2021 CCPC (哈尔滨) 试题分析

Fudan University

November 28, 2021

A. So Many Lucky Strings

考虑 dp 算法。我们令 n 表示字符串个数，m 表示字符串总长度。

A. So Many Lucky Strings

考虑 dp 算法。我们令 n 表示字符串个数，m 表示字符串总长度。

考虑回文串从两边向中间构造，令 $f(l, r, p)$ 表示左边最后选到了 s_l，右边最后选到了 s_r，同时 s_r 多出来了一个长度为 p 的前缀的方案数。

A. So Many Lucky Strings

考虑 dp 算法。我们令 n 表示字符串个数，m 表示字符串总长度。

考虑回文串从两边向中间构造，令 $f(l, r, p)$ 表示左边最后选到了 s_l，右边最后选到了 s_r，同时 s_r 多出来了一个长度为 p 的前缀的方案数。

类似的，令 $g(l, r, p)$ 表示左边最后选到了 s_l，右边最后选到了 s_r，同时 s_l 多出来了一个长度为 p 的后缀的方案数。

A. So Many Lucky Strings

考虑 dp 算法。我们令 n 表示字符串个数，m 表示字符串总长度。

考虑回文串从两边向中间构造，令 $f(l, r, p)$ 表示左边最后选到了 s_l，右边最后选到了 s_r，同时 s_r 多出来了一个长度为 p 的前缀的方案数。

类似的，令 $g(l, r, p)$ 表示左边最后选到了 s_l，右边最后选到了 s_r，同时 s_l 多出来了一个长度为 p 的后缀的方案数。

所谓的多出来指的是：左右两边选的串相互回文，最后某一侧剩余出来的部分。

A. So Many Lucky Strings

考虑如何统计答案，因为回文串是 $s_{i_1} + s_{i_2} + ... + s_{i_k}$ 组成的，每个回文串有唯一的中心，考虑四种情况：

A. So Many Lucky Strings

考虑如何统计答案，因为回文串是 $s_{i_1} + s_{i_2} + ... + s_{i_k}$ 组成的，每个回文串有唯一的中心，考虑四种情况：

1. 中心在两个串之间。这样的答案贡献其实就是枚举 l, r，也就是中心两侧的两个串的序号，然后把 $f(l, r, 0)$ 求和即可。

A. So Many Lucky Strings

考虑如何统计答案，因为回文串是 $s_{i_1} + s_{i_2} + ... + s_{i_k}$ 组成的，每个回文串有唯一的中心，考虑四种情况：

1. 中心在两个串之间。这样的答案贡献其实就是枚举 l, r，也就是中心两侧的两个串的序号，然后把 $f(l, r, 0)$ 求和即可。2. 中心在某个串上，但是该中心不是该子串的中心。这种情况下，答案是把满足对应剩余的前/后缀 p 是回文串的 $f(l, r, p)$ 以及 $g(l, r, p)$ 进行求和。注意，p 表示的剩余部分不能为整个串。

A. So Many Lucky Strings

考虑如何统计答案，因为回文串是 $s_{i_1} + s_{i_2} + ... + s_{i_k}$ 组成的，每个回文串有唯一的中心，考虑四种情况：

1. 中心在两个串之间。这样的答案贡献其实就是枚举 l, r，也就是中心两侧的两个串的序号，然后把 $f(l, r, 0)$ 求和即可。2. 中心在某个串上，但是该中心不是该子串的中心。这种情况下，答案是把满足对应剩余的前/后缀 p 是回文串的 $f(l, r, p)$ 以及 $g(l, r, p)$ 进行求和。注意，p 表示的剩余部分不能为整个串。3. 中心在某个串上，并且该中心也是该子串的中心。这种情况要特殊统计，如果和上面一种情况一起统计会出现重复。只需要对 s_r 是回文串的 $f(l, r, |s_r|)$ 进行统计即可。

A. So Many Lucky Strings

考虑如何统计答案，因为回文串是 $s_{i_1} + s_{i_2} + ... + s_{i_k}$ 组成的，每个回文串有唯一的中心，考虑四种情况：

1. 中心在两个串之间。这样的答案贡献其实就是枚举 l, r，也就是中心两侧的两个串的序号，然后把 $f(l, r, 0)$ 求和即可。2. 中心在某个串上，但是该中心不是该子串的中心。这种情况下，答案是把满足对应剩余的前/后缀 p 是回文串的 $f(l, r, p)$ 以及 $g(l, r, p)$ 进行求和。注意，p 表示的剩余部分不能为整个串。3. 中心在某个串上，并且该中心也是该子串的中心。这种情况要特殊统计，如果和上面一种情况一起统计会出现重复。只需要对 s_r 是回文串的 $f(l, r, |s_r|)$ 进行统计即可。4. 上面的情况考虑的答案都至少是两个串拼起来的，因此最后，考虑单个串回文的情况。

