

Cuadernillo de Soluciones

2da Fase de la Competencia Boliviana de Programación
ICPC Bolivia 2026

2026

Nota: este documento contiene la solución completa de cada problema, con pistas progresivas al estilo de un editorial de Codeforces. No lo distribuya a concursantes antes de que termine el concurso.

Problema A — Fronteras

Dificultad estimada: Medio-fácil (aprox. 1500) • *Temas: Geometría computacional y Union-Find*

Pistas

Pista 1. Cada conflicto solo une dos comunidades, nunca las separa. Eso suena a un problema donde conviene mantener una estructura de conjuntos disjuntos (Union-Find / DSU) indexada por pueblo, y guardar *algo* asociado a cada componente.

Pista 2. No necesitas guardar todos los pueblos de una comunidad para responder la pregunta. El área del territorio depende solo de su envolvente convexa, así que basta con guardar, para cada componente, los vértices de su propia envolvente.

Pista 3. Cuando fusionas dos comunidades, ¿alcanza con juntar las dos fronteras (las dos listas de vértices) y ya tener la frontera de la unión? Dibuja dos triángulos cercanos y únelos: ¿algún vértice de un triángulo puede quedar *adentro* del pentágono/hexágono resultante?

Pista 4. Sí puede pasar (mira el ejemplo 3 del enunciado). Por lo tanto, tras unir dos componentes hay que recalcular la envolvente convexa de la unión de sus vértices *desde cero*, no simplemente concatenar las dos listas. Un algoritmo tipo *monotone chain* (Andrew) hace esto en $O(k \log k)$, donde k es la cantidad de vértices que se están uniendo.

Pista 5. El área se pide como el doble del área real, precisamente para poder calcularla con enteros usando la fórmula de Gauss (*shoelace*) sin preocuparte por fracciones. ¿Qué tan grande es en el peor caso la suma de tamaños de los conjuntos que se recalculan a lo largo de todos los conflictos? Con $N \leq 2000$, esa cota total termina siendo manejable incluso si cada fusión cuesta $O(N \log N)$.

Solución

Mantenemos una estructura Union-Find sobre los N pueblos. Lo importante es qué información guarda cada componente: **no** guardamos todos los pueblos que pertenecen a ella, sino únicamente los vértices de su envolvente convexa actual. Al inicio, cada pueblo es su propia comunidad y su "envolvente" es el punto único.

Cuando llega un conflicto entre los pueblos a y b :

- Si ya están en la misma componente ($\text{Find}(a) == \text{Find}(b)$), no hay nada que hacer: la envolvente no cambia.
- Si están en componentes distintas, las fusionamos con el DSU (unión por tamaño/rango para mantener la complejidad amortizada), y **recalculamos desde cero** la envolvente convexa de la unión: se concatenan los vértices de ambas envolventes en una sola lista de puntos y se corre *monotone chain* sobre esa lista completa. El resultado reemplaza la envolvente guardada en la raíz del componente unido; el componente que dejó de ser raíz puede limpiar su lista (ya no se consulta más).

¿Por qué hay que recalcular en vez de simplemente combinar las dos fronteras (por ejemplo, "empalmarlas" ordenadas angularmente)? Porque un vértice que era extremo de *su propia* comunidad puede quedar estrictamente en el interior de la envolvente de la unión. El ejemplo 3 del enunciado es exactamente este caso: dos triángulos que, al fusionarse, forman un pentágono, y uno de los seis vértices originales queda adentro. No hay forma barata de saber de antemano cuál sobrevive sin, en esencia, volver a correr el algoritmo de envolvente convexa sobre el conjunto unido.

Una vez tenemos los vértices de la envolvente (en orden, sea horario o antihorario), el área con signo se calcula con la fórmula de Gauss:

$$2 \cdot \text{Área} = \left| \sum_{i=0}^{k-1} (x_i y_{i+1} - x_{i+1} y_i) \right|,$$

donde los índices son módulo k (el vértice después del último es el primero). Si la envolvente tiene menos de 3 vértices (comunidad de tamaño 1 o 2, o todos sus puntos colineales), el área es 0. Como todas las coordenadas son enteras, esta expresión se calcula exactamente con aritmética entera de 64 bits (`long long`), y por eso el problema pide el *doble* del área: así la respuesta siempre es un entero.

Después de cada conflicto (haya habido fusión real o no), se imprime el área doble de la envolvente de la componente que ahora contiene a a (y a b , es la misma).

Fragmento clave de la fusión (adaptado de `sols/main.cpp`):

```
1  int root = dsu.Union(x, y);
2  int other = (root == x) ? y : x;
3
4  vector<Point> merged = hull[root];
5  for (int k = 0; k < hull[other].size(); k++)
6      merged.push_back(hull[other][k]);
7  hull[root] = convexHull(merged);
8  hull[other].clear();
```

Complejidad. Tiempo: $O(N^2 \log N + M)$ Espacio: $O(N)$

Cada uno de los $N - 1$ conflictos que realmente fusiona dos componentes recalcula una envolvente sobre, en el peor caso, $O(N)$ puntos en $O(N \log N)$; el resto de los conflictos (hasta $M - (N - 1)$ de ellos) ya están en la misma componente y cuestan $O(1)$. Con $N \leq 2000$ esto es totalmente viable dentro de 2 segundos.

Problema B — MiniMax

Dificultad estimada: Medio (aprox. 1950) • *Temas: Estructuras de datos, greedy, binary lifting*

Pistas

Pista 1. Fija un punto de partida i y olvídate por un momento de las consultas. ¿Cuál es, para ese único i , el segmento válido (con $\max - \min \leq k$) más largo que se puede formar empezando en i ? ¿Ese segmento es único?

Pista 2. Para resolver una consulta $[l, r]$ con la menor cantidad de segmentos, conviene que cada segmento sea lo más largo posible. Un argumento de intercambio muestra que nunca es mejor cortar antes de lo necesario: si un segmento puede seguir creciendo sin romper la condición, cortarlo antes nunca ayuda y puede obligarte a usar un segmento extra más adelante.

Pista 3. Define $\text{next}[i]$ como la primera posición que, si se agrega al segmento que arranca en i , hace que $\max - \min$ supere k (es decir, $[i, \text{next}[i] - 1]$ es el segmento válido más largo que empieza en i). Calcular $\text{next}[i]$ para un solo i por fuerza bruta cuesta $O(n)$; hacerlo así para los n valores de i sería $O(n^2)$. ¿Se puede calcular *todo* el arreglo $\text{next}[\cdot]$ en una sola pasada de $O(n)$ total, con dos punteros y algo que te dé el máximo y el mínimo de una ventana en $O(1)$ amortizado?

Pista 4. Con $\text{next}[\cdot]$ ya calculado, cada consulta se podría responder saltando $l \rightarrow \text{next}[l] \rightarrow \text{next}[\text{next}[l]] \rightarrow \dots$ hasta pasarte de r , contando los saltos. Pero eso cuesta hasta $O(n)$ por consulta, y con q hasta 10^5 consultas eso es demasiado lento. ¿Qué técnica clásica permite *duplicar* la longitud de un salto en cada nivel para recorrer una cadena de saltos en $O(\log n)$?

