Problema 2

Link
https://codeforces.com/problemset/problem/1777/F

Definicion
Se te da un arreglo `a` que consiste en `n` enteros no negativos.

La "apatía" de un subarreglo $a_l, a_{l+1}, ..., a_r$ se define como

$$\max(a_l, a_{l+1}, ..., a_r) \oplus (a_l \oplus a_{l+1} \oplus ... \oplus a_r)$$

donde $\oplus$ denota la operación XOR bit a bit.

Encuentra la apatía máxima entre todos los subarreglos.

Conceptos claves

Prefijo XOR
El prefijo XOR se define como la operacion XOR a todos los elementos desde el inicio de un array `a` hasta un indice `i`. Formalmente queda

$$\text{prefix}[i] = a_1 \oplus a_2 \oplus ... \oplus a_i$$

La funcion XOR possee propiedades conmutativa, transitiva y distributiva, y no solo eso, sino también posee la propiedad de que $a \oplus a = 0$. Lo cual hace esta operacion muy maleable a la hora de tratar con el prefijo ya que dado `l` y `r` enteros donde `l < r`

$$\text{XOR}(a_1, a_2, ..., a_l) = \text{prefix}[l]$$

$$\text{XOR}(a_1, a_2, ..., a_l, a_{l+1}, ..., a_r) = \text{prefix}[r]$$

$$\text{Y como } a_i \oplus a_i = 0$$

$$\therefore \text{XOR}(a_l, a_{l+1}, ..., a_r) = \text{prefix}[r] \oplus \text{prefix}[l-1]$$

Por tanto hallar el XOR de un subarreglo es equivalente a hallar el XOR de los prefijos de los indices `l` y `r` y hacer la operacion XOR entre ellos.

Trie
El Trie es una estructura de datos que permite almacenar y buscar cadenas de strings y en este case en concreto secuencias de bits de manera eficiente. En este caso, cada valor de prefijo XOR puede verse como una secuencia de bits, lo que convierte al Trie en una herramienta adecuada para buscar combinaciones óptimas de XOR entre subarreglos.

Solucion usando DP
Voy primero a describir la solucion y despues mas adelante por que es correcta a profundidad.

Pasos del Algoritmo:

1. **Inicializar el Arreglo de Trie**:
   - El arreglo Trie `t` se inicializa de tal manera que para cada índice `i`, almacena los valores de prefijo XOR hasta el índice `i-1`.
   - Esta parte es crucial porque nos permite calcular eficientemente el XOR de subarreglos que terminan antes o en el índice `i`. La idea clave es que el XOR de cualquier subarreglo puede derivarse de la diferencia entre los prefijos XOR.

- La razón por la que insertamos el prefijo XOR hasta `i-1` es que, al calcular la "apatía" de los subarreglos que comienzan en o después del índice `i`, queremos comparar el prefijo XOR actual con los de los subarreglos anteriores, demostrado anteriormente.

