

Problem A. 莫卡与 MCPC

只有 2 既是偶数又是质数，只需要判断状态码是不是 2 即可。

时间复杂度 $O(n)$ 。

Problem B. bb 去食堂

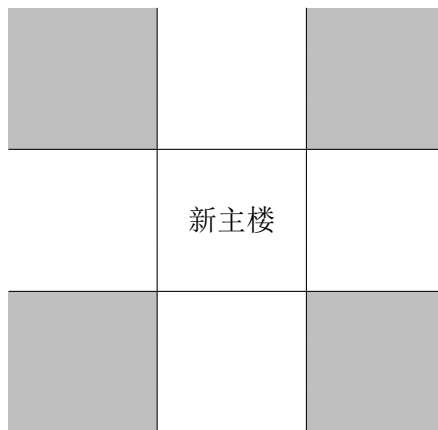


图 B.1: 区域划分

如图 B.1 所示，可以根据新主楼的位置，将平面直角坐标系划分为九个区域。在四角区域内的点，最近距离为该点到对应矩形顶点连线段的长度。在空白区域内的点，最近距离为该点到对应矩形边垂线段的长度。枚举所有点，分类讨论计算距离即可。

时间复杂度 $O(n)$ 。

Problem C. 抵御阿草

本题可以分若干种情况讨论。

- $n = 0$ 时，特判输出 0。
- 否则，首先考虑没有异或的情况：
 - 考虑第一次操作，对于每一对需要合并的数，只有最后一位不同。因此与/或后只有最后一位变化了，其余的位没有发生改变，并且会发现这次操作后所有数的最后一位都相同。如果这次操作是或，最后一位是 1，否则为 0。
 - 考虑第 i 次操作，最后 $i - 1$ 位完全相同，本次操作后不会发生改变。因此忽略后 $i - 1$ 位可变为第一次操作的情况。

这种情况下，根据与/或的操作次序向答案添加 1,0 即可。

- 考虑异或出现次数为 1：
 - 假设此时是第 i 次操作。由于所有数后 $i - 1$ 位全部相同，因此异或后所有数的后 $i - 1$ 位会全部变为 0。对于第 $i + 1$ 位及前面的位，在合并的时候，相邻的两个数一定是相同的，因此也会全部变为 0。第 i 位则会变为 1。因此操作后所有数均会变为 2^{i-1} ，并且会发现之后的所有与/或对答案不会产生任何影响。

- 考虑异或出现次数大于 1:

– 由于第一次异或后所有数都相同了, 因此第二次异或后所有数都会变为 0, 答案为 0。

时间复杂度 $O(n)$ 。

Problem D. 皮卡丘与宝可梦对战模拟器

本题是一道模拟题, 只要按照严格题目的描述, 几乎使用任何语言、任何实现方式都能通过。

题目的第一个难点是输入。使用 Python 语言实现起来会比较方便, 例如可以写成以下形式。这段代码含义是先把两侧的 `SS:` 和空格等字符删去, 再按照 `/` 分割, 即可得到这六项数值。

```
SSs = input().strip('Ss: ').split('/')
```

使用 C/C++ 语言编写输入稍麻烦一些。可以将整个字符串读入进来, 再自行处理, 也可以利用格式化输入的技巧来简化实现。在下面代码中, `scanf` 里的 `SSs:` 恰好消耗输入的 `SSs:`, 随后的空格消耗任意数量的空白符 (因而也可省略, `%d` 会消耗这些多余的空白符)。`fgets` 的作用是消耗上一次 `scanf` 没有消耗的换行符。

```
scanf("%d", &level);
fgets(input, 100, stdin);
scanf("SSs: %d/%d/%d/%d/%d/%d", &ss[0], &ss[1], &ss[2], &ss[3], &ss[4], &ss[5]);
fgets(input, 100, stdin);
scanf("IVs: %d/%d/%d/%d/%d/%d", &iv[0], &iv[1], &iv[2], &iv[3], &iv[4], &iv[5]);
fgets(input, 100, stdin);
scanf("EVs: %d/%d/%d/%d/%d/%d", &ev[0], &ev[1], &ev[2], &ev[3], &ev[4], &ev[5]);
```