Pista 5. Construye una tabla de *binary lifting* $\text{up}[j][i]$ = posición a la que se llega tras aplicar 2^j veces el salto next desde i (con $\text{up}[0][i] = \text{next}[i]$ y $\text{up}[j][i] = \text{up}[j-1][\text{up}[j-1][i]]$). Para responder $[l, r]$, recorre los niveles de mayor a menor y tomá el salto de 2^j segmentos siempre que el destino no se pase de r ; al final sumá 1 por el último segmento, que queda recortado en r .

Solución

La observación clave es que la partición óptima siempre puede construirse de forma **greedy**: para minimizar el número de segmentos en $[l, r]$, cada segmento debe extenderse tanto como sea posible antes de cortar. Si en algún punto cortáramos un segmento antes de que realmente se vuelva inválido, nunca ganamos nada (el segmento restante que queda a la derecha solo se hace más difícil de cubrir), y en el peor caso terminamos necesitando un segmento adicional. Por lo tanto, alcanza con saber, para cada posición i , cuál es el segmento válido más largo que arranca ahí.

Definimos $\text{next}[i]$ como la primera posición (0-indexada) tal que $[i, \text{next}[i] - 1]$ es válido (diferencia $\leq k$) pero agregar el elemento en $\text{next}[i]$ ya no lo es. Este valor se calcula para *todos* los i en una sola pasada usando dos punteros: mantenemos una ventana $[\text{left}, \text{right}]$ y dos colas

monotónicas – una creciente que siempre tiene al frente el índice del mínimo de la ventana, y otra decreciente que tiene al frente el índice del máximo. Cada vez que la ventana deja de ser válida (el $\max - \min$ de las colas supera k), fijamos $\text{next}[\text{left}] = \text{right}$ y avanzamos left , expulsando de las colas los índices que quedan fuera de la nueva ventana. Cada índice entra y sale de cada cola a lo sumo una vez, así que el costo total es $O(n)$.

Con $\text{next}[\cdot]$ calculado, cada consulta $[l, r]$ podría responderse saltando next repetidamente hasta pasarse de r , pero eso es $O(n)$ por consulta en el peor caso (por ejemplo, un arreglo que alterna $-10^9/10^9$, donde cada segmento válido tiene longitud 1). Para acelerar esto usamos **binary lifting**: precomputamos una tabla $\text{up}[j][i]$ = posición tras 2^j saltos de next desde i , con $\text{up}[0][i] = \text{next}[i]$ y $\text{up}[j][i] = \text{up}[j-1][\text{up}[j-1][i]]$. Para responder una consulta, recorremos los niveles j de mayor a menor: si $\text{up}[j][\text{pos}] \leq r$ (0-indexado), tomamos el salto (avanzamos pos y sumamos 2^j al contador de segmentos); al terminar, sumamos 1 por el último segmento, que llega hasta r recortado.

Complejidad. Tiempo: $O((n + q) \log n)$ Espacio: $O(n \log n)$

Un fragmento del cálculo de $\text{next}[\cdot]$ (equivalente a $\text{up}[0][\cdot]$) con las dos colas monotónicas:

```

1  std::deque<int> minimum, maximum;
2  int left = 0;
3  for (int right = 0; right < n; ++right) {
4      while (!minimum.empty() && a[minimum.back()] >= a[right])
5          minimum.pop_back();
6      minimum.push_back(right);
7      while (!maximum.empty() && a[maximum.back()] <= a[right])
8          maximum.pop_back();
9      maximum.push_back(right);
10
11     while (a[maximum.front()] - a[minimum.front()] > k) {
12         up[0][left] = right; // next[left] = right
13         if (minimum.front() == left) minimum.pop_front();
14         if (maximum.front() == left) maximum.pop_front();
15         ++left;
16     }
17 }
```

Y la respuesta de una consulta $[l, r]$ (ya 0-indexada) usando la tabla de binary lifting:

```

1  int position = l;
2  int answer = 1; // el ultimo segmento, recortado en r
3  for (int level = levels - 1; level >= 0; --level) {
4      if (up[level][position] <= r) {
5          position = up[level][position];
6          answer += (1 << level);
7      }
8  }
```

Problema C — Leña en Corani

Dificultad estimada: Fácil (aprox. 1300) • *Temas: Implementación, estructuras de datos*
(`std::set`)

Pistas

Pista 1. Olvidáte de la pirámide completa por un momento y mira solo la base. Cuando Pakai retira una posición, ésta parte la base en pedazos. Un tronco de una fila superior depende de dos apoyos consecutivos: ¿qué tiene que pasar en la base para que un tronco arriba siga en pie?

Pista 2. Un tronco de la fila superior sobrevive si y solo si las dos posiciones de la base de las que depende (y, transitivamente, todo el rango de posiciones de la base bajo él) siguen intactas. En otras palabras: un tramo **contiguo** de posiciones intactas de la base sostiene una sub-pirámide propia, independiente de los demás tramos.

Pista 3. ¿Cuántos troncos en total sostiene un tramo intacto de longitud L (contando el propio tramo de la base y todo lo que se apila encima)? Es una suma $L + (L - 1) + \dots + 1$: un número triangular.

Pista 4. Cada retiro parte un tramo existente en dos tramos más chicos (o lo deja en uno solo, si se retira justo en un extremo). Si en todo momento sabes la longitud del tramo que contiene a la posición retirada, puedes actualizar la respuesta total en tiempo constante por operación, sin reconstruir nada. Necesitas una estructura que te dé, para cualquier posición, sus vecinos retirados más cercanos a la izquierda y a la derecha, en tiempo logarítmico: un `std::set` con las posiciones retiradas (más dos centinelas en 0 y $N + 1$ para no tratar los extremos como casos especiales) resuelve exactamente eso.

Solución

La observación clave es que un tronco de una fila superior está en pie si y solo si todo un rango contiguo de posiciones de la base bajo él sigue intacto. Por lo tanto, la pila se descompone en tiempo real en **tramos maximales de posiciones intactas** de la base, y cada tramo sostiene su propia sub-pirámide, sin interactuar con los demás tramos.

Un tramo intacto de longitud L sostiene exactamente

$$T(L) = 1 + 2 + \dots + L = \frac{L(L + 1)}{2}$$

troncos en total (el propio tramo de la base, más una fila de $L - 1$ encima, etc., hasta el ápice de esa sub-pirámide), con $T(0) = 0$. La respuesta en todo momento es simplemente la suma de $T(L)$ sobre todos los tramos intactos vivos, y al inicio (sin retiros) es un único tramo de longitud N , es decir $T(N)$.

