Tutorial: "Diophantine Equations!"

Para resolver este problema, basta considerar la identidad de Bézout para determinar si existe una solución o no.

Llegamos a que si:

$$c \not\equiv 0 \bmod mcd(a,b) \rightarrow \text{No existe solución}$$

Por lo tanto, solo debemos enfocarnos en el caso en que haya solución:

Subtask 1

Para el subtask 1 basta con iterar sobre un rango prudente de x y luego verificar si existe el respectivo y; dado que, una vez fijado el x, se vuelve todo una ecuación lineal respecto a y.

```
# Este es un pseudocódigo referencial haciendo uso de algunas sintaxis de Python
# No necesariamente va a compilar en el lenguaje Python
for x in range(-L,L+1):
    y = solveLinearEquation(a,x,b,c)
    if a*x + b*y == c:
        return x,y
return -1
```

En donde la función $solveLinearEquation$ da la solución a la ecuación $ax + by = c$ y devuelve y.

La complejidad de este algoritmo es $O(L)$, donde L es el límite fijado. A pesar de ello, si se fija el L demasiado grande el algoritmo demorará mucho y no podrá obtener respuesta para los casos más grandes.

Subtask 2

Una vez más nos vamos a basar en la identidad de Bézout, la cual señala que existen $x, y \in \mathbb{Z}$ tales que:

$$ax' + by' = mcd(a,b)$$

Por lo tanto, si podemos hallar dichos x', y', una solución a nuestro problema sería:

$$x = \frac{c}{mcd(a,b)}x'$$
$$y = \frac{c}{mcd(a,b)}y'$$

Lo cual se verifica por simple sustitución.

Una observación que nos da la clave para resolver el problema es el **Algoritmo de Euclides**, el cual nos da un proceso recursivo para determinar el mcd de dos enteros positivos.

Notemos que en la recursión original podemos llevar más variables:

```
def gcd(a,b):
    if b == 0: return a
    return gcd(b, a % b)
```

Supongamos que ahora nuestra función va a ser:

```
def gcd(a,b):
    # Hacer magia para obtener los x,y
    # Necesarios para que
    # ax + by = mcd(a,b)
```

Consideremos la última recursión en el algoritmo de euclides, cuando $b = 0$: Claramente la respuesta para este caso es $(1, 0)$, puesto que a en ese momento toma el valor del $mcd(a, b)$, así que:

$$1 \cdot mcd(a, b) + 0 \cdot 0 = mcd(a, b)$$

Lo cual es una identidad trivial.

Ahora, dado que la recursión halla una solución para el caso base, solo debemos plantear una relación entre un estado y el siguiente a este:

$$ax + by = mcd(a, b) \text{ estado original}$$

$$bx^* + (a \bmod b)\, y^* = mcd(a, b) \text{ siguiente estado en la recursion}$$

Algo que podemos notar es que $a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b$, así que si reemplazamos en la segunda igualdad:

$$bx^* + \left(a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b\right) y^* = mcd(a, b)$$

Realizando las asociaciones correspondientes, podemos transformar la ecuación anterior a la siguiente:

$$ay^* + b\left(x^* - \left\lfloor \frac{a}{b} \right\rfloor y^*\right) = mcd(a, b)$$

Por lo que hemos encontrado una posible solución al estado original usando la respuesta del estado posterior en la recursión, así que obtenemos un algoritmo así para hallar los x', y':

```
def gcd(a,b):
    if b == 0: return 1,0
    x1, y1 = gcd(b,a%b)
    x = y1
    y = x1 - (a // b) * y1 # a // b es la parte entera de la division
    return x,y
```

Cuya complejidad es la misma que el algoritmo de Euclides, que es $O(\ln(a + b))$ y es suficientemente rápido para los casos de prueba dados.

Tutorial: "Funny Root"

Para resolver este problema basta notar que la suma de dígitos decrece muy rápidamente, por lo que se puede realizar la recursión de manera natural y aún así el algoritmo será lo suficientemente rápido.

Subtask 1

Basta con realizar la implementación usando variables enteras de 64 bits.

Subtask 2

Para leer por primera vez los datos, basta usar cadenas de caracteres, luego la manipulación se puede realizar con enteros de 32 bits, dado que la suma máxima de dígitos para los n dados es de:

$$sum(10^{100000} - 1) = 100000 \cdot 9 = 900000$$

Y repetir lo del subtask 1.