第二个难点在题目中也有提示。由于本题需要舍入为整数, 因此不论多高精度的浮点数都不能保证正确。例如 1 可能被计算为 0.999...999 (很多个 9), 在向下取整时被舍去, 导致错误。本题提到的所有公式都需要手动进行分数计算, 相信这对于大家不成问题。:)

本题最暴力的算法如下。首先枚举 i 和 j , 计算 i 与 j 对战时的答案。不考虑随机数, 每个技能造成的伤害是固定的。要使 i 尽可能获胜, 显然 i 每次的随机数要是 1, j 每次的随机数要是 0.85, i 使用最好的技能, j 使用最劣的技能。为了方便起见, 可以分别枚举 i , j 使用哪个技能, 这样总是能够枚举到对 i 最优的情况。之后进行对战的模拟, 如果 i 的速度大于等于 j 的速度, 最优情况下 i 总能先手。这样的时间复杂度是 $O(n^2 m^2 \text{HP})$, 其中 $m = 4$ 表示技能数量, HP 是宝可梦最大可能的 HP, 为 714。

考虑优化, 可以事先将 i 最优和 j 最劣的技能及其伤害计算出来, 再模拟对战, 这样就去掉了 m^2 的复杂度。接下来还可以发现只需要计算 i 击败 j 及 j 击败 i 所需的回合数, 结合速度即可判断胜负。复杂度可以做到 $O(n^2)$, 但是本题不要求实现这些优化。

Problem E. 圣莫卡造题的七天

显然每次操作的路径上的点只能在同一个连通块中。且整体的最大值最小, 相当于要求每个连通块的最大值最小。因此对每个连通块分别求解。

对于一个连通块, 其答案即为该连通块中所有点的权值进行与运算的结果, 记为 a 。注意到 a 中为 1 的位在所有点中都为 1, 无论怎么操作都不会改变, 因此答案不能比 a 更小。对于 a 中为 0 的位, 若存

在某个点该位为 1，必然还存在某个点该位为 0。从后者开始进行 dfs，dfs 过程中，对于每个新点，将它与它的父亲形成的路径（长度为 2）进行操作。由于父亲在之前的操作中该位已经为 0，因此新点该位也会变成 0。最终所有点该位都会变成 0。

只需根据该结论求出每个连通块的答案，取所有连通块的最大值即可。

时间复杂度 $\mathcal{O}(n)$ 。

Problem F. PotasHub Copylot

F.1 出题人的做法

对于固定的 x ，假设其下标为 pos，让原数组变成 $[p_i < x]$ ，则 x 在某个区间的排名等价于新数组该区间的和 +1。令 f 为现数组的前缀和，则答案等于

$$\sum_{i=\text{pos}}^n \sum_{j=1}^{\text{pos}} f(i) - f(j-1) + 1$$

这个式子可以拆解出来进行处理，实际上只要维护 f 的前缀和即可。从小到大枚举 x ，则每次都只会将 f 区间加等差数列，可以线段树维护。

F.2 验题人的做法

对每个 a_i ，答案为三个部分的贡献：

- 左边比 a_i 小的数的个数 $\times$ 合法的右区间端点个数
- 右边比 a_i 小的数的个数 $\times$ 合法的左区间端点个数
- 包含了 a_i 的合法区间个数

第一部分和第二部分都可以用树状数组维护，因此是一个小清新数据结构题。

时间复杂度 $\mathcal{O}(n \log n)$ 。

Problem G. Serval 的字符串

G.1 方法一

对于给出的字符串 T_i ，朴素的做法是直接判断 S 的每一位是否与 T_i 拼接后字符串的对应位置相同，这样处理的时间复杂度是 $\mathcal{O}(|S|)$ 。总时间复杂度为 $\mathcal{O}(q|S|)$ ，并不能通过本题。

另一种做法是，注意到对于串长为 $|T_i|$ 的任意串， S 的每个字符只会与这个字符串中的某个字符对应。这个对应关系只与 $|T_i|$ 有关，而与具体的字符串无关，因此我们可以预处理 $f(|T_i|, r, c)$ ，表示当串长为 $|T_i|$ 时， S 所有对应到字符串第 r 个字符的这些字符，有多少个是 c 。利用这些信息，我们可以在 $\mathcal{O}(|T_i|)$ 的时间内求出一个字符串的答案。但如果对 $t \leq |S|$ 的均预处理 $f(t, r, c)$ ，则计算答案部分的时间复杂度为 $\mathcal{O}(\sum_{i=1}^q |T_i|)$ ，非常优秀，但预处理部分的复杂度为 $\mathcal{O}(|S|^2)$ ，并不能通过本题。

