Olimpiada Peruana de Informática 2024 Fase 1 - Solucionario

Federación Olímpica Peruana de Informática

26 de noviembre de 2023

Tutorial del problema: "¡Bits!"

Autor(es): Racsó Galván
Desarrollador(es): Racsó Galván

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

Para resolver este grupo basta con iterar sobre todos los pares posibles usando un doble `for`, así que la complejidad será de $O(n^2)$.

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

Para resolver este grupo, es de nuestro interés el poder contar cuántos pares tienen prendido el k-ésimo bit, de manera que si dicho valor es impar, entonces el XOR resultante tendría dicho bit prendido; por otro lado, si el valor es positivo, entonces el OR resultante tendría dicho bit prendido.

Ahora la pregunta es ¿Cómo podríamos calcular dicha cantidad? Supongamos que hemos fijado el bit k y deseamos saber, en primera instancia, aquellos valores a_i que tienen el k-ésimo bit prendido y cuales no. Supongamos que estos conjuntos sean S_0 y S_1 (bit apagado y bit prendido, respectivamente).

Entonces, lo que debemos tomar en cuenta es que cuando sumamos dos elementos de un mismo conjunto, el k-ésimo bit no se va a prender directamente, así que la única forma en la que se pueda prender es analizando el resultado de la suma de los bits anteriores (es decir, de los valores módulo 2^k), ya que si tenemos dos valores en el rango $[0, 2^k - 1]$, entonces su suma estará en el rango $[0, 2^{k+1} - 2]$, así que si el resultado es mayor o igual a 2^k, este obligatoriamente tendrá el k-ésimo bit prendido.

Por otro lado, solo nos faltaría considerar las sumas de elementos en diferentes conjuntos, en los cuales es importante notar que el bit ya estará prendido, así que lo que necesitamos es que la suma de los residuos respecto a 2^k sea estrictamente menor que 2^k.

Podríamos calcular todas estas cantidades en tiempo lineal si tuviéramos los valores de a ordenados por su residuo respecto a 2^k en cada iteración, lo cual nos daría una complejidad de $O(n \log n \log MAX)$; sin embargo, esto nos daría un Time Limit Exceeded.

La observación que nos ayudará a resolver el problema en un mejor tiempo es que el tener el residuo respecto a 2^k es equivalente a tener ordenados los primeros k bits de los números, lo que nos da una idea de iterar por k creciente y aplicar un radix sort sobre los valores (como solo son 0s y 1s esto se podría mantener en dos vectores y luego unir la concatenación), evitando usar un `std::sort` en cada iteración y evitando el $O(\log n)$ extra.

La complejidad final será $O(n \log MAX)$.

Tutorial del problema: "Endoprimos"

Autor(es): Racsó Galván
Desarrollador(es): Racsó Galván

Grupo 1 (Complejidad Esperada: $O(10^n \cdot n)$)

Para resolver este grupo, nos basta notar que los primos que analizaremos estarán en el rango $[1, 10^6]$, así que podemos usar una criba de Eratostenes para saber qué valores son primos y luego iterar sobre todos los números en el rango $[10^{n-1}, 10^n - 1]$ y validar cuáles son endoprimos respecto al k dado.

La complejidad sería de $O(10^n \log \log 10^n + 10^n \cdot n) = O(10^n(\log n + n)) = O(10^n \cdot n)$.

Grupo 2 (Complejidad Esperada: $O(n \cdot 10^k)$)

Para poder manejar este grupo, primero debemos considerar los siguientes casos por separado:

- $k = 1$, en este caso nos basta con responder 4^n, ya que solo hay 4 primos de un digito.

- $n = k$, en este caso nos basta con buscar todos los primos en el rango $[10^{k-1}, 10^k - 1]$, para ello podemos usar una criba en $O(10^k \log\log 10^k) = O(10^k \log k)$.
- En otro caso, tenemos que buscar primos de longitud mayor a 1, así que usaremos DP sobre dígitos definiendo la siguiente función:

$$f(pos, rem) = \text{Cantidad de números que se pueden formar dado que ya colocamos los primeros } pos \text{ y los últimos } (k-1) \text{ forman el número } rem$$

De esta manera podremos considerar el asignar un dígito d a la posición pos solo si $10 \cdot rem + d$ es un número primo, y la transición correspondiente sería hacia el estado $f(pos + 1, (10 \cdot rem + d) \bmod 10^{k-1})$.

Ya que tenemos a lo mucho 10 dígitos a considerar y la cantidad de estados es $O(n \cdot 10^{k-1})$, la solución tendrá una complejidad de $O(n \cdot 10^k)$.

Para hacer más simple el proceso de construcción del número podríamos iterar sobre los posibles inicios de rem y $pos = k - 1$, aunque esto no afecta a la complejidad.

Tutorial del problema: "Regalo de los Dioses"

Autor(es): Jonathan Alberth Quispe Fuentes
Desarrollador(es): Racsó Galván

Para resolver este problema nos basta con notar que para cada posición i, necesitamos saber todos los posibles extremos que puedan contenerla y así calcular la máxima cantidad de agua que se podría guardar.