A. So Many Lucky Strings

考虑转移，以 $f(l, r, p)$ 为例，$|s_r| - p$ 是 s_r 串已经匹配的部分的长度，并且 s_l 完全匹配。

A. So Many Lucky Strings

考虑转移，以 $f(l, r, p)$ 为例，$|s_r| - p$ 是 s_r 串已经匹配的部分的长度，并且 s_l 完全匹配。

- 如果 $|s_r| - p \geq |s_l|$ ，说明如果去掉 s_l ，s_r 仍有长度为 $p + |s_l|$ 的前缀，此时如果 s_l 能和 s_r 的第 p 个字符之后的一段匹配上，就能从 $f(i, r, p + |s_l|), (i < l)$ 转移过来。

A. So Many Lucky Strings

考虑转移，以 $f(l, r, p)$ 为例，$|s_r| - p$ 是 s_r 串已经匹配的部分的长度，并且 s_l 完全匹配。

- 如果 $|s_r| - p \geq |s_l|$ ，说明如果去掉 s_l ，s_r 仍有长度为 $p + |s_l|$ 的前缀，此时如果 s_l 能和 s_r 的第 p 个字符之后的一段匹配上，就能从 $f(i, r, p + |s_l|), (i < l)$ 转移过来。
- 类似地，如果 $|s_r| - p < |s_l|$ ，且对应的串能匹配上，就能从 $g(l, j, |s_r| - p), (j > r)$ 转移过来。

A. So Many Lucky Strings

考虑转移，以 $f(l, r, p)$ 为例，$|s_r| - p$ 是 s_r 串已经匹配的部分的长度，并且 s_l 完全匹配。

- 如果 $|s_r| - p \geq |s_l|$ ，说明如果去掉 s_l ，s_r 仍有长度为 $p + |s_l|$ 的前缀，此时如果 s_l 能和 s_r 的第 p 个字符之后的一段匹配上，就能从 $f(i, r, p + |s_l|), (i < l)$ 转移过来。
- 类似地，如果 $|s_r| - p < |s_l|$ ，且对应的串能匹配上，就能从 $g(l, j, |s_r| - p), (j > r)$ 转移过来。

用前缀和优化可以把转移做到 $O(1)$。此外串的匹配可以用 KMP 做，判断一个串是否回文可以用 Manacher 做。

A. So Many Lucky Strings

考虑转移，以 $f(l, r, p)$ 为例，$|s_r| - p$ 是 s_r 串已经匹配的部分的长度，并且 s_l 完全匹配。

- 如果 $|s_r| - p \geq |s_l|$ ，说明如果去掉 s_l ，s_r 仍有长度为 $p + |s_l|$ 的前缀，此时如果 s_l 能和 s_r 的第 p 个字符之后的一段匹配上，就能从 $f(i, r, p + |s_l|), (i < l)$ 转移过来。
- 类似地，如果 $|s_r| - p < |s_l|$ ，且对应的串能匹配上，就能从 $g(l, j, |s_r| - p), (j > r)$ 转移过来。

用前缀和优化可以把转移做到 $O(1)$。此外串的匹配可以用 KMP 做，判断一个串是否回文可以用 Manacher 做。

时间复杂度：$O(nm)$。

B. Magical Subsequence

做法很多，下面说一下标程的做法。

B. Magical Subsequence

做法很多，下面说一下标程的做法。

记 $Dp[s][i]$ 表示两两相加的和都为 s 的情况下，前 i 个数的最长 Magical 序列长度。那么转移就是 $Dp[s][i] = \max(Dp[s][i-1], Dp[s][Last[i][s-A[i]]+2)$，其中 $Last[i][x]$ 表示 i 之前第一个 x 所在的位置。

B. Magical Subsequence

做法很多，下面说一下标程的做法。

记 $Dp[s][i]$ 表示两两相加的和都为 s 的情况下，前 i 个数的最长 Magical 序列长度。那么转移就是 $Dp[s][i] = \max(Dp[s][i-1], Dp[s][Last[i][s-A[i]]+2)$，其中 $Last[i][x]$ 表示 i 之前第一个 x 所在的位置。

时间复杂度：$O(nA_i)$。

C. Colorful Tree

首先染色可以从根往下染，然后考虑一个 $O(n^2)$ 的动态规划做法，记 $dp[i,j]$ 表示在子树 i 中，所有的叶子被 i 的祖先节点预先染成颜色 j 的前提下，染好里面所有的叶子的颜色的最少操作次数。特别的，记 $dp[i,0]$ 表示子树 i 没有被祖先节点预染色的最少操作次数。

