# Guess and lies

我们考虑如何记录一个局面。容易发现，对一个数 $i$ ，假如 Alice 想的数是 $i$，她在之前说谎的次数是固定的，计作 $a_i$。当只有一个 $a_i \leq 1$ 时，Bob可以确定答案。则我们将游戏转化为：Bob 将序列分为前后两段，Alice选择一段加1，Bob希望用尽量少的步骤使得只有一个数$\leq 1$。

容易发现，在进程中，$\leq 1$的数一定形如第一段 $a$ 个 1，第二段 $b$ 个 0，第三段 $c$ 个 1。我们记 $dp_{a,b,c}$ 表示这种情况下的答案，暴力转移复杂度为 $O(n^4)$。利用决策单调性可以优化到 $O(n^3)$。

另外，注意到固定 $a, b$ 时，$dp_{a,b,c}$ 关于 $c$ 是单调的，且值域最大不超过 $O(logn)$ 。交换值域和第三维可以做到 $O(n^2 logn)$ ，可以通过此题。

更高复杂度的做法通过打表也可以通过。

# Black and White

对于一个位置 $i, j$，如果该格子是黑色，我们连一条 $A_i$ 到 $B_j$ 的边。操作不改变连通性，由此容易证明，可以全部染黑等价于初始联通。利用 Prim 算法求解最小生成树即可。复杂度为 $O(n^2)$。

# Minimum grid

我们称 $b_i$ 和 $c_i$ 为行限定数和列限定数，统称限定数.

考虑最大的那个限定数，不妨设为 $x$；找到行、列限定数是 $x$ 的那些行和列组成的子矩形.

显然，其他数都不能满足 $x$ 的要求.

假设这个子矩形 $n'$ 行 $m'$ 列，我们把每行建一个点，每列建一个点，可以填数的位置建一条边，得到一个二分图；这个二分图的最大匹配 $t$ 就是我们能节省出的 $x$ 的个数.

那么这个子矩形对答案的贡献就为 $(n' + m' - t) \times x$.

解决了这些限定数以后考虑次大的限定数 $y$，注意到次大的数可以"放进"限定数是最大的数的那些行列中，但是这样也不能节省任何的 $y$；因此还是建立二分图求最大匹配，然后算贡献.

这样把每个限定数的二分图都建出来跑一遍匹配，算一下贡献即可.

# Count

先容斥 $K$ 个格子是否必选。之后对于行、列、对角线容斥是否不能选。只需统计出剩余 $D$ 个格子候选的答案即可。

容易发现候选格子数 = $AB - X$，其中 $A, B, X$ 分别为未被 ban 的行数，未被 ban 的列数，行列未被 ban 但因为对角线被 ban 的格子数。

我们可以对每个 $i$ ，考虑 $i, n+1-i$ 行、列是否被 ban，计算出其对于 $A, B, X$ 的贡献，而后三维背包即可。

复杂度为 $O(n^4 * 2^k)$。

# Math

考虑 $x^2+y^2=k(xy+1)$ 的解。不妨 $x\le y$，由韦达定理，$(x,kx-y)$ 也是一组解，并且 $x(kx-y)=(y^2-1)$。

由于 $kx-y$ 的绝对值更小且依然非负（代入原方程可知非负），因此对于该 $k$，绝对值最小的一组正整数解一定满足 $kx-y=0$。可以发现这组解具有 $(a,a^3)$ 的形式。又因为其余解均可通过 $(x,y)\to(x,kx-y)$ 的递降得到该解，我们也可以反推回去。

最终我们可以预处理出范围在 $10^{18}$ 以内的所有解，每次询问二分即可。复杂度 $O(n_{max}^{1/3}+T\log_2 n_{max})$。

# 24dian

按照题面模拟即可。暴力枚举第一次或最后一次的两数均可。

# Yu Ling(Ling YueZheng) and Colorful Tree

## 粗略版

~~有多少人写虚树、树套树啊~~

把点按 `dfn` 序重编号使得在同一条链上，编号小的深度大.

