最大团 http://oj.ecustacm.cn/problem.php?id=1762

问题描述：数轴上有 n 个点，第 i 个点的坐标为 x_i，权值为 w_i。两个点 i、j 之间存在一条边当且仅当 $abs(x_i-x_j)\geqslant w_i+w_j$。请求出这张图的最大团的点数。团是两两之间存在边的定点集合。

输入：输入的第一行为 n，$n\leqslant 200000$。接下来 n 行，每行两个整数 x_i、w_i，$0\leqslant|x_i|$，$w_i\leqslant 10^9$。

输出：输出一行，包含一个整数，表示最大团的点数。

输入样例：	输出样例：
4 2 3 3 1 6 1 0 2	3

最大团是一个图论问题：在一个无向图中找出一个点数最多的完全子图。所谓完全图，就是图中所有的点之间都有边。n 个点互相连接，共有 n(n−1)/2 条边。

普通图上的最大团问题是 NP 问题，计算复杂度是指数级的。例如常见的 Bron-Kerbosch 算法是一个暴力搜索算法，复杂度为 $O(3^{n/3})$。所以如果出最大团的题目，一般不会在普通图上求最大团，而是在一些特殊的图上，用巧妙的、非指数复杂度的算法求最大团。本题 $n\leqslant 200000$，只能用复杂度小于 $O(n\log_2 n)$的巧妙算法。

本题的图比较特殊，所有的点都在一条直线上。在样例中，存在的边有(0-3)、(0-6)、(2-6)、(3-6)，其中 0、3、6 这 3 点之间都有边，它们构成了一个最大团。

另外，本题对边的定义也很奇怪："两个点 i、j 之间存在一条边当且仅当 $abs(x_i-x_j)\geqslant w_i+w_j$"。

考虑以下两个问题：

(1) 哪些点之间有边？题目的定义是 $abs(x_i-x_j)\geqslant w_i+w_j$，若事先把 x 排序，设 $x_j\geqslant x_i$，移位得 $x_j-w_j\geqslant x_i+w_i$。这样就把每个点的 x 和 w 统一起来，方便计算。

(2) 哪些点构成了团？是最大团吗？考察 3 个点 $x_1\leqslant x_2\leqslant x_3$，若 x_1 和 x_2 有边，则应有 $x_1+w_1\leqslant x_2-w_2$；若 x_2 和 x_3 有边，则 $x_2+w_2\leqslant x_3-w_3$。推导 x_1 和 x_3 的关系：$x_1+w_1\leqslant x_2-w_2\leqslant x_2+w_2\leqslant x_3-w_3$，即 $x_1+w_1\leqslant x_3-w_3$，说明 x_1 和 x_3 也有边。x_1、x_2、x_3 这 3 点

之间都有边，它们构成了一个团。依次这样操作，符合条件的 x_1、x_2、x_3、……构成了一个团。但是，用这个方法得到的团是最大的吗？

为了方便思考，把上述讨论画成图，如图 5.10 所示。

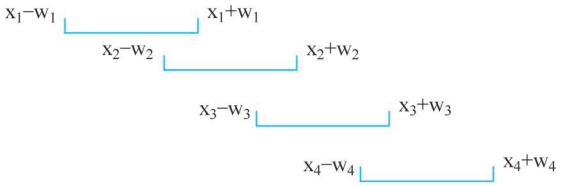

图 5.10　最大团建模

把每个点的信息画成线段，左端点是 x－w，右端点是 x＋w。问题建模：在 n 个线段中，找出最多的线段，使得它们互相不交叉。

读者学过贪心，发现它是经典的“活动安排问题”，或者称为“区间调度问题”，即在 n 个活动中安排尽量多的活动，使得它们互相不冲突。其贪心解法如下：

（1）按活动的结束时间排序。本题按 x＋w 排序。

（2）选择第一个结束的活动，跳过与它时间冲突的活动。

（3）重复（2），直到活动为空。每次选择剩下活动中最早结束的活动，并跳过与它冲突的活动。

下面的代码，排序的计算复杂度为 $O(n\log_2 n)$，贪心的计算复杂度为 $O(n)$，总复杂度为 $O(n\log_2 n)$。

```
n = int(input())
a = []
for i in range(n):
    x, w = map(int, input().split())
    a.append({'l': x-w, 'r': x+w})
a.sort(key=lambda x: x['r'])            # 按右端点排序
R, ans = a[0]['r'], 1
for i in range(1, n):                   # 选剩下活动中最早结束的活动，跳过冲突的活动
    if a[i]['l'] >= R:
        R = a[i]['r']
        ans += 1
print(ans)
```

