Antes de empezar a leer las soluciones es muy recomendable que entendáis todas las soluciones y las implementéis (hasta conseguir 100 puntos de cada problema). En muchas de las soluciones se da una demostración formal de por qué funciona y enlaces a manuales de temas que pueden resultar útiles para entender la solución. Aunque durante el concurso no hayáis resuelto algún problema no significa que no podáis entender la solución de este y de los siguientes. Mucho ánimo y ¡si tenéis cualquier pregunta no dudeis en preguntar en el canal de Discord de la OIE!

OOOIIIEEE!!!

Empieza la OIE 2020, con la emoción la gente ha empezado a publicar en las redes la palabra "OOOIIIEEE!!!" con cada vez más letras. Con tanta emoción han conseguido saturar todos los servidores. Para que no vuelva a pasar semejante locura los propietarios de las redes solo permiten publicar mensajes que contengan cuatro enteros. En seguida, los fans de la OIE han aprovechado esta norma para seguir expresando su emoción, ahora escriben mensajes con cuatro enteros: n_o, n_i, n_e, $n_!$. Indicando el número de "*O*", el de "*I*", el de "*E*" y el de "*!*".

Dados los cuatro enteros escribid la palabra que representa.

Estrategia de la solución

Este problema no tiene complicación algorítmica, simplemente hay que hacer lo que dice el enunciado.

Solución sencilla en Python

```
LETRAS = "OIE!"

t = int(input())
for test in range(t):
    nums = []
    for s in input().split():
        nums.append(int(s))
    s = ""
    for i in range(4):
        for j in range(nums[i]):
            s += LETRAS[i]
    print(s)
```

Autor de la solución: Izan Beltrán (Miembro del comité organizador)

Solución optimizada en Python

```
t = int(input())
for test in range(t):
    a, b, c, d = map(int, input().split())
    print("O"*a + "I"*b + "E"*c + "!"*d)
```

Autor de la solución: Izan Beltrán (Miembro del comité organizador)

Solución en C++

```cpp
#include <iostream>

using namespace std;

int main() {
  int t;
  cin >> t;

  // Iteramos hasta que t es 0
  while (t--) {

        // Leemos cuatro enteros
        int a, b, c, d;
        cin >> a >> b >> c >> d;

        // Escribimos a O's
        for (int i = 0; i < a; ++i)
        cout << 'O';

        // Escribimos b I's
        for (int i = 0; i < b; ++i)
        cout << 'I';

        // Escribimos c E's
        for (int i = 0; i < c; ++i)
        cout << 'E';

        // Escribimos d !'s
        for (int i = 0; i < d; ++i)
        cout << '!';

        // Escribimos un salto de linea
        cout << endl;
  }
}
```

Autor de la solución: Cesc Folch (Miembro del comité organizador)

Caramelos

Un grupo de *n* amigos ha entrado en una tienda de caramelos con la intención de acabar con todas la existencias. En la tienda tienen *m* tipos de caramelos, y de cada tipo *i* tienen c_i caramelos. Los amigos van a repartir cada tipo de caramelos en *n* montones, posiblemente vacíos, de manera que la diferencia entre el menor montón con el mayor sea mínima. Una vez creados los *n*m* montones, cada amigo va a coger un montón de cada tipo de caramelos de manera que la diferencia entre el amigo con más caramelos y el amigo con menos caramelos sea mínima.

¿Cuántos caramelos tienen el que tiene más y el que tiene menos?

