



难度

签到: K I B

简单: C D E

中等: A L M F

难: G J H

A

数论/打表

下面给出题目的思路分析、最优解及其正确性证明。

一、关键观察

- 定义：一个自然数 x （写作各位数字为 $a_1 a_2 \cdots a_n$ ）若满足对于 $2 \leq i \leq n-1$

$$a_i < \frac{a_{i-1} + a_{i+1}}{2},$$

则称它是“低保数”。

- 题目进一步要求判断给定的 x 是否是 **最大的** 低保数。

我们需要找到在十进制下满足上述条件的所有低保数中的最大者，设为 M ，然后仅需判断 $x = M$ 。

二、构造最大低保数

要让数值尽可能大，第一位 a_1 尽量取 9；最后一位 a_n 也应取 9（因为尾位没有右邻居约束，可以直接最大化）。中间各位我们要在满足

$$2a_i < a_{i-1} + a_{i+1} \iff a_{i+1} > 2a_i - a_{i-1},$$

且 $0 \leq a_{i+1} \leq 9$ 的前提下，尽量让数的**位数**最长并且**数值**最大。

- 最长长度**：要使命题两侧延伸得尽可能远，中间数字要“下降得慢”。若取

$$a_{i+1} = \lceil 2a_i - a_{i-1} + \epsilon \rceil \quad (\epsilon > 0 \text{ 取最小})$$

达到严格不等式即可。这样的线性递推会让中间段先降到 0 再越过界限，从而确定最大长度。

- 最大值**：在已定最大长度 n 下，越左侧（更高权重）的数字应越大。对称地，最优解是一段**回文**：

$$a_1 = 9, a_2 = 6, a_3 = 4, a_4 = 3, \dots, a_{n-1} = 6, a_n = 9.$$

具体计算可递推得到

9, 6, 4, 3, 3, 4, 6, 9,

共 8 位, 再往右一位就会超出 $a_i \leq 9$ 的范围, 不再满足。

因此, 所有低保数中数值最大的那一个就是

96433469.

三、正确性证明

1. 它本身是低保数

验证中间各位:

$$\begin{aligned} 6 < \frac{9+4}{2} = 6.5, & \quad 4 < \frac{6+3}{2} = 4.5, & \quad 3 < \frac{4+3}{2} = 3.5, \\ 3 < \frac{3+4}{2} = 3.5, & \quad 4 < \frac{3+6}{2} = 4.5, & \quad 6 < \frac{4+9}{2} = 6.5. \end{aligned}$$

均满足严格不等式。

2. 不能有更大的低保数

- 若尝试首位 $a_1 < 9$, 数值小于 9 开头的所有数, 必然小于 96433469。
- 若尝试首位仍为 9 而尾位 < 9 , 则数字末尾权重大幅下降, 也无法超过尾位为 9 的 8 位数。
- 若延长到 9 位, 则从递推关系可算出第 9 位至少要大于 9, 违反 $a_i \leq 9$ 的限制。

综上, **96433469** 是唯一的、也是最大的低保数。

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    cin >> T;
    while (T--){
        ll x;
        cin >> x;
        // 只有 x == 96433469 时输出 YES, 其余全部 NO
        cout << (x == 96433469 ? "YES\n" : "NO\n");
    }
    return 0;
}
```

- 时间复杂度 $O(T)$ ，只做一次整数比较，适合 T 高达 10^6 的场景。
- 空间复杂度 $O(1)$ 。

B

来源: [The 2021 CCPC Weihai Onsite A. Goodbye, Ziyin!](#)

树

一、题意重述

给定一棵 **无根** 树 (n 个节点, $n - 1$ 条边), 我们要统计有多少个节点 r , 使得当以 r 作为**根**来看时, 整棵树成为一棵**二叉树**。也就是说, 对于这棵以 r 为根的有向树中的任意节点, 其子节点数 **最多为 2**。

二、关键观察

- 在无根树中, 每个节点的**无向度数** $\deg(v)$ 已知。
- 若以某节点 r 作为根, 则:
 - i. 对根节点 r 本身: 它的孩子数 = $\deg(r)$ 。
 - ii. 对非根节点 $v \neq r$: 它有一个“父亲”方向, 需要排除这一条边, 其孩子数 = $\deg(v) - 1$ 。

因此, 以 r 为根后要满足 “每个节点孩子数 ≤ 2 ”, 等价于:

$$\begin{cases} \deg(r) \leq 2, \\ \deg(v) - 1 \leq 2, \quad \forall v \neq r \iff \deg(v) \leq 3, \quad \forall v \neq r. \end{cases}$$

由此我们只需对原树的度数分布进行统计:

1. 若存在某个节点 v 使得 $\deg(v) > 3$, 那么**无论选谁做根**, 当遍历到这个度数超过 3 的节点时, 都会出现孩子数 > 2 , 因此答案为 0。
2. 否则, 最大度数 Δ 满足 $\Delta \leq 3$:
 - 如果 $\Delta \leq 2$, 则**所有**节点都满足根节点度 ≤ 2 , 且所有非根度 $\leq 2 \leq 3$, 故每个节点都能作为根: 答案为 n 。
 - 如果 $\Delta = 3$, 则只能选择度 ≤ 2 的节点做根, 答案为 “度数 ≤ 2 的节点个数”。

三、算法流程

对于每组测试:

1. 读入 n 和 $n - 1$ 条边, 统计每个节点的度数数组 `deg[1..n]`。
2. 扫一遍 `deg` :
 - 记录最大度 `maxd = max(deg[i])`。
 - 统计 `cnt_le2 = |\{i : deg[i] \leq 2\}|`。
3. 根据 `maxd` 分三种情况输出:
 - `maxd > 3` $\rightarrow$ `0`
 - `maxd \leq 2` $\rightarrow$ `n`
 - `maxd == 3` $\rightarrow$ `cnt_le2`

整条流程时间复杂度 $O(n)$, 符合所有 T 组、 n 累计至 10^7 左右的要求。

四、正确性证明

1. **必要性**
 - 若某节点 v 度 $\deg(v) > 3$, 无论如何选择根, 在把树看作有根树时, 都有一次访问到 v 时其孩子数 $\geq \deg(v) - 1 > 2$ (若 v 不是根) 或 $\deg(v) > 2$ (若 v 是根), 都不符合二叉树定义。
2. **充分性**
 - 当所有节点度 ≤ 3 时, 若选某节点 r 作为根:
 - 根的孩子数 $= \deg(r) \leq 2$ 当且仅当 $\deg(r) \leq 2$;
 - 其余任意节点 $v \neq r$ 孩子数 $= \deg(v) - 1 \leq 3 - 1 = 2$.
 - 因此只需保证 $\deg(r) \leq 2$ 即可, 正好与上面算法选取 “度 ≤ 2 的节点” 相符。

```

#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    cin >> T;
    while (T--){
        int n;
        cin >> n;
        vector<int> deg(n+1, 0);
        for (int i = 1; i < n; i++){
            int u, v;
            cin >> u >> v;
            deg[u]++;
            deg[v]++;
        }

        int maxd = 0, cnt_le2 = 0;
        for (int i = 1; i <= n; i++){
            maxd = max(maxd, deg[i]);
            if (deg[i] <= 2)
                cnt_le2++;
        }

        if (maxd > 3) {
            cout << 0 << "\n";
        }
        else if (maxd <= 2) {
            cout << n << "\n";
        }
        else { // maxd == 3
            cout << cnt_le2 << "\n";
        }
    }
    return 0;
}

```

- **时间复杂度：**每组 $O(n)$ 读取与扫描，累计至 $O(\sum n)$ 。
- **空间复杂度：** $O(n)$ 存度数。

C

来源: [The 2021 ICPC Asia Shanghai Regional Programming Contest E. Strange Integers](#)

排序/贪心

一、题意重述

给定长度为 n 的整数序列 $\{A_i\}$ 和一个非负整数 k , 从序列中选出若干个元素, 使得任意两选中元素的差的绝对值至少为 k , 求最多能选多少个。

等价地, 在数轴上放置点, 要求任意两点距离至少为 k , 求最多能放多少个点。

二、贪心算法

1. **预处理**: 先对数组 A 进行升序排序。
2. **贪心选点**: 维护上一次选中位置的下标 `last` (初始为 “无”), 从小到大扫描排序后数组:
 - 若当前元素 $A[i]$ 与上一次选中元素 $A[\text{last}]$ 的差 $\geq k$, 则选中它, 并更新 `last = i`, 计数器 `ans++`。
 - 否则跳过它, 继续扫描。

最终 `ans` 即为最多可选的元素个数。

该过程时间复杂度 $O(n \log n)$ (排序) 加 $O(n)$ (扫描), 空间 $O(n)$ 。

三、正确性证明

典型的 “最远点覆盖” 或 “活动安排” 风格:

- **局部最优**: 每次在可选范围内取最小的那个点 (即当前最小的 $A[i]$) 作为新选点, 能为后续保留最大的剩余空间, 从而不会减少后续可选的可能。
- **交换论证**: 任意最优解若在某一步没有选取当前最小可行点, 可以将其替换为更小的这个可行点, 替换后不影响与其他已选点的间距, 且剩余点的可用范围更宽, 得到同样大小的方案。因此贪心选点不劣于任何最优解。

由此贪心算法能得到最优解。

