

?? (Table of Contents)

1. Basic - OI Wiki	1
2. Digit - OI Wiki	8
3. Interval - OI Wiki	10
4. Knapsack - OI Wiki	13
5. Probability - OI Wiki	26
6. State - OI Wiki	36
7. Tree - OI Wiki	41
8. 2 Sat - OI Wiki	48
9. Bipartite - OI Wiki	58
10. Concept - OI Wiki	60
11. Flow - OI Wiki	71
12. Mst - OI Wiki	73
13. Shortest Path - OI Wiki	101
14. Store - OI Wiki	119
15. Strongly Connected Components - OI Wiki	121
16. Topological - OI Wiki	123
17. Traverse - OI Wiki	125
18. Binary - OI Wiki	127
19. Divide And Conquer - OI Wiki	139
20. Greedy - OI Wiki	148

动态规划基础

本页面主要介绍了动态规划的基本思想，以及动态规划中状态及状态转移方程的设计思路，帮助各位初学者对动态规划有一个初步的了解。

本部分的其他页面，将介绍各种类型问题中动态规划模型的建立方法，以及一些动态规划的优化技巧。

引入



[IOI1994] 数字三角形

给定一个 r 行的数字三角形 ($r \leq 1000$)，需要找到一条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。每一步可以走到当前点左下方的点或右下方的点。

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

在上面这个例子中，最优路径是 $7 \rightarrow 3 \rightarrow 8 \rightarrow 7 \rightarrow 5$ 。

最简单粗暴的思路是尝试所有的路径。因为路径条数是 $O(2^r)$ 级别的，这样的做法无法接受。

注意到这样一个事实，一条最优的路径，它的每一步决策都是最优的。

以例题里提到的最优路径为例，只考虑前四步 $7 \rightarrow 3 \rightarrow 8 \rightarrow 7$ ，不存在一条从最顶端到 4 行第 2 个数的权值更大的路径。

而对于每一个点，它的下一步决策只有两种：往左下角或者往右下角（如果存在）。因此只需要记录当前点的最大权值，用这个最大权值执行下一步决策，来更新后续点的最大权值。

这样做还有一个好处：我们成功缩小了问题的规模，将一个问题分成了多个规模更小的问题。要想得到从顶端到第 r 行的最优方案，只需要知道从顶端到第 $r - 1$ 行的最优方案的信息就可以了。

这时候还存在一个问题：子问题间重叠的部分会有很多，同一个子问题可能会被重复访问多次，效率还是不高。解决这个问题的方法是把每个子问题的解存储下来，通过记忆化的方式限制访问顺序，确保每个子问题只被访问一次。

上面就是动态规划的一些基本思路。下面将会更系统地介绍动态规划的思想。

动态规划原理

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 (参见: [DAG 上的 DP](#))。

最长公共子序列



最长公共子序列问题



给定一个长度为 n 的序列 A 和一个长度为 m 的序列 B ($n, m \leq 5000$), 求出一个最长的序列, 使得该序列既是 A 的子序列, 也是 B 的子序列。

子序列的定义可以参考 [子序列](#)。一个简要的例子: 字符串 `abcde` 与字符串 `acde` 的公共子序列有 `a`、`c`、`d`、`e`、`ac`、`ad`、`ae`、`cd`、`ce`、`de`、`ade`、`ace`、`cde`、`acde`, 最长公共子序列的长度是 4。

设 $f(i, j)$ 表示只考虑 A 的前 i 个元素, B 的前 j 个元素时的最长公共子序列的长度, 求这时的最长公共子序列的长度就是 **子问题**。 $f(i, j)$ 就是我们所说的 **状态**, 则 $f(n, m)$ 是最终要达到的状态, 即为所求结果。

对于每个 $f(i, j)$, 存在三种决策: 如果 $A_i = B_j$, 则可以将它接到公共子序列的末尾; 另外两种决策分别是跳过 A_i 或者 B_j 。状态转移方程如下:

$$f(i, j) = \begin{cases} f(i - 1, j - 1) + 1 & A_i = B_j \\ \max(f(i - 1, j), f(i, j - 1)) & A_i \neq B_j \end{cases}$$

可参考 [SourceForge 的 LCS 交互网页](#) 来更好地理解 LCS 的实现过程。

该做法的时间复杂度为 $O(nm)$ 。

另外, 本题存在 $O\left(\frac{nm}{w}\right)$ 的算法¹。有兴趣的同学可以自行探索。

```
1 int a[MAXN], b[MAXM], f[MAXN][MAXM];
2
3 int dp() {
4     for (int i = 1; i <= n; i++)
5         for (int j = 1; j <= m; j++)
6             if (a[i] == b[j])
7                 f[i][j] = f[i - 1][j - 1] + 1;
8             else
9                 f[i][j] = std::max(f[i - 1][j], f[i][j - 1]);
10            return f[n][m];
11 }
```

最长不下降子序列

最长不下降子序列问题

给定一个长度为 n 的序列 A ($n \leq 5000$)，求出一个最长的 A 的子序列，满足该子序列的后一个元素不小于前一个元素。

算法一

设 $f(i)$ 表示以 A_i 为结尾的最长不下降子序列的长度，则所求为 $\max_{1 \leq i \leq n} f(i)$ 。

计算 $f(i)$ 时，尝试将 A_i 接到其他的最长不下降子序列后面，以更新答案。于是可以写出这样的状态转移方程： $f(i) = \max_{1 \leq j < i, A_j \leq A_i} (f(j) + 1)$ 。

容易发现该算法的时间复杂度为 $O(n^2)$ 。

C++

```
1 int a[MAXN], d[MAXN];
2
3 int dp() {
4     d[1] = 1;
5     int ans = 1;
6     for (int i = 2; i <= n; i++) {
7         d[i] = 1;
8         for (int j = 1; j < i; j++)
9             if (a[j] <= a[i]) {
10                 d[i] = max(d[i], d[j] + 1);
11                 ans = max(ans, d[i]);
12             }
13     }
14     return ans;
15 }
```

Python

```
1 a = [0] * MAXN
2 d = [0] * MAXN
3
4 def dp():
5     d[1] = 1
6     ans = 1
7     for i in range(2, n + 1):
8         for j in range(1, i):
9             if a[j] <= a[i]:
10                 d[i] = max(d[i], d[j] + 1)
11                 ans = max(ans, d[i])
12
13 return ans
```



算法二²

当 n 的范围扩大到 $n \leq 10^5$ 时，第一种做法就不够快了，下面给出了一个 $O(n \log n)$ 的做法。

回顾一下之前的状态： (i, l) 。

但这次，我们不是要按照相同的 i 处理状态，而是直接判断合法的 (i, l) 。

再看一下之前的转移： $(j, l - 1) \rightarrow (i, l)$ ，就可以判断某个 (i, l) 是否合法。

初始时 $(1, 1)$ 肯定合法。

那么，只需要找到一个 l 最大的合法的 (i, l) ，就可以得到最终最长不下降子序列的长度了。

那么，根据上面的方法，我们就需要维护一个可能的转移列表，并逐个处理转移。

所以可以定义 $a_1 \dots a_n$ 为原始序列， d_i 为所有的长度为 i 的不下降子序列的末尾元素的最小值， len 为子序列的长度。

初始化： $d_1 = a_1, len = 1$ 。

现在我们已知最长的不下降子序列长度为 1，那么我们让 i 从 2 到 n 循环，依次求出前 i 个元素的最长不下降子序列的长度，循环的时候我们只需要维护好 d 这个数组还有 len 就可以了。**关键在于如何维护。**

考虑进来一个元素 a_i ：

1. 元素大于等于 d_{len} ，直接将该元素插入到 d 序列的末尾。
2. 元素小于 d_{len} ，找到 **第一个** 大于它的元素，用 a_i 替换它。

为什么：

- 对于步骤 1：

由于我们是从前往后扫，所以说当元素大于等于 d_{len} 时一定会有一个不下降子序列使得这个不下降子序列的末项后面可以接这个元素。如果 d 不接这个元素，可以发现既不符合定义，又不是最优解。

- 对于步骤 2：

同步骤 1，如果插在 d 的末尾，那么由于前面的元素大于要插入的元素，所以不符合 d 的定义，因此必须先找到 **第一个** 大于它的元素，再用 a_i 替换。

步骤 2 如果采用暴力查找，则时间复杂度仍然是 $O(n^2)$ 的。但是根据 d 数组的定义，又由于本题要求不下降子序列，所以 d 一定是 **单调不减** 的，因此可以用二分查找将时间复杂度降至 $O(n \log n)$ 。

参考代码如下：

C++

```
1 for (int i = 0; i < n; ++i) scanf("%d", a + i);
2 memset(dp, 0x1f, sizeof dp);
3 mx = dp[0];
4 for (int i = 0; i < n; ++i) {
5     *std::upper_bound(dp, dp + n, a[i]) = a[i];
6 }
7 ans = 0;
8 while (dp[ans] != mx) ++ans;
```

Python

```
1 dp = [0x1F1F1F1F] * MAXN
2 mx = dp[0]
3 for i in range(0, n):
4     bisect.insort_left(dp, a[i], 0, len(dp))
5 ans = 0
6 while dp[ans] != mx:
7     ans += 1
```

⚠ 注意

对于最长 **上升** 子序列问题，类似地，可以令 d_i 表示所有长度为 i 的最长上升子序列的末尾元素的最小值。

需要注意的是，在步骤 2 中，若 $a_i \leq d_{len}$ ，由于最长上升子序列中相邻元素不能相等，需要在 d 序列中找到 **第一个不小于** a_i 的元素，用 a_i 替换之。

在实现上（以 C++ 为例），需要将 `upper_bound` 函数改为 `lower_bound`。

参考资料与注释

1. 位运算求最长公共子序列 - -Wallace- - 博客园 ↪
2. 最长不下降子序列 nlogn 算法详解 - lvmememe - 博客园 ↪

⌚ 本页面最近更新：2025/5/3 19:43:25，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[lr1d](#), [Chrogeek](#), [ChungZH](#), [ouuan](#), [Xeonacid](#), [CBW2007](#), [ksyx](#), [Marcythm](#), [StudyingFather](#), [tptpp](#), [Enter-tainer](#), [greyqz](#), [HeRaNO](#), [hhc0001](#), [hsfzLZH1](#), [partychicken](#), [Persdre](#), [xhn16729](#), [XiaoSuan250](#), [xyf007](#), [zhb2000](#), [c-forrest](#), [caoji2001](#), [dong628](#), [iamtwz](#), [LincolnYe](#), [Menci](#), [NachtgeistW](#), [ree-chee](#), [shawlleyw](#), [shuzhouliu](#), [Taoran-01](#), [Taoran_01](#), [Tiphereth-A](#), [TrisolarisHD](#), [WAAutoMaton](#), [xhn16729](#), [yusancy](#), [ZhangZhanhaoxiang](#),

[zchen20](#), [zzhx2006](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





区间 DP

定义

区间类动态规划是线性动态规划的扩展，它在分阶段地划分问题时，与阶段中元素出现的顺序和由前一阶段的哪些元素合并而来有很大的关系。

令状态 $f(i, j)$ 表示将下标位置 i 到 j 的所有元素合并能获得的价值的最大值，那么 $f(i, j) = \max\{f(i, k) + f(k + 1, j) + cost\}$, $cost$ 为将这两组元素合并起来的价值。

性质

区间 DP 有以下特点：

合并：即将两个或多个部分进行整合，当然也可以反过来；

特征：能将问题分解为能两两合并的形式；

求解：对整个问题设最优值，枚举合并点，将问题分解为左右两个部分，最后合并两个部分的最优值得到原问题的最优值。

解释

例题



「NOI1995」石子合并



题目大意：在一个环上有 n 个数 a_1, a_2, \dots, a_n ，进行 $n - 1$ 次合并操作，每次操作将相邻的两堆合并成一堆，能获得新的一堆中的石子数量的和的得分。你需要最大化你的得分。

需要考虑不在环上，而在一条链上的情况。

令 $f(i, j)$ 表示将区间 $[i, j]$ 内的所有石子合并到一起的最大得分。

写出 **状态转移方程**： $f(i, j) = \max\{f(i, k) + f(k + 1, j) + \sum_{t=i}^j a_t\}$ ($i \leq k < j$)

令 sum_i 表示 a 数组的前缀和，状态转移方程变形为

$f(i, j) = \max\{f(i, k) + f(k + 1, j) + sum_j - sum_{i-1}\}$ 。

怎样进行状态转移

由于计算 $f(i, j)$ 的值时需要知道所有 $f(i, k)$ 和 $f(k + 1, j)$ 的值，而这两个中包含的元素的数量都小于 $f(i, j)$ ，所以我们以 $len = j - i + 1$ 作为 DP 的阶段。首先从小到大枚举 len ，然后枚举 i 的值，根据 len 和 i 用公式计算出 j 的值，然后枚举 k ，时间复杂度为 $O(n^3)$

怎样处理环

题目中石子围成一个环，而不是一条链，怎么办呢？

方法一：由于石子围成一个环，我们可以枚举分开的位置，将这个环转化成一个链，由于要枚举 n 次，最终的时间复杂度为 $O(n^4)$ 。

方法二：我们将这条链延长两倍，变成 $2 \times n$ 堆，其中第 i 堆与第 $n + i$ 堆相同，用动态规划求解后，取 $f(1, n), f(2, n + 1), \dots, f(n, 2n - 1)$ 中的最优值，即为最后的答案。时间复杂度 $O(n^3)$ 。

实现

C++

```
1 for (len = 2; len <= n; len++)  
2     for (i = 1; i <= 2 * n - len; i++) {  
3         int j = len + i - 1;  
4         for (k = i; k < j; k++)  
5             f[i][j] = max(f[i][j], f[i][k] + f[k + 1][j] + sum[j] - sum[i - 1]);  
6     }
```

Python

```
1 for len in range(2, n + 1):  
2     for i in range(1, 2 * n - len + 1):  
3         j = len + i - 1  
4         for k in range(i, j):  
5             f[i][j] = max(f[i][j], f[i][k] + f[k + 1][j] + sum[j] - sum[i - 1])
```

几道练习题

[NOIP 2006 能量项链](#)

[NOIP 2007 矩阵取数游戏](#)

[「IOI2000」邮局](#)

 本页面最近更新：2025/8/30 13:34:30，[更新历史](#)

 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

 本页面贡献者：[Ir1d](#), [Henry-ZHR](#), [hsfzLZH1](#), [iamtwz](#), [ouuan](#), [Xeonacid](#), [AFOBJECT](#), [billchenchina](#), [Chlero](#), [EarlyOvO](#), [Enter-tainer](#), [fyulingi](#), [greyqz](#), [HeRaNO](#), [ImpleLee](#), [isdanni](#), [ksyx](#), [Menci](#), [OYoooooo](#), [partychicken](#), [shawlleyw](#), [shenshuaijie](#), [StudyingFather](#), [thredreams](#), [Wang Hongtian](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用

背包 DP

前置知识：[动态规划部分简介](#)。



引入

在具体讲解何为「背包 dp」前，先来看如下的例题：



「USACO07 DEC」 Charm Bracelet



题意概要：有 n 个物品和一个容量为 W 的背包，每个物品有重量 w_i 和价值 v_i 两种属性，要求选若干物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。

在上述例题中，由于每个物体只有两种可能的状态（取与不取），对应二进制中的 0 和 1，这类问题便被称为「0-1 背包问题」。

0-1 背包

解释

例题中已知条件有第 i 个物品的重量 w_i ，价值 v_i ，以及背包的总容量 W 。

设 DP 状态 $f_{i,j}$ 为在只能放前 i 个物品的情况下，容量为 j 的背包所能达到的最大总价值。

考虑转移。假设当前已经处理好了前 $i - 1$ 个物品的所有状态，那么对于第 i 个物品，当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变，故这种情况的最大价值为 $f_{i-1,j}$ ；当其放入背包时，背包的剩余容量会减小 w_i ，背包中物品的总价值会增大 v_i ，故这种情况的最大价值为 $f_{i-1,j-w_i} + v_i$ 。

由此可以得出状态转移方程：

$$f_{i,j} = \max(f_{i-1,j}, f_{i-1,j-w_i} + v_i)$$

这里如果直接采用二维数组对状态进行记录，会出现 MLE。可以考虑改用滚动数组的形式来优化。

由于对 f_i 有影响的只有 f_{i-1} ，可以去掉第一维，直接用 f_i 来表示处理到当前物品时背包容量为 i 的最大价值，得出以下方程：

$$f_j = \max(f_j, f_{j-w_i} + v_i)$$

务必牢记并理解这个转移方程，因为大部分背包问题的转移方程都是在此基础上推导出来的。

实现

还有一点需要注意的是，很容易写出这样的 错误核心代码：

C++

```
1 for (int i = 1; i <= n; i++)
2     for (int l = 0; l <= W - w[i]; l++)
3         f[l + w[i]] = max(f[l] + v[i], f[l + w[i]]);
4 // 由 f[i][l + w[i]] = max(max(f[i - 1][l + w[i]], f[i - 1][l] + w[i]),
5 // f[i][l + w[i]]); 简化而来
```

Python

```
1 for i in range(1, n + 1):
2     for l in range(0, W - w[i] + 1):
3         f[l + w[i]] = max(f[l] + v[i], f[l + w[i]])
4 # 由 f[i][l + w[i]] = max(max(f[i - 1][l + w[i]], f[i - 1][l] + w[i]),
5 # f[i][l + w[i]]); 简化而来
```

这段代码哪里错了呢？枚举顺序错了。

仔细观察代码可以发现：对于当前处理的物品 i 和当前状态 $f_{i,j}$ ，在 $j \geq w_i$ 时， $f_{i,j}$ 是会被 $f_{i,j-w_i}$ 所影响的。这就相当于物品 i 可以多次被放入背包，与题意不符。（事实上，这正是完全背包问题的解法）

为了避免这种情况发生，我们可以改变枚举的顺序，从 W 枚举到 w_i ，这样就不会出现上述的错误，因为 $f_{i,j}$ 总是在 $f_{i,j-w_i}$ 前被更新。

因此实际核心代码为

C++

```
1 for (int i = 1; i <= n; i++)
2     for (int l = W; l >= w[i]; l--) f[l] = max(f[l], f[l - w[i]] + v[i]);
```

Python

```
1 for i in range(1, n + 1):
2     for l in range(W, w[i] - 1, -1):
3         f[l] = max(f[l], f[l - w[i]] + v[i])
```



例题代码

```
1 #include <iostream>
2 using namespace std;
3 constexpr int MAXN = 13010;
4 int n, W, w[MAXN], v[MAXN], f[MAXN];
5
6 int main() {
7     cin >> n >> W;
8     for (int i = 1; i <= n; i++) cin >> w[i] >> v[i]; // 读入数据
9     for (int i = 1; i <= n; i++)
10         for (int l = W; l >= w[i]; l--)
11             if (f[l - w[i]] + v[i] > f[l]) f[l] = f[l - w[i]] + v[i];
12     // 状态方程
13     cout << f[W];
14     return 0;
}
```

完全背包

解释

完全背包模型与 0-1 背包类似，与 0-1 背包的区别仅在于一个物品可以选取无限次，而非仅能选取一次。

我们可以借鉴 0-1 背包的思路，进行状态定义：设 $f_{i,j}$ 为只能选前 i 个物品时，容量为 j 的背包可以达到的最大价值。

需要注意的是，虽然定义与 0-1 背包类似，但是其状态转移方程与 0-1 背包并不相同。

过程

可以考虑一个朴素的做法：对于第 i 件物品，枚举其选了多少个来转移。这样做的时间复杂度是 $O(n^3)$ 的。

状态转移方程如下：

$$f_{i,j} = \max_{k=0}^{+\infty} (f_{i-1,j-k \times w_i} + v_i \times k)$$

考虑做一个简单的优化。可以发现，对于 $f_{i,j}$ ，只要通过 $f_{i,j-w_i}$ 转移就可以了。因此状态转移方程为：

$$f_{i,j} = \max(f_{i-1,j}, f_{i,j-w_i} + v_i)$$

理由是当我们这样转移时， $f_{i,j-w_i}$ 已经由 $f_{i,j-2 \times w_i}$ 更新过，那么 $f_{i,j-w_i}$ 就是充分考虑了第 i 件物品所选次数后得到的最优结果。换言之，我们通过局部最优子结构的性质重复使用了之前的枚举过程，优化了枚举的复杂度。

与 0-1 背包相同，我们可以将第一维去掉来优化空间复杂度。如果理解了 0-1 背包的优化方式，就不难明白压缩后的循环是正向的（也就是上文中提到的错误优化）。

「Luogu P1616」 疯狂的采药

题意概要：有 n 种物品和一个容量为 W 的背包，每种物品有重量 w_i 和价值 v_i 两种属性，要求选出若干个物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。

例题代码

```
1 #include <iostream>
2 using namespace std;
3 constexpr int MAXN = 1e4 + 5;
4 constexpr int MAXW = 1e7 + 5;
5 int n, W, w[MAXN], v[MAXN];
6 long long f[MAXW];
7
8 int main() {
9     cin >> W >> n;
10    for (int i = 1; i <= n; i++) cin >> w[i] >> v[i];
11    for (int i = 1; i <= n; i++)
12        for (int l = w[i]; l <= W; l++)
13            if (f[l - w[i]] + v[i] > f[l]) f[l] = f[l - w[i]] + v[i];
14    // 核心状态方程
15    cout << f[W];
16    return 0;
}
```

多重背包

多重背包也是 0-1 背包的一个变式。与 0-1 背包的区别在于每种物品有 k_i 个，而非一个。

一个很朴素的想法就是：把「每种物品选 k_i 次」等价转换为「有 k_i 个相同的物品，每个物品选一次」。这样就转换成了一个 0-1 背包模型，套用上文所述的方法就可已解决。状态转移方程如下：

$$f_{i,j} = \max_{k=0}^{k_i} (f_{i-1,j-k \times w_i} + v_i \times k)$$

时间复杂度 $O(W \sum_{i=1}^n k_i)$ 。

核心代码

```
1 for (int i = 1; i <= n; i++) {  
2     for (int weight = W; weight >= w[i]; weight--) {  
3         // 多遍历一层物品数量  
4         for (int k = 1; k * w[i] <= weight && k <= cnt[i]; k++) {  
5             dp[weight] = max(dp[weight], dp[weight - k * w[i]] + k *  
6             v[i]);  
7         }  
8     }  
}
```

二进制分组优化

考虑优化。我们仍考虑把多重背包转化成 0-1 背包模型来求解。

解释

显然，复杂度中的 $O(nW)$ 部分无法再优化了，我们只能从 $O(\sum k_i)$ 处入手。为了表述方便，我们用 $A_{i,j}$ 代表第 i 种物品拆分出的第 j 个物品。

在朴素的做法中， $\forall j \leq k_i$ ， $A_{i,j}$ 均表示相同物品。那么我们效率低的原因主要在于我们进行了大量重复性的工作。举例来说，我们考虑了「同时选 $A_{i,1}, A_{i,2}$ 」与「同时选 $A_{i,2}, A_{i,3}$ 」这两个完全等效的情况。这样的重复性工作我们进行了许多次。那么优化拆分方式就成为了解决问题的突破口。

过程

我们可以通过「二进制分组」的方式使拆分方式更加优美。

具体地说就是令 $A_{i,j}$ ($j \in [0, \lfloor \log_2(k_i + 1) \rfloor - 1]$) 分别表示由 2^j 个单个物品「捆绑」而成的大物品。特殊地，若 $k_i + 1$ 不是 2 的整数次幂，则需要在最后添加一个由 $k_i - 2^{\lfloor \log_2(k_i + 1) \rfloor - 1}$ 个单个物品「捆绑」而成的大物品用于补足。

举几个例子：

- $6 = 1 + 2 + 3$
- $8 = 1 + 2 + 4 + 1$
- $18 = 1 + 2 + 4 + 8 + 3$
- $31 = 1 + 2 + 4 + 8 + 16$

显然，通过上述拆分方式，可以表示任意 $\leq k_i$ 个物品的等效选择方式。将每种物品按照上述方式拆分后，使用 0-1 背包的方法解决即可。

时间复杂度 $O(W \sum_{i=1}^n \log_2 k_i)$

实现

二进制分组代码

C++

```
1 index = 0;
2 for (int i = 1; i <= m; i++) {
3     int c = 1, p, h, k;
4     cin >> p >> h >> k;
5     while (k > c) {
6         k -= c;
7         list[++index].w = c * p;
8         list[index].v = c * h;
9         c *= 2;
10    }
11    list[++index].w = p * k;
12    list[index].v = h * k;
13 }
```

Python

```
1 index = 0
2 for i in range(1, m + 1):
3     c = 1
4     p, h, k = map(int, input().split())
5     while k > c:
6         k -= c
7         index += 1
8         list[index].w = c * p
9         list[index].v = c * h
10        c *= 2
11        index += 1
12        list[index].w = p * k
13        list[index].v = h * k
```

单调队列优化

见 [单调队列/单调栈优化](#)。

习题：[「Luogu P1776」宝物筛选_NOI 导刊 2010 提高（02）](#)

混合背包

混合背包就是将前面三种的背包问题混合起来，有的只能取一次，有的能取无限次，有的只能取 k 次。

这种题目看起来很吓人，可是只要领悟了前面几种背包的中心思想，并将其合并在一起就可以了。下面给出伪代码：

```
1 for (循环物品种类) {  
2     if (是 0 - 1 背包)  
3         套用 0 - 1 背包代码;  
4     else if (是完全背包)  
5         套用完全背包代码;  
6     else if (是多重背包)  
7         套用多重背包代码;  
8 }
```

例题



「Luogu P1833」樱花

有 n 种樱花树和长度为 T 的时间，有的樱花树只能看一遍，有的樱花树最多看 A_i 遍，有的樱花树可以看无数遍。每棵樱花树都有一个美学值 C_i ，求在 T 的时间内看哪些樱花树能使美学值最高。



核心代码

```
1 for (int i = 1; i <= n; i++) {  
2     if (cnt[i] == 0) { // 如果数量没有限制使用完全背包的核心代码  
3         for (int weight = w[i]; weight <= W; weight++) {  
4             dp[weight] = max(dp[weight], dp[weight - w[i]] + v[i]);  
5         }  
6     } else { // 物品有限使用多重背包的核心代码，它也可以处理0-1背包问题  
7         for (int weight = W; weight >= w[i]; weight--) {  
8             for (int k = 1; k * w[i] <= weight && k <= cnt[i]; k++) {  
9                 dp[weight] = max(dp[weight], dp[weight - k * w[i]] + k *  
10                    v[i]);  
11            }  
12        }  
13    }  
}
```

习题：[HDU 5410 CRB and His Birthday](#)

二维费用背包



「Luogu P1855」榨取 kkksc03



有 n 个任务需要完成，完成第 i 个任务需要花费 t_i 分钟，产生 c_i 元的开支。

现在有 T 分钟时间， W 元钱来处理这些任务，求最多能完成多少任务。

这道题是很明显的 0-1 背包问题，可是不同的是选一个物品会消耗两种价值（经费、时间），只需在状态中增加一维存放第二种价值即可。

这时候就要注意，再开一维存放物品编号就不合适了，因为容易MLE。

实现

C++

```
1 for (int k = 1; k <= n; k++)
2     for (int i = m; i >= mi; i--)    // 对经费进行一层枚举
3         for (int j = t; j >= ti; j--) // 对时间进行一层枚举
4             dp[i][j] = max(dp[i][j], dp[i - mi][j - ti] + 1);
```

Python

```
1 for k in range(1, n + 1):
2     for i in range(m, mi - 1, -1): # 对经费进行一层枚举
3         for j in range(t, ti - 1, -1): # 对时间进行一层枚举
4             dp[i][j] = max(dp[i][j], dp[i - mi][j - ti] + 1)
```

分组背包



「Luogu P1757」通天之分组背包



有 n 件物品和一个大小为 m 的背包，第 i 个物品的价值为 w_i ，体积为 v_i 。同时，每个物品属于一个组，同组内最多只能选择一个物品。求背包能装载物品的最大总价值。

这种题怎么想呢？其实是从「在所有物品中选择一件」变成了「从当前组中选择一件」，于是就对每一组进行一次 0-1 背包就可以了。

再说一说如何进行存储。我们可以将 $t_{k,i}$ 表示第 k 组的第 i 件物品的编号是多少，再用 cnt_k 表示第 k 组物品有多少个。

实现

C++

```
1  for (int k = 1; k <= ts; k++)           // 循环每一组
2      for (int i = m; i >= 0; i--)         // 循环背包容量
3          for (int j = 1; j <= cnt[k]; j++) // 循环该组的每一个物品
4              if (i >= w[t[k][j]])           // 背包容量充足
5                  dp[i] = max(dp[i],
6                               dp[i - w[t[k][j]]] + c[t[k][j]]); // 像0-1背包一样状态转移
```

移

Python

```
1  for k in range(1, ts + 1): # 循环每一组
2      for i in range(m, -1, -1): # 循环背包容量
3          for j in range(1, cnt[k] + 1): # 循环该组的每一个物品
4              if i >= w[t[k][j]]: # 背包容量充足
5                  dp[i] = max(
6                      dp[i], dp[i - w[t[k][j]]] + c[t[k][j]])
7          ) # 像0-1背包一样状态转移
```

这里要注意：**一定不能搞错循环顺序**，这样才能保证正确性。

有依赖的背包



「Luogu P1064」金明的预算方案



金明有 n 元钱，想要买 m 个物品，第 i 件物品的价格为 v_i ，重要度为 p_i 。有些物品是从属于某个主件物品的附件，要买这个物品，必须购买它的主件。

目标是让所有购买的物品的 $v_i \times p_i$ 之和最大。

考虑分类讨论。对于一个主件和它的若干附件，有以下几种可能：只买主件，买主件 + 某些附件。因为这几种可能性只能选一种，所以可以将这看成分组背包。

如果是多叉树的集合，则要先算子节点的集合，最后算父节点的集合。

泛化物品的背包

这种背包，没有固定的费用和价值，它的价值是随着分配给它的费用而定。在背包容量为 V 的背包问题中，当分配给它的费用为 v_i 时，能得到的价值就是 $h(v_i)$ 。这时，将固定的价值换成函数的引用即可。

杂项

小优化

根据贪心原理，当费用相同时，只需保留价值最高的；当价值一定时，只需保留费用最低的；当有两件物品 i, j 且 i 的价值大于 j 的价值并且 i 的费用小于 j 的费用时，只需保留 i 。

背包问题变种

输出方案

输出方案其实就是记录下来背包中的某一个状态是怎么推出来的。我们可以用 $g_{i,v}$ 表示第 i 件物品占用空间为 v 的时候是否选择了此物品。然后在转移时记录是选用了哪一种策略（选或不选）。输出时的伪代码：

```
1 int v = V; // 记录当前的存储空间
2
3 // 因为最后一件物品存储的是最终状态，所以从最后一件物品进行循环
4 for (从最后一件循环至第一件) {
5     if (g[i][v]) {
6         选了第 i 项物品;
7         v -= 第 i 项物品的重量;
8     } else {
9         未选第 i 项物品;
10    }
11 }
```

