# Problem A. Salvando el cine

Debes hacer lo que dice el problema con una declaración if. Ten cuidado de agregar el punto final.

# Problem B. Operación

Primero debemos darnos cuenta de que la resta y la división de números enteros positivos solo hacen que el resultado sea menor, mientras que la multiplicación y la suma no lo hacen.

Entonces, dado que solo hay dos opciones posibles para optimizar la solución, podemos simplemente comparar ambas y mostrar la más grande.

# Problem C. Máximo beneficio

Para resolver este problema, necesitamos ordenar los cheques en orden no decreciente y luego calcular la suma de los últimos k elementos.

Complejidad de la solución: $\mathcal{O}(n \cdot log_n)$.

# Problem D. jbum

Para encontrar $m$, que es la cantidad mínima de minutos que Javier necesita esperar hasta que pueda realizar con el peso deseado, solo necesitamos calcular $\lceil log_2(n) \rceil$, porque en el $i$-ésimo minuto podemos agregar $2^{(i-1)}$ discos si siempre ponemos en el duplicador todos los discos que tenemos.

Una vez que conocemos $m$, necesitamos encontrar la secuencia lexicográficamente más pequeña de $x$, donde $x_i$ es la cantidad de discos que Javier coloca en el *duplicador* en el $i$-ésimo momento. Llamaremos $a_i$ a la cantidad de discos que tenemos en el $i$-ésimo minuto. Entonces tenemos que encontrar un array $a$ que cumpla $a_i \geq 2 \cdot a_{i+1}$, pero como necesitamos encontrar el más pequeño lexicográficamente entre todos los posibles $x$, $a$ también debe cumplir que $a_i = \lceil \frac{a_{i+1}}{2} \rceil$. Esto sucede porque si $a_i > \lceil \frac{a_{i+1}}{2} \rceil$ significa que podríamos disminuir $a_i$, hasta que se vuelva igual a $\lceil \frac{a_{i+1}}{2} \rceil$. Y como sabemos que $a_r = n$, podemos calcular cuántos discos Javier necesita colocar en el *duplicador* en cada minuto, primero calculando el $(m-1)$-ésimo, y terminando sabiendo cuántos discos tenemos que colocar en el *duplicador* en el primer minuto.

# Problem E. Buscando palindromios

La respuesta es $10^{\lceil \frac{n-2}{2} \rceil} \cdot 90 \cdot \lceil \frac{n-1}{2} \rceil$. Esta fórmula básicamente representa el número de palíndromos de longitud $n-2$, multiplicado por las posiciones y combinaciones de los dígitos recién agregados.

# Problem F. Harry potter en CMS

Este problema se resolvió utilizando la estructura de datos *set*, o algún otro árbol de búsqueda binario balanceado.

Si dejamos que $s$ sea el conjunto que almacena la primera consulta que resuelve cada subtarea correctamente, la respuesta es el tamaño de $s$. La idea fue almacenar, para cada subtarea, las consultas en las que Harry resolvió esa subtarea correctamente y agregar la primera de esas consultas a $s$. Luego, para la consulta de eliminación, actualizamos el conjunto de las subtareas que fueron resueltas correctamente en esa consulta y actualizamos el conjunto $s$ en consecuencia. Esta parte fue principalmente implementación, recomendamos revisar el código si aún tienes dudas.

Complejidad final: $O(q \log q + k \log k)$

# Problem G. XOR + Constructivo = Amor

En primer lugar, en lugar de usar el arreglo $b$ mencionado en el enunciado, usaremos otro arreglo $cnt$ que contará la cantidad de elementos que tienen cada bit, similar al arreglo $a$ descrito en el enunciado, lo cual es más fácil de trabajar. Debido a la primera condición, para todo $i$, $cnt_i$ debe ser al menos $a_i$, por lo que inicialmente establecemos $cnt_i = a_i$ para $i < 30$, y $cnt_i = 0$ para $i \geq 30$. Luego, también es fácil verificar la condición XOR para cada bit. Si el bit $i$ no cumple la condición XOR, agregamos 1 a $cnt_i$.