```

#include <bits/stdc++.h>
using namespace std;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    cin >> T;
    while (T--){
        int n;
        long long k;
        cin >> n >> k;
        vector<long long> A(n);
        for (int i = 0; i < n; i++){
            cin >> A[i];
        }
        sort(A.begin(), A.end());
        int ans = 0;
        long long last = LLONG_MIN; // 上一次选中的值
        for (int i = 0; i < n; i++){
            if (ans == 0 || A[i] - last >= k){
                last = A[i];
                ans++;
            }
        }
        cout << ans << "\n";
    }
    return 0;
}

```

- 排序: $O(n \log n)$
- 一次扫描: $O(n)$
- 总复杂度: $O(n \log n)$, 满足 $n \leq 10^4$ 时足够高效。

D

来源: [The 2021 CCPC Guilin Onsite \(XXII Open Cup, Grand Prix of EDG\) I. PTSD](#)

贪心

一、题意重述

- 共有 n 名士兵, 编号即为他们的“力量值” $1, 2, \dots, n$ 。
- 一个士兵如果患有 PTSD (由字符串 s 指示), 并且在其所在的小组中恰好有**且只有**一名比他力量值更大的士兵, 他就会“感到不安”。

- 我们可以任意地把 n 名士兵分成若干组，每人恰属于一个组，组大小可以是 1，也可以更大。
- 目标是：**让感到不安的 PTSD 士兵的力量值之和最大。**

等价于：我们要挑选一些 PTSD 士兵，让他们在自己的组里恰好是第二强；每个这样的 PTSD 选手都要“绑定”一个比他强（力量更大）的士兵；且任何士兵只能参与一次分组（即对应匹配）。

二、贪心匹配思路

1. 只考虑 PTSD 士兵。

因为只有 PTSD 士兵的“第二强”身份才会产生得分；普通士兵当做“资源”或“强者”使用即可。

2. 转化为“匹配”问题。

- 每个 PTSD 士兵 i （力量值为 i ）如果要得分，必须配到一个编号 $j > i$ 的任意士兵，使得组为 $\{i, j\}$ （或更大，也不影响第二强身份）。
- 当士兵 j 被用作“强者”后，就不能再用来让别的 PTSD 士兵得分（每人只能属于一个组）。

3. 贪心策略：

- 我们优先考虑力量值大的 PTSD 士兵去“抢”更大的配对位置，目的是保留更多更大的 PTSD 士兵去后面匹配，使其也能得分。
- 实现上，从右向左扫描字符串：每遇到一个 PTSD “1” 位置 i ，就在它左边找最近的、还未被“消耗”的 PTSD 士兵（位置 $j < i$ 且也是 ‘1’）来配对。配对后，让左边那个 j 成为第二强，贡献得分 j ；并把两者都标记为“已用”，继续向左。
- 这样，每次都在当前位置 i 处“消耗”一个未用的左侧 PTSD 士兵，且选取的是距离最近的那个，保证被“二号”选手的力量 j 尽可能大，从而总和最大。

（**注：虽然直觉上看似用 PTSD 配 PTSD，实际上普通士兵也是可用的“强者”资源，但用 PTSD 做“强者”不会浪费对答案的潜力，因为普通士兵与 PTSD 士兵都可以当“强者”；而将 PTSD 士兵留作“强者”也不会减少总得分——反而，匹配 PTSD 对 PTSD 能让两个 PTSD 中更小的那个得分，优于只单用普通士兵让更小 PTSD 得分的情况。）

三、正确性证明

1. 贪心合法性：

- 若某最优方案中，右侧的 PTSD 士兵 i 没拿到与它最近的左侧 PTSD 士兵 j （而是去配了更远的 $j' < j$ ），我们可以交换：让 i 去配 j 、原来配给 i 的 j' 去配给那个本来配 j 的更左边的，再保持所有人都是第二强。
- 交换后被选为第二强的 PTSD 力量值更大（因为 $j > j'$ ），总和不减，方案仍然可行。因此贪心“尽量用最近的左侧 PTSD”并不会比最优差。

2. 无遗漏最优：

- 我们从右向左每个 PTSD 都尝试去“消耗”一个左侧 PTSD。如果某个 PTSD i 真有左侧未用 PTSD，就必然在某个最优方案中有配对上的；若真的没有，则它也得不到配对。
- 因此，贪心不会漏掉任何原本能配对的二号 PTSD，也不会错配。

```

#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    cin >> T;
    while (T--){
        int n;
        string s;
        cin >> n >> s;
        // j 指向“下一次可用的”左侧 PTSD 士兵的位置
        int j = n - 1;
        ll ans = 0;
        // 从右向左遍历
        for (int i = n - 1; i >= 0; --i) {
            if (s[i] == '1') {
                // 保证 j < i
                j = min(j, i - 1);
                // 跳过非 PTSD (字符 '0')，找到真正的“1”
                while (j >= 0 && s[j] == '0') {
                    --j;
                }
                if (j >= 0) {
                    // 匹配成功，让位置 j 的 PTSD 士兵做第二强，贡献力量 j+1
                    ans += (j + 1);
                    // 两个士兵都“用掉”：标记为 '0'
                    s[j] = '0';
                    s[i] = '0';
                    --j; // 下一次再往左找
                } else {
                    // 左侧已无 PTSD 可配
                    break;
                }
            }
        }
        cout << ans << "\n";
    }
    return 0;
}

```

- **时间复杂度**：每个位置最多访问常数次数，整体 $O(n)$ ；
- **空间复杂度**：使用原字符串+常数额外变量，即 $O(n)$ 。

E

来源: The 2021 ICPC Asia Nanjing Regional Contest (XXII Open Cup, Grand Prix of Nanjing) M.
Windblume Festival

思维

一、题意转换

- 有一个环上的数组 $a_1, \dots, a_n$ 。
- 每一步选一对相邻的元素 (x, y) (这里我们记作 y 紧随 x 之后, 若 $x = n$ 则 $y = 1$) , 将它们 “合并” :

$$a_x \leftarrow a_x - a_y, \quad \text{删除 } a_y, \quad \text{环缩为 } n - 1 \text{ 项。}$$

- 重复直到只剩一个数, 问通过最优的删除顺序, 最后剩下的数能有多大?

二、关键发现

1. 合并不变式

无论怎样合并, 最终剩下的数都可以写成下面两部分之和:

- **所有被删除元素的 “绝对值贡献”** : 每次删除 a_y 前, 若我们把它 “看作” 贡献了 $|a_y|$;
- **最后一次合并双生的 “有向差”** : 假设最后一步是在原环中的一对邻居 (i, j) (顺时针方向) 上执行的, 那么最终多出来一项

$$a_i - a_j \quad (\text{与上面那些 } |a| \text{ 无关})$$

2. **因此**, 不妨先把「所有被删除的元素」都算一遍绝对值, 再考虑最后剩下的这一对 (i, j) :

$$\text{最终值} = \sum_{k=1}^n |a_k| - (|a_i| + |a_j|) + (a_i - a_j).$$

其中 (i, j) 必须是环中相邻的。

3. 最优策略

- 先把环上所有元素的绝对值之和记作

$$S = \sum_{k=1}^n |a_k|.$$

- 枚举所有相邻对 $(i, i + 1)$ (下标循环) , 计算

$$S - (|a_i| + |a_{i+1}|) + (a_i - a_{i+1}),$$

- 取其中的最大值, 就是答案。

三、正确性证明概要

- **合并贡献拆分：**

每一步删除 y 都可以看作把 $|a_y|$ 加入“答案池”，并把差值 $a_x - a_y$ 继续留给“存活者” x 。一路递推，所有被删的节点都贡献了它们的绝对值，最后剩下的节点身上只剩下「它与被它最后合并的那个节点的差」。

- **只需关注“最后一次合并”：**

其它所有合并，你都可以视为“先把两个数都拆成绝对值再重构”，其结果总和不依赖于具体顺序，只和各绝对值之和有关。唯独最后一步，因为剩下的只有一对，无法再“拆和重构”，所以要保留那一步 $a_i - a_j$ 的有向差。

- **为什么选最大的一对：**

枚举所有可能做最后一步的相邻对，取能让上述表达式最大的那一对，即能让最终值最大。