考虑只预处理出 $t \leq t'$ 时的 $f(t, r, c)$, 当 $|T_i| > t'$ 时利用朴素做法求出答案, 当 $|T_i| \leq t'$ 时利用预处理的信息求出答案。我们需要求出 t' 取何值时有最优复杂度。

由于 $\max\{\sum_{i=1}^q |T_i|\} = \max\{|S|\} = 2 \times 10^5$, 不妨记参数 $T = 2 \times 10^5$ 。

我们简单估计满足 $|T_i| > t'$ 的字符串 T_i 的数量, 显然不超过 $\frac{T}{t'}$, 于是朴素做法部分的复杂度上界为 $\mathcal{O}(\frac{T}{t'}T)$ 。显然地, 预处理的复杂度上界为 $\mathcal{O}(t'T)$, 利用预处理信息计算答案的复杂度上界为 $\mathcal{O}(T)$, 于是合并两种做法后总时间复杂度上界为 $\mathcal{O}(\frac{T}{t'}T + t'T + T)$ 。

依均值不等式有:

$$\frac{T}{t'}T + t'T \geq 2\sqrt{\frac{T}{t'}T \cdot t'T} = 2T\sqrt{t'}$$

当且仅当 $\frac{T}{t'}T = t'T$ 即 $t' = \sqrt{T}$ 时取得等号, 因此取 $t' = \sqrt{T}$ 时有最优复杂度 $\mathcal{O}(T\sqrt{T})$, 可以通过本题。

G.2 方法二

注意到不同的 $|T_i|$ 至多只有 $\mathcal{O}(\sqrt{T})$ 种, 因此可以只预处理这些 $|T_i|$ 的信息 $f(|T_i|, r, c)$, 并利用这些信息求出答案。时间复杂度也为 $\mathcal{O}(T\sqrt{T})$ 。

Problem H. 狂乱

首先考虑两个随从之间互相攻击的情况, 假设第一个随从是 a 攻、 b 血, 第二个随从是 c 攻、 d 血。

那么第一个随从想要杀死第二个, 需要攻击 $\lceil \frac{d}{a} \rceil$ 次, 在这个过程中第一个随从会受到 $c \times \lceil \frac{d}{a} \rceil$ 的伤害。

要想存活下来, 需要满足 $b > c \times \lceil \frac{d}{a} \rceil$, 因此 $\frac{b}{c} > \lceil \frac{d}{a} \rceil \geq \frac{d}{a}$, 即必须满足 $b \times a > c \times d$ 的条件。

所以, 攻击力乘血量较小的一方必定不可能存活下来。这一点样例五已有提示。

那么只需要按照攻击力乘血量排序, 取最大的两个作为指定随从, 测试是否能活下来即可。从第三个开始, 必定在与第一个或者第二个战斗中死去, 从而无法杀死另一个。

判定一个随从是否能消灭其他所有随从, 由于所有其他随从对该随从的伤害总量是固定的, 所以只需要考虑这个随从在攻击完其他随从之后血量是否小于等于 0, 以及最后一次攻击前血量是否大于 0 即可。

可以发现, 最后一次攻击选择最高攻击力的敌人最有可能达成上述条件, 这一点可以从样例二中得知。所以这里直接贪心地选择, 计算总伤害减掉最高攻击力即可。

需要特别注意一下攻击力为 0 的情况。

时间复杂度 $\mathcal{O}(n \log n)$ 。

Problem I. Serval 的相对论

容易观察发现距离 d 单调不增。为了获得胜利, 我们需要找到一种方式使得距离 d 固定在某个值。

设玩家位于 (x, y) , 小 bot 位于 (x', y') 。当 $y \neq y'$ 且 $|x - x'| = 1$ 时, 小 bot 将会选择移动到 (x, y') , 此时玩家选择移动到 (x', y) 将会使距离 d 不变, 且仍有 $|x - x'| = 1$ 。因此, 当条件满足时, 使得距离 d 不变的一个可行的策略是交替左右移动或右左移动, 我们称这个策略为左起或右起**反复横跳**, 起跳方向取决于小 bot 相对于玩家的位置。

