Olimpiada Peruana de Informática 2022
Fase 2 - Solucionario

Federación Olímpica Peruana de Informática

13 de febrero de 2022

Tutorial del problema: "Mismos Divisores"

Autor(es): Paul Luyo

El primer paso común para resolver todas las subtareas consiste en verificar que la entrada es consistente. Esto es, debemos verificar que los m números escritos por Pablo son diferentes entre ellos, están entre 1 y n (inclusive) y que poseen la misma cantidad de divisores que la posición correspondiente. En caso esta propiedad no sea satisfecha, la respuesta es 0 automáticamente.

Para implementar la primera parte de la verificación, podemos mantener un arreglo $mark[j]$ que contenga 1 en caso j haya aparecido anteriormente en la entrada y 0 en caso contrario. La complejidad de esta parte en tiempo será $O(m)$ y en memoria $O(n)$. La segunda parte, verificar que cada número tiene la misma cantidad de divisores que su posición depende de la subtarea que deseamos resolver.

Grupo 1 (Complejidad Esperada: $O(n!)$)

En este caso, podemos calcular los divisores de cada número en tiempo $O(\sqrt{n})$ ya que solo basta buscar los divisores $\leq \sqrt{n}$ para encontrar todos los divisores de un número. Aquí estamos usando la propiedad de que si x es divisor de n, entonces $\frac{n}{x}$ también es divisor de n. De esta manera, podemos calcular y almacenar los divisores $d(i)$ para $1 \leq i \leq n$ en tiempo $O(n\sqrt{n})$.

Finalmente, una vez concluida la etapa de verificación, podemos observar que la única posibilidad para la posición 1 es tener escrito el número 1, si este no es el caso porque Pablo escribió otro número, la respuesta es 0. Por lo tanto, solo debemos prestar atención a los restantes $n-1$ números entre 2 y n. En este punto, podemos probar todas las permutaciones de los números restantes (que no hayan sido escritos) y contar qué permutaciones satisfacen que la cantidad de divisores del número escrito coincide con la cantidad de divisores de la posición. Simular todas las permutaciones toma tiempo $O((n-1)!)$ y verificar que una permutación satisface la propiedad será $O(n)$ debido a que podemos acceder a la cantidad de divisores a través del arreglo en tiempo constante. La complejidad total será $O(n!)$ en tiempo y $O(n)$ en memoria.

Grupo 2 (Complejidad Esperada: $O(n\sqrt{n})$)

Nuevamente en esta tarea, calculamos y almacenamos la cantidad de divisores de cada número entre 1 y n como en la subtarea anterior. Así, la complejidad en tiempo será $O(n\sqrt{n})$ y en memoria, $O(n)$.

La diferencia viene de la manera en calcular la respuesta. En efecto, esta vez agruparemos los números que no hayan sido escrito por grupos donde comparten la misma cantidad de divisores. Por ejemplo, el 1 estará solo, los primos en otro grupo y así sucesivamente. Digamos que se forman r grupos, cada uno de tamaños $sz_1, sz_2, \ldots, sz_r$. Entonces, debemos notar que los números en un mismo grupo se pueden repartir de cualquier manera y que para cualquier configuración válida, el número x_k ocupa la posición k si y solo si $d(x_k) = d(k)$[1]. Por lo tanto, la cantidad de configuraciones válidas es equivalente a la cantidad de maneras de repartir los números del mismo grupo entre ellos, lo cual es igual a $sz_1! \times sz_2! \times \cdots \times sz_r!$.

Finalmente, la única dificultad en esta parte sería calcular $sz_1! \times sz_2! \times \cdots \times sz_r!$ módulo 10^9+7. Lo cual puede realizarse precomputando los valores de $factorial[i] \equiv i!\ mod(10^9+7)$ ya que $factorial[i+1] \equiv (factorial[i] * (i+1))\%(10^9+7)$. Por lo tanto, esto se puede realizar en $O(n)$ tiempo adicional y memoria adicional. En total, la complejidad en tiempo será $O(n\sqrt{n})$ y en memoria, $O(n)$.

Grupo 3 (Complejidad Esperada: $O(n\log(n))$)

En esta subtarea, se puede terminar como en la anterior. Es decir, calculando $sz_1! \times sz_2! \times \cdots \times sz_r!$ en tiempo y memoria $O(n)$. La única mejora será en el cálculo de la cantidad de divisores.