```

#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    cin >> T;
    while (T--){
        int n;
        cin >> n;
        vector<ll> a(n);
        ll S = 0;
        for (int i = 0; i < n; i++){
            cin >> a[i];
            S += llabs(a[i]);
        }
        // 若只有一个元素，直接输出它本身
        if (n == 1) {
            cout << a[0] << "\n";
            continue;
        }

        // 枚举所有相邻对 (i, i+1 mod n)
        ll ans = LLONG_MIN;
        for (int i = 0; i < n; i++){
            int j = (i + 1) % n;
            ll cur = S
                - (llabs(a[i]) + llabs(a[j]))
                + (a[i] - a[j]);
            ans = max(ans, cur);
        }
        cout << ans << "\n";
    }
    return 0;
}

```

F

枚举/思维

来源: [The 2021 ICPC Asia Nanjing Regional Contest \(XXII Open Cup, Grand Prix of Nanjing\)](#) C. Klee in Solitary Confinement

一、题意概述

给定长度为 n 的整数序列 $a_1, \dots, a_n$ 以及一个整数 k ，我们可以**任选一次**（也可不做）将区间 $[l, r]$ 中的每个元素都加上 k ，问在最优选择下，最终序列的众数的出现次数最大是多少？

- **众数的出现次数**：指某个值在序列中出现的次数，上述过程后我们只关心最频繁的数出现了多少次。

二、思路与算法

1. 分析可执行的变化

对某个区间 $[l, r]$ 加上 k 相当于：

- 区间外的元素保持原值；
- 区间内的元素从原来值 x 变为 $x + k$ 。

等价于：我们要统计经过一次“整体把一段原值 x 变成 $x + k$ ”后，哪个数值出现得最频繁，以及它的频次。

2. 原始统计与事件序列

记原序列中，每个值 x 的出现次数为 $\text{cnt}[x]$ 。

若不做任何操作，答案至少是

$$\max_x \text{cnt}[x].$$

若做一次操作，将一次性把所有在 $[l, r]$ 上出现的值 x 变成 $x + k$ ：

- 对于原本等于 x 而在区间内的那些，原先为 $\text{cnt}[x]$ ，其中若有 t 个落在区间内，就会使得：
 - 原值 x 的剩余次数变为 $\text{cnt}[x] - t$ ；
 - 值 $x + k$ 的次数增加 t 。

我们希望通过选择区间 $[l, r]$ ，使得对于某个“目标值” v （可能是原值，或是某个值加 k 后的新值），它的出现次数最大。

3. 将问题转为区间加减的“差分扫描”

对固定目标值 v ，其最终出现次数为

$$\underbrace{\text{cnt}[v]}_{\text{原来就有的}} + \underbrace{|\{i \in [l, r] : a_i = v - k\}|}_{\text{把 } v-k \text{ 变到 } v} - \underbrace{|\{i \in [l, r] : a_i = v\}|}_{\text{把 } v \text{ 变走}}.$$

把上式改写为

$$\text{cnt}[v] + \left[\#(v - k) \text{ in } [l, r] - \#(v) \text{ in } [l, r] \right].$$

其中第二项取最大，区间可以任意选。

- 设我们对原序列做一次「扫描」，按原序列位置 i 从 1 到 n ：
 - 当 $a_i = v - k$ 时，相当于“在当前位置做 +1”；
 - 当 $a_i = v$ 时，相当于“在当前位置做 -1”；
 - 其它值时做 0。

那么想要这段增量 $[\#(v - k) - \#(v)]$ 在某个连续区间上最大，就等价于对这个 “+1/-1/0” 数组求**最大子段和**。

4. 总体流程

1. 统计原频次

对每个出现过的值 x ，建立哈希或映射： $cnt[x] \leftarrow$ 出现的总次数。

2. 枚举所有可能的“目标值”

目标值其实只可能出现在原值集合和“原值加 k ”的集合里。对每个曾出现过的 x ：

- 如果我们考虑目标 $v = x$ （即操作后希望它最频繁），就要看把所有原来等于 $x - k$ 的元素搬到它上面，扣掉把原来等于 x 的搬走的效果。
- 如果 $k = 0$ ，那么“搬来搬去”相互抵消，无需再做额外的区间；最优答案就是原有众数频次。

3. 对每个候选值 v

- 构造一个长度为 n 的序列 d_i ,

$$d_i = \begin{cases} +1, & a_i = v - k, \\ -1, & a_i = v, \\ 0, & \text{otherwise,} \end{cases}$$

- 在 d 上用**Kadane 算法**求最大子段和 Δ 。
- 它代表在某个最优区间 $[l, r]$ 上能给目标 v 带来的净增量。
- 那么对于 v ，操作后最大可能出现次数就是

$$cnt[v] + \max(0, \Delta).$$

4. 答案

取所有 v 上述值的最大者。

三、复杂度分析

- 序列长度 $n \leq 10^4$ ，值域大约 $\pm 10^6$ 。
- 扫描输入一次 $O(n)$ 建原频次与收集“出现过的值”列表。
- 对于每个**出现过的** x （至多 n 个），构造 d 并 Kadane 一次，都是 $O(n)$ 。
- 因此总时间为 $O(n^2)$ 最坏。但由于实际“不同值”往往远少于 n ，均摊下来足够通过 $n \leq 10^4$ 的约 10^8 量级。

```

#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    cin >> T;
    while (T--){
        int n;
        ll k;
        cin >> n >> k;
        vector<ll> a(n);
        for (int i = 0; i < n; i++) {
            cin >> a[i];
        }

        // 1) 统计原频次 & 收集出现过的值
        unordered_map<ll,int> cnt;
        cnt.reserve(n*2);
        vector<ll> vals;
        for (auto x : a) {
            if (++cnt[x] == 1) {
                vals.push_back(x);
            }
        }

        int answer = 0;
        // 不做任何操作时的最大众数频次
        for (auto &p : cnt) {
            answer = max(answer, p.second);
        }
        // 若 k == 0, 不会改变众数分布
        if (k == 0) {
            cout << answer << "\n";
            continue;
        }

        // 2) 枚举每个可能的“目标值” v = x
        for (ll v : vals) {
            // 构造增量数组 d[i]
            int best = 0, cur = 0;
            for (int i = 0; i < n; i++) {
                int delta = 0;
                if (a[i] == v - k) delta = +1;
            }
        }
    }
}

```

```

        else if (a[i] == v) delta = -1;
        // Kadane
        cur = max(0, cur + delta);
        best = max(best, cur);
    }
    // 操作后 v 的最大可能出现次数
    answer = max(answer, cnt[v] + best);
}