这样对于一个非叶子节点，首先将所有子树的 dp 值聚合起来，然后再决定在这个点是被提前染成什么颜色。转移如下：

C. Colorful Tree

- $$dp[i,j] = \sum_{u} \min(dp[u,j], dp[u,0])$$

，其中 u 为 i 的所有子节点，j 从 0 到 n

C. Colorful Tree

- $$dp[i,j] = \sum_{u} \min(dp[u,j], dp[u,0])$$

  ，其中 u 为 i 的所有子节点，j 从 0 到 n
- 然后考虑提前在当前点染色，即

$$dp[i,0] = \min_{1 \le k \le n} (dp[i,0], dp[i,k] + 1)$$

C. Colorful Tree

- $$dp[i,j] = \sum_{u} \min(dp[u,j], dp[u,0])$$

  ，其中 u 为 i 的所有子节点，j 从 0 到 n
- 然后考虑提前在当前点染色，即

$$dp[i,0] = \min_{1 \le k \le n} (dp[i,0], dp[i,k] + 1)$$

最后的答案是 $dp[root, 0]$。

C. Colorful Tree

注意到转移的时候根的 dp 数组和子树的区别不大，并且在保留一个颜色时，保留子树外面的颜色是没有意义的。可以在节点 i 用一个 map 表示 $dp[i,*]$。在聚合子树 dp 值时，启发式合并把小的 map 暴力插入到大的 map 里面就可以维护所有的 $dp[i,j],(j\neq 0)$。

C. Colorful Tree

注意到转移的时候根的 dp 数组和子树的区别不大，并且在保留一个颜色时，保留子树外面的颜色是没有意义的。可以在节点 i 用一个 map 表示 $dp[i,*]$。在聚合子树 dp 值时，启发式合并把小的 map 暴力插入到大的 map 里面就可以维护所有的 $dp[i,j],(j\neq 0)$。合并 map 时候颜色 j 如果是在小的 map A 中出现则可以暴力更新；如果不在小的 map A 出现，但是在大的 map B 中出现，就需要 $B[j]\leftarrow B[j]+A[0]$。这一步暴力实现会导致复杂度变成 $O(n^2)$，所以需要对 map 附加维护一个懒标记进行全局加法。

C. Colorful Tree

注意到转移的时候根的 dp 数组和子树的区别不大，并且在保留一个颜色时，保留子树外面的颜色是没有意义的。可以在节点 i 用一个 map 表示 $dp[i,*]$。在聚合子树 dp 值时，启发式合并把小的 map 暴力插入到大的 map 里面就可以维护所有的 $dp[i,j],(j\neq 0)$。合并 map 时候颜色 j 如果是在小的 map A 中出现则可以暴力更新；如果不在小的 map A 出现，但是在大的 map B 中出现，就需要 $B[j]\leftarrow B[j]+A[0]$。这一步暴力实现会导致复杂度变成 $O(n^2)$，所以需要对 map 附加维护一个懒标记进行全局加法。

时间复杂度：$O(n\log^2 n)$，用 DSU on tree 可做到 $O(n\log n)$。

D. Math master

数字最多 $w = 19$ 位，可以 $O(2^w)$ 暴力枚举所有的数字组合，然后 $O(w)$ 检验即可。

D. Math master

数字最多 $w = 19$ 位，可以 $O(2^w)$ 暴力枚举所有的数字组合，然后 $O(w)$ 检验即可。

时间复杂度：$O(Tw2^w)$。

E. Power and Modulo

一般来说 $A_1 = 1$，然后大概思路就是找到 $2A_i \neq A_{i+1}$ 的位置，这样可以确定 $M = 2A_i - A_{i+1}$，如果这样的 M 合法且唯一，再对每个元素判一下是否符合要求即可。

E. Power and Modulo

一般来说 $A_1 = 1$，然后大概思路就是找到 $2A_i \neq A_{i+1}$ 的位置，这样可以确定 $M = 2A_i - A_{i+1}$，如果这样的 M 合法且唯一，再对每个元素判一下是否符合要求即可。

然后对于 $A_1 = 0$ 的情况，此时 M 只能等于 1，判一下其他的 A_i 是否都是 0 即可。

E. Power and Modulo

一般来说 $A_1 = 1$，然后大概思路就是找到 $2A_i \neq A_{i+1}$ 的位置，这样可以确定 $M = 2A_i - A_{i+1}$，如果这样的 M 合法且唯一，再对每个元素判一下是否符合要求即可。

然后对于 $A_1 = 0$ 的情况，此时 M 只能等于 1，判一下其他的 A_i 是否都是 0 即可。

否则如果 $A_1 > 1$ 则直接输出 “-1”。

E. Power and Modulo

一般来说 $A_1 = 1$，然后大概思路就是找到 $2A_i \neq A_{i+1}$ 的位置，这样可以确定 $M = 2A_i - A_{i+1}$，如果这样的 M 合法且唯一，再对每个元素判一下是否符合要求即可。