2. **Ordenar los Elementos en Orden Ascendente**:
   - Ordenar los elementos por valor se hace para procesar los subarreglos en un orden específico. Al ordenar, procesamos primero los elementos más pequeños, asegurando que podamos usar técnicas de programación dinámica (como la fusión de Tries) de manera más eficiente. Esto es similar al **algoritmo de Mo**, donde ordenar las consultas ayuda a reducir el reprocesamiento innecesario.
   - Procesar primero los elementos más pequeños garantiza que cuando procesamos un elemento más grande, ya tenemos los prefijos XOR de los subarreglos dominados por elementos más pequeños. Esto nos permite manejar cada elemento una sola vez y calcular la "apatía" de manera eficiente.
3. **Calcular los Límites Izquierdo y Derecho Usando una Pila Monótona**:
   - Por cada elemento del arreglo, usamos una pila monotona para encontrar los elementos más cercanos a la izquierda y a la derecha que son mayores que el elemento actual. Esto ayuda a determinar el rango de subarreglos donde el elemento actual es el máximo.
   - Específicamente:
     - El límite izquierdo es el elemento más grande más cercano a la izquierda, por lo que cualquier subarreglo que contenga el elemento actual como máximo solo puede extenderse a la izquierda hasta este límite.
     - El límite derecho es el elemento más grande más cercano a la derecha, por lo que el subarreglo solo puede extenderse a la derecha hasta este límite.
   - Esto reduce el espacio de búsqueda para cada elemento, asegurando que solo calculemos la "apatía" para subarreglos donde el elemento actual está garantizado como máximo.
4. **Elegir el Subarreglo Más Pequeño entre el Izquierdo y el Derecho**:
   - Para cada valor (en el arreglo ordenado), se verifica si la mitad izquierda o la derecha del subarreglo (definida por los límites izquierdo y derecho) es más pequeña.
   - Por qué se hace esto: Esta optimización minimiza el número de operaciones al actualizar el Trie. Siempre trabajamos primero en el subarreglo más pequeño para reducir el número de inserciones y consultas de prefijo XOR.
   - Una vez que determinamos el subarreglo más pequeño, realizamos lo siguiente:
     - Calculamos la "apatía" máxima consultando el Trie opuesto (es decir, si estamos procesando el lado izquierdo, consultamos el lado derecho y viceversa).
     - La consulta usa $\text{prexor}_j \oplus a[x]$, como se mencionó correctamente, para calcular el XOR del subarreglo que incluye `a[x]`.
5. **Fusionar los Tries**:
   - Después de calcular la "apatía" para los subarreglos que incluyen a `a[x]` como el máximo, fusionamos los dos Tries (izquierdo y derecho).
   - El Trie fusionado ahora almacenará los prefijos XOR para ambos subarreglos y permitirá realizar consultas de manera eficiente sobre subarreglos más grandes.
   - Esta fusion se hace desde el Trie mas pequeno al mas grande, para minimizar la cantidad de operaciones de insercion, algo exactamente igual a lo que se hace en un **disjoint set** con sus uniones.

Demostracion

Definición del subproblema

El algoritmo divide el problema en subproblemas donde, para cada elemento a[x] , se calcula la apatia de todos los subarreglos en los que a[x] es el elemento máximo. El arreglo se divide en dos regiones:

- Izquierda de a[x]: Los subarreglos de $a[l], a[l+1], ..., a[x]$.
- Derecha de a[x]: Los subarreglos de $a[x], a[x+1], ..., a[r]$.

Estos subarreglos se resuelven de manera independiente y sus resultados se combinan usando programación dinámica y operaciones basadas en Trie.

Propiedad de subestructura óptima

Para demostrar que el enfoque de programación dinámica es correcto, debemos establecer que el problema exhibe la propiedad de subestructura óptima, es decir, que la solución óptima para todo el problema se puede construir a partir de las soluciones óptimas de sus subproblemas.

1. Caso base:
- Cuando el arreglo tiene tamaño 1 (es decir, solo un elemento), el único subarreglo posible es el propio elemento. En este caso, el elemento máximo y el XOR del subarreglo son ambos a_1 , y la apatia es:

$$\text{apatia} = a_1 \oplus a_1 = 0$$

- Por lo tanto, el caso base se cumple de manera trivial.

2. Caso recursivo:
- Para un subarreglo de longitud n , se debe encontrar la apatia máxima considerando todos los subarreglos que contienen un elemento máximo específico `a[x]` .
- El algoritmo divide el arreglo en dos mitades según la posición de `a[x]` y calcula la apatia de los subarreglos izquierdo y derecho de manera independiente.
- Estos subproblemas se pueden resolver recursivamente aplicando el mismo procedimiento (dividiendo los subarreglos izquierdo y derecho en subarreglos más pequeños). Esto garantiza que todo el problema se resuelva de manera óptima si todos los subarreglos más pequeños se resuelven de manera óptima.

3. Combinando los resultados de los subproblemas:
- Una vez que tenemos las soluciones óptimas para los subarreglos a la izquierda y derecha de `a[x]` , se calcula la apatia máxima para los subarreglos que contienen a `a[x]` consultando el subarreglo opuesto usando el Trie.
- La consulta maximiza el XOR de los subarreglos que involucran ambas mitades y se almacena el resultado.
- Finalmente, fusionamos los Tries de los subarreglos izquierdo y derecho para mantener consultas y procesamiento eficientes de subarreglos más grandes.
- Resultando en que eventualmente hice queries de maxima apatia a todos los subarreglos posibles.