Una vez que hayamos hecho esto, solo debemos preocuparnos por cumplir la condición de suma, sabiendo que debemos agregar bits en pares, es decir, la paridad de cada $cnt_i$ debe permanecer constante.

Ahora, una de las principales observaciones es que, si existe una respuesta, la respuesta será como máximo $\max_{0 \leq i < 60} cnt_i + 2$. Esto facilita las cosas porque puedes iterar sobre los tamaños posibles y verificar si existe una solución de ese tamaño.

Para hacer esto, sea el tamaño actual $m$ y sea $left$ igual a $s - \sum_{0 \leq i < 60} cnt_i \cdot 2^i$. Si $left$ es negativo, es imposible cumplir la condición. De lo contrario, simplemente iteramos sobre los bits de más significativo a menos significativo y agregamos tantos como podamos. Como no queremos exceder $left$ y solo podemos agregar como máximo $m - cnt_i$ a cada bit $i$, podríamos agregar $\min(\lfloor \frac{left}{2^i} \rfloor, m - cnt_i)$ al bit $i$. Además, debemos verificar una vez más si se cumple la condición XOR (dejaría de cumplirse si agregamos un número impar de ocurrencias para un bit), y si no se cumple, restamos 1 a la cantidad que estamos agregando. Actualizamos $left$ y repetimos el proceso para el siguiente bit. Si al final del proceso $left = 0$, entonces es posible construir un arreglo $b$ de tamaño $m$.

Si no encontramos ningún tamaño posible, debemos imprimir $-1$.

Complejidad final: $O(\log s)$.

# Problem H. Hormigas menorquinas

Imaginemos que tenemos un subconjunto de $x$ hormigas donde el número máximo de hormigas del mismo tipo es $y$ y queremos saber el número mínimo de lanzamientos necesarios para lanzar todas las hormigas.

La condición $m$ depende de $x$ y la condición $k$ depende de $y$. Se puede demostrar que la respuesta será $max((x+m-1)/m, (y+k-1)/k)$.

Prueba:

$(x+m-1)/m$ = número mínimo de lanzamientos necesarios sin la condición $k$, y no depende del orden de las hormigas lanzadas.

$(y+k-1)/k$ = número mínimo de lanzamientos necesarios sin la condición $m$, y depende del orden de las hormigas lanzadas.

Siempre podemos usar el orden óptimo para minimizar el número máximo de hormigas del mismo tipo utilizadas y tendremos suficientes movimientos.

Lo último que necesitamos es encontrar un subconjunto de $p$ hormigas con el número máximo de hormigas del mismo tipo minimizado. Para hacer esto, simplemente podemos realizar una búsqueda binaria en la respuesta $(z)$ y comprobar si la suma de $min(z, a_i)$ para cada $i$ es $\geq p$.

Hay otras formas de hacer esto, por ejemplo, con ordenamiento.

Complejidad de la solución: $\mathcal{O}(n \cdot log_A)$ donde $A$ es el máximo $a_i$ entre todos los $i$.

# Problem I. Billetes falsos

La observación principal para este problema es que el estado de la habitación es cíclico cada 4 turnos, porque las cámaras vuelven a su posición inicial.

Con esa observación, puedes almacenar si cada celda estará cubierta por una cámara o no para cada uno de los 4 estados, de modo que puedas mantener estados de la forma $(t, r, c)$, donde $t$ es el tiempo que ha pasado (denotando el estado de la celda), $r$ es la fila y $c$ es la columna. Para almacenar esta información, es importante tener en cuenta que no puedes simplemente iterar sobre las cámaras y luego sobre las celdas que cada cámara cubre, porque esto tiene una complejidad de $O(m \cdot n)$, que es demasiado lenta. En su lugar, debes mantener las coordenadas mínimas y máximas de cualquier cámara que apunte en cada dirección para cada fila o columna, y luego determinar si una celda está cubierta en un estado utilizando esta información en $O(n^2)$.