求方案数

对于给定的一个背包容量、物品费用、其他关系等的问题，求装到一定容量的方案总数。

这种问题就是把求最大值换成求和即可。

例如 0-1 背包问题的转移方程就变成了：

$$dp_i = \sum (dp_i, dp_{i-c_i})$$

初始条件： $dp_0 = 1$

因为当容量为 0 时也有一个方案，即什么都不装。

求最优方案总数

要求最优方案总数，我们要对 0-1 背包里的 dp 数组的定义稍作修改，DP 状态 $f_{i,j}$ 为在只能放前 i 个物品的情况下，容量为 j 的背包「正好装满」所能达到的最大总价值。

这样修改之后，每一种 DP 状态都可以用一个 $g_{i,j}$ 来表示方案数。

$f_{i,j}$ 表示只考虑前 i 个物品时背包体积「正好」是 j 时的最大价值。

$g_{i,j}$ 表示只考虑前 i 个物品时背包体积「正好」是 j 时的方案数。

转移方程：

如果 $f_{i,j} = f_{i-1,j}$ 且 $f_{i,j} \neq f_{i-1,j-v} + w$ 说明我们此时不选择把物品放入背包更优，方案数由 $g_{i-1,j}$ 转移过来，

如果 $f_{i,j} \neq f_{i-1,j}$ 且 $f_{i,j} = f_{i-1,j-v} + w$ 说明我们此时选择把物品放入背包更优，方案数由 $g_{i-1,j-v}$ 转移过来，

如果 $f_{i,j} = f_{i-1,j}$ 且 $f_{i,j} = f_{i-1,j-v} + w$ 说明放入或不放入都能取得最优解，方案数由 $g_{i-1,j}$ 和 $g_{i-1,j-v}$ 转移过来。

初始条件：

```
1 memset(f, 0x3f3f, sizeof(f)); // 避免没有装满而进行了转移
2 f[0] = 0;
3 g[0] = 1; // 什么都不装是一种方案
```

因为背包体积最大值有可能装不满，所以最优解不一定是 f_m 。

最后我们通过找到最优解的价值，把 g_j 数组里取到最优解的所有方案数相加即可。

实现

```
1 for (int i = 0; i < N; i++) {
2     for (int j = V; j >= v[i]; j--) {
3         int tmp = std::max(dp[j], dp[j - v[i]] + w[i]);
4         int c = 0;
5         if (tmp == dp[j]) c += cnt[j]; // 如果从
6         dp[j] 转移
7         if (tmp == dp[j - v[i]] + w[i]) c += cnt[j - v[i]]; // 如果从
8         dp[j-v[i]] 转移
9         dp[j] = tmp;
10        cnt[j] = c;
11    }
12 }
13 int max = 0; // 寻找最优解
14 for (int i = 0; i <= V; i++) {
15     max = std::max(max, dp[i]);
16 }
17 int res = 0;
18 for (int i = 0; i <= V; i++) {
19     if (dp[i] == max) {
20         res += cnt[i]; // 求和最优解方案数
21     }
22 }
```

背包的第 k 优解

普通的 0-1 背包是要求最优解，在普通的背包 DP 方法上稍作改动，增加一维用于记录当前状态下的前 k 优解，即可得到求 0-1 背包第 k 优解的算法。具体来讲： $dp_{i,j,k}$ 记录了前 i 个物品中，选择的物品总体积为 j 时，能够得到的第 k 大的价值和。这个状态可以理解为将普通 0-1 背包只用记录一个数据的 $dp_{i,j}$ 扩展为记录一个有序的优解序列。转移时，普通背包最优解的求法是 $dp_{i,j} = \max(dp_{i-1,j}, dp_{i-1,j-v_i} + w_i)$ ，现在我们则是要合并 $dp_{i-1,j}$, $dp_{i-1,j-v_i} + w_i$ 这两个大小为 1 的递减序列，并保留合并后前 k 大的价值记在 $dp_{i,j}$ 里，这一步利用双指针法，复杂度是 $O(k)$ 的，整体时间复杂度为 $O(nmk)$ 。空间上，此方法与普通背包一样可以压缩掉第一维，复杂度是 $O(mk)$ 的。

例题 HDU 2639 Bone Collector II

求 0-1 背包的严格第 k 优解。 $n \leq 100, v \leq 1000, k \leq 30$

实现

```
1  memset(dp, 0, sizeof(dp));
2  int i, j, p, x, y, z;
3  scanf("%d%d%d", &n, &m, &K);
4  for (i = 0; i < n; i++) scanf("%d", &w[i]);
5  for (i = 0; i < n; i++) scanf("%d", &c[i]);
6  for (i = 0; i < n; i++) {
7      for (j = m; j >= c[i]; j--) {
8          for (p = 1; p <= K; p++) {
9              a[p] = dp[j - c[i]][p] + w[i];
10             b[p] = dp[j][p];
11         }
12         a[p] = b[p] = -1;
13         x = y = z = 1;
14         while (z <= K && (a[x] != -1 || b[y] != -1)) {
15             if (a[x] > b[y])
16                 dp[j][z] = a[x++];
17             else
18                 dp[j][z] = b[y++];
19             if (dp[j][z] != dp[j][z - 1]) z++;
20         }
21     }
22 }
23 printf("%d\n", dp[m][K]);
```

参考资料与注释

- 背包问题九讲 - 崔添翼。

-  本页面最近更新：2025/8/9 14:37:03，[更新历史](#)
-  发现错误？想一起完善？[在 GitHub 上编辑此页！](#)
-  本页面贡献者：[lr1d](#), [sshy](#), [StudyingFather](#), [Marcythm](#), [partychicken](#), [H-J-Granger](#), [NachtgeistW](#), [countercurrent-time](#), [Enter-tainer](#), [weiranfu](#), [greyqz](#), [iamtwz](#), [Konano](#), [ksyx](#), [ouuan](#), [paigeman](#), [Tiphereth-A](#), [wolfdan666](#), [AngelKitty](#), [CCXXXI](#), [cjsoft](#), [diauweb](#), [Early0ver](#), [ezoixx130](#), [GekkaSaori](#), [GoodCoder666](#), [Henry-ZHR](#), [HeRaNO](#), [Link-cute](#), [LovelyBuggies](#), [LuoshuiTianyi](#), [Makkiy](#), [mgt](#), [minghu6](#), [odeinjul](#), [oldoldtea](#), [P-Y-Y](#), [PotassiumWings](#), [SamZhangQingChuan](#), [shenshuaijie](#), [Suyun514](#), [weiyong1024](#), [Xeonacid](#), [xyf007](#), [Alisahhh](#), [Alphnia](#), [CBW2007](#), [dhbloo](#), [fps5283](#), [GavinZhengOI](#), [Gesrua](#), [hsfzLZH1](#), [hydingsy](#), [kenlig](#), [kxccc](#), [lychees](#), [Menci](#), [Peanut-Tang](#), [Planarialce](#), [sbofgayschool](#), [shawlleyw](#), [Siyuan](#), [SukkaW](#), [tLLWtG](#), [WAAutoMaton](#), [x4Cx58x54](#), [xk2013](#), [zhb2000](#), [zhufengning](#)
- © 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



概率 DP

引入

概率 DP 用于解决概率问题与期望问题，建议先对 [概率 & 期望](#) 的内容有一定了解。一般情况下，解决概率问题需要顺序循环，而解决期望问题使用逆序循环，如果定义的状态转移方程存在后效性问题，还需要用到 [高斯消元](#) 来优化。概率 DP 也会结合其他知识进行考察，例如 [状态压缩](#)，树上进行 DP 转移等。

DP 求概率

这类题目采用顺推，也就是从初始状态推向结果。同一般的 DP 类似的，难点依然对状态转移方程的刻画，只是这类题目经过了概率论知识的包装。



例题 [Codeforces 148 D Bag of mice](#)



题目大意：袋子里有 w 只白鼠和 b 只黑鼠，公主和龙轮流从袋子里抓老鼠。谁先抓到白色老鼠谁就赢，如果袋子里没有老鼠了并且没有谁抓到白色老鼠，那么算龙赢。公主每次抓一只老鼠，龙每次抓完一只老鼠之后会有一只老鼠跑出来。每次抓的老鼠和跑出来的老鼠都是随机的。公主先抓。问公主赢的概率。

过程

设 $f_{i,j}$ 为轮到公主时袋子里有 i 只白鼠， j 只黑鼠，公主赢的概率。初始化边界， $f_{0,j} = 0$ 因为没有白鼠了算龙赢， $f_{i,0} = 1$ 因为抓一只就是白鼠，公主赢。考虑 $f_{i,j}$ 的转移：

- 公主抓到一只白鼠，公主赢了。概率为 $\frac{i}{i+j}$ ；
- 公主抓到一只黑鼠，龙抓到一只白鼠，龙赢了。概率为 $\frac{j}{i+j} \cdot \frac{i}{i+j-1}$ ；
- 公主抓到一只黑鼠，龙抓到一只黑鼠，跑出来一只黑鼠，转移到 $f_{i,j-3}$ 。概率为 $\frac{j}{i+j} \cdot \frac{j-1}{i+j-1} \cdot \frac{j-2}{i+j-2}$ ；
- 公主抓到一只黑鼠，龙抓到一只黑鼠，跑出来一只白鼠，转移到 $f_{i-1,j-2}$ 。概率为 $\frac{j}{i+j} \cdot \frac{j-1}{i+j-1} \cdot \frac{i}{i+j-2}$ ；

考虑公主赢的概率，第二种情况不参与计算。并且要保证后两种情况合法，所以还要判断 i, j 的大小，满足第三种情况至少要有 3 只黑鼠，满足第四种情况要有 1 只白鼠和 2 只黑鼠。

实现

参考实现

```
1 #include <cstring>
2 #include <iomanip>
3 #include <iostream>
4 using namespace std;
5
6 using ll = long long;
7 int w, b;
8 double dp[1010][1010];
9
10 int main() {
11     cin.tie(nullptr)->sync_with_stdio(false);
12     cin >> w >> b;
13     memset(dp, 0, sizeof(dp));
14     for (int i = 1; i <= w; i++) dp[i][0] = 1; // 初始化
15     for (int i = 1; i <= b; i++) dp[0][i] = 0;
16     for (int i = 1; i <= w; i++) {
17         for (int j = 1; j <= b; j++) { // 以下为题面概率转移
18             dp[i][j] += (double)i / (i + j);
19             if (j >= 3) {
20                 dp[i][j] += (double)j / (i + j) * (j - 1) / (i + j - 1) *
21 (j - 2) /
22                     (i + j - 2) * dp[i][j - 3];
23             }
24             if (i >= 1 && j >= 2) {
25                 dp[i][j] += (double)j / (i + j) * (j - 1) / (i + j - 1) *
26 i /
27                     (i + j - 2) * dp[i - 1][j - 2];
28             }
29         }
30     }
31     cout << fixed << setprecision(9) << dp[w][b] << '\n';
32     return 0;
33 }
```

习题

- [CodeForces 148 D Bag of mice](#)
- [POJ3071 Football](#)
- [CodeForces 768 D Jon and Orbs](#)

DP 求期望

例一



例题 POJ2096 Collecting Bugs



题目大意：一个软件有 s 个子系统，会产生 n 种 bug。某人一天发现一个 bug，这个 bug 属于某种 bug 分类，也属于某个子系统。每个 bug 属于某个子系统的概率是 $\frac{1}{s}$ ，属于某种 bug 分类的概率是 $\frac{1}{n}$ 。求发现 n 种 bug，且 s 个子系统都找到 bug 的期望天数。

过程

令 $f_{i,j}$ 为已经找到 i 种 bug 分类， j 个子系统的 bug，达到目标状态的期望天数。这里的目标状态是找到 n 种 bug 分类， s 个子系统的 bug。那么就有 $f_{n,s} = 0$ ，因为已经达到了目标状态，不需要用更多的天数去发现 bug 了，于是就以目标状态为起点开始递推，答案是 $f_{0,0}$ 。

考虑 $f_{i,j}$ 的状态转移：

- $f_{i,j}$ ，发现一个 bug 属于已经发现的 i 种 bug 分类， j 个子系统，概率为 $p_1 = \frac{i}{n} \cdot \frac{j}{s}$
- $f_{i,j+1}$ ，发现一个 bug 属于已经发现的 i 种 bug 分类，不属于已经发现的子系统，概率为 $p_2 = \frac{i}{n} \cdot (1 - \frac{j}{s})$
- $f_{i+1,j}$ ，发现一个 bug 不属于已经发现 bug 分类，属于 j 个子系统，概率为 $p_3 = (1 - \frac{i}{n}) \cdot \frac{j}{s}$
- $f_{i+1,j+1}$ ，发现一个 bug 不属于已经发现 bug 分类，不属于已经发现的子系统，概率为 $p_4 = (1 - \frac{i}{n}) \cdot (1 - \frac{j}{s})$

再根据期望的线性性质，就可以得到状态转移方程：

$$f_{i,j} = p_1 \cdot f_{i,j} + p_2 \cdot f_{i,j+1} + p_3 \cdot f_{i+1,j} + p_4 \cdot f_{i+1,j+1} + 1 \\ = \frac{p_2 \cdot f_{i,j+1} + p_3 \cdot f_{i+1,j} + p_4 \cdot f_{i+1,j+1} + 1}{1 - p_1}$$

实现

参考实现

```
1 #include <iomanip>
2 #include <iostream>
3 using namespace std;
4 int n, s;
5 double dp[1010][1010];
6
7 int main() {
8     cin.tie(nullptr)->sync_with_stdio(false);
9     cin >> n >> s;
10    dp[n][s] = 0;
11    for (int i = n; i >= 0; i--) {
12        for (int j = s; j >= 0; j--) {
13            if (i == n && s == j) continue;
14            dp[i][j] = (dp[i][j + 1] * i * (s - j) + dp[i + 1][j] * (n
15 - i) * j +
16                         dp[i + 1][j + 1] * (n - i) * (s - j) + n * s) /
17                         (n * s - i * j); // 概率转移
18        }
19    }
20    cout << fixed << setprecision(4) << dp[0][0] << '\n';
21    return 0;
}
```

例二

例题 「NOIP2016」换教室

题目大意：牛牛要上 n 个时间段的课，第 i 个时间段在 c_i 号教室，可以申请换到 d_i 号教室，申请成功的概率为 p_i ，至多可以申请 m 节课进行交换。第 i 个时间段的课上完后要走到第 $i+1$ 个时间段的教室，给出一张图 v 个教室 e 条路，移动会消耗体力，申请哪几门课程可以使他在教室间移动耗费的体力值的总和的期望值最小，也就是求出最小的期望路程和。

过程

对于这个无向连通图，先用 Floyd 求出最短路，为后续的状态转移带来便利。以移动一步为一个阶段（从第 i 个时间段到达第 $i+1$ 个时间段就是移动了一步），那么每一步就有 p_i 的概率到 d_i ，不过在所有的 d_i 中只能选 m 个，有 $1 - p_i$ 的概率到 c_i ，求出在 n 个阶段走完后的最小期望路程和。定义 $f_{i,j,0/1}$ 为在第 i 个时间段，连同这一个时间段已经用了 j 次换教室的机会，在这个时间段换（1）或者不换（0）教室的最小期望路程和，那么答案就是 $\min\{f_{n,i,0}, f_{n,i,1}\}, i \in [0, m]$ 。注意边界 $f_{1,0,0} = f_{1,1,1} = 0$ 。

考虑 $f_{i,j,0/1}$ 的状态转移：

- 如果这一阶段不换，即 $f_{i,j,0}$ 可能是由上一次不换的状态转移来的，那么就是 $f_{i-1,j,0} + w_{c_{i-1},c_i}$ ，也有可能是由上一次交换的状态转移来的，这里结合条件概率和全概率的知识分析可以得到 $f_{i-1,j,1} + w_{d_{i-1},c_i} \cdot p_{i-1} + w_{c_{i-1},c_i} \cdot (1 - p_{i-1})$ ，状态转移方程就有

$$f_{i,j,0} = \min(f_{i-1,j,0} + w_{c_{i-1},c_i}, f_{i-1,j,1} + w_{d_{i-1},c_i} \cdot p_{i-1} + w_{c_{i-1},c_i} \cdot (1 - p_{i-1}))$$

- 如果这一阶段交换，即 $f_{i,j,1}$ 。类似地，可能由上一次不换的状态转移来，也可能由上一次换的状态转移来。那么遇到不换的就乘上 $(1 - p_i)$ ，遇到交换的就乘上 p_i ，将所有会出现的情况都枚举一遍进行计算就好了。这里不再赘述各种转移情况，相信通过上一种阶段例子，这里的状态转移应该能够很容易写出来。

实现

参考实现

```
1 #include <algorithm>
2 #include <iomanip>
3 #include <iostream>
4
5 using namespace std;
6
7 constexpr int MAXN = 2010;
8 int n, m, v, e;
9 int f[MAXN][MAXN], c[MAXN], d[MAXN];
10 double dp[MAXN][MAXN][2], p[MAXN];
11
12 int main() {
13     cin.tie(nullptr)->sync_with_stdio(false);
14     cin >> n >> m >> v >> e;
15     for (int i = 1; i <= n; i++) cin >> c[i];
16     for (int i = 1; i <= n; i++) cin >> d[i];
17     for (int i = 1; i <= n; i++) cin >> p[i];
18     for (int i = 1; i <= v; i++)
19         for (int j = 1; j < i; j++) f[i][j] = f[j][i] = 1e9;
20
21     int u, V, w;
22     for (int i = 1; i <= e; i++) {
23         cin >> u >> V >> w;
24         f[u][V] = f[V][u] = min(w, f[u][V]);
25     }
26
27     for (int k = 1; k <= v; k++)
28         for (int i = 1; i <= n; i++) // 前面的，按照前面的题解进行一个
29             state transfer
30         for (int j = 1; j < i; j++)
31             if (f[i][k] + f[k][j] < f[i][j]) f[i][j] = f[j][i] = f[i]
32 [k] + f[k][j];
33
34     for (int i = 1; i <= n; i++)
35         for (int j = 0; j <= m; j++) dp[i][j][0] = dp[i][j][1] = 1e9;
36
37     dp[1][0][0] = dp[1][1][1] = 0;
38     for (int i = 2; i <= n; i++) // 有后效性方程
39         for (int j = 0; j <= min(i, m); j++) {
40             dp[i][j][0] = min(dp[i - 1][j][0] + f[c[i - 1]][c[i]],
41                               dp[i - 1][j][1] + f[c[i - 1]][c[i]] * (1
42 - p[i - 1]) +
43                               f[d[i - 1]][c[i]] * p[i - 1]);
44             if (j != 0) {
45                 dp[i][j][1] = min(dp[i - 1][j - 1][0] + f[c[i - 1]][d[i]]
46 * p[i] +
47                               f[c[i - 1]][c[i]] * (1 - p[i]),
48                 dp[i - 1][j - 1][1] +
49                               f[c[i - 1]][c[i]] * (1 - p[i - 1]))
```

```

50 * (1 - p[i]) +
51 f[c[i - 1]][d[i]] * (1 - p[i - 1])
52 * p[i] +
53 f[d[i - 1]][c[i]] * (1 - p[i]) *
54 p[i - 1] +
55 f[d[i - 1]][d[i]] * p[i - 1] *
56 p[i]);
57 }
}

double ans = 1e9;
for (int i = 0; i <= m; i++) ans = min(dp[n][i][0], min(dp[n]
[i][1], ans));
cout << fixed << setprecision(2) << ans;

return 0;
}

```

比较这两个问题可以发现，DP 求期望题目在对具体是求一个值或是最优化问题上会对方程得到转移方式有一些影响，但无论是 DP 求概率还是 DP 求期望，总是离不开概率知识和列出、化简计算公式的步骤，在写状态转移方程时需要思考的细节也类似。

习题

- [POJ2096 Collecting Bugs](#)
- [HDU3853 LOOPS](#)
- [HDU4035 Maze](#)
- [「NOIP2016」换教室](#)
- [「SCOI2008」奖励关](#)

有后效性 DP

CodeForces 24 D Broken robot ▼

题目大意：给出一个 $n \times m$ 的矩阵区域，一个机器人初始在第 x 行第 y 列，每一步机器人会等概率地选择停在原地，左移一步，右移一步，下移一步，如果机器人在边界则不会往区域外移动，问机器人到达最后一行的期望步数。

过程

在 $m = 1$ 时每次有 $\frac{1}{2}$ 的概率不动，有 $\frac{1}{2}$ 的概率向下移动一格，答案为 $2 \cdot (n - x)$ 。设 $f_{i,j}$ 为机器人从第 i 行第 j 列出发到达第 n 行的期望步数，最终状态为 $f_{n,j} = 0$ 。由于机器人会等

概率地选择停在原地，左移一步，右移一步，下移一步，考虑 $f_{i,j}$ 的状态转移：

- $f_{i,1} = \frac{1}{3} \cdot (f_{i+1,1} + f_{i,2} + f_{i,1}) + 1$
- $f_{i,j} = \frac{1}{4} \cdot (f_{i,j} + f_{i,j-1} + f_{i,j+1} + f_{i+1,j}) + 1$
- $f_{i,m} = \frac{1}{3} \cdot (f_{i,m} + f_{i,m-1} + f_{i+1,m}) + 1$

在行之间由于只能向下移动，是满足无后效性的。在列之间可以左右移动，在移动过程中可能产生环，不满足无后效性。将方程变换后可以得到：

- $2f_{i,1} - f_{i,2} = 3 + f_{i+1,1}$
- $3f_{i,j} - f_{i,j-1} - f_{i,j+1} = 4 + f_{i+1,j}$
- $2f_{i,m} - f_{i,m-1} = 3 + f_{i+1,m}$

由于是逆序的递推，所以每一个 $f_{i+1,j}$ 是已知的。由于有 m 列，所以右边相当于是一个 m 行的列向量，那么左边就是 m 行 m 列的矩阵。使用增广矩阵，就变成了 m 行 $m+1$ 列的矩阵，然后进行 [高斯消元](#) 即可解出答案。

实现

参考实现

```
1 #include <cstdio>
2 #include <cstring>
3 using namespace std;
4
5 constexpr int MAXN = 1e3 + 10;
6
7 double a[MAXN][MAXN], f[MAXN];
8 int n, m;
9
10 void solve(int x) {
11     memset(a, 0, sizeof a);
12     for (int i = 1; i <= m; i++) {
13         if (i == 1) {
14             a[i][i] = 2;
15             a[i][i + 1] = -1;
16             a[i][m + 1] = 3 + f[i];
17             continue;
18         } else if (i == m) {
19             a[i][i] = 2;
20             a[i][i - 1] = -1;
21             a[i][m + 1] = 3 + f[i];
22             continue;
23         }
24         a[i][i] = 3;
25         a[i][i + 1] = -1;
26         a[i][i - 1] = -1;
27         a[i][m + 1] = 4 + f[i];
28     }
29
30     for (int i = 1; i < m; i++) {
31         double p = a[i + 1][i] / a[i][i];
32         a[i + 1][i] = 0;
33         a[i + 1][i + 1] -= a[i][i + 1] * p;
34         a[i + 1][m + 1] -= a[i][m + 1] * p;
35     }
36
37     f[m] = a[m][m + 1] / a[m][m];
38     for (int i = m - 1; i >= 1; i--)
39         f[i] = (a[i][m + 1] - f[i + 1] * a[i][i + 1]) / a[i][i];
40 }
41
42 int main() {
43     scanf("%d %d", &n, &m);
44     int st, ed;
45     scanf("%d %d", &st, &ed);
46     if (m == 1) {
47         printf("%.10f\n", 2.0 * (n - st));
48         return 0;
49     }
```

```
50     for (int i = n - 1; i >= st; i--) {
51         solve(i);
52     }
53     printf("%.10f\n", f[ed]);
54     return 0;
55 }
```

习题

- [CodeForce 24 D Broken robot](#)
- [HDU 4418 Time Travel](#)
- [「HNOI2013」游走](#)

参考文献

[kuangbin 概率 DP 总结](#)

🔧 本页面最近更新：2025/5/3 19:43:25，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[Tiphereth-A](#), [ShaoChenHeng](#), [Enter-tainer](#), [ksyx](#), [StudyingFather](#), [c-forrest](#), [H-J-Granger](#), [iamtwz](#), [imp2002](#), [Ir1d](#), [kenlig](#), [LeBronGod](#), [Marcythm](#), [MegaOwler](#), [NachtgeistW](#), [ouuan](#), [Patchouliys](#), [Soohti](#), [TianKong-y](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



状压 DP

简介

状压 DP 是动态规划的一种，通过将状态集合转化为整数记录在 DP 状态中来实现状态转移的目的。

为了达到更低的时间复杂度，通常需要寻找更低状态数的状态。大部分题目中会利用二元状态，用 n 位二进制数表示 n 个独立二元状态的情况。

使用状态压缩通常涉及位运算，关于基础位运算详见 [位运算](#) 页面。

例题 1



「SCOI2005」互不侵犯



在 $N \times N$ 的棋盘里面放 K 个国王 ($1 \leq N \leq 9, 1 \leq K \leq N \times N$)，使他们互不攻击，共有多少种摆放方案。

国王能攻击到它上下左右，以及左上左下右上右下八个方向上附近的各一个格子，共 8 个格子。

解释

设 $f(i, j, l)$ 表示前 i 行，第 i 行的状态为 j ，且棋盘上已经放置 l 个国王时的合法方案数。

对于编号为 j 的状态，我们用二进制整数 $sit(j)$ 表示国王的放置情况， $sit(j)$ 的某个二进制位为 0 表示对应位置不放国王，为 1 表示在对应位置上放置国王；用 $sta(j)$ 表示该状态的国王个数，即二进制数 $sit(j)$ 中 1 的个数。例如，如下图所示的状态可用二进制数 100101 来表示（棋盘左边对应二进制低位），则有 $sit(j) = 100101_{(2)} = 37, sta(j) = 3$ 。



设当前行的状态为 j ，上一行的状态为 x ，可以得到下面的状态转移方程：

$$f(i, j, l) = \sum f(i - 1, x, l - sta(j))。$$

设上一行的状态编号为 x , 在保证当前行和上一行不冲突的前提下, 枚举所有可能的 x 进行转移, 转移方程:

$$f(i, j, l) = \sum f(i - 1, x, l - sta(j))$$

实现

参考代码

```
1 #include <algorithm>
2 #include <iostream>
3 using namespace std;
4 long long sta[2005], sit[2005], f[15][2005][105];
5 int n, k, cnt;
6
7 void dfs(int x, int num, int cur) {
8     if (cur >= n) { // 有新的合法状态
9         sit[++cnt] = x;
10        sta[cnt] = num;
11        return;
12    }
13    dfs(x, num, cur + 1); // cur位置不放国王
14    dfs(x + (1 << cur), num + 1,
15         cur + 2); // cur位置放国王, 与它相邻的位置不能再放国王
16 }
17
18 bool compatible(int j, int x) {
19     if (sit[j] & sit[x]) return false;
20     if ((sit[j] << 1) & sit[x]) return false;
21     if (sit[j] & (sit[x] << 1)) return false;
22     return true;
23 }
24
25 int main() {
26     cin >> n >> k;
27     dfs(0, 0, 0); // 先预处理一行的所有合法状态
28     for (int j = 1; j <= cnt; j++) f[1][j][sta[j]] = 1;
29     for (int i = 2; i <= n; i++)
30         for (int j = 1; j <= cnt; j++) {
31             for (int x = 1; x <= cnt; x++) {
32                 if (!compatible(j, x)) continue; // 排除不合法转移
33                 for (int l = sta[j]; l <= k; l++) f[i][j][l] += f[i - 1]
34 [x][l - sta[j]];
35             }
36             long long ans = 0;
37             for (int i = 1; i <= cnt; i++) ans += f[n][i][k]; // 累加答案
38             cout << ans << endl;
39         }
40 }
```

例题 2



[POI2004] PRZ

有 n 个人需要过桥，第 i 的人的重量为 w_i ，过桥用时为 t_i . 这些人过桥时会分成若干组，只有在某一组的所有人全部过桥后，其余的组才能过桥。桥最大承重为 W ，问这些人全部过桥的最短时间。

$$100 \leq W \leq 400, 1 \leq n \leq 16, 1 \leq t_i \leq 50, 10 \leq w_i \leq 100.$$

解释

我们用 S 表示所有人物构成集合的一个子集，设 $t(S)$ 表示 S 中人的最长过桥时间， $w(S)$ 表示 S 中所有人的总重量， $f(S)$ 表示 S 中所有人全部过桥的最短时间，则：

$$\begin{cases} f(\emptyset) = 0, \\ f(S) = \min_{T \subseteq S; w(T) \leq W} \{t(T) + f(S \setminus T)\}. \end{cases}$$

需要注意的是这里不能直接枚举集合再判断是否为子集，而应使用 [子集枚举](#)，从而使时间复杂度为 $O(3^n)$.