cout << answer << "\n";
}
return 0;
}

```

• 步骤详解

- i. 用 `cnt` 哈希记录每个值出现次数，同时 `vals` 列出所有不同值。
- ii. 初始 `answer` 设为原众数频次。
- iii. 若 $k = 0$ 跳过枚举，直接输出。
- iv. 否则对每个候选值 v ，运行一次 Kadane 找到对它的最佳区间增益 `best`，并更新 `answer = max(answer, cnt[v] + best)`。

• 正确性

- 对每个 v 考虑，区间中每多一个值 $v - k$ 就能把它“变成” $v (+1)$ ；每出现一个原值 v 就损失一个 (-1) ；Kadane 保证选到净增量最大的一段。
- 最后再与“不做区间操作”时的原始众数次数做比较，取更优。

G

来源：The 2021 ICPC Asia Shanghai Regional Programming Contest G. Edge Groups

树形DP/计数递推

一、问题重述

一棵有 n 个顶点、 $n - 1$ 条边的树 (n 为奇数)，要求将这 $n - 1$ 条边分成 $\frac{n-1}{2}$ 组，每组恰有两条边，且同组的两条边必须共用一个端点。求这样的分组方案数，答案对 998244353 取模。

等价地：将树的边两两配对，每对配在同一个公共“中心”顶点上。要求每条边恰好配到一个组中。

二、转化为树形 DP

我们把树任意根到节点 1。对每个节点 x ，考虑它与父亲边（如果有）以及它和每个孩子的边。一共是 d_x 条“待配边”（若 x 不是根，则 $d_x = \deg(x)$ ，若是根则也是 $\deg(x)$ ）。

- 我们需要在这些边上做“完美配对”：每两条配在一起，配对只能在这些边内部，但允许其中一条是“父边”，另一条是“子边”，也允许两条都来自子边，即两条子边相互配。

定义状态：

- $dp0[x]$ ：在子树 x 内，**不使用**“父边”与任何子边配对的合法方案数。换言之，父边要么留到上层去配（若 x 不是根），要么（若 x 是根）就不配。
- $dp1[x]$ ：在子树 x 内，**恰好**使用“父边”与某一条子边配在一起的合法方案数。

令 $k =$ 节点 x 的子节点个数，则节点 x 实际上要处理的子边数也是 k 。

1. 子树内递推

先分别计算各个孩子 $c_1, \dots, c_k$ 的 $dp0, dp1$ ，再合成 x 的状态。

• 子树贡献的乘积：

每个子节点 c_i 的“父边”就是那条 (x, c_i) 边。

- 如果在 x 这一层，没有用 (x, c_i) 去配父边（无论是当父边配给它，还是它配给别的子边），那么在子树里它处于“↑父边未被配”的状态，即乘入 $dp0[c_i]$ 。
- 如果在 x 这一层， (x, c_i) 这条边被配给了 x 的父边（仅可能用于 $dp1[x]$ 情况下的“一条子边与父边配”），那么该子节点处于“↑父边已被配”的状态，即用 $dp1[c_i]$ 。

• 配对子边或父边的计数：

节点 x 处一共有

- k 条子边；
- 以及 1 条父边（若 x 是根，则父边实际上不存在，我们把“父边”当作“留空”且只使用 $dp0[x]$ ）。

记 $g[m]$ 为“把 m 条边成对配完，且两两配对”——即“ m 条物品的完美匹配数”。注意若 m 为奇数，无完美匹配， $g[m] = 0$ ；若 m 为偶数，

$$g[m] = (m-1) \times (m-3) \times \dots \times 1 \pmod{998244353}.$$

我们预先用一维数组 $\{g[0], g[1], \dots\}$ 线性填出所有至多 $\max \deg$ 的值。

- 对于 $dp0[x]$ ：

不使用父边配对子边，也就是说只要把 k 条子边两两在子边之间配对，方案数为

$$dp0[x] = g[k] \times \prod_{i=1}^k dp0[c_i].$$

- 对于 $dp1[x]$ （父边必配一个子边）：

- 先从 k 条子边里选择一条与父边配对；
- 剩下的 $k-1$ 条子边要在子边中两两配对——这有 $g[k-1]$ 种；
- 选中与父边配对的那条子边对应的子节点 c_j 要乘入 $dp1[c_j]$ ，其他没有与父边配的子边仍乘 $dp0[c_i]$

因此

$$dp1[x] = g[k-1] \times \sum_{j=1}^k \left(dp1[c_j] \times \prod_{i \neq j} dp0[c_i] \right).$$

为了高效计算对所有 j 的求和，我们可以先算

$$P = \prod_{i=1}^k \text{dp0}[c_i]$$

以及

$$S = \sum_{j=1}^k (\text{dp1}[c_j] \times \text{inv}(\text{dp0}[c_j]))$$

(这里假设 $\text{dp0}[c_j] \neq 0$ ，若为 0 则对应项自然为 0)，

则

$$\sum_{j=1}^k (\text{dp1}[c_j] \times \prod_{i \neq j} \text{dp0}[c_i]) = P \times S.$$

故

$$\text{dp1}[x] = g[k-1] \times P \times S.$$

2. 根节点的答案

根节点 $r = 1$ 没有父边，所以只能用 $\text{dp0}[1]$ 。最终答案即

$$\boxed{\text{dp0}[1] \bmod 998244353}.$$

三、正确性证明要点

1. 完美分组等价于配对

每条边必须恰好属于一个组，每组两条边，且共用一个顶点，等价于把每个顶点“看作”有若干条半边（子边 + 父边），要在它“拥有”的这些半边里做完美配对，且所有半边都要被配掉。分治到子树，若父边在本子树里不作配对，则留给上层；若作配对，则必与本层某子边配对。

2. 子树独立性

树的无环结构保证：对子树的配对选择只通过“子边是否用来与父边配对”这一二状态影响，从而可做形如上面的 DFS 递推。

四、复杂度与实现细节

• 预处理：

- 读入整棵树，建邻接表。
- 构造 $g[0..n]$ ： $g[0] = 1$, $g[1] = 0$ ，对偶数 $i \geq 2$ ，令

$$g[i] = g[i-2] \times (i-1) \bmod 998244353.$$

- **DFS 一次**，在每个节点做 $O(\deg(x))$ 的乘积和加和，总计 $O(n)$ 。

- 空间 $O(n)$ 。

总体对于所有测试用例 $\sum n \leq 10^6$ 也能在一两秒内跑完。

```

#include <bits/stdc++.h>
using namespace std;
using ull = unsigned long long int;
using ll = long long int;
using ld = long double;
using PII=pair<ll,ll>;
const ld ESP = 1e-10;
const ld PI = acosl(-1);
const int N=1e6+10;
const int M=1e5+10;
const int inf = 0x3f3f3f3f;
const int mod = 998244353;
int d[M];
vector<vector<int>> e(M);
vector<ll> dp(M,1),g(M,0);
void dfs(int x,int f){
    for(auto y:e[x]){
        if(y==f)continue;
        else{
            dfs(y,x);
            dp[x]=(dp[x]*dp[y])%mod;
        }
    }
    if(d[x]>=2){
        if(d[x]%2==0){
            d[f]--;
            dp[x]=(dp[x]*g[d[x]])%mod;
        }
        else{
            dp[x]=(dp[x]*g[d[x]-1])%mod;
        }
    }
}
void solve(){
    int n;
    cin>>n;
    for(int i=1;i<n;i++){
        int u,v;
        cin>>u>>v;
        e[v].push_back(u);
        e[u].push_back(v);
        d[v]++;
        d[u]++;
    }
    dfs(1,0);
    cout<<dp[1]%mod<<'\n';
    for(int i=1;i<=n;i++){

```

```

        e[i].clear();
        dp[i]=1;
        d[i]=0;
    }
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    g[2]=1;
    for(ll i=4;i<=10000;i+=2){
        g[i]=(g[i-2]*(i-1))%mod;
    }
    int _;
    _=1;
    cin>>_;
    while(_--){
        solve();
    }
    return 0;
}

```

```

#include <bits/stdc++.h>
using namespace std;

const int mod = 998244353;

int fac[200005];
int ifac[200005];

int C(int n, int m) {
    if (m < 0 || m > n) return 0;
    return 111 * fac[n] * ifac[m] % mod * ifac[n - m] % mod;
}

int pw(int b, int n) {
    int a = 1;
    while (n) {
        if (n & 1) a = 111 * a * b % mod;
        b = 111 * b * b % mod;
        n >>= 1;
    }
    return a;
}

int inv(int x) {
    return pw(x, mod - 2);
}

vector<int> g[200005];

pair<int, int> dfs(int now, int pa) {
    int mul = 1;
    int cntfre = 0;
    for (int i : g[now]) if (i != pa) {
        auto [res, fre] = dfs(i, now);
        mul = 111 * mul * res % mod;
        cntfre += fre;
    }

    int add = 1;
    int grp = 0;
    while (cntfre >= 2) {
        add = 111 * C(cntfre, 2) * add % mod;
        cntfre -= 2;
        ++grp;
    }

    add = 111 * add * ifac[grp] % mod;
}

```

```

        return pair{(int)(1ll * mul * add % mod), 1 - cntfre};
    }

int main() {
    // 预处理阶乘和逆元
    fac[0] = 1;
    for (int i = 1; i <= 200000; ++i) fac[i] = 1ll * fac[i - 1] * i % mod;
    ifac[200000] = inv(fac[200000]);
    for (int i = 199999; i >= 0; --i) ifac[i] = 1ll * ifac[i + 1] * (i + 1) % mod;

    ios_base::sync_with_stdio(0); cin.tie(0);

    int T; cin >> T;
    while (T--) {
        int n; cin >> n;