Para hacer esto, si una cámara apunta hacia arriba, actualizas el máximo de esa columna con la fila de la cámara. Si apunta hacia abajo, haces lo mismo con el mínimo de esa columna. Si apunta hacia la derecha, actualizas el mínimo de la fila con la columna de la cámara, de manera similar con el máximo si apunta hacia la izquierda. Luego, una celda en $(r, c)$ está cubierta si y solo si:

- $\max_r \geq c$ o,
- $\min_r \leq c$ o,
- $\max_c \geq r$ o,
- $\min_c \leq r$

Luego, haces un BFS o DFS de la habitación, comenzando en $(0, 0, 0)$ y yendo a las celdas adyacentes con $(t + 1) \mod 4$ como estado que no estén cubiertas por ninguna cámara.

Si llegas a la celda $(n, n)$ en algún momento, un ladrón puede llegar a la bóveda. De lo contrario, será imposible para él.

Complejidad final: $O(n^2)$.

# Problem J. cPerturbación en la Fuerza

Este problema se resuelve utilizando programación dinámica de la mochila. Calcula, para cada número de elementos, la suma más cercana a $x$ que es posible obtener sin usar un elemento dos veces. Denotemos esa suma para un número de elementos $i$ como $dp_i$.

Entonces, la respuesta es $\min_{i=1}^{n} \lceil \frac{x-dp_i}{i} \rceil$.

Complejidad final: $O(n^2 \cdot x)$.

# Problem K. Óscar y su batalla

En primer lugar, ordenamos los personajes por su ataque y los monstruos por su defensa. Luego, iteramos sobre los personajes. Cada vez que llegamos a un nuevo personaje $i$, procesamos todos los monstruos $j$ tal que $d_j \leq a_i$. Esto se hace utilizando dos punteros para mantener una complejidad de $O(n \log n + m \log m)$, debido a la ordenación inicial. Cuando procesamos un monstruo, lo que hacemos es agregar $e_j$ al índice $c_j$ de un Árbol de Segmentos o Árbol de Fenwick.

Una vez que hemos procesado todos los monstruos, para saber cuántas monedas podríamos ganar con un personaje $i$, solo tenemos que consultar la suma de los elementos menores o iguales a $b_i$ en nuestra estructura de datos. Esto se debe a que, debido a la ordenación inicial, sabemos que todos los monstruos en la estructura de datos tienen una defensa menor o igual al ataque del personaje actual, y luego solo sumamos las monedas dadas por los monstruos con un ataque menor o igual a la defensa del personaje actual, por lo que sabemos que el personaje podrá derrotar a todos ellos.

Mantenemos el máximo de las respuestas para cada personaje como la respuesta final al problema.

Tenga en cuenta que, debido a la magnitud de las estadísticas de los jugadores y monstruos, se debe comprimir las coordenadas de los valores o utilizar una estructura de datos dispersa. En resumen, comprimir las coordenadas de $x$ valores significa asignar un identificador único entre 1 y $x$ a cada valor diferente, preservando su orden relativo.

Complejidad final: $O(n \log n + m \log m)$.

# Problem L. Intervalos aleatorios

En primer lugar, debemos verificar si los intervalos dados se intersectan, esto se puede hacer fácilmente ordenando los intervalos.

Para resolver este problema, necesitamos calcular un $dp_{[i][j]}$ = número de formas de colocar $i$ intervalos en los primeros $j$ espacios entre los intervalos existentes, $O(n^2)$ estados con $O(n)$ transiciones.

Si ya hemos calculado $dp_{[i][j]}$, podemos hacer transiciones de la siguiente forma: $dp_{[i+k][j+1]} \mathrel{+}= dp_{[i][j]} * W$ para $0 \leq k \leq n - i$, donde $W$ es el número de formas de colocar $k$ intervalos en el espacio $j + 1$.

Para calcular la dp, necesitamos saber que el número de formas de colocar $a$ intervalos en un espacio de longitud $b$ es $\binom{b+2 \cdot a-1}{2 \cdot a}$. Puedes obtener más información buscando combinatoria de bolas y cajas en internet.

