Editorial MOG Round #26

A – Componentes electrónicos I

Lo esencial en este problema es darse cuenta de dos cosas: primero, que las operaciones de combinar componentes son ambas conmutativas y asociativas, lo que facilita el proceso de hallar todos los componentes electrónicos resultantes de combinar exactamente *k* componentes simples (de resistencia 1 ó 2), que se puede hacer combinando todos los que están compuestos de exactamente *k-i* componentes simples con los de exactamente *i* componentes simples; y segundo, que la cantidad de componentes distintos que se pueden formar con a lo sumo 10 componentes simples es aproximadamente 100000 (10^5). Dicho esto, la solución esperada es: **construir todos los componentes distintos de a lo sumo N=10 componentes simples.**

```
def comb_serie(x, y):
    return x + y

def comb_paral(x, y):
    return 1 / (1/x + 1/y)

comp[1].add(Fraction(1, 1))
comp[1].add(Fraction(2, 1))

for k = 2 to 10:
    for i = 1 to k/2:
        j = k-i
        for x in comp[i]:
            for y in comp[j]:
                comp[k].add(comb_serie(x, y))
                comp[k].add(comb_paral(x, y))

sol = -1
f = Fraction(A, B)
for k = 1 to 10:
    if f in comp[k]:
        sol = k
        break
```

B – Componentes electrónicos II

El enfoque para lograr resolver el ejercicio con *N*=16 es **meet in the middle.** Después de haber calculado todos los posibles componentes de a lo sumo 10 componentes simples, si el componente deseado no ha sido calculado aún, entonces se parte de éste y se va "descomponiendo" en componentes más pequeños, y se verifica si el componente resultante es posible construirlo.

```
def decomb_serie(f, x):
    return f - x

def decomb_paral(f, x):
    return 1 / (1/f - 1/x)

def possible(f, k):
    if f[0] < 0 or f[1] < 0:
        return False

    if k <= 10:
        return f in comp[k]

    for i = 1 to k/2:
        for x in comp[i]:
            y = decomb_serie(f, x)
            if possible(y, k-i):
                return True
            y = decomb_paral(f, x)
            if possible(y, k-i):
                return True

    return False

sol = -1
f = Fraction(A, B)
for k = 1 to 16:
    if possible(f, k):
        sol = k
        break
```

C – Puntos alcanzables

Para resolver este ejercicio la idea es mantener dos valores R y r que indican la distancia más lejana y la más cercana del punto inicial que se puede alcanzar utilizando exactamente i saltos. El valor R siempre se incrementa, eso está claro, el problema es cómo actualizar r. Supongamos que estamos analizando el salto de longitud x, entonces hay **tres casos**: $x \leq r$, $r < x \leq R$, $R < x$. Les dejo a ustedes cómo resolver cada caso.

D – Toros y Vacas

La idea aquí era **probar todos las posibles soluciones** (de 1111 a 6666) y determinar cuáles de ellas no contradicen la información de preguntas y respuestas. La parte interesante del ejercicio es, fijada una solución, dar la respuesta para una pregunta, y luego comprobar que la respuesta obtenida sea igual a la que tenemos como dato.

```
solutions = []

for solution = 1111 to 6666:
    ok = True
    for query, answer in input_data:
        if judge(query, solution) != answer:
            ok = False
            break
    if ok:
        solutions.add(solution)
```

E – Juego LCM GCD

Si construimos un grafo, donde cada número de la lista es un vértice y cada restricción *<i, j, G, L>* es una arista, entonces se forman varias componentes conexas, por lo que habría que comprobar que hay solución para todas las componentes conexas. Comprobemos sin hay solución para una componente conexa: como en cada restricción *<i, j, G, L>* se cumple que *L* ≤ 10000, entonces el *i*-ésimo número y el *j*-ésimo número son ambos ≤ *L* y por tanto ≤ 10000, además si se fija un valor para el *i*-ésimo número, entonces el *j*-ésimo queda determinado a causa de los valores de *G* y *L* (notar que $LCM(x,y) = \frac{x*y}{GCD(x,y)}$ y entonces $y = \frac{LCM(x,y)*GCD(x,y)}{x}$), por lo que para verificar si hay solución o no, basta con **hacer DFS a partir de cualquier nodo de la componente conexa (asignándole un valor entre 1 y 10000) e ir determinando los valores para el resto de los vértices de la componente conexa**, si se encuentra una contradicción, entonces el valor inicial dado no es válido; si ningún valor inicial es válido, entonces no hay solución. Recordar hacer esto para cada componente conexa.

```
def clean(node):
    val[node] = -1
    foreach (v, G, L) in adj[node]:
        if val[v] != -1:
            clean(v)


def fill(node, x):
    if val[node] != -1:
        return val[node] == x

    val[node] = x
    foreach (v, G, L) in adj[node]:
        y = G*L
        if y % x != 0:
            return False
        y /= x
        if gcd(x, y) != G or lcm(x, y) != L:
            return False
        if not fill(v, y):
            return False
    return True


def brute_force(node):
    for value = 1 to 10000:
        clean(node)
        if fill(node, value):
            return True
    return False
```