La complejidad del algoritmo se basa en la longitud de los números que se usan en cada iteración, así que será:

$$T(n) = T(digitSum(n)) + digits(n)$$

Ahora, podemos notar que la cantidad de dígitos de n está acotada por el valor $\log_{10} n + 1$, así que:

$$T(n) = T(digitSum(n)) + \log_{10} n + 1$$

Además de lo anterior, también notemos que la mayor parte del trabajo será realizada en el primer número, y que en general el valor de la instancia se reduce tan rápido como la inversa de la función de Ackermann, por lo que una cota superior de la profundidad de las iteraciones es $\alpha(n, n)$.

Por último, como todos los logaritmos de cada iteración son menores o iguales que la del primero, una cota superior será:

$$T(n) \leq c \cdot \log_{10} n \cdot \alpha(n, n), \text{ para algun } c > 0$$

Por lo tanto la complejidad será $O(\alpha(n, n) \cdot \log n)$

```
def suma(x):
    ans = 0
    while x > 0:
        ans += x % 10
        x = x // 10
    return ans

def funnyRoot(n):
    if n < 10: return n
    return funnyRoot(suma(n))
```

Tutorial: "Maximum Remainder 1"

Subtask 1

Para este caso, nos basta con realizar una implementación eficiente y probar con todos los pares posibles, llegando a una complejidad de $O(n^2)$.

```
for i in range(0,n):
    for j in range(0,n):
        if i == j: continue
        ans = max(ans,a[i]%a[j])
```

Subtask 2

Para resolver este subproblema, debemos considerar que al momento de obtener el módulo de x respecto a algún número menor o igual a x, el módulo r cumplirá con la siguiente desigualdad:

Lema

$$x \geq m \Longrightarrow r \leq \frac{x}{2}, \text{ donde } r \text{ es el residuo de dividir } x \text{ entre m}$$

Prueba

Nos basta demostrar la contrapositiva de la proposición original, la cual es:

$$r > \frac{x}{2} \Longrightarrow x < m$$

Notemos que la primera parte de la implicancia es Falsa:

$$r > \frac{x}{2} \rightarrow 2r > x$$

Dado que $x = mq + r$, con $q \geq 1$ (debido a que $x \geq m$), entonces:

$$2r > mq + r \rightarrow r > mq \geq m \rightarrow r > m$$

Sin embargo, por defínción de módulo, esto es una contradicción, por lo que $r > \frac{x}{2}$ es falso.

Al ser la primera parte de la implicancia falsa, entonces no importa el valor de verdad de la segunda parte dado que la implicancia siempre será verdadera.

Considerando el lema, podemos notar que el máximo valor con el que puede aportar un elemento a_i es su mismo valor si y solo si existe algún a_j estrictamente mayor.

Por esta observación, es evidente que el máximo valor que podríamos obtener sería **el segundo máximo estricto**, por lo que tendremos solo 2 situaciones:

- Todos los elementos son iguales: La respuesta es 0.
- Existen al menos dos elementos diferentes: La respuesta es el segundo máximmo estricto

Para hallar el segundo máximo estricto, podríamos ordenar los elementos ascendentemente y hallar el máximo valor que sea menor estricto al último elemento.

```
sort(a)
ans = 0
i = n-2
while i > 0:
    if a[i] < a[n-1]:
        ans = a[i]
        break
```

Finalmente la complejidad depende del algoritmo de ordenamiento que se use, así que puede ser $O(n)$ o $O(n \log n)$

Una alternativa es no ordenar y maximizar el valor del segundo máximo estricto:

```
maximo = 0
for x in a:
    maximo = max(maximo,x)
ans = 0
for x in a:
    if x < maximo:
        ans = max(ans,x)
```

Cuya complejidad es de $O(n)$.

Tutorial: "Unique Substrings 1"

Para resolver el problema, debemos considerar que tenemos un alfabeto (conjunto de caracteres diferentes que pueden estar en la cadena de entrada) Σ, y notemos que por principio del palomar, cualquier cadena con longitud mayor que $|\Sigma|$ tendrá al menos un caracter repetido.