Luego, necesitamos eliminar los duplicados, esto ocurre cuando tenemos intervalos de longitud uno porque podemos colocar el mismo intervalo tanto a la izquierda como a la derecha. Para hacer esto, simplemente calculamos para cada espacio $dp2_{[i][j]}$ = número de formas de colocar $i$ intervalos en ese espacio con $j$ intervalos de longitud uno a la izquierda.

También necesitamos precalcular los factoriales y los factoriales inversos.

Complejidad de la solución: $\mathcal{O}(n^3 + m)$.

# Problem M. La batalla del abismo de Helm

Lo primero que hay que notar es que las torres son independientes entre sí, el estado de una torre no afecta a otra torre, solo al daño final. Por lo tanto, esto nos lleva a pensar que podemos calcular, para cada torre, cuánto daño sufrirían las paredes internas si asignamos cierta cantidad de soldados a ella.

Además, una observación importante es que el daño que sufren las paredes internas de una torre depende solo de cuándo cae la torre, por lo que podemos calcular cuántos soldados necesitamos asignar a la torre $i$ para que caiga en la oleada $j$. Como hay un total de $q$ oleadas en todas las torres, esto significa que solo necesitamos calcular esta información $O(q)$ veces. Entonces, para cada torre, iteraremos sobre las oleadas que atacan esa torre y mantendremos el número mínimo de soldados necesarios para que la torre caiga en esa oleada (consideración especial para el caso en que una torre no cae en absoluto). Hay varias formas de hacer esto, algunas más fáciles y otras más difíciles, el autor lo hace en $O(m + q \log q)$. Sin embargo, una solución más fácil en $O(m \cdot q)$ que también pasa es para cada número de soldados, iterar sobre todos los ataques a esa torre en orden y calcular cuándo caería. El código utiliza este enfoque, ya que es más fácil de entender.

Con esta información, podemos calcular fácilmente $dmg_{i,k}$, que denota el daño que las paredes internas sufrirían de la $i$-ésima torre si hay $k$ soldados en ella. Esto será útil más adelante.

Una vez que hayas calculado el número mínimo de soldados necesarios para que la torre caiga en alguna oleada $j$, puedes convertir eso en ciertos "pesos"de la forma (soldados, daño), donde el daño es $q - j$, y hacer DP de la mochila con ellos en $O(m \cdot q)$. El DP de la mochila tendría estados $dp_{i,k}$, que denota el daño mínimo que las paredes internas pueden sufrir si consideramos las **últimas** $i$ torres y colocamos un total de $k$ soldados.

Ya podemos conocer el daño mínimo que las paredes internas pueden sufrir, dado por $dp_{n,m}$, pero ahora necesitamos construir la secuencia mínima lexicográficamente tal que las paredes internas sufran ese daño. Para hacer esto, iteramos desde la primera hasta la última torre, manteniendo dos variables $s$ y $d$, que denotan los soldados que ya hemos utilizado y el daño que ya hemos sufrido. Luego, para una torre $i$ (indexada desde 1), iteramos sobre el número de soldados que queremos colocar en ella, y colocamos el número mínimo de soldados $k$ tal que $d + dmg_{i,k} + dp_{n-i,m-s-k} = dp_{n,m}$.

Nota: El código tiene un pequeño giro, en lugar de tener $dp_{i,j}$ que denota el daño mínimo considerando las últimas $i$ torres si se colocan $j$ soldados, denota el daño mínimo colocando $j$ soldados en torres desde $i$ hasta $n$. El primero se describió en el editorial porque parece más fácil de entender.

Complejidad final: $O(m \cdot (n + q))$.

# Problem N. El naranjo de Omer

Resolveremos este problema procesando las consultas sin conexión y utilizando un barrido de línea. Esto es posible porque $\sum_{i=a}^{b} f(u,i) = \sum_{i=1}^{b} f(u,i) - \sum_{i=1}^{a-1} f(u,i)$. Por lo tanto, iteraremos sobre cada $j$ tal que $1 \leq j \leq n$ y mantendremos una estructura de datos que nos permita calcular $\sum_{i=1}^{j} f(u,i)$ rápidamente para cualquier $u$. Además, almacenaremos la respuesta de cada consulta en un arreglo $ans$, de modo que $ans_k$ sea la respuesta a la $k$-ésima consulta, que actualizaremos durante el barrido de línea.