树分块，处理出表示树上 $u$ 的所有祖先的 `Bitset`；然后分块处理出表示树上颜色 $\in[l,r]$ 的所有结点的 `Bitset`；最后根号分治处理出表示树上颜色是 $x$ 的倍数的所有结点的 `Bitset`.

把这些 `Bitset` 按位与以下，用 `_find_next()` 找到第一个位即可.

## 详细版

将树上的点按照 `Dfn` 序重新编号，考虑使用 `Bitset` 存储所有符合颜色要求的点的编号，然后在 `Bitset` 上找到最后一个颜色要求的点的编号.

具体地，对于询问 `1 u l r x` 和 `2 u l r x`，可以处理出如下四个 `Bitset`：$F,P,S,\mu$，其中 $F$ 存储所有 $u$ 的祖先，$P$ 存储所有颜色 $\le l-1$ 的结点，$S$ 存储所有颜色 $\le r$ 的结点，$\mu$ 存储所有颜色是 $x$ 的倍数的结点，那么对于询问 `1`，$\vartheta=F\cap(P\oplus S)\cap\mu$ 即为储存所有符合颜色要求的点的编号的 `Bitset`，在 $\vartheta$ 上找到最后一个为 1 的位即可，询问 `2` 同理，将 $\mu$ 变为 $\complement_\mu$ 即可，注意需要手写 `Bitset`；为了减小常数，集合的位运算表达式求值需要对每一位独立地作位运算表达式求值之后再合并，也就是 $\vartheta_i=F_i\cap(P_i\oplus S_i)\cap\mu_i$.

考虑如何求 $F$：设置参数 $\alpha$，重复以下过程：找到树上最深的叶子，标记其 $\lceil\frac{n}{\alpha}\rceil$ 代祖先结点，删除该结点对应的子树；直到根节点被标记，这样标记了不超过 $\alpha$ 个结点，使得每个结点和其最近的被标记的祖先之间的距离不超过 $\lceil\frac{n}{\alpha}\rceil$；在所有标记点处预处理出该结点的 $F$，对于其他的结点，求其 $F$ 只需要找到其最近的被标记的祖先，之后取该被标记的祖先的 $F$ 和一路上所有结点编号的并即可.

考虑如何求 $P,S$：设置参数 $\beta$，离线预处理出有多少个结点最终被染成颜色 $x$，记为 $c_x$，按照 $\lceil\frac{m}{\beta}\rceil$ 为块长作带权分块，在所有与上一个标记点之间的 $\sum c_x$ 达到 $\lceil\frac{m}{\beta}\rceil$ 的颜色 $x$ 处设立标记点，这样标记了不超过 $\beta$ 种颜色，使得每种颜色和其最大的不超过其颜色的被标记的颜色之间的距离不超过 $\lceil\frac{m}{\beta}\rceil$；在所有标记颜色出设立 $P$，对于每次修改操作暴力更新所有不超过其颜色的标记颜色处的 $P$，对于其他的颜色，求其 $P$ 同理.

考虑如何求 $\mu$：设置参数 $\gamma$，对于每种颜色 $x$ 我们建立一个数据结构 $\kappa_x$，在修改的过程中，对于修改到的颜色 $x$，我们将 $x$ 的所有因子 $y$ 对应的结点加入其数据结构 $\kappa_y$ 中，对于 $\kappa_x$，当其中结点数达到 $\lceil\frac{\sum d(i)}{\gamma}\rceil$ 的时候，我们就单独对其维护一个 `Bitset`，否则就直接使用一个链表进行维护.