这个条件过于苛刻, 我们不能通过距离判断出 (x', y') 。观察发现, 当 $|y - y'| \geq 2$ 时, 左右横跳一定不会被抓到。我们将证明这一点:

- 情况一: $x = x'$

在这一轮中, 小 bot 将选择向上或向下移动, 而玩家无论选择向左移动还是向右移动, 均使得移动后 $|x - x'| = 1$, 并且有 $|y - y'| \geq 1$, 即 $y \neq y'$ 。结合之前的讨论可以知道, 此时左起或右起反复横跳均不会被抓到。

- 情况二: $|x - x'| = 1$

不失一般性地, 假设 $x = x' - 1$ 。在这一轮中, 小 bot 将选择向左移动。若玩家选择向右移动, 则相当于右起反复横跳, 不会被抓到; 若向左移动, 则 $|x - x'|$ 不变, 下一步玩家选择向右移动, 开始右起反复横跳, 也不会被抓到。因此当前选择左起或右起反复横跳均不会被抓到。

- 情况三: $|x - x'| \geq 2$

每轮移动不会改变 $|y - y'|$, 并且 $|x - x'|$ 交替不变和减少 2, 从而在有限轮后转化为情况一和情况二。因此反复横跳不会被抓到。

此时策略非常明显了, 我们只需要保证 $|y - y'| \geq 2$ 再反复横跳即可。初始时, 我们需要判断 $y' \leq y$ 还是 $y' > y$ 。不难发现, 若第一轮选择向上移动后距离减少了, 那么有 $y' > y$, 否则有 $y' \leq y$ 。当 $y' \leq y$ 时选择向上移动两轮 (实际上一轮即可保证), 当 $y' > y$ 时选择向下移动两轮, 由初始距离至少为 4 可保证有 $|y - y'| \geq 2$ 。

综上所述, 我们可以给出如下构造:

- 记初始距离为 d_0 。
- 第一轮选择向上移动, 记移动后距离为 d_1 。
- 若 $d_0 \neq d_1$, 则接下来两轮中选择向下移动, 否则在接下来一轮中选择向上移动。
- 交替选择向左向右移动, 直至游戏结束。

Problem J. 数据重命名

J.1 添加约束下的最优解

不妨先考虑为原问题添加一个附加条件: 移动完最后一步时, 需要将光标重新移回最后一个文件的下一个位置 (即使不存在), 那么每次移动 i 实际上可以分为两部分: 将光标从第一个未改名的位置移动到 p_i , 再将飞到 q_i 的光标移动回第一个未改名的位置。令 t_i 表示文件 i 移动的次序。

对于第一部分, 光标需要经过所有 $t_j > t_i$ 且 $p_j < p_i$ 的文件, 这一部分需要移动的次数为 $\sum_j [t_j > t_i \wedge p_j < p_i]$ 。注意到

$$\begin{aligned}
& \sum_j [t_j > t_i \wedge p_j < p_i] \\
&= (p_i - 1) - \sum_j [t_j < t_i \wedge p_j < p_i] \\
&= (p_i - 1) - (t_i - 1 - \sum_j [t_j < t_i \wedge p_j > p_i]) \\
&= p_i - t_i + \sum_j [t_j < t_i \wedge p_j > p_i]
\end{aligned}$$

对于第二部分, 光标需要经过所有 $t_j < t_i$ 且 $q_j > q_i$ 的文件, 这一部分需要移动的次数为 $\sum_j [t_j < t_i \wedge q_j > q_i]$, 然后多操作一次将光标移到第一个未改名的位置。

综上, 在该附加条件下的总操作次数为

$$\begin{aligned}
& \sum_{i=1}^n \left\{ p_i - t_i + \sum_j [t_j < t_i \wedge p_j > p_i] + \sum_j [t_j < t_i \wedge q_j > q_i] + 1 \right\} \\
&= \sum_{i=1}^n p_i - \sum_{i=1}^n t_i + \sum_i \sum_j \{ [t_j < t_i \wedge p_j > p_i] + [t_j < t_i \wedge q_j > q_i] \} + n
\end{aligned}$$