然后对于 $A_1 = 0$ 的情况，此时 M 只能等于 1，判一下其他的 A_i 是否都是 0 即可。

否则如果 $A_1 > 1$ 则直接输出 “-1”。

时间复杂度：$O(\sum n)$。

F. Master Spark

已知有两种做法。

F. Master Spark

已知有两种做法。

一种是对每条 Spark 边界线，看其他 Spark 是否覆盖掉了它在游戏区域内的所有区段，如果没有覆盖完，则说明有安定点，这样就是一个线段覆盖问题。

F. Master Spark

已知有两种做法。

一种是对每条 Spark 边界线，看其他 Spark 是否覆盖掉了它在游戏区域内的所有区段，如果没有覆盖完，则说明有安定点，这样就是一个线段覆盖问题。

另外一种就是把所有交点求出来，然后扫描线 + 点事件来维护所有 Spark 边界线的上下位置，然后维护一个前缀和之类的来看中间是否有空隙。

F. Master Spark

已知有两种做法。

一种是对每条 Spark 边界线，看其他 Spark 是否覆盖掉了它在游戏区域内的所有区段，如果没有覆盖完，则说明有安定点，这样就是一个线段覆盖问题。

另外一种就是把所有交点求出来，然后扫描线 + 点事件来维护所有 Spark 边界线的上下位置，然后维护一个前缀和之类的来看中间是否有空隙。

时间复杂度：$O(n^2 \log n)$。

G. Damaged Bicycle

首先求出每辆单车到其他所有点的最短路，然后考虑根据“是否访问过每辆单车”进行状压 DP。

G. Damaged Bicycle

首先求出每辆单车到其他所有点的最短路，然后考虑根据“是否访问过每辆单车” 进行状压 DP。

设 $Pro[S]$ 为 S 集合内的单车都是坏的的概率，$Dp[S][i]$ 为访问过 S 集合内所有单车，最后停在 i 号单车上的最少期望时间，其中有可能中途有好单车就直接骑着走了，也有可能所有单车都是坏的最后真还在 i 号点。

G. Damaged Bicycle

首先求出每辆单车到其他所有点的最短路，然后考虑根据“是否访问过每辆单车” 进行状压 DP。

设 $Pro[S]$ 为 S 集合内的单车都是坏的的概率，$Dp[S][i]$ 为访问过 S 集合内所有单车，最后停在 i 号单车上的最少期望时间，其中有可能中途有好单车就直接骑着走了，也有可能所有单车都是坏的最后真还在 i 号点。转移的话枚举 i 前面一辆单车 j，然后有三种情况：在访问 i 之前就找到了好单车、i 是第一辆好单车、i 还是坏的。

G. Damaged Bicycle

首先求出每辆单车到其他所有点的最短路，然后考虑根据“是否访问过每辆单车” 进行状压 DP。

设 $Pro[S]$ 为 S 集合内的单车都是坏的的概率，$Dp[S][i]$ 为访问过 S 集合内所有单车，最后停在 i 号单车上的最少期望时间，其中有可能中途有好单车就直接骑着走了，也有可能所有单车都是坏的最后真还在 i 号点。转移的话枚举 i 前面一辆单车 j，然后有三种情况：在访问 i 之前就找到了好单车、i 是第一辆好单车、i 还是坏的。具体转移式这里就不赘述了。

G. Damaged Bicycle

首先求出每辆单车到其他所有点的最短路，然后考虑根据“是否访问过每辆单车” 进行状压 DP。

设 $Pro[S]$ 为 S 集合内的单车都是坏的的概率，$Dp[S][i]$ 为访问过 S 集合内所有单车，最后停在 i 号单车上的最少期望时间，其中有可能中途有好单车就直接骑着走了，也有可能所有单车都是坏的最后真还在 i 号点。转移的话枚举 i 前面一辆单车 j，然后有三种情况：在访问 i 之前就找到了好单车、i 是第一辆好单车、i 还是坏的。具体转移式这里就不赘述了。

最后答案即为 $\min\{Dp[s][i] + Pro[s] \times \frac{Dist(i,n)}{t}\}$。

G. Damaged Bicycle

首先求出每辆单车到其他所有点的最短路，然后考虑根据“是否访问过每辆单车” 进行状压 DP。

设 $Pro[S]$ 为 S 集合内的单车都是坏的的概率，$Dp[S][i]$ 为访问过 S 集合内所有单车，最后停在 i 号单车上的最少期望时间，其中有可能中途有好单车就直接骑着走了，也有可能所有单车都是坏的最后真还在 i 号点。转移的话枚举 i 前面一辆单车 j，然后有三种情况：在访问 i 之前就找到了好单车、i 是第一辆好单车、i 还是坏的。具体转移式这里就不赘述了。

最后答案即为 $\min\{Dp[s][i] + Pro[s] \times \frac{Dist(i,n)}{t}\}$。

时间复杂度：$O(m + nk\log m + k^2 2^k)$。

H. What logic for?

首先考虑把串按照下标对 k 取模的结果分组，即 $\{\{S_1, S_{k+1}, \cdots\}, \{S_2, S_{k+2}, \cdots\}, \cdots, \{S_k, S_{2k}, \cdots\}\}$ 以及 $\{\{T_1, T_{k+1}, \cdots\}, \{T_2, T_{k+2}, \cdots\}, \cdots, \{T_k, T_{2k}, \cdots\}\}$。如果满足以下条件，则答案是 TAK，否则是 NIE：