设 $m,q=\mathcal{O}(n)$，$k=\max d(i)$，这三部分的预处理时间复杂度之和、以及空间复杂度均为 $\mathcal{O}(n\log n+\frac{n(\alpha+\beta+\gamma)}{\omega})$，查询时间复杂度之和为 $\mathcal{O}(\frac{n^2}{\min(\omega,\alpha,\beta,\gamma)})$，容易得出应有 $\alpha,\beta,\gamma=\mathcal{O}(\omega)$，这时时间复杂度为 $\mathcal{O}(\frac{n^2}{\omega})$，空间复杂度为 $\mathcal{O}(n\log n)$.

需要注意在各个部分涉及到 `Bitset` 的操作的时候，尽量节省操作的总次数.

该算法时间复杂度为 $\mathcal{O}(\frac{n^2}{\omega})$，空间复杂度为 $\mathcal{O}(n\log n)$.

# Ling Qiu, Luna and Triple Backpack

## 粗略版

咕咕咕.....

## 详细版

我们考虑令 $\xi_a=\max\limits_{x=1}^{3}\max\limits_{i\in S_x}a_i$，$\xi_b=\sum\limits_{x=1}^{3}\max\limits_{i\in S_x}b_i$，$\xi_c=\max\limits_{x=1}^{3}\sum\limits_{i\in S_x}c_i$，$\xi_d=\sum\limits_{x=1}^{3}\sum\limits_{i\in S_x}d_i$，那么 $\Xi=\xi_a\times\xi_b\times\xi_c\times\xi_d$；注意到对于一组给定的书本来说，$\xi_a$ 和 $\xi_d$ 是容易计算的定值，无论怎样划分，这两个数都是确定的，因此只需要最小化 $\xi_b\times\xi_c$ 即可.

考虑到 $\xi_b$ 是三个集合中 $b_i$ 的最大值的和，一定是某三本书的 $b_i$ 之和，因此比较能够确定下来，考虑枚举这三本书.

我们把所有的书按照 $b_i$ 从小到大进行排序，会发现最后选择的这三本书一定有一本书是第 $n$ 本，假设另外两本是第 $l,r$ 本，那么 $1,\dots,l$ 中的书可以被归为任何一个集合，而 $l+1,\dots,r$ 中的书只能被归为第二、三个集合，最后 $r+1,\dots,n$ 中的书只能被归为第三个集合，除此以外在 $\xi_b$ 上没有别的限制，也就是说，我们在确定了 $l,r$ 之后，有 $\xi_b=b_l+b_r+b_n$，其实就确定了 $\xi_a,\xi_b,\xi_d$，只差一个 $\xi_c$ 了，这个时候我们只需要考虑令 $\xi_c$ 最小化，称最小可能的 $\xi_c$ 为 $R_{l,r}$，那么对于所有的 $1\le l<r<n$，都有一个 $R_{l,r}$ 的定义，最后的答案 $\mathrm{Ans}=\min_{1\le l<r<n}(\xi'_a\times(b_l+b_r+b_n)\times R_{l,r}\times\xi'_d)$.

现在唯一的任务，就是计算所有 $1\le l<r<n$ 下的 $R_{l,r}$，考虑到这是一个双背包问题，我们思考复杂度和 $\sum c_i$ 有关的算法，对于一个 $R_{l,r}$ 来说，我们需要将 $1,\dots,l$ 中的书加入任何一个背包中，将 $l+1,\dots,r$ 中的书加入第二、三个背包中，$r+1,\dots,n$ 中的书加入第三个背包中，然后得到三个背包容量最大值的最小可能值.

如果对于每个 $R_{l,r}$ 都进行一次加入背包的话，那么就进行了太多无意义的操作，显然对于背包问题来说，加入物品的顺序是不太重要的，考虑从这个点着手优化整个过程，不难想到预处理，具体地，我们预处理出一部分 $\mathrm{Pre}$ 信息描述 $1,\dots,l$，再预处理出一部分 $\mathrm{Suf}$ 信息描述 $l+1,\dots,n$，最后在计算 $R_{l,r}$ 的时候想方设法合并这两个背包即可.