显然 $\sum_{i=1}^n p_i - \sum_{i=1}^n t_i$ 对固定的 n 来说一定是定值。对于后半部分, 其等价于将 (p_i, q_i) 按一定顺序放在二维平面上, 每新放一个点时, 若有点在其右方或上方, 那么答案将增加相应的个数。可以发现为了最小化答案, 若两个点 a, b 在二维偏序下满足 $a < b$, 那么一定是先放 a 后放 b ; 若 a, b 无偏序关系, 那么无论按什么顺序都会产生 1 的贡献。因此, 任何满足二维偏序的拓扑序都是最优的。我们很高兴地发现, 按照 p_i 排序的结果就是一个合法的拓扑序, 因此问题得解。

J.2 原问题下的最优解

原问题中在改完最后一个文件名后, 不需要将光标重新移回最后一个文件的下一个位置, 因此我们可以枚举最后一个修改的文件, 然后考虑将前面的位置调整为最优解。

根据上一个问题的讨论, 可以发现此时最优解等于 n 个文件在有附加条件下的最优解去除我们所枚举的文件。去除这个文件的贡献值是便于计算的, 可以用树状数组或线段树等结构维护“它之前有多少 q_i 比它大”和“它之后有多少 q_i 比它小”即可。

时间复杂度 $O(n \log n)$ 。

Problem K. 皮卡丘与 Minimum Spanning Tree-I

题解中, 称输入的边的普通边, 其它边为额外边。这里给出两种解法。

K.1 Kruskal 算法

本题中除普通的并查集外, 还需要用 `set` 维护所有连通块, 用 `vector` 维护每个连通块内的点。合并两个连通块时, 将 `vector` 启发式合并即可。

将所有边按照边权、所有点按照点权一同排序，从小到大枚举。如果枚举到一条普通边，正常合并即可。如果枚举到了一个点，那么枚举所有连通块（除去当前点所在的连通块），依次检查连通块中每个点和当前点是否有普通边，如果没有普通边，说明找到了一条额外边，即可合并两个连通块并 `break`。

唯一需要分析复杂度的是检查普通边是否存在的部分。注意到每次检查失败，就说明存在一条普通边，而每条普通边 $(u, v)(a[u] < a[v])$ 只会在枚举点 u 时检查一次，因此这部分只会检查 $\mathcal{O}(m)$ 次。

总时间复杂度 $\mathcal{O}(m \log n)$ 。

如果使用链表维护连通块和连通块中的点，并用一个 `vis` 数组记录存在的普通边，可以将排序以外的复杂度降低到 $\mathcal{O}(m \alpha(n))$ 。

K.2 Borůvka 算法

算法内容可参考[维基百科](#)。简单来说，初始时每个点是孤立点，算法进行若干轮，每轮给每个点选取一条最小的连边。可以证明必然有一棵最小生成树包含这些边，因此可将它们都连接起来，缩点后完成一轮。这样操作若干轮后，最终只剩一个结点，算法结束。由于每轮会使得点至少减少一半，它的复杂度显然是 $\mathcal{O}(m \log n)$ 。对于权值相同的边，需要人为指定一个顺序，否则可能导致成环。如果人为指定了顺序，可以认为所有边权互不相同。这种情况下可以证明不会成环，留给读者自己思考。

本题中，需要较为小心地处理额外边，可能存在很多种不同的实现方式，这里给出其中一种。

首先与 K.1 节相同，需要用 `set` 维护所有连通块，用 `vector` 维护每个连通块内的点。每轮合并结束后，可以把连通块内的非树边删除。

为了把 `min` 拆开，可以假定额外边为两条，一条权值为 a_u ，另一条权值为 a_v ，这不会影响答案。每一轮，除了需要点 u 相邻的普通边的最小值外，还要考虑额外边。对于权值为 a_u 的这类额外边，只需要看连通块大小加上边集大小是否小于 n 即可判断额外边是否存在。对于所有存在额外边的结点，只需要取 a_u 最小的那个。另外，还需要找到一条具体的边用于合并，可以将该连通块与边集归并，归并后的数组中间的“空隙”即为一个解。对于权值为 a_v 的这类额外边，需要考虑哪些 v 与连通块内所有点都有普通边，否则 a_v 就与该连通块有额外边。这里可以也可以通过归并将连通块内所有点连同它们的普通边合并起来并计数。注意到需要权值最小的 a_v ，因此需要事先将点按照 a 排序。