        // 清空图
        for (int i = 1; i <= n; ++i) g[i].clear();

        for (int i = 1; i < n; ++i) {
            int u, v;
            cin >> u >> v;
            g[u].push_back(v);
            g[v].push_back(u);
        }

        auto [dp, fre] = dfs(1, 0);
        assert(fre == 1);
        cout << dp << '\n';
    }
}

```

- **核心思路**：把每个节点“拥有”的与父边 + 子边合计的半边，在本层做配对，配对方式枚举后利用子树 DP 乘法合并。
- **时间复杂度**： $O(n)$ / 每个测试用例；所有测试用例累计 $\sum n \leq 10^6$ 。
- **空间复杂度**： $O(n)$ 。

H

[Codeforces Round 988 \(Div. 3\) F - Ardent Flames](#)

二分/前缀和

一、题意重述

- 线上共有 n 只龙，编号为 $1, 2, \dots, n$ ，它们位于横坐标等于编号的位置，第 i 只龙的生命值为 h_i 。

- 你事先选定一个固定的攻击中心位置 $p \in \mathbb{Z}$ 。
- 每次发动一次强度为 m 的攻击，第 i 只龙受到的伤害为

$$\max(0, m - |p - i|).$$

- 一只龙的生命值降到 0 或以下即算被击败。你要最少打出多少次攻击，才能**至少**击败 k 只龙？如果无论怎么选 p 都不能同时把 k 只龙打死，答案是 -1 。

二、关键思路

1. 二分答案

设我们猜“最少攻击次数”为 T 。我们需要判断：是否存在一个攻击中心 p ，使得在 T 次攻击后能击败至少 k 条龙？

- 若存在，则可以尝试更小的 T ；
- 否则增大 T 。

由于攻击次数越多，能击败的龙数不会减少，判定条件对 T 是**单调**的，适合二分。

2. 对于固定 T 如何检测？

- 第 i 只龙在一次攻击中对其的伤害为

$$d_i(p) = \max\{0, m - |p - i|\}.$$

在 T 次攻击后总伤害为 $T d_i(p)$ 。要把它打死，需要

$$T d_i(p) \geq h_i \iff d_i(p) \geq \left\lceil \frac{h_i}{T} \right\rceil.$$

- 写 $\Delta_i = \left\lceil h_i/T \right\rceil$ 。
若 $\Delta_i > m$ ，则无论 p 放在哪儿， $d_i(p) \leq m < \Delta_i$ ，**永远打不死**，可以直接不考虑这条龙。
否则我们要求

$$m - |p - i| \geq \Delta_i \iff |p - i| \leq m - \Delta_i.$$

令 $R_i = m - \Delta_i \geq 0$ ，则这条龙对中心 p 可被击败的“可行区间”是

$$p \in [i - R_i, i + R_i].$$

- 问题转化为：在所有这些区间（共有至多 n 条）中，是否存在一个位置 p ，被**至少** k 条区间覆盖？

3. 区间覆盖的快速判定

- 把每个区间 $[L_i, R_i]$ 分解为两个事件：

$$(L_i, +1), \quad (R_i + 1, -1).$$

- 按“横坐标”对所有事件排序，扫描累加“活动计数”——即当前有多少个区间在覆盖这个位置。
- 若在任意位置上扫描到的活动计数 $\geq k$ ，则说明可选该 p ，判定为“ T 次攻击可以击败至少 k 条龙”。

4. 综合复杂度

- 二分攻击次数 T 的范围从 1 到一个上界（例如 $\max_i h_i$ ，或 10^{12} ），需要约 $O(\log H)$ 步；

- 每步里，我们对 n 条龙计算 Δ_i 并收集最多 $2n$ 个事件，然后排序并线性扫描，都是 $O(n \log n)$ 。
- 因为 $\sum n \leq 10^5$, $\log H \approx 40$, 总时间足够在几百万量级操作内完成。

三、正确性证明

1. 二分单调

若对于某 T 存在能打死 k 条龙的中心 p , 那么对任意更大的 $T' > T$, $\Delta'_i = \lceil h_i/T' \rceil \leq \lceil h_i/T \rceil = \Delta_i$, 对应的可行区间长度

$$R'_i = m - \Delta'_i \geq m - \Delta_i = R_i.$$

区间增大，覆盖数不减，因此 T' 也可行。单调性成立。

2. 区间模型等价性

对某龙 i , 中心 p 能在 T 次内击败它，当且仅当

$$|p - i| \leq m - \left\lceil \frac{h_i}{T} \right\rceil.$$

这正好是一个整数区间。要同时击败至少 k 只龙，相当于找到一个 p 落在**至少** k 条这样的区间的交集中——恰好是区间重叠计数问题。

```

#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int t;
    cin >> t;
    while (t--){
        int n, m, k;
        cin >> n >> m >> k;
        vector<ll> h(n);
        vector<ll> x(n);
        for (int i = 0; i < n; i++) {
            cin >> h[i];
            x[i] = i + 1; // 横坐标就是编号
        }

        // 判定给定 T 次攻击是否可行
        auto can = [&](ll T)->bool {
            vector<pair<ll,int>> evs;
            evs.reserve(2*n);
            for (int i = 0; i < n; i++){
                // 最少每次需打出的伤害
                ll need = (h[i] + T - 1) / T;
                if (need > m)
                    continue; // 永远打不死, 跳过
                ll R = m - need; // |p - x[i]| <= R
                ll Lpos = x[i] - R;
                ll Rpos = x[i] + R;
                evs.emplace_back(Lpos, +1);
                evs.emplace_back(Rpos + 1, -1);
            }
            if ((int)evs.size() < 2*k) {
                // 区间总数 < k, 最多只能覆盖这么多
                return false;
            }
            sort(evs.begin(), evs.end());
            int cover = 0;
            for (int i = 0; i < (int)evs.size(); i++){
                cover += evs[i].second;
                // 如果下一个事件与当前横坐标相同, 则先累加完再判断
                if (i+1 < (int)evs.size() && evs[i+1].first == evs[i].first)
                    continue;
                if (cover >= k)

```

```

        return true;
    }
    return false;
};

// 二分最小 T
ll lo = 1, hi = 1;
// hi 可以初始化为 max(h_i), 因为每次攻击至多 m, 若 T > max(h)/1, safe upper bound
for (int i = 0; i < n; i++)
    hi = max(hi, (h[i] + m - 1) / m);
ll ans = -1;
while (lo <= hi){
    ll mid = (lo + hi) >> 1;
    if (can(mid)){
        ans = mid;
        hi = mid - 1;
    } else {
        lo = mid + 1;
    }
}
cout << ans << "\n";
}
return 0;
}

```

I

来源: [Codeforces Round 776 \(Div. 3\) B. DIV + MOD](#)

数学

一、题目回顾

定义

$$f_a(x) = \left\lfloor \frac{x}{a} \right\rfloor + (x \bmod a).$$

给定参数 a 及区间 $[l, r]$, 要求找出 $\max_{x \in [l, r]} f_a(x)$ 。

二、关键观察

1. 写成商与余数的形式

令

$$q = \left\lfloor \frac{x}{a} \right\rfloor, \quad r' = x \bmod a, \quad \text{则 } x = qa + r', \quad 0 \leq r' < a.$$

此时

$$f_a(x) = q + r'.$$

2. 在同一个商 q 下如何最大

若两个 x 落在同一个整除区间 $[qa, qa + (a - 1)]$, 它们的商相同, 都等于 q , 而余数越大 r' 越大,

$f_a(x) = q + r'$ 也越大。所以在商 q 区间内, $f_a(x)$ 于区间右端点 $x = qa + (a - 1)$ 处取得最大值, 值为

$$f_{\max | q} = q + (a - 1).$$

3. 跨商的比较

当 x 跨到下一个区间 $[(q + 1)a, (q + 1)a + (a - 1)]$, 商 q 加 1, 但此时余数从 0 开始。

- 右端 $x = (q + 1)a + (a - 1)$ 处有

$$f_a = (q + 1) + (a - 1) = q + a.$$

- 而在前一商区最优点 $x = qa + (a - 1)$ 处是

$$f_a = q + (a - 1).$$

相差正好 1, 所以跨到下一个区间通常能够**提升 1**。

4. 结合区间 $[l, r]$

令

$$Q_l = \lfloor l/a \rfloor, \quad Q_r = \lfloor r/a \rfloor.$$

- 若 $Q_l = Q_r$, 即 $[l, r]$ 完全处于同一个区间的商组里, 那么 x 的商不变, 必然在区间**右端** $x = r$ 处取到最大余数, 从而

$$\max_{x \in [l, r]} f_a(x) = f_a(r) = \left\lfloor \frac{r}{a} \right\rfloor + (r \bmod a).$$

- 否则, $[l, r]$ 至少跨了一个商区。此时有两个**候选最优点**:

- 区间末尾 r 处:

$$f_a(r) = Q_r + (r \bmod a).$$

- 紧挨 r 商区开始的那条**上一区间的最大点**

$$x_2 = Q_r \cdot a - 1$$

但要保证 $x_2 \geq l$ (即它确实在 $[l, r]$ 内)。此时

$$f_a(x_2) = \underbrace{\left\lfloor \frac{x_2}{a} \right\rfloor}_{=Q_r-1} + (x_2 \bmod a) = (Q_r - 1) + (a - 1) = Q_r + a - 2.$$

- 上述两者中取最大值即可。

三、总结执行步骤

1. 读入 l, r, a , 计算商
 $Q_l = \lfloor l/a \rfloor$,
 $Q_r = \lfloor r/a \rfloor$.
2. 若 $Q_l = Q_r$: 直接返回

$$Q_r + (r \bmod a).$$

3. 否则:
 - i. 令候选一: $v_1 = Q_r + (r \bmod a)$.
 - ii. 令 $x_2 = Q_r \times a - 1$.
如果 $x_2 < l$, 则不能取此点, 忽略; 否则
候选二: $v_2 = Q_r + a - 2$.
 - iii. 答案为 $\max(v_1, v_2)$.

算法时间 $O(1)$ / 测试用例, 极为高效。