Finalmente, podemos iterar sobre cada posición i de la cadena y extendernos hasta que exista un caracter repetido, en cuyo caso saldremos del bucle interno.

```
E = 26
ans = 0
n = len(s)
for i in range(0,n):
    for len in range(1,E+1): # longitud desde 1 hasta |Sigma|
        j = i + len - 1
        if not unique(substring(i,j)): break
        ans += 1
```

La complejidad final es de $O(|s| \cdot |\Sigma|)$ para casos generales.

Tutorial: "NON-repeated element"

Para resolver este problema se dará una solución general, las características de los subtask fueron dadas solamente para considerar el caso en el que a alguien se le ocurriera alguna solución con algún ordenamiento de complejidad $O(n^2)$.

La idea de solución es evidente, por lo que se discutirán las ideas $O(n)$ y $O(n \log n)$:

Idea $O(n \log n)$

Para resolver el problema con esta complejidad, podemos usar alguna estructura de datos que permita mantener frecuencias de datos grandes en $O(\log n)$ y luego iterar sobre cada elemento para obtener el que se repita una cantidad impar de veces.

```
map<int,int> F;
for(int i=0; i<n; i++){
    F[a[i]] += 1;
}
for(auto x:F){
    if(x.second & 1) return x.first;
}
```

Otra manera de resolverlo, es ordenar todos los elementos con Mergesort o QuickSort para finalmente realizar una técnica llamada *two pointers* y determinar la frecuencia de cada elemento.

```
sort(a)
int L = 0;
int R = 0;
while(L < n){
    while(R < n and a[L] == a[R]){
        R++;
    }
    int frec = R - L;
    if(frec&1) return a[L];
    L = R;
}
```

La prueba de la correctitud se da mediante una invariante sobre la iteración, la cual se deja como ejercicio para el lector.

Idea $O(n)$

Para resolver el problema con esta complejidad, podríamos usar un algoritmo de ordenamiento lineal y realizar el *two pointers* o simplemente considerar las propiedades del **Bitwise XOR**:

$$x \oplus x = 0$$
$$x \oplus 0 = x$$
$$x \oplus y = y \oplus x$$
$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

Por lo que al considerar los elementos que se repiten una cantidad par de veces, el XOR los volverá 0, mientras que el único elemento que se repita una cantidad impar de veces se mantendrá.

Tutorial: "Matrix XOR"

Para resolver el problema, debemos recordar algunas propiedades del **Bitwise XOR** del solucionario del problema anterior:

Subtask 1

Para resolver este subtask, podemos simplemente generar la matriz implícitamente con un doble for y realizar el aporte al XOR que tiene cada elemento:

```
for i in range(0,n):
    for j in range(0,n):
        x = a[i] + a[j]
        ans ^= x
```

Finalmente resolveremos el problema con una complejidad de $O(n^2)$

Subtask 2

Para resolver este subtask, debemos considerar que la matriz B es simétrica, debido a que:

$$B_{ij} = a_i + a_j = a_j + a_i = B_{ji}$$

Por lo tanto, todos los elementos que no estén en la diagonal se anularán con el XOR. Aquellos elementos que pertenecen a la diagonal son:

$$B_{ii} = a_i + a_i = 2a_i$$

Los cuales no se anulan de manera trivial, así que realizaremos un solo bucle para aportar a la respuesta:

```
for i in range(0,n):
    x = 2*a[i]
    ans ^= x
```

Obteniendo una complejidad de $O(n)$.

Tutorial: "Good subsequences 2"

Subtask 1

Para resolver este subtask, podemos simplemente simular todas las secuencias posibles con alguna de las siguientes 2 técnicas: *backtracking* o *bitmask*.

Al momento de analizar una secuencia, podemos analizar todas sus subsecuencias y verificar que la suma de los elementos de cada una sea un múltiplo de m.

```
len = 2**n
for mask in range(1,len):
    isGood = True
    for submask in range(1,len):
        if mask & submask == submask:
            if sumaElementos(submask) % m != 0:
                isGood = False
        if not isGood: break
    if isGood: ans += 1
```

La implementación anterior usa bitmask para representar un subconjunto de posiciones, además la función *sumaElementos* es autoexplicativa. Esta solución tiene una complejidad de $O(n2^{2n})$.