这样做的好处是预处理的信息是可以递推一遍就下来的，只需要将所有物品都加入一次背包即可，我们考虑令 $P_{i,x,y,z}$ 表示考虑如果 $1,\dots,i$ 的物品可以加入任何一个背包中，那么三个背包的容量分别是 $x,y,z$ 是否有可能，这显然是有很多浪费的，注意到 $x+y+z$ 就是前 $i$ 个物品的 $c_i$ 之和，因此 $z$ 这一维可以省略掉；又由于 $P_{i,x,y}$ 是一个取值在 `0,1` 之间的布尔变量，考虑使用位集，即 `Bitset` 进行优化.

从而，也就不难想到构造 $\Theta(n \times \sum c_i)$ 个长度为 $\sum c_i$ 的 `Bitset`，即 $P_{i,x}$，表示考虑如果 $1,\dots,i$ 的物品可以加入任何一个背包中，那么第一个背包的容量是 $x$，第二个背包的容量 $y$ 是否可以是这一个比特的位置；构造 $\Theta(n^2)$ 个长度为 $\sum c_i$ 的 `Bitset`，即 $S_{i,j}$，表示考虑如果 $i+1,\dots,j$ 中的书只能被归为第二、三个书包中，$j+1,\dots,n$ 中的书只能被归为第三个背包中，那么第二个背包的容量 $y$ 是否可以是这一个比特的位置.

显然，有 `Pre[i][t]=Pre[i-1][t]|(Pre[i-1][t]<<c[i])|Pre[i-1][t-c[i]];` 以及 `Suf[i][j]=Suf[i+1][j]|(Suf[i+1][j]<<c[i]);`，注意一下递推顺序即可.

我们在得到了 $P$ 和 $S$ 之后，就应该着手考虑如何计算 $R$ 了；显然，对于可行小背包容量，比其更大的背包容量也一定可行，故而我们考虑二分答案转为判定问题，对于 `Bitset` 会比较方便，因此我们只需要判定是否 $R_{l,r} \le \lambda$，也就是说，是否存在一种分组物品的方式能够装进大小均为 $\lambda$ 的三个背包中.

通过 $P$ 和 $S$，可以把判定是否 $R_{l,r} \le \lambda$ 形式化地转化为如下的一个问题，即是否存在满足要求的一组 $x,t,t' \in \mathbb{Z}$，使得 $P_{l,x,t} = \texttt{true}, S_{l,r,t'} = \texttt{true}$，并且三个条件同时成立：$0 \le x \le \lambda$（第一个背包），$0 \le t+t' \le \lambda$（第二个背包），$0 \le \sum c_i - (x+t+t') \le \lambda$（第三个背包）.

首先合并第二个背包和第三个背包的条件，第三个背包即 $-\sum c_i \le -(x+t+t') \le \lambda - \sum c_i$，$\sum c_i - \lambda \le x+t+t' \le \sum c_i$，也就等价于是 $(\sum c_i - \lambda) - x \le t+t' \le (\sum c_i) - x$，综合第二个背包，得到 $\max(0, (\sum c_i - \lambda) - x) \le t+t' \le \min(\lambda, (\sum c_i) - x)$.

仔细观察会发现，如果 $P_{l,x,t} = \texttt{true}, S_{l,r,t'} = \texttt{true}$，那么 $0 \le x$，$0 \le t+t'$，以及 $t+t' \le (\sum c_i) - x$，这三个条件都是必然的，因此剩下需要考虑的条件就只有 $x \le \lambda$ 以及 $(\sum c_i - \lambda) - x \le t+t' \le \lambda$ 这两个条件了.