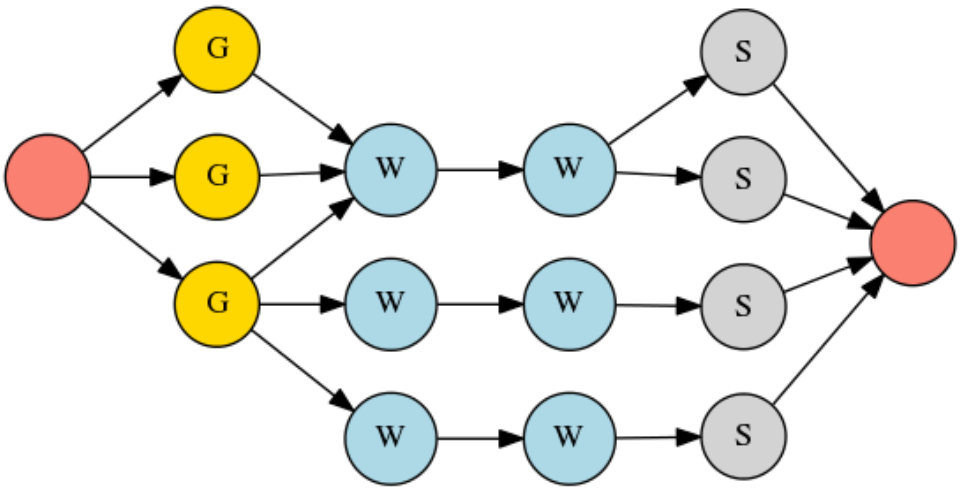
```
sol = True
for node = 1 to N:
    if val[node] == -1: // not visited
        if not brute_force(node):
            sol = False
            break
```

F – Mineros

Solución: **Construir una red de flujos** donde hay un vértice por cada mina de oro, por cada mina de plata, un vértice inicial, uno final y dos vértices por cada minero; las aristas van desde el nodo inicial hacia las minas de oro, desde las minas de oro hacia el primer vértice de cada minero, desde el primer vértice de cada minero hacia el segundo, desde el segundo vértice de cada minero hacia las minas de plata, y desde las minas de plata hacia el nodo final. Todas las aristas tienen capacidad 1 y el objetivo de tener 2 vértices por minero es el de limitar a 1 la cantidad máxima de minas de cada tipo que se le asignan. OJO: las aristas entre una mina y un vértice de un minero sólo se añade a la red si el minero puede llegar en a lo sumo k pasos a la mina analizada, y para esto es posible hacer BFS por cada minero.

Ejemplo:

```
5 5 3
..SGG
SX...
..GWS
W..X.
..W.S
```



G – Jugando con círculos

Como no hay intersección entre los perímetros de cada círculo, **entonces se puede formar un bosque, donde cada nodo representa un círculo y su padre será (si tiene) el nodo que representa al círculo más pequeño que lo contiene**. Poner una piedra sobre un círculo, implica no poder usar más ese círculo ni ninguno de sus ancestros en su árbol. Usar un poco de teoría de juegos (***Grundy number***, etc) para poder caracterizar un juego. El *Grundy number* de un bosque es el XOR de los *Grundy numbers* de todos los árboles que lo componen. Para hallar el *Grundy number* de un árbol, es necesario hacer todas las posibles jugadas que involucren a algunos de sus nodos. Analicemos solamente un árbol: una jugada consiste en eliminar un nodo y todos sus ancestros, el XOR de los *Grundy numbers* todos sus subárboles que quedaron será el *Grundy number* del bosque que quedó, el *Grundy number* del árbol será el menor entero al que no se pueda llegar en una jugada. Tener en cuenta que el *Grundy number* de un bosque vacío es 0, pues es una posición perdedora y, por tanto, el *Grundy number* de una hoja es 1.