Lo primero que debemos hacer es convertir el árbol en un arreglo unidimensional utilizando la técnica del recorrido de Euler, para poder calcular la suma en el subárbol de algún nodo en $O(\log n)$ utilizando un árbol de Fenwick o un árbol de segmentos.

Una vez que hayamos calculado el arreglo del recorrido de Euler, iteraremos sobre todos los $j$ tal que $1 \leq j \leq n$. En primer lugar, necesitaremos actualizar nuestra estructura de datos teniendo en cuenta el nuevo $j$, por lo que iteraremos sobre todos los múltiplos de $j$ y agregaremos 1 en la posición del arreglo del recorrido de Euler del nodo con un peso igual a ese múltiplo. Una vez que hayamos hecho esto, podemos calcular $\sum_{i=1}^{j} f(u,i)$ para cualquier $u$ en tiempo $O(\log n)$ realizando una consulta de suma de rango en nuestra estructura de datos. Ahora, iteraremos sobre todas las consultas de la forma $(u,a,b)$ tal que $j+1 = a$ y para cada consulta $k$, restaremos $\sum_{i=1}^{j} f(u,i)$ de $ans_k$. De manera similar, iteraremos sobre todas las consultas tal que $j = b$ y para cada consulta $k$, agregaremos $\sum_{i=1}^{j} f(u,i)$ a $ans_k$.

Para calcular la complejidad de la solución, se debe tener en cuenta que, en total, visitaremos $\sum_{i=1}^{n} \frac{n}{i}$ múltiplos. Esa suma se conoce como el número armónico $n$-ésimo, que tiene un valor aproximado de $n \ln n$. Debido a que realizamos una consulta de actualización puntual en nuestra estructura de datos para cada uno de esos múltiplos, la complejidad de esta parte es $O(n \log^2 n)$.

También se puede resolver el problema de forma *online* usando una estructura de datos conocida como *merge sort tree*.

Complejidad final: $O(n \log^2 n)$.

# Problem O. Bea la maximizadora

Para resolver este problema, convertiremos los arreglos en un grafo bipartito llamado $G$, donde los nodos en la izquierda serán las posiciones en el arreglo $a$ y los nodos en la derecha serán las posiciones en el arreglo $b$.

Definamos $sz_x$ como la cardinalidad del emparejamiento bipartito máximo si hay una arista entre dos nodos $u$ y $v$ si $a_u + b_v$ es una submáscara de $x$.

Primero construiremos la respuesta bit por bit, en orden decreciente de los bits. Al verificar si podemos establecer un cierto bit $i$ teniendo una solución actual $x$, comprobaremos si $sz_{x+2^i} = n$.

Después de hacer esto, conoceremos el valor máximo posible, y para encontrar la distancia máxima mínima, realizaremos una búsqueda binaria de la distancia y eliminaremos las aristas del grafo entre dos nodos $u$ y $v$ si $|u - v|$ es mayor que el valor que se está verificando, luego comprobaremos si el tamaño del emparejamiento bipartito máximo es $n$.

Tanto el algoritmo de Khun como el de Hopcroft Karp pueden obtener AC.

Complejidad de la solución: $\mathcal{O}(n^3 \cdot \log A)$ donde A es el límite superior de los valores en los arreglos.

# Problem P. Pistas de esquí

La primera observación en este problema es que podemos comprimir el grafo en un grafo de $O(k)$ nodos con aristas ponderadas, los nodos son los nodos desde los cuales hay un teleférico de salida, que denotaremos como nodos *especiales*, y las aristas son el número máximo de segundos que se puede tardar en ir de $a$ a $b$ utilizando **exactamente** 1 teleférico. Para calcular el peso de las aristas, se puede almacenar, para cada nodo, un arreglo $dist$, donde $dist_{i,j}$ denota la distancia desde el nodo $i$ hasta el $j$-ésimo nodo *especial*.