实现

参考代码

```
1 #include <iostream>
2 #include <limits>
3 #include <vector>
4 using namespace std;
5
6 int main() {
7     ios::sync_with_stdio(false);
8     cin.tie(nullptr);
9     int W, n;
10    cin >> W >> n;
11    const int S = (1 << n) - 1;
12    vector<int> ts(S + 1), ws(S + 1);
13    for (int j = 0, t, w; j < n; ++j) {
14        cin >> t >> w;
15        for (int i = 0; i <= S; ++i)
16            if (i & (1 << j)) {
17                ts[i] = max(ts[i], t);
18                ws[i] += w;
19            }
20    }
21    vector<int> dp(S + 1, numeric_limits<int>::max() / 2);
22    for (int i = 0; i <= S; ++i) {
23        if (ws[i] <= W) dp[i] = ts[i];
24        for (int j = i; j; j = i & (j - 1))
25            if (ws[i ^ j] <= W) dp[i] = min(dp[i], dp[j] + ts[i ^ j]);
26    }
27    cout << dp[S] << '\n';
28    return 0;
29 }
```

习题

- 「NOI2001」炮兵阵地
- 「USACO06NOV」玉米田 Corn Fields
- 「九省联考 2018」一双木棋

本页面最近更新：2025/8/8 20:46:23，[更新历史](#)

发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

本页面贡献者：[StudyingFather](#), [Ir1d](#), [H-J-Granger](#), [NachtgeistW](#), [countercurrent-time](#), [Enter-tainer](#), [ouuan](#), [Marcyhm](#), [sshwy](#), [AngelKitty](#), [CCXXXI](#), [cjsoft](#), [diauweb](#), [Early0v0](#), [ezoixx130](#), [GekkaSaori](#), [Henry-ZHR](#), [HeRaNO](#), [Konano](#), [LovelyBuggies](#), [Makkiy](#), [mgt](#), [minghu6](#), [P-Y-Y](#), [PotassiumWings](#), [SamZhangQingChuan](#), [Suyun514](#), [weiyong1024](#), [chieh2lu2](#), [Chrogeek](#),

GavinZhengOI, Gesrua, hsfzLZH1, iamtwz, kenlig, ksyx, kxccc, Link-cute, lychees, Peanut-Tang, REYwmp, shinzanmono, SukkaW, TianKong-y, Tiphereth-A, Xeonacid, YuJunDongGit, zhb2000

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用

树形 DP

树形 DP，即在树上进行的 DP。由于树固有的递归性质，树形 DP 一般都是递归进行的。



基础

以下面这道题为例，介绍一下树形 DP 的一般过程。



例题 洛谷 P1352 没有上司的舞会



某大学有 n 个职员，编号为 $1 \sim N$ 。他们之间有从属关系，也就是说他们的关系就像一棵以校长为根的树，父结点就是子结点的直接上司。现在有个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数 a_i ，但是呢，如果某个职员的直接上司来参加舞会了，那么这个职员就无论如何也不肯来参加舞会了。所以，请你编程计算，邀请哪些职员可以使快乐指数最大，求最大的快乐指数。

我们设 $f(i, 0/1)$ 代表以 i 为根的子树的最优解（第二维的值为 0 代表 i 不参加舞会的情况，1 代表 i 参加舞会的情况）。

对于每个状态，都存在两种决策（其中下面的 x 都是 i 的儿子）：

- 上司不参加舞会时，下属可以参加，也可以不参加，此时有
$$f(i, 0) = \sum \max\{f(x, 1), f(x, 0)\};$$
- 上司参加舞会时，下属都不会参加，此时有
$$f(i, 1) = \sum f(x, 0) + a_i.$$

我们可以通过 DFS，在返回上一层时更新当前结点的最优解。

```
1 #include <algorithm>
2 #include <iostream>
3 using namespace std;
4
5 struct edge {
6     int v, next;
7 } e[6005];
8
9 int head[6005], n, cnt, f[6005][2], ans, is_h[6005], vis[6005];
10
11 void addedge(int u, int v) { // 建图
12     e[++cnt].v = v;
13     e[cnt].next = head[u];
14     head[u] = cnt;
15 }
16
17 void calc(int k) {
```

```

18     vis[k] = 1;
19     for (int i = head[k]; i; i = e[i].next) { // 枚举该结点的每个子结点
20         if (vis[e[i].v]) continue;
21         calc(e[i].v);
22         f[k][1] += f[e[i].v][0];
23         f[k][0] += max(f[e[i].v][0], f[e[i].v][1]); // 转移方程
24     }
25     return;
26 }
27
28 int main() {
29     cin.tie(nullptr)->sync_with_stdio(false);
30     cin >> n;
31     for (int i = 1; i <= n; i++) cin >> f[i][1];
32     for (int i = 1; i < n; i++) {
33         int l, k;
34         cin >> l >> k;
35         is_h[l] = 1;
36         addedge(k, l);
37     }
38     for (int i = 1; i <= n; i++)
39         if (!is_h[i]) { // 从根结点开始DFS
40             calc(i);
41             cout << max(f[i][1], f[i][0]);
42             return 0;
43         }
44     }
}

```

通常，树形 DP 状态一般都为当前节点的最优解。先 DFS 遍历子树的所有最优解，然后向上传递给子树的父节点来转移，最终根节点的值即为所求的最优解。

习题

- [HDU 2196 Computer](#)
- [POJ 1463 Strategic game](#)
- [\[POI2014\]FAR-FarmCraft](#)

树上背包

树上的背包问题，简单来说就是背包问题与树形 DP 的结合。



例题 洛谷 P2014 CTSC1997 选课



现在有 n 门课程，第 i 门课程的学分为 a_i ，每门课程有零门或一门先修课，有先修课的课程需要先学完其先修课，才能学习该课程。

一位学生要学习 m 门课程，求其能获得的最多学分数。

$n, m \leq 300$

每门课最多只有一门先修课的特点，与有根树中一个点最多只有一个父亲结点的特点类似。

因此可以想到根据这一性质建树，从而所有课程组成了一个森林的结构。为了方便起见，我们可以新增一门 0 学分的课程（设这个课程的编号为 0），作为所有无先修课课程的先修课，这样我们就将森林变成了一棵以 0 号课程为根的树。

我们设 $f(u, i, j)$ 表示以 u 号点为根的子树中，已经遍历了 u 号点的前 i 棵子树，选了 j 门课程的最大学分。

转移的过程结合了树形 DP 和 背包 DP 的特点，我们枚举 u 点的每个子结点 v ，同时枚举以 v 为根的子树选了几门课程，将子树的结果合并到 u 上。

记点 x 的儿子个数为 s_x ，以 x 为根的子树大小为 siz_x ，可以写出下面的状态转移方程：

$$f(u, i, j) = \max_{v, k \leq j, k \leq siz_v} f(u, i - 1, j - k) + f(v, s_v, k)$$

注意上面状态转移方程中的几个限制条件，这些限制条件确保了一些无意义的状态不会被访问到。

f 的第二维可以很轻松地用滚动数组的方式省略掉，注意这时需要倒序枚举 j 的值。

可以证明，该做法的时间复杂度为 $O(nm)^1$ 。

参考代码

```
1 #include <algorithm>
2 #include <iostream>
3 #include <vector>
4 using namespace std;
5 int f[305][305], s[305], n, m;
6 vector<int> e[305];
7
8 int dfs(int u) {
9     int p = 1;
10    f[u][1] = s[u];
11    for (auto v : e[u]) {
12        int siz = dfs(v);
13        // 注意下面两重循环的上界和下界
14        // 只考虑已经合并过的子树，以及选的课程数超过 m+1 的状态没有意义
15        for (int i = min(p, m + 1); i; i--)
16            for (int j = 1; j <= siz && i + j <= m + 1; j++)
17                f[u][i + j] = max(f[u][i + j], f[u][i] + f[v][j]); // 转
18        移方程
19        p += siz;
20    }
21    return p;
22 }
23
24 int main() {
25     cin.tie(nullptr)->sync_with_stdio(false);
26     cin >> n >> m;
27     for (int i = 1; i <= n; i++) {
28         int k;
29         cin >> k >> s[i];
30         e[k].push_back(i);
31     }
32     dfs(0);
33     cout << f[0][m + 1];
34     return 0;
}
```

习题

- 「CTSC1997」选课
- 「JSOI2018」潜入行动
- 「SDOI2017」苹果树
- 「Codeforces Round 875 Div. 1」 Problem D. Mex Tree

换根 DP

树形 DP 中的换根 DP 问题又被称为二次扫描，通常不会指定根结点，并且根结点的变化会对一些值，例如子结点深度和、点权和等产生影响。

通常需要两次 DFS，第一次 DFS 预处理诸如深度，点权和之类的信息，在第二次 DFS 开始运行换根动态规划。

接下来以一些例题来带大家熟悉这个内容。

例题 [POI2008]STA-Station

给定一个 n 个点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

不妨令 u 为当前结点， v 为当前结点的子结点。首先需要用 s_i 来表示以 i 为根的子树中的结点个数，并且有 $s_u = 1 + \sum s_v$ 。显然需要一次 DFS 来计算所有的 s_i ，这次的 DFS 就是预处理，我们得到了以某个结点为根时其子树中的结点总数。

考虑状态转移，这里就是体现 "换根" 的地方了。令 f_u 为以 u 为根时，所有结点的深度之和。

$f_v \leftarrow f_u$ 可以体现换根，即以 u 为根转移到以 v 为根。显然在换根的转移过程中，以 v 为根或以 u 为根会导致其子树中的结点的深度产生改变。具体表现为：

- 所有在 v 的子树上的结点深度都减少了一，那么总深度和就减少了 s_v ；
- 所有不在 v 的子树上的结点深度都增加了一，那么总深度和就增加了 $n - s_v$ ；

根据这两个条件就可以推出状态转移方程 $f_v = f_u - s_v + n - s_v = f_u + n - 2 \times s_v$ 。

于是在第二次 DFS 遍历整棵树并状态转移 $f_v = f_u + n - 2 \times s_v$ ，那么就能求出以每个结点为根时的深度和了。最后只需要遍历一次所有根结点深度和就可以求出答案。

参考代码

```
1 #include <iostream>
2 using namespace std;
3
4 int head[1000010 << 1], tot;
5 long long n, sz[1000010], dep[1000010];
6 long long f[1000010];
7
8 struct node {
9     int to, next;
10 } e[1000010 << 1];
11
12 void add(int u, int v) { // 建图
13     e[++tot] = {v, head[u]};
14     head[u] = tot;
15 }
16
17 void dfs(int u, int fa) { // 预处理dfs
18     sz[u] = 1;
19     dep[u] = dep[fa] + 1;
20     for (int i = head[u]; i; i = e[i].next) {
21         int v = e[i].to;
22         if (v != fa) {
23             dfs(v, u);
24             sz[u] += sz[v];
25         }
26     }
27 }
28
29 void get_ans(int u, int fa) { // 第二次dfs换根dp
30     for (int i = head[u]; i; i = e[i].next) {
31         int v = e[i].to;
32         if (v != fa) {
33             f[v] = f[u] - sz[v] * 2 + n;
34             get_ans(v, u);
35         }
36     }
37 }
38
39 int main() {
40     cin.tie(nullptr)->sync_with_stdio(false);
41     cin >> n;
42     int u, v;
43     for (int i = 1; i <= n - 1; i++) {
44         cin >> u >> v;
45         add(u, v);
46         add(v, u);
47     }
48     dfs(1, 1);
49     for (int i = 1; i <= n; i++) f[1] += dep[i];
```

```
50     get_ans(1, 1);
51     long long int ans = -1;
52     int id;
53     for (int i = 1; i <= n; i++) { // 统计答案
54         if (f[i] > ans) {
55             ans = f[i];
56             id = i;
57         }
58     }
59     cout << id << '\n';
60     return 0;
61 }
```

习题

- Atcoder Educational DP Contest, Problem V, Subtree
- Educational Codeforces Round 67, Problem E, Tree Painting
- POJ 3585 Accumulation Degree
- [USACO10MAR]Great Cow Gathering G
- CodeForce 708C Centroids

参考资料与注释

-
1. 子树合并背包类型的 dp 的复杂度证明 - LYD729 的 CSDN 博客 ↪



本页面最近更新: 2025/7/13 17:32:21, [更新历史](#)



发现错误? 想一起完善? [在 GitHub 上编辑此页!](#)



本页面贡献者: [StudyingFather](#), [H-J-Granger](#), [Ir1d](#), [NachtgeistW](#), [countercurrent-time](#), [Early0v0](#), [Enter-tainer](#), [ShaoChenHeng](#), [sshwy](#), [aaron20100919](#), [AngelKitty](#), [CCXXXI](#), [cjsoft](#), [diauweb](#), [ezoixx130](#), [GekkaSaori](#), [greyqz](#), [Henry-ZHR](#), [Konano](#), [LovelyBuggies](#), [lychees](#), [Makkiy](#), [mgt](#), [minghu6](#), [ouuan](#), [P-Y-Y](#), [PotassiumWings](#), [SamZhangQingChuan](#), [Suyun514](#), [Tiphereth-A](#), [weiyong1024](#), [amakerlife](#), [billchenchina](#), [GavinZhengOI](#), [Gesrua](#), [isdanni](#), [kenlig](#), [ksyx](#), [kxccc](#), [Marcythm](#), [Peanut-Tang](#), [qz-cqy](#), [ShizubaAki](#), [SukkaW](#), [thredreams](#), [widsnoy](#), [Xeonacid](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供, 附加条款亦可能应用

2-SAT

SAT 是适定性 (Satisfiability) 问题的简称。一般形式为 k - 适定性问题，简称 k -SAT。而当 $k > 2$ 时该问题为 NP 完全的。所以我们只研究 $k = 2$ 的情况。

定义

2-SAT，简单的说就是给出 n 个布尔方程，每个方程和两个变量相关，如 $a \vee b$ ，表示变量 a, b 至少满足一个。然后判断是否存在可行方案，显然可能有多种选择方案，一般题中只需要求出一种即可。另外， $\neg a$ 表示 a 取反。

解决思路

洛谷 P4782 「模板」2-SAT

有 n 个布尔变量 $x_1 \sim x_n$ ，另有 m 个需要满足的条件，每个条件的形式都是「 x_i 为 true / false 或 x_j 为 true / false」。比如「 x_1 为真或 x_3 为假」、「 x_7 为假或 x_2 为假」。

2-SAT 问题的目标是给每个变量赋值使得所有条件得到满足。

使用布尔方程表示上述问题。设 a 表示 x_a 为真 ($\neg a$ 就表示 x_a 为假)。如果有个人提出的要求分别是 a 和 b ，即 $(a \vee b)$ (变量 a, b 至少满足一个)。对这些变量关系建有向图，则把 a 成立或不成立用图中的点表示， $\neg a \rightarrow b \rightarrow a$ ，表示 a 不成立 则 b 一定成立；同理， b 不成立 则 a 一定成立。建图之后，我们就可以使用缩点算法来求解 2-SAT 问题了。

原式	建图
$\neg a \vee b$	$a \rightarrow b$ 和 $\neg b \rightarrow \neg a$
$a \vee b$	$\neg a \rightarrow b$ 和 $\neg b \rightarrow a$
$\neg a \vee \neg b$	$a \rightarrow \neg b$ 和 $b \rightarrow \neg a$

许多 2-SAT 问题都需要找出如 a 不成立，则 b 成立 的关系。

求解

思考如果两点在同一强连通分量里有什么含义。根据前文边的逻辑意义可知：若两点在同一强连通分量内，则这两点代表的条件 **要么都满足，要么都不满足**。

建图后我们使用 [Tarjan 算法找 SCC](#)，判断对于任意布尔变量 a ，表示 a 成立的点和表示 a 不成立的点是否在同一个 SCC 中（同一条件不可能既满足又不满足，或既不满足又并非不满足），若有则输出无解，否则有解。

输出方案时可以通过变量在图中的拓扑序确定该变量的取值。如果变量 x 的拓扑序在 $\neg x$ 之后，那么取 x 值为真。应用到 Tarjan 算法的缩点，即 x 所在 SCC 编号在 $\neg x$ 之前时，取 x 为真。因为 Tarjan 算法求强连通分量时使用了栈，如果跑完 Tarjan 缩点之后呈现出的拓扑序更大，在 Tarjan 会更晚被遍历到，就会更早地被弹出栈而缩点，分量编号会更小，所以 Tarjan 求得的 SCC 编号相当于 **反拓扑序**。

算法会把整张图遍历一遍，由于这张图 n 和 m 同阶，计算答案时复杂度为 $O(n)$ ，因此总复杂度为 $O(n)$ 。

代码实现

```
1 #include <algorithm>
2 #include <cstdio>
3 #include <stack>
4 using namespace std;
5 const int N = 2e6 + 2;
6 int n, m, dfn[N], low[N], t, tot, head[N], a[N];
7 bool vis[N];
8 stack<int> s;
9
10 struct node {
11     int to, Next;
12 } e[N];
13
14 void adde(int u, int v) {
15     e[++tot].to = v;
16     e[tot].Next = head[u];
17     head[u] = tot;
18 }
19
20 void Tarjan(int u) {
21     dfn[u] = low[u] = ++t;
22     s.push(u);
23     vis[u] = 1;
24     for (int i = head[u]; i; i = e[i].Next) {
25         int v = e[i].to;
26         if (!dfn[v]) {
27             Tarjan(v);
28             low[u] = min(low[u], low[v]);
29         } else if (vis[v])
30             low[u] = min(low[u], dfn[v]);
31     }
32     if (dfn[u] == low[u]) {
33         int cur;
34         ++tot;
35         do {
36             cur = s.top();
37             s.pop();
38             vis[cur] = 0;
39             a[cur] = tot;
40         } while (cur != u);
41     }
42 }
43
44 int main() {
45     scanf("%d%d", &n, &m);
46     for (int i = 1, I, J, A, B; i <= m; i++) {
47         scanf("%d%d%d%d", &I, &A, &J, &B);
48         adde(A ? I + n : I, B ? J : J + n);
49         adde(B ? J + n : J, A ? I : I + n);
50 }
```

```
50     }
51     tot = 0;
52     for (int i = 1; i <= (n << 1); i++)
53         if (!dfn[i]) Tarjan(i);
54     for (int i = 1; i <= n; i++) {
55         if (a[i] == a[i + n]) {
56             printf("IMPOSSIBLE");
57             return 0;
58         }
59     }
60     puts("POSSIBLE");
61     for (int i = 1; i <= n; i++)
62         printf("%c%c", a[i] < a[i + n] ? '1' : '0', " \n"[i == n]);
63     return 0;
64 }
```

例题

例题 1



HDU3062 Party



有 n 对夫妻被邀请参加一个聚会，因为场地的问题，每对夫妻中只有一人可以列席。在 $2n$ 个人中，某些人之间有着很大的矛盾（当然夫妻之间是没有矛盾的），有矛盾的两个人是不会同时出现在聚会上的。有没有可能会有 n 个人同时列席？

按照上面的分析，如果 a_1 中的丈夫和 a_2 中的妻子不合，我们就把 a_1 中的丈夫和 a_2 中的丈夫连边，把 a_2 中的妻子和 a_1 中的妻子连边，然后缩点染色判断即可。

参考代码

```
1 #include <algorithm>
2 #include <cstring>
3 #include <iostream>
4 constexpr int MAXN = 2018;
5 constexpr int MAXM = 4000400;
6 using namespace std;
7 int Index, instack[MAXN], DFN[MAXN], LOW[MAXN];
8 int tot, color[MAXN];
9 int numedge, head[MAXN];
10
11 struct Edge {
12     int nxt, to;
13 } edge[MAXM];
14
15 int sta[MAXN], top;
16 int n, m;
17
18 void add(int x, int y) {
19     edge[++numedge].to = y;
20     edge[numedge].nxt = head[x];
21     head[x] = numedge;
22 }
23
24 void tarjan(int x) { // 缩点看不懂请移步强连通分量上面有一个链接可以
25     sta[++top] = x;
26     instack[x] = 1;
27     DFN[x] = LOW[x] = ++Index;
28     for (int i = head[x]; i; i = edge[i].nxt) {
29         int v = edge[i].to;
30         if (!DFN[v]) {
31             tarjan(v);
32             LOW[x] = min(LOW[x], LOW[v]);
33         } else if (instack[v])
34             LOW[x] = min(LOW[x], DFN[v]);
35     }
36     if (DFN[x] == LOW[x]) {
37         tot++;
38         do {
39             color[sta[top]] = tot; // 染色
40             instack[sta[top]] = 0;
41         } while (sta[top--] != x);
42     }
43 }
44
45
46 bool solve() {
47     for (int i = 0; i < 2 * n; i++)
48         if (!DFN[i]) tarjan(i);
49     for (int i = 0; i < 2 * n; i += 2)
```

```

50     if (color[i] == color[i + 1]) return false;
51     return true;
52 }
53
54 void init() {
55     top = 0;
56     tot = 0;
57     Index = 0;
58     numedge = 0;
59     memset(sta, 0, sizeof(sta));
60     memset(DFN, 0, sizeof(DFN));
61     memset(instack, 0, sizeof(instack));
62     memset(LOW, 0, sizeof(LOW));
63     memset(color, 0, sizeof(color));
64     memset(head, 0, sizeof(head));
65 }
66
67 int main() {
68     cin.tie(nullptr)->sync_with_stdio(false);
69     while (cin >> n >> m) {
70         init();
71         for (int i = 1; i <= m; i++) {
72             int a1, a2, c1, c2;
73             cin >> a1 >> a2 >> c1 >> c2;
74             add(2 * a1 + c1, 2 * a2 + 1 - c2);
75             // 对于第 i 对夫妇，我们用 2i+1 表示丈夫，2i 表示妻子。
76             add(2 * a2 + c2, 2 * a1 + 1 - c1);
77         }
78         if (solve())
79             cout << "YES\n";
80         else
81             cout << "NO\n";
82     }
83     return 0;
}

```

例题 2



2018-2019 ACM-ICPC Asia Seoul Regional K TV Show Game



有 k 盏灯，每盏灯是红色或者蓝色，但是初始的时候不知道灯的颜色。有 n 个人，每个人选择三盏灯并猜灯的颜色。一个人猜对两盏灯或以上的颜色就可以获得奖品。判断是否存在一个灯的着色方案使得每个人都能领奖，若有则输出一种灯的着色方案。

根据 [伍昱 - 《由对称性解 2-sat 问题》](#)，我们可以得出：如果要输出 2-SAT 问题的一个可行解，只需要在 tarjan 缩点后所得的 DAG 上自底向上地进行选择和删除。

具体实现的时候，可以通过构造 DAG 的反图后在反图上进行拓扑排序实现；也可以根据 tarjan 缩点后，所属连通块编号越小，节点越靠近叶子节点这一性质，优先对所属连通块编号小的节点进行选择。

下面给出第二种实现方法的代码。





参考代码

```
1 #include <algorithm>
2 #include <iostream>
3 using namespace std;
4 constexpr int MAXN = 1e4 + 5;
5 constexpr int MAXK = 5005;
6
7 int n, k;
8 int id[MAXN][5];
9 char s[MAXN][5][5], ans[MAXK];
10 bool vis[MAXN];
11
12 struct Edge {
13     int v, nxt;
14 } e[MAXN * 100];
15
16 int head[MAXN], tot = 1;
17
18 void addedge(int u, int v) {
19     e[tot].v = v;
20     e[tot].nxt = head[u];
21     head[u] = tot++;
22 }
23
24 int dfn[MAXN], low[MAXN], color[MAXN], stk[MAXN], ins[MAXN], top,
25 dfs_clock, c;
26
27 void tarjan(int x) { // tarjan算法求强联通
28     stk[++top] = x;
29     ins[x] = 1;
30     dfn[x] = low[x] = ++dfs_clock;
31     for (int i = head[x]; i; i = e[i].nxt) {
32         int v = e[i].v;
33         if (!dfn[v]) {
34             tarjan(v);
35             low[x] = min(low[x], low[v]);
36         } else if (ins[v])
37             low[x] = min(low[x], dfn[v]);
38     }
39     if (dfn[x] == low[x]) {
40         c++;
41         do {
42             color[stk[top]] = c;
43             ins[stk[top]] = 0;
44         } while (stk[top--] != x);
45     }
46 }
47
48 int main() {
49     cin.tie(nullptr)->sync_with_stdio(false);
```

```

50    cin >> k >> n;
51    for (int i = 1; i <= n; i++) {
52        for (int j = 1; j <= 3; j++) cin >> id[i][j] >> s[i][j];
53
54        for (int j = 1; j <= 3; j++) {
55            for (int k = 1; k <= 3; k++) {
56                if (j == k) continue;
57                int u = 2 * id[i][j] - (s[i][j][0] == 'B');
58                int v = 2 * id[i][k] - (s[i][k][0] == 'R');
59                addedge(u, v);
60            }
61        }
62    }
63
64    for (int i = 1; i <= 2 * k; i++)
65        if (!dfn[i]) tarjan(i);
66
67    for (int i = 1; i <= 2 * k; i += 2)
68        if (color[i] == color[i + 1]) {
69            cout << "-1\n";
70            return 0;
71        }
72
73    for (int i = 1; i <= 2 * k; i += 2) {
74        int f1 = color[i], f2 = color[i + 1];
75        if (vis[f1]) {
76            ans[(i + 1) >> 1] = 'R';
77            continue;
78        }
79        if (vis[f2]) {
80            ans[(i + 1) >> 1] = 'B';
81            continue;
82        }
83        if (f1 < f2) {
84            vis[f1] = true;
85            ans[(i + 1) >> 1] = 'R';
86        } else {
87            vis[f2] = true;
88            ans[(i + 1) >> 1] = 'B';
89        }
90    }
91    ans[k + 1] = 0;
92    cout << (ans + 1) << '\n';
93    return 0;
}

```

习题

- 洛谷 P5782 和平委员会

- POJ3683 Priest John's Busiest Day
-

🔧 本页面最近更新：2025/8/28 21:35:18，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[AndrewWayne](#), [Ir1d](#), [Backl1ght](#), [Tiphereth-A](#), [chu-yuehan](#), [Early0v0](#), [Enter-tainer](#), [frank-xjh](#), [H-J-Granger](#), [akakw1](#), [algosheep](#), [Anguei](#), [c-forrest](#), [felixesintot](#), [guodong2005](#), [HeRaNO](#), [jpy-cpp](#), [kenlig](#), [Konano](#), [ksyx](#), [ouuan](#), [sshwy](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



图论相关概念

本页面概述了图论中的一些概念，这些概念并不全是在 OI 中常见的，对于 OIer 来说，只需掌握本页面中的基础部分即可，如果在学习中碰到了不懂的概念，可以再来查阅。

⚠ Warning

图论相关定义在不同教材中往往会有不同，遇到的时候需根据上下文加以判断。

图

图 (graph) 是一个二元组 $G = (V(G), E(G))$ 。其中 $V(G)$ 是非空集，称为 **点集 (vertex set)**，对于 V 中的每个元素，我们称其为 **顶点 (vertex)** 或 **节点 (node)**，简称 **点**； $E(G)$ 为 $V(G)$ 各结点之间边的集合，称为 **边集 (edge set)**。

常用 $G = (V, E)$ 表示图。

当 V, E 都是有限集合时，称 G 为 **有限图**。

当 V 或 E 是无限集合时，称 G 为 **无限图**。

图有多种，包括 **无向图 (undirected graph)**，**有向图 (directed graph)**，**混合图 (mixed graph)** 等。

若 G 为无向图，则 E 中的每个元素为一个无序二元组 (u, v) ，称作 **无向边 (undirected edge)**，简称 **边 (edge)**，其中 $u, v \in V$ 。设 $e = (u, v)$ ，则 u 和 v 称为 e 的 **端点 (endpoint)**。

若 G 为有向图，则 E 中的每一个元素为一个有序二元组 (u, v) ，有时也写作 $u \rightarrow v$ ，称作 **有向边 (directed edge)** 或 **弧 (arc)**，在不引起混淆的情况下也可以称作 **边 (edge)**。设 $e = u \rightarrow v$ ，则此时 u 称为 e 的 **起点 (tail)**， v 称为 e 的 **终点 (head)**，起点和终点也称为 e 的 **端点 (endpoint)**。并称 u 是 v 的直接前驱， v 是 u 的直接后继。

为什么起点是 tail，终点是 head？

边通常用箭头表示，而箭头是从「尾」指向「头」的。

若 G 为混合图，则 E 中既有 **有向边**，又有 **无向边**。

若 G 的每条边 $e_k = (u_k, v_k)$ 都被赋予一个数作为该边的 **权**，则称 G 为 **赋权图**。如果这些权都是正实数，就称 G 为 **正权图**。

图 G 的点数 $|V(G)|$ 也被称作图 G 的 阶 (order)。

形象地说，图是由若干点以及连接点与点的边构成的。

相邻

在无向图 $G = (V, E)$ 中，若点 v 是边 e 的一个端点，则称 v 和 e 是 **关联的 (incident)** 或 **相邻的 (adjacent)**。对于两顶点 u 和 v ，若存在边 (u, v) ，则称 u 和 v 是 **相邻的 (adjacent)**。

一个顶点 $v \in V$ 的 **邻域 (neighborhood)** 是所有与之相邻的顶点所构成的集合，记作 $N(v)$ 。

一个点集 S 的邻域是所有与 S 中至少一个点相邻的点所构成的集合，记作 $N(S)$ ，即：

$$N(S) = \bigcup_{v \in S} N(v)$$

简单图

自环 (loop): 对 E 中的边 $e = (u, v)$ ，若 $u = v$ ，则 e 被称作一个自环。

重边 (multiple edge): 若 E 中存在两个完全相同的元素（边） e_1, e_2 ，则它们被称作（一组）重边。

简单图 (simple graph): 若一个图中没有自环和重边，它被称为简单图。具有至少两个顶点的简单无向图中一定存在度相同的结点。（鸽巢原理）

如果一张图中有自环或重边，则称它为 **多重图 (multigraph)**。

⚠ Warning

在无向图中 (u, v) 和 (v, u) 算一组重边，而在有向图中， $u \rightarrow v$ 和 $v \rightarrow u$ 不为重边。

⚠ Warning

在题目中，如果没有特殊说明，是可以存在自环和重边的，在做题时需特殊考虑。

度数

与一个顶点 v 关联的边的条数称作该顶点的 **度 (degree)**，记作 $d(v)$ 。特别地，对于边 (v, v) ，则每条这样的边要对 $d(v)$ 产生 2 的贡献。

对于无向简单图，有 $d(v) = |N(v)|$ 。

握手定理（又称图论基本定理）：对于任何无向图 $G = (V, E)$ ，有 $\sum_{v \in V} d(v) = 2|E|$ 。

推论：在任意图中，度数为奇数的点必然有偶数个。

若 $d(v) = 0$ ，则称 v 为 **孤立点 (isolated vertex)**。

| 若 $d(v) = 1$ ，则称 v 为 **叶节点 (leaf vertex)/悬挂点 (pendant vertex)**。

若 $2 \mid d(v)$ ，则称 v 为 **偶点 (even vertex)**。

若 $2 \nmid d(v)$ ，则称 v 为 **奇点 (odd vertex)**。图中奇点的个数是偶数。

若 $d(v) = |V| - 1$ ，则称 v 为 **支配点 (universal vertex)**。

对一张图，所有节点的度数的最小值称为 G 的 **最小度 (minimum degree)**，记作 $\delta(G)$ ；最大值称为 **最大度 (maximum degree)**，记作 $\Delta(G)$ 。即： $\delta(G) = \min_{v \in G} d(v)$ ， $\Delta(G) = \max_{v \in G} d(v)$ 。

在有向图 $G = (V, E)$ 中，以一个顶点 v 为起点的边的条数称为该顶点的 **出度 (out-degree)**，记作 $d^+(v)$ 。以一个顶点 v 为终点的边的条数称为该节点的 **入度 (in-degree)**，记作 $d^-(v)$ 。显然 $d^+(v) + d^-(v) = d(v)$ 。

对于任何有向图 $G = (V, E)$ ，有：

$$\sum_{v \in V} d^+(v) = \sum_{v \in V} d^-(v) = |E|$$

若对一张无向图 $G = (V, E)$ ，每个顶点的度数都是一个固定的常数 k ，则称 G 为 **k -正则图 (k -regular graph)**。

如果给定一个序列 a ，可以找到一个图 G ，以其为度数列，则称 a 是 **可图化** 的。

如果给定一个序列 a ，可以找到一个简单图 G ，以其为度数列，则称 a 是 **可简单图化** 的。

路径

途径 (walk)：途径是连接一连串顶点的边的序列，可以为有限或无限长度。形式化地说，一条有限途径 w 是一个边的序列 e_1, e_2, \dots, e_k ，使得存在一个顶点序列 v_0, v_1, \dots, v_k 满足 $e_i = (v_{i-1}, v_i)$ ，其中 $i \in [1, k]$ 。这样的途径可以简写为 $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$ 。通常来说，边的数量 k 被称作这条途径的 **长度**（如果边是带权的，长度通常指途径上的边权之和，题目中也可能另有定义）。