Cada retiro de una posición p toma el tramo que contenga a p (digamos que va de $a + 1$ a $b - 1$, con longitud $b - a - 1$, donde a y b son las posiciones retiradas más cercanas a la izquierda

y a la derecha de p) y lo reemplaza por dos tramos nuevos: uno de longitud $p - a - 1$ y otro de longitud $b - p - 1$. La respuesta se actualiza en $O(1)$:

$$\text{respuesta} -= T(b - a - 1); \quad \text{respuesta} += T(p - a - 1) + T(b - p - 1).$$

Para encontrar a y b rápido, se mantiene un `std::set` con todas las posiciones ya retiradas, más dos centinelas fijos en 0 y en $N + 1$ (así un tramo siempre tiene vecinos válidos, aunque toque un extremo real de la base). Con `lower_bound` se ubica b en $O(\log Q)$, y su predecesor en el set es a . Después se inserta p al set y se sigue con el siguiente retiro.

```

1  std::set<i64> removed{0, n + 1};
2  i64 remaining = triangular(n);
3
4  for (int i = 0; i < q; ++i) {
5      i64 p; std::cin >> p;
6      const auto right_it = removed.lower_bound(p);
7      const i64 right = *right_it;
8      const i64 left = *std::prev(right_it);
9
10     remaining -= triangular(right - left - 1);
11     remaining += triangular(p - left - 1);
12     remaining += triangular(right - p - 1);
13     removed.insert(right_it, p);
14
15     std::cout << remaining << '\n';
16 }
```

Un cuidado importante: $T(L)$ y la respuesta acumulada pueden llegar hasta del orden de $N^2 \approx 10^{18}$, así que hay que usar enteros de 64 bits (`int64_t/long long`) en todo el cálculo, nunca `int` de 32 bits.

Complejidad. Tiempo: $O(Q \log Q)$ Espacio: $O(Q)$

Problema D — Galletas para el feed

Dificultad estimada: Fácil (aprox. 1100) • Temas: Implementación / simulación

Pistas

Pista 1. Cada letra del alfabeto es completamente independiente de las demás: los moldes de A nunca se usan para hornear una B. Pensá el problema letra por letra en vez de palabra por palabra.

Pista 2. Para una letra fija, lo único que importa de cada tanda es *cuántas veces* aparece esa letra en la palabra de esa tanda. El resto de la palabra es irrelevante para esa letra.

Pista 3. Un molde usado en la tanda i no se puede reutilizar en la tanda $i + 1$ (sigue enfriándose), pero sí en la tanda $i + 2$. Es decir, en cualquier par de tandas *consecutivas* los moldes usados no se pueden compartir: hacen falta moldes distintos para cada uso en esas dos tandas.

Pista 4. Para una letra x , mirá cada par de tandas consecutivas $(i - 1, i)$ y sumá cuántas veces aparece x en esas dos palabras juntas. El peor par (el que pide más moldes de esa letra en simultáneo) determina cuántos moldes de x hay que comprar.

Solución

Como cada letra usa su propio conjunto de moldes, alcanza con resolver el problema por separado para cada una de las 26 letras y después sumar los resultados.

Fijémonos en una letra x . Sea $c_i(x)$ la cantidad de veces que x aparece en la palabra de la tanda i (con $c_0(x) = 0$, ya que antes de la primera tanda no se había usado ningún molde). La regla del enunciado dice que los moldes de x usados en la tanda $i - 1$ todavía están enfriándose durante la tanda i , así que no sirven para esa tanda: hacen falta moldes *nuevos* (o ya fríos, de dos tandas atrás o más) para cubrir los $c_i(x)$ usos de la tanda actual.

Esto quiere decir que en cualquier tanda i , la cantidad de moldes de x que están "comprometidos" (usados en la tanda anterior o en la actual) es $c_{i-1}(x) + c_i(x)$, y esa cantidad nunca puede superar el total de moldes de x que compramos. Por lo tanto, el mínimo de moldes de x necesarios es

$$M(x) = \max_{1 \leq i \leq N} (c_{i-1}(x) + c_i(x)),$$

es decir, el mayor valor entre dos tandas consecutivas cualesquiera. Alcanza con recorrer las tandas una vez, guardando el conteo por letra de la tanda anterior, e ir actualizando $M(x)$ con $c_{i-1}(x) + c_i(x)$ en cada paso.

La respuesta final es simplemente $\sum_x M(x)$ sobre las 26 letras.

Con el primer ejemplo (ANA, LIA, ANA) se ve la idea: para la letra A, $c_1 = 2$, $c_2 = 1$, $c_3 = 2$, y el máximo de sumas consecutivas es $\max(2 + 1, 1 + 2) = 3$; para N, $\max(1 + 0, 0 + 1) = 1$; para L e I, 1 cada una. En total $3 + 1 + 1 + 1 = 6$ moldes, que es la respuesta esperada.

Complejidad. Tiempo: $O(L + 26N)$ Espacio: $O(L_{\max} + 26)$

donde L es la suma de las longitudes de todas las palabras, N el número de tandas y $L_{\max}$ la longitud de la palabra más larga (no hace falta guardar todas las palabras a la vez, solo la tanda actual y la anterior).

El núcleo del algoritmo es muy corto:

```
1  std::array<int, 26> anterior{};
2  std::array<int, 26> necesarios{};
3
4  for (int i = 0; i < n; ++i) {
5      std::string palabra;
6      std::cin >> palabra;
7      std::array<int, 26> actual{};
8
9      for (char letra : palabra)
10         ++actual[static_cast<std::size_t>(letra - 'A')];
11
12     for (std::size_t letra = 0; letra < actual.size(); ++letra)
13         necesarios[letra] = std::max(necesarios[letra],
14                                       anterior[letra] + actual[letra]);
15     anterior = actual;
16 }
```

Problema E — Número Waso

Dificultad estimada: Medio-fácil (aprox. 1700) • Temas: Combinatoria (combinaciones con repetición / stars and bars)

Pistas

Pista 1. Un número waso es simplemente un entero no negativo cuyos dígitos, leídos de izquierda a derecha, nunca decrecen (por ejemplo, 1, 23, 113, 99). Antes de pensar en cómo hallar el k -ésimo, preguntáte: para una longitud L fija, ¿cuántos números wasos de exactamente L dígitos existen?

Pista 2. Un número waso de L dígitos (sin cero a la izquierda) es lo mismo que elegir, sin importar el orden, un multiconjunto de tamaño L de los dígitos $\{1, \dots, 9\}$: una vez elegido el multiconjunto, hay una única forma de ordenarlo de manera no decreciente. Contar multiconjuntos es el problema clásico de *combinaciones con repetición*, que se resuelve con la técnica de *stars and bars*.

Pista 3. Como k puede llegar hasta 10^{18} , la respuesta puede tener centenares de dígitos: no puedes generar los wasos uno por uno ni guardarlos en un entero de 64 bits. Necesitas primero **fixar la longitud L** del número buscado, comparando k contra sumas acumuladas de los conteos por longitud (y esas sumas acumuladas también tienen una fórmula cerrada, no hace falta sumarlas una por una).

Pista 4. Una vez fijada L , construye el número **dígito por dígito**, de izquierda a derecha: en cada posición prueba el menor dígito válido (mayor o igual al último elegido) y usa el conteo de cuántas maneras hay de completar las posiciones restantes con dígitos $\geq d$ para decidir si el k -ésimo cae dentro de ese bloque o si hay que descartarlo (restando su tamaño de k) y probar con $d + 1$. Es la misma idea que hallar el k -ésimo elemento de un orden lexicográfico, pero contando con binomiales en vez de recorrer explícitamente todas las posibilidades.