考虑条件 $x \le \lambda$，这个条件只对第一个背包的容量 $x$ 产生限制，我们考虑贪心地将其转化为 $\max(0, \lambda - \max c_i) \le x \le \lambda$；这是因为第一个背包尽量多装对第二个、第三个背包具有决策包容性，让第一个背包尽量多装肯定不会使答案变劣；除非第一个背包已经无法尽量多装了，所有物品都已经在第一个背包中了，也就是在 $\sum c_i \le \lambda$ 的时候，我们只需要考察一个点处的 $x$，也就是 $x = \sum c_i$ 时可不可行；否则我们需要考察 $\mathcal{O}(\max c_i)$ 个 $x$ 可不可行.

在考察一个 $x$ 可不可行的时候，经过上述转化，我们只需要考虑是否存在 $t,t' \in \mathbb{Z}$，满足 $P_{l,x,t} = \texttt{true}, S_{l,r,t'} = \texttt{true}$，使得 $(\sum c_i - \lambda) - x \le t+t' \le \lambda$ 即可，实质上是查询两个 `Bitset` 中是否分别存在一个值使得这两个值的和在给定区间内.

单独看这个问题：假设给定的两个 `Bitset` 分别为 $A$ 和 $B$，我们需要知道是否分别存在两个位 $t,t'$ 使得 $u \le t+t' \le v$，我们首先将 $B$ 逆过来得到新的 `Bitset` 也就是 $B'$，设置一个阈值 $w$ 使得 $B'_i = B_{w-i}$，然后就将问题转化为 $A$ 和 $B'$ 中是否分别存在两个位 $t,t'$ 使得 $u \le t + (w-t') \le v$，也就是 $u-w \le t-t' \le v-w$.

实际上，我们无法利用 `Bitset` 快速地找到一个 `Bitset` 的逆转，在这个算法中，我们注意到唯一需要被逆转的 `Bitset` 就是 $S_{l,r}$，因此我们在一开始求 $S_{l,r}$ 的时候就只求 $S_{l,r}$ 在某个固定阈值 $w$ 下的逆转即可，这样逆转操作就被事先计算好了.

转化为 $u-w \le t-t' \le v-w$ 之后，我们将 $B'$ 左移 $u-w$ 位，或者，当 $u < w$ 时，相应地右移 $w-u$ 位，得到 $B''$，如此，条件进一步转化为 $u-w \le t-(t'-(u-w)) \le v-w$，也就是 $0 \le t-t' \le v-u$；这里我们选取合适的 $w$ 的目的是为了让 $B'$ 转化为 $B''$ 的过程中不出现 `Bitset` 左移、右移的溢出，省去维护不必要的位，降低常数.

令 $\Delta = v-u$，得到 $0 \le t-t' \le \Delta$，如果 $\Delta = 0$，那么显然直接做 $A \cap B''$，如果 $A \cap B''$ 中有位存在，就说明存在 $t,t'$；如果 $\Delta = 1$，那么就可以求 $(A \cup A_1) \cap B''$，观察 $(A \cup A_1) \cap B''$ 中是否有位存在，其中 $A_1$ 是 $A$ 左移 $1$ 位后的 `Bitset`；类似地，对于任意的 $\Delta$，我们需要求出 $(\cup_{i=0}^{\Delta} A_i) \cap B''$ 中是否有位存在，显然可以针对 $\Delta$ 使用倍增算法解决.

这样，检查是否存在 $(\sum c_i-\lambda)-x\le t+t'\le\lambda$ 的过程可以在 $\mathcal{O}(\frac{\sum c_i\log\sum c_i}{\omega})$ 的时间内解决；但是这样是不足够的，因为我们还要枚举 $\mathcal{O}(\max c_i)$ 个 $x$.