迹 (trail)：对于一条途径 w ，若 e_1, e_2, \dots, e_k 两两互不相同，则称 w 是一条迹。

路径 (path)（又称 **简单路径 (simple path)**）：对于一条迹 w ，若其连接的点的序列中点两两不同，则称 w 是一条路径。

回路 (circuit)：对于一条迹 w ，若 $v_0 = v_k$ ，则称 w 是一条回路。

环/圈 (cycle) (又称 简单回路/简单环 (simple circuit))：对于一条回路 w ，若 $v_0 = v_k$ 是点序列中唯一重复出现的点对，则称 w 是一个环。

⚠ Warning

关于路径的定义在不同地方可能有所不同，如，「路径」可能指本文中的「途径」，「环」可能指本文中的「回路」。如果在题目中看到类似的词汇，且没有「简单路径」 / 「非简单路径」（即本文中的「途径」）等特殊说明，最好询问一下具体指什么。

子图

对一张图 $G = (V, E)$ ，若存在另一张图 $H = (V', E')$ 满足 $V' \subseteq V$ 且 $E' \subseteq E$ ，则称 H 是 G 的**子图 (subgraph)**，记作 $H \subseteq G$ 。

若对 $H \subseteq G$ ，满足 $\forall u, v \in V'$ ，只要 $(u, v) \in E$ ，均有 $(u, v) \in E'$ ，则称 H 是 G 的**导出子图/诱导子图 (induced subgraph)**。

容易发现，一个图的导出子图仅由子图的点集决定，因此点集为 $V'(V' \subseteq V)$ 的导出子图称为 V' 导出的子图，记作 $G[V']$ 。

若 $H \subseteq G$ 满足 $V' = V$ ，则称 H 为 G 的**生成子图/支撑子图 (spanning subgraph)**。

显然， G 是自身的子图，支撑子图，导出子图；[无边图](#)是 G 的支撑子图。原图 G 和无边图都是 G 的平凡子图。

如果一张无向图 G 的某个生成子图 F 为 k - 正则图，则称 F 为 G 的一个 **k -因子 (k -factor)**。

如果有向图 $G = (V, E)$ 的导出子图 $H = G[V^*]$ 满足 $\forall v \in V^*, (v, u) \in E$ ，有 $u \in V^*$ ，则称 H 为 G 的一个**闭合子图 (closed subgraph)**。

连通

无向图

对于一张无向图 $G = (V, E)$ ，对于 $u, v \in V$ ，若存在一条途径使得 $v_0 = u, v_k = v$ ，则称 u 和 v 是**连通的 (connected)**。由定义，任意一个顶点和自身连通，任意一条边的两个端点连通。

若无向图 $G = (V, E)$ ，满足其中任意两个顶点均连通，则称 G 是**连通图 (connected graph)**， G 的这一性质称作**连通性 (connectivity)**。

若 H 是 G 的一个连通子图，且不存在 F 满足 $H \subsetneq F \subseteq G$ 且 F 为连通图，则 H 是 G 的一个**连通块/连通分量 (connected component)**（极大连通子图）。

有向图

对于一张有向图 $G = (V, E)$, 对于 $u, v \in V$, 若存在一条途径使得 $v_0 = u, v_k = v$, 则称 u 可达 v 。由定义, 任意一个顶点可达自身, 任意一条边的起点可达终点。(无向图中的连通也可以视作双向可达。)

若一张有向图的节点两两互相可达, 则称这张图是 **强连通的 (strongly connected)**。

若一张有向图的边替换为无向边后可以得到一张连通图, 则称原来这张有向图是 **弱连通的 (weakly connected)**。

与连通分量类似, 也有 **弱连通分量 (weakly connected component)** (极大弱连通子图) 和 **强连通分量 (strongly connected component)** (极大强连通子图)。

相关算法请参见 [强连通分量](#)。

割

相关算法请参见 [割点和桥](#) 以及 [双连通分量](#)。

在本部分中, 有向图的「连通」一般指「强连通」。

对于连通图 $G = (V, E)$, 若 $V' \subseteq V$ 且 $G[V \setminus V']$ (即从 G 中删去 V' 中的点) 不是连通图, 则 V' 是图 G 的一个 **点割集 (vertex cut/separating set)**。大小为一的点割集又被称作 **割点 (cut vertex)**。

对于连通图 $G = (V, E)$ 和整数 k , 若 $|V| \geq k + 1$ 且 G 不存在大小为 $k - 1$ 的点割集, 则称图 G 是 **k -点连通的 (k -vertex-connected)**, 而使得上式成立的最大的 k 被称作图 G 的 **点连通度 (vertex connectivity)**, 记作 $\kappa(G)$ 。(对于非完全图, 点连通度即为最小点割集的大小, 而完全图 K_n 的点连通度为 $n - 1$ 。)

对于图 $G = (V, E)$ 以及 $u, v \in V$ 满足 $u \neq v$, u 和 v 不相邻, u 可达 v , 若 $V' \subseteq V$, $u, v \notin V'$, 且在 $G[V \setminus V']$ 中 u 和 v 不连通, 则 V' 被称作 u 到 v 的点割集。 u 到 v 的最小点割集的大小被称作 u 到 v 的 **局部点连通度 (local connectivity)**, 记作 $\kappa(u, v)$ 。

还可以在边上作类似的定义:

对于连通图 $G = (V, E)$, 若 $E' \subseteq E$ 且 $G' = (V, E \setminus E')$ (即从 G 中删去 E' 中的边) 不是连通图, 则 E' 是图 G 的一个 **边割集 (edge cut)**。大小为一的边割集又被称作 **桥 (bridge)**。

对于连通图 $G = (V, E)$ 和整数 k , 若 G 不存在大小为 $k - 1$ 的边割集, 则称图 G 是 **k -边连通的 (k -edge-connected)**, 而使得上式成立的最大的 k 被称作图 G 的 **边连通度 (edge connectivity)**, 记作 $\lambda(G)$ 。(对于任何图, 边连通度即为最小边割集的大小。)

对于图 $G = (V, E)$ 以及 $u, v \in V$ 满足 $u \neq v$, u 可达 v , 若 $E' \subseteq E$, 且在 $G' = (V, E \setminus E')$ 中 u 和 v 不连通, 则 E' 被称作 u 到 v 的边割集。 u 到 v 的最小边割集的大小被称作 u 到 v 的 **局部边连通度 (local edge-connectivity)**, 记作 $\lambda(u, v)$ 。

点双连通 (biconnected) 几乎与 2- 点连通完全一致，除了一条边连接两个点构成的图，它是点双连通的，但不是 2- 点连通的。换句话说，没有割点的连通图是点双连通的。

边双连通 (2-edge-connected) 与 2- 边双连通完全一致。换句话说，没有桥的连通图是边双连通的。

与连通分量类似，也有 **点双连通分量 (biconnected component)**（极大点双连通子图）和 **边双连通分量 (2-edge-connected component)**（极大边双连通子图）。

Whitney 定理：对任意的图 G ，有 $\kappa(G) \leq \lambda(G) \leq \delta(G)$ 。（不等式中的三项分别为点连通度、边连通度、最小度。）

稀疏图 / 稠密图

若一张图的边数远小于其点数的平方，那么它是一张 **稀疏图 (sparse graph)**。

若一张图的边数接近其点数的平方，那么它是一张 **稠密图 (dense graph)**。

这两个概念并没有严格的定义，一般用于讨论 **时间复杂度** 为 $O(|V|^2)$ 的算法与 $O(|E|)$ 的算法的效率差异（在稠密图上这两种算法效率相当，而在稀疏图上 $O(|E|)$ 的算法效率明显更高）。

补图

对于无向简单图 $G = (V, E)$ ，它的 **补图 (complement graph)** 指的是这样的一张图：记作 \bar{G} ，满足 $V(\bar{G}) = V(G)$ ，且对任意节点对 (u, v) ， $(u, v) \in E(\bar{G})$ 当且仅当 $(u, v) \notin E(G)$ 。

反图

对于有向图 $G = (V, E)$ ，它的 **反图 (transpose graph)** 指的是点集不变，每条边反向得到的图，即：若 G 的反图为 $G' = (V, E')$ ，则 $E' = \{(v, u) | (u, v) \in E\}$ 。

特殊的图

若无向简单图 G 满足任意不同两点间均有边，则称 G 为 **完全图 (complete graph)**， n 阶完全图记作 K_n 。若有向图 G 满足任意不同两点间都有两条方向不同的边，则称 G 为 **有向完全图 (complete digraph)**。

边集为空的图称为 **无边图 (edgeless graph)**、**空图 (empty graph)** 或 **零图 (null graph)**， n 阶无边图记作 \bar{K}_n 或 N_n 。 N_n 与 K_n 互为补图。

⚠ Warning

零图 (null graph) 也可指 零阶图 (order-zero graph) K_0 , 即点集与边集均为空的图。

若有向简单图 G 满足任意不同两点间都有恰好一条边 (单向), 则称 G 为 竞赛图 (tournament graph)。

若无向简单图 $G = (V, E)$ 的所有边恰好构成一个圈, 则称 G 为 环图/圈图 (cycle graph), $n(n \geq 3)$ 阶圈图记作 C_n 。易知, 一张图为圈图的充分必要条件是, 它是 2- 正则连通图。

若无向简单图 $G = (V, E)$ 满足, 存在一个点 v 为支配点, 其余点之间没有边相连, 则称 G 为 星图/菊花图 (star graph), $n+1(n \geq 1)$ 阶星图记作 S_n 。

若无向简单图 $G = (V, E)$ 满足, 存在一个点 v 为支配点, 其它点之间构成一个圈, 则称 G 为 轮图 (wheel graph), $n+1(n \geq 3)$ 阶轮图记作 W_n 。

若无向简单图 $G = (V, E)$ 的所有边恰好构成一条简单路径, 则称 G 为 链 (chain/path graph), n 阶的链记作 P_n 。易知, 一条链由一个圈图删去一条边而得。

如果一张无向连通图不含环, 则称它是一棵 树 (tree)。相关内容详见 [树基础](#)。

如果一张无向连通图包含恰好一个环, 则称它是一棵 基环树 (pseudotree)。

如果一张有向弱连通图每个点的入度都为 1, 则称它是一棵 基环外向树。

如果一张有向弱连通图每个点的出度都为 1, 则称它是一棵 基环内向树。

多棵树可以组成一个 森林 (forest), 多棵基环树可以组成 基环森林 (pseudoforest), 多棵基环外向树可以组成 基环外向树森林, 多棵基环内向树可以组成 基环内向森林 (functional graph)。

如果一张无向连通图的每条边最多在一个环内, 则称它是一棵 仙人掌 (cactus)。多棵仙人掌可以组成 沙漠。

如果一张图的点集可以被分为两部分, 每一部分的内部都没有连边, 那么这张图是一张 二分图 (bipartite graph)。如果二分图中任何两个不在同一部分的点之间都有连边, 那么这张图是一张 完全二分图 (complete bipartite graph/biclique), 一张两部分分别有 n 个点和 m 个点的完全二分图记作 $K_{n,m}$ 。相关内容详见 [二分图](#)。

如果一张图可以画在一个平面上, 且没有两条边在非端点处相交, 那么这张图是一张 平面图 (planar graph)。一张图的任何子图都不是 K_5 或 $K_{3,3}$ 是其为一张平面图的充要条件。对于简单连通平面图 $G = (V, E)$ 且 $V \geq 3$, $|E| \leq 3|V| - 6$ 。

同构

两个图 G 和 H , 如果存在一个双射 $f : V(G) \rightarrow V(H)$, 且满足 $(u, v) \in E(G)$, 当且仅当 $(f(u), f(v)) \in E(H)$, 则我们称 f 为 G 到 H 的一个 同构 (isomorphism), 且图 G 与图 H 是 同

构的 (isomorphic)，记作 $G \cong H$ 。

从定义可知，若 $G \cong H$ ，必须满足：

- $|V(G)| = |V(H)|, |E(G)| = |E(H)|$
- G 和 H 结点度的非增序列相同
- G 和 H 存在同构的导出子图

无向简单图的二元运算

对于无向简单图，我们可以定义如下二元运算：

交 (intersection)：图 $G = (V_1, E_1), H = (V_2, E_2)$ 的交定义成图 $G \cap H = (V_1 \cap V_2, E_1 \cap E_2)$ 。

容易证明两个无向简单图的交还是无向简单图。

并 (union)：图 $G = (V_1, E_1), H = (V_2, E_2)$ 的并定义成图 $G \cup H = (V_1 \cup V_2, E_1 \cup E_2)$ 。

和 (sum)/直和 (direct sum)：对于 $G = (V_1, E_1), H = (V_2, E_2)$ ，任意构造 $H' \cong H$ 使得 $V(H') \cap V_1 = \emptyset$ (H' 可以等于 H)。此时与 $G \cup H'$ 同构的任何图称为 G 和 H 的和/直和/不交并，记作 $G + H$ 或 $G \oplus H$ 。

若 G 与 H 的点集本身不相交，则 $G \cup H = G + H$ 。

比如，森林可以定义成若干棵树的和。

并与和的区别

可以理解为，「并」会让两张图中「名字相同」的点、边合并，而「和」则不会。

特殊的点集/边集

支配集

对于无向图 $G = (V, E)$ ，若 $V' \subseteq V$ 且 $\forall v \in (V \setminus V')$ 存在边 $(u, v) \in E$ 满足 $u \in V'$ ，则 V' 是图 G 的一个 **支配集 (dominating set)**。

无向图 G 最小的支配集的大小记作 $\gamma(G)$ 。求一张图的最小支配集是 NP 困难 的。

对于有向图 $G = (V, E)$ ，若 $V' \subseteq V$ 且 $\forall v \in (V \setminus V')$ 存在边 $(u, v) \in E$ 满足 $u \in V'$ ，则 V' 是图 G 的一个 **出 - 支配集 (out-dominating set)**。类似地，可以定义有向图的 **入 - 支配集 (in-dominating set)**。

有向图 G 最小的出 - 支配集大小记作 $\gamma^+(G)$ ，最小的入 - 支配集大小记作 $\gamma^-(G)$ 。

边支配集

对于图 $G = (V, E)$, 若 $E' \subseteq E$ 且 $\forall e \in (E \setminus E')$ 存在 E' 中的边与其有公共点, 则称 E' 是图 G 的一个 **边支配集 (edge dominating set)**。

求一张图的最小边支配集是 **NP 困难** 的。

独立集

对于图 $G = (V, E)$, 若 $V' \subseteq V$ 且 V' 中任意两点都不相邻, 则 V' 是图 G 的一个 **独立集 (independent set)**。

图 G 最大的独立集的大小记作 $\alpha(G)$ 。求一张图的最大独立集是 **NP 困难** 的。

匹配

对于图 $G = (V, E)$, 若 $E' \subseteq E$ 且 E' 中任意两条不同的边都没有公共的端点, 且 E' 中任意一条边都不是自环, 则 E' 是图 G 的一个 **匹配 (matching)**, 也可以叫作 **边独立集 (independent edge set)**。如果一个点是匹配中某条边的一个端点, 则称这个点是 **被匹配的 (matched)/饱和的 (saturated)**, 否则称这个点是 **不被匹配的 (unmatched)**。

边数最多的匹配被称作一张图的 **最大匹配 (maximum-cardinality matching)**。图 G 的最大匹配的大小记作 $\nu(G)$ 。

如果边带权, 那么权重之和最大的匹配被称作一张图的 **最大权匹配 (maximum-weight matching)**。

如果一个匹配在加入任何一条边后都不再是一个匹配, 那么这个匹配是一个 **极大匹配 (maximal matching)**。最大的极大匹配就是最大匹配, 任何最大匹配都是极大匹配。极大匹配一定是边支配集, 但边支配集不一定是匹配。最小极大匹配和最小边支配集大小相等, 但最小边支配集不一定是匹配。求最小极大匹配是 **NP 困难的**。

如果在一个匹配中所有点都是被匹配的, 那么这个匹配是一个 **完美匹配 (perfect matching)**。如果在一个匹配中只有一个点不被匹配, 那么这个匹配是一个 **准完美匹配 (near-perfect matching)**。

求一张普通图或二分图的匹配或完美匹配个数都是 **#P 完全** 的。

对于一个匹配 M , 若一条路径以非匹配点为起点, 每相邻两条边的其中一条在匹配中而另一条不在匹配中, 则这条路径被称作一条 **交替路径 (alternating path)**; 一条在非匹配点终止的交替路径, 被称作一条 **增广路径 (augmenting path)**。

托特定理: n 阶无向图 G 有完美匹配当且仅当对于任意的 $V' \subset V(G)$, $p_{\text{odd}}(G - V') \leq |V'|$, 其中 p_{odd} 表示奇数阶连通分支数。

托特定理 (推论): 任何无桥 3 - 正则图都有完美匹配。

点覆盖

对于图 $G = (V, E)$, 若 $V' \subseteq V$ 且 $\forall e \in E$ 满足 e 的至少一个端点在 V' 中, 则称 V' 是图 G 的一个 **点覆盖 (vertex cover)**。

点覆盖集必为支配集, 但极小点覆盖集不一定是极小支配集。

一个点集是点覆盖的充要条件是其补集是独立集, 因此最小点覆盖的补集是最大独立集。求一张图的最小点覆盖是 **NP 困难** 的。

一张图的任何一个匹配的大小都不超过其任何一个点覆盖的大小。完全二分图 $K_{n,m}$ 的最大匹配和最小点覆盖大小都为 $\min(n, m)$ 。

边覆盖

对于图 $G = (V, E)$, 若 $E' \subseteq E$ 且 $\forall v \in V$ 满足 v 与 E' 中的至少一条边相邻, 则称 E' 是图 G 的一个 **边覆盖 (edge cover)**。

最小边覆盖的大小记作 $\rho(G)$, 可以由最大匹配贪心扩展求得: 对于所有非匹配点, 将其一条邻边加入最大匹配中, 即得到了一个最小边覆盖。

最大匹配也可以由最小边覆盖求得: 对于最小边覆盖中每对有公共点的边删去其中一条。

一张图的最小边覆盖的大小加上最大匹配的大小等于图的点数, 即 $\rho(G) + \nu(G) = |V(G)|$ 。

一张图的最大匹配的大小不超过最小边覆盖的大小, 即 $\nu(G) \leq \rho(G)$ 。特别地, 完美匹配一定是一个最小边覆盖, 这也是上式取到等号的唯一情况。

一张图的任何一个独立集的大小都不超过其任何一个边覆盖的大小。完全二分图 $K_{n,m}$ 的最大独立集和最小边覆盖大小都为 $\max(n, m)$ 。

团

对于图 $G = (V, E)$, 若 $V' \subseteq V$ 且 V' 中任意两个不同的顶点都相邻, 则 V' 是图 G 的一个 **团 (clique)**。团的导出子图是完全图。

如果一个团在加入任何一个顶点后都不再是一个团, 则这个团是一个 **极大团 (maximal clique)**。

一张图的最大团的大小记作 $\omega(G)$, 最大团的大小等于其补图最大独立集的大小, 即 $\omega(G) = \alpha(\bar{G})$ 。求一张图的最大团是 **NP 困难** 的。

参考资料

[OI 中转站 - 图论概念梳理](#)

[Wikipedia](#) (以及相关概念的对应词条)

离散数学（修订版），田文成 周禄新 编著，天津文学出版社，P184-187

戴一奇，胡冠章，陈卫。图论与代数结构 [M]. 北京：清华大学出版社，1995.

🕒 本页面最近更新：2025/8/5 15:53:45，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[ouuan](#), [CCXXXI](#), [Enter-tainer](#), [Backl1ght](#), [EndlessCheng](#), [Ir1d](#), [Tiphereth-A](#), [c-forrest](#), [Great-designer](#), [IceySakura](#), [Kaiser-Yang](#), [mgt](#), [shuzhouliu](#), [sshwy](#), [Steaunk](#), [StudyingFather](#), [xiaoh1024](#), [zidian257](#), [zjxx](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



网络流简介

本页面主要介绍网络流相关的基本知识。

概述

网络 (network) 是指一个特殊的有向图 $G = (V, E)$, 其与一般有向图的不同之处在于有容量和源汇点。

- E 中的每条边 (u, v) 都有一个被称为容量 (capacity) 的权值, 记作 $c(u, v)$ 。当 $(u, v) \notin E$ 时, 可以假定 $c(u, v) = 0$ 。
- V 中有两个特殊的点: 源点 (source) s 和汇点 (sink) t ($s \neq t$)。

对于网络 $G = (V, E)$, 流 (flow) 是一个从边集 E 到整数集或实数集的函数, 其满足以下性质。

1. 容量限制: 对于每条边, 流经该边的流量不得超过该边的容量, 即 $0 \leq f(u, v) \leq c(u, v)$;
2. 流守恒性: 除源汇点外, 任意结点 u 的净流量为 0。其中, 我们定义 u 的净流量为 $f(u) = \sum_{x \in V} f(u, x) - \sum_{x \in V} f(x, u)$ 。

对于网络 $G = (V, E)$ 和其上的流 f , 我们定义 f 的流量 $|f|$ 为 s 的净流量 $f(s)$ 。作为流守恒性的推论, 这也等于 t 的净流量的相反数 $-f(t)$ 。

对于网络 $G = (V, E)$, 如果 $\{S, T\}$ 是 V 的划分 (即 $S \cup T = V$ 且 $S \cap T = \emptyset$), 且满足 $s \in S, t \in T$, 则我们称 $\{S, T\}$ 是 G 的一个 $s-t$ 割 (cut)。我们定义 $s-t$ 割 $\{S, T\}$ 的容量为 $||S, T|| = \sum_{u \in S} \sum_{v \in T} c(u, v)$ 。

常见问题

常见的网络流问题包括但不限于以下类型问题。

- 最大流问题: 对于网络 $G = (V, E)$, 给每条边指定流量, 得到合适的流 f , 使得 f 的流量尽可能大。此时我们称 f 是 G 的最大流。
- 最小割问题: 对于网络 $G = (V, E)$, 找到合适的 $s-t$ 割 $\{S, T\}$, 使得 $\{S, T\}$ 的总容量尽可能小。此时我们称 $\{S, T\}$ 的总容量是 G 的最小割。
- 最小费用最大流问题: 在网络 $G = (V, E)$ 上, 对每条边给定一个权值 $w(u, v)$, 称为费用 (cost), 含义是单位流量通过 (u, v) 所花费的代价。对于 G 所有可能的最大流, 我们称其中总费用最小的一者为最小费用最大流。

我们将在稍后的章节中对它们进行详细介绍。

例题：网络流 24 题

网络流 24 题是中文互联网上广泛流传的一个题单（LibreOJ/洛谷），至少在 2010 年前后就已经存在。该题单引入了一些经典的将其他问题建模为网络流问题的技巧。由于时代的局限性，这些问题未必是最具代表性的网络流问题，但仍值得有志于算法竞赛的读者一阅。

🔧 本页面最近更新：2025/6/25 00:27:36，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[Ir1d](#), [StudyingFather](#), [MegaOwler](#), [sshwy](#), [MingqiHuang](#), [Nanarikom](#), [Tiphereth-A](#), [Anguei](#), [Chrogeek](#), [EndlessCheng](#), [Enter-tainer](#), [liaoyanxu](#), [Macesuted](#), [ouuan](#), [Xarfa](#), [Xeonacid](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



最小生成树

定义

在阅读下列内容之前，请务必阅读 [图论相关概念](#) 与 [树基础](#) 部分，并了解以下定义：

1. 生成子图
2. 生成树

我们定义无向连通图的 **最小生成树** (Minimum Spanning Tree, MST) 为边权和最小的生成树。

注意：只有连通图才有生成树，而对于非连通图，只存在生成森林。

Kruskal 算法

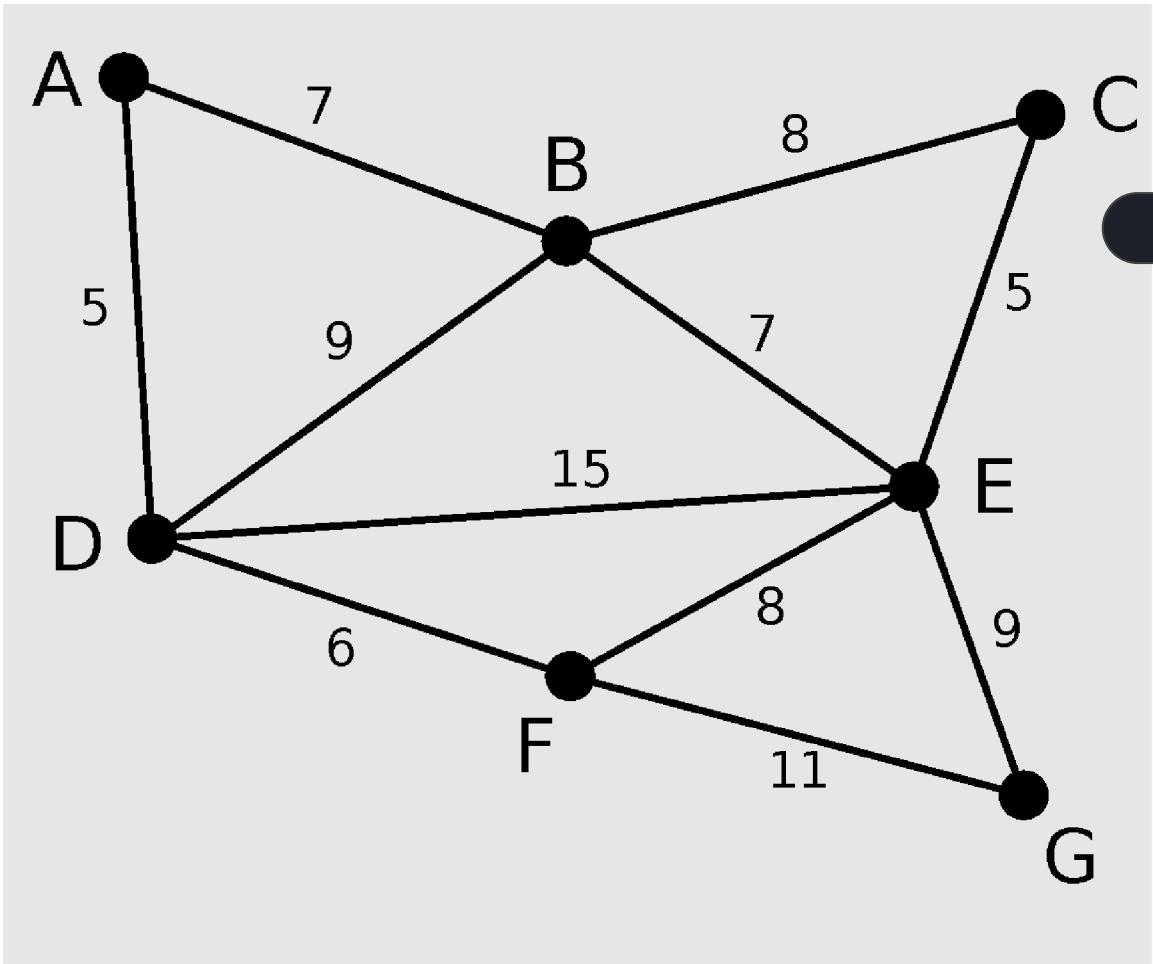
Kruskal 算法是一种常见并且好写的最小生成树算法，由 Kruskal 发明。该算法的基本思想是从小到大加入边，是个贪心算法。

前置知识

[并查集](#)、[贪心](#)、[图的存储](#)。

实现

图示：



伪代码：

```

1 Input. The edges of the graph  $e$ , where each element in  $e$  is  $(u, v, w)$   

   denoting that there is an edge between  $u$  and  $v$  weighted  $w$ .  

2 Output. The edges of the MST of the input graph.  

3 Method.  

4  $result \leftarrow \emptyset$   

5 sort  $e$  into nondecreasing order by weight  $w$   

6 for each  $(u, v, w)$  in the sorted  $e$   

7     if  $u$  and  $v$  are not connected in the union-find set  

8         connect  $u$  and  $v$  in the union-find set  

9          $result \leftarrow result \cup \{(u, v, w)\}$   

10 return  $result$ 
```

算法虽简单，但需要相应的数据结构来支持.....具体来说，维护一个森林，查询两个结点是否在同一棵树中，连接两棵树。

抽象一点地说，维护一堆 **集合**，查询两个元素是否属于同一集合，合并两个集合。

其中，查询两点是否连通和连接两点可以使用并查集维护。

如果使用 $O(m \log m)$ 的排序算法，并且使用 $O(m\alpha(m, n))$ 或 $O(m \log n)$ 的并查集，就可以得到时间复杂度为 $O(m \log m)$ 的 Kruskal 算法。

证明

思路很简单，为了造出一棵最小生成树，我们从最小边权的边开始，按边权从小到大依次加入，如果某次加边产生了环，就扔掉这条边，直到加入了 $n - 1$ 条边，即形成了一棵树。

证明：使用归纳法，证明任何时候 K 算法选择的边集都被某棵 MST 所包含。

基础：对于算法刚开始时，显然成立（最小生成树存在）。

归纳：假设某时刻成立，当前边集为 F ，令 T 为这棵 MST，考虑下一条加入的边 e 。

如果 e 属于 T ，那么成立。

否则， $T + e$ 一定存在一个环，考虑这个环上不属于 F 的另一条边 f （至少存在一条）。

首先， f 的权值一定不会比 e 小，不然 f 会在 e 之前被选取。

然后， f 的权值一定不会比 e 大，不然 $T + e - f$ 就是一棵比 T 还优的生成树了。

所以， $T + e - f$ 包含了 F ，并且也是一棵最小生成树，归纳成立。

例题



洛谷 P1195 口袋的天空



有 n 朵云，你要将它们连成 k 个棉花糖，将 X_i 云朵和 Y_i 连接起来需要 L_i 的代价，求最小代价。



例题代码



C++

```
1 #include <algorithm>
2 #include <iostream>
3 using namespace std;
4
5 int fa[1010]; // 定义父亲
6 int n, m, k;
7
8 struct edge {
9     int u, v, w;
10};
11
12 int l;
13 edge g[10010];
14
15 void add(int u, int v, int w) {
16     l++;
17     g[l].u = u;
18     g[l].v = v;
19     g[l].w = w;
20 }
21
22 // 标准并查集
23 int findroot(int x) { return fa[x] == x ? x : fa[x] = findroot(fa[x]); }
24
25 void Merge(int x, int y) {
26     x = findroot(x);
27     y = findroot(y);
28     fa[x] = y;
29 }
30
31
32 bool cmp(edge A, edge B) { return A.w < B.w; }
33
34 // Kruskal 算法
35 void Kruskal() {
36     int tot = 0; // 存已选了的边数
37     int ans = 0; // 存总的代价
38     for (int i = 1; i <= m; i++) {
39         int xr = findroot(g[i].u), yr = findroot(g[i].v);
40         if (xr != yr) { // 如果父亲不一样
41             Merge(xr, yr); // 合并
42             tot++; // 边数增加
43             ans += g[i].w; // 代价增加
44             if (tot == n - k) { // 检查选的边数是否满足 k 个棉花糖
45                 cout << ans << '\n';
46                 return;
47             }
48     }
49 }
```

```

48     }
49   }
50   cout << "No Answer\n"; // 无法连成
51 }
52
53 int main() {
54   cin >> n >> m >> k;
55   if (n == k) { // 特判边界情况
56     cout << "0\n";
57     return 0;
58   }
59   for (int i = 1; i <= n; i++) { // 初始化
60     fa[i] = i;
61   }
62   for (int i = 1; i <= m; i++) {
63     int u, v, w;
64     cin >> u >> v >> w;
65     add(u, v, w); // 添加边
66   }
67   sort(g + 1, g + m + 1, cmp); // 先按边权排序
68   kruskal();
69   return 0;
}