Pista 5. Todos los binomiales que necesitas calcular son pequeños ($\binom{n}{r}$ con $r \leq 9$ y n acotado por unos pocos cientos): caben perfectamente en un entero de 64 bits (**long long**). Lo único que sí puede ser enorme (hasta ~ 410 dígitos) es la **respuesta final**, y esa se construye directamente como **string**, nunca como un número.

Solución

Un número waso es un entero no negativo cuyos dígitos, leídos de izquierda a derecha, forman una secuencia no decreciente. Vamos a contar y construir la respuesta en tres pasos.

Paso 1: los casos triviales de un dígito. Los primeros 10 números wasos son, en orden, 0, 1, 2, $\dots$, 9 (un solo dígito siempre es trivialmente no decreciente, y el enunciado considera al 0 como el primero). Así que si $k \leq 10$, la respuesta es simplemente $k - 1$.

Paso 2: contar por longitud y fijar L . Para $k > 10$, restamos esos 10 casos y buscamos la longitud $L \geq 2$ del número buscado. Un waso de L dígitos sin cero a la izquierda es una secuencia no decreciente de L dígitos de $\{1, \dots, 9\}$, que equivale a elegir un multiconjunto de tamaño L de esos 9 símbolos. Por *stars and bars* hay exactamente

$$\binom{L+8}{8}$$

tales secuencias. Recorremos $L = 2, 3, 4, \dots$ acumulando este conteo y restando de k hasta que el conteo de la longitud actual ya alcance el k restante: esa es la longitud correcta.

Paso 3: construir la respuesta dígito por dígito. Con L fijo, construimos el número de izquierda a derecha. Sea

$$\text{seqCount}(\ell, d) = \binom{\ell + (9 - d)}{9 - d}$$

el número de secuencias no decrecientes de longitud ℓ usando solo dígitos de $\{d, \dots, 9\}$. En cada posición, con `low` igual al último dígito colocado (o 1 en la primera posición, para evitar el cero a la izquierda), probamos $d = \text{low}, \text{low} + 1, \dots, 9$: si $k \leq \text{seqCount}(\text{restantes}, d)$, colocamos ese dígito y seguimos; si no, restamos ese conteo de k y probamos el siguiente d .

```

1 long long seqCount(long long len, int d) {
2     return nCr((int)len + (9 - d), 9 - d);
3 }
4 ...
5 string res;
6 int low = 1;
7 for (long long pos = 0; pos < L; pos++) {
8     for (int d = low; d <= 9; d++) {
9         long long cnt = seqCount(L - pos - 1, d);
10        if (k <= cnt) { res += char('0' + d); low = d; break; }
11        k -= cnt;
12    }
13 }
```

Todos los binomiales involucrados tienen $r \leq 9$ y n acotado por unos pocos cientos (para $k \leq 10^{18}$ la longitud máxima resulta $L = 410$), así que se calculan sin problema con una tabla de Pascal en `long long` precomputada una sola vez; no hace falta aritmética de precisión arbitraria para los conteos, solo para *imprimir* la respuesta (que se construye directamente como cadena de caracteres).

Complejidad. Tiempo: $O(L_{\max}) \approx O(410)$ por consulta, tras un precomputo $O(500 \times 10)$ de la tabla de binomiales Espacio: $O(500 \times 10)$ para la tabla + $O(L_{\max})$ para la respuesta

Problema F — Luces para la SanSiCup

Dificultad estimada: Muy difícil (aprox. 2450) • Temas: Álgebra lineal sobre $\mathbb{F}_2$ (base XOR) y códigos lineales

Pistas

Pista 1. Una configuración de luces alcanzable es exactamente el XOR de las máscaras de algún subconjunto de los controles originales. Es decir, el conjunto de configuraciones alcanzables es el *espacio generado* (sobre $\mathbb{F}_2$) por los N vectores $a_1, \dots, a_N$. Lo único que le importa al problema de un conjunto de controles es ese espacio generado, no los controles individualmente. ¿Qué control(es) son realmente imprescindibles para no perder dimensiones de ese espacio?

Pista 2. Construí una base XOR de los a_i por eliminación gaussiana bit a bit, de más significativo a menos significativo. Es clave que cada elemento de la base sea literalmente uno de los controles originales (no una combinación reducida): así, "perder un elemento de la base" es perder un control físico real y concreto. Cada control que *no* entra en la base es dependiente: se puede escribir como XOR de un subconjunto de los r elementos de la base, y ese subconjunto (un vector de r bits) es su "firma".

Pista 3. Pensá qué significa, para un elemento B_i de la base, que *ningún* otro control (existente o nuevo) lo referencie en su firma: si B_i se pierde, nada puede reconstruir esa dirección y alguna configuración se vuelve inalcanzable. Y si dos elementos de la base B_i, B_j son referenciados por **exactamente** el mismo conjunto de otros controles (misma firma), perder justo esos dos (B_i y B_j) también es fatal, aunque cada uno por separado esté "acompañado".

Pista 4. La condición exacta para sobrevivir a la pérdida de hasta 2 controles cualesquiera es: extendiendo cada firma con los controles nuevos que se fabriquen, todas las firmas deben tener peso ≥ 2 y ser **todas distintas** entre sí. Esto es, ni más ni menos, pedir un código lineal de distancia mínima ≥ 3 (tolera 2 borrados). Agrupá los índices de la base que ya tienen la misma firma actual: cada control nuevo que fabriqués le agrega un bit extra (libre) a la firma de *cada* elemento de la base, y dentro de un mismo grupo esos bits nuevos son lo único que puede distinguir a sus miembros.

Pista 5. Para un grupo de tamaño g cuyo peso actual (antes de agregar los k controles nuevos) es w (saturado a 0, 1 o ≥ 2 , porque más de 2 ya no aporta nada), la cantidad de etiquetas de k bits utilizables de forma que la firma final tenga peso ≥ 2 es $\text{cap}(0, k) = 2^k - k - 1$, $\text{cap}(1, k) = 2^k - 1$, $\text{cap}(\geq 2, k) = 2^k$. El grupo cabe con k controles nuevos si y solo si $g \leq \text{cap}(w, k)$. Probá $k = 0, 1, 2, \dots$ y quedáte con el primero que satisfaga a todos los grupos simultáneamente (nunca hace falta pasar de $k = 7$).

Solución

Primero, una reformulación: una configuración de luces es alcanzable si y solo si es el XOR de las máscaras de algún subconjunto de controles, de modo que el conjunto de configuraciones

alcanzables con un grupo de controles es exactamente el *espacio* que generan sobre $\mathbb{F}_2^b$. Fabricar controles nuevos y luego perder hasta dos (originales o nuevos) debe dejar intacto ese espacio: necesitamos que, para *cualquier* elección de hasta dos controles perdidos, los controles sobrevivientes sigan generando el mismo espacio de dimensión r (el rango de los a_i).