Subtask 2

Para resolver este subtask, debemos analizar la siguiente pregunta: ¿Qué hace a una secuencia **buena**?

La respuesta es sencilla, pero dado que todas sus subsecuencias deben tener una suma múltiplo de m, entonces aquellas subsecuencias de tamaño 1 (los elementos solos) también deben cumplir con esta característica. La conclusión a la que llegamos es que todos los elementos deben ser múltiplos de m.

Entonces, si contamos los elementos múltiplos de m en una variable C, la respuesta sería $2^C - 1$, dado que es la cantidad de secuencias no vacías que serían buenas.

Esto nos da una complejidad de $O(n)$, dado que C está acotado por n:

```
C = 0
for x in a:
    if x % m == 0: C += 1
ans = 1
for step in range(C):
    ans *= 2
ans -= 1
```

Tutorial: "Minimal Deletions"

Subtask 1

Para resolver este subtask, se podía usar una técnica como *backtracking* o *bitmask* para probar con todos los posibles subconjuntos no vacíos del arreglo original y minimizar la cantidad de elementos borrados.

```
len = 2**n
ans = -1
for mask in range(1,len):
    mcd = 0
    for i in range(n):
        if mask & (1<<n) > 0:
            mcd = gcd(mcd,a[i])
```

```
    if mcd == 1:
        ans = max(ans,bitCount(mask)) # Minimizar los borrados es maximizar los que se quedan
ans = -1 if ans == -1 else n - ans
```

Lo que nos da una solución $O(n2^n \cdot \log Max\_Ai)$, donde Max_Ai es la cota superior de los valores de los elementos del arreglo original.

Subtask 2

Para resolver este subtask, debemos notar que si $g = mcd(a_i) > 1$, entonces cada elemento es múltiplo de $g > 1$, lo que implica que todo subconjunto de a tiene:

$$mcd(a_S) \geq g > 1$$

Donde a_S es una notación refiriéndose a los elementos de a cuyos índices están en el conjunto S.

Por lo anterior, llegamos a la conclusión de que si existe respuesta, esta es 0.

```
mcd = 0
for x in a:
    mcd = gcd(mcd,x)
if x == 1: return 0
else: return -1
```

Finalmente, nuestra complejidad será de $O(n \log Max\_Ai)$, donde Max_Ai toma la misma definición dada en el subtask 1.

Tutorial: "Apple Fight!"

Subtask 1

Para resolver este subtask, se puede realizar la simulación natural del problema sin ningún problema:

```
while X > 0 && Y > 0 && X != Y:
    if X < Y:
        Y -= X
    else:
        X -= Y
return X + Y
```

Lo que nos da una complejidad de $O(max(X, Y))$. La correctitud del algoritmo se basa en la misma definición del problema.

Subtask 2

Para resolver este subtask, debemos analizar un poco más lo que se da en la simulación:

Dado $X < Y$, entonces a Y se le resta X hasta que:

- $Y = X$, en cuyo caso la respuesta será $2X$.
- $Y < X$, en cuyo caso se reanuda la simulación pero ahora el que restará será Y.

Definamos dos roles: *atacante* y *defensor*, donde *atacante* es aquel que quita las manzanas y *defensor* es aquel al que se las quitan.

Entonces, por los dos casos analizados entre atacante y defensor definimos la función que nos da la solución al problema:

```
def solve(atacante,defensor):
    if defensor == 0: return atacante+defensor
    if atacante % defensor == 0: return 2*defensor
    return solve(defensor,atacante % defensor)
```

Y la respuesta es calculada llamando a la función $solve(\max\{x,y\}, \min\{x,y\})$

Podemos plantear una relación entre los tiempos de ejecución del algoritmo *solve* y el algoritmo de Euclides, considerando el hecho de que *solve* tiene más condiciones para detenerse:

$$T(x,y) \leq Euclides(x,y) = O(\ln(x+y))$$

Tutorial: "Digit sum equation"

Subtask 1

Para resolver este subtask, podemos considerar la ecuación respecto al valor de x y armar la siguiente desigualdad:

$$x^2 + x \cdot s(x) = n$$

Dado que $s(x) > 0$:

$$x^2 < x^2 + x \cdot s(x) = n$$

Por lo tanto, reducimos nuestro espacio de búsqueda a $[1, \sqrt{n}]$ mediante una iteración simple y prueba con cada número:

```
ans = -1
for x in range(1,sqrt(n)):
    if x*x + x*digitSum(x) == n:
        ans = x
        break
```

Donde la función *digitSum* es autoexplicativa.