注意到在 $\max(0,\lambda-\max c_i)\le x\le\lambda$ 时，条件 $(\sum c_i-\lambda)-x\le t+t'\le\lambda$ 有很大一部分是重合的，也就是 $\sum c_i-\lambda\le t+t'\le\lambda$ 的部分，我们可以先判定这部分重合的条件是否被满足，也就是是否存在 $\max(0,\lambda-\max c_i)\le x\le\lambda$，使得存在 $t,t'\in\mathbb{Z}$，使得 $P_{l,x,t}=\texttt{true},S_{l,r,t'}=\texttt{true}$，并且 $\sum c_i-\lambda\le t+t'\le\lambda$，我们需要将所有满足 $\max(0,\lambda-\max c_i)\le x\le\lambda$ 的 $P_{l,x}$ 作并集，这部分可以使用双栈模拟队列预处理出来；得到了并集 `Bitset` 之后，将这个并集 `Bitset` 和 $S_{l,r}$ 作查询即可.

剩下的不重合的部分，也就是 $(\sum c_i-\lambda)-x\le t+t'<\sum c_i-\lambda$，则对于每个 `Bitset` 都和 $S_{l,r}$ 作一遍查询即可.

通过这样的过程，我们可以 $\mathcal{O}(\frac{\sum c_i(\log\sum c_i+\max c_i\log\max c_i)}{\omega})$ 地判定是否存在一种分组物品的方式能够装进大小均为 $\lambda$ 的三个背包中，进而得到是否 $R_{l,r}\le\lambda$；考虑如何利用这样的过程求出所有 $1\le l<r<n$ 下的 $R_{l,r}$.

我们称一次判定是否 $R_{l,r}\le\lambda$ 的过程为进行了一次“操作“，那么现在我们就需要构造一种使用这些”操作“的方案来求出所有 $1\le l<r<n$ 下的 $R_{l,r}$；我们注意到，回归 $R_{l,r}$ 的定义，可以将 $1,\dots,l$ 中的书加入任何一个背包中，将 $l+1,\dots,r$ 中的书加入第二、三个背包中，$r+1,\dots,n$ 中的书加入第三个背包中，然后 $R_{l,r}$ 是三个背包容量最大值的最小可能值；因此 $l,r$ 越大，这个限制就越宽松，背包容量最大值的最小可能值就越小，也就是说，对于同样的 $l$ 以及 $r<r'$，有 $R_{l,r}\ge R_{l,r'}$，对于同样的 $r$ 以及 $l\le l'$，有 $R_{l,r}\le R_{l,r'}$；我们考虑利用 $R$ 矩阵的这种二维单调性，先提取出 $R$ 矩阵的一行：令函数 `Solve(l,r,s,t)` 函数的功能是对于某一特定的行 $\star$，求出所有 $R_{\star,i}$，其中 $i\in[l,r]$，而且我们已知这些 $R_{\star,i}\in[s,t]$；显然，我们可以找到中点位置 $m=\lfloor\frac{l+r}{2}\rfloor$，通过上述”操作“暴力二分出 $R_{\star,m}$，然后递归到两边 `Solve(l,m-1,s,R[][m])` 以及 `Solve(m+1,r,R[][m],t)` 进行处理；分析这个过程的时间复杂度，令 $F(n,m)$ 为区间长度大小为 $n$、值域大小为 $m$ 时所消耗的”操作“次数，可以得到 $F(n,m)=2F(\frac{n}{2},\frac{m}{2})+\mathcal{O}(\log m)$；使用主定理求解，得到 $m=\mathcal{O}(n)$ 时 $F(n,m)=\mathcal{O}(n)$；而当 $m$ 在此基础上每翻一番，变为 $2m,4m,8m,\dots$ 时，每个点出就多消耗一次”操作“，总复杂度就增长 $\mathcal{O}(n)$；故而 $F(n,m)=\mathcal{O}(n\log m-n\log n)$，也就是说，对于 $R$ 矩阵的每一行，求解这一行所有的值的二分“操作”次数为 $\mathcal{O}(n\times(\log\sum c_i-\log n))$ 级别的；二维情况下并不会有其他的优化技巧，我们最优的求解二维完全单调性的算法的复杂度和暴力枚举每一行套用一维完全完全单调性的算法的复杂度是一样的，而且暴力枚举每一行，就可以将每一行离线，使得消耗空间较大的 $P$ 数组可以通过滚动数组来计算，去除了空间复杂度的瓶颈，使得我们只需要维护 $\mathcal{O}(n^2+\sum c_i)$ 个 `Bitset` 即可，这样空间复杂度就被降为了 $\mathcal{O}(\frac{\sum c_i(n^2+\sum c_i)}{\omega})$.