Paso 1: construir la base. Hacé eliminación gaussiana de los a_i , bit por bit de $b - 1$ a 0: para cada control, mientras tenga un bit fijado en una posición donde ya hay un pivote, se lo cancela con ese pivote; si llega a una posición sin pivote, ese control se convierte él mismo en el nuevo pivote de esa posición. Los controles que se anulan por completo son *dependientes*: su máscara es el XOR de un subconjunto de los r controles que sí quedaron en la base, y ese subconjunto (representado como un vector de r bits) es su **firma**. Es esencial llevar la cuenta de qué controles originales concretos componen cada pivote (no solo su máscara ya reducida), porque el argumento de correctitud necesita que cada elemento de la base sea un control físico real que puede perderse.

Paso 2: firma de cada elemento de la base. Simétricamente, para cada elemento de la base B_i definimos su propia firma como el conjunto de controles dependientes (existentes, y luego los nuevos que se fabriquen) que lo usan en su combinación. Si a B_i se lo pierde solo, hace falta al menos *otro* control que lo referencie para reconstruirlo; pero ese otro control podría ser justamente el segundo control perdido, así que en realidad hace falta que la firma de B_i tenga peso ≥ 2 ; si a B_i y B_j se los pierde juntos, hace falta que exista algún control que los distinga (que referencie a exactamente uno de los dos), es decir, que sus firmas sean distintas.

Paso 3: agrupar y calcular k . Agrupá los índices de la base por firma actual idéntica (antes de agregar controles nuevos). Cada control nuevo que se fabrique aporta un bit adicional, completamente libre, a la firma de cada elemento de la base (porque cualquier vector de $\mathbb{F}_2^r$ es expresable como XOR de un subconjunto de los r elementos de la base). Con k controles nuevos, cada grupo de tamaño g y peso actual w (saturado a $0/1/\geq 2$) necesita $g \leq \text{cap}(w, k)$, con

$$\text{cap}(0, k) = 2^k - k - 1, \quad \text{cap}(1, k) = 2^k - 1, \quad \text{cap}(\geq 2, k) = 2^k.$$

La respuesta es el menor k que satisface esta desigualdad en **todos** los grupos a la vez; alcanza con probar $k = 0, 1, 2, \dots$ (nunca hace falta pasar de $k = 7$, porque $2^7 - 7 - 1 = 120$ ya cubre cualquier $r \leq 60$). Si el rango de los a_i es 0 (todas las máscaras son nulas), la única configuración alcanzable es "todo apagado" y la respuesta es 0 sin más análisis.

Con el ejemplo 1 ($N = 5$, $b = 3$, máscaras 4, 2, 1, 6, 5): la base queda formada por los controles 4, 2, 1 (rango 3), y los controles 6 y 5 son dependientes con firmas $\{4, 2\}$ y $\{4, 1\}$ respectivamente (en coordenadas de la base). Agrupando los tres índices de la base por su firma actual (quién los referencia) se obtienen tres grupos de tamaño 1 con pesos 1, 1, 2; con $k = 1$ control nuevo alcanza para que los dos grupos de peso 1 alcancen el peso 2 (necesitan una etiqueta no nula de 1 bit, y solo hay una: 1, que alcanza porque cada grupo tiene tamaño 1), dando la respuesta 1 del enunciado.

Complejidad. Tiempo: $O(N \cdot b)$ Espacio: $O(N + b)$

El núcleo del algoritmo (construcción de la base y firmas de los controles dependientes) es corto:

```

1  for (Mask control : controls) {
2      Mask value = control;
3      Mask used = 0;
4      for (int bit = bits - 1; bit >= 0; --bit) {
5          if (((value >> bit) & 1ULL) == 0) continue;
6          if (pivot[bit] != 0) {
7              value ^= pivot[bit];
8              used ^= coordinates[bit];
9          } else {
10             pivot[bit] = value;                // mascara reducida

```

```

11         coordinates[bit] = used ^ (1ULL << rank); // en controles
12             originales
13         ++rank;
14         break;
15     }
16     if (value == 0) dependent.push_back(used); // firma del control
17     dependiente
18 }

```

Problema G — La oficina más feliz de Sergio

Dificultad estimada: Medio-difícil (aprox. 2000) • Temas: Greedy, búsqueda binaria sobre la respuesta, DSU, Fenwick tree

Pistas

Pista 1. Si lográs que todos los oficinistas reciban al menos T unidades de luz, ¿podés siempre lograr que reciban al menos $T' < T$ con el mismo presupuesto (o menos)? Esa monotonía debería sugerirte una técnica muy estándar para atacar “maximizar el mínimo”.

Pista 2. Bien, entonces el problema se reduce a: fijado un objetivo T , ¿existe una asignación de aperturas adicionales x_j (con $0 \leq x_j \leq h_j - a_j$ y $\sum x_j \leq k$) tal que todo oficinista reciba al menos T ? Pensá esto como un problema de *cubrir* cada rango $[l_i, r_i]$ con “déficit” $d_i = \max(0, T - \text{luz actual})$, gastando la menor capacidad posible.

Pista 3. El orden en que decidís satisfacer a los oficinistas importa. Probá procesarlos de izquierda a derecha según su extremo derecho r_i . Cuando te toca satisfacer a uno que todavía tiene déficit, ¿qué ventana conviene usar primero: la más a la izquierda o la más a la derecha dentro de $[l_i, r_i]$? Pensá en qué ventanas van a poder *reciclarse* para oficinistas futuros (que ya sabés van a tener $r_j \geq r_i$).

Pista 4. Toda luz que agregues en la ventana r_i (el extremo derecho del rango actual) queda automáticamente disponible para **cualquier** oficinista futuro que se solape con el actual, sin importar dónde empiece su rango – porque el solapamiento siempre incluye a r_i . En cambio, luz agregada cerca de l_i solo sirve a los futuros oficinistas cuyo rango también empiece muy a la izquierda. Por eso conviene llenar siempre la ventana libre **más a la derecha posible** dentro del rango.

Pista 5. Para hacer esto rápido: usá un DSU sobre las ventanas donde cada ventana saturada (sin margen restante) apunta a la ventana libre más cercana a su izquierda, así saltás en $O(\alpha(m))$ amortizado directo a la próxima ventana usable. Y usá un Fenwick tree para saber, en $O(\log m)$, cuánta luz ya se agregó dentro de cualquier rango $[l_i, r_i]$. Con eso, cada chequeo de factibilidad de un T candidato corre en tiempo casi lineal, y podés envolver todo en una búsqueda binaria sobre T .

Solución

Como en cualquier problema de “maximizar el mínimo”, la función

$\text{factible}(T) = \text{¿existe una asignación válida de } x_1, \dots, x_m \text{ tal que todo oficinista reciba } \geq T?$

es monótona: si es posible alcanzar T , también es posible alcanzar cualquier $T' < T$ (la misma asignación sirve). Esto permite hacer **búsqueda binaria sobre la respuesta**: el límite superior de la búsqueda puede fijarse, por ejemplo, tomando para cada oficinista la luz que recibiría si él solo (ignorando a los demás) usara toda la capacidad disponible en su rango, acotada por el

presupuesto k ; el mínimo de esos valores es una cota válida (nadie puede superar lo que podría lograr en solitario).