Finalmente, notamos que el bucle tiene complejidad $O(\sqrt{n})$ y en cada una de las iteraciones realiza la función *digitSum*, que tiene complejidad $O(\log x)$ para cada x visitado, pero que en particular está acotada por $O(\log n)$:

$$T(n) = O(\sqrt{n} \cdot \log n)$$

Subtask 2

Para resolver este subtask, tendremos que enfocarnos en otra parte de la ecuación: la cantidad de valores diferentes de $s(x)$.

Debido a que $x \leq 10^{18}$, esto implica que el máximo $s(x)$ es $18 \cdot 9 = 162$, por lo que podemos fijar el valor de $s(x)$, luego resolver la ecuación cuadrática respecto a x y finalmente verificar si la suma de los dígitos de la raiz de la ecuación x cumple con ser la suma fijada.

Lema

$$s(x) = S \land x_1, x_2 \text{ son raices positivas de la ecuacion } x^2 + x \cdot S = n \implies x_1 = x_2$$

Prueba

Dado que ambos valores son raíces, se cumple que:

$$x_1^2 + x_1 \cdot S - n = 0$$
$$x_2^2 + x_2 \cdot S - n = 0$$

Entonces podemos restar ambas igualdades:

$$x_1^2 - x_2^2 + x_1 \cdot S - x_2 \cdot S = 0$$

Factorizamos $(x_1 - x_2)$ para obtener:

$$(x_1 - x_2)(x_1 + x_2 + S) = 0$$

Y dado que $x_1, x_2, S > 0$ por condición y definición de S, entonces la única solución es que:

$$x_1 = x_2$$

Usando el lema anterior, podemos iterar sobre los posibles valores de la suma de dígitos sin temor a perder alguna posible respuesta.

```
def solve(S,n):
    Discriminante = S*S + 4*n
    r = (int)(Discriminante**0.5)
    if r*r == Discriminante:
        if r % 2 == S % 2:
            return (r - S) // 2
        else:
            return -1
    else:
        return -1

ans = -1
for S in range(1,163):
    x = solve(S,n)
    if x > 0 and digitSum(x) == S:
        if ans == -1: ans = x
        else: ans = min(ans,x)
```

Finalmente, dado que tenemos una cantidad de pasos que depende de la cantidad de dígitos del número máximo, tendremos $O(\log^2 Max\_n)$, debido a que realizamos la función $digitSum$ en $O(\log Max\_n)$ y la cantidad de iteraciones es $O(\log Max\_n)$.

Tutorial: "Minimum cost OR deletion"

Subtask 1

Para resolver este subtask, se podía usar una recursión usando bitmask para poder reconocer si un elemento ya habia sido eliminado o aún no. En sí la recursión es simple y, si uno conoce la técnica de bitmask, es de implementación sencilla.

Definiremos la recursión como $solve(mask)$, la cual nos dará el mínimo costo de eliminar los elementos que pertenezcan a $mask$. En este caso, para $mask$ usaremos un entero, en cuya representación binaria el i-ésimo bit será 1 si el i-ésimo elemento aún no ha sido eliminado y 0 si ya lo está.

Entonces, un paso posible es probar todos los subconjuntos no vacíos de $mask$ y obtener el mínimo costo de los valores que quedan y aplicar una vez más la recursión, de la siguiente manera:

$$solve(mask) = \min_{\substack{m \subset mask \\ m \neq \varnothing}} \{getCost(m) + solve(mask \backslash m)\}$$

Donde $getCost(m)$ es una función que calcula el costo de usar los elementos de la máscara m.