Estrategia de la solución

Es fácil ver que la diferencia entre el mayor y el menor montón de cada tipo de caramelos es como mucho 1. Una vez vemos esto, supongamos que A es el amigo con más caramelos y B es el que tiene menos. Diremos que c_A es el número de caramelos de A y c_B el de B. Si $c_A > c_B + 1$, entonces hay como mínimo un tipo de caramelo del que A tiene más que B, al principio hemos visto que solo puede tener uno más. Si A le da este caramelo extra a B vemos que la diferencia entre c_A y c_B se reduce en 2 y los montones de cada tipo de caramelo siguen cumpliendo la condición inicial. Entonces mientras tengamos amigos A y B tales que $c_A > c_B + 1$ podemos hacer este cambio, esto va a terminar en algún momento porque después de cada operación se sigue cumpliendo $c_A >= c_B$ y solo hay un número finito de amigos. Entonces vemos que la diferencia es como mucho 1. Por tanto, si *S* es el número total de caramelos, tenemos que la solución es *[S/n]*(superior) y *[S/n]*(inferior), que representan la parte entera
superior e inferior del número *S/n*. En este problema la suma *S* podía ser mayor que 2^31 ~ 2*109 y por lo tanto era necesario usar *long long* para hacer los cálculos.

Solución sencilla en Python

```
t = int(input())
for test in range(t):
    line = input().split()
    n = int(line[0])
    m = int(line[1])
    c = []
    for s in input().split():
        c.append(int(s))
    least = 0
    extra = 0
    for i in range(m):
        least += c[i]//n
        extra += c[i]%n
    least += extra//n
    extra %= n
    print(least + (1 if extra > 0 else 0), least)
```

Autor de la solución: Izan Beltrán (Miembro del comité organizador)

Solución optimizada en Python

```python
t = int(input())
for test in range(t):
    n, m = map(int, input().split())
    c = list(map(int, input().split()))
    least = 0
    extra = 0
    for i in range(m):
        least += c[i]//n
        extra += c[i]%n
    least += extra//n
    extra %= n
    print(least + (1 if extra > 0 else 0), least)
```

Autor de la solución: Izan Beltrán (Miembro del comité organizador)

Solución en C++

```cpp
#include <iostream>

typedef long long ll;

using namespace std;

int main() {
  int t;
  cin >> t;

  while (t--) {
       ll n;
       int m;

       cin >> n >> m;

       // Calculamos la suma de todos los caramelos
       ll sum = 0;
       while (m--) {
       ll k;
       cin >> k;
       sum += k;
       }

       // Escribimos la parte entera superior y la
       // parte entera inferior de sum/n
       cout << (sum + n -1)/n << ' ' << sum/n << endl;
  }
}
```

Autor de la solución: Cesc Folch (Miembro del comité organizador)

Ruleta

Miguel tiene una ruleta con n posiciones, en cada posición hay un entero (puede ser negativo). Pero esta ruleta está trucada. La primera vez que se tira sale la posición p_1, la segunda la p_2, hasta la p_n en la n-ésima tirada, en la (n+1)-ésima tirada vuelva a salir la p_1, luego p_2... Todos los p_i son diferentes, es decir p es una permutación de {1...n}. En cada posición hay un entero positivo r_i.

Miguel ha abierto una parada de apuestas con su ruleta, el juego consiste en apostar una cantidad de dinero d, tirar la ruleta, y cobrar $r*d$, donde r es el número que ha salido en la ruleta. En caso de que $r*d$ sea negativo se cobrará, si no, se pagará a Miguel.

Hoy Miguel tiene m clientes esperando en una fila ordenada para jugar, como son clientes habituales ya sabe cuanto van a apostar. Sabiendo que el i-ésimo jugador apostará d_i, Miguel quiere modificar la permutación p de su ruleta para ganar el máximo dinero, es decir, va a reordenar la posiciones p_i de su ruleta para tener el máximo beneficio. Recordad que Miguel tiene beneficio si r_i*d_i es positivo, de lo contrario pierde dinero.

¿Cual es la máxima cantidad de dinero que puede ganar?

Estrategia de la solución

Supongamos que hemos fijado el orden de la ruleta, entonces es fácil ver que al i-ésimo jugador le va a salir el mismo número que al (i+n)-ésimo, y que al (i+$2n$)-ésimo, (i+$3n$)-ésimo... Entonces podemos agrupar estos jugadores en uno solo que va a apostar la suma de sus apuestas y el resultado será el mismo. Por lo tanto podemos reducir el problema a tener n posiciones de la ruleta y n participantes, en caso que $n > m$ podemos crear jugadores “imaginarios” que apuesten 0 para llegar hasta n. Una vez tenemos esto el problema consiste en asignar a cada jugador una posición de la ruleta para que la suma sea máxima. Se puede demostrar[1] que el valor máximo se consigue ordenando las dos listas (los valores de la ruleta y los valores que apuestan los jugadores) y asignar al i-ésimo jugador en el orden la i-ésima posición de la ruleta en el nuevo orden. Para poder ordenar era necesario usar la función *sort* de C++ o equivalentes en otros lenguajes. El resultado podía ser mayor que 2^31 ~ $2*10^9$ y por lo tanto era necesario usar *long long* para hacer los cálculos.