时间复杂度 $\mathcal{O}(m \log n)$ 。

Problem L. 皮卡丘与 Minimum Spanning Tree-II

L.1 朱刘算法

作为本题的前置知识，有必要在这里回顾一下最小树形图的算法。作为参考，可以阅读这份[课件](#)。本文的部分示例也改编自该课件。

考虑一种简单的策略，为根以外的每个点选取它权值最小的入边。如果很幸运地没有产生环，那么显然该解是一个树形图，是最优解。但是如图 L.1 所示，一般情况下最小入边可能成环，此时该解不是树形图。

可以证明，存在一个最优解包含该环中的一条链。任选一个最优解，设环上的点依次为 $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_r \rightarrow v_1$ ，其中 v_1 是该环在该解中离根最近的（任意一个）点。考虑修改该解

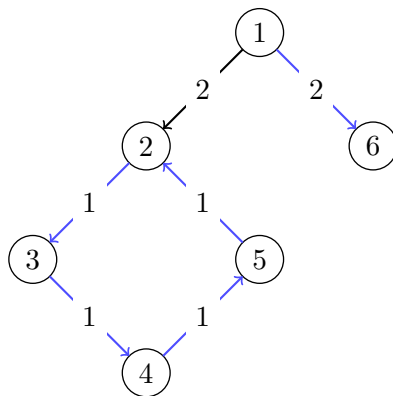


图 L.1: 一个产生环的示例

使其包含环中的一条链。由于 v_1 离根最近，因而 v_2 必然不是 v_1 的祖先，可以把 v_2 所在的子树整体移动到 v_1 下面。由于 $v_1 \rightarrow v_2$ 是 v_2 的最小入边，因此这样移动不会使得代价增加，从而得到的仍是最优解。同理， v_i 不会是 $v_1, v_2, \dots, v_{i-1}$ 的祖先，可以将它所在的子树整体移动到 v_{i-1} 的下面，且不会使得代价增加。最终通过调整得到了包含环中一条链的最优解。

基于这个性质，可以强制要求选取该环中的一条链。注意到选定该环的入边后，不妨设该边的终点为 v_i ，那么只能取 $v_i \rightarrow \dots \rightarrow v_r \rightarrow v_1 \rightarrow \dots \rightarrow v_{i-1}$ 这条链，它的权值相当于整个环的权值减去边 $v_{i-1} \rightarrow v_i$ 的权值。考虑将环缩成一个点，给答案预先加上环的总权值。每个点的出边不受影响，简单合并即可。点 v_i 的所有入边，需要减去边 $v_{i-1} \rightarrow v_i$ 的权值。之后再将所有点的入边合并起来。环内的边可直接删去。在新图上继续求解最小树形图。

注意到每次缩点至少使得点数减 1，算法至多进行 n 轮。每轮的复杂度是线性的，因此该算法的复杂度是 $\mathcal{O}(n(n+m))$ 。

L.2 Tarjan 的优化

该算法假定图是强连通的。如原图不强连通，可加入 $1 \rightarrow 2 \rightarrow \dots \rightarrow n \rightarrow 1$ 这 n 条权值为 $+\infty$ 的边使得图强连通。

算法从任一结点出发，维护一个结点的栈，栈中 $s_1, s_2, \dots, s_t$ 表示 s_i 的最小入边的起点为 s_{i+1} 。开始时栈中只有初始结点。每一步，算法选取 s_t 边权最小的入边，分几种情况讨论：

1. 若边的起点不在栈中，将它加入栈
2. 若边的起点是 s_i ($i < t$)，将 $s_i, \dots, s_t$ 全部出栈，相应修改它们入边的边权，合并为一个新的超级结点，将超级结点加入栈。这相当于朱刘算法中的缩点操作
3. 若边的起点为 s_t (因为有缩点操作)，不进行任何操作

操作后，将该条边删除。由于图强连通，最终必然能够将所有点缩成一个点。这一步骤称为收缩。可以注意到，收缩操作与根无关。这也是该算法的一个优点，经过收缩预处理后，可以线性求出以任意点为根的最小树形图。