H – Recordando nombres I

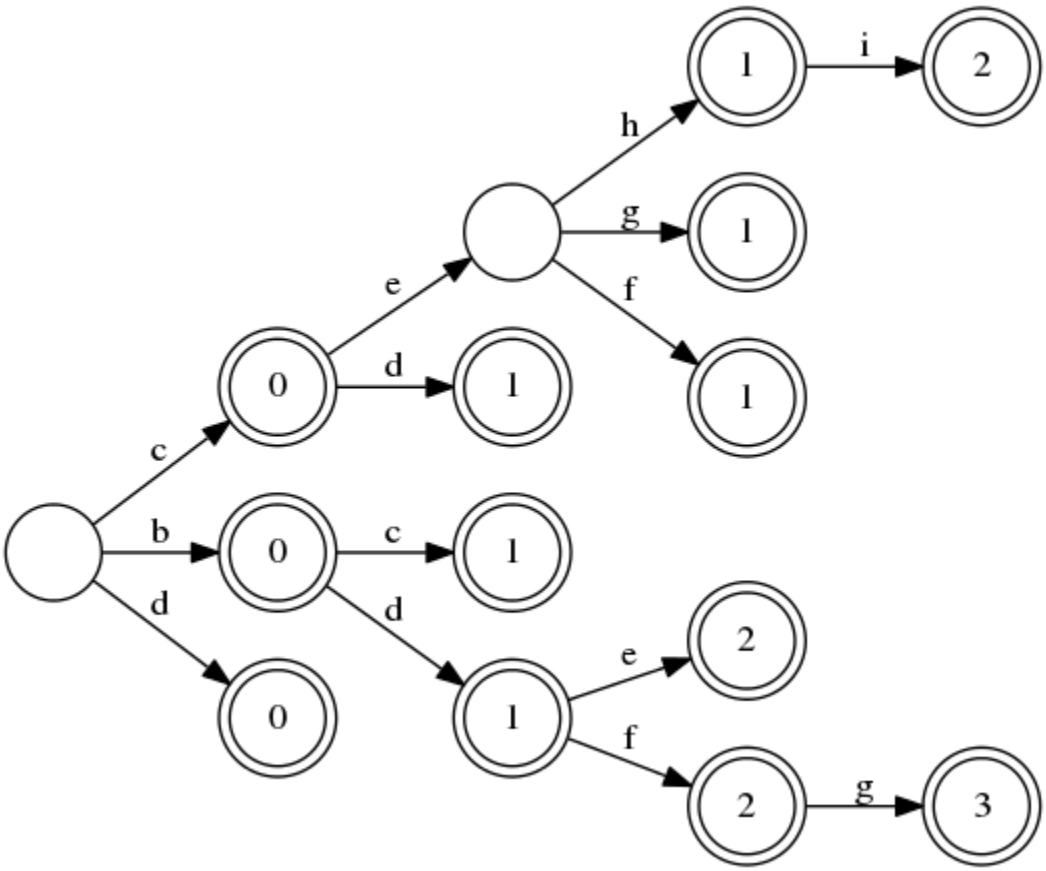
Notar que es posible dividir el problema en dos partes: el prefijo de tamaño K de la lista y las restantes $N-K$ palabras. Una vez hallada la cantidad de formas de obtener un prefijo de tamaño K, la solución será el producto de esa cantidad con $(N-K)!$, pues las restantes $N-K$ palabras pueden estar en cualquier orden en la lista. Veamos cómo podemos obtener la cantidad de prefijos de tamaño K de la lista. Una solución es posible utilizando **programación dinámica**. Definamos $dp[h][i]$ como la cantidad de prefijos de tamaño *h* colocando a la i-ésima palabra en la posición *h*. Entonces $dp[1][i] = 1$ $(1 \leq i \leq N)$ y además $dp[h][i] = \sum_1^N dp[h-1][j]$ $(1 \leq i \leq N, h \geq 2, i \neq j, s[j]\ es\ prefijo\ de\ s[i])$. La solución será $\sum_1^N dp[K][i] * (N-K)!$.

```
for i = 1 to N:
    dp[1][i] = 1

for h = 2 to K:
    for i = 1 to N:
        for j = 1 to N:
            if i != j:
                continue
            if is_prefix(s[j], s[i]):
                dp[h][i] += dp[h-1][j]

sol = 0
for i = 1 to N:
    sol += dp[K][i]
sol *= fact(N-K)
```