Para ello, implementamos un algoritmo inspirado en la criba de Erastótenes que permite calcular los valores de $d(\cdot)$ a través de un solo procedimiento en lugar de hacerlo individualmente para cada número. En efecto, inicializamos un arreglo $d[i] = 0$ que contendrá la cantidad de divisores del número i. Luego, para cada $1 \leq i \leq n$, recorremos los números $i, 2i, \ldots, \lfloor \frac{n}{i} \rfloor \cdot i$ y vamos incrementando los valores de $d[i], d[2i], \ldots, d[\lfloor \frac{n}{i} \rfloor \cdot i]$ en 1. La complejidad en tiempo será $n + \frac{n}{2} + \cdots + \frac{n}{n} = n\left(1 + \frac{1}{2} + \cdots + \frac{1}{n}\right) \sim n\log(n)$.

[1] $d(\cdot)$ denota la función cantidad de divisores.

Tutorial del problema: "Camino de cadena"

Autor(es): Racsó Galván

Grupo 1 (Complejidad Esperada: $O\left(n^4\right)$)

Para resolver este subtask debemos notar que siempre se puede formar un camino de cadena A tal que $|A_i| = |A_{i+1}| + 1$ para todo i válido. Esto nos permite entender que la máxima longitud de un camino de cadena es $O(\sqrt{n})$.

Gracias a lo anterior, podemos definir $f(i, j)$ como una matriz booleana que nos dará 1 si existe un camino de cadena A tal que A_1 empieza en la posición i y la longitud del camino es de j y 0 en caso contrario. De esta manera podemos plantear la siguiente recurrencia:

$$f(i, 1) = 1$$

$$f(i, j) = \max_{k=i+j}^{n-j+2}\{f(k, j-1) \cdot I(a[k, k+j-2] \preceq \min\{a[i, i+j-2], a[i+1, i+j-1]\})\}$$

Donde $\preceq$ es una comparación de menor o igual lexicográficamente y $I(P)$ da 1 si la proposición P es verdadera y 0 si es falsa.

Ya que j va hasta $O(\sqrt{n})$, tendremos una complejidad de $O(n^4)$ si implementamos esta idea.

Grupo 2 (Complejidad Esperada: $O\left(n\sqrt{n}\log n\right)$)

Para resolver este subtask debemos enfocarnos en ir avanzando por niveles, es decir, por j ascendente. Esto nos permite ahorrar memoria, ya que nos bastarán dos arreglos de tamaño $O(n)$, los cuales llamaremos *last* y *memo*, de manera que $last[i]$ guarda $f(i, j-1)$ y $memo[i]$ guardará $f(i, j)$.

Una observación importante es que si logramos mapear las subcadenas de una longitud fija L a enteros del 1 al n con una función $\sigma(s)$, podemos realizar la comparación lexicográfica en $O(1)$; es más, si iteramos por i decreciente y llevamos en una variable *minimo* el valor asociado a la menor cadena lexicográfica $a[k, k+j-2]$ tal que $last[k]$ sea 1 y $i + j \leq k$, podemos hallar verificar si $minimo \leq \min\{\sigma(a[i, i+j-2]), \sigma(a[i+1, i+j-1])\}$ y con esto nos bastará para definir el valor de $memo[i]$.

Ahora el problema se reduce a poder hallar los mapeos de manera eficiente; para ello podemos plantear un mapeo que vaya construyendo nivel por nivel (el mapeo del nivel $L = 1$ es simplemente a), de manera que el nivel $(L+1)$ se puede construir en base a los mapeos del nivel L y 1, pues solo tenemos que agregar un valor al final. Por teoría, podemos comparar las cadenas de longitud $L + 1$ como si fueran pares con los mapeos de las longitudes L y 1; es decir, $(a[i], \sigma(s[i+1, i+L]),)$. Esto nos permite obtener los nuevos mapeos ordenando en $O(n \log n)$ las posiciones en función a dichos pares y luego asociar el mapeo correspondiente a cada posición en $O(n)$.

Ya que el j va hasta $O(\sqrt{n})$, realizaremos el proceso de construcción de mapeos en $O(\sqrt{n})$ veces, llevándonos a una complejidad de $O(n\sqrt{n}\log n)$.

Grupo 3 (Complejidad Esperada: $O\left(n\sqrt{n}\right)$)

Para resolver este subtask debemos optimizar el método de ordenamiento del subtask anterior a Radix Sort (pues los valores de los mapeos están entre 1 y n), evitando el factor logarítmico extra y logrando una complejidad de $O(n\sqrt{n})$.

Una solución en $O(n\sqrt{n}\log n)$ pero con constante baja también podría pasar este subtask.