Lo único que falta es resolver **factible**(T) de forma eficiente. Para un T fijo, cada oficinista i tiene un déficit $d_i = \max(0, T - \text{luz_actual}_i)$, y necesitamos repartir el presupuesto k entre las ventanas (respetando $x_j \leq h_j - a_j$) para cubrir todos los déficits simultáneamente, usando el menor presupuesto total posible (si ese mínimo supera k , T no es factible).

La observación clave es de **orden**: si procesamos a los oficinistas ordenados por r_i creciente, cualquier oficinista futuro que se solape con el actual comparte necesariamente la ventana r_i (y las inmediatamente a su izquierda) – nunca solo las ventanas cercanas a l_i en exclusiva. Por lo tanto, cuando haya que agregar luz para cubrir el déficit del oficinista actual, conviene usar la ventana libre **más a la derecha posible** dentro de $[l_i, r_i]$: esa luz queda “reciclada” automáticamente para todo oficinista futuro que se solape, sea cual sea su extremo izquierdo.

El algoritmo completo, para un T candidato:

1. Ordenar los oficinistas por r_i creciente.
2. Mantener, por ventana, la capacidad restante $h_j - a_j$ (que se va consumiendo), y un **DSU** donde toda ventana saturada apunta a la ventana libre más cercana a su izquierda (así **find**(r) da directamente la ventana libre más a la derecha $\leq r$, saltando las saturadas).
3. Mantener un **Fenwick tree** con la luz ya agregada, para calcular en $O(\log m)$ la luz acumulada de cualquier rango.
4. Para cada oficinista, si su déficit es positivo: repetidamente tomar la ventana libre más a la derecha dentro de su rango (vía el DSU), consumir su capacidad restante, actualizar el Fenwick tree y el presupuesto usado, hasta cubrir el déficit. Si en algún momento no queda ninguna ventana libre dentro del rango, o se excede k , T no es factible.

```

1  auto feasible = [&](int64 target) {
2      FenwickTree added_light(Windows);
3      ...
4      for (const Interval& interval : intervals) {
5          int64 current_light = opening_prefix[interval.right] -
6              opening_prefix[interval.left - 1];
7          current_light += added_light.sum(interval.right) - added_light
8              .sum(interval.left - 1);
9          int64 needed = max(0LL, target - current_light);
10         while (needed > 0) {
11             int window = find(interval.right);
12             if (window < interval.left) return false;
13             int64 added = min(needed, remaining_capacity[window]);
14             ...
15         }
16     }
17     return true;
18 };

```

Complejidad. Tiempo: $O((n \log n + m) \log(\text{respuesta}))$ Espacio: $O(n + m)$

Problema H — El tren de juguete de Racsó

Dificultad estimada: Medio-difícil (aprox. 2200-2300) • Temas: Grafos (árbol de puentes + LCA)

Pistas

Pista 1. Olvídate por un momento de que hay muchas consultas. Piénsalo con una sola arista nueva: si agregas una arista entre dos vértices u y v de un grafo conexo, ¿qué conjunto de vértices termina 2-arista-conexo entre sí que antes no lo estaba? (2-arista-conexo quiere decir: siguen conectados aunque se borre cualquier arista única).

Pista 2. Agregar la arista (u, v) crea un único ciclo nuevo, y todos los vértices y aristas de ese ciclo dejan de tener puentes entre sí. Formalmente: dos vértices x, y se vuelven 2-arista-conexos si y solo si *todos* los puentes originales del grafo que separaban a x de y son exactamente los puentes que estaban en el camino entre u y v (es decir, si x y y quedan “atrapados” dentro del ciclo recién creado).

Pista 3. Esto sugiere trabajar con la estructura de puentes del grafo original, que es la misma para todas las consultas (no depende de las aristas hipotéticas). Contrae cada componente 2-arista-conexa del grafo original a un solo nodo: lo que queda es un árbol (el *árbol de puentes*), donde cada arista del árbol es un puente real y cada nodo pesa tantos vértices como la componente que representa. Agregar una arista nueva (u, v) equivale, en este árbol, a fusionar en una sola componente **todo el camino** entre $\text{comp}(u)$ y $\text{comp}(v)$.

Pista 4. Cada consulta (a, b, c, d) se traduce entonces en dos caminos P_1 (entre $\text{comp}(a)$ y $\text{comp}(b)$) y P_2 (entre $\text{comp}(c)$ y $\text{comp}(d)$) del árbol de puentes. Si P_1 y P_2 no comparten ningún nodo, ninguna de las dos aristas nuevas ayuda a la otra: la respuesta es 0. Si comparten un sub-camino I , todo par (x, y) con x en la parte de P_1 que **no** es de I y y en la parte de P_2 que tampoco es de I se vuelve 2-arista-conexo únicamente cuando **ambas** aristas están presentes (cada una cierra su propia mitad exclusiva; sin la otra, esa mitad se queda sin cerrar). Cuenta esos pares en función de los *pesos* (tamaños, en vértices) de las partes exclusivas de P_1 y de P_2 .

Pista 5. Para calcular caminos, pesos de caminos y la intersección de dos caminos en un árbol, alcanza con un puñado de LCAs por consulta (a lo sumo 3, con la técnica de las “medianas”: el punto medio entre tres vértices x, y, z es el de mayor profundidad entre $\text{lca}(x, y)$, $\text{lca}(x, z)$, $\text{lca}(y, z)$). Con *binary lifting* precalculado sobre el árbol de puentes, cada LCA (y por lo tanto cada consulta completa) sale en $O(\log n)$.

Solución

Primero se calculan los puentes del grafo original con una única búsqueda en profundidad (Tarjan, $O(n + m)$), y se contrae cada componente 2-arista-conexa a un solo nodo. Lo que queda es un árbol: el **árbol de puentes**. A cada nodo del árbol se le asigna un peso igual al número de vértices originales que contiene (el tamaño de su componente), y cada arista del árbol es exactamente

uno de los puentes originales. Cada estación del enunciado se mapea a un único nodo del árbol de puentes (el de su componente).

La observación clave es que agregar una arista nueva entre dos vértices u y v del grafo original tiene, sobre la 2-arista-conexidad, exactamente el mismo efecto que fusionar en el árbol de puentes todos los nodos del camino entre $\text{comp}(u)$ y $\text{comp}(v)$ en una sola componente: la nueva arista, junto con ese camino, forma un ciclo, y todo ciclo es 2-arista-conexo.

Entonces cada consulta (a, b, c, d) se traduce a dos caminos en el árbol de puentes: $P_1 = \text{camino}(\text{comp}(a), \text{comp}(b))$ y $P_2 = \text{camino}(\text{comp}(c), \text{comp}(d))$. Sea $W(P)$ la suma de los pesos de los nodos de un camino P (es decir, el número de vértices originales que cubre). Hay dos casos:

- Si P_1 y P_2 **no se intersectan** como conjuntos de nodos del árbol, ninguna de las dos aristas nuevas depende de la otra para cerrar su propio ciclo, así que ningún par nuevo requiere *ambas* aristas a la vez: la respuesta es 0.
- Si se intersectan, la intersección $I = P_1 \cap P_2$ es en sí misma un sub-camino contiguo del árbol (una propiedad general de los caminos en un árbol). La respuesta es

$$(W(P_1) - W(I)) \cdot (W(P_2) - W(I)).$$

Intuitivamente: al agregar ambas aristas, todo $P_1 \cup P_2$ termina en una sola componente 2-arista-conexa gigante. Un par (x, y) queda “recientemente” 2-arista-conexo (es decir, no lo estaba ya antes de agregar ninguna arista) solo si al menos uno de los dos no estaba en la parte común I . Pero además, si x está en la parte exclusiva de P_1 , **necesita** la arista de P_1 para conectarse (la arista de P_2 sola no alcanza a cerrar su ciclo), y simétricamente para y en la parte exclusiva de P_2 . Por eso el conteo real, el de pares que requieren AMBAS aristas, es el producto de los tamaños de las dos partes exclusivas.