H. What logic for?

首先考虑把串按照下标对 k 取模的结果分组，即 $\{\{S_1, S_{k+1}, \cdots\}, \{S_2, S_{k+2}, \cdots\}, \cdots, \{S_k, S_{2k}, \cdots\}\}$ 以及 $\{\{T_1, T_{k+1}, \cdots\}, \{T_2, T_{k+2}, \cdots\}, \cdots, \{T_k, T_{2k}, \cdots\}\}$。如果满足以下条件，则答案是 TAK，否则是 NIE：

- 对于每个 i，$\{S_i, S_{k+i}, \cdots\}$ 和 $\{T_i, T_{k+i}, \cdots\}$ 构成的可重集都相等

H. What logic for?

首先考虑把串按照下标对 k 取模的结果分组，即 $\{\{S_1, S_{k+1}, \cdots\}, \{S_2, S_{k+2}, \cdots\}, \cdots, \{S_k, S_{2k}, \cdots\}\}$ 以及 $\{\{T_1, T_{k+1}, \cdots\}, \{T_2, T_{k+2}, \cdots\}, \cdots, \{T_k, T_{2k}, \cdots\}\}$。如果满足以下条件，则答案是 TAK，否则是 NIE：

- 对于每个 i，$\{S_i, S_{k+i}, \cdots\}$ 和 $\{T_i, T_{k+i}, \cdots\}$ 构成的可重集都相等
- 不存在一组 i, j，使得 $\{S_i, S_{k+i}, \cdots\}$ 和 $\{S_j, S_{k+j}, \cdots\}$ 分别没有重复元素，且其中一个的逆序对奇偶性与 T 对应的那一组串的逆序对奇偶性相同，另一个的逆序对奇偶性与 T 对应的那一组串的逆序对奇偶性不同

H. What logic for?

关于第一点，因为无论怎么操作，元素都不会跨组交换，所以这是比较显然的必要条件。

H. What logic for?

关于第一点，因为无论怎么操作，元素都不会跨组交换，所以这是比较显然的必要条件。

关于第二点，因为每次操作，我们都会同时改变所有组的逆序对奇偶性（如果没有重复元素），所以这也是个必要条件。

H. What logic for?

关于第一点，因为无论怎么操作，元素都不会跨组交换，所以这是比较显然的必要条件。

关于第二点，因为每次操作，我们都会同时改变所有组的逆序对奇偶性（如果没有重复元素），所以这也是个必要条件。

接下来考虑充分性。不妨设所有组的逆序对奇偶性都对应相同，且都是偶数，那么如果只要能把 S 和 T 的每一组都还原成形态唯一的非降序列，再根据操作的可逆性，便可以求得 S 变到 T 的一组方案。

H. What logic for?

关于第一点，因为无论怎么操作，元素都不会跨组交换，所以这是比较显然的必要条件。

关于第二点，因为每次操作，我们都会同时改变所有组的逆序对奇偶性（如果没有重复元素），所以这也是个必要条件。

接下来考虑充分性。不妨设所有组的逆序对奇偶性都对应相同，且都是偶数，那么如果只要能把 S 和 T 的每一组都还原成形态唯一的非降序列，再根据操作的可逆性，便可以求得 S 变到 T 的一组方案。

下面关于如何还原进行讨论：

H. What logic for?

如果某一组的长度小于等于 2，因为逆序对奇偶性是偶数，则其必然是不降的。

H. What logic for?

如果某一组的长度小于等于 2，因为逆序对奇偶性是偶数，则其必然是不降的。否则不妨设这一组是 i，那么考虑连续进行 $a = uk + i$ 和 $a = uk + i + 1$ 的两次操作，我们便可以把 $S_{uk+i}, S_{(u+1)k+i}, S_{(u+2)k+i}$ 这三个元素进行一次轮转，且不改动其他的元素。

H. What logic for?

如果某一组的长度小于等于 2，因为逆序对奇偶性是偶数，则其必然是不降的。否则不妨设这一组是 i，那么考虑连续进行 $a = uk + i$ 和 $a = uk + i + 1$ 的两次操作，我们便可以把 $S_{uk+i}, S_{(u+1)k+i}, S_{(u+2)k+i}$ 这三个元素进行一次轮转，且不改动其他的元素。基于上述操作，我们可以用冒泡排序的思路把它还原成非降序列，最后最大的两个元素一定也是符合非降顺序的，否则与逆序对奇偶性为偶数相矛盾。

H. What logic for?

如果某一组的长度小于等于 2，因为逆序对奇偶性是偶数，则其必然是不降的。否则不妨设这一组是 i，那么考虑连续进行 $a = uk + i$ 和 $a = uk + i + 1$ 的两次操作，我们便可以把 $S_{uk+i}, S_{(u+1)k+i}, S_{(u+2)k+i}$ 这三个元素进行一次轮转，且不改动其他的元素。基于上述操作，我们可以用冒泡排序的思路把它还原成非降序列，最后最大的两个元素一定也是符合非降顺序的，否则与逆序对奇偶性为偶数相矛盾。如果有重复元素，可以给重复的元素强制规定顺序，这里因为可以通过标顺序来调整逆序对的奇偶性所以无论如何都是有解的。