```

Python

```

1 class Edge:
2   def __init__(self, u, v, w):
3     self.u = u
4     self.v = v
5     self.w = w
6
7
8   fa = [0] * 1010 # 定义父亲
9   g = []
10
11
12   def add(u, v, w):
13     g.append(Edge(u, v, w))
14
15
16   # 标准并查集
17   def findroot(x):
18     if fa[x] == x:
19       return x
20     fa[x] = findroot(fa[x])
21     return fa[x]
22
23
24   def Merge(x, y):
25     x = findroot(x)

```

```

26     y = findroot(y)
27     fa[x] = y
28
29
30 # Kruskal 算法
31 def kruskal():
32     tot = 0 # 存已选了的边数
33     ans = 0 # 存总的代价
34     for e in g:
35         x = findroot(e.u)
36         y = findroot(e.v)
37         if x != y: # 如果父亲不一样
38             fa[x] = y # 合并
39             tot += 1 # 边数增加
40             ans += e.w # 代价增加
41         if tot == n - k: # 检查选的边数是否满足 k 个棉花糖
42             print(ans)
43             return
44     print("No Answer") # 无法连成
45
46
47 if __name__ == "__main__":
48     n, m, k = map(int, input().split())
49     if n == k: # 特判边界情况
50         print("0")
51         exit()
52     for i in range(1, n + 1): # 初始化
53         fa[i] = i
54     for i in range(1, m + 1):
55         u, v, w = map(int, input().split())
56         add(u, v, w) # 添加边
57     g.sort(key=lambda edge: edge.w) # 先按边权排序
58     kruskal()

```

Java

```

1 import java.util.Arrays;
2 import java.util.Scanner;
3
4 class Edge {
5     int u;
6     int v;
7     int w;
8
9     Edge(int u, int v, int w) {
10         this.u = u;
11         this.v = v;
12         this.w = w;
13     }
14 }
15

```

```

16 public class Main {
17     static int[] parent = new int[1010]; // 定义父亲
18     static int m, n, k; // n 表示点的数量， m 表示边的数量， k 表示
19     需要的棉花糖个数
20
21     static Edge[] edges = new Edge[10010];
22     static int l;
23
24     static void addEdge(int u, int v, int w) {
25         edges[++l] = new Edge(u, v, w);
26     }
27
28     // 标准并查集
29     static int findroot(int x) {
30         if (parent[x] != x) {
31             parent[x] = findroot(parent[x]);
32         }
33         return parent[x];
34     }
35
36     static void Merge(int x, int y) {
37         x = findroot(x);
38         y = findroot(y);
39         parent[x] = y;
40     }
41
42     static boolean cmp(Edge A, Edge B) {
43         return A.w < B.w;
44     }
45
46     // Kruskal 算法
47     static void kruskal() {
48         int tot = 0; // 存已选了的边数
49         int ans = 0; // 存总的代价
50
51         for (int i = 1; i <= m; i++) {
52             int xr = findroot(edges[i].u);
53             int yr = findroot(edges[i].v);
54             if (xr != yr) { // 如果父亲不一样
55                 Merge(xr, yr); // 合并
56                 tot++; // 边数增加
57                 ans += edges[i].w; // 代价增加
58                 if (tot == n - k) { // 检查选的边数是否满足 k 个棉
59                     花糖
60                     System.out.println(ans);
61                     return;
62                 }
63             }
64         }
65         System.out.println("No Answer"); // 无法连成
66     }
67

```

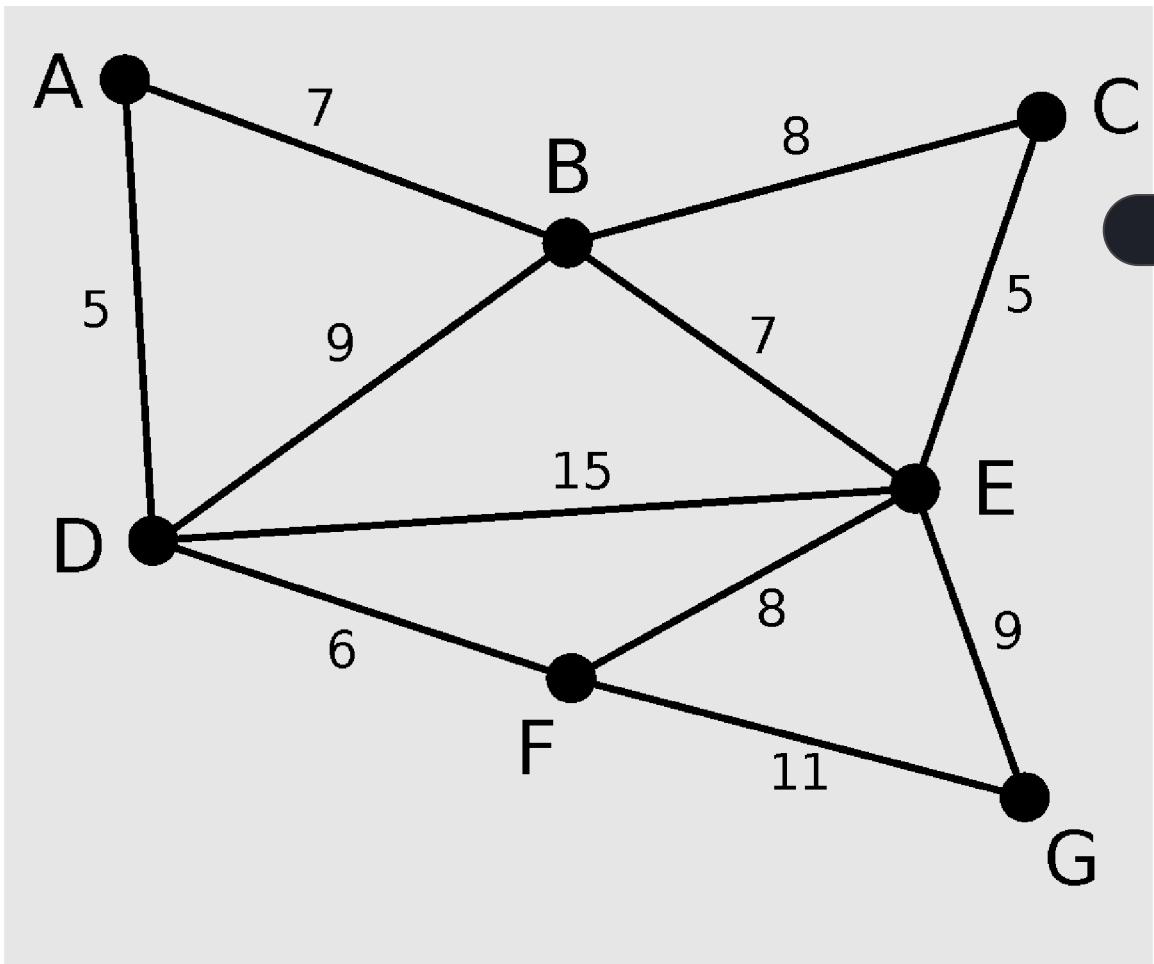
```
68     public static void main(String[] args) {
69         Scanner scanner = new Scanner(System.in);
70         n = scanner.nextInt();
71         m = scanner.nextInt();
72         k = scanner.nextInt();
73
74         if (n == k) { // 特判边界情况
75             System.out.println("0");
76             return;
77         }
78
79         // 初始化
80         for (int i = 1; i <= n; i++) {
81             parent[i] = i;
82         }
83         for (int i = 1; i <= m; i++) {
84             int u = scanner.nextInt();
85             int v = scanner.nextInt();
86             int w = scanner.nextInt();
87             addEdge(u, v, w); // 添加边
88         }
89         Arrays.sort(edges, 1, m + 1, (a, b) ->
90         Integer.compare(a.w, b.w)); // 先按边权排序
91         kruskal();
92         scanner.close();
93     }
94 }
```

Prim 算法

Prim 算法是另一种常见并且好写的最小生成树算法。该算法的基本思想是从一个结点开始，不断加点（而不是 Kruskal 算法的加边）。

实现

图示：



具体来说，每次要选择距离最小的一个结点，以及用新的边更新其他结点的距离。

其实跟 Dijkstra 算法一样，每次找到距离最小的一个点，可以暴力找也可以用堆维护。

堆优化的方式类似 Dijkstra 的堆优化，但如果使用二叉堆等不支持 $O(1)$ decrease-key 的堆，复杂度就不优于 Kruskal，常数也比 Kruskal 大。所以，一般情况下都使用 Kruskal 算法，在稠密图尤其是完全图上，暴力 Prim 的复杂度比 Kruskal 优，但 **不一定** 实际跑得更快。

暴力： $O(n^2 + m)$ 。

二叉堆： $O((n + m) \log n)$ 。

Fib 堆： $O(n \log n + m)$ 。

伪代码：

```
1 Input. The nodes of the graph  $V$ ; the function  $g(u, v)$  which  
means the weight of the edge  $(u, v)$ ; the function  $adj(v)$  which  
means the nodes adjacent to  $v$ .  
2 Output. The sum of weights of the MST of the input graph.  
3 Method.  
4  $result \leftarrow 0$   
5 choose an arbitrary node in  $V$  to be the  $root$   
6  $dis(root) \leftarrow 0$   
7 for each node  $v \in (V - \{root\})$   
8      $dis(v) \leftarrow \infty$   
9      $rest \leftarrow V$   
10   while  $rest \neq \emptyset$   
11      $cur \leftarrow$  the node with the minimum  $dis$  in  $rest$   
12      $result \leftarrow result + dis(cur)$   
13      $rest \leftarrow rest - \{cur\}$   
14     for each node  $v \in adj(cur)$   
15          $dis(v) \leftarrow \min(dis(v), g(cur, v))$   
16 return  $result$ 
```

注意：上述代码只是求出了最小生成树的权值，如果要输出方案还需要记录每个点的 dis 代表的是哪条边。

代码实现

```
1 // 使用二叉堆优化的 Prim 算法。
2 #include <cstring>
3 #include <iostream>
4 #include <queue>
5 using namespace std;
6 constexpr int N = 5050, M = 2e5 + 10;
7
8 struct E {
9     int v, w, x;
10 } e[M * 2];
11
12 int n, m, h[N], cnte;
13
14 void adde(int u, int v, int w) { e[++cnte] = E{v, w, h[u]}, h[u]
15 = cnte; }
16
17 struct S {
18     int u, d;
19 };
20
21 bool operator<(const S &x, const S &y) { return x.d > y.d; }
22
23 priority_queue<S> q;
24 int dis[N];
25 bool vis[N];
26
27 int res = 0, cnt = 0;
28
29 void Prim() {
30     memset(dis, 0x3f, sizeof(dis));
31     dis[1] = 0;
32     q.push({1, 0});
33     while (!q.empty()) {
34         if (cnt >= n) break;
35         int u = q.top().u, d = q.top().d;
36         q.pop();
37         if (vis[u]) continue;
38         vis[u] = true;
39         ++cnt;
40         res += d;
41         for (int i = h[u]; i; i = e[i].x) {
42             int v = e[i].v, w = e[i].w;
43             if (w < dis[v]) {
44                 dis[v] = w, q.push({v, w});
45             }
46         }
47     }
48 }
49 }
```

```

50 int main() {
51     cin >> n >> m;
52     for (int i = 1, u, v, w; i <= m; ++i) {
53         cin >> u >> v >> w, adde(u, v, w), adde(v, u, w);
54     }
55     Prim();
56     if (cnt == n)
57         cout << res;
58     else
59         cout << "No MST.";
60     return 0;
}

```

证明

从任意一个结点开始，将结点分成两类：已加入的，未加入的。

每次从未加入的结点中，找一个与已加入的结点之间边权最小值最小的结点。

然后将这个结点加入，并连上那条边权最小的边。

重复 $n - 1$ 次即可。

证明：还是说明在每一步，都存在一棵最小生成树包含已选边集。

基础：只有一个结点的时候，显然成立。

归纳：如果某一步成立，当前边集为 F ，属于 T 这棵 MST，接下来要加入边 e 。

如果 e 属于 T ，那么成立。

否则考虑 $T + e$ 中环上另一条可以加入当前边集的边 f 。

首先， f 的权值一定不小于 e 的权值，否则就会选择 f 而不是 e 了。

然后， f 的权值一定不大于 e 的权值，否则 $T + e - f$ 就是一棵更小的生成树了。

因此， e 和 f 的权值相等， $T + e - f$ 也是一棵最小生成树，且包含了 F 。

Boruvka 算法

接下来介绍另一种求解最小生成树的算法——Boruvka 算法。该算法的思想是前两种算法的结合。它可以用于求解无向图的最小生成森林。（无向连通图就是最小生成树。）

在边具有较多特殊性质的问题中，Boruvka 算法具有优势。例如 [CF888G](#) 的完全图问题。

为了描述该算法，我们需要引入一些定义：

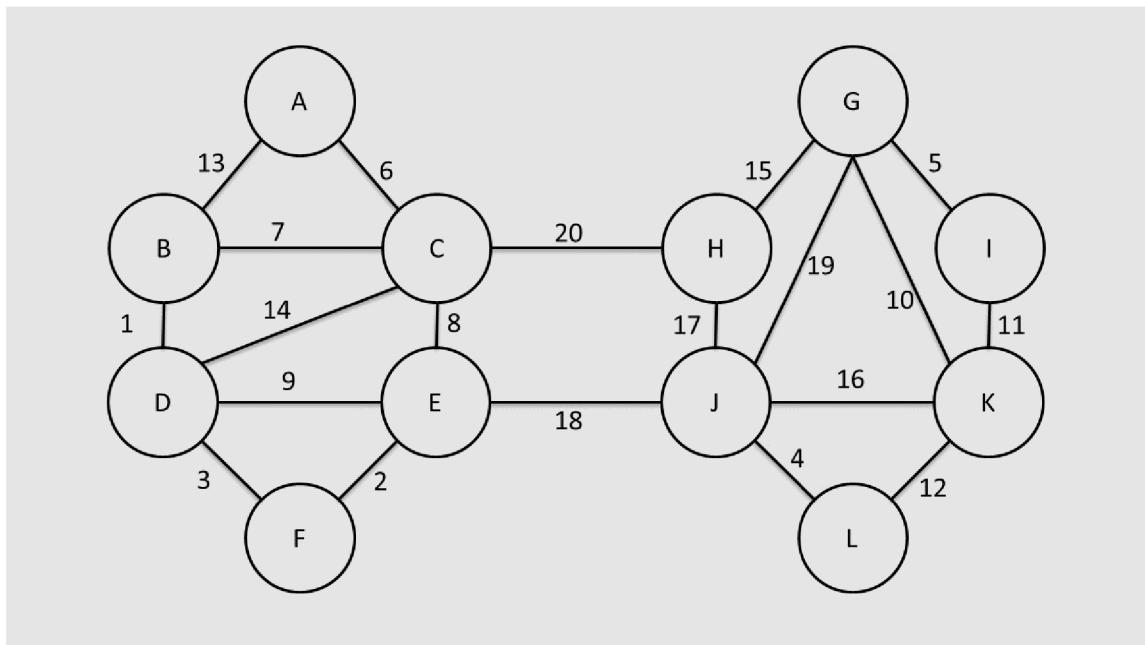
1. 定义 E' 为我们当前找到的最小生成森林的边。在算法执行过程中，我们逐步向 E' 加边，定义 **连通块** 表示一个点集 $V' \subseteq V$ ，且这个点集中的任意两个点 u, v 在 E' 中的边构成的子图上是连通的（互相可达）。

2. 定义一个连通块的 **最小边** 为它连向其它连通块的边中权值最小的那一条。

初始时， $E' = \emptyset$ ，每个点各自是一个连通块：

1. 计算每个点分别属于哪个连通块。将每个连通块都设为「没有最小边」。
2. 遍历每条边 (u, v) ，如果 u 和 v 不在同一个连通块，就用这条边的边权分别更新 u 和 v 所在连通块的最小边。
3. 如果所有连通块都没有最小边，退出程序，此时的 E' 就是原图最小生成森林的边集。否则，将每个有最小边的连通块的最小边加入 E' ，返回第一步。

下面通过一张动态图来举一个例子（图源自 [维基百科](#)）：



当原图连通时，每次迭代连通块数量至少减半，算法只会迭代不超过 $O(\log V)$ 次，而原图不连通时相当于多个子问题，因此算法复杂度是 $O(E \log V)$ 的。给出算法的伪代码：(修改自 [维基百科](#))

```

1  Input. A graph  $G$  whose edges have distinct weights.
2  Output. The minimum spanning forest of  $G$ .
3  Method.
4  Initialize a forest  $F$  to be a set of one-vertex trees
5  while True
6      Find the components of  $F$  and label each vertex of  $G$  by its component
7      Initialize the cheapest edge for each component to "None"
8      for each edge  $(u, v)$  of  $G$ 
9          if  $u$  and  $v$  have different component labels
10         if  $(u, v)$  is cheaper than the cheapest edge for the component of  $u$ 
11             Set  $(u, v)$  as the cheapest edge for the component of  $u$ 
12         if  $(u, v)$  is cheaper than the cheapest edge for the component of  $v$ 
13             Set  $(u, v)$  as the cheapest edge for the component of  $v$ 
14         if all components'cheapest edges are "None"
15             return  $F$ 
16         for each component whose cheapest edge is not "None"
17             Add its cheapest edge to  $F$ 

```

需要注意边与边的比较通常需要第二关键字（例如按编号排序），以便当边权相同时分出边的大小。

习题

- 「HAOI2006」聪明的猴子
- 「SCOI2005」繁忙的都市

最小生成树的唯一性

考虑最小生成树的唯一性。如果一条边 **不在最小生成树的边集中**，并且可以替换与其 **权值相同、并且在最小生成树边集** 的另一条边。那么，这个最小生成树就是不唯一的。

对于 Kruskal 算法，只要计算为当前权值的边可以放几条，实际放了几条，如果这两个值不一样，那么就说明这几条边与之前的边产生了一个环（这个环中至少有两条当前权值的边，否则根据并查集，这条边是不能放的），即最小生成树不唯一。

寻找权值与当前边相同的边，我们只需要记录头尾指针，用单调队列即可在 $O(\alpha(m))$ (m 为边数) 的时间复杂度里优秀解决这个问题（基本与原算法时间相同）。



例题：POJ 1679

```
1 #include <algorithm>
2 #include <iostream>
3
4 struct Edge {
5     int x, y, z;
6 };
7
8 int f[100001];
9 Edge a[100001];
10
11 int cmp(const Edge& a, const Edge& b) { return a.z < b.z; }
12
13 int find(int x) { return f[x] == x ? x : f[x] = find(f[x]); }
14
15 using std::cin;
16 using std::cout;
17
18 int main() {
19     cin.tie(nullptr)->sync_with_stdio(false);
20     int t;
21     cin >> t;
22     while (t--) {
23         int n, m;
24         cin >> n >> m;
25         for (int i = 1; i <= n; i++) f[i] = i;
26         for (int i = 1; i <= m; i++) cin >> a[i].x >> a[i].y >>
27             a[i].z;
28         std::sort(a + 1, a + m + 1, cmp); // 先排序
29         int num = 0, ans = 0, tail = 0, sum1 = 0, sum2 = 0;
30         bool flag = true;
31         for (int i = 1; i <= m + 1; i++) { // 再并查集加边
32             if (i > tail) {
33                 if (sum1 != sum2) {
34                     flag = false;
35                     break;
36                 }
37                 sum1 = 0;
38                 for (int j = i; j <= m + 1; j++) {
39                     if (j > m || a[j].z != a[i].z) {
40                         tail = j - 1;
41                         break;
42                     }
43                     if (find(a[j].x) != find(a[j].y)) ++sum1;
44                 }
45                 sum2 = 0;
46             }
47             if (i > m) break;
48             int x = find(a[i].x);
49             int y = find(a[i].y);
```

```

50         if (x != y && num != n - 1) {
51             sum2++;
52             num++;
53             f[x] = f[y];
54             ans += a[i].z;
55         }
56     }
57     if (flag)
58         cout << ans << '\n';
59     else
60         cout << "Not Unique!\n";
61     }
62     return 0;
}

```

次小生成树

非严格次小生成树

定义

在无向图中，边权和最小的满足边权和 **大于等于** 最小生成树边权和的生成树

求解方法

- 求出无向图的最小生成树 T ，设其权值和为 M
- 遍历每条未被选中的边 $e = (u, v, w)$ ，找到 T 中 u 到 v 路径上边权最大的一条边 $e' = (s, t, w')$ ，则在 T 中以 e 替换 e' ，可得一棵权值和为 $M' = M + w - w'$ 的生成树 T' .
- 对所有替换得到的答案 M' 取最小值即可

如何求 u, v 路径上的边权最大值呢？

我们可以使用倍增来维护，预处理出每个节点的 2^i 级祖先及到达其 2^i 级祖先路径上最大的边权，这样在倍增求 LCA 的过程中可以直接求得。

严格次小生成树

定义

在无向图中，边权和最小的满足边权和 **严格大于** 最小生成树边权和的生成树

求解方法

考虑刚才的非严格次小生成树求解过程，为什么求得的解是非严格的？

因为最小生成树保证生成树中 u 到 v 路径上的边权最大值一定 **不大于** 其他从 u 到 v 路径的边权最大值。换言之，当我们用于替换的边的权值与原生成树中被替换边的权值相等时，得到的次小

生成树是非严格的。

解决的办法很自然：我们维护到 2^i 级祖先路径上的最大边权的同时维护 **严格次大边权**，当用于替换的边的权值与原生成树中路径最大边权相等时，我们用严格次大值来替换即可。

这个过程可以用倍增求解，复杂度 $O(m \log m)$ 。



代码实现

```
1 #include <algorithm>
2 #include <iostream>
3
4 constexpr int INF = 0x3fffffff;
5 constexpr long long INF64 = 0x3fffffffffffffLL;
6
7 struct Edge {
8     int u, v, val;
9
10    bool operator<(const Edge &other) const { return val <
11        other.val; }
12 };
13
14 Edge e[300010];
15 bool used[300010];
16
17 int n, m;
18 long long sum;
19
20 class Tr {
21 private:
22     struct Edge {
23         int to, nxt, val;
24     } e[600010];
25
26     int cnt, head[100010];
27
28     int pnt[100010][22];
29     int dpth[100010];
30     // 到祖先的路径上边权最大的边
31     int maxx[100010][22];
32     // 到祖先的路径上边权次大的边，若不存在则为 -INF
33     int minn[100010][22];
34
35 public:
36     void addedge(int u, int v, int val) {
37         e[++cnt] = Edge{v, head[u], val};
38         head[u] = cnt;
39     }
40
41     void insedge(int u, int v, int val) {
42         addedge(u, v, val);
43         addedge(v, u, val);
44     }
45
46     void dfs(int now, int fa) {
47         dpth[now] = dpth[fa] + 1;
48         pnt[now][0] = fa;
49         minn[now][0] = -INF;
```

```

50     for (int i = 1; (1 << i) <= dpth[now]; i++) {
51         pnt[now][i] = pnt[pnt[now][i - 1]][i - 1];
52         int kk[4] = {maxx[now][i - 1], maxx[pnt[now][i - 1]][i -
53             1],
54                 minn[now][i - 1], minn[pnt[now][i - 1]][i -
55             1]};
56         // 从四个值中取得最大值
57         std::sort(kk, kk + 4);
58         maxx[now][i] = kk[3];
59         // 取得严格次大值
60         int ptr = 2;
61         while (ptr >= 0 && kk[ptr] == kk[3]) ptr--;
62         minn[now][i] = (ptr == -1 ? -INF : kk[ptr]);
63     }
64
65     for (int i = head[now]; i; i = e[i].nxt) {
66         if (e[i].to != fa) {
67             maxx[e[i].to][0] = e[i].val;
68             dfs(e[i].to, now);
69         }
70     }
71 }
72
73 int lca(int a, int b) {
74     if (dpth[a] < dpth[b]) std::swap(a, b);
75
76     for (int i = 21; i >= 0; i--)
77         if (dpth[pnt[a][i]] >= dpth[b]) a = pnt[a][i];
78
79     if (a == b) return a;
80
81     for (int i = 21; i >= 0; i--) {
82         if (pnt[a][i] != pnt[b][i]) {
83             a = pnt[a][i];
84             b = pnt[b][i];
85         }
86     }
87     return pnt[a][0];
88 }
89
90 int query(int a, int b, int val) {
91     int res = -INF;
92     for (int i = 21; i >= 0; i--) {
93         if (dpth[pnt[a][i]] >= dpth[b]) {
94             if (val != maxx[a][i])
95                 res = std::max(res, maxx[a][i]);
96             else
97                 res = std::max(res, minn[a][i]);
98             a = pnt[a][i];
99         }
100    }
101    return res;

```

```

102     }
103 } tr;
104
105 int fa[100010];
106
107 int find(int x) { return fa[x] == x ? x : fa[x] = find(fa[x]); }
108
109 void Kruskal() {
110     int tot = 0;
111     std::sort(e + 1, e + m + 1);
112     for (int i = 1; i <= n; i++) fa[i] = i;
113
114     for (int i = 1; i <= m; i++) {
115         int a = find(e[i].u);
116         int b = find(e[i].v);
117         if (a != b) {
118             fa[a] = b;
119             tot++;
120             tr.insedge(e[i].u, e[i].v, e[i].val);
121             sum += e[i].val;
122             used[i] = true;
123         }
124         if (tot == n - 1) break;
125     }
126 }
127
128 int main() {
129     std::ios::sync_with_stdio(false);
130     std::cin.tie(nullptr);
131
132     std::cin >> n >> m;
133     for (int i = 1; i <= m; i++) {
134         int u, v, val;
135         std::cin >> u >> v >> val;
136         e[i] = Edge{u, v, val};
137     }
138
139     Kruskal();
140     long long ans = INF64;
141     tr.dfs(1, 0);
142
143     for (int i = 1; i <= m; i++) {
144         if (!used[i]) {
145             int _lca = tr.lca(e[i].u, e[i].v);
146             // 找到路径上不等于 e[i].val 的最大边权
147             long long tmpa = tr.query(e[i].u, _lca, e[i].val);
148             long long tmpb = tr.query(e[i].v, _lca, e[i].val);
149             // 这样的边可能不存在，只在这样的边存在时更新答案
150             if (std::max(tmpa, tmpb) > -INF)
151                 ans = std::min(ans, sum - std::max(tmpa, tmpb) +
152 e[i].val);
153         }

```

```
153     }
154 // 次小生成树不存在时输出 -1
155 std::cout << (ans == INF64 ? -1 : ans) << '\n';
156 return 0;
157 }
```

瓶颈生成树

定义

无向图 G 的瓶颈生成树是这样的一个生成树，它的最大的边权值在 G 的所有生成树中最小。

性质

最小生成树是瓶颈生成树的充分不必要条件。 即最小生成树一定是瓶颈生成树，而瓶颈生成树不一定是最小生成树。

关于最小生成树一定是瓶颈生成树这一命题，可以运用反证法证明：我们设最小生成树中的最大边权为 w ，如果最小生成树不是瓶颈生成树的话，则瓶颈生成树的所有边权都小于 w ，我们只需删去原最小生成树中的最长边，用瓶颈生成树中的一条边来连接删去边后形成的两棵树，得到的新生成树一定比原最小生成树的权值和还要小，这样就产生了矛盾。

例题



POJ 2395 Out of Hay

给出 n 个农场和 m 条边，农场按 1 到 n 编号，现在有一人要从编号为 1 的农场出发到其他的农场去，求在这途中他最多需要携带的水的重量，注意他每到达一个农场，可以对水进行补给，且要使总共的路径长度最小。题目要求的就是瓶颈树的最大边，可以通过求最小生成树来解决。

最小瓶颈路

定义

无向图 G 中 x 到 y 的最小瓶颈路是这样的一类简单路径，满足这条路径上的最大的边权在所有 x 到 y 的简单路径中是最小的。

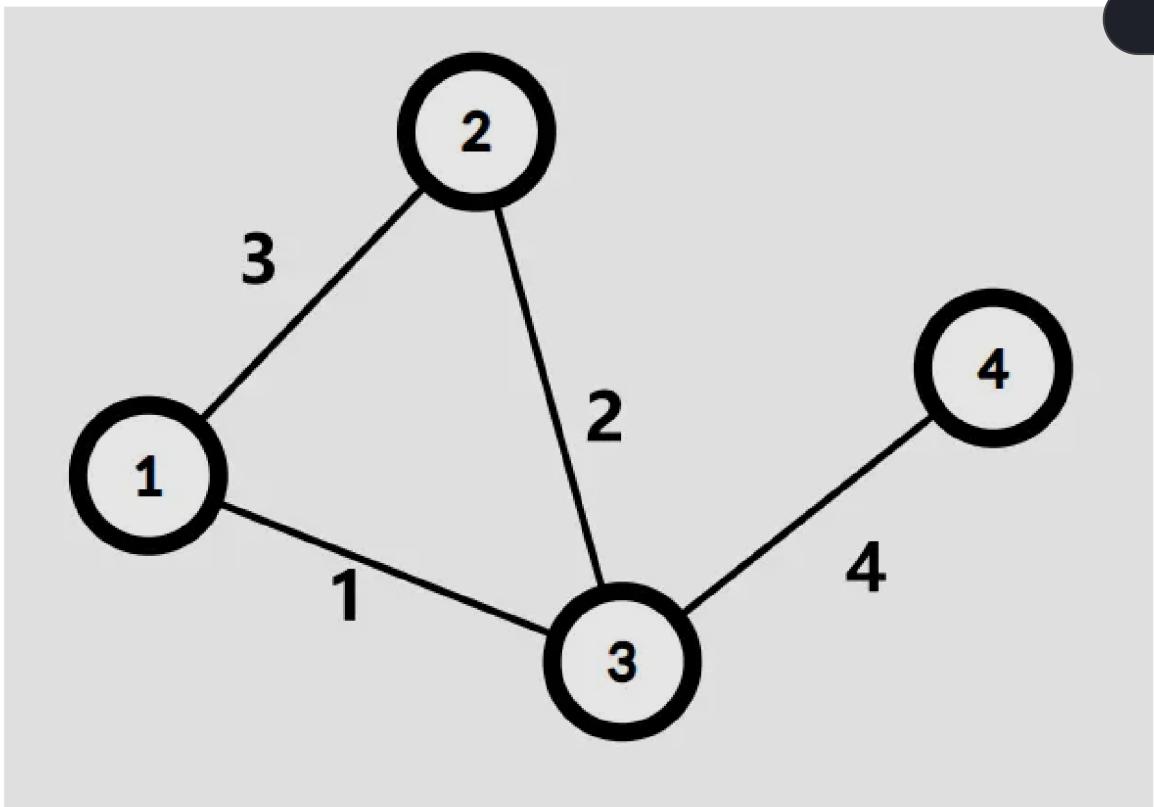
性质

根据最小生成树定义， x 到 y 的最小瓶颈路上的最大边权等于最小生成树上 x 到 y 路径上的最大边权。虽然最小生成树不唯一，但是每种最小生成树 x 到 y 路径的最大边权相同且为最小值。也

