EGOI Qualifier 2022 Contest 2 - Solucionario

Federación Olímpica Peruana de Informática

17 de julio de 2022

Tutorial del problema: "Puntos balanceados"

Autor(es): Racsó Galván

Grupo 1 (Complejidad Esperada: $O(n^2)$)

Para resolver este grupo nos basta con iterar sobre todos los subarreglos fijando el limite izquierdo l y a medida que movemos hacia la derecha el límite derecho r vamos contando los puntajes de ambos jugadores.

La complejidad será de $O(n^2)$ por el hecho de fijar ambas variables.

Grupo 2 (Complejidad Esperada: $O(n)$)

Para resolver este grupo debemos definir el aporte de cada `C` como -1 y el de cada `S` como $+2$. Debido a que hay solo un positivo y un negativo, un subarreglo es balanceado solo si tiene suma igual a 0.

Entonces, nuestro problema se reduce a hallar la cantidad de subarreglos con suma igual a 0. Ahora debemos plantear la siguiente observación:

- Todo balance de un subarreglo $[l, r]$ es la diferencia de los prefijos p_r y p_{l-1}, es decir, $sum(l, r) = sum(1, r) - sum(1, l-1)$.

Entonces podemos usar solo las sumas de los prefijos (esto lo obtenemos en $O(n)$ o sobre la marcha en una variable) para fijar cada posición r, hallar la cantidad de posiciones $j = l - 1 < r$ tales que $p_j = p_r$ y agregar 1 a la frecuencia de p_r.

Esto lo podemos calcular con un `std::map<int, int>` debido a que tenemos valores negativos, pero también podemos simplemente almacenar en un arreglo de tamaño $3n + 1$ y mapeamos el valor x a la posición $x + n$ en el arreglo, así podemos consultar y actualizar en $O(1)$.

La complejidad será de $O(n)$.

Tutorial del problema: "Arreglos"

Autor(es): Racsó Galván

Grupo 1 (Complejidad Esperada: $O(nq + m)$)

Para resolver este grupo nos basta con ejecutar cada una de las consultas en $O(n)$ y al final revisar las posiciones brindadas en $O(m)$. Solo es cuestión de implementar adecuadamente lo que se pide.

Grupo 2 (Complejidad Esperada: $O(qm)$))

Para resolver este grupo podemos revertir lo que buscamos: En vez de calcular en qué posición terminará una posición inicial i, calcularemos qué posición inicial i ocupará la posición final j. De esta forma podemos solo evaluar las posiciones que nos interesan.

Ahora, para poder calcular lo que queremos debemos ejecutar las consultas en orden revertido y hacer los siguientes cambios:

- Si la posición actual no está dentro del rango de operación, la ignoramos y continuamos.
- `1 l r`: Si $l < x \leq r$, entonces x se reduce en 1, en caso contrario x se vuelve igual a r.
- `2 l r`: x se vuelve igual a $r + l - x$.

Con lo anterior, podemos calcular la posición inicial correspondiente en $O(q)$, llevándonos a una complejidad total de $O(qm)$.

Tutorial del problema: "Paquetes costosos"

Autor(es): Racsó Galván

Grupo 1 (Complejidad Esperada: $O\left(n^2\right)$)

Para resolver este grupo podemos usar Programación Dinámica y definir la función $memo(i)$ que nos dará el menor costo de empaquetar las primeras i cajas.

Entonces, podemos definir la recursión:

$$memo(i) = \begin{cases} 0 & i = 00 \\ memo(i) = \min\limits_{\substack{1 \le j \le i \\ Unique(i,j)}} \{OR(i,j) + memo(j-1)\} & i > 0 \end{cases}$$

Donde $OR(i,j)$ es el Bitwise OR de $b_i, \ldots, b_j$ y $Unique(i,j)$ es una función booleana que nos da `V` si todos los colores en el rango $[i,j]$ son diferentes y `F` si no.

Si iteramos desde $j = i$ hacia abajo podremos obtener el $OR(i,j)$ sobre la marcha, así como el verificar que los colores del rango analizado sean diferentes (manteniendo un arreglo de visitados).

La complejidad será de $O(n^2)$.

Grupo 2 (Complejidad Esperada: $O\left(n \log n \log MAX\right)$)

Para resolver este grupo podemos definir $leftmost(r)$ como la menor posición j tal que el rango $[j,r]$ contiene colores diferentes. Entonces, queremos probar todos los j en el rango $[leftmost(r), r]$ de manera eficiente. Esto podemos calcularlo en $O(n)$ u $O(n \log MAX)$ usando *two pointers*.

Una observación que debemos realizar es que hay **a lo mucho 20 valores diferentes para el** $OR(i,r)$. Esto se da debido a que cada vez que el Bitwise OR cambia, al menos 1 bit es prendido, y como los valores van hasta 10^6, solo hay 20 bits por prender (si decimos que el máximo es MAX, esto es $O(\log MAX)$).

Por otro lado, si tenemos múltiples candidatos j con el mismo resultado de $OR(j,r)$, siempre nos conviene tomar el que esté más a la izquierda para minimizar el resultado. Si tuviéramos alguna forma de consultar el valor de $OR(j,r)$ de manera eficiente, podríamos usar búsqueda binaria para hallar dicho j.

En este momento viene la estructura de *Sparse Table* a salvarnos, ya que en la función Bitwise OR no importa si hay elementos repetidos (igual dará el mismo resultado), podemos construir el Sparse Table de Bitwise OR en $O(n \log n)$ y consultar el resultado de un rango en $O(1)$.

Ya que fijaremos $O(n)$ posiciones, en las cuales tendremos a lo mucho $O(\log MAX)$ búsquedas binarias, tendremos una complejidad final de $O(n \log n \log MAX)$.

Nota: El tiempo límite estuvo pensado con la idea de evitar de que soluciones que usen estructuras que consulten en $O(\log n)$ no pasen, así que si alguna pasa debió haber estado bien optimizada.

Solución extra: Se puede resolver el problema en $O(n \log MAX \log\log MAX)$ si definimos una tabla $next(i,k)$ que nos diga la máxima posición $j \le i$ tal que el valor b_j tiene el k-ésimo bit prendido, de esta forma podemos obtener los $O(\log MAX)$ resultados, ordenarlos de mayor a menor y así saber los cambios de valor para evitar usar búsqueda binaria.

Tutorial del problema: "Just Kidding"

Autor(es): Racsó Galván

Grupo 1 (Complejidad Esperada: $k = 8$, 4 `O`'s y 4 `I`'s en cada bloque)

Para resolver este grupo podemos plantear que las 4 primeras `I`'s vayan de manera vertical en las dos primeras columnas (2 en cada una para no salirnos del tablero) y usar las 4 `O`'s para rellenar las 8 columnas restantes en las dos primeras filas, de esta forma siempre podremos formar una fila en el primer bloque de piezas.

Grupo 2 (Complejidad Esperada: $k = 7$, Todas las piezas son diferentes en cada bloque)

Para resolver este grupo debemos notar que siempre se puede formar la segunda fila usando las primeras 7 piezas del bloque.

Una forma para lograrlo es usando la pieza `I` en la primera columna de forma vertical y el resto de piezas rotarlas para que ocupen siempre todas las celdas de su anchura en la 2da fila, así tendremos dos tipos de pieza: Las que ocupan 3 celdas y las que ocupan 2. Podemos tranquilamente colocar las que ocupan 3 celdas en el rango $[2, 4]$ como nos convenga y las otras 3 piezas que ocupan 2 celdas las colocamos en las posiciones $5, 7, 9$ para completar la 2da fila sin problema usando el primer bloque de piezas.