举例来说，考虑图 L.2 中的有向图。从 1 开始，收缩的过程如表 L.1 所示。

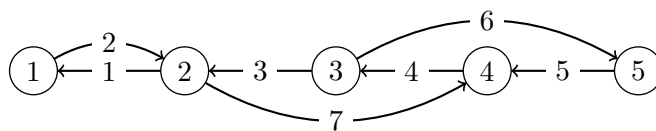


图 L.2: 示例图

栈	事件	入边
{1}	-	-
{1, 2}	2 入栈	In[1] = (2, 1)
{6}	1, 2 缩环成为 6	In[2] = (1, 2)
{6, 3}	3 入栈	In[6] = (3, 2)
{6, 3, 4}	4 入栈	In[3] = (4, 3)
{6, 3, 4, 5}	5 入栈	In[4] = (5, 4)
{6, 7}	3, 4, 5 缩环成为 7	In[5] = (3, 5)
{8}	6, 7 缩环成为 8	In[7] = (2, 4)

表 L.1: 收缩过程

收缩操作建立了一棵收缩树。这棵树是一棵有序的有根树，其叶子结点是有来的结点；非叶子节点是缩环产生的结点，以形成它的环的结点按顺序作为孩子；根节点是收缩完成时剩下的单点。图 L.2 的收缩树如图 L.3 所示。

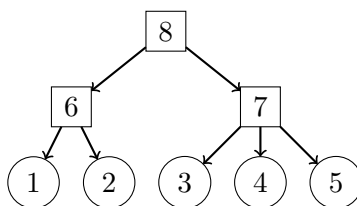


图 L.3: 收缩树示例

为了在收缩操作中取得优于平方的复杂度，需要一种数据结构支持快速查询最小值，删除最小值，合并两个集合，以及给集合整体加上一个值。带 lazy 标记的可并堆（例如左偏树）即可实现。时间复杂度为 $\mathcal{O}(m \log n)$ 。

给定一个根 r ，需要从收缩树中构建出最小树形图。设收缩树的根到 r 的路径依次为 $f_1 \rightarrow \dots \rightarrow f_t = r$ ，从最后的单点 f_1 开始逆向拆环。所有组成 f_1 的点中，由于 f_2 包含根 r ，因此必然是选取从它开始的链。对于 f_1 的其它孩子，从 In 可以得知它们的入边，不妨设点 v 的入边为 $u \rightarrow v'$ 。由于最小树形图中已经有 $u \rightarrow v'$ 这条边，因此在 v 的子树中不可能再选取任何 v' 的入边。故可将 v 的子收缩树以 v' 为根作为一个子问题求解。同理可以将 $f_2, \dots, f_{t-1}$ 的环都拆开。容易看出，该算法是线性的。下面是一段示例代码，in_result 求得每个点的入边。

```
void dismantle(int u, std::vector<int> &vec){
    while (parent[u]){
        for (auto v : children[parent[u]]){
            if (v == u){
                continue;
            }
        }
    }
}
```

```

    }
    parent[v] = 0;
    vec.emplace_back(v);
}
u = parent_back[u];
}
}

void expand(int rt){
    std::vector<int> vec;
    dismantle(rt, vec);
    while (!vec.empty()){
        int c = vec.back();
        vec.pop_back();
        int id = in[c];
        in_result[v[id]] = id;
        dismantle(v[id], vec);
    }
}
}

```

若选取 1 为根，示例图的拆环过程如表 L.2 所示。

子树	根	环	选取的边
8	1	8	$2 \rightarrow 4$
8	1	6	$1 \rightarrow 2$
7	4	7	$4 \rightarrow 3, 3 \rightarrow 5$

表 L.2: 拆环过程

因此，总时间复杂度为 $\mathcal{O}(m \log n)$ 。

L.3 题解

题解中，称输入的边为普通边，其它边为额外边。

L.3.1 收缩部分对额外边的维护

收缩过程的主要难点是对额外边的维护。给每个点的边集中加入一条特殊边，表示所有以该点为终点的额外边。两类边用相同的堆一起维护。

从堆中取出的最小入边，如果是普通边，那么自然可以用原始算法解决。如果取出的边是特殊边，那么其代表的所有额外边权值均相同，可以任选一条边。不妨规定先拓展那些不在栈中的结点，对于这些结点，按编号选取最小的那个。若没有这样的结点，再尝试选一条在栈中的入边（即缩环）。