H. What logic for?

如果某一组的长度小于等于 2，因为逆序对奇偶性是偶数，则其必然是不降的。否则不妨设这一组是 i，那么考虑连续进行 $a = uk + i$ 和 $a = uk + i + 1$ 的两次操作，我们便可以把 $S_{uk+i}, S_{(u+1)k+i}, S_{(u+2)k+i}$ 这三个元素进行一次轮转，且不改动其他的元素。基于上述操作，我们可以用冒泡排序的思路把它还原成非降序列，最后最大的两个元素一定也是符合非降顺序的，否则与逆序对奇偶性为偶数相矛盾。如果有重复元素，可以给重复的元素强制规定顺序，这里因为可以通过标顺序来调整逆序对的奇偶性所以无论如何都是有解的。

时间复杂度：$O(\sum((l_S + l_T)\log(l_S + l_T)))$

I. Power and Zero

做法很多，说一种基于二分答案的做法。

I. Power and Zero

做法很多，说一种基于二分答案的做法。

首先对于每个二进制位 i，记录有多少个数在第 i 位是 1，记作 $Cnt[i]$。

I. Power and Zero

做法很多，说一种基于二分答案的做法。

首先对于每个二进制位 i，记录有多少个数在第 i 位是 1，记作 $Cnt[i]$。然后答案一定是 $Cnt[0] + 2x$，即奇数的个数加 $2x$，因为如果某一次操作把一个偶数减去 1，那么后续肯定需要再对其进行一次减 1 的操作。

I. Power and Zero

做法很多，说一种基于二分答案的做法。

首先对于每个二进制位 i，记录有多少个数在第 i 位是 1，记作 $Cnt[i]$。然后答案一定是 $Cnt[0]+2x$，即奇数的个数加 $2x$，因为如果某一次操作把一个偶数减去 1，那么后续肯定需要再对其进行一次减 1 的操作。

所以我们二分这个 x，检验答案的核心在于是否能调整这个 Cnt 数组，使得所有 $Cnt[i] \le Cnt[0]+2x$ 且 $Cnt[i] \ge Cnt[i+1]$。

I. Power and Zero

做法很多，说一种基于二分答案的做法。

首先对于每个二进制位 i，记录有多少个数在第 i 位是 1，记作 $Cnt[i]$。然后答案一定是 $Cnt[0]+2x$，即奇数的个数加 $2x$，因为如果某一次操作把一个偶数减去 1，那么后续肯定需要再对其进行一次减 1 的操作。

所以我们二分这个 x，检验答案的核心在于是否能调整这个 Cnt 数组，使得所有 $Cnt[i] \le Cnt[0]+2x$ 且 $Cnt[i] \ge Cnt[i+1]$。其中 $Cnt[i] \le Cnt[0]+2x$ 意思就是操作次数不能超过这么多，$Cnt[i] \ge Cnt[i+1]$ 意思是能够直接按照题意去减数字。

I. Power and Zero

做法很多，说一种基于二分答案的做法。

首先对于每个二进制位 i，记录有多少个数在第 i 位是 1，记作 $Cnt[i]$。然后答案一定是 $Cnt[0]+2x$，即奇数的个数加 $2x$，因为如果某一次操作把一个偶数减去 1，那么后续肯定需要再对其进行一次减 1 的操作。

所以我们二分这个 x，检验答案的核心在于是否能调整这个 Cnt 数组，使得所有 $Cnt[i] \le Cnt[0]+2x$ 且 $Cnt[i] \ge Cnt[i+1]$。其中 $Cnt[i] \le Cnt[0]+2x$ 意思就是操作次数不能超过这么多，$Cnt[i] \ge Cnt[i+1]$ 意思是能够直接按照题意去减数字。调整就是把一个大幂次拆成若干小幂次。

I. Power and Zero

初始令 $Cnt[0]+=2x, Cnt[1]-=x$，然后从 1 开始枚举 i，如果 $Cnt[i] > Cnt[i-1]$ 则无解，否则尽可能把 $Cnt[i]$ 调整到小于等于 $Cnt[i-1]$ 最大的数，即 $Cnt[i]+=2t, Cnt[i+1]-=t$，其中 $t=\lfloor \frac{Cnt[i-1]-Cnt[i]}{2} \rfloor$。

I. Power and Zero

初始令 $Cnt[0]+=2x, Cnt[1]-=x$，然后从 1 开始枚举 i，如果 $Cnt[i] > Cnt[i-1]$ 则无解，否则尽可能把 $Cnt[i]$ 调整到小于等于 $Cnt[i-1]$ 最大的数，即 $Cnt[i]+=2t, Cnt[i+1]-=t$，其中 $t=\lfloor \frac{Cnt[i-1]-Cnt[i]}{2} \rfloor$。把某些 $Cnt[i]$ 弄出负数是可以的，因为总可以借若干小幂次凑回大幂次来归零。