例 5.19　奶牛优惠券　https://www.luogu.com.cn/problem/solution/P3045

问题描述：农夫约翰需要新奶牛。目前有 N 头奶牛出售，农夫的预算只有 M 元，奶牛 i 花费 P_i。但是农夫有 K 张优惠券，当对奶牛 i 使用优惠券时只需要花费 C_i（$C_i \leqslant P_i$）。每头奶牛只能使用一张优惠券。求农夫最多可以养多少头奶牛？

输入：第一行 3 个正整数 N、K、M，$1 \leqslant N \leqslant 50000$，$1 \leqslant M \leqslant 10^{14}$，$1 \leqslant K \leqslant N$。接下来 N 行，每行两个整数 P_i 和 C_i，$1 \leqslant P_i \leqslant 10^9$，$1 \leqslant C_i \leqslant P_i$。

输出：输出一个整数，表示答案。

输入样例：	输出样例：
4 1 7 3 2 2 2 8 1 4 3	3

题意简述如下：有 n 个数，每个数可以替换为较小的数；从 n 个数中选出一些数，在选的时候允许最多替换 k 个，要求这些数相加不大于 m，问最多能选出多少个数。

这个问题可以用女生买衣服类比。女生带着 m 元去买衣服，目标是尽量多买几件，越多越好。每件衣服都有优惠，但是必须使用优惠券，一件衣服只能用一张优惠券。衣服的优惠幅度不一样，有可能原价贵的优惠后反而更便宜。女生有 k 张优惠券，问她最多能买多少件衣服。

男读者可以问一问女朋友，她会怎么买衣服。聪明的她可能会马上问：优惠券是不是无限多？如果优惠券用不完，那么衣服的原价就形同虚设，按优惠价从小到大买就行。可惜优惠券总是不够用。

她想出了这个方法：按优惠价排序，先买优惠价便宜的，直到用完优惠券；如果还有钱，再买原价便宜的。

但是这个方法不是最优的，因为优惠价格低的可能优惠幅度小，导致优惠券被浪费了。例如：

衣　服：a，b，c，d，e

优惠价：3，4，5，6，7

原　价：4，5，6，15，10

设有 m＝20 元，k＝3 张优惠券。把 3 张优惠券用在 a、b、c 上并不是最优的，这样只能买 3 件衣服。最优解是买 4 件衣服：a、b、d 用优惠价，c 用原价，共 19 元。

下面对这个方法进行改进。既然有优惠幅度很大的衣服，就试一试把优惠券转移到这件衣服上，看能不能获得更大的优惠。把这次转移称为“反悔”。

设优惠价最便宜的前 k 件衣服用完了 k 张优惠券。现在看第 i＝k＋1 件衣服，要么用原价买，要么转移一张优惠券过来用优惠价买，看哪种结果更好。设原价是 p，优惠价是 c。

在反悔前，第 i 件用原价 p_i 买，前面第 j 件用优惠价 c_j 买，共花费 $tot+p_i+c_j$，其中 tot 是其他已经买的衣服的花费。

在反悔后，第 j 件把优惠券转给第 i 件，改成原价 p_j，第 i 件用优惠价 c_i，共花费 $tot+c_i+p_j$。

如果反悔更好，则有 $tot+p_i+c_j<tot+c_j+p_i$，即 $p_j-c_j<p_i-c_i$，设 $\Delta=p-c$，有 $\Delta_j<\Delta_i$，Δ 是原价和优惠价的差额。

也就是说，只要在使用优惠券的衣服中存在一个 j 有 $\Delta_j<\Delta_i$，即第 j 件的优惠幅度不如第 i 件的优惠幅度，那么把 j 的优惠券转给 i 会有更好的结果。

但是上述讨论还是有问题，它可能导致超过总花费。例如：

衣　服：a，　b，　c

优惠价：20，40，42

原　价：30，80，49

m＝69 元，k＝1 张优惠券。先用优惠券买 a；下一步发现 $\Delta_a < \Delta_b$，把优惠券转移给 b，现在的花费是 30＋40＝70，超过 m 了。最优解是 a 仍然用优惠价，c 用原价。所以简单地计算候选衣服 i 的差额 $\Delta_i = p_i - c_i$，然后与 Δ_j 比较并不行。