La prueba del por qué esta recursión es correcta es por la misma defínción de ella: Dado el m fijo, $getCost(m)$ es un valor constante, mientras que $solve(mask \backslash m)$ es el mínimo valor que se pueda obtener del resto de elementos, por lo tanto el total es el valor mínimo para m. Finalmente, dado que se da una prueba con todos los m posibles, se logra un mínimo global.

```
len = 2**n
def solve(mask):
    if mask == 0: return 0
    ans = 10**10000
    for m in range(1,len):
        if (mask & m) == m:
            ans = min(ans,getCost(m) + solve(mask ^m))
    return ans
```

Finalmente tendremos una complejidad de $O(n \cdot 2^{2n})$, debido a que la cantidad de valores diferentes de $mask$ es $O(2^n)$, además por cada una de ellas realizamos un bucle de $O(2^n)$, cada uno de ellos con un trabajo extra de $O(n)$ por la función $getCost$.

Subtask 2

Para resolver este subtask, basta con notar que la función $bitwiseOR$ tiene dos características que nos ayudarán a llegar a la respuesta final:

- Dado que la función se realiza bit a bit en cada orden posible, entonces se da la expresión equivalente:

  $$a|b = \sum_{i=0}^{63} (a_i|b_i) \cdot 2^i$$

  Así que podemos analizar los órdenes de manera independiente.

- La función es idempotente:

  $$a|a = a$$

  Entonces, si analizamos todos los casos usando bits:

  $$0|0 = 0$$
  $$0|1 = 1$$
  $$1|0 = 1$$
  $$1|1 = 1$$

  Ahora comparamos los resultados con los de la suma:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 2$$

Notamos que se da siempre que:

$$a_i | b_i \leq a_i + b_i$$

Por lo tanto, si reemplazamos en el punto anterior:

$$a|b = \sum_{i=0}^{63} (a_i|b_i) \cdot 2^i \leq \sum_{i=0}^{63} (a_i + b_i) \cdot 2^i = \sum_{i=0}^{63} a_i \cdot 2^i + \sum_{i=0}^{63} b_i \cdot 2^i = a + b$$

Finalmente obtenemos que:

$$a|b \leq a + b$$

Usando los puntos anteriores, podemos realizar una inducción sobre la siguiente proposición:

$$OR(S) \leq solve(S)$$

- Caso Base: $|mask| = 1$, se cumple trivialmente, puesto que:

$$OR(a_i) = a_i \leq a_i = solve(a_i)$$

- Hipótesis Inductiva: Para todos los $|mask| < len$ se cumple que:

$$OR(mask) \leq solve(mask)$$

- Sea una máscara $mask$ tal que $|mask| = len$, entonces:

$$solve(mask) = \min_{\substack{m \subset mask \\ m \neq \varnothing}} \{getCost(m) + solve(mask \backslash m)\}$$

Ahora, si queremos probar que $solve(mask) \geq OR(mask)$, debemos probar que para cada elemento:

$$getCost(m) + solve(mask \backslash m) \geq OR(mask)$$

Entonces analizamos todas las máscaras m y notamos dos cosas:

$$getCost(m) = OR(m)$$
$$|mask \backslash m| < len$$

Nuestra expresión se transforma a:

$$getCost(m) + solve(mask \backslash m) = OR(m) + solve(mask \backslash m)$$

Pero como $|mask \backslash m| < len$, cumple la hipótesis inductiva:

$$OR(m) + solve(mask \backslash m) \geq OR(m) + OR(mask \backslash m)$$

Finalmente, usando la relación del OR y la suma:

$$OR(m) + OR(mask \backslash m) \geq OR(OR(m), OR(mask \backslash m)) = OR(mask)$$

Hemos probado que para todos los elementos candidatos se cumple que:

$$candidato(m) \geq OR(mask)$$

Por lo tanto, el mínimo también cumple la desigualdad:

$$OR(mask) \leq candidato(m)$$

$$OR(mask) \leq \min_{\substack{m \subset mask \\ m \neq \emptyset}} \{candidato(m)\} = solve(mask)$$

Con esto termina nuestra prueba.

Gracias al análisis anterior, podemos afirmar que la solución es solamente el bitwise OR de todos los elementos:

```
ans = 0
for x in v:
    ans |= x
```

Obtenemos una complejidad de $O(n)$.

Tutorial: "Single sided relation"

Subtask 1

Para resolver este subtask, bastaba con iterar sobre todas las posibles parejas $i \neq j$ y realizar una búsqueda lineal sobre los elementos del arreglo para determinar si existia una single sided relation o no.

```
ans = "Yes"
for i in range(n):
    for j in range(n):
        if i == j: continue
        found = False
        for k in range(n):
            if a[i]*a[j] == k: found = True
        if !found: ans = "No"
```

Terminamos con una complejidad de $O(n^3)$.