I. Power and Zero

初始令 $Cnt[0]+=2x, Cnt[1]-=x$，然后从 1 开始枚举 i，如果 $Cnt[i] > Cnt[i-1]$ 则无解，否则尽可能把 $Cnt[i]$ 调整到小于等于 $Cnt[i-1]$ 最大的数，即 $Cnt[i]+=2t, Cnt[i+1]-=t$，其中 $t=\lfloor\frac{Cnt[i-1]-Cnt[i]}{2}\rfloor$。把某些 $Cnt[i]$ 弄出负数是可以的，因为总可以借若干小幂次凑回大幂次来归零。

时间复杂度：$O(\sum n + T\log^2 A_i)$。

J. Local Minimum

维护每一行/每一列的最小值，再依次枚举每个元素看是否满足要求即可。

J. Local Minimum

维护每一行/每一列的最小值，再依次枚举每个元素看是否满足要求即可。

时间复杂度：$O(nm)$。

K. Wonder Egg Priority

首先对于每一个位置，经过若干次操作之后会变成 $k_p^{k_{p-1}^{\cdots}}$ 的形式，从最底层开始套用拓展欧拉定理，则经过 $O(\log M)$ 层，模数就变成 1 了，所以更上面的数字就没有意义了，暴力的做法就是对于每个位置维护一个 $O(\log M)$ 的幂塔。

K. Wonder Egg Priority

首先对于每一个位置，经过若干次操作之后会变成 $k_p^{k_{p-1}^{\cdots}}$ 的形式，从最底层开始套用拓展欧拉定理，则经过 $O(\log M)$ 层，模数就变成 1 了，所以更上面的数字就没有意义了，暴力的做法就是对于每个位置维护一个 $O(\log M)$ 的幂塔。

进一步分析，对于一段区间，如果对其进行 $O(\log M)$ 次相同的修改操作，最后这段区间所有位置就会等价，我们就可以合并这段区间。

K. Wonder Egg Priority

首先对于每一个位置，经过若干次操作之后会变成 $k_p^{k_{p-1}^{\cdots}}$ 的形式，从最底层开始套用拓展欧拉定理，则经过 $O(\log M)$ 层，模数就变成 1 了，所以更上面的数字就没有意义了，暴力的做法就是对于每个位置维护一个 $O(\log M)$ 的幂塔。

进一步分析，对于一段区间，如果对其进行 $O(\log M)$ 次相同的修改操作，最后这段区间所有位置就会等价，我们就可以合并这段区间。所以考虑给 $n-1$ 个间隔设置势能，一次 $[l,r]$ 的区间修改可以把 $[l,r]$ 内的间隔的势能都减去 1，然后把 $l-1$ 到 l 和 r 到 $r+1$ 这两个间隔的势能加上 $O(\log M)$。

K. Wonder Egg Priority

首先对于每一个位置，经过若干次操作之后会变成 $k_p^{k_{p-1}^{\cdots}}$ 的形式，从最底层开始套用拓展欧拉定理，则经过 $O(\log M)$ 层，模数就变成 1 了，所以更上面的数字就没有意义了，暴力的做法就是对于每个位置维护一个 $O(\log M)$ 的幂塔。

进一步分析，对于一段区间，如果对其进行 $O(\log M)$ 次相同的修改操作，最后这段区间所有位置就会等价，我们就可以合并这段区间。所以考虑给 $n-1$ 个间隔设置势能，一次 $[l,r]$ 的区间修改可以把 $[l,r]$ 内的间隔的势能都减去 1，然后把 $l-1$ 到 l 和 r 到 $r+1$ 这两个间隔的势能加上 $O(\log M)$。可以用线段树加 set 或者 splay 来维护每个间隔的势能，以及合并等价区段。

K. Wonder Egg Priority

首先对于每一个位置，经过若干次操作之后会变成 $k_p^{k_{p-1}^{\cdots}}$ 的形式，从最底层开始套用拓展欧拉定理，则经过 $O(\log M)$ 层，模数就变成 1 了，所以更上面的数字就没有意义了，暴力的做法就是对于每个位置维护一个 $O(\log M)$ 的幂塔。

进一步分析，对于一段区间，如果对其进行 $O(\log M)$ 次相同的修改操作，最后这段区间所有位置就会等价，我们就可以合并这段区间。所以考虑给 $n-1$ 个间隔设置势能，一次 $[l,r]$ 的区间修改可以把 $[l,r]$ 内的间隔的势能都减去 1，然后把 $l-1$ 到 l 和 r 到 $r+1$ 这两个间隔的势能加上 $O(\log M)$。可以用线段树加 set 或者 splay 来维护每个间隔的势能，以及合并等价区段。这部分的复杂度即为 $O(n \log n \log M)$。

K. Wonder Egg Priority

基于以上的分析，我们需要进行 $O(n \log M)$ 次幂塔结果计算，也即需要计算 $O(n \log^2 M)$ 次快速幂，所以我们希望可以快速地计算快速幂。