Sólo falta calcular caminos e intersecciones de caminos rápido. Con *binary lifting* sobre el árbol de puentes se obtiene el LCA de dos nodos en $O(\log n)$, y con eso: la distancia entre dos nodos, el peso de un camino (usando pesos-prefijo acumulados desde la raíz, análogos a sumas prefijas sobre el camino a la raíz), y el *punto medio* de tres nodos x, y, z (el de mayor profundidad entre los tres LCAs por pares). Los extremos de la intersección de $P_1 = (s_1, e_1)$ y $P_2 = (s_2, e_2)$ son $\text{median}(s_1, e_1, s_2)$ y $\text{median}(s_1, e_1, e_2)$ (hay que verificar que ese segmento realmente está contenido en ambos caminos; si no lo está, la intersección es vacía y la respuesta es 0).

```

1  int64 answer(int first_start, int first_end, int second_start, int
    second_end) const {
2      int first_intersection = median(first_start, first_end,
        second_start);
3      int last_intersection = median(first_start, first_end, second_end)
        ;
4
5      if (distance(second_start, first_intersection)
6          + distance(first_intersection, second_end) != distance(
            second_start, second_end)) {
7          return 0;
8      }
9
10     int64 common_weight = path_weight(first_intersection,
        last_intersection);
11     int64 first_exclusive = path_weight(first_start, first_end) -
        common_weight;
12     int64 second_exclusive = path_weight(second_start, second_end) -
        common_weight;
13     return first_exclusive * second_exclusive;
14 }
```

Los puentes y las componentes 2-arista-conexas se calculan una única vez al principio ($O(n + m)$); después, cada una de las q consultas se responde en $O(\log n)$ con el árbol de puentes ya construido y el *binary lifting* ya precalculado.

Complejidad. Tiempo: $O(n + m + (n + q) \log n)$ Espacio: $O(n + m)$

Problema I — Tinta de respaldo

Dificultad estimada: Muy difícil (aprox. 2650) • Temas: Cadenas: autómata de sufijos, fusión small-to-large

Pistas

Pista 1. Fija una longitud L y mira solo las apariciones de un patrón de esa longitud, ordenadas por posición. Si cancelás una aparición interior, ¿qué otras apariciones podrían estar cubriendo las mismas casillas? Convencete de que alcanza con mirar a la aparición inmediatamente anterior y a la inmediatamente siguiente: ninguna aparición más lejana puede cubrir una casilla que sus vecinas no cubran ya.

Pista 2. Con eso en mente, para una aparición interior con distancia a a su vecina izquierda y b a su vecina derecha, calculá explícitamente cuántas casillas de su propio intervalo (de largo L) quedan cubiertas por cada vecina, y restá del total. En los extremos solo hay una vecina de cada lado, así que el análisis es todavía más simple.

Pista 3. Deberías llegar a que el daño de cancelar una aparición interior es $\max(0, \min(L, a) + \min(L, b) - L)$, y en los extremos es $\min(L, \text{distancia a la única vecina})$ (con daño L si la aparición es única). Fijáte que si $a \leq k$ o $b \leq k$, el daño de esa aparición interior ya es $\leq k$ automáticamente, sin importar el otro lado. Solo los pares de distancias consecutivas donde **ambas** superan k son peligrosos.

Pista 4. Para un par peligroso (a, b) con $a, b > k$, encontrá para qué valores de L el daño de esa aparición interior es $\leq k$. Vas a obtener una condición de la forma $L \geq a + b - k$. Combinando esto con la condición de los extremos (las dos distancias de borde deben ser $\leq k$), cada longitud $L > k$ de un patrón fijo es resistente si y solo si supera un cierto umbral que depende únicamente del *conjunto de posiciones* donde aparece ese patrón.

Pista 5. Muchas subcadenas distintas comparten exactamente el mismo conjunto de posiciones de aparición (mismo `endpos`): eso es justo lo que agrupa un autómata de sufijos, un estado por cada clase de equivalencia. Procesa los estados de mayor a menor `len`, mantené por estado un `std::set` de posiciones y un `std::multiset` con las sumas $a + b$ de los pares peligrosos, y al terminar con un estado fusioná su conjunto con el de su enlace de sufijo usando *small-to-large* (insertando siempre el conjunto más chico dentro del más grande).

Solución

Fijemos una longitud L y llamemos $p_1 < p_2 < \dots < p_m$ a las posiciones (de inicio) de las apariciones de un patrón de esa longitud. Cada aparición i pinta el intervalo $[p_i, p_i + L - 1]$. Como dos apariciones de la *misma* cadena no pueden “saltar por encima” de una tercera aparición intermedia sin que ésta cubra ya toda la zona de solapamiento, el único solapamiento relevante para la aparición i es con sus vecinas inmediatas $i - 1$ e $i + 1$.

Con $a = p_i - p_{i-1}$ y $b = p_{i+1} - p_i$, la vecina izquierda cubre los primeros $\max(0, L - a)$ celdas del intervalo de i , y la derecha las últimas $\max(0, L - b)$. El daño de cancelar i es lo que queda

sin cubrir por ninguna de las dos, que tras despejar da exactamente

$$\text{daño}(i) = \max(0, \min(L, a) + \min(L, b) - L).$$

En los extremos ($i = 1$ o $i = m$) solo hay una vecina, y el daño es $\min(L, \text{distancia})$ (o L si $m = 1$, la aparición es única).

Un patrón es resistente si y solo si el daño de *todas* sus apariciones es $\leq k$. Si $L \leq k$, el daño de cualquier aparición (a lo sumo L) ya cumple la cota, así que **toda longitud $\leq k$ es automáticamente resistente**. Para $L > k$ hacen falta al menos dos apariciones, y las dos distancias extremas deben ser $\leq k$ (si no, esa aparición de borde por sí sola ya rompe la cota). Para cada par de distancias consecutivas (a, b) con **ambas** mayores que k , un poco de álgebra muestra que el daño de esa aparición interior baja de k justo cuando $L \geq a + b - k$ (para cualquier otro par, con $a \leq k$ o $b \leq k$, esa aparición nunca es un problema). Entonces, fijado el conjunto de apariciones de un patrón, existe un umbral T (el máximo de $a + b - k$ sobre los pares peligrosos, o $k + 1$ si no hay ninguno) tal que el patrón de longitud L es resistente exactamente cuando $L \leq k$ o $L \geq T$ (más las condiciones de borde, que no dependen de L).