I – Recordando nombres II

En este subproblema ya no es posible utilizar la solución anterior *O(N*N*K)* pues N es muy grande (*≤200000)*. Es necesario entonces otro enfoque, que también es válido, por supuesto, para el primer subproblema. Construyamos un árbol, donde cada nodo representa una palabra *S* y su padre (si tiene) es el nodo que representa a la mayor palabra *W* que es prefijo de *S*, donde además la raíz representa a la cadena vacía. **De esta forma, un prefijo de tamaño *K* de la lista sería una secuencia de nodos que pertenecen todos a la misma rama del árbol, en orden ancestro-descendiente**. La cantidad de posibles prefijos de tamaño *K* se pueden formar, donde la *K*-ésima palabra es *S*, es $\binom{D[S]}{K-1}$ donde *D[S]* es la profundidad de *S* en el árbol (la cantidad de ancestros que tiene, sin contar a la raíz). La solución es $\left(\sum \binom{D[S]}{K-1}\right)(N-K)!$. Esto se puede lograr eficientemente insertando las palabras en un **trie** e ir guardando las profundidades de cada palabra en su nodo terminal.

Este sería el trie resultante en el tercer ejemplo.



J – Triángulos rectángulos

Lo esencial de este problema es darse cuenta de que los triángulos rectángulos se forman solamente cuando dos de sus vértices son los extremos de un diámetro de la circunferencia circunscrita del polígono regular (es útil aquí el **Teorema de Tales**). Notar que si *N* es impar no hay forma de que dos vértices del polígono formen un diámetro y por tanto la solución es 0. Cada diámetro distinto que se pueda obtener con dos de los vértices dados, aporta *K-2* triángulos rectángulos (tomando como tercer vértice a cada uno de los *K-2* restantes).

```
foreach point in input:
    points.add(point)

sol = 0
if N % 2 == 0:
    for i = 0 to N/2 - 1:
        if (i + N/2) in points:
            sol += K-2
```

K – Numeritos especiales

La solución es iterar por cada *Y* entre *A* y *B* (incluidos) y verificar si *Y* tiene hijos o no. Si *X* es hijo de *Y*, entonces *X = D[X] * Y*, por lo que *X* es múltiplo de *Y*, y la verificación se limita a analizar solamente los múltiplos de *Y*. Ahora tratemos de demostrar que sólo es necesario analizar una cantidad pequeña de ellos. Fijemos un valor *k* para *D[X]*, entonces la cantidad mínima de dígitos que debe tener *X* para que *D[X]* sea igual a *k* es de $t = ceil\left(\frac{k}{9}\right)$, o sea, el menor entero mayor o igual que $\frac{k}{9}$. Notar que cualquier número con menos de *t* dígitos tendrá un *D[X]* de a los sumo *9(t-1) < k.* Ahora sabemos que *X* tiene al menos *t* dígitos, y por tanto $X \geq 10^{t-1}$. Entonces $\frac{X}{k}$ será al menos de $\frac{10^{t-1}}{9t}$ (importante ver que es una función creciente), pues *k* es a lo sumo *9t*. Ahora, tomemos un *k* mayor que 108=9*12, entonces $t \geq 13$ y por tanto cualquier $\frac{X}{k}$ ya es mayor que 10^9 (recordar que $Y \leq A \leq 10^9$), luego, **solamente es necesario analizar los primeros 108 múltiplos de *Y***.

NOTA: en realidad la cota más ajustada es 93, pues el *k* más grande se logra con Y=860,202,043 y su hijo X = 93*Y = 79,998,789,999.

```
def is_childless(y):
    for i = 1 to 108:
        x = i * y
        if sum_of_digits(x) == i:
            return False
    return True

sol = 0
for i = A to B:
    if is_childless(i):
        sol++
```