Olimpiada Peruana de Informática 2026 Fase 1A - Solucionario

Víctor Racsó Galván Oyola
Rosangel Alexandra Bullon Linares

19 de noviembre de 2025

Tutorial del problema: "Adjacent Sort?"

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

Grupo 1 (Complejidad Esperada: $O(kn)$)

Para resolver este grupo basta con realizar la simulación de las k iteraciones del algoritmo, lo cual tomaría $O(kn)$ de complejidad.

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

Para resolver este grupo debemos realizar algunas observaciones; la primera de todas es la siguiente:

Observación 1.1. *En la i-ésima iteración de una ejecución de* `adjacent_sort`*, se cumplirá que* $\max\limits_{0 \le j < i} \{a_j\} = a_{i-1}$.

Esto quiere decir que en cada iteración haremos swap entre el máximo de los elementos a la izquierda y la posición i. Esta observación nos lleva a la siguiente como consecuencia:

Observación 1.2. *Dada una posición i con exactamente k posiciones $j < i$ tales que $a_j > a_i$; luego de una iteración de* `adjacent_sort` *habrán exactamente* $\max\{k-1, 0\}$ *posiciones $j < i$ tales que $a_j > a_i$*

Lo anterior quiere decir que la cantidad de elementos a la izquierda estrictamente mayores que a_i terminará reduciéndose en 1 luego de la iteración; además, la nueva posición de a_i será $i - 1$ debido al swap correspondiente.

Podemos usar esto para asociar a cada valor a_i con la cantidad de inversiones inv_{a_i} que tiene; es decir, la cantidad de elementos a su izquierda que sean estrictamente mayores. Podemos calcular dicha cantidad iterando de izquierda a derecha y usando un Fenwick tree para contar cuántos elementos son mayores que a_i. A esta cantidad le restaremos k (máxima cantidad de veces que dicha cantidad se reduce en 1) y si se vuelve negativa la dejaremos en 0.

Luego de haber asignado las inversiones por valor, iteraremos desde el 1 hasta el n y colocaremos el i-ésimo valor en la $(inv_i + 1)$-ésima posición libre, la cual también se puede obtener usando Fenwick tree.

Ya que todas las operaciones pueden ser realizadas en tiempo logarítmico, la complejidad se vuelve $O(n \log n)$.

Tutorial del problema: "Emparejamiento cercano"

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

Grupo 1 (Complejidad Esperada: $O(n^3)$)

Para resolver este grupo podemos mantener los valores que todavía no han sido elegidos en alguna pareja y realizar las n iteraciones.

En cada iteración, verificaremos todas las parejas y obtendremos la más adecuada en $O(n^2)$.

Finalmente, la complejidad será de $O(n^3)$.

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

Para resolver este grupo debemos notar que cualquier par elegido consistirá de dos elementos que sean consecutivos en la secuencia ordenada de valores, así que podemos ordenar a una única vez y luego comparar solo los pares de posiciones consecutivas para elegir el par más adecuado en cada iteración. Esto nos reduce un factor de n, así que la complejidad terminaría siendo $O(n^2)$.

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

Para resolver este grupo debemos usar la observación del grupo 2 e intentar mantener las diferencias de posiciones consecutivas de manera eficiente.

Para ello usaremos un `std::set` S que contenga las tuplas $(-diferencia, suma, i, j)$ que representen al negativo de la diferencia, la suma y las dos posiciones $(i < j)$ del arreglo ordenado para poder realizar inserción, eliminación y consulta del máximo en tiempo $O(\log n)$ o inferior. Para poder eliminar una posición i tendremos que eliminar de S la tupla correspondiente a i con su predecesor en los valores ordenados restantes (sea L_i) y también la tupla correspondiente a i con su sucesor (sea R_i) y agregar la tupla entre L_i y R_i en caso ambos sean posiciones válidas; por último se deben redirigir los L y R de R_i y L_i, respectivamente. Obviamente aplicaremos este procedimiento tanto a i como a j.