Subtask 2

Para resolver este subtask, debíamos analizar un poco más a profundidad los posibles valores que podríamos encontrar en el arreglo.

Notemos que 0 y 1 son valores "comodin", dado que si $a_i = 0$, entonces:

$$a_i \cdot a_j = 0 = a_i, \text{ entonces elegimos } k = i \text{ y es valido}$$

Mientras que para $a_i = 1$:

$$a_i \cdot a_j = a_j, \text{ entonces elegimos } k = j \text{ y es valido}$$

Entonces, nos interesaremos en los valores diferentes a 0 y 1. Consideremos que tenemos los valores mayores que 1 en un arreglo ordenado a' de tamaño m, en este arreglo se cumple que:

$$a'_1 \cdot a'_m > a'_m$$

Esto debido a que $a'_1 > 1$.

Ahora, notemos que si existen al menos 2 elementos en este arreglo, automáticamente el producto anterior no pertenece al arreglo original: Concluimos que solo puede existir a lo mucho 1 elemento mayor que 1 para que exista una single sided relation.

```
mayores = 0
for x in a:
    if x > 1: mayores += 1
ans = "Yes" if mayores <= 1 else "No"
```

Este algoritmo tiene una complejidad de $O(n)$.

Tutorial: "Different Prime Factors"

Subtask 1

Para resolver este subtask, bastaba con responder a cada consulta con una complejidad lo suficientemente rápida. Recordando que existe un algoritmo con complejidad $O(\sqrt{n})$ para determinar los factores primos diferentes de un número, entonces lo usamos para responder las consultas, tomando un total de $O(q\sqrt{Max\_n})$.

```
def solve(x): # Responder por consulta
    ans = 0
    for i in range(2,n**0.5):
        if n % i == 0:
            while n % i == 0:
                n //= i
            ans += 1
    return ans

for caso in range(q):
    x = int(input())
    print(solve(x))
```

La correctitud del algoritmo se basa en la correctitud de la función *solve*, cuya demostración se realiza por contradicción sobre la siguiente proposición: Sea $S(n)$ el conjunto de divisores del número n ordenados de manera ascendente, entonces el segundo elemento de $S(n)$ es un número primo. Se deja como ejercicio la prueba de la proposición.

Subtask 2

Para resolver este subtask, reconocemos que debemos mantener una complejidad de respuesta mucho menor que $O(\sqrt{Max\_n})$, preferiblemente $O(\log Max\_n)$ en el peor de los casos.

En este paso, recordamos la criba de Eratóstenes, cuya implementación toma $O(L \log\log L)$, siendo L el límite superior del intervalo a analizar. Sin embargo, la criba original determina si un número es primo o no; a pesar de este obstáculo, se puede realizar una modificación sobre ella: en vez de guardar un booleano que determine si el número es primo o no, guardamos algún factor primo del número.

Luego de una buena implementación de lo anterior, podemos considerar el realizar una factorización usando la criba modificada, usando el siguiente bucle:

```
ans = 0
while x != 1:
    f = pf[x]
    while x % f == 0:
        x //= f
    ans += 1
```

Esto puede ser realizado si se mantiene el arreglo pf, el cual almacenara en $pf[i]$ algún factor primo de i, para todos los números hasta el 10^6.

No es difícil notar que el bucle interno se demora α_i si $f = p_i$ en la representación canónica de x, por lo tanto, el bucle completo se demorará $\sum_{i=1}^{r} \alpha_i$.

Proponemos que:

$$\sum_{i=1}^{r} \alpha_i \leq \log_2 x$$

Lo cual es cierto, debido a que todos los factores primos de x que no sean 2 van a aportar a la suma menos que si fueran un 2, por lo que la cota de la factorización usando este algoritmo será:

$$T(factorizacion) = O(\log_2 x) = O(\log x)$$

Finalmente, tendremos un algoritmo de complejidad $O(q \log Max\_n + Max\_n \log\log Max\_n)$, lo cual es suficiente para pasar el límite de tiempo establecido.

Nota: Para lenguajes lentos como Java y Python se debe usar métodos de lectura y escritura rápidos.