Solución sencilla en Python

```
t = int(input())
for test in range(t):
```

[1] (*) Supongamos que es otra la asignación con el mayor resultado, entonces existen r_a y r_b con $r_a > r_b$ que están asignadas a d_a y d_b con $d_a < d_b$, Si calculamos (r_a*d_b + r_b*d_a) - (r_a*d_a + r_b*d_b) = r_a*(d_b – d_a) + r_b*(d_a – d_b) = (r_a – r_b)*(d_b – d_a) > 0 ya que $r_a > r_b$ y $d_b > d_a$, y por lo tanto (r_a*d_b + r_b*d_a) > (r_a*d_a + r_b*d_b) lo que contradice que sea la máxima asignación y por tanto no puede ser otra la asignación máxima.

```
    line = input().split()
    n = int(line[0])
    m = int(line[1])
    p = []
    for s in input().split():
        p.append(int(s))
    d = []
    for s in input().split():
        d.append(int(s))
    x = []
    for i in range(n):
        x.append(0)
    for i in range(m):
        x[i%n] += d[i]
    x = sorted(x)
    p = sorted(p)
    ans = 0
    for i in range(n):
        ans += p[i]*x[i]
    print(ans)
```

Autor de la solución: Izan Beltrán (Miembro del comité organizador)

Solución optimizada en Python

```
t = int(input())
for test in range(t):
    n, m = map(int, input().split())
    p = list(map(int, input().split()))
    d = list(map(int, input().split()))
    x = [0 for i in range(n)]
    for i in range(m):
        x[i%n] += d[i]
    x = sorted(x)
    p = sorted(p)
    ans = 0
    for i in range(n):
        ans += p[i]*x[i]
    print(ans)
```

Autor de la solución: Izan Beltrán (Miembro del comité organizador)

Solución en C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef long long ll;
typedef vector<ll> vi;
```

```
int main() {
  int t;
  cin >> t;

  while (t--) {
        int n, m;
        cin >> n >> m;

        // Leemos los valores de la ruleta
        vi V(n);
        for (int i = 0; i < n; ++i) cin >> V[i];

        // Leemos los valores de las apuestas
        // y los sumamos al jugador equivalente
        // de los n primeros
        vi B(n, 0);
        for (int i = 0; i < m; ++i) {
        ll k;
        cin >> k;
        B[i%n] += k;
        }

        // Ordenamos las dos listas
        sort(V.begin(), V.end());
        sort(B.begin(), B.end());

        // Calculamos el resultado
        ll res = 0;
        for (int i = 0; i < n; ++i)
        res += V[i]*B[i];

        cout << res << endl;
  }
}
```

Autor de la solución: Cesc Folch (Miembro del comité organizador)

Diámetro

Dado un árbol, un grafo conexo no dirigido sin ciclos, se define el diámetro como la mayor distancia entre dos vértices del grafo.

Dado un árbol, calculad el diámetro.

Si no sabéis lo que es un grafo, o queréis repasar el concepto, ¡siempre podéis consultar los manuales de la OIE!

Estrategia de la solución

En caso de no saber lo que es un grafo recomendamos leer primero este artículo del manual.
En este problema se pedía la implementación de un algoritmo para calcular una propiedad importante de los grafos: el diámetro. En este caso el grafo dado era un árbol, lo que simplifica el problema.
Es importante también estar familiarizado con los algoritmos de DFS y/o BFS, en caso contrario recomendamos leer este artículo del manual.
El algoritmo para calcular el diámetro en un árbol es el siguiente:

1. Escogemos un vértice V cualquiera y calculamos la distancia de este vértice a todos los demás usando el algoritmo BFS (al ser un árbol también podemos usar el DFS para calcular distancias mínimas)
2. Sea U el vértice más alejado de V, o uno de ellos si hay más de uno. Ahora calculamos la distancia de U a los demás vértices como en el paso anterior
3. Sea W el vértice más alejado de U, o uno de ellos si hay más de uno. El diámetro del grafo va de U a W y por lo tanto la respuesta es la distancia entre U y W

Podéis encontrar distintas demostraciones de porqué este algoritmo funciona en este artículo (en inglés).

Solución sencilla en Python