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

Tutorial del problema: "Carrera arreglada"

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

Grupo 1 (Complejidad Esperada: $O(n!n)$)

Para resolver este grupo podemos probar con todas las permutaciones p posibles de las intersecciones para fijar el ciclo que seguiremos. Una vez fijada la permutación, podemos iterar sobre todas las longitudes válidas que puedan llegar al mismo nodo p_0.

Ya que tendremos un trabajo de $O(n)$ por cada permutación, la complejidad será de $O(n!n)$.

Grupo 2 (Complejidad Esperada: $O(n^4)$)

Para resolver este grupo podemos hacer la observación de que cualquier ciclo válido tendrá una longitud que no exceda a n. Por otro lado, consideraremos el siguiente grafo:

$$V = \{1, 2, \ldots, n\}$$
$$E = \{(U_i, V_i) | \forall\ i = 1, \ldots, m\}$$
$$w(U_i, V_i) = B_i - A_i\ \forall\ i = 1, \ldots, m$$

Notemos que si fijamos una longitud L, vamos a querer calcular algún ciclo de longitud L con suma de pesos positiva, para lo cual nos basta calcular la máxima suma de pesos — Si esta es positiva, entonces sí existe alguna respuesta; en caso contrario, ningun cíclo de longitud L tiene suma de pesos válida.

Para poder calcular los ciclos y caminos de longitud L usaremos la matriz de adyacencia M del grafo, donde

$$M_{i,j} = \begin{cases} 0 & i = j \\ -\infty & \text{No existe la arista } (i,j) \\ w(i,j) & \text{Existe la arista } (i,j) \end{cases}$$

Si operamos bajo la función $(\text{máx}, +)$ para el producto de las matrices (notemos que el producto típico es $(+, \times)$), podremos calcular el máximo camino de longitud L del nodo i al nodo j en la matriz M^L.

Algoritmo 1: PRODUCTO-MATRICES(n, A, B)

```
Sea C una nueva matriz de dimensión n ;
para i ← 1 a n hacer
    para j ← 1 a n hacer
        C[i][j] ← ∞ ;
        para k ← 1 a n hacer
            C[i][j] ← máx{C[i][j], A[i][k] + B[k][j]}
devolver C.
```

Si probamos todas las posibles longitudes, tendremos una complejidad de $O(n)$ productos de matriz, lo cual termina con $O(n^4)$ de complejidad.

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

Para resolver este grupo aplicaremos la misma idea que el grupo 2, pero usaremos binary lifting para encontrar la primera longitud L tal que el máximo de los ciclos de M^L es positivo. Para esto, calcularemos todas las matrices M^{2^k} para todo $k = 0, \ldots, \lfloor \log_2 n \rfloor$.

Luego, mantendremos una matriz X de referencia (inicializada en como la matriz M pero considerando un grafo con n nodos sin aristas, denotaremos esta matriz por I_n) y tomaremos una decisión dependiendo del producto de $X \times M^{2^k}$:

- Si $X \times M^{2^k}$ tiene un ciclo de peso positivo, entonces no hacemos nada.
- Si $X \times M^{2^k}$ no tiene un ciclo de peso positivo, asignamos $X \leftarrow X \times M^{2^k}$.

Luego de obtener el X final, este reflejará la longitud máxima tal que no hay un ciclo positivo todavía, así que si calculamos $X \times M$ obtendremos la menor longitud con un ciclo positivo.

Algoritmo 2: SOLUCION(n, M)

```
Sea P[⌊log2 n⌋ + 1] un arreglo de matrices ;
P[0] ← M ;
para k ← 1 a ⌊log2 n⌋ hacer
    P[i] ← P[i − 1] × P[i − 1] ;
X ← I_n ;
L ← 0 ;
para k ← ⌊log2 n⌋ a 0 hacer
    X' ← X × P[k] ;
    si máx_{i=1}^{n} {X'_{i,i}} ≤ 0 entonces
        X ← X' ;
        L ← L + 2^k ;
L ← L + 1 ;
X ← X × M ;
devolver {L, máx_{i=1}^{n} {X_{i,i}}}
```

Ya que k tiene una cantidad $O(\log n)$ de valores diferentes, la complejidad termina siendo $O(n^3 \log n)$.