那么如何在候选衣服中选一件才能最优惠呢？

（1）用原价买，那么应该是这些衣服中的最低原价。

（2）用优惠价买，那么应该是这些衣服中的最低优惠价。

所以在候选衣服中的最低原价和最低优惠价之间计算差额，并与 Δ_j 比较才是有意义的。

在编码时，用 3 个优先队列处理 3 个关键数据：

（1）已使用优惠券的衣服的优惠幅度 Δ。在已经使用优惠券的衣服中，谁应该拿出来转移优惠券？应该是那个优惠幅度 Δ 最小的，这样转移之后才能获得更大优惠。用优先队列 d 找最小的 Δ。

（2）没使用优惠券的衣服的原价。用一个优先队列 p 找最便宜的原价。

（3）没使用优惠券的衣服的优惠价。用一个优先队列 c 找最便宜的优惠价。

在代码中这样处理优惠券：

（1）先用完 k 个优惠券，从 c 中连续取出 k 个最便宜的即可。

（2）优惠券替换。从 p 中取出原价最便宜的 p_1，从 c 中取出优惠价最便宜的 c_2，然后从 d 中取出优惠幅度最小的 d：

① 若 $d > p_1 - c_2$，说明替换优惠券不值得，不用替换。下一件衣服用原价 p_1 买。

② 若 $d \leqslant p_1 - c_2$，说明替换优惠券值得，下一件衣服用优惠价 c_2 买，原来用优惠券的改成用原价。

本题总体上是贪心，用 3 个优先队列处理贪心，但是优惠券的替换操作是贪心的“反悔”，所以称为“反悔贪心”。贪心是连续做局部最优操作，但是有时局部最优推不出全局最优，此时可以用反悔贪心，撤销之前做出的决策，换条路重新贪心。

```
import heapq
n, k, m = map(int, input().split())
p = [0] * (n+1)
c = [0] * (n+1)
buy = [False] * (n+1)                    # buy[i]=1：第 i 个物品被买了
P = []
C = []
D = []
for i in range(1, n+1):
    p[i], c[i] = map(int, input().split())
    heapq.heappush(P, (p[i], i))
                    # 原价，还没买的在这里，如果买了就弹出去
    heapq.heappush(C, (c[i], i))
                    # 优惠价，还没买的在这里，如果买了就弹出去
for i in range(1, k+1):
    heapq.heappush(D, 0)                 # k 张优惠券，开始时每个的优惠为 0
ans = 0
while P and C:
    p1 = P[0]                            # 取出原价最便宜的
    c2 = C[0]                            # 取出优惠价最便宜的
```

```
    if buy[p1[1]]:                          # 这个已经买了,跳过
        heapq.heappop(P); continue
    if buy[c2[1]]:                          # 这个已经买了,跳过
        heapq.heappop(C); continue
    if D[0] > p1[0] - c2[0]:                # 用原价买 i 更划算,不用替换优惠券
        m -= p1[0]                          # 买原价最便宜的
        heapq.heappop(P)                    # 这里不要 C.pop(),因为买的是 p1,不是 c2
        buy[p1[1]] = True                   # 标记 p1 买了
    else:                                   # 替换优惠券。前 k 个都先执行这里
        m -= c2[0] + D[0]                   # 买优惠价最便宜的
        heapq.heappop(C)                    # 这里不要 p.pop(),因为买的是 c2,不是 p1
        buy[c2[1]] = True                   # 标记 c2 买了
        heapq.heappop(D)                    # 原来用优惠券的退回优惠券
        heapq.heappush(D, p[c2[1]] - c[c2[1]])
                    # c2 使用优惠券,重新计算 delta 并进队列
    if m >= 0:ans += 1
    else:break
print(ans)
```

【练习题】

lanqiaoOJ：三国游戏 3518、平均 3532、答疑 1025、身份证 3849、找零钱 3854、01 搬砖 2201、卡牌游戏 1057、寻找和谐音符 3975、小蓝的旅行计划 3534、翻硬币 209、防御力 226。

洛谷：排座椅 P1056、母舰 P2813、排队接水 P1223、小 A 的糖果 P3817、加工生产调度 P1248、负载平衡问题 P4016。

5.6 扩展学习

从本章开始进入了算法学习阶段。本章的“基本算法”是一些“通用”的算法，可以在很多场景下应用。

本书 2.1 节中曾提到计算思维的作用，通过做杂题可以帮助大家建立基本的、自发的计算思维。在计算机科学中有大量经典的算法和数据结构，它们代表了计算机科学中最璀璨的花朵。在这些知识点中基本算法易于理解、适用面广、精巧高效，是计算思维的奠基，真正的计算思维从这里开始。

除了本章介绍的前缀和、差分、二分、贪心，基本算法还有尺取法、倍增法、离散化、分治等。在逐渐深入学习算法的过程中，这些基本算法都是必须掌握的知识点。