Nota curiosa: Me basé en el problema String Journey de Codeforces para crear este problema, pues la primera vez que intenté resolverlo mandé la idea en $O(n\sqrt{n})$ pero me daba TLE. De todas formas es un problema diferente pues String Journey se resuelve con Suffix Array fácilmente, mientras que aplicar Suffix Array a este problema simplemente lo complica más.

Tutorial del problema: "Ventanas"

Autor(es): Walter Erquínigo

Para cuestiones de complejidad, asumamos que $N \geq M$.

Grupo 1 (Complejidad Esperada: $O(1)$)

En este caso, no hay ninguna ventana, así que podemos usar la pantalla lo máximo posible. Esto es simplemente $min(N, M)$, porque el cuadrado máximo no puede ser mayor que el lado más pequeño de la pantalla.

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

Este caso se puede descomponer en 4 casos de la subtarea anterior. Basta notar que la única ventana existente divide la pantalla en 4 subpantallas, una a la izquierda, una a la derecha, una abajo y otra arriba. Por ejemplo, la subpantalla de la izquierda tendría M columnas y R_1 filas, y la de la derecha M columnas y $N - 1 - R_2$ filas. Un cálculo similar se puede realizar para las subpantallas de arriba y abajo. Finalmente, se reduce cada subpantalla al caso anterior y se toma la mejor solución.

Grupo 3 (Complejidad Esperada: $O\left(N^3\right)$)

Se puede reducir el problema al mayor rectángulo en un histograma. https://www.geeksforgeeks.org/largest-rectangular-area-in-a-histogram-set-1/ es una buena referencia. Existen muchas soluciones con complejidades distintas. En este caso, se puede primero pintar las partes de la pantalla que están cubiertas por ventanas en $O(VN^2)$ y luego, para cada fila, se calcula el cuadrado libre más grande que usa esa fila como base. Una solución cuadrática bastaría. O sea, para cada fila, se eligen todas las posibles dos columnas de inicio y fin del cuadrado, y se debe determinar si es un cuadrado válido o no. Esto daría una complejidad de $O(N^3)$.

Grupo 4 (Complejidad Esperada: $O\left(N^2\right)$ o $O\left(N^2 log(N)\right)$)

Podemos usar un algoritmo lineal para resolver el subproblema del histograma para cada fila. El algoritmo que usa stacks es muy eficiente y lo pueden encontrar aquí https://www.geeksforgeeks.org/largest-rectangle-under-histogram/. Eso deja en $O(N^2)$ esa parte de la tarea. Sin embargo, aún tenemos que optimizar el pintado de las ventanas. Una manera eficiente de realizar el pintado es, para cada fila, marcar el inicio y el final de cada ventana que está en esa fila y luego calcular cuántas ventanas cubren cada celda en esa fila de manera lineal. Por ejemplo, si una ventana cubre la fila R desde la columna C_1 a la C_2, se puede marcar (R, C_1) con un $+1$ y $(R, C_2 + 1)$ con -1. Luego de haber marcado para la fila R todos los inicios y finales de todas las ventanas que la cubren, se puede proceder a barrer por todas las columnas manteniendo un contador de cuantas ventanas activas están en cada celda simplemente realizando una acumulación de esos contadores $+1$ y -1. Todo eso se puede hacer en $O(VN)$, lo que da un total de $O(N^2)$.

El tiempo límite del problema es un poco alto, por lo que soluciones $O(N^2 log(N))$ serían posibles. Una posibilidad es, para cada celda vacía, realizar una búsqueda binaria del tamaño del cuadrado maximo usando esa celda como esquina superior izquierda. Cuando se define un tamaño del cuadrado, el problema se reduce a determinar el número de celdas ocupadas en un subcuadrado de una matriz dada. Esto se puede resolver con una programación dinámica clásica con queries. Pueden leer más aquí https://www.geeksforgeeks.org/submatrix-sum-queries/.