Tutorial del problema: "Un problema de Sumatorias"

Autor(es): Rosangel Bullon
Desarrollador(es): Rosangel Bullon

Nota: Asumir que las operaciones en los pseudocódigos mostrados en esta sección son todas modulares con el módulo del problema.

Grupo 1 (Complejidad Esperada: $O(M\sqrt{M}+T)$)

Para resolver este problema plantearemos un preprocesamiento tomando como referencia el máximo valor de N, el cual denotaremos por M. Como este valor es a lo más 10^4 podemos hacer esto con fuerza bruta para saber los valores $f(N)$ y guardarlos en un arreglo para al final de saber todos los valores, guardar la sumatoraria de dichos valores en otro arreglo y así se puedan responder las consultas en $O(1)$.

Denotaremos $g(n) = n^2 + 2026n - 2026$, entonces el enunciado señala que:

$$g(n) = \sum_{d|n} f(d)$$

Así que podemos deducir el valor correcto de f de la siguiente manera:

$$g(n) = \sum_{\substack{d|n \\ d<n}} f(d) + f(n) \rightarrow f(n) = g(n) - \sum_{\substack{d|n \\ d<n}} f(d)$$

Así que solo nos basta iterar sobre n de manera creciente y calcular $f(n)$ en función de sus divisores estrictamente menores.

Este procedimiento se vería de esta forma:

Algoritmo 3: PREPROCESAMIENTO(M)

```
Sea f un arreglo de enteros de tamaño M + 1 ;
f[1] ← 1 ;
para i ← 2 a M hacer
    f[i] ← n^2 + 2026n − 2026 ;
    f[i] ← f[i] − f[1] ;
    para j ← 2 a √i hacer
        // Solo necesitamos verificar hasta la √M para hallar los divisores ;
        si j es divisor de i entonces
            f[i] ← f[i] − f[j] ;
            si j ∗ j ≠ i entonces
                f[i] ← f[i] − f[⌊i/j⌋] ;
Sea res un arreglo de enteros de tamaño M + 1 inicializado en 0 ;
para i ← 1 a M hacer
    res[i] ← res[i − 1] + f[i] ;
```

La complejidad será de $O(M\sqrt{M}+T)$ porque las consultas se responden usando el arreglo res.

Grupo 2 (Complejidad Esperada: $O(M \log M + T)$)

Para este grupo el valor máximo de N puede ser tan grande como 10^6, por lo que el enfoque del Grupo 1, que realizaba una búsqueda de divisores para cada número, resulta demasiado lento. Por ello utilizaremos una técnica de preprocesamiento basada en recorrer múltiplos, lo cual es mucho más eficiente.

La idea general es la siguiente:

- Mantendremos un arreglo g donde $g[n]$ almacena temporalmente la suma $\sum_{d|n} f(d)$, de la cual iremos quitando los valores $f(d)$, donde d es un divisor de n con $d < n$.

Entonces:

$$g[n] = n^2 + 2026n - 2026.$$

- Cuando procesemos un número n, ya tendremos en $g[n]$ el valor de $f(n)$. Al tener el valor de $f(n)$ lo que haremos después sera visitar todos los múltiplos de n que son mayores que él para propagar la resta de los $f(d)$.

  Entonces:

$$S[k] \mathrel{-}= f(n) \qquad \text{para } k = 2n, 3n, 4n, \ldots$$

A continuación mostramos el procedimiento en forma de algoritmo.

Algoritmo 4: PREPROCESAMIENTO(M)

```
Sea g un arreglo de enteros de tamaño M + 1 inicializado en 0 ;
Sea res un arreglo de enteros de tamaño M + 1 inicializado en 0 ;
// Aquí inicializamos g[n] = ∑_{d|n} f(d) ;
para n ← 1 a M hacer
    g[n] ← n^2 + 2026n − 2026;
// Aquí quitaremos el valor f(d) en S[n] para todo n que es múltiplo de d. ;
para i ← 1 a M hacer
    //Para este punto g[i] será el valor de f(i). ;
    j ← 2i ;
    mientras j ≤ M hacer
        g[j] ← g[j] − g[i] ;
        j ← j + i ;
para i ← 1 a M hacer
    res[i] ← res[i − 1] + f[i] ;
```

La complejidad final es:

$$O(M \log M + T),$$

ya que las consultas se resuelven en $O(1)$ utilizando el arreglo de prefijos res.