就是说，每种最小生成树上的 x 到 y 的路径均为最小瓶颈路。

但是，并不是所有最小瓶颈路都存在一棵最小生成树满足其为树上 x 到 y 的简单路径。

例如下图：



1 到 4 的最小瓶颈路显然有以下两条：1-2-3-4。1-3-4。

但是，1-2 不会出现在任意一种最小生成树上。

应用

由于最小瓶颈路不唯一，一般情况下会询问最小瓶颈路上的最大边权。

也就是说，我们需求最小生成树链上的 \max 。

倍增、树剖都可以解决，这里不再展开。

Kruskal 重构树

定义

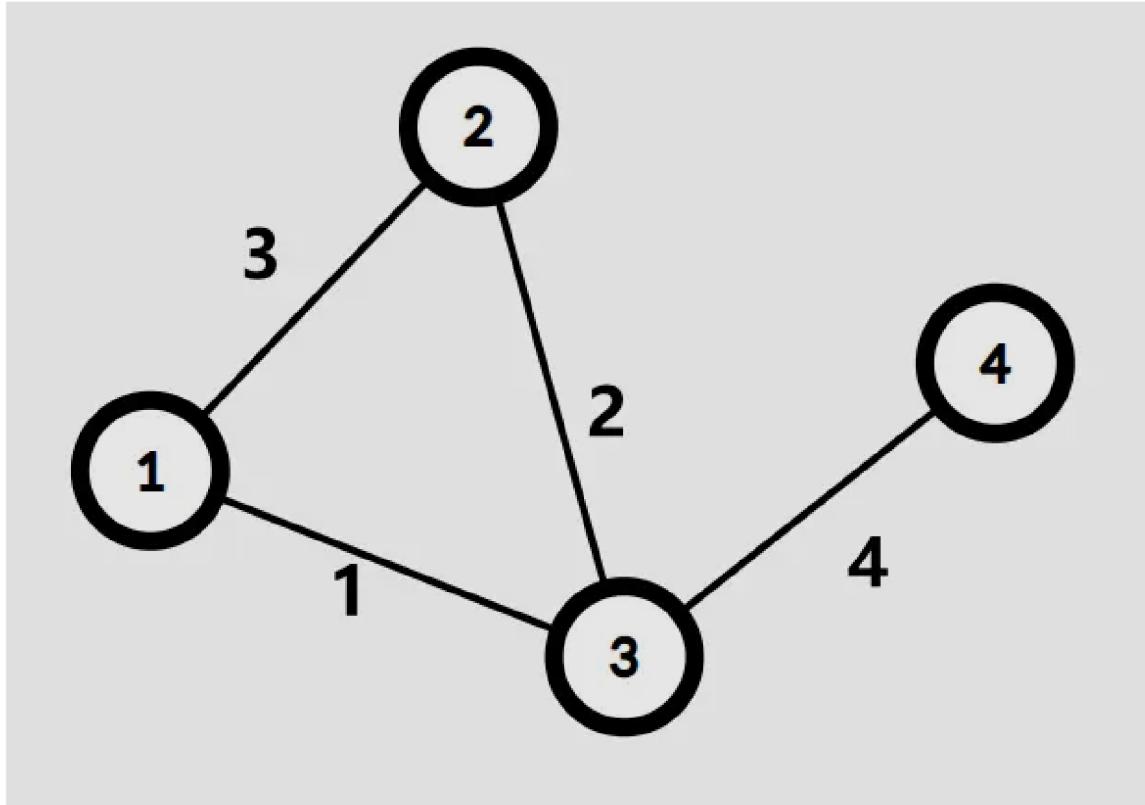
在跑 Kruskal 的过程中我们会从小到大加入若干条边。现在我们仍然按照这个顺序。

首先新建 n 个集合，每个集合恰有一个节点，点权为 0。

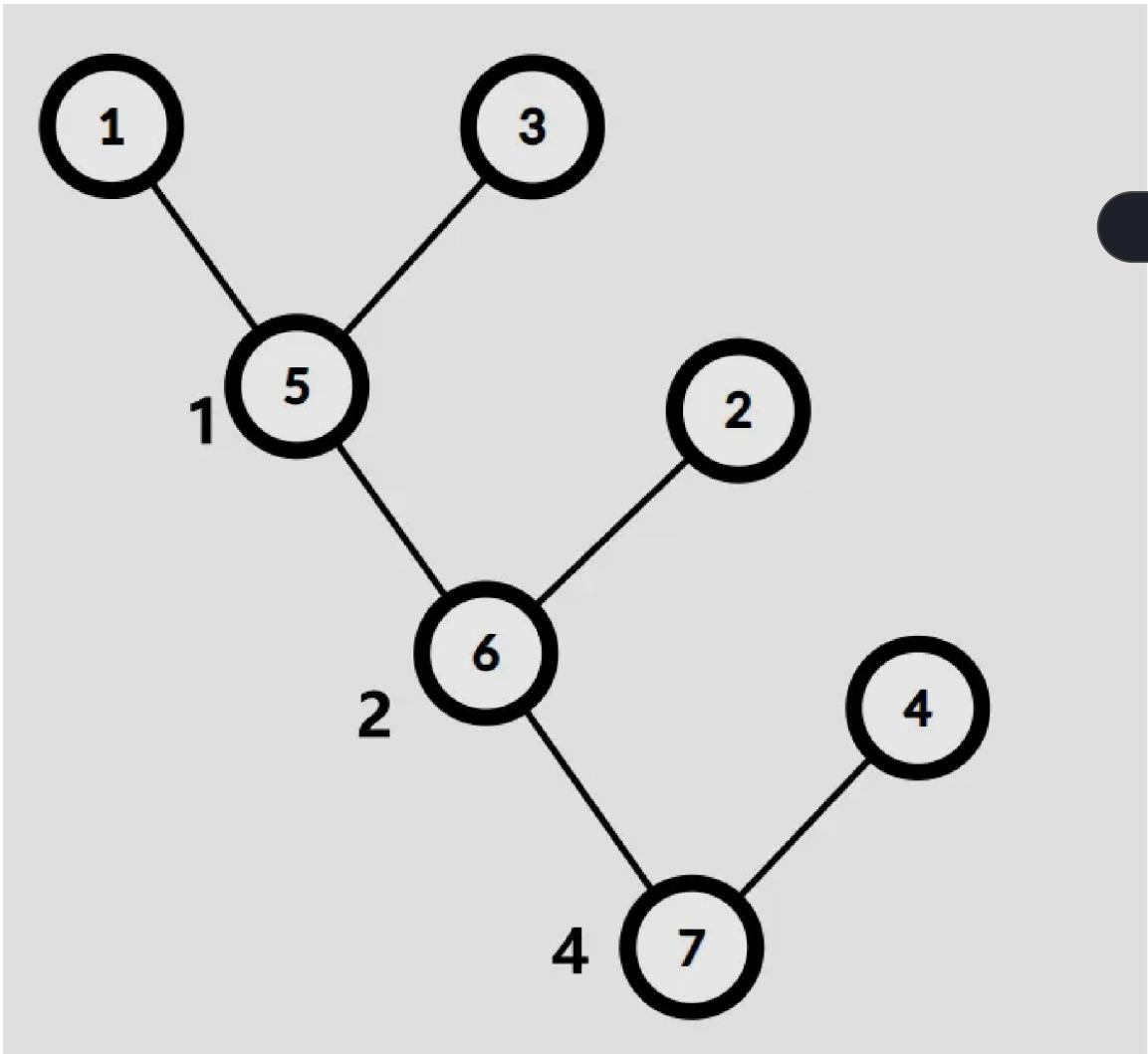
每一次加边会合并两个集合，我们可以新建一个点，点权为加入边的边权，同时将两个集合的根节点分别设为新建点的左儿子和右儿子。然后我们将两个集合和新建点合并成一个集合。将新建点设为根。

不难发现，在进行 $n - 1$ 轮之后我们得到了一棵恰有 n 个叶子的二叉树，同时每个非叶子节点恰好有两个儿子。这棵树就叫 Kruskal 重构树。

举个例子：



这张图的 Kruskal 重构树如下：



性质

不难发现，原图中两个点之间的所有简单路径上最大边权的最小值 = 最小生成树上两个点之间的简单路径上的最大值 = Kruskal 重构树上两点之间的 LCA 的权值。

也就是说，到点 x 的简单路径上最大边权的最小值 $\leq val$ 的所有点 y 均在 Kruskal 重构树上的某一棵子树内，且恰好为该子树的所有叶子节点。

我们在 Kruskal 重构树上找到 x 到根的路径上权值 $\leq val$ 的最浅的节点。显然这就是所有满足条件的节点所在的子树的根节点。

如果需要求原图中两个点之间的所有简单路径上最小边权的最大值，则在跑 Kruskal 的过程中按边权大到小的顺序加边。



「LOJ 137」最小瓶颈路 加强版



```
1 #include <algorithm>
2 #include <iostream>
3
4 using namespace std;
5
6 constexpr int MAX_VAL_RANGE = 280010;
7
8 int n, m, log2Values[MAX_VAL_RANGE + 1];
9
10 namespace TR {
11     struct Edge {
12         int to, nxt, val;
13     } e[400010];
14
15     int cnt, head[140010];
16
17     void addedge(int u, int v, int val = 0) {
18         e[++cnt] = Edge{v, head[u], val};
19         head[u] = cnt;
20     }
21
22     int val[140010];
23
24     namespace LCA {
25         int sec[280010], cnt;
26         int pos[140010];
27         int dpth[140010];
28
29         void dfs(int now, int fa) {
30             dpth[now] = dpth[fa] + 1;
31             sec[++cnt] = now;
32             pos[now] = cnt;
33
34             for (int i = head[now]; i; i = e[i].nxt) {
35                 if (fa != e[i].to) {
36                     dfs(e[i].to, now);
37                     sec[++cnt] = now;
38                 }
39             }
40         }
41
42         int dp[280010][20];
43
44         void init() {
45             dfs(2 * n - 1, 0);
46             for (int i = 1; i <= 4 * n; i++) {
47                 dp[i][0] = sec[i];
48             }
49             for (int j = 1; j <= 19; j++) {
```

```

50     for (int i = 1; i + (1 << j) - 1 <= 4 * n; i++) {
51         dp[i][j] = dpth[dp[i][j - 1]] < dpth[dp[i + (1 << (j -
52             1))][j - 1]]
53             ? dp[i][j - 1]
54             : dp[i + (1 << (j - 1))][j - 1];
55     }
56 }
57 }

58 int lca(int x, int y) {
59     int l = pos[x], r = pos[y];
60     if (l > r) {
61         swap(l, r);
62     }
63     int k = log2Values[r - l + 1];
64     return dpth[dp[l][k]] < dpth[dp[r - (1 << k) + 1][k]]
65         ? dp[l][k]
66         : dp[r - (1 << k) + 1][k];
67     }
68 } // namespace LCA
69 } // namespace TR

70 using TR::addedge;

71
72 namespace GR {
73     struct Edge {
74         int u, v, val;
75
76         bool operator<(const Edge &other) const { return val <
77             other.val; }
78     } e[100010];
79
80     int fa[140010];
81
82     int find(int x) { return fa[x] == 0 ? x : fa[x] = find(fa[x]); }
83
84     void kruskal() { // 最小生成树
85         int tot = 0, cnt = n;
86         sort(e + 1, e + m + 1);
87         for (int i = 1; i <= m; i++) {
88             int fau = find(e[i].u), fav = find(e[i].v);
89             if (fau != fav) {
90                 cnt++;
91                 fa[fau] = fa[fav] = cnt;
92                 addedge(fau, cnt);
93                 addedge(cnt, fau);
94                 addedge(fav, cnt);
95                 addedge(cnt, fav);
96                 TR::val[cnt] = e[i].val;
97                 tot++;
98             }
99         }
100     }
101     if (tot == n - 1) {

```

```
102         break;
103     }
104 }
105 }
106 } // namespace GR
107
108 int ans;
109 int A, B, C, P;
110
111 int rnd() { return A = (A * B + C) % P; }
112
113 void initLog2() {
114     for (int i = 2; i <= MAX_VAL_RANGE; i++) {
115         log2Values[i] = log2Values[i >> 1] + 1;
116     }
117 }
118
119 int main() {
120     initLog2(); // 预处理
121     cin >> n >> m;
122     for (int i = 1; i <= m; i++) {
123         int u, v, val;
124         cin >> u >> v >> val;
125         GR::e[i] = GR::Edge{u, v, val};
126     }
127     GR::kruskal();
128     TR::LCA::init();
129     int Q;
130     cin >> Q;
131     cin >> A >> B >> C >> P;
132
133     while (Q--) {
134         int u = rnd() % n + 1, v = rnd() % n + 1;
135         ans += TR::val[TR::LCA::lca(u, v)];
136         ans %= 1000000007;
137     }
138     cout << ans;
139     return 0;
140 }
```



首先预处理出来每一个点到根节点的最短路。

我们构造出来根据海拔的最大生成树。显然每次询问可以到达的节点是在最大生成树中和询问点的路径上最小边权 $> p$ 的节点。

根据 Kruskal 重构树的性质，这些节点满足均在一棵子树内同时为其所有叶子节点。

也就是说，我们只需要求出 Kruskal 重构树上每一棵子树叶子的权值 \min 就可以支持子树询问。

询问的根节点可以使用 Kruskal 重构树上倍增的方式求出。

时间复杂度 $O((n + m + Q) \log n)$ 。



本页面最近更新：2025/7/11 13:49:57，[更新历史](#)

发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

本页面贡献者：[lr1d](#), [ouuan](#), [sshwy](#), [zhouyuyang2002](#), [Enter-tainer](#), [Marcythm](#), [partychicken](#), [bear-good](#), [HeRaNO](#), [billchenchina](#), [diauweb](#), [StudyingFather](#), [abc1763613206](#), [Chrogeek](#), [greyqz](#), [Hszzx](#), [renbaoshuo](#), [ShadowsEpic](#), [stevebraveman](#), [Tiphereth-A](#), [toprise](#), [Xeonacid](#), [y-kx-b](#), [ayuusweetfish](#), [Backl1ght](#), [CCXXXI](#), [ChungZH](#), [Fomalhauthmj](#), [Haohu Shen](#), [Henry-ZHR](#), [kawa-yoiko](#), [kenlig](#), [ksyx](#), [Menci](#), [mgt](#), [nalemy](#), [VLTHellolin](#), [xzdeyg](#), [ylxmf2005](#), [YOYO-UIAT](#), [yzxoi](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



最短路

定义

(还记得这些定义吗？在阅读下列内容之前，请务必了解 [图论相关概念](#) 中的基础部分。)

- 路径
- 最短路
- 有向图中的最短路、无向图中的最短路
- 单源最短路、每对结点之间的最短路

记号

为了方便叙述，这里先给出下文将会用到的一些记号的含义。

- n 为图上点的数目， m 为图上边的数目；
- s 为最短路的源点；
- $D(u)$ 为 s 点到 u 点的 **实际** 最短路长度；
- $dis(u)$ 为 s 点到 u 点的 **估计** 最短路长度。任何时候都有 $dis(u) \geq D(u)$ 。特别地，当最短路算法终止时，应有 $dis(u) = D(u)$ 。
- $w(u, v)$ 为 (u, v) 这一条边的边权。

性质

对于边权为正的图，任意两个结点之间的最短路，不会经过重复的结点。

对于边权为正的图，任意两个结点之间的最短路，不会经过重复的边。

对于边权为正的图，任意两个结点之间的最短路，任意一条的结点数不会超过 n ，边数不会超过 $n - 1$ 。

Floyd 算法

是用来求任意两个结点之间的最短路的。

复杂度比较高，但是常数小，容易实现（只有三个 `for`）。

适用于任何图，不管有向无向，边权正负，但是最短路必须存在。（不能有个负环）

实现

我们定义一个数组 $f[k][x][y]$ ，表示只允许经过结点 1 到 k （也就是说，在子图 $V' = 1, 2, \dots, k$ 中的路径，注意， x 与 y 不一定在这个子图中），结点 x 到结点 y 的最短路长度。

很显然， $f[n][x][y]$ 就是结点 x 到结点 y 的最短路长度（因为 $V' = 1, 2, \dots, n$ 即为 V 本身，其表示的最短路径就是所求路径）。

接下来考虑如何求出 f 数组的值。

$f[0][x][y]$ ： x 与 y 的边权，或者 0，或者 $+\infty$ ($f[0][x][y]$ 什么时候应该是 $+\infty$ ？当 x 与 y 间有直接相连的边的时候，为它们的边权；当 $x = y$ 的时候为零，因为到自身的距离为零；当 x 与 y 没有直接相连的边的时候，为 $+\infty$ ）。

$f[k][x][y] = \min(f[k-1][x][y], f[k-1][x][k] + f[k-1][k][y])$ ($f[k-1][x][y]$ ，为不经过 k 点的最短路径，而 $f[k-1][x][k] + f[k-1][k][y]$ ，为经过了 k 点的最短路)。

上面两行都显然是对的，所以说这个做法空间是 $O(N^3)$ ，我们需要依次增加问题规模（ k 从 1 到 n ），判断任意两点在当前问题规模下的最短路。

C++

```
1  for (k = 1; k <= n; k++) {
2      for (x = 1; x <= n; x++) {
3          for (y = 1; y <= n; y++) {
4              f[k][x][y] = min(f[k - 1][x][y], f[k - 1][x][k] + f[k - 1][k][y]);
5          }
6      }
7  }
```

Python

```
1  for k in range(1, n + 1):
2      for x in range(1, n + 1):
3          for y in range(1, n + 1):
4              f[k][x][y] = min(f[k - 1][x][y], f[k - 1][x][k] + f[k - 1][k]
[y])
```

因为第一维对结果无影响，我们可以发现数组的第一维是可以省略的，于是可以直接改成 $f[x][y] = \min(f[x][y], f[x][k] + f[k][y])$ 。



证明第一维对结果无影响

对于给定的 k ，当更新 $f[k][x][y]$ 时，涉及的元素总是来自 $f[k-1]$ 数组的第 k 行和第 k 列。然后我们可以发现，对于给定的 k ，当更新 $f[k][k][y]$ 或 $f[k][x][k]$ ，总是不会发生数值更新，因为按照公式 $f[k][k][y] = \min(f[k-1][k][y], f[k-1][k][k] + f[k-1][k][y])$ ， $f[k-1][k][k]$ 为 0，因此这个值总是 $f[k-1][k][y]$ ，对于 $f[k][x][k]$ 的证明类似。

因此，如果省略第一维，在给定的 k 下，每个元素的更新中使用到的元素都没有在这次迭代中更新，因此第一维的省略并不会影响结果。

C++

```
1 for (k = 1; k <= n; k++) {  
2     for (x = 1; x <= n; x++) {  
3         for (y = 1; y <= n; y++) {  
4             f[x][y] = min(f[x][y], f[x][k] + f[k][y]);  
5         }  
6     }  
7 }
```

Python

```
1 for k in range(1, n + 1):  
2     for x in range(1, n + 1):  
3         for y in range(1, n + 1):  
4             f[x][y] = min(f[x][y], f[x][k] + f[k][y])
```

综上时间复杂度是 $O(N^3)$ ，空间复杂度是 $O(N^2)$ 。

应用



给一个正权无向图，找一个最小权值和的环。



首先这一定是一个简单环。

想一想这个环是怎么构成的。

考虑环上编号最大的结点 u 。

$f[u-1][x][y]$ 和 $(u, x), (u, y)$ 共同构成了环。

在 Floyd 的过程中枚举 u ，计算这个和的最小值即可。

时间复杂度为 $O(n^3)$ 。

更多参见 [最小环](#) 部分内容。

已知一个有向图中任意两点之间是否有连边，要求判断任意两点是否连通。



该问题即是求 **图的传递闭包**。

我们只需要按照 Floyd 的过程，逐个加入点判断一下。

只是此时的边的边权变为 $1/0$ ，而取 \min 变成了 **或** 运算。

再进一步用 `bitset` 优化，复杂度可以到 $O(\frac{n^3}{w})$ 。

```
1 // std::bitset<SIZE> f[SIZE];
2 for (k = 1; k <= n; k++)
3     for (i = 1; i <= n; i++)
4         if (f[i][k]) f[i] = f[i] | f[k];
```

Bellman–Ford 算法

Bellman–Ford 算法是一种基于松弛（relax）操作的最短路算法，可以求出有负权的图的最短路，并可以对最短路不存在的情况进行判断。

在国内 OI 界，你可能听说过的「SPFA」，就是 Bellman–Ford 算法的一种实现。

过程

先介绍 Bellman–Ford 算法要用到的松弛操作（Dijkstra 算法也会用到松弛操作）。

对于边 (u, v) ，松弛操作对应下面的式子： $dis(v) = \min(dis(v), dis(u) + w(u, v))$ 。

这么做的含义是显然的：我们尝试用 $S \rightarrow u \rightarrow v$ （其中 $S \rightarrow u$ 的路径取最短路）这条路径去更新 v 点最短路的长度，如果这条路径更优，就进行更新。

Bellman–Ford 算法所做的，就是不断尝试对图上每一条边进行松弛。我们每进行一轮循环，就对图上所有的边都尝试进行一次松弛操作，当一次循环中没有成功的松弛操作时，算法停止。

每次循环是 $O(m)$ 的，那么最多会循环多少次呢？

在最短路存在的情况下，由于一次松弛操作会使最短路的边数至少 $+1$ ，而最短路的边数最多为 $n - 1$ ，因此整个算法最多执行 $n - 1$ 轮松弛操作。故总时间复杂度为 $O(nm)$ 。

但还有一种情况，如果从 S 点出发，抵达一个负环时，松弛操作会无休止地进行下去。注意到前面的论证中已经说明了，对于最短路存在的图，松弛操作最多只会执行 $n - 1$ 轮，因此如果第 n 轮循环时仍然存在能松弛的边，说明从 S 点出发，能够抵达一个负环。

⚠ 负环判断中存在的常见误区

▼

需要注意的是，以 S 点为源点跑 Bellman-Ford 算法时，如果没有给出存在负环的结果，只能说明从 S 点出发不能抵达一个负环，而不能说明图上不存在负环。

因此如果需要判断整个图上是否存在负环，最严谨的做法是建立一个超级源点，向图上每个节点连一条权值为 0 的边，然后以超级源点为起点执行 Bellman-Ford 算法。

实现



参考实现



C++

```
1 struct Edge {
2     int u, v, w;
3 };
4
5 vector<Edge> edge;
6
7 int dis[MAXN], u, v, w;
8 constexpr int INF = 0x3f3f3f3f;
9
10 bool bellmanford(int n, int s) {
11     memset(dis, 0x3f, (n + 1) * sizeof(int));
12     dis[s] = 0;
13     bool flag = false; // 判断一轮循环过程中是否发生松弛操作
14     for (int i = 1; i <= n; i++) {
15         flag = false;
16         for (int j = 0; j < edge.size(); j++) {
17             u = edge[j].u, v = edge[j].v, w = edge[j].w;
18             if (dis[u] == INF) continue;
19             // 无穷大与常数加减仍然为无穷大
20             // 因此最短路长度为 INF 的点引出的边不可能发生松弛操作
21             if (dis[v] > dis[u] + w) {
22                 dis[v] = dis[u] + w;
23                 flag = true;
24             }
25         }
26         // 没有可以松弛的边时就停止算法
27         if (!flag) {
28             break;
29         }
30     }
31     // 第 n 轮循环仍然可以松弛时说明 s 点可以抵达一个负环
32     return flag;
33 }
```

Python

```
1 class Edge:
2     def __init__(self, u=0, v=0, w=0):
3         self.u = u
4         self.v = v
5         self.w = w
6
7     INF = 0x3F3F3F3F
8     edge = []
9
10
11
```

```
12 def bellmanford(n, s):
13     dis = [INF] * (n + 1)
14     dis[s] = 0
15     for i in range(1, n + 1):
16         flag = False
17         for e in edge:
18             u, v, w = e.u, e.v, e.w
19             if dis[u] == INF:
20                 continue
21             # 无穷大与常数加减仍然为无穷大
22             # 因此最短路长度为 INF 的点引出的边不可能发生松弛操作
23             if dis[v] > dis[u] + w:
24                 dis[v] = dis[u] + w
25                 flag = True
26             # 没有可以松弛的边时就停止算法
27             if not flag:
28                 break
29     # 第 n 轮循环仍然可以松弛时说明 s 点可以抵达一个负环
30     return flag
```

队列优化：SPFA

即 Shortest Path Faster Algorithm。

很多时候我们并不需要那么多无用的松弛操作。

很显然，只有上一次被松弛的结点，所连接的边，才有可能引起下一次的松弛操作。

那么我们用队列来维护「哪些结点可能会引起松弛操作」，就能只访问必要的边了。

SPFA 也可以用于判断 s 点是否能抵达一个负环，只需记录最短路经过了多少条边，当经过了至少 n 条边时，说明 s 点可以抵达一个负环。

实现

C++

```
1 struct edge {
2     int v, w;
3 };
4
5 vector<edge> e[MAXN];
6 int dis[MAXN], cnt[MAXN], vis[MAXN];
7 queue<int> q;
8
9 bool spfa(int n, int s) {
10     memset(dis, 0x3f, (n + 1) * sizeof(int));
11     dis[s] = 0, vis[s] = 1;
12     q.push(s);
13     while (!q.empty()) {
14         int u = q.front();
15         q.pop(), vis[u] = 0;
16         for (auto ed : e[u]) {
17             int v = ed.v, w = ed.w;
18             if (dis[v] > dis[u] + w) {
19                 dis[v] = dis[u] + w;
20                 cnt[v] = cnt[u] + 1; // 记录最短路经过的边数
21                 if (cnt[v] >= n) return false;
22                 // 在不经过负环的情况下，最短路至多经过 n - 1 条边
23                 // 因此如果经过了多于 n 条边，一定说明经过了负环
24                 if (!vis[v]) q.push(v), vis[v] = 1;
25             }
26         }
27     }
28     return true;
29 }
```

Python

```
1 from collections import deque
2
3
4 class Edge:
5     def __init__(self, v=0, w=0):
6         self.v = v
7         self.w = w
8
9
10 e = [[Edge() for i in range(MAXN)] for j in range(MAXN)]
11 INF = 0x3F3F3F3F
12
13
14 def spfa(n, s):
15     dis = [INF] * (n + 1)
```

```

16     cnt = [0] * (n + 1)
17     vis = [False] * (n + 1)
18     q = deque()
19
20     dis[s] = 0
21     vis[s] = True
22     q.append(s)
23     while q:
24         u = q.popleft()
25         vis[u] = False
26         for ed in e[u]:
27             v, w = ed.v, ed.w
28             if dis[v] > dis[u] + w:
29                 dis[v] = dis[u] + w
30                 cnt[v] = cnt[u] + 1 # 记录最短路经过的边数
31             if cnt[v] >= n:
32                 return False
33             # 在不经过负环的情况下, 最短路至多经过 n - 1 条边
34             # 因此如果经过了多于 n 条边, 一定说明经过了负环
35             if not vis[v]:
36                 q.append(v)
37                 vis[v] = True

```

虽然在大多数情况下 SPFA 跑得很快，但其最坏情况下的时间复杂度为 $O(nm)$ ，将其卡到这个复杂度也是不难的，所以考试时要谨慎使用（在没有负权边时最好使用 Dijkstra 算法，在有负权边且题目中的图没有特殊性质时，若 SPFA 是标算的一部分，题目不应当给出 Bellman–Ford 算法无法通过的数据范围）。

Bellman–Ford 的其他优化

除了队列优化（SPFA）之外，Bellman–Ford 还有其他形式的优化，这些优化在部分图上效果明显，但在某些特殊图上，最坏复杂度可能达到指数级。

- 堆优化：将队列换成堆，与 Dijkstra 的区别是允许一个点多次入队。在有负权边的图可能被卡成指数级复杂度。
- 栈优化：将队列换成栈（即将原来的 BFS 过程变成 DFS），在寻找负环时可能具有更高效率，但最坏时间复杂度仍然为指数级。
- LLL 优化：将普通队列换成双端队列，每次将入队结点距离和队内距离平均值比较，如果更大则插入至队尾，否则插入队首。
- SLF 优化：将普通队列换成双端队列，每次将入队结点距离和队首比较，如果更大则插入至队尾，否则插入队首。
- D'Esopo–Pape 算法：将普通队列换成双端队列，如果一个节点之前没有入队，则将其插入队尾，否则插入队首。

更多优化以及针对这些优化的 Hack 方法，可以看 [fstqwj 在知乎上的回答](#)。

Dijkstra 算法

Dijkstra (/dɪkstra/或/'dɛɪkstra/) 算法由荷兰计算机科学家 E. W. Dijkstra 于 1956 年发现，1959 年公开发表。是一种求解 **非负权图** 上单源最短路径的算法。

过程

将结点分成两个集合：已确定最短路长度的点集（记为 S 集合）的和未确定最短路长度的点集（记为 T 集合）。一开始所有的点都属于 T 集合。

初始化 $dis(s) = 0$ ，其他点的 dis 均为 $+\infty$ 。

然后重复这些操作：

1. 从 T 集合中，选取一个最短路长度最小的结点，移到 S 集合中。
2. 对那些刚刚被加入 S 集合的结点的所有出边执行松弛操作。

直到 T 集合为空，算法结束。

时间复杂度

朴素的实现方法为每次 2 操作执行完毕后，直接在 T 集合中暴力寻找最短路长度最小的结点。2 操作总时间复杂度为 $O(m)$ ，1 操作总时间复杂度为 $O(n^2)$ ，全过程的时间复杂度为 $O(n^2 + m) = O(n^2)$ 。

可以用堆来优化这一过程：每成功松弛一条边 (u, v) ，就将 v 插入堆中（如果 v 已经在堆中，直接执行 Decrease-key），1 操作直接取堆顶结点即可。共计 $O(m)$ 次 Decrease-key， $O(n)$ 次 pop，选择不同堆可以取到不同的复杂度，参考 [堆](#) 页面。堆优化能做到的最优复杂度为 $O(n \log n + m)$ ，能做到这一复杂度的有斐波那契堆等。