```
from collections import deque

t = int(input())

for test in range(t):
    n = int(input())
    adj = []
    for i in range(n):
        adj.append([])
    for i in range(n-1):
        line = input().split()
        x = int(line[0])
        y = int(line[1])
        adj[x].append(y)
        adj[y].append(x)
```

```
    # Encontrar el mas lejano usando BFS:
    dist = 0
    furthest = -1
    q = deque()
    visited = []
    for i in range(n):
        visited.append(False)
    q.append((0, 0))
    visited[0] = True
    while len(q) > 0:
        u, d = q.popleft()
        if d > dist:
            dist = d
            furthest = u
        for v in adj[u]:
            if not visited[v]:
                visited[v] = True
                q.append((v, d+1))

    # Computar diametro usando BFS:
    dist = 0
    q = deque()
    visited = []
    for i in range(n):
        visited.append(False)
    q.append((furthest, 0))
    visited[furthest] = True
    while len(q) > 0:
        u, d = q.popleft()
        if d > dist:
            dist = d
        for v in adj[u]:
            if not visited[v]:
                visited[v] = True
                q.append((v, d+1))

    print(dist)
```

Autor de la solución: Izan Beltrán (Miembro del comité organizador)

Solución optimizada en Python

```
from collections import deque

t = int(input())

for test in range(t):
    n = int(input())
    adj = [[] for i in range(n)]
    for i in range(n-1):
        x, y = map(int, input().split())
        adj[x].append(y)
        adj[y].append(x)
```

```
    # Encontrar el mas lejano:
    dist = 0
    furthest = -1
    q = deque()
    visited = [False for i in range(n)]
    q.append((0, 0))
    visited[0] = True
    while len(q) > 0:
        u, d = q.popleft()
        if d > dist:
            dist = d
            furthest = u
        for v in adj[u]:
            if not visited[v]:
                visited[v] = True
                q.append((v, d+1))

    # Computar diametro:
    dist = 0
    q = deque()
    visited = [False for i in range(n)]
    q.append((furthest, 0))
    visited[furthest] = True
    while len(q) > 0:
        u, d = q.popleft()
        if d > dist:
            dist = d
        for v in adj[u]:
            if not visited[v]:
                visited[v] = True
                q.append((v, d+1))

    print(dist)
```

Autor de la solución: Izan Beltrán (Miembro del comité organizador)

Solución en C++

```
#include <iostream>
#include <vector>

using namespace std;

typedef pair<int, int> pi;
typedef vector<int> vi;
typedef vector<vi> vvi;

vvi G;

/**
 * Esta funcion retorna un par de enteros
 * El primero representa la distancia maxima al vertice actual
 * Y el segundo el vertice que esta a esta distancia
 *
```

```
 * El entero p indica el ultimo vertice que hemos visitad
 * para no volver hacia atras, si no se indica es -1
 */
pi dfs(int x, int p = -1) {
  // Por defecto el vertice mas lejano es el
  // mismo y esta a distancia 0
  pi res(0, x);

  // Recorremos todos los vecinos
  for (auto y : G[x]) {
       // Si el vecino es el ultimo vertice que
       // hemos visitado continuamos sin visitarlo
       if (y == p) continue;

       // Visitamos el vecino
       pi k = dfs(y, x);

       // Si el resultado esta a distancia d
       // del vecino y, esta a d+1 de x
       k.first++;