```

#include <bits/stdc++.h>
using namespace std;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    cin >> T;
    while (T--){
        long long l, r, a;
        cin >> l >> r >> a;
        long long ql = l / a;
        long long qr = r / a;
        // 情况 1: 完全在同一商区
        if (ql == qr) {
            cout << qr * a << "\n";
        } else {
            // 候选1: 区间末尾 r
            long long v1 = qr * a;
            // 候选2: 前一区间最优解 x2 = (qr-1)*a - 1
            long long x2 = (qr - 1) * a - 1;
            long long v2 = LLONG_MIN;
            if (x2 >= l) {
                // f(x2) = (qr-1) * a + (a-1) = qr * a - 1
                v2 = x2 + a - 1;
            }
            cout << max(v1, v2) << "\n";
        }
    }
    return 0;
}

```

J

P4568 [JLOI2011] 飞行路线

分层图最短路

一、题意重述

- 有 n 座城市，编号 0 到 $n - 1$ ，有 m 条双向航线，第 i 条航线连接城市 a 和 b ，票价为 c 。
- 从起点 s 到终点 t 之间，可以转机；并且在最多 k 条航线上可以**免费**乘坐（即那条边权视为 0 ）。
- 问在最优选择免费边与付费边的策略下，从 s 到 t 的**最小总花费**。

二、建模与算法

1. 分层图建模

我们将原图“展开”成 $k + 1$ 层，每层都是原有 n 个节点的一个拷贝。节点 (u, j) 表示“当前在城市 u ，已经用了 j 次免费机会”。

- **付费边**

对于每一条原图中的边 $(u - v, c)$ ，在每一层 $j = 0, 1, \dots, k$ 都有一条“跨节点不换层”的有向边

$$(u, j) \xrightarrow{c} (v, j), \quad (v, j) \xrightarrow{c} (u, j).$$

表示在第 j 次免费机会之后，若在这一条边上正常付费，就仍然保持已用免费次数不变。

- **免费边**

若当前已用免费次数为 $j < k$ ，则可以把某条航线当做免费乘坐，相当于在图中有条“跨到下一层”的 0 权重边：

$$(u, j) \xrightarrow{0} (v, j + 1), \quad (v, j) \xrightarrow{0} (u, j + 1).$$

这样，从 $(s, 0)$ 出发，到达任何 (t, j) 的最短路长度即是“使用恰好 j 次免费，最少付费”下的最小花费。取所有 $j = 0 \dots k$ 中最小者即为答案。

2. 最短路求解

- 节点总数： $(k + 1) \times n$ ，最多 11×10^4 （因为 $k \leq 10, n \leq 10^4$ ）。
- 边数：
 - 付费边： $(k + 1) \times 2m$ 。
 - 免费边： $k \times 2m$ 。总计 $O(km) \approx 10^6$ 级别，Dijkstra 可接受。

我们对这张图运行**Dijkstra**（优先队列实现），从源点 $(s, 0)$ 出发，最终记录所有 $(t, 0), (t, 1), \dots, (t, k)$ 的最短距离，取最小即为答案。

三、正确性证明

1. 状态含义完备

节点 (u, j) 完全表示：当前抵达城市 u ，并且已经用了 j 次免费机会。

2. 边的含义

- 普通付费边：不改变免费机会计数；
- 免费边：消耗一次免费机会 $(j \rightarrow j + 1)$ ，且不增加花费。

3. 最优子结构

若某条最短路在使用了 j 次免费后到达了 (u, j) ，无论后续如何选择收费或免费，都不会影响此前的最小前缀花费。标准最短路最优子结构成立。

4. 取最小

由于最终可以使用 0 到 k 次免费的任意次数，答案就是在所有可能到达层 (t, j) 的最小值。

复杂度

- **节点数**: $(k + 1) \times n \leq 11 \times 10^4$ 。
- **边数**: $O(km) \leq 10 \times 5 \times 10^4 = 5 \times 10^5$ 条双向边。
- **Dijkstra 复杂度**: $O((N + E) \log N) \approx O(6 \times 10^5 \log 10^5)$, 在时间限制内能高效完成。

```

#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const ll INF = (ll)4e18;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m, k;
    cin >> n >> m >> k;
    int s, t;
    cin >> s >> t;

    // 建图: 分 (k+1) 层, 每层 n 个节点, 编号 (layer * n + u)
    int N = (k+1) * n;
    vector<vector<pair<int,int>>> G(N);

    auto idx = [&](int u, int layer){
        return layer * n + u;
    };

    for (int i = 0; i < m; i++){
        int u, v, c;
        cin >> u >> v >> c;
        // 对每一层都加付费边
        for (int layer = 0; layer <= k; layer++){
            int U = idx(u, layer), V = idx(v, layer);
            G[U].emplace_back(V, c);
            G[V].emplace_back(U, c);
        }
        // 对每一层 layer < k 加免费边到下一层
        for (int layer = 0; layer < k; layer++){
            int U = idx(u, layer), V = idx(v, layer);
            int U2 = idx(u, layer+1), V2 = idx(v, layer+1);
            G[U].emplace_back(V2, 0);
            G[V].emplace_back(U2, 0);
        }
    }

    // Dijkstra
    vector<ll> dist(N, INF);
    using pli = pair<ll,int>;
    priority_queue<pli, vector<pli>, greater<pli>> pq;

    dist[idx(s,0)] = 0;
    pq.emplace(0, idx(s,0));

```

```

while (!pq.empty()){
    auto [d,u] = pq.top();
    pq.pop();
    if (d > dist[u]) continue;
    for (auto &e : G[u]){
        int v = e.first, w = e.second;
        if (dist[v] > d + w){
            dist[v] = d + w;
            pq.emplace(dist[v], v);
        }
    }
}