Por lo tanto, se cumple la propiedad de subestructura óptima porque resolver el problema para todo el arreglo se reduce a resolver subarreglos más pequeños de manera óptima y combinar sus resultados de manera eficiente.

Subproblemas superpuestos

En programación dinámica, los subproblemas superpuestos implican que el algoritmo resuelve el mismo subproblema varias veces, pero en lugar de volver a calcularlo, se reutilizan los resultados.

1. Estructura Trie:
- El Trie almacena los valores de XOR de prefijo para los subarreglos que ya se han procesado. Esto permite que el algoritmo reutilice estos valores al calcular la apatia de subarreglos más grandes.
- En lugar de recalcular el XOR para subarreglos superpuestos varias veces, el algoritmo simplemente consulta el Trie, que ya ha almacenado los valores de XOR de prefijo necesarios de cálculos anteriores.

2. Fusión de los Tries:
- Al fusionar el Trie más pequeño en el más grande, el algoritmo asegura que los resultados del subarreglo más pequeño se reutilicen al procesar el subarreglo más grande. Esta reutilización de información evita que el algoritmo resuelva el mismo subproblema varias veces.
- El paso de fusión garantiza que la complejidad computacional siga siendo eficiente, ya que el Trie de cada subarreglo se utiliza para calcular rápidamente el XOR de subarreglos que abarcan ambas mitades.

Por lo tanto, el algoritmo satisface la propiedad de subproblemas superpuestos al reutilizar de manera eficiente los valores de XOR de prefijo almacenados en el Trie, evitando cálculos redundantes.

Inducción matemática para la corrección
Para demostrar la corrección mediante inducción, seguimos estos pasos:

1. Caso base:
- Para un solo elemento (arreglo de tamaño `n = 1`), el algoritmo calcula correctamente la apatia como 0, que es el único resultado válido. Por lo tanto, el caso base se cumple.

2. Hipótesis inductiva:
- Supongamos que el algoritmo calcula correctamente la apatia máxima para todos los arreglos de tamaño `k`, donde `k < n`.

3. Paso inductivo:
- Para un arreglo de tamaño `n`, el algoritmo divide el arreglo en subarreglos a la izquierda y derecha del elemento máximo `a[x]`.
- Según la hipótesis inductiva, asumimos que el algoritmo calcula correctamente la apatia máxima para estos subarreglos más pequeños.
- Luego, el algoritmo combina los resultados de los subarreglos izquierdo y derecho consultando el Trie, que almacena los valores de XOR de prefijo. La corrección del cálculo de XOR está garantizada por las propiedades del XOR de prefijo, y la consulta del Trie asegura que la apatia se maximice.
- Por lo tanto, el algoritmo calcula correctamente la apatia máxima para todo el arreglo al combinar los resultados de los subarreglos más pequeños.

De esta manera, mediante inducción, el algoritmo es correcto para arreglos de todos los tamaños.

Complejidad temporal y eficiencia
La eficiencia del algoritmo es otro aspecto importante:

1. Operaciones en Trie:
- Cada inserción y consulta en el Trie toma `O(log 32)` tiempo, ya que el Trie almacena valores de XOR de prefijo de 32 bits.
- Ordenar los elementos toma `O(n log n)`, y las operaciones en el Trie se realizan `O(n log n)` veces.
- Por lo tanto, la complejidad temporal total es `O(n log n log 32 + n log n) = O(n log n)`.

2. Fusión de los Tries:
- La estrategia de fusión de menor a mayor garantiza que las operaciones en el Trie sigan siendo eficientes incluso cuando los subarreglos crecen en tamaño, evitando cálculos redundantes.

Por lo tanto, el algoritmo es tanto correcto como eficiente, como lo demuestra su adherencia a los principios de la programación dinámica y su complejidad temporal óptima.