       // Actualizamos el resultado, podemos utilizar
       // la comparacion de pares de enteros directamente
       res = max(res, k);
  }
  return res;
}

int main() {
  int t;
  cin >> t;

  while (t--) {
       int n;
       cin >> n;

       // Creamos el grafo
       G = vvi(n);
       for (int i = 1; i < n; ++i) {
       int x, y;
       cin >> x >> y;
       G[x].push_back(y);
       G[y].push_back(x);
       }


       // Primer DFS para encontrar el vertice mas lejano
       // que es uno de los extremos del diametro
       pi r = dfs(0);

       // Segundo DFS para encontrar el otro extremo del diametro
       r = dfs(r.second);

       // La longitud del diametro es la distancia entre
```

```
        // los dos extremos
        cout << r.first << endl;
    }
}
```

Autor de la solución: Cesc Folch (Miembro del comité organizador)

¡Orden!

La comisión de orden de la OIE está intentando ordenar los mensajes de emoción de los futuros participantes de la OIE 2020. Recordemos que los mensajes están definidos por cuatro enteros no negativos n_o, n_i, n_e, $n_!$, donde son el número de "*O*", el de "*I*", el de "*E*" y el de "*!*" respectivamente. Por ejemplo 3 2 1 2 sería "*OOOIIE!!*" y 0 1 2 0 sería "*IEE*".

Para poder ordenar los diferentes mensajes han definido el siguiente orden, sean m_1 y m_2 dos mensajes, si el número de caracteres de m_1 es menor que el de m_2, m_1 va antes que m_2. Si es mayor va primero . En caso de igualdad se ordenan por orden alfabético (*E* < *I* < *O* < *!*). Es decir, se ordenan por número creciente de caracteres y en caso de igualdad por orden alfabético.

Por ejemplo:

"OOO" < "OIE!"

"OIE!" < "OOOO"

"!" < "EE"

"OO!" < "O!!"

Ahora la comisión se pregunta, dado *n*, cual es el *n*-ésimo mensaje en el orden definido si consideramos todos los mensajes posibles? Recordad que el mensaje tiene que tener como mínimo una letra.

La lista de todos los mensajes empezaría con:

"E","I","O","!","EE",E!","IE","II","I!","OE","OI","OO","O!","!!","EEE","EE!","E!!","IEE","IE!","IIE"...

Estrategia de la solución

Este era el problema más difícil con diferencia, aunque conseguir alguna puntuación parcial era muy fácil (los primeros casos). Vamos a descubrir cómo conseguir los 100 puntos con la solución del autor, aunque también existen otras soluciones no tan eficientes que podían conseguir los 100 puntos.
Antes de empezar a leer la solución es importante estar familiarizado con algunos conceptos básicos de combinatoria:

- PDF corto introductorio: http://www.dmae.upct.es/~mcruiz/Telem06/Teoria/conteo2.pdf
- Página de Wikipedia: https://es.wikipedia.org/wiki/Coeficiente_binomial

Analizando los casos públicos, o haciendo algunos cálculos es fácil ver que ninguna palabra de las que nos piden va a tener más de 70000 letras (es una cota superior muy generosa).
El algoritmo va a consistir en varios pasos, en cada paso vamos a determinar una de las

características de la palabra. Primero determinaremos el número de letras, después el número de O's, seguidamente el de I's y finalmente el de E's y por lo tanto el de !'s. En la mayoría de los pasos vamos a utilizar una búsqueda binaria (aquí el enlace al artículo del manual).

Paso 1, número de letras:
Queremos contar cuántas palabras hay con *n* o menos letras, Si queremos contar cuántas palabras hay exactamente con *n* letras podemos ver que tenemos que repartir las letras en O's, I's, E's y !'s. Esto lo podemos calcular con $\binom{n+3}{3}$. Ahora si queremos calcular el número de palabras con *n* letras o menos podemos añadir un quinto carácter, el espacio, y entonces tenemos que repartir las *n* letras en O's, I's, E's, !'s y espacios, el resultado es $\binom{n+4}{4}$, tenemos que tener en cuenta que la palabra vacía, todo espacios, no puede existir. Entonces es $\binom{n+4}{4}$-1. Esta función es creciente, por lo que podemos hacer una búsqueda binaria para calcular el número de letras de nuestra palabra.

Paso 2, número de O's:
Partimos de que nuestra palabra tiene *n* letras, podemos ver que en el orden alfabético primero van a ir todas las palabras que tienen alguna I o E y luego las que solo están formadas por O's y !'s (solamente hay *n*+1 palabras de este tipo). En caso de que se trate del segundo caso, es decir, que la palabra esté entre las *n*+1 últimas de las que tienen *n* letras, podemos encontrar directamente de que palabra se trata y ya hemos terminado. En caso contrario, tenemos que nuestra palabra tiene alguna I o E, por lo tanto primero van a ir todas las palabras que no tienen ninguna O, después las que tienen una, dos, tres, etc. Igual que en el paso 1, queremos calcular cuántas palabras hay con *k* o menos O's, que contengan una E o I. Para esto vamos a contar cuántas palabras tienen más de *k* O's, y lo vamos a restar del total, que ya vimos en el paso 1 que era $\binom{n+3}{3}$. Además también vamos a tener que restar las palabras con *k* o menos O's que no contienen ni I's ni E's, que son exactamente (*k*+1) palabras. Para contar el número de palabras con más de k O's hace falta una observación, las palabras de *n* letras que empiezan con más de *k* O's son exactamente todas las palabras con *n*-(*k*+1) letras añadiendoles *k*+1 O's al principio. Visto esto y utilizando la observación del apartado 1 tenemos que este número es $\binom{n-(k+1)+3}{3}$. Y por lo tanto, el número de palabras de *n* letras que contienen alguna I o E con *k* o menos O's es: $\binom{n+3}{3}-\binom{n-(k+1)+3}{3}$-(*k*+1). Esto es creciente y por lo tanto, igual que antes podemos hacer una búsqueda binaria para encontrar el número exacto de O's.

Paso 3, número de I's:
Vamos a considerar que ahora las palabras son solo I E !, y que son de longitud *m* = *n* - *k*. Entonces vemos que este caso es muy parecido al anterior. Las últimas (*m*+1) palabras son las que no contienen ninguna E, el número total de palabras es $\binom{m+2}{2}$, en este caso solo tenemos 3 letras. Y con argumentos parecidos al apartado anterior vemos que el número de palabras con *q* o menos I's es: $\binom{m+2}{2}-\binom{m-(q+1)+2}{2}$-(*q*+1). Igual que antes hacemos la búsqueda binaria en el caso general o encontramos la palabra directamente en caso que esté entre las (*m*+1) últimas de las que tienen *n* letras y *k* O's.

Paso 4, número de E's:

Sea *l* = *m* - *q* = *n* - *k* - *q*. Ahora tenemos *l* opciones, recordemos que hay como mínimo una E, ya que en el caso anterior hemos descartado los que no tienen E's, entonces podemos encontrar directamente el número de E's que necesitamos. A partir de aquí tenemos también el número de !'s y por lo tanto ya hemos terminado.

Solución en C++