No es difícil notar que necesitaremos hallar la máxima altura hacia ambos lados, denotémoslas como l y r, respectivamente. Ahora, si el mínimo de l y r es menor o igual a a_i, entonces no se puede guardar agua (no habrá quién la contenga lateralmente), así que solo tenemos la siguiente condicional:

$$Aporte_i = \begin{cases} 0 & \min\{l, r\} \leq a_i \\ \min\{l, r\} - a_i & \min\{l, r\} > a_i \end{cases}$$

Ahora el problema se reduce a poder calcular l y r para cada $i = 1, \ldots, n$.

Grupo 1 (Complejidad Esperada: $O(\sum_{i=1}^{t} n_i^2)$)

Para resolver este grupo podemos hacer una búsqueda lineal por cada posición, obteniendo un $O(n^2)$ por caso, lo cual es lo suficientemente rápido para pasar el TL.

Grupo 1 (Complejidad Esperada: $O(\sum_{i=1}^{t} n_i)$)

Para resolver este grupo debemos usar prefix y suffix maximums, de manera que podemos preprocesar el máximo de todas las posiciones $[1, i]$ para todos los $i = 1, \ldots, n$ en $O(n)$, así como de todas las posiciones $[i, n]$ con otra iteración en $O(n)$. Finalmente calcularemos el aporte de cada posición.

Tutorial del problema: "Ordenamiento Hanoi"

Autor(es): Miguel Miní
Desarrollador(es): Miguel Miní

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

Podemos pasar todas las fichas, menos la mayor, al segundo poste y la mayor al tercer poste. Luego regreamos la mayor primero y luego las demás. esto nos da una cantidad de operaciones $2\binom{n+1}{2} = n(n+1) \leq 250500$.

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

En vez de solo pasar la mayor al tercer poste, podemos pasar los k mayores, y luego retornandolos en $\binom{k}{2}$ pasos como en el anterior test, si elegimos $k = \lfloor\sqrt{n}\rfloor$, obtendremos $O(n\sqrt{n})$ pasos.

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

Para lograr todos los puntos, podemos usar un **radix sort, la idea es ir ordenando desde el primer bit en adelante, manteniendo la estabilidad, basta con que los que tengan un bit dado terminen más abajo que los que no tienen el bit y así para cada iteración en bits sucesivos.**

Tutorial del problema: "Víctor y el Mago"

Autor(es): Miguel Miní
Desarrollador(es): Miguel Miní

Grupo 1 (Complejidad Esperada: $O(n + m^2\alpha(n))$)

Aprovechando que las aristas siempre están ordenadas crecientemente, podemos mantener la lista ordenada en $O(m)$ por cada iteración. y luego aplicando el algoritmo de kruskal para hallar el mst.

Grupo 2 (Complejidad Esperada: $O(mn\alpha(n))$)

Siendo que al final solo me basta mantener las aristas del mst, puedo solo guardar las aristas del mst. siendo lo mismo que en el caso anterior.

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

Lo importante a observar es que solo necesito recuperar una respuesta anterior en vez de actualizar para los casos ACCIÓN 1, Return. Ya que ACCIÓN 1 siempre será invalidada. Para las otras accioones que no siguen ese patrón, puedo usar un dsu con rollback, para ello no uso la compresión de caminos y mantengo las aristas que se unieron a los roots luego de cada operación. De esta forma es más sencillo volver a un estado anterior.

Tutorial del problema: "El número de la suerte 112012"

Autor(es): Miguel Miní
Desarrollador(es): Miguel Miní

Grupo 1 (Complejidad Esperada: $O(|s_i|^6)$)

Genero todas las secuencias válidas con sus respectivas dos subsecuencias. Esta cantidad es pequeña y se puede hacer exaustivamente.

Grupo 2 (Complejidad Esperada: $O(|s_i|^3)$)

Puedo usar un trie para colocar todas las secuencias y luego realizar un fuerza bruta por todas las posibilidades. El hecho de no usar todas las posibilidades reduce mucho las posibilidades.

Grupo 2 (Complejidad Esperada: $O(|s_i|)$)

Para obtener esta complejidad, primero debemos fijarnos en como funciona el patrón. Podemos pensar el patrón de la siguiente forma: "(()|()", donde cada 1 y 2 están emparejados.

Afirmación: El último 2 puede estar emparejado al último 1.

- De otra forma, ese 1 pertenece a otro 2, si luego viene el 0 que esta emparejado a dicho 2, puedo intercambiar el segundo 2 con el último sin perjudicar mi resultado.

Afirmación: De derecha a izquierda, podemos asignar cada 1 con el primer 2 disponible de decha a izquierda.

- Si el siguiente 1 de derecha a izquierda no dispone de algún 2, entonces debe ser un 1 inicial, en otro caso sería una contradicción. De no ser así, puede tratarse de un 2 final, el cual solo se podria intercambiar por un 2 a la derecha, pero ya han sido tomados. Solo así, debe ser un 2 inicial y si se intercambia con un 2 más a la izquierda, este también debería ser un dos inicial para ese, los cuales son intercambiables.

Con esto en mente, podemos ir recorriendo de izquierda a derecha, y cada 1 no tomado sera siempre un inicio, luego golosamente cada 1 emparejado lo puedo tomar como un 12 inicial siempre y cuando tenga un 1 no emparejado libre a su izquierda.