Lo clave es que, para *todas* las subcadenas que comparten el mismo conjunto de posiciones de aparición, este umbral T es el mismo – porque las distancias entre posiciones consecutivas no dependen de L , solo de dónde aparecen. Y ese agrupamiento (mismo conjunto de posiciones = mismo **endpos**) es exactamente lo que hace un **autómata de sufijos**: cada estado v representa todas las subcadenas de longitudes en $(\text{len}(\text{link}(v)), \text{len}(v)]$, todas con el mismo **endpos**.

Se procesan los estados en orden decreciente de len (que es un orden topológico del árbol de enlaces de sufijo, de hojas hacia la raíz). Cada estado “hoja” arranca con su propia posición; al terminar de contar un estado, su conjunto de posiciones se fusiona con el de su enlace de sufijo, así que cuando le toca el turno a un estado, su conjunto ya está completo (incluye todo lo aportado por sus descendientes). Para no pagar $O(n)$ por fusión, se usa *small-to-large*: siempre se inserta el conjunto más chico dentro del más grande, lo que da $O(n \log n)$ inserciones en total. Junto con un **std::multiset** que mantiene incrementalmente las sumas $a + b$ de los pares peligrosos (cada inserción de una posición nueva solo cambia, a lo sumo, 3 pares consecutivos), el umbral de un estado se calcula en $O(\log n)$:

```

1  int threshold(int n) const {
2      if (positions.size() < 2) return n + 1;
3      if (*next(positions.begin()) - *positions.begin() > k ||
4          *positions.rbegin() - *next(positions.rbegin()) > k) return n
5          + 1;
6      return bad.empty() ? k + 1 : max(k + 1, *bad.rbegin() - k);
7  }
```

Con el umbral T de un estado v (rango de longitudes $[low, high]$), su aporte a la respuesta es $\max(0, \min(high, k) - low + 1) + \max(0, high - \max(low, T) + 1)$: las longitudes automáticamente resistentes ($\leq k$) más las que superan el umbral. Sumando esto sobre todos los estados se obtiene la respuesta. Los casos $k = 0$ (respuesta 0, porque la aparición más a la izquierda de cualquier patrón siempre tiene al menos una celda cubierta únicamente por ella) y $k = n$ (toda subcadena es resistente, la respuesta es la cantidad total de subcadenas distintas) se resuelven aparte, sin construir los conjuntos de posiciones.

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

Problema J — La cruz de un solo corte

Dificultad estimada: Difícil (aprox. 2250) • Temas: Geometría computacional (barrido angular exacto)

Pistas

Pista 1. Para una dirección fija d , pensá cómo saber si la recta infinita que pasa por el origen con dirección d corta el interior de un cable AB . No hace falta trigonometría: alcanza con comparar de qué lado de esa recta caen A y B , y eso es exactamente el signo de un producto cruzado.

Pista 2. El segundo láser es perpendicular al primero. Si ya sabés testear "¿me corta la recta con dirección d ?" con un producto cruzado, ¿qué operación parecida sirve para testear la recta perpendicular *sin* rotar explícitamente el vector d ? (Pensalo en términos de un producto punto en vez de cruzado.)

Pista 3. No hay que probar todas las orientaciones posibles (son infinitas). El conjunto de cables cortados por cada recta sólo puede cambiar cuando alguna recta pasa exactamente por un extremo de algún cable, y esas orientaciones ya están explícitamente prohibidas por el enunciado. ¿Cuántas direcciones "críticas" hay en total, y qué pasa con la configuración de cortes en el resto de las orientaciones?

Pista 4. Girar la cruz completa 90° la deja exactamente igual como *par de rectas* (las dos simplemente intercambian cuál es "la primera" y cuál "la segunda"). Eso significa que alcanza con barrer un cuarto de vuelta para ver todas las configuraciones posibles.

Pista 5. Doblá cada extremo de cada cable, rotándolo de a 90° , hasta que caiga en el primer cuadrante; la paridad de rotaciones te dice si ese extremo es un evento para la primera o la segunda recta. Ordená todas esas direcciones con productos cruzados *exactos* (nada de ángulos ni **double**), agrupá los eventos que caen en exactamente la misma dirección, y mantené, mientras barrés, el peso cortado por exactamente una recta y la cantidad de cables cortados por las dos. Sólo evaluás la respuesta en los huecos abiertos entre grupos de eventos.

Solución

Fijemos una dirección $d = (d_x, d_y)$ y la recta infinita L_d que pasa por el origen con esa dirección. Para un punto P , el signo de $\text{cross}(d, P) = d_x P_y - d_y P_x$ indica de qué lado de L_d cae P (cero exactamente sobre la recta). Un cable con extremos A, B tiene su interior cortado por L_d si y sólo si $\text{cross}(d, A)$ y $\text{cross}(d, B)$ tienen signos estrictamente opuestos.

La segunda recta es $L_{d^\perp}$ con $d^\perp = (-d_y, d_x)$. Como $\text{cross}(d^\perp, P) = -\text{dot}(d, P)$, testear $L_{d^\perp}$ es exactamente comparar los signos de $\text{dot}(d, A)$ y $\text{dot}(d, B)$: nunca hace falta rotar d de verdad. Cada cable tiene entonces un *estado* de 2 bits: bit 1 si lo corta L_d , bit 2 si lo corta $L_{d^\perp}$. Estado 3 (ambos bits) invalida toda la orientación; estado 1 o 2 aporta el peso del cable; estado 0 no aporta nada.

Simetría de 90° . Como par de rectas (sin orientación), $\{\theta, \theta + 90^\circ\}$ y $\{\theta + 90^\circ, \theta + 180^\circ \equiv \theta\}$ son literalmente el mismo par: rotar la cruz 90° la deja igual, solo intercambiando qué recta es "la primera". Por lo tanto basta barrer $\theta \in [0^\circ, 90^\circ)$.

Direcciones críticas. Para un extremo P , el signo de $\text{cross}(d(\theta), P)$ es una función continua de θ que solo se anula cuando $\theta \equiv \arg(P) \pmod{180^\circ}$; fuera de esos puntos no puede cambiar de signo. Lo mismo vale para dot con $\theta \equiv \arg(P) + 90^\circ \pmod{180^\circ}$. Reducidos módulo 90° , ambos valores caen en el mismo residuo $\arg(P) \pmod{90^\circ}$: cada extremo aporta un único evento en el cuarto de vuelta, etiquetado con un bit (1 o 2) según la paridad de rotaciones de 90° necesarias para plegarlo al primer cuadrante. Entre dos eventos consecutivos el estado de todos los cables es constante, así que basta evaluar un candidato por cada hueco entre eventos.

El algoritmo entonces:

1. Plegar cada uno de los $2n$ extremos al primer cuadrante, generando un evento (**dirección**, **cable**, **bit**).
2. Ordenar los eventos por dirección creciente con un comparador *exacto*: para vectores del primer cuadrante, el signo de $\text{cross}(u, v)$ ya determina sin ambigüedad el orden (la diferencia angular nunca supera 90°).
3. Barrer los eventos agrupándolos en bloques de dirección *exactamente* igual (