Para calcular este arreglo, como el grafo es acíclico, se puede iniciar un BFS desde los nodos que no tienen aristas de salida. Si actualmente estás en el nodo $u$, calculas $dist_u$ de la siguiente manera: si $u$ es el nodo *especial* número $i$, entonces $dist_{u,i} = 0$, si no es un nodo *especial*, no se realiza este paso. Luego, para cada arista de salida desde $u$ hasta $v$, para cada $i$, $dist_{u,i} = \max(dist_{u,i}, 1 + dist_{v,i})$.

Después de calcular $dist_u$, para todas las aristas desde algún $v$ hasta $u$, se resta 1 al grado de salida de $v$, si el grado de salida de $v$ se vuelve 0, se agrega a la cola.

Este algoritmo asegura que la distancia siempre es máxima, ya que se están considerando todas las posibilidades y se toma el máximo de todas ellas, debido al orden en que se realiza.

Algoritmos similares podrían funcionar, por ejemplo, utilizando el orden topológico del grafo. El aspecto importante es que todos los hijos de un nodo se procesen antes que él.

Para la representación del grafo comprimido, utilizaremos una matriz $A$, donde $A_{i,j}$ denota la distancia máxima desde el $i$-ésimo nodo *especial* hasta el $j$-ésimo nodo *especial*, utilizando uno de los teleféricos de salida desde el $i$-ésimo nodo *especial*.

Una vez que hemos calculado $dist_{i,j}$, podemos calcular la matriz $A$ iterando sobre los teleféricos. Supongamos que el teleférico actual va desde $a$ hasta $b$, entonces para todos los nodos *especiales* $c$ alcanzables desde $b$, $A_{a,c} = \max(A_{a,c}, 1 + dist_{b,c})$.

Ahora que hemos calculado esta matriz, es fácil observar que podemos obtener fácilmente la distancia desde cualquier nodo especial hasta cualquier otro nodo especial utilizando $k$ teleféricos mediante la "combinación"de $A$ consigo misma $k-1$ veces. Utilizamos "combinación"en lugar de elevar $A$ a $k$, porque la operación no es exactamente la misma que la multiplicación de matrices, aunque el algoritmo para calcularla y la complejidad son los mismos. En su lugar, si "combinamos"dos matrices $A$ y $B$ de tamaño $k \times k$, en este problema necesitamos que la matriz resultante $C$ se defina de la siguiente manera:

$$C_{i,j} = \max_{1 \le q \le k} A_{i,q} + B_{q,j}$$

Aunque no es exactamente la multiplicación de matrices, denotaremos $A$ combinada consigo misma $k-1$ veces como $A^k$, por simplicidad. Por lo tanto, podemos calcular $A^{(2^p)}$, para cada $p$ desde 0 hasta $\log_2 x$, en $O(k^3 \cdot \log x)$. Con estas matrices, realizaremos un algoritmo voraz para determinar el número mínimo de teleféricos necesarios. Tenga en cuenta que consideramos por separado si es posible pasar $x$ minutos utilizando 0 o 1 teleféricos, pero no lo discutiremos ya que es muy fácil.

Iteraremos sobre las potencias de 2 de la matriz que calculamos previamente, desde $p = \log_2 x$ hasta $p = 0$, mientras mantenemos una matriz actual, los teleféricos actuales que hemos tomado y la respuesta. Luego, para cada $p$, si combinar nuestra matriz actual con $A^{(2^p)}$ resulta en otra matriz $B$ tal que el elemento máximo de $B$ (teniendo en cuenta la ruta inicial al teleférico y la ruta final desde el teleférico, que es solo un pequeño análisis de casos) es al menos $x$, entonces establecemos la respuesta como $2^p$ + los teleféricos actuales que hemos tomado. De lo contrario, sumamos $2^p$ a los teleféricos actuales que hemos tomado y establecemos la matriz actual como la matriz $B$.

Complejidad final: $O(k^3 \log x)$