先讨论如何查找不在栈中的结点。使用一个 `set` 维护当前不在栈中的点，另外对每个结点 u ，记录之前查询过的结点编号的最大值 `check[u]`。初始时 `check[u] = 0`。当枚举到 u 的特殊边时，显然可以

从 `set.upper_bound(check[u])` 开始查找, 查找的方式是枚举 `set` 中的每一个结点 v , 检查 $v \rightarrow u$ 是否存在。若不存在, 说明有相应的额外边, 因此找到了一个点。此时就可以 `break`, 将该额外边像普通边一样加入栈。检查每条边后, 都要相应更新 `check` 数组。

这样做的正确性在于, 点不会离开栈, 并且已经 `check` 过的点也不需要再次 `check`, 因此这样没有漏解。对于复杂度, 首先计算 `upper_bound`, 由于入栈和缩环分别至多进行 $\mathcal{O}(n)$ 轮, 因此它的复杂度是 $\mathcal{O}(n \log n)$ 。再考虑查找了但是没找到的边数, 可以发现, 额外边检查失败, 一定是因为存在相应的普通边。由于记录了 `check[u]`, 因此每条边至多被检查 1 次。若使用 `set` 记录普通边是否存在, 时间复杂度为 $\mathcal{O}(m \log n)$ 。显然, 如果不记录 `check` 数组, 而是每次从头检查的话, 复杂度将退化成平方, 题目有相应的数据卡这种做法。

如果不存在合法的未入栈点, 还需要查找是否有栈中的点到该点有额外边。考虑用 `vector` 维护每个点包含的所有普通结点, 缩环时启发式合并即可。按顺序枚举栈中的每个结点包含的所有普通结点, 检查其到 u 是否有普通边, 如果找到了额外边, 则可以像普通边一样缩环。这里注意不要查找 u 所在的连通块 (即只检查到栈顶前一个点)。不论是否找到额外边, 这条特殊边以后都不会再起作用了, 可以删除。因此这部分每条特殊边只会做一遍, 复杂度显然也是 $\mathcal{O}(m \log n)$ 的。这一性质的证明如下。不妨设找到的边是 $v \rightarrow u$, 对于 v 之前的点, 它们已经经过检查不含到 u 的额外边; 对于 v 到 u 之间的点, 它们已经合并成一个连通块, 以后都不能再连边; 对于未入栈的点, 也已经经过检查不含额外边 (事实上, 即使含有额外边, 从算法流程可以看出未入栈的点不会跑到 u 的左边, 因此以后也不会有用)。

可见, 收缩部分的时间复杂度与没有额外边的情况相同, 仍为 $\mathcal{O}(m \log n)$ 。

L.3.2 同时求出 n 个点为根的答案

考虑使用树 `dp` 求解。对于每个点, `dfs` 到它时, 求出以它为子树, 以它包含的每个普通结点为根时的最小树形图 (的总权值)。不妨设当前结点为 u , 它的任意孩子 v , v 包含任意普通结点 w 。 u 子树, w 为根的总权值包含几个部分。一是 v 子树, w 为根的总权值, 这已经计算出来了。二是拆解 u 这个环时的权值, 它等于 u 这个环的总权值减去 v 的入边的权值, 可以线性计算。三是 u 的其它孩子带来的权值, 同样可以用所有孩子的总权值减去 v 带来的权值。写成式子大致是

$$dp_{v_i, w} + \sum_{j=1, j \neq i}^d \text{weight}(\text{In}[v_j]) + \sum_{j=1, j \neq i}^d dp_{v_j, \text{end}[\text{In}[v_j]]}$$

注意到第二、三部分对所有 w 都是一样的, 所以只需要使用数据结构支持区间修改、单点查询即可, 比如树状数组。同时注意到 `dp` 数组可以只用一个, 不同的结点不会互相干扰。因为对一个点来说, 非祖先、非子孙的结点与它的区间一定不会相交。这部分的时间复杂度为 $\mathcal{O}(n \log n)$ 。还可以使用带权并查集将复杂度优化为 $\mathcal{O}(n \alpha(n))$, 由于不影响总复杂度, 这里就不赘述了, 留给读者自己思考。

总时间复杂度为 $\mathcal{O}(m \log n)$ 。实现时有巨量的细节, 请参考标程。