特别地，可以使用优先队列维护，此时无法执行 Decrease-key 操作，但可以通过每次松弛时重新插入该结点，且弹出时检查该结点是否已被松弛过，若是则跳过，复杂度 $O(m \log n)$ ，优点是实现较简单。

这里的堆也可以用线段树来实现，复杂度为 $O(m \log n)$ ，在一些特殊的非递归线段树实现下，该做法常数比堆更小。并且线段树支持的操作更多，在一些特殊图问题上只能用线段树来维护。

在稀疏图中， $m = O(n)$ ，堆优化的 Dijkstra 算法具有较大的效率优势；而在稠密图中， $m = O(n^2)$ ，这时候使用朴素实现更优。

正确性证明

下面用数学归纳法证明，在 **所有边权值非负** 的前提下，Dijkstra 算法的正确性¹。

简单来说，我们要证明的，就是在执行 1 操作时，取出的结点 u 最短路均已经被确定，即满足 $D(u) = dis(u)$ 。

初始时 $S = \emptyset$ ，假设成立。

接下来用反证法。

设 u 点为算法中第一个在加入 S 集合时不满足 $D(u) = dis(u)$ 的点。因为 s 点一定满足 $D(u) = dis(u) = 0$ ，且它一定是第一个加入 S 集合的点，因此将 u 加入 S 集合前， $S \neq \emptyset$ ，如果不存在 s 到 u 的路径，则 $D(u) = dis(u) = +\infty$ ，与假设矛盾。

于是一定存在路径 $s \rightarrow x \rightarrow y \rightarrow u$ ，其中 y 为 $s \rightarrow u$ 路径上第一个属于 T 集合的点，而 x 为 y 的前驱结点（显然 $x \in S$ ）。需要注意的是，可能存在 $s = x$ 或 $y = u$ 的情况，即 $s \rightarrow x$ 或 $y \rightarrow u$ 可能是空路径。

因为在 u 结点之前加入的结点都满足 $D(u) = dis(u)$ ，所以在 x 点加入到 S 集合时，有 $D(x) = dis(x)$ ，此时边 (x, y) 会被松弛，从而可以证明，将 u 加入到 S 时，一定有 $D(y) = dis(y)$ 。

下面证明 $D(u) = dis(u)$ 成立。在路径 $s \rightarrow x \rightarrow y \rightarrow u$ 中，因为图上所有边权非负，因此 $D(y) \leq D(u)$ 。从而 $dis(y) = D(y) \leq D(u) \leq dis(u)$ 。但是因为 u 结点在 1 过程中被取出 T 集合时， y 结点还没有被取出 T 集合，因此此时有 $dis(u) \leq dis(y)$ ，从而得到 $dis(y) = D(y) = D(u) = dis(u)$ ，这与 $D(u) \neq dis(u)$ 的假设矛盾，故假设不成立。

因此我们证明了，1 操作每次取出的点，其最短路均已经被确定。命题得证。

注意到证明过程中的关键不等式 $D(y) \leq D(u)$ 是在图上所有边权非负的情况下得出的。当图上存在负权边时，这一不等式不再成立，Dijkstra 算法的正确性将无法得到保证，算法可能会给出错误的结果。

实现

这里同时给出 $O(n^2)$ 的暴力做法实现和 $O(m \log m)$ 的优先队列做法实现。

朴素实现

C++

```
1 struct edge {
2     int v, w;
3 };
4
5 vector<edge> e[MAXN];
6 int dis[MAXN], vis[MAXN];
7
8 void dijkstra(int n, int s) {
9     memset(dis, 0x3f, (n + 1) * sizeof(int));
10    dis[s] = 0;
11    for (int i = 1; i <= n; i++) {
12        int u = 0, mind = 0x3f3f3f3f;
13        for (int j = 1; j <= n; j++)
14            if (!vis[j] && dis[j] < mind) u = j, mind = dis[j];
15        vis[u] = true;
16        for (auto ed : e[u]) {
17            int v = ed.v, w = ed.w;
18            if (dis[v] > dis[u] + w) dis[v] = dis[u] + w;
19        }
20    }
21 }
```

Python

```
1 class Edge:
2     def __init__(self, v=0, w=0):
3         self.v = v
4         self.w = w
5
6
7 e = [[Edge() for i in range(MAXN)] for j in range(MAXN)]
8 INF = 0x3F3F3F3F
9
10
11 def dijkstra(n, s):
12     dis = [INF] * (n + 1)
13     vis = [0] * (n + 1)
14
15     dis[s] = 0
16     for i in range(1, n + 1):
17         u = 0
18         mind = INF
19         for j in range(1, n + 1):
20             if not vis[j] and dis[j] < mind:
21                 u = j
22                 mind = dis[j]
23             vis[u] = True
```

```
24     for ed in e[u]:  
25         v, w = ed.v, ed.w  
26         if dis[v] > dis[u] + w:  
27             dis[v] = dis[u] + w
```

优先队列实现

C++

```
1 struct edge {
2     int v, w;
3 };
4
5 struct node {
6     int dis, u;
7
8     bool operator>(const node& a) const { return dis > a.dis; }
9 };
10
11 vector<edge> e[MAXN];
12 int dis[MAXN], vis[MAXN];
13 priority_queue<node, vector<node>, greater<node>> q;
14
15 void dijkstra(int n, int s) {
16     memset(dis, 0x3f, (n + 1) * sizeof(int));
17     memset(vis, 0, (n + 1) * sizeof(int));
18     dis[s] = 0;
19     q.push({0, s});
20     while (!q.empty()) {
21         int u = q.top().u;
22         q.pop();
23         if (vis[u]) continue;
24         vis[u] = 1;
25         for (auto ed : e[u]) {
26             int v = ed.v, w = ed.w;
27             if (dis[v] > dis[u] + w) {
28                 dis[v] = dis[u] + w;
29                 q.push({dis[v], v});
30             }
31         }
32     }
33 }
```

Python

```
1 def dijkstra(e, s):
2     """
3         输入:
4         e:邻接表
5         s:起点
6         返回:
7         dis:从s到每个顶点的最短路长度
8     """
9     dis = defaultdict(lambda: float("inf"))
10    dis[s] = 0
11    q = [(0, s)]
```

```

12     vis = set()
13     while q:
14         _, u = heapq.heappop(q)
15         if u in vis:
16             continue
17         vis.add(u)
18         for v, w in e[u]:
19             if dis[v] > dis[u] + w:
20                 dis[v] = dis[u] + w
21                 heapq.heappush(q, (dis[v], v))
22     return dis

```

Johnson 全源最短路径算法

Johnson 和 Floyd 一样，是一种能求出无负环图上任意两点间最短路径的算法。该算法在 1977 年由 Donald B. Johnson 提出。

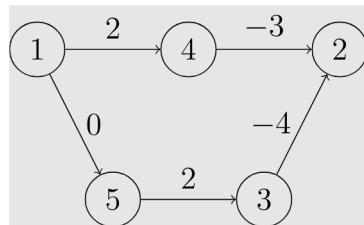
任意两点间的最短路可以通过枚举起点，跑 n 次 Bellman–Ford 算法解决，时间复杂度是 $O(n^2m)$ 的，也可以直接用 Floyd 算法解决，时间复杂度为 $O(n^3)$ 。

注意到堆优化的 Dijkstra 算法求单源最短路径的时间复杂度比 Bellman–Ford 更优，如果枚举起点，跑 n 次 Dijkstra 算法，就可以在 $O(nm \log m)$ （取决于 Dijkstra 算法的实现）的时间复杂度内解决本问题，比上述跑 n 次 Bellman–Ford 算法的时间复杂度更优秀，在稀疏图上也比 Floyd 算法的时间复杂度更加优秀。

但 Dijkstra 算法不能正确求解带负权边的最短路，因此我们需要对原图上的边进行预处理，确保所有边的边权均非负。

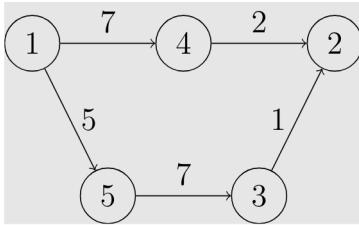
一种容易想到的方法是给所有边的边权同时加上一个正数 x ，从而让所有边的边权均非负。如果新图上起点到终点的最短路经过了 k 条边，则将最短路减去 kx 即可得到实际最短路。

但这样的方法是错误的。考虑下图：



$1 \rightarrow 2$ 的最短路为 $1 \rightarrow 5 \rightarrow 3 \rightarrow 2$ ，长度为 -2 。

但假如我们把每条边的边权加上 5 呢？



新图上 $1 \rightarrow 2$ 的最短路为 $1 \rightarrow 4 \rightarrow 2$, 已经不是实际的最短路了。

Johnson 算法则通过另外一种方法来给每条边重新标注边权。

我们新建一个虚拟节点 (在这里我们就设它的编号为 0)。从这个点向其他所有点连一条边权为 0 的边。

接下来用 Bellman–Ford 算法求出从 0 号点到其他所有点的最短路, 记为 h_i 。

假如存在一条从 u 点到 v 点, 边权为 w 的边, 则我们将该边的边权重新设置为 $w + h_u - h_v$ 。

接下来以每个点为起点, 跑 n 轮 Dijkstra 算法即可求出任意两点间的最短路了。

一开始的 Bellman–Ford 算法并不是时间上的瓶颈, 若使用 `priority_queue` 实现 Dijkstra 算法, 该算法的时间复杂度是 $O(nm \log m)$ 。

正确性证明

为什么这样重新标注边权的方式是正确的呢?

在讨论这个问题之前, 我们先讨论一个物理概念——势能。

诸如重力势能, 电势能这样的势能都有一个特点, 势能的变化量只和起点和终点的相对位置有关, 而与起点到终点所走的路径无关。

势能还有一个特点, 势能的绝对值往往取决于设置的零势能点, 但无论将零势能点设置在哪里, 两点间势能的差值是一定的。

接下来回到正题。

在重新标记后的图上, 从 s 点到 t 点的一条路径 $s \rightarrow p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_k \rightarrow t$ 的长度表达式如下:

$$(w(s, p_1) + h_s - h_{p_1}) + (w(p_1, p_2) + h_{p_1} - h_{p_2}) + \dots + (w(p_k, t) + h_{p_k} - h_t)$$

化简后得到:

$$w(s, p_1) + w(p_1, p_2) + \dots + w(p_k, t) + h_s - h_t$$

无论我们从 s 到 t 走的是哪一条路径, $h_s - h_t$ 的值是不变的, 这正与势能的性质相吻合!

为了方便, 下面我们就把 h_i 称为 i 点的势能。

上面的新图中 $s \rightarrow t$ 的最短路的长度表达式由两部分组成，前面的边权和为原图中 $s \rightarrow t$ 的最短路，后面则是两点间的势能差。因为两点间势能的差为定值，因此原图上 $s \rightarrow t$ 的最短路与新图上 $s \rightarrow t$ 的最短路相对应。

到这里我们的正确性证明已经解决了一半——我们证明了重新标注边权后图上的最短路径仍然是原来的最短路径。接下来我们需要证明新图中所有边的边权非负，因为在非负权图上，Dijkstra 算法能够保证得出正确的结果。

根据三角形不等式，图上任意一边 (u, v) 上两点满足： $h_v \leq h_u + w(u, v)$ 。这条边重新标记后的边权为 $w'(u, v) = w(u, v) + h_u - h_v \geq 0$ 。这样我们证明了新图上的边权均非负。

这样，我们就证明了 Johnson 算法的正确性。

不同方法的比较

最短路算法	Floyd	Bellman-Ford	Dijkstra	Johnson
最短路类型	每对结点之间的最短路	单源最短路	单源最短路	每对结点之间的最短路
作用于	任意图	任意图	非负权图	任意图
能否检测负环？	能	能	不能	能
时间复杂度	$O(N^3)$	$O(NM)$	$O(M \log M)$	$O(NM \log M)$

注：表中的 Dijkstra 算法在计算复杂度时均用 `priority_queue` 实现。

输出方案

开一个 `pre` 数组，在更新距离的时候记录下来后面的点是如何转移过去的，算法结束前再递归地输出路径即可。

比如 Floyd 就要记录 `pre[i][j] = k;`，Bellman-Ford 和 Dijkstra 一般记录 `pre[v] = u`。

一些特殊情形

- 边权只由 0 和 1 组成的图上最短路：[0-1 BFS](#)；
- 允许至多 k 次改变路径成本等操作的最短路问题：[分层图最短路](#)。

参考资料与注释

1. 《算法导论（第3版中译本）》，机械工业出版社，2013年，第384 - 385页。 ↪

⌚ 本页面最近更新：2025/6/5 01:26:43，[更新历史](#)

✍ 发现错误？想一起完善？[在GitHub上编辑此页！](#)

👤 本页面贡献者：[StudyingFather](#), [Ir1d](#), [Tiphereth-A](#), [Enter-tainer](#), [H-J-Granger](#), [Yanjun-Zhao](#), [countercurrent-time](#), [greyqz](#), [NachtgeistW](#), [PeterlitsZo](#), [Anguei](#), [CCXXXI](#), [Early0v0](#), [Haohu Shen](#), [ImpleLee](#), [ksyx](#), [lingkerio](#), [mgt](#), [Steaunk](#), [Xeonacid](#), [AngelKitty](#), [Chrogeek](#), [ChungZH](#), [cjsoft](#), [diauweb](#), [du33169](#), [ezoixx130](#), [GekkaSaori](#), [iamtwz](#), [Konano](#), [LovelyBuggies](#), [Makkiy](#), [minghu6](#), [ouuan](#), [ouuan](#), [P-Y-Y](#), [PotassiumWings](#), [SamZhangQingChuan](#), [sshwy](#), [Suyun514](#), [Taoran-01](#), [weiyong1024](#), [abc1763613206](#), [AljcC](#), [alphagocc](#), [AndrewWayne](#), [ArcticLampyrid](#), [boristown](#), [c-forrest](#), [Eletary](#), [Error-Eric](#), [FinBird](#), [GavinZhengOI](#), [Gesrua](#), [hensier](#), [isdanni](#), [Kaiser-Yang](#), [kxccc](#), [LiserverYang](#), [lychees](#), [mcendu](#), [Menci](#), [miaotony](#), [MingqiHuang](#), [Nanarikom](#), [Peanut-Tang](#), [r-value](#), [Redstix](#), [renbaoshuo](#), [Reqwey](#), [shawlleyw](#), [SukkaW](#), [TrisolarisHD](#), [wplf](#), [zzjjbb](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





二分

本页面将简要介绍二分查找，由二分法衍生的三分法以及二分答案。

二分法

定义

二分查找（英语：binary search），也称折半搜索（英语：half-interval search）、对数搜索（英语：logarithmic search），是用来在一个有序数组中查找某一元素的算法。

过程

以在一个升序数组中查找一个数为例。

它每次考察数组当前部分的中间元素，如果中间元素刚好是要找的，就结束搜索过程；如果中间元素小于所查找的值，那么左侧的只会更小，不会有所查找的元素，只需到右侧查找；如果中间元素大于所查找的值同理，只需到左侧查找。

性质

时间复杂度

二分查找的最优时间复杂度为 $O(1)$ 。

二分查找的平均时间复杂度和最坏时间复杂度均为 $O(\log n)$ 。因为在二分搜索过程中，算法每次都把查询的区间减半，所以对于一个长度为 n 的数组，至多会进行 $O(\log n)$ 次查找。

空间复杂度

迭代版本的二分查找的空间复杂度为 $O(1)$ 。

递归（无尾调用消除）版本的二分查找的空间复杂度为 $O(\log n)$ 。

实现

```
1 int binary_search(int start, int end, int key) {
2     int ret = -1; // 未搜索到数据返回-1下标
3     int mid;
4     while (start <= end) {
5         mid = start + ((end - start) >> 1); // 直接平均可能会溢出，所以用这个算法
6         if (arr[mid] < key)
7             start = mid + 1;
```

```
8     else if (arr[mid] > key)
9         end = mid - 1;
10    else { // 最后检测相等是因为多数搜索情况不是大于就是小于
11        ret = mid;
12        break;
13    }
14}
15return ret; // 单一出口
16}
```

Note

参考 [编译优化 #位运算代替乘法](#)，对于 n 是有符号数的情况，当你可以保证 $n \geq 0$ 时，`n >> 1` 比 `n / 2` 指令数更少。

最大值最小化

注意，这里的有序是广义的有序，如果一个数组中的左侧或者右侧都满足某一种条件，而另一侧都不满足这种条件，也可以看作是一种有序（如果把满足条件看做 1，不满足看做 0，至少对于这个条件的这一维度是有序的）。换言之，二分搜索法可以用来查找满足某种条件的最大（最小）的值。

要求满足某种条件的最大值的最小可能情况（最大值最小化），首先的想法是从小到大枚举这个作为答案的「最大值」，然后去判断是否合法。若答案单调，就可以使用二分搜索法来更快地找到答案。因此，要想使用二分搜索法来解这种「最大值最小化」的题目，需要满足以下三个条件：

1. 答案在一个固定区间内；
2. 可能查找一个符合条件的值不是很容易，但是要求能比较容易地判断某个值是否是符合条件的；
3. 可行解对于区间满足一定的单调性。换言之，如果 x 是符合条件的，那么有 $x + 1$ 或者 $x - 1$ 也符合条件。（这样下来就满足了上面提到的单调性）

当然，最小值最大化是同理的。

STL 的二分查找

C++ 标准库中实现了查找首个不小于给定值的元素的函数 `std::lower_bound` 和查找首个大于给定值的元素的函数 `std::upper_bound`，二者均定义于头文件 `<algorithm>` 中。

二者均采用二分实现，所以调用前必须保证元素有序。

bsearch

bsearch 函数为 C 标准库实现的二分查找，定义在 `<stdlib.h>` 中。在 C++ 标准库里，该函数定义在 `<cstdlib>` 中。qsort 和 bsearch 是 C 语言中唯二的两个算法类函数。

bsearch 函数相比 qsort（[排序相关 STL](#)）的四个参数，在最左边增加了参数「待查元素的地址」。之所以按照地址的形式传入，是为了方便直接套用与 qsort 相同的比较函数，从而实现排序后的立即查找。因此这个参数不能直接传入具体值，而是要先将待查值用一个变量存储，再传入该变量地址。

于是 bsearch 函数总共有五个参数：待查元素的地址、数组名、元素个数、元素大小、比较规则。比较规则仍然通过指定比较函数实现，详见 [排序相关 STL](#)。

bsearch 函数的返回值是查找到的元素的地址，该地址为 `void` 类型。

注意：bsearch 与上文的 `lower_bound` 和 `upper_bound` 有两点不同：

- 当符合条件的元素有重复多个的时候，会返回执行二分查找时第一个符合条件的元素，从而这个元素可能位于重复多个元素的中间部分。
- 当查找不到相应的元素时，会返回 `NULL`。

用 `lower_bound` 可以实现与 `bsearch` 完全相同的功能，所以可以使用 `bsearch` 通过的题目，直接改写成 `lower_bound` 同样可以实现。但是鉴于上述不同之处的第二点，例如，在序列 1、2、4、5、6 中查找 3，`bsearch` 实现 `lower_bound` 的功能会变得困难。

利用 `bsearch` 实现 `lower_bound` 的功能比较困难，是否一定就不能实现？答案是否定的，存在比较 `tricky` 的技巧。借助编译器处理比较函数的特性：总是将第一个参数指向待查元素，将第二个参数指向待查数组中的元素，也可以用 `bsearch` 实现 `lower_bound` 和 `upper_bound`，如下文示例。只是，这要求待查数组必须是全局数组，从而可以直接传入首地址。

```
1 int A[100005]; // 示例全局数组
2
3 // 查找首个不小于待查元素的元素的地址
4 int lower(const void *p1, const void *p2) {
5     int *a = (int *)p1;
6     int *b = (int *)p2;
7     if ((b == A || compare(a, b - 1) > 0) && compare(a, b) > 0)
8         return 1;
9     else if (b != A && compare(a, b - 1) <= 0)
10        return -1; // 用到地址的减法，因此必须指定元素类型
11    else
12        return 0;
13 }
14
15 // 查找首个大于待查元素的元素的地址
16 int upper(const void *p1, const void *p2) {
17     int *a = (int *)p1;
18     int *b = (int *)p2;
19     if ((b == A || compare(a, b - 1) >= 0) && compare(a, b) >= 0)
20         return 1;
21     else if (b != A && compare(a, b - 1) < 0)
22        return -1; // 用到地址的减法，因此必须指定元素类型
23     else
```

```
24     return 0;  
25 }
```

因为现在的 OI 选手很少写纯 C，并且此方法作用有限，所以不是重点。对于新手而言，建议老老实实地使用 C++ 中的 `lower_bound` 和 `upper_bound` 函数。

二分答案

解题的时候往往会考虑枚举答案然后检验枚举的值是否正确。若满足单调性，则满足使用二分法的条件。把这里的枚举换成二分，就变成了「二分答案」。

Luogu P1873 砍树

伐木工人米尔科需要砍倒 M 米长的木材。这是一个对米尔科来说很容易的工作，因为他有一个漂亮的新伐木机，可以像野火一样砍倒森林。不过，米尔科只被允许砍倒单行树木。

米尔科的伐木机工作过程如下：米尔科设置一个高度参数 H （米），伐木机升起一个巨大的锯片到高度 H ，并锯掉所有的树比 H 高的部分（当然，树木不高于 H 米的部分保持不变）。米尔科就得得到树木被锯下的部分。

例如，如果一行树的高度分别为 20, 15, 10, 17，米尔科把锯片升到 15 米的高度，切割后树木剩下的高度将是 15, 15, 10, 15，而米尔科将从第 1 棵树得到 5 米木材，从第 4 棵树得到 2 米木材，共 7 米木材。

米尔科非常关注生态保护，所以他不会砍掉过多的木材。这正是他尽可能高地设定伐木机锯片的原因。你的任务是帮助米尔科找到伐木机锯片的最大的整数高度 H ，使得他能得到木材至少为 M 米。即，如果再升高 1 米锯片，则他将得不到 M 米木材。

解题思路

我们可以在 1 到 10^9 中枚举答案，但是这种朴素写法肯定拿不到满分，因为从 1 枚举到 10^9 太耗时间。我们可以在 $[1, 10^9]$ 的区间上进行二分作为答案，然后检查各个答案的可行性（一般使用贪心法）。**这就是二分答案。**

参考代码

```
1 int a[1000005];
2 int n, m;
3
4 bool check(int k) { // 检查可行性, k 为锯片高度
5     long long sum = 0;
6     for (int i = 1; i <= n; i++) // 检查每一棵树
7         if (a[i] > k) // 如果树高于锯片高度
8             sum += (long long)(a[i] - k); // 累加树木长度
9     return sum >= m; // 如果满足最少长度代表可行
10 }
11
12 int find() {
13     int l = 1, r = 1e9 + 1; // 因为是左闭右开的, 所以 10^9 要加 1
14     while (l + 1 < r) { // 如果两点不相邻
15         int mid = (l + r) / 2; // 取中间值
16         if (check(mid)) // 如果可行
17             l = mid; // 升高锯片高度
18         else
19             r = mid; // 否则降低锯片高度
20     }
21     return l; // 返回左边值
22 }
23
24 int main() {
25     cin >> n >> m;
26     for (int i = 1; i <= n; i++) cin >> a[i];
27     cout << find();
28     return 0;
29 }
```

看完了上面的代码，你肯定会有两个疑问：

1. 为何搜索区间是左闭右开的？

因为搜到最后，会这样（以合法的最大值为例）：

合法			不合法			
最小值	L	MID	R	最大值

或者

合法			不合法			
最小值	L	MID	R	最大值

然后会

合法			不合法			
最小值	L,MID	R	最大值

或者

合法				不合法		
最小值	L	MID,R	最大值

合法的最小值恰恰相反。

2. 为何返回左边值？

同上。

三分法

引入

二分法可以用于近似求出函数的零点。如果需要求出单峰函数的极值点，通常需要使用三分法 (ternary search)。

对于一个函数 $f(x)$ ，如果存在 x^* 使得 $f(x)$ 在 $x < x^*$ 时单调递增且 $f(x)$ 在 $x > x^*$ 时单调递减，就称 $f(x)$ 为单峰函数 (unimodal function)。显然， x^* 就是它的最大值点，而 $f(x^*)$ 则是它的最大值。

为什么不能通过求导函数的零点来求极值点？

客观上，求出导数后，通过二分法求出导数的零点（由于函数是单峰函数，其导数在同一范围内的零点是唯一的）得到单峰函数的极值点是可行的。

但首先，对于一些函数，求导的过程和结果比较复杂。

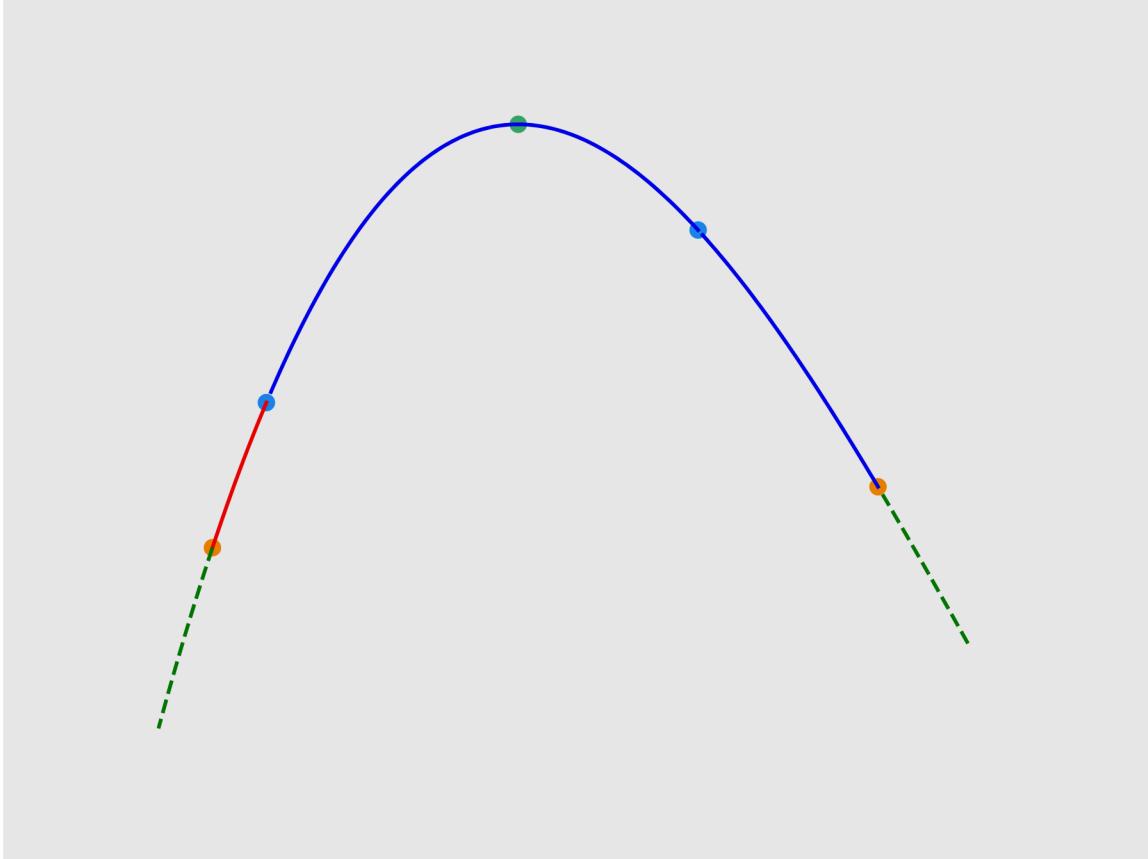
其次，某些题中需要求极值点的单峰函数并非一个单独的函数，而是多个函数进行特殊运算得到的函数（如求多个单调性不完全相同的一次函数的最小值的最大值）。此时函数的导函数可能是分段函数，且在函数某些点上可能不可导。

注意

三分法既可以求出单峰函数的最大值，也可以求出「单谷函数」的最小值。为行文方便，除特殊说明外，下文中均以求单峰函数的最大值为例。

过程

三分法与二分法的基本思想类似，但每次操作需在当前区间 $[l, r]$ （下图中两个橙点之间）内任取两点 $lmid < rmid$ （下图中的两个蓝点）。如下图所示，如果 $f(lmid) < f(rmid)$ ，则在 $[l, lmid]$ （下图中的红色部分）中函数必然单调递增，最大值点（下图中的绿点）必然不在这一区间内，可舍去这一区间；但是，无法排除最大值点在 $rmid$ 右侧的可能性，所以无法舍去更多区间。之亦然。



三分法的正确性并不依赖于 $lmid$ 和 $rmid$ 的选择，通常可以取两个三等分点。但是，它们的选择确实会影响三分法的效率。这是因为三分法的每次操作都会舍去两侧区间中的其中一个。为减少三分法的操作次数，应使两侧区间尽可能大。因此，每一次操作时的 $lmid$ 和 $rmid$ 分别取 $mid - \varepsilon$ 和 $mid + \varepsilon$ 是一个不错的选择。

实现

伪代码如下：

```

Algorithm TernarySearch( $f, l, r$ ) :
Input. A unimodal function  $f(x)$  and its domain  $[l, r]$ .
Output. The maximizer  $x^*$ , up to an error of  $\varepsilon$ , and its value  $f(x^*)$ .
Method.
1   while  $r - l > \varepsilon$ 
2      $mid \leftarrow (l + r)/2$ 
3      $lmid \leftarrow mid - \varepsilon$ 
4      $rmid \leftarrow mid + \varepsilon$ 
5     if  $f(lmid) < f(rmid)$ 
6        $l \leftarrow lmid$ 
7     else
8        $r \leftarrow rmid$ 
9    $x^* \leftarrow (l + r)/2$ 
10  return  $x^*, f(x^*)$ 

```

整数的情形

如果函数 $f(x)$ 的定义域是整数，那么上述三分法和后文的黄金分割法都应该在 $r - l$ 很小时就终止。对于 $r - l$ 很小的情形，需要通过暴力遍历的方法求得最大值点。

优化：黄金分割法

如果单次调用 $f(x)$ 的成本很高，需要进一步减少 $f(x)$ 的调用次数，可以通过黄金分割法 (golden-section search) 进一步改进三分法的常数。这也是华罗庚提出的优选法的重要内容。