// 在所有层 t,0 ... t,k 中取最小
ll ans = INF;
for (int layer = 0; layer <= k; layer++){
    ans = min(ans, dist[idx(t, layer)]);
}
cout << ans << "\n";
return 0;
}

```

K

来源: [Educational Codeforces Round 120 \(Rated for Div. 2\) A.Construct a Rectangle](#)

签到

一、题意回顾

有三根木棍，长度为 l_1, l_2, l_3 。我们要选一根把它恰好分成两段（正整数长度），与剩余两根一起共四根木棍，能正好组成一个矩形（包括正方形）——也就是“配成两对相等的边”。

二、可行的两种情况

将三根棒分成两段后，一共四条边，设矩形边长为 x 和 y ，则必须恰好有两条长度为 x 、两条长度为 y 。

我们从原始三根中选一根去“拆”，设被拆的那根原长为 L ，拆成两段后长度分别为 u 和 v ($u + v = L$ ，且 $u, v \geq 1$)。其余两根长度不变，记为 a 与 b 。因此四条长度为 multiset $\{u, v, a, b\}$ 。要能配成矩形的两对边，就必须是下面两种**互斥**场景之一：

1. 用拆出的两段作为一对，对应相等边

- 即 $u = v$ ，且同时剩下的两根 a 与 b 也相等：

$$u = v = \frac{L}{2}, \quad a = b.$$

- 要求 L 是偶数, 且 $a = b$ 。

2. 分别用拆出的两段各与一根原棒配对

- 即拆出的两段长度分别与 a 、 b 对应相等:

$$u = a, \quad v = b \quad (\text{或 } u = b, v = a), \quad u + v = L.$$

- 等价于:

$$a + b = L.$$

没有其它可能, 因为要配成“两对相等”, 四个数必须要么两两相同, 要么成两对横跨拆分。

三、综合判定

给定 (l_1, l_2, l_3) , 只需检查以下 6 种可能 (对称于三根木棍谁被拆):

1. 和等式型

- $l_1 + l_2 = l_3$
- $l_2 + l_3 = l_1$
- $l_3 + l_1 = l_2$

这三种情况对应“把第三根拆成长度正好等于前两根”的场景。

2. 平分型

- l_1 被拆: 若 l_1 偶数且 $l_2 = l_3$
- l_2 被拆: 若 l_2 偶数且 $l_3 = l_1$
- l_3 被拆: 若 l_3 偶数且 $l_1 = l_2$

这三种情况对应“把一根平分成两段相等, 用来配成正方形的一对边, 其它两根也相等”场景。

只要任意一条成立, 即可输出 **YES**; 否则输出 **NO**。

四、正确性证明

- 若存在其它拆法可以配成矩形, 则四根长的集合必有恰好两条最长相等、两条次长相等。拆分后得到的两段之和等于被拆原长, 故可能情况必落在上面两类:
 - 平分配对**: 拆出的两段互配;
 - 交叉配对**: 每段各配一个原棒。
- 我们穷举了这两类所有对称情况, 且逻辑互斥、完备, 不会漏解。

```

#include <bits/stdc++.h>
using namespace std;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    cin >> T;
    while (T--){
        long long l1, l2, l3;
        cin >> l1 >> l2 >> l3;

        bool ok = false;
        // 1. 拆第三根, 能凑前两根之和吗?
        if (l1 + l2 == l3) ok = true;
        // 拆第一根
        if (l2 + l3 == l1) ok = true;
        // 拆第二根
        if (l3 + l1 == l2) ok = true;

        // 2. 平分型: 拆 l1, l2 == l3 且 l1 偶数
        if (!ok && (l1 % 2 == 0) && (l2 == l3)) ok = true;
        // 拆 l2
        if (!ok && (l2 % 2 == 0) && (l3 == l1)) ok = true;
        // 拆 l3
        if (!ok && (l3 % 2 == 0) && (l1 == l2)) ok = true;

        cout << (ok ? "YES\n" : "NO\n");
    }
    return 0;
}

```

L

[Codeforces Round 982 \(Div. 2\) C. Add Zeros](#)

DP

下面给出本题的**思路分析**、**正确性证明**以及**高效实现**。

一、题意转化

- 初始数组长度为 n , 元素为 $a_1, \dots, a_n$ 。
- 每次操作选 i ($1 < i \leq |a|$) 满足

$$a_i = |a| + 1 - i,$$

然后把当前长度 $|a|$ 增长为 $|a| + (i - 1)$ (末尾补 $i - 1$ 个零)。

- 零不会触发新的操作, 因为若新补的零在位置 $j > n$ 上, 要想满足 $0 = |a| + 1 - j$ 则需要 $j = |a| + 1$, 但实际 $j \leq |a|$ 。
- 因此所有能做的操作都来自于最初的三百多万个 a_i 。

定义

$$x_i = a_i + i - 1, \quad y_i = x_i + (i - 1) = a_i + 2i - 2.$$

从一个长度 S 出发, 能对位置 i 做操作的**必要且充分条件**是

$$S = x_i.$$

操作后长度变为

$$S' = S + (i - 1) = x_i + (i - 1) = y_i.$$

题目等价于: 从起点 $S_0 = n$, 给定若干“跳跃”规则

若当前是在 $S = x_i$, 可跳到 y_i ,

可重复使用不同的 i (每个 i 最多一次, 因为若用同一个 i 再次要求 $S = x_i$ 才能再跳, 但 S 已经变成更大值), 求能到达的最大 S 。

二、建图与 DP

1. 收集可跳跃的点

对每个 $i = 2, 3, \dots, n$ (注意 $i = 1$ 无法操作), 计算

$$x_i = a_i + i - 1, \quad y_i = x_i + (i - 1).$$

如果 $x_i < n$, 从初始 n 出发决不可能回退到 x_i , 可以直接忽略。

2. DP/贪心策略

- 初始化一个映射 (或哈希表) `dp`, 记录“某长度 S 可到达的最大长度”。开始时 `dp[n] = n`。
- 将所有对 (x_i, y_i) 按照 x_i **降序** 排好。
- 依次处理每个对 (x, y) :
 - 如果此时 `dp` 中存在键 `x` (意味着可以达到长度 x), 则可以跳到长度

$$y = y_i,$$

并更新 $dp[x] = \max(dp[x], dp[y])$ (也即跳过这条后从 x 最多能走到 $dp[y]$, 而后可能再利用更小的 x' 继续跳)。

- 最终答案即 $dp[n]$ 。

此法等价于在有向无环图 (因为每次跳跃都会严格增加 S) 上做可达性+最长路径。

- **ops 含义**: 每个 (x, y) 对应一次可走的跳跃 $x \rightarrow y$ 。
- **排序并降序遍历**: 保证当处理一个 x 时, 所有更大的 x' 已经被考虑过, $dp[y]$ 也已计算完毕 (拓扑序)。
- **dp 映射**: 稀疏存储只有那些 x 值出现过且可达的条目。

这样即可得到初始长度 n 能达到的最长最终长度。

三、正确性说明

- **跳跃条件完备**: 只有当当前长度 S 恰好等于某个 x_i 时, 原题中才能对位置 i 进行操作, 进而把长度变为 y_i 。
 - **递增无环**: 每次跳跃都会增加 S (因为 $y_i = x_i + (i - 1) > x_i$), 不存在环, 能用简单的 “降序扫描 + DP” 或 “拓扑 + DP” 完成最长可达值。
 - **忽略小于 n 的 x_i** : 任何 $x_i < n$ 都永远无法从初始 $S_0 = n$ 走到, 故无需考虑。
-

四、复杂度

- 每个测试用例处理 n 个元素, 收集 (x_i, y_i) 对并排序, $O(n \log n)$ 。
- 用 `std::map` 或 `unordered_map` 记录 dp , 最坏插入和查询 $O(\log n)$, 总计 $O(n \log n)$ 。
- 总体 $\sum n \leq 3 \times 10^5$, 完全可接受。

```

#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T;
    cin >> T;
    while (T--) {
        int n;
        cin >> n;
        vector<ll> a(n+1);
        for (int i = 1; i <= n; i++) {
            cin >> a[i];
        }
        // 收集所有可操作的 (x,y) 对
        vector<pair<ll,ll>> ops;
        ops.reserve(n);
        for (int i = 2; i <= n; i++) {
            ll x = a[i] + i - 1;
            if (x < n) continue; // 无法从初始长度 n 到达
            ll y = x + (i - 1);
            ops.emplace_back(x, y);
        }
        // 按 x 降序
        sort(ops.begin(), ops.end(), greater<>());

        // dp[S] = 能从 S 走到的最大长度
        map<ll,ll> dp;
        dp[n] = n;

        for (auto &pr : ops) {
            ll x = pr.first, y = pr.second;
            auto it = dp.find(x);
            if (it != dp.end()) {
                // 可从 x 跳到 y
                ll reachable = max(it->second, dp.count(y) ? dp[y] : y);
                dp[x] = reachable;
            }
        }