像这样执行二分套二分，我们就能够求出所有的 $R_{l,r}$，初步分析其复杂度上界为 $\mathcal{O}(\frac{n^2\times(\log\sum c_i-\log n)\sum c_i(\log\sum c_i+\max c_i\log\max c_i)}{\omega})$，去除低次项，也就是 $\mathcal{O}(\frac{n^2\sum c_i\max c_i\times\log^2\max c_i}{\omega})$，这显然是非常糟糕的，考虑到原题动态规划解法的复杂度为 $\mathcal{O}(n(\sum c_i)^2)$；我们考虑均摊“触发”了上式 $\max c_i\log\max c_i$ 部分复杂度二分判定的代价，大致如下：对于一个 $l,r$ 的 $\mathcal{O}(\log\max c_i)$ 次判定，只有 $\mathcal{O}(\log\log\max c_i)$ 次判定有可能“卡进”那个不重合的部分；如果想在这 $\mathcal{O}(\log\log\max c_i)$ 次中卡出若干次，那么 $\lambda$ 就需要比较小，否则卡一次就会出现很多个 `Bitset` 位置对必须不都为 `true`，那么 $S_{l,r}$ 就会受限制，限制是 $\mathcal{O}(\frac{\lambda}{\max c_i})$ 级别的；受限制的话就会降低第一个二分的复杂度，降低的程度是一个 $\log$，而卡出的次数会导致 $\lambda$ 除以 2 的这么多次方的级别的一个限制，因此可以去掉一个 $\log$；另外，$P$ 数组的递推可以使用滚动数组，这是因为处理二维单调性的复杂度和第二个维度是没有关系的.

对手写 `Bitset` 的精细实现要求比较高，需要以一种常数比较小的写法实现复杂的左移、右移和部分位提取操作、还需要注意在左移、右移操作中可能导致的数组溢出问题.

该算法时间复杂度为 $\mathcal{O}(\frac{n^2\sum c_i\max c_i\log\max c_i}{\omega})$，空间复杂度为 $\mathcal{O}(\frac{\sum c_i(n^2+\sum c_i)}{\omega})$.

# Kuriyama Mirai and Exclusive Or

## 粗略版

异或同一个数，差分处理即可.

异或一个递增的数列，我们考虑这样的操作对答案数列的某一个二进制位的贡献.

容易发现按位考虑，对第 $i$ 个二进制位的贡献一定是类似于 `000111100001111100001111...` 这样的；我们把这个数列看成 $2^{i+1}$ 列的一个矩阵，那么贡献就可以被拆分为若干个连续子矩形，可以作二维差分，最后还原贡献即可.

节省空间可以开 `Bitset`，或者把询问离线下来枚举位处理.

## 详细版

作 $a$ 的差分数组 $b$，令 $b_i=a_i\oplus a_{i-1}$，则 $a_i=\oplus_{i=1}^n b_i$；再创建一个数组 $\Delta$.

考虑将原来的 $a$ 数组以及所有的 `0` 操作带来的影响记在 $b$ 数组中，将所有的 `1` 操作带来的影响记在 $\Delta$ 数组中，最后的答案 $a'_i=(\oplus_{j=1}^i b_j)\oplus\Delta_i$.

显然，原来的 $a_i$ 带来的影响是 $b_i\leftarrow b_i\oplus a_i$、$b_{i+1}\leftarrow b_{i+1}\oplus a_i$，一个 `0 l r x` 操作带来的影响是 $b_l\leftarrow b_l\oplus x$、$b_{r+1}\leftarrow b_{r+1}\oplus x$，实时更新 $b$ 即可.