```
#include <iostream>
#include <vector>

using namespace std;

typedef long long ll;
typedef vector<ll> vi;


// Funcion que calcula el maximo comun divisor
ll gcd(ll a, ll b) {
  if (b == 0) return a;
  return gcd(b, a%b);
}

/**
 * En esta funcion calculamos n sobre k
 * Par evitar overflow primero hacemos todas
 * las divisiones y luego multiplicamos.
 *
 * Con las condiciones del problema puede
 * que no haga falta pero siempre es una
 * buena practica cuando trabajamos con numeros
 * cercanos al limite de la capacidad
 *
 * La formula es n*(n-1)*...*(n-k+1)/k*(k-1)*...*1
 */
ll nSk(ll n, ll k) {
  // Guardamos en T, los k factores del numerador
  ll T[k];
  for (int i = 0; i < k; ++i) T[i] = n-i;

  // Recorremos todos los factores del denominador
  // y los vamos dividiendo de los factores que
  // podamos
  for (ll i = 1; i <= k; ++i) {
        // Para cada valor de i recorremos todos
        // los valores de T, hasta que ya hemos utilizado
        // todos los factores primos de i
        int j = 0;
        ll val = i;
        // Cuando val == 1 ya no quedan factores primos
        // por usar
        while (val > 1) {
        // Tenemos que encontrar el siguiente T[j] que
        // comparta factores primos con val
```

```
        while (gcd(val, T[j]) == 1) ++j;

        // Una vez encontrado dividimos val y T[j]
        // entre su maximo comun divisor
        ll g = gcd(val, T[j]);
        T[j] /= g;
        val /= g;
        }
  }

  // Una vez hechas todas las division solo
  // falta multiplicar los factores del numerador
  ll res = 1;
  for (int i = 0; i < k; ++i) res *= T[i];
  return res;
}

// Es una cota superior de la longitud de las palabras
const ll M = 70000;

int main() {
  int t;
  cin >> t;

  while (t--) {
        ll n;
        cin >> n;

        // Paso 1: Encontrar la longitud de la palabra
        ll a0 = 0;
        ll b0 = M;
        while (a0 + 1 < b0) {
        ll c0 = (a0 + b0)/2;
        if (n >= nSk(c0 + 4, 4)) a0 = c0;
        else b0 = c0;
        }
        ll numll = b0;

        // La longitud es b0 = a0 + 1, ahora actualizamos n
        // El valor de n ahora es la posicion de la palabra
        // dentro de las  que tienen exactamente numll caracteres
        n -= (nSk(a0 + 4, 4) - 1);

        // Total indica el numero total de palabras
        // con numll letras
        ll total = nSk(b0 + 3, 3);

        // Paso 2: Encontrar el numero de O's

        // En caso que n este entre las (numll + 1) ultimas
        // posiciones es un caso especial donde no hay
        // ni E's ni I's
        if (n + numll >= total) {
        int kind = n + numll - total;
```

```
cout << numll - kind << " 0 0 " << kind << endl;
continue;
}

// Caso general, tenemos que encontrar el numero de O's
ll a1 = -1;
ll b1 = numll;
while (a1 + 1 < b1) {
ll c1 = (a1 + b1)/2;
ll k = total - (nSk(numll - (c1+1) + 3, 3) + (c1+1));
if (k < n) a1 = c1;
else b1 = c1;
}
ll numO = b1;

// Actualizamos el valor de n, ahora es la posicion
// dentro de las que tienen numll letras y numO O's
n -= (total - (nSk(numll - b1 + 3, 3) + b1));

// numll2 es el numero de letras sin determinar
// total2 el numero de palabras con numll letras
// y numO O's
ll numll2 = numll - numO;
ll total2 = nSk(numll2 + 2, 2);


// Paso 3: Encontrar el numero de I's

// En caso que n este entre las (numll2+1) ultimas posiciones
// es un caso especial donde no hay E's
if (n + numll2 >= total2) {
int kind = n + numll2 - total2;
cout << numO << ' ' << numll2 - kind << " 0 " << kind << endl;
continue;
}

//Caso general, tenemos que encontrar el numero de I's
ll a2 = -1;
ll b2 = numll2;
while (a2 + 1 < b2) {
ll c2 = (a2 + b2)/2;
ll k = total2 - (nSk(numll2 - (c2+1) + 2, 2) + (c2+1));
if (k < n) a2 = c2;
else b2 = c2;
}
ll numI = b2;

// n es la posicion entre las palabras que tienen
// numll letras, numO O's y numI I's
// numll3 son las letras que quedan por determinar
n -= (total2 - (nSk(numll2 - b2 + 2, 2) + b2));
ll numll3 = numll2 - numI;

// Paso 4: Encontrar el numero de E's
```

```
        // Se hace directamente con la informacion que
        // tenemos
        ll numE = numll3 - n + 1;

        // Con toda la informacion que tenemos ya conocemos la palabra
        cout << numO << ' ' << numI << ' ' << numE << ' ' << numll - numO - numE - numI <<
endl;

  }
}
```

Autor de la solución: Cesc Folch (Miembro del comité organizador)