三分法中，每轮迭代需要两次函数调用，且单轮迭代后区间长度至多缩短到原来的 $1/2$ 。这意味着，要达到精度 ε ，至少需要

$$2 \log_2 \frac{r - l}{\varepsilon}$$

次函数调用。这是三分法能够取得的最好的结果。如果选取其他分点，例如三等分点，那么调用次数会进一步增加，因为单轮迭代后区间缩短得更慢。

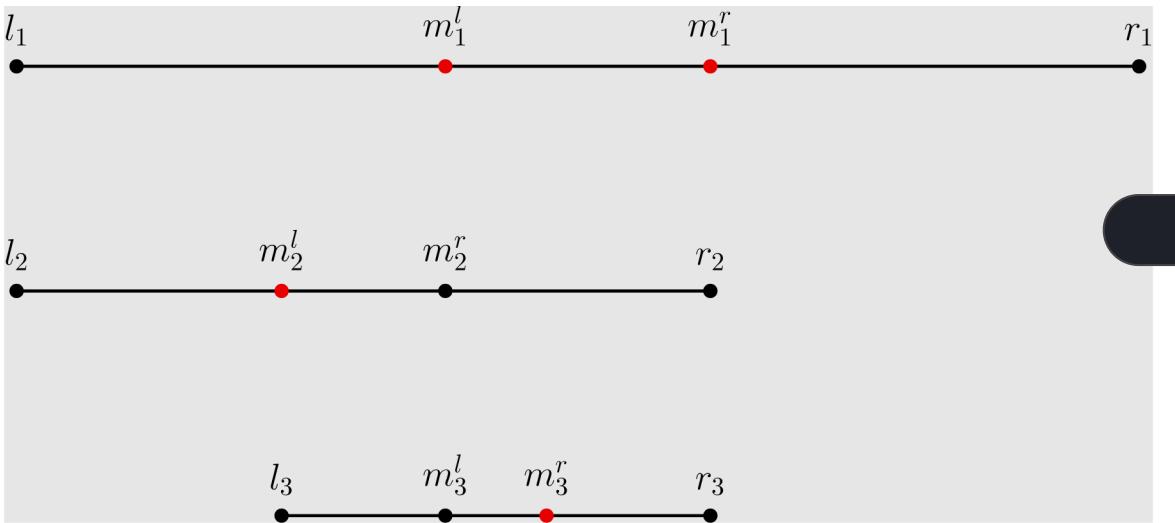
黄金分割法的改进思路是，复用前文已经计算过的分点。这样，除了第一轮迭代需要两次函数调用外，其余轮次的迭代只需要一次函数调用。设黄金分割比为

$$\phi = \frac{\sqrt{5} - 1}{2} \approx 0.618.$$

每轮迭代时，选取的分点是左右两个黄金分割点：

$$m^l = \phi l + (1 - \phi)r, m^r = (1 - \phi)l + \phi r.$$

黄金分割点分割线段具有自相似结构。也就是说， m^l 是线段 $[l, r]$ 的左黄金分割点，也是线段 $[l, m^r]$ 的右黄金分割点。这样选取分点的好处是，第 $k > 1$ 轮迭代选取的分点中，一定有一个分点是之前已经计算过的，可以直接复用之前的计算结果。



这样选取分点后，要达到精度 ε ，只需要

$$1 + \log_{\phi^{-1}} \frac{r-l}{\varepsilon} \approx 1 + 1.44 \log_2 \frac{r-l}{\varepsilon}$$

次函数调用。渐进意义上，函数的调用次数更少。

伪代码如下：

```

Algorithm GoldenSectionSearch( $f, l, r$ ) :
Input. A unimodal function  $f(x)$  and its domain  $[l, r]$ .
Output. The maximizer  $x^*$ , up to an error of  $\varepsilon$ , and its value  $f(x^*)$ .
Method.
1    $lmid \leftarrow \phi l + (1 - \phi)r$ 
2    $rmid \leftarrow (1 - \phi)l + \phi r$ 
3    $lval \leftarrow f(lmid)$ 
4    $rval \leftarrow f(rmid)$ 
5   while  $r - l > \varepsilon$ 
6       if  $lval > rval$ 
7            $r \leftarrow rmid$ 
8            $rmid \leftarrow lmid$ 
9            $rval \leftarrow lval$ 
10           $lmid \leftarrow \phi l + (1 - \phi)r$ 
11           $lval \leftarrow f(lmid)$ 
12      else
13           $l \leftarrow lmid$ 
14           $lmid \leftarrow rmid$ 
15           $lval \leftarrow rval$ 
16           $rmid \leftarrow (1 - \phi)l + \phi r$ 
17           $rval \leftarrow f(rmid)$ 
18   $x^* \leftarrow (l + r)/2$ 
19  return  $x^*, f(x^*)$ 
```

例题

洛谷 P3382 - 【模板】三分法

给定一个 N 次函数和范围 $[l, r]$, 求出使函数在 $[l, x]$ 上单调递增且在 $[x, r]$ 上单调递减的唯一的 x 的值。

解题思路

本题要求求 N 次函数在 $[l, r]$ 取最大值时自变量的值, 显然可以使用三分法。



参考代码

C++

```
1 #include <cmath>
2 #include <iomanip>
3 #include <iostream>
4 using namespace std;
5
6 constexpr double eps = 1e-7;
7 int N;
8 double l, r, A[20], mid, lmid, rmid;
9
10 double f(double x) {
11     double res = (double)0;
12     for (int i = N; i >= 0; i--) res += A[i] * pow(x, i);
13     return res;
14 }
15
16 int main() {
17     cin.tie(nullptr)->sync_with_stdio(false);
18     cin >> N >> l >> r;
19     for (int i = N; i >= 0; i--) cin >> A[i];
20     while (r - l > eps) {
21         mid = (l + r) / 2;
22         lmid = mid - eps;
23         rmid = mid + eps;
24         if (f(lmid) > f(rmid))
25             r = mid;
26         else
27             l = mid;
28     }
29     cout << fixed << setprecision(6) << l;
30     return 0;
31 }
```

Python

```
1 eps = 1e-6
2 n, l, r = map(float, input().split())
3 a = tuple(map(float, input().split()))[::-1]
4
5
6 def f(x):
7     return sum(x**i * j for i, j in enumerate(a))
8
9
10 while r - l > eps:
11     mid = (l + r) / 2
12     if f(mid - eps) > f(mid + eps):
13         r = mid
```

```
14     else:
15         l = mid
16     print(l)
```

习题

- [UVa 1476 - Error Curves](#)
- [UVa 10385 - Duathlon](#)
- [UOJ 162 - 【清华集训 2015】灯泡测试](#)
- [洛谷 P7579 - 「RdOI R2」称重 \(weigh\)](#)

分数规划

参见：[分数规划](#)

分数规划通常描述为下列问题：每个物品有两个属性 c_i, d_i ，要求通过某种方式选出若干个，使得 $\frac{\sum c_i}{\sum d_i}$ 最大或最小。

经典的例子有最优化率环、最优化率生成树等等。

分数规划可以用二分法来解决。

参考资料

- [Ternary search - Wikipedia](#)
- [Golden-section search - Wikipedia](#)
- [Ternary search - CP Algotihms](#)

🔑 本页面最近更新：2025/8/24 15:26:32，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[Ir1d](#), [H-J-Granger](#), [StudyingFather](#), [NachtgeistW](#), [sshwy](#), [yusancky](#), [countercurrent-time](#), [Enter-tainer](#), [AngelKitty](#), [CBW2007](#), [CCXXXI](#), [cjsoft](#), [diauweb](#), [Early0v0](#), [ezoixx130](#), [GekkaSaori](#), [Henry-ZHR](#), [Konano](#), [ksyx](#), [LovelyBuggies](#), [Makkiy](#), [mgt](#), [minghu6](#), [P-Y-Y](#), [PotassiumWings](#), [SamZhangQingChuan](#), [Suyun514](#), [weiyong1024](#), [Xeonacid](#), [billchenchina](#), [c-forrest](#), [ChungZH](#), [FinParker](#), [flylai](#), [gavinliu266](#), [GavinZhengOI](#), [Gesrua](#), [Great-designer](#), [HanwGeek](#), [HeRaNO](#), [i-yyi](#), [iamtwz](#), [inclc](#), [kxccc](#), [LeiJinpeng](#), [leoleoasd](#), [lychees](#), [Marcythm](#), [Peanut-Tang](#), [Selflocking](#), [shawlleyw](#), [shuzhouliu](#), [SukkaW](#), [Tiphereth-A](#), [TNO-C137](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



递归 & 分治

本页面将介绍递归与分治算法的区别与结合运用。

递归

定义

递归（英语：Recursion），在数学和计算机科学中是指在函数的定义中使用函数自身的方法，在计算机科学中还额外指一种通过重复将问题分解为同类的子问题而解决问题的方法。

引入

要理解递归，就得先理解什么是递归。

递归的基本思想是某个函数直接或者间接地调用自身，这样原问题的求解就转换为了许多性质相同但是规模更小的子问题。求解时只需要关注如何把原问题划分成符合条件的子问题，而不需要过分关注这个子问题是如何被解决的。

以下是一些有助于理解递归的例子：

1. [什么是递归？](#)
2. 如何给一堆数字排序？答：分成两半，先排左半边再排右半边，最后合并就行了，至于怎么排左边和右边，请重新阅读这句话。
3. 你今年几岁？答：去年的岁数加一岁，1999年我出生。



递归在数学中非常常见。例如，集合论对自然数的正式定义是：1是一个自然数，每个自然数都有一个后继，这一个后继也是自然数。

递归代码最重要的两个特征：结束条件和自我调用。自我调用是在解决子问题，而结束条件定义了最简子问题的答案。

```
1 int func(传入数值) {  
2     if (终止条件) return 最小子问题解;
```

```
3     return func(缩小规模);  
4 }
```

为什么要写递归

1. 结构清晰，可读性强。例如，分别用不同的方法实现 归并排序：

C++

```
1 // 不使用递归的归并排序算法  
2 template <typename T>  
3 void merge_sort(vector<T> a) {  
4     int n = a.size();  
5     for (int seg = 1; seg < n; seg = seg + seg)  
6         for (int start = 0; start < n - seg; start += seg + seg)  
7             merge(a, start, start + seg - 1, std::min(start + seg + seg - 1,  
8 n - 1));  
9 }  
10  
11 // 使用递归的归并排序算法  
12 template <typename T>  
13 void merge_sort(vector<T> a, int front, int end) {  
14     if (front >= end) return;  
15     int mid = front + (end - front) / 2;  
16     merge_sort(a, front, mid);  
17     merge_sort(a, mid + 1, end);  
18     merge(a, front, mid, end);  
19 }
```

Python

```
1 # 不使用递归的归并排序算法  
2 def merge_sort(a):  
3     n = len(a)  
4     seg, start = 1, 0  
5     while seg < n:  
6         while start < n - seg:  
7             merge(a, start, start + seg - 1, min(start + seg + seg -  
8 1, n - 1))  
9             start = start + seg + seg  
10            seg = seg + seg  
11  
12  
13 # 使用递归的归并排序算法  
14 def merge_sort(a, front, end):  
15     if front >= end:  
16         return  
17     mid = front + (end - front) / 2  
18     merge_sort(a, front, mid)  
19     merge_sort(a, mid + 1, end)  
20     merge(a, front, mid, end)
```

显然，递归版本比非递归版本更易理解。递归版本的做法一目了然：把左半边排序，把右半边排序，最后合并两边。而非递归版本看起来不知所云，充斥着各种难以理解的边界计算细节，特别容易出 bug，且难以调试。

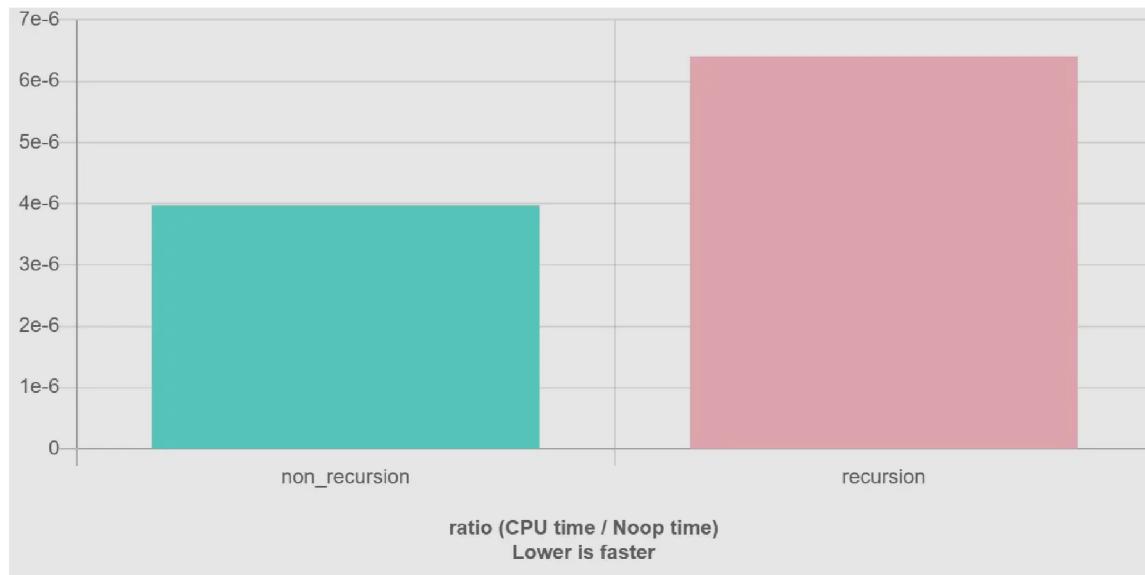
2. 练习分析问题的结构。当发现问题可以被分解成相同结构的小问题时，递归写多了就能敏锐发现这个特点，进而高效解决问题。

递归的缺点

在程序执行中，递归是利用堆栈来实现的。每当进入一个函数调用，栈就会增加一层栈帧，每次函数返回，栈就会减少一层栈帧。而栈不是无限大的，当递归层数过多时，就会造成 **栈溢出** 的后果。

显然有时候递归处理是高效的，比如归并排序；**有时候是低效的**，比如数孙悟空身上的毛，因为堆栈会消耗额外空间，而简单的递推不会消耗空间。比如这个例子，给一个链表头，计算它的长度：

```
1 // 典型的递推遍历框架
2 int size(Node *head) {
3     int size = 0;
4     for (Node *p = head; p != nullptr; p = p->next) size++;
5     return size;
6 }
7
8 // 我就是要写递归，递归天下第一
9 int size_recursion(Node *head) {
10    if (head == nullptr) return 0;
11    return size_recursion(head->next) + 1;
12 }
```



递归的优化

主页面：[搜索优化](#) 和 [记忆化搜索](#)

比较初级的递归实现可能递归次数太多，容易超时。这时需要对递归进行优化。¹

分治

定义

分治（英语：Divide and Conquer），字面上的解释是「分而治之」，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

过程

分治算法的核心思想就是「分而治之」。

大概的流程可以分为三步：分解 -> 解决 -> 合并。

1. 分解原问题为结构相同的子问题。
2. 分解到某个容易求解的边界之后，进行递归求解。
3. 将子问题的解合并成原问题的解。

分治法能解决的问题一般有如下特征：

- 该问题的规模缩小到一定的程度就可以容易地解决。
- 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质，利用该问题分解出的子问题的解可以合并为该问题的解。
- 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

⚠ 注意

如果各子问题是不独立的，则分治法要重复地解公共的子问题，也就做了许多不必要的工作。此时虽然也可用分治法，但一般用 [动态规划](#) 较好。

以归并排序为例。假设实现归并排序的函数名为 `merge_sort`。明确该函数的职责，即 **对传入的一个数组排序**。这个问题显然可以分解。给一个数组排序等于给该数组的左右两半分别排序，然后合并成一个数组。

```
1 void merge_sort(一个数组) {  
2     if (可以很容易处理) return;  
3     merge_sort(左半个数组);  
4     merge_sort(右半个数组);
```

```
5     merge(左半个数组, 右半个数组);  
6 }
```

传给它半个数组，那么处理完后这半个数组就已经被排好了。注意到，`merge_sort` 与二叉树的后序遍历模板极其相似。因为分治算法的套路是 **分解 -> 解决（触底） -> 合并（回溯）**，先左右分解，再处理合并，回溯就是在退栈，即相当于后序遍历。

`merge` 函数的实现方式与两个有序链表的合并一致。

要点

写递归的要点

明白一个函数的作用并相信它能完成这个任务，千万不要跳进这个函数里面企图探究更多细节，否则就会陷入无穷的细节无法自拔，人脑能压几个栈啊。

以遍历二叉树为例。

```
1 void traverse(TreeNode* root) {  
2     if (root == nullptr) return;  
3     traverse(root->left);  
4     traverse(root->right);  
5 }
```

这几行代码就足以遍历任何一棵二叉树了。对于递归函数 `traverse(root)`，只要相信给它一个根节点 `root`，它就能遍历这棵树。所以只需要把这个节点的左右节点再传给这个函数就行了。

同样扩展到遍历一棵 N 叉树。与二叉树的写法一模一样。不过，对于 N 叉树，显然没有中序遍历。

```
1 void traverse(TreeNode* root) {  
2     if (root == nullptr) return;  
3     for (auto child : root->children) traverse(child);  
4 }
```

区别

递归与枚举的区别

递归和枚举的区别在于：枚举是横向地把问题划分，然后依次求解子问题；而递归是把问题逐级分解，是纵向的拆分。

递归与分治的区别

递归是一种编程技巧，一种解决问题的思维方式；分治算法很大程度上是基于递归的，解决更具体问题的算法思想。

例题详解

437. 路径总和 III

给定一个二叉树，它的每个结点都存放着一个整数值。

找出路径和等于给定数值的路径总数。

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

二叉树不超过 1000 个节点，且节点数值范围是 [-1000000,1000000] 的整数。

示例：

```
1 root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8
2
3     10
4     /   \
5     5     -3
6     / \     \
7     3   2     11
8     / \     \
9     3   -2    1
10
11 返回 3。和等于 8 的路径有：
12
13 1.  5 -> 3
14 2.  5 -> 2 -> 1
15 3. -3 -> 11
```

```
1 // divide-and-conquer_1.h
2 // 二叉树结点的定义
3 struct TreeNode {
4     int val;
5     TreeNode *left;
6     TreeNode *right;
7
8     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
9 };
```



参考代码

```
1 #include "divide-and-conquer_1.h"
2
3 int count(TreeNode *node, int sum) {
4     if (node == nullptr) return 0;
5     return (node->val == sum) + count(node->left, sum - node->val)
6     +
7         count(node->right, sum - node->val);
8 }
9
10 int pathSum(TreeNode *root, int sum) {
11     if (root == nullptr) return 0;
12     return count(root, sum) + pathSum(root->left, sum) +
13         pathSum(root->right, sum);
14 }
```



题目解析

题目看起来很复杂，不过代码却极其简洁。

首先明确，递归求解树的问题必然是要遍历整棵树的，所以二叉树的遍历框架（分别对左右子树递归调用函数本身）必然要出现在主函数 pathSum 中。那么对于每个节点，它们应该干什么呢？它们应该看看，自己和它们的子树包含多少条符合条件的路径。好了，这道题就结束了。

按照前面说的技巧，根据刚才的分析来定义清楚每个递归函数应该做的事：

`PathSum` 函数：给定一个节点和一个目标值，返回以这个节点为根的树中，和为目标值的路径总数。

`count` 函数：给定一个节点和一个目标值，返回以这个节点为根的树中，能凑出几个以该节点为路径开头，和为目标值的路径总数。

参考代码（附注释）

```
1 int pathSum(TreeNode *root, int sum) {
2     if (root == nullptr) return 0;
3     int pathImLeading = count(root, sum); // 自己为开头的路径数
4     int leftPathSum = pathSum(root->left, sum); // 左边路径总数
5     (相信它能算出来)
6     int rightPathSum =
7         pathSum(root->right, sum); // 右边路径总数 (相信它能算出来)
8     return leftPathSum + rightPathSum + pathImLeading;
9 }
10
11
12 int count(TreeNode *node, int sum) {
13     if (node == nullptr) return 0;
14     // 能不能作为一条单独的路径呢？
15     int isMe = (node->val == sum) ? 1 : 0;
16     // 左边的，你那边能凑几个 sum - node.val ?
17     int leftNode = count(node->left, sum - node->val);
18     // 右边的，你那边能凑几个 sum - node.val ?
19     int rightNode = count(node->right, sum - node->val);
20     return isMe + leftNode + rightNode; // 我这能凑这么多个
21 }
```

还是那句话，**明白每个函数能做的事，并相信它们能够完成。**

总结下，`PathSum` 函数提供了二叉树遍历框架，在遍历中对每个节点调用 `count` 函数（这里用的是先序遍历，不过中序遍历和后序遍历也可以）。`count` 函数也是一个二叉树遍历，用于寻找以该节点开头的目标值路径。

习题

- LeetCode 上的递归专题练习
- LeetCode 上的分治算法专项练习

参考资料与注释

1. labuladong 的算法小抄 - 递归详解 ←



本页面最近更新：2024/11/10 20:22:04，[更新历史](#)



发现错误？想一起完善？[在 GitHub 上编辑此页！](#)



本页面贡献者：[Ir1d](#), [NachtgeistW](#), [CBW2007](#), [Enter-tainer](#), [sshyw](#), [AngelKitty](#), [houbaron](#), [iamtwz](#), [invalid-email-address](#), [abc1763613206](#), [alhofighter](#), [Alisahhh](#), [ChungZH](#), [flyhahe](#), [fudonglai](#), [Haohu Shen](#), [Konano](#), [ksyx](#), [labuladong](#), [leoleoasd](#), [Lutra-Fs](#), [melxy1997](#), [Menci](#), [shawlleyw](#), [StudyingFather](#), [Tiphereth-A](#), [TrickEye](#), [TrisolarisHD](#), [wongsyrone](#), [Xeonacid](#)



© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用

贪心

本页面将简要介绍贪心算法。

引入

贪心算法（英语：greedy algorithm），是用计算机来模拟一个「贪心」的人做出决策的过程。这个人十分贪婪，每一步行动总是按某种指标选取最优的操作。而且他目光短浅，总是只看眼前，并不考虑以后可能造成的影响。

可想而知，并不是所有的时候贪心法都能获得最优解，所以一般使用贪心法的时候，都要确保自己能证明其正确性。

解释

适用范围

贪心算法在有最优子结构的问题中尤为有效。最优子结构的意思是问题能够分解成子问题来解决，子问题的最优解能递推到最终问题的最优解。¹

证明

贪心算法有两种证明方法：反证法和归纳法。一般情况下，一道题只会用到其中的一种方法来证明。

1. 反证法：如果交换方案中任意两个元素/相邻的两个元素后，答案不会变得更好，那么可以推定目前的解已经是最优解了。
2. 归纳法：先算得出边界情况（例如 $n = 1$ ）的最优解 F_1 ，然后再证明：对于每个 n ， F_{n+1} 都可以由 F_n 推导出结果。

要点

常见题型

在提高组难度以下的题目中，最常见的贪心有两种。

- 「我们将 XXX 按照某某顺序排序，然后按某种顺序（例如从小到大）选择。」。
- 「我们每次都取 XXX 中最大/小的东西，并更新 XXX。」（有时「XXX 中最大/小的东西」可以优化，比如用优先队列维护）

二者的区别在于一种是离线的，先处理后选择；一种是在线的，边处理边选择。

排序解法

用排序法常见的情况是输入一个包含几个（一般一到两个）权值的数组，通过排序然后遍历模型计算的方法求出最优值。

后悔解法

思路是无论当前的选项是否最优都接受，然后进行比较，如果选择之后不是最优了，则反悔，舍弃掉这个选项；否则，正式接受。如此往复。

区别

与动态规划的区别

贪心算法与动态规划的不同在于它对每个子问题的解决方案都做出选择，不能回退。动态规划则会保存以前的运算结果，并根据以前的结果对当前进行选择，有回退功能。

例题详解

邻项交换法的例题

NOIP 2012 国王游戏

恰逢 H 国国庆，国王邀请 n 位大臣来玩一个有奖游戏。首先，他让每个大臣在左、右手上面分别写下一个整数，国王自己也在左、右手上各写一个整数。然后，让这 n 位大臣排成一排，国王站在队伍的最前面。排好队后，所有的大臣都会获得国王奖赏的若干金币，每位大臣获得的金币数分别是：排在该大臣前面的所有人的左手上的数的乘积除以他自己右手上的数，然后向下取整得到的结果。

国王不希望某一个大臣获得特别多的奖赏，所以他想请你帮他重新安排一下队伍的顺序，使得获得奖赏最多的大臣，所获奖赏尽可能的少。注意，国王的位置始终在队伍的最前面。

解题思路

设排序后第 i 个大臣左右手上的数分别为 a_i, b_i 。考虑通过邻项交换法推导贪心策略。

用 s 表示第 i 个大臣前面所有人的 a_i 的乘积，那么第 i 个大臣得到的奖赏就是 $\frac{s}{b_i}$ ，第 $i+1$ 个大臣得到的奖赏就是 $\frac{s \cdot a_i}{b_{i+1}}$ 。

如果我们交换第 i 个大臣与第 $i+1$ 个大臣，那么此时的第 i 个大臣得到的奖赏就是 $\frac{s}{b_{i+1}}$ ，第 $i+1$ 个大臣得到的奖赏就是 $\frac{s \cdot a_{i+1}}{b_i}$ 。

如果交换前更优当且仅当

$$\max\left(\frac{s}{b_i}, \frac{s \cdot a_i}{b_{i+1}}\right) < \max\left(\frac{s}{b_{i+1}}, \frac{s \cdot a_{i+1}}{b_i}\right)$$

提取出相同的 s 并约分得到

$$\max\left(\frac{1}{b_i}, \frac{a_i}{b_{i+1}}\right) < \max\left(\frac{1}{b_{i+1}}, \frac{a_{i+1}}{b_i}\right)$$

然后分式化成整式得到

$$\max(b_{i+1}, a_i \cdot b_i) < \max(b_i, a_{i+1} \cdot b_{i+1})$$

实现的时候我们将输入的两个数用一个结构体来保存并重载运算符：

```
1 struct uv {
2     int a, b;
3
4     bool operator<(const uv &x) const {
5         return max(x.b, a * b) < max(b, x.a * x.b);
6     }
7 }
```

后悔法的例题



「USACO09OPEN」工作调度 Work Scheduling

约翰的工作日从 0 时刻开始，有 10^9 个单位时间。在任一单位时间，他都可以选择编号 1 到 N 的 $N(1 \leq N \leq 10^5)$ 项工作中的任意一项工作来完成。工作 i 的截止时间是 $D_i(1 \leq D_i \leq 10^9)$ ，完成后获利是 $P_i(1 \leq P_i \leq 10^9)$ 。在给定的工作利润和截止时间下，求约翰能够获得的利润最大为多少。



解题思路

1. 先假设每一项工作都做，将各项工作按截止时间排序后入队；
2. 在判断第 i 项工作做与不做时，若其截至时间符合条件，则将其与队中报酬最小的元素比较，若第 i 项工作报酬较高（后悔），则 $ans += a[i].p - q.top()$ 。
用优先队列（小根堆）来维护队首元素最小。
3. 当 $a[i].d \leq q.size()$ 可以这么理解从 0 开始到 $a[i].d$ 这个时间段只能做 $a[i].d$ 个任务，而若 $q.size() >= a[i].d$ 说明完成 $q.size()$ 个任务时间大于等于 $a[i].d$ 的时间，所以当第 i 个任务获利比较大的时候应该把最小的任务从优先级队列中换出。



参考代码

C++

```
1 #include <algorithm>
2 #include <cmath>
3 #include <cstring>
4 #include <iostream>
5 #include <queue>
6 using namespace std;
7
8 struct f {
9     long long d;
10    long long p;
11 } a[100005];
12
13 bool cmp(f A, f B) { return A.d < B.d; }
14
15 // 小根堆维护最小值
16 priority_queue<long long, vector<long long>, greater<long long>>
17 q;
18
19 int main() {
20     long long n, i;
21     cin >> n;
22     for (i = 1; i <= n; i++) {
23         cin >> a[i].d >> a[i].p;
24     }
25     sort(a + 1, a + n + 1, cmp);
26     long long ans = 0;
27     for (i = 1; i <= n; i++) {
28         if (a[i].d <= (int)q.size()) { // 超过截止时间
29             if (q.top() < a[i].p) { // 后悔
30                 ans += a[i].p - q.top();
31                 q.pop();
32                 q.push(a[i].p);
33             }
34         } else { // 直接加入队列
35             ans += a[i].p;
36             q.push(a[i].p);
37         }
38     }
39     cout << ans << endl;
40     return 0;
41 }
```

Python

```
1 from collections import defaultdict
2 from heapq import heappush, heappop
3
```

```
4  a = defaultdict(list)
5  for _ in range(int(input())):
6      d, p = map(int, input().split())
7      a[d].append(p) # 存放对应时间的收益
8
9  ans = 0 # 记录总收益
10 q = [] # 小根堆维护最小值
11 l = sorted(a.keys(), reverse=True)
12 for i, j in zip(l, l[1:] + [0]):
13     for k in a.pop(i):
14         heappush(q, -k)
15     for _ in range(i - j):
16         if q: # 从堆中取出收益最多的工作
17             ans += -heappop(q)
18         else: # 堆为空时退出循环
19             break
20 print(ans)
```

复杂度分析

- 空间复杂度：当输入 n 个任务时使用 n 个 a 数组元素，优先队列中最差情况下会储存 n 个元素，则空间复杂度为 $O(n)$ 。
- 时间复杂度：`std::sort` 的时间复杂度为 $O(n \log n)$ ，维护优先队列的时间复杂度为 $O(n \log n)$ ，综上所述，时间复杂度为 $O(n \log n)$ 。

习题

- [P1209\[USACO1.3\] 修理牛棚 Barn Repair - 洛谷](#)
- [P2123 皇后游戏 - 洛谷](#)
- [LeetCode 上标签为贪心算法的题目](#)

参考资料与注释

1. 贪心算法 - 维基百科，自由的百科全书 [←](#)

本页面最近更新：2025/8/30 13:34:30，[更新历史](#)
发现错误？想一起完善？[在 GitHub 上编辑此页！](#)
本页面贡献者：[StudyingFather](#), [Ir1d](#), [H-J-Granger](#), [NachtgeistW](#), [ksyx](#), [countercurrent-time](#), [Enter-tainer](#), [mgt](#), [abc1763613206](#), [Makkiy](#), [niltok](#), [sshwy](#), [AngelKitty](#), [CCXXXI](#), [cjsoft](#), [diauweb](#), [Early0v0](#), [ezoixx130](#), [GekkaSaori](#), [Henry-ZHR](#), [HeRaNO](#), [hsfzLZH1](#), [Konano](#),

[LovelyBuggies](#), [Marcythm](#), [minghu6](#), [ouuan](#), [P-Y-Y](#), [PotassiumWings](#), [SamZhangQingChuan](#), [Suyun514](#), [Tiphereth-A](#), [weiyong1024](#), [Chrogeek](#), [ChungZH](#), [FTYC919](#), [GavinZhengOI](#), [Gesrua](#), [Great-designer](#), [iamtwz](#), [kenlig](#), [kxccc](#), [LeiJinpeng](#), [leoleoasd](#), [lychees](#), [Peanut-Tang](#), [Planet6174](#), [qiqistyle](#), [RoxasKing](#), [shawlleyw](#), [SukkaW](#), [tinjyu](#), [Xeonacid](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用