接下来考虑计算 $\Delta$ 数组，按位考虑，对于第 $i$ 个二进制位，也就是位权为 $2^i$ 的位，异或上等差数列对 $\Delta$ 上这一位的影响是以 $2^{i+1}$ 为周期的.

例如，$3,4,5,\ldots,29$ 这个数列的第 $2$ 个二进制位，也就是位权为 $2^2=4$ 的位，这一位上的值依次为 $0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1$，是以 $0,1,1,1,1,0,0,0$ 为周期的，利用这个性质，我们建立二维布尔数组，长为 $\lceil\frac{n}{2^{i+1}}\rceil$，宽为 $2^{i+1}$.

那么在该二维布尔数组上，每次 `1` 操作带来的影响大致为一个矩阵，类似于刚才 $b$ 作为差分的思想，作二维差分 $\delta$ 来维护，最后再做二维后缀和还原回 $\Delta$ 即可.

实现时，我们考虑实现函数 $F(p,x)$，表示对于 $1\le i\le p$，将 $a_i$ 异或上 $x+(i-1)$.

还是考虑第 $2$ 个二进制位，以 $F(27,3)$ 为例，说明一些具体的细节.

| $\equiv 1$ | $\equiv 2$ | $\equiv 3$ | $\equiv 4$ | $\equiv 5$ | $\equiv 6$ | $\equiv 7$ | $\equiv 0$ | 分割线 | $\equiv 1$ | $\equiv 2$ | $\equiv 3$ | $\equiv 4$ | $\equiv 5$ | $\equiv 6$ | $\equiv 7$ | $\equiv 0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ■ | ■ | ■ | ■ | | | | | | | | | | | | |
| | ■ | ■ | ■ | ■ | | | | | | | | | | | | |
| | ■ | ■ | ■ | ■ | | | | | ■■ | | ■ | | ■ | | | |
| | ■ | ■ | | | | | | | ■ | | ■ | | | | | |

左侧为 $F(27,3)$ 对 $\Delta$ 的影响，■ 表示 $\Delta$ 异或一次；右侧为 $F(27,3)$ 对 $\delta$ 的影响，■ 表示 $\delta$ 异或一次，也就是在 $\Delta$ 上所有在这个位置左上方的位置都异或一次.

由 $\lfloor \frac{27}{8} \rfloor = 3$，有 3 行包含完整的 ■, ■, ■, ■，并且 ■, ■, ■, ■ 从 $(4-3) \mod 8 = 1$ 开始，到 $4+1=5$ 结束，故 $\delta_{3,1}, \delta_{3,5}$ 都要异或一次；又由 $27 \mod 8 = 3$，第 4 行包含 3 个合法位置，有 $\min(4, 3-1) = 2$ 个位置受影响，故 $\delta_{3,1}, \delta_{3,3}, \delta_{4,1}, \delta_{4,3}$ 都要异或一次.

其他的若干种情况同理，考虑就地计算 $b$、并且使用 `Bitset` 记录 $w = 32$ 个二进制位的 $\delta$ 并且就地还原 $\Delta$，对于 $2^i \geq n$ 的情况，我们只需要维护一行长度为 $2n$ 的 $\delta$ 数组即可，这样空间消耗大致为
$1 + 4 + 8 = 13\ \mathrm{MiB} < 16\ \mathrm{MiB}$，实际上并不能跑满.

该算法时间复杂度为 $\mathcal{O}(n \log a_i)$，空间复杂度为 $\mathcal{O}(n)$.

# Counting triangles

注意到一个神奇的性质：每个三角形要么同色，要么有两边同色另一边异色。对于后者，三角形有恰有两个异色角，而前者没有异色角。

因此异色角数/2 即为不符合条件的三角个数。用总数减去即可。复杂度 $O(n^2)$。