Grupo 5 (Complejidad Esperada: $O\left(V^2\right)$)

Este caso es una mejora simple de la subtarea anterior. Vemos que las coordenadas pueden ser muy grandes, pero el número de ventanas es limitado. Asumamos que tenemos un cuadrado solución. A él siempre lo podemos mover hacia la izquierda y hacia arriba hasta que por ambas direcciones toque a los límites de la pantalla o a ventanas. Eso limita nuestra búsqueda de cuadrados soluciones a todos los cuadrados que para sus esquinas inferior izquierda usan a columnas o filas de la pantalla o de las

ventanas. Como hay V ventanas, entonces hay máximo aproximadamente $(2V)^2$ coordenadas distintas. Cada ventana aporta 2 filas y 2 columnas, entonces hay $2V$ posibles diferentes valores de filas y columnas. Así, podemos realizar una compresión de coordenadas, reduciendo la pantalla a una matriz donde cada celda corresponde únicamente a valores de filas y columnas dadas por las ventanas, lo cual pueden leer aquí https://medium.com/algorithms-digest/coordinate-compression-2fff95326fb, y luego podemos aplicar el algoritmo de hallar el máximo rectángulo en un histograma modificado para resolver el problema en $O(V^2)$. En este grupo no es posible usar búsqueda binaria para hallar el lado máximo en $O(V^2 \log(\min\{n, m\}))$ pero sí se puede usar la versión con DSU para el problema del histograma, así que soluciones $O(V^2 \log V)$ también son aceptadas.

Tutorial del problema: "Altura Promedio Exacta"

Autor(es): Racsó Galván

Grupo 1 (Complejidad Esperada: $O(n! \cdot n \cdot \log n)$)

Para resolver este subtask basta con usar `next_permutation` para generar todas las permutaciones y simular la inserción de los valores en dicho orden. Ya que el promedio de alturas es $O(\log n)$, la complejidad no será de $O(n!n^2)$, sino de $O(n!n \log n)$, ya que podemos calcular la altura del árbol en $O(n)$ usando la recursión $h(T) = 1 + \max\{h(T \to izq), h(T \to der)\}$.

Las implementaciones usando punteros no fueron consideradas al momento de desarrollar el problema, por lo que se aumentó el ML a 1024 MB y el TL a 6s.

Grupo 2 (Complejidad Esperada: $O(n^4)$)

Para resolver este subtask debemos notar que podemos elegir la raíz x del árbol actual de elementos que tenemos (inicialmente todo el rango $[1, n]$) y luego de ello los elementos menores que x se tendrán que distribuir en el árbol izquierdo y los mayores que x en el árbol derecho. Gracias a la recursión $h(T) = 1 + \max\{h(T \to izq), h(T \to der)\}$ podemos ver que si logramos saber la altura de los subárboles izquierdo y derecho podemos saber la altura total del árbol, esto significa que tanto la altura como la cantidad de nodos son imprescindibles para hallar la suma, así que definiremos $memo(n, h)$ como la cantidad de formas en las que podemos generar un árbol binario de búsqueda de altura h usando n valores diferentes.

Para calcular *memo* debemos fijar la raíz x y luego de ello fijar las alturas de los subárboles izquierdo y derecho; sin embargo, el orden en el que insertemos los elementos del subárbol izquierdo no afectan al subárbol derecho y viceversa, así que son independientes entre sí. Lo anterior significa que de las $n-1$ posiciones restantes, debemos elegir $x-1$ posiciones y en ellas colocar los elementos del subárbol izquierdo, para colocar los del subárbol derecho en el resto, así que la expresión que se forma es:

$$\binom{n-1}{x-1} memo(x-1, h_L) \cdot memo(n-x, h_R)$$

Y este valor lo tendremos que agregar a $memo(n, \max\{h_L, h_R\} + 1)$, ya que la altura del árbol correspondiente es $\max\{h_L, h_R\} + 1$. Notemos que el fijar tanto h_L como h_R toma $O(n)$, así que tendremos una complejidad de $O(n^4)$.

Grupo 3 (Complejidad Esperada: $O(n^3)$)

Para resolver este subtask solo debemos optimizar la forma en la que propagamos la información: Aislamos los 3 siguientes casos:

1. $h_L > h_R$: Tendremos que sumar $memo(x-1, h_L) \cdot \sum_{h_R=0}^{h_L-1} memo(n-x, h_R)$ a $memo(n, h_L + 1)$.

2. $h_L < h_R$: Tendremos que sumar $memo(n-x, h_R) \cdot \sum_{h_L=0}^{h_R-1} memo(x-1, h_L)$ a $memo(n, h_R+1)$.

3. $h_L = h_R$: Simplemente accedemos a $memo(x-1, h_L)$ y $memo(n-x, h_R)$.

De esta forma no tendremos que fijar ambas alturas, reduciendo la complejidad a $O(n^3)$ si usamos prefix sums para calcular las sumatorias en $O(1)$.

Nota curiosa: Si a uno no se le ocurría cómo mejorar la complejidad y tenía la solución $O(n^4)$, podía simplemente preprocesar las respuestas para todos los n desde el 1 al 700, esto tomaría máximo 10 minutos en una computadora.