Grupo 3 (Complejidad Esperada: $O(n^{\frac{2}{3}})$ por caso)

Para resolver este grupo debemos plantear el uso de la Convolución de Dirichlet ($*$), la cual está definida así:

Definición 4.1 (Convolución de Dirichlet)**.** *La convolución de dirichlet de dos funciones aritméticas f y g está denotada por $f * g$ y tiene la siguiente definición:*

$$(f * g)(n) = \sum_{d|n} f(d) \times g\left(\frac{n}{d}\right)$$

En este caso, tenemos que:

$$g(n) = \sum_{d|n} f(d) = f * 1$$

Donde 1 es la función constante $h(n) = 1$, y deseamos calcular $\sum_{i=1}^{n} f(i)$.

En un caso general, para dos funciones f y g, podemos plantear que:

$$(f * g)(n) = \sum_{d|n} f(d) \times g\left(\frac{n}{d}\right) = \sum_{d|n} g(d) \times f\left(\frac{n}{d}\right)$$

$$\sum_{i=1}^{n}(f * g)(i) = \sum_{i=1}^{n} g(i) \times \sum_{j=1}^{\left\lfloor \frac{n}{i} \right\rfloor} f(j)$$

Si denotamos por $F(n) = \sum_{i=1}^{n} f(i)$, tendremos:

$$\sum_{i=1}^{n}(f * g)(i) = \sum_{i=1}^{n} g(i) \times F\left(\left\lfloor \frac{n}{i} \right\rfloor\right)$$

Si aislamos $i = 1$ en el lado derecho:

$$\sum_{i=1}^{n}(f * g)(i) = g(1) \times F(n) + \sum_{i=2}^{n} g(i) \times F\left(\left\lfloor \frac{n}{i} \right\rfloor\right)$$

$$F(n) = \frac{\sum_{i=1}^{n}(f * g)(i) - \sum_{i=2}^{n} g(i) \times F\left(\left\lfloor \frac{n}{i} \right\rfloor\right)}{g(1)}$$

Reemplazando los datos del problema; es decir, $f * g = g(n)$, $f = f$ y $g = 1$:

$$F(n) = \sum_{i=1}^{n}(i^2 + 2026i - 2026) - \sum_{i=2}^{n} F\left(\left\lfloor \frac{n}{i} \right\rfloor\right)$$

$$F(n) = \frac{n(n+1)(2n+1)}{6} + 1013n(n+1) - 2026n - \sum_{i=2}^{n} F\left(\left\lfloor \frac{n}{i} \right\rfloor\right)$$

Notemos que hay $O(\sqrt{n})$ valores diferentes de $\left\lfloor \frac{n}{i} \right\rfloor$ (para el cual se tiene un algoritmo clásico), así que si almacenamos las respuestas parciales en un `std::unordered_map` podremos tener una complejidad de $O(n^{\frac{3}{4}})$; sin embargo, esto no es suficientemente rápido. Es posible mejorar la complejidad preprocesando la respuesta de todos los $F(i)$ con $i \leq K$, donde la elección de K definirá la complejidad final. Para este caso, tomar $K \approx n^{\frac{2}{3}}$ nos genera una complejidad de $O(n^{\frac{2}{3}})$ por caso.

Algoritmo 5: F(n)

```
si n ≤ 10^6 entonces
    devolver res[n] ;
si memo contiene a n como llave entonces
    devolver memo[n] ;
res ← n(n+1)(2n+1)/6 + 1013n(n+1) − 2026n ;
l ← 2 ;
mientras l ≤ n hacer
    x ← ⌊n/l⌋ ;
    r ← ⌊n/x⌋ ;
    // Todos los valores i ∈ [l, r] aportan con el mismo F(x) ;
    res ← res − (r − l + 1) × F(x) ;
    l ← r + 1 ;
memo[n] ← res ;
devolver memo[n] ;
```

Donde *memo* es un `std::unordered_map` global que almacena las respuestas y se asume que se ha ejecutado PREPROCESAMIENTO(10^6) antes de responder a las consultas.