K. Wonder Egg Priority

基于以上的分析，我们需要进行 $O(n \log M)$ 次幂塔结果计算，也即需要计算 $O(n \log^2 M)$ 次快速幂，所以我们希望可以快速地计算快速幂。这里因为 M 较小，且只会有 $O(\log M)$ 种有用的模数，更关键的是因为欧拉函数每迭代两次就至少会把一个数变成一半。

K. Wonder Egg Priority

基于以上的分析，我们需要进行 $O(n\log M)$ 次幂塔结果计算，也即需要计算 $O(n\log^2 M)$ 次快速幂，所以我们希望可以快速地计算快速幂。这里因为 M 较小，且只会有 $O(\log M)$ 种有用的模数，更关键的是因为欧拉函数每迭代两次就至少会把一个数变成一半。所以考虑对每个模数下的每个数字的幂次结果进行类似 BSGS 的预处理，可知复杂度为：

$$O\left(M^{\frac{3}{2}}+\left(\frac{M}{2}\right)^{\frac{3}{2}}+\cdots\right)=O(M^{\frac{3}{2}})$$

K. Wonder Egg Priority

基于以上的分析，我们需要进行 $O(n\log M)$ 次幂塔结果计算，也即需要计算 $O(n\log^2 M)$ 次快速幂，所以我们希望可以快速地计算快速幂。这里因为 M 较小，且只会有 $O(\log M)$ 种有用的模数，更关键的是因为欧拉函数每迭代两次就至少会把一个数变成一半。所以考虑对每个模数下的每个数字的幂次结果进行类似 BSGS 的预处理，可知复杂度为：

$$O\left(M^{\frac{3}{2}}+\left(\frac{M}{2}\right)^{\frac{3}{2}}+\cdots\right)=O(M^{\frac{3}{2}})$$

这样算一次快速幂就可以做到 $O(1)$ 了。

K. Wonder Egg Priority

基于以上的分析，我们需要进行 $O(n \log M)$ 次幂塔结果计算，也即需要计算 $O(n \log^2 M)$ 次快速幂，所以我们希望可以快速地计算快速幂。这里因为 M 较小，且只会有 $O(\log M)$ 种有用的模数，更关键的是因为欧拉函数每迭代两次就至少会把一个数变成一半。所以考虑对每个模数下的每个数字的幂次结果进行类似 BSGS 的预处理，可知复杂度为：

$$O\left(M^{\frac{3}{2}} + \left(\frac{M}{2}\right)^{\frac{3}{2}} + \cdots\right) = O(M^{\frac{3}{2}})$$

这样算一次快速幂就可以做到 $O(1)$ 了。

时间复杂度：$O(n(\log n + \log M) \log M + M^{\frac{3}{2}})$。

L. Karshilov's Matching Problem

首先建一个关于所有 t 串的 AC 的自动机，然后 dfs 一下 fail 树维护 AC 自动机每个节点的权值，其中每个点的权值即为其 fail 树上所有祖先的权值。

L. Karshilov's Matching Problem

首先建一个关于所有 t 串的 AC 的自动机，然后 dfs 一下 fail 树维护 AC 自动机每个节点的权值，其中每个点的权值即为其 fail 树上所有祖先的权值。

然后维护一个倍增数组 $Go[u][c][i]$，表示 AC 自动机上从 u 节点开始沿着 c 字符走 2^i 步之后所到达的节点，同理记一个 $W[u][c][i]$ 记录所经过的点的权值和。

L. Karshilov's Matching Problem

首先建一个关于所有 t 串的 AC 的自动机，然后 dfs 一下 fail 树维护 AC 自动机每个节点的权值，其中每个点的权值即为其 fail 树上所有祖先的权值。

然后维护一个倍增数组 $Go[u][c][i]$，表示 AC 自动机上从 u 节点开始沿着 c 字符走 2^i 步之后所到达的节点，同理记一个 $W[u][c][i]$ 记录所经过的点的权值和。

这样我们可以维护一个栈，栈里面的元素是一个连续字母段，换字母操作就可以直接暴力维护这个栈，询问操作就先二分找到询问的位置所对应的栈的元素，然后把该元素的字母段长度改一改然后跑一次倍增即可得到当前询问的答案。

L. Karshilov's Matching Problem

首先建一个关于所有 t 串的 AC 的自动机，然后 dfs 一下 fail 树维护 AC 自动机每个节点的权值，其中每个点的权值即为其 fail 树上所有祖先的权值。

然后维护一个倍增数组 $Go[u][c][i]$，表示 AC 自动机上从 u 节点开始沿着 c 字符走 2^i 步之后所到达的节点，同理记一个 $W[u][c][i]$ 记录所经过的点的权值和。

这样我们可以维护一个栈，栈里面的元素是一个连续字母段，换字母操作就可以直接暴力维护这个栈，询问操作就先二分找到询问的位置所对应的栈的元素，然后把该元素的字母段长度改一改然后跑一次倍增即可得到当前询问的答案。

时间复杂度：$O(n + |S| + (m + \sum |t_i|) \log (\sum |t_i|))$。

谢谢大家！