        // 答案是从初始 n 出发能达到的最大长度
        cout << dp[n] << "\n";
    }
    return 0;
}

```

```
}
```

```
#include <bits/stdc++.h>
const int N=2e5+10;
const int inf = 0x3f3f3f3f;
using namespace std;
using ull = unsigned long long int;
using ll = long long int;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    int _;
    cin>>_;
    while (_--){
        ll n;
        cin>>n;
        vector<pair<ll,ll>> a(n);
        for(int i=0;i<n;i++){
            cin>>a[i].second;
            a[i].first=a[i].second+i;
            a[i].second=i;
        }
        sort(a.begin(),a.end());
        map<ll,ll> dp;
        dp[n]=1;
        ll ans=n;
        for(int i=0;i<n;i++){
            dp[a[i].first+a[i].second]=dp[a[i].first];
            if(dp[a[i].first+a[i].second]){
                ans=max(ans,a[i].first+a[i].second);
            }
        }
        cout<<ans<<'\n';
    }
    return 0;
}
```

M

[Codeforces Round 1037 \(Div. 3\) G1. Big Wins! \(easy version\)](#)最小值所在最大区间

栈/并查集/笛卡尔树

一、思路概述

对于每个位置 i ，我们想找出包含 i 且以 a_i 为该区间最小值的 **最长** 连续区间 $[L_i, R_i]$ 。查询时给定值 x ，其在数组中的下标是 $\text{pos}[x]$ ，直接输出 $[L_{\text{pos}[x]}, R_{\text{pos}[x]}]$ 。

怎么求 $[L_i, R_i]$ ？这是典型的「下一个更小元素」（Next Smaller Element, NSE）问题：

- 设

$$L_i = \max\{j + 1 : j < i, a_j < a_i\} \quad (\text{如果左侧没有更小, 取 } 1),$$

也就是说，区间左端不能越过最靠近 i 且比它小的那个元素的右边界。

- 类似地

$$R_i = \min\{j - 1 : j > i, a_j < a_i\} \quad (\text{如果右侧没有更小, 取 } n).$$

那么 $[L_i, R_i]$ 上所有元素都 $\geq a_i$ ，又包含 i ，并且一旦扩到左边再跨过一个更小的或右边再跨过一个更小的，就会让区间最小值小于 a_i 。所以这是最长的满足“最小值恰好是 a_i ”的区间。

1. 单调栈能正确找到最近更小元素

- 在从左到右的遍历中，维护一个**递增栈**（栈底到栈顶对应的 a -值严格递增）。
- 入栈前把栈顶所有 $\geq a[i]$ 的弹出，剩下的栈顶就是最近的比它小的下标。
- 因此 $L[i] = \text{prevSmaller}[i] + 1$ 给出了最长左扩的边界。
- 同理从右向左求得 $R[i]$ 。

2. 区间最长且最小值为 a_i

- 对任何区间 $[l, r]$ 包含 i ，若要保证它的最小值 $\geq a_i$ ，就必须在左侧 $j < i$ 的位置上没有 $a[j] < a_i$ ，同样右侧也不能有。
- 因此合法的最大 l 是 $\text{prevSmaller}[i] + 1$ ，最大 r 是 $\text{nextSmaller}[i] - 1$ 。
- 这个区间必然包含 i ，并且若再伸出一格就会碰到一个更小的值，导致最小值下降，因此它是最长的。

3. 针对查询的正确性

- 查询给出的为值 x ，我们通过 $\text{pos}[x]$ 找到该值在数组中的下标 i 。然后直接报告 $[L[i], R[i]]$ ，即该值作为最小值能撑起的最长区间。

4. 复杂度

- 单调栈两次扫描 $O(n)$ 。
- 构造 pos 数组和回答查询都是 $O(n)$ 。
- 总体时间与空间均为 $O(n)$ ，满足 $n \leq 3 \times 10^5$ 。

二、单调栈算法

我们只要在 $O(n)$ 内，用两个单调栈分别求出「对于每个 i ，左侧第一个比它小的位置」和「右侧第一个比它小的位置」，然后拼出 $[L_i, R_i]$ 。

```

#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;
    vector<int> a(n+2), pos(n+1), q(n);
    // 边界哨兵, a[0]=a[n+1]= -∞
    a[0] = a[n+1] = 0;

    for (int i = 1; i <= n; i++){
        cin >> a[i];
        pos[a[i]] = i;
    }
    for (int i = 0; i < n; i++){
        cin >> q[i];
    }

    vector<int> L(n+2), R(n+2);
    // --- 求左侧第一个更小的位置 prevSmaller[i] ---
    // 我们定义 prevSmaller[i] = 最大 j < i 且 a[j] < a[i], 若不存在则 j=0
    {
        stack<int> st;
        st.push(0); // 哨兵
        for (int i = 1; i <= n; i++){
            while (st.top() != 0 && a[st.top()] >= a[i])
                st.pop();
            prevSmaller:
            L[i] = st.top() + 1;
            st.push(i);
        }
    }

    // --- 求右侧第一个更小的位置 nextSmaller[i] ---
    // nextSmaller[i] = 最小 j > i 且 a[j] < a[i], 若不存在则 j=n+1
    {
        stack<int> st;
        st.push(n+1); // 哨兵
        for (int i = n; i >= 1; i--){
            while (st.top() != n+1 && a[st.top()] >= a[i])
                st.pop();
            R[i] = st.top() - 1;
            st.push(i);
        }
    }
}

```

```

    }
}

// 现在 [L[i], R[i]] 就是包含 i 的、以 a[i] 为最小值的最长区间

// 按查询顺序输出
for (int x : q){
    int i = pos[x];
    cout << L[i] << " " << R[i] << "\n";
}
return 0;
}

```

N

P5057 [CQOI2006] 简单题

树状数组

下面给出一个基于「差分 + 树状数组(Binary Indexed Tree)」的高效方案，以及它的正确性说明。

一、思路与数据结构

我们需要支持两种操作在长度为 n 的二进制数组 a （初始全 0）上：

1. **区间翻转** $[l, r]$ ：将区间内每个元素取反（ $0 \rightarrow 1, 1 \rightarrow 0$ ）。
2. **点询问** x ：输出当前 a_x 。

显然直接维护 $O(n)$ 的区间翻转会太慢（最坏 $O(nq)$ ），我们需要 $O(\log n)$ 的数据结构。关键是注意到“翻转”其实对该区间元素做了加 1（模 2），即按位异或 1。使用差分技巧：

- 维护一个差分数组 $d[1 \dots n + 1]$ ，初始全 0。定义

$$a_x = \sum_{i=1}^x d_i \pmod{2}.$$

这里我们使用「前缀和」表示原数组。

- 要把 $[l, r]$ 之间的每个 a_i 加 1（模 2），只需要在差分上做

$$d_l \oplus = 1, \quad d_{r+1} \oplus = 1.$$

这样在任意位置 x 的前缀和 $\sum_{i=1}^x d_i$ 会在区间 $[l, r]$ 内多一（模2），即实现了翻转。

- 然后一次 **点查询** a_x 就是

$$a_x = \left(\sum_{i=1}^x d_i \right) \bmod 2.$$

要支持差分点更新和前缀和查询，我们可以用 **树状数组**（BIT）：

- `update(p, v)`：将差分点 $d_p += v$ （这里取 $v = 1$ ，后面查询时取模2）。
- `query(x)`：计算 $\sum_{i=1}^x d_i$ 。

每次**翻转**区间 $[l, r]$ 执行两次 `update`：

```
update(l, 1);
update(r+1, 1);
```

每次**询问**点 x 执行

```
int s = query(x);
answer = s % 2;
```

树状数组支持 $O(\log n)$ 的 `update` 和 `query`，总体 $O((n + q) \log n)$ ，在 $n, q \leq 2 \times 10^5$ 内能高效运行。

三、正确性说明

1. 差分转化

维护差分数组 d ，原数组通过前缀和恢复：

$$a_x = \sum_{i=1}^x d_i \pmod{2}.$$

2. 区间加法

在差分上做 `d[1] += 1`，`d[r+1] += 1`（模任意，最后看前缀和的奇偶）等价于对原数组区间 $[l, r]$ 做“每位加1（模2）”，即取反。

3. 树状数组支持

- `update(p, 1)` 即差分点加1。
- `query(x)` 即累加前缀和。

4. 时间复杂度

每次操作 $O(\log n)$ ，总计 $O((n + q) \log n)$ ，能在 2×10^5 量级内高效运行。

```

#include <bits/stdc++.h>
using namespace std;
struct BIT {
    int n;
    vector<int> t;
    BIT(int _n): n(_n), t(n+1, 0) {}
    // 在位置 p 上加 v
    void update(int p, int v) {
        for (; p <= n; p += p & -p)
            t[p] += v;
    }
    // 查询前缀和 sum[1..p]
    int query(int p) const {
        int s = 0;
        for (; p > 0; p -= p & -p)
            s += t[p];
        return s;
    }
};
int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, q;
    cin >> n >> q;
    BIT bit(n+1); // 可以更新到 r+1 = n+1
    while (q--) {
        int type;
        cin >> type;
        if (type == 1) {
            int l, r;
            cin >> l >> r;
            // 区间 [l,r] 翻转: 差分 d[l]++, d[r+1]++
            bit.update(l, 1);
            bit.update(r+1, 1);
        } else {
            int x;
            cin >> x;
            // a[x] = (sum d[1..x]) % 2
            int flips = bit.query(x);
            cout << (flips & 1) << "\n";
        }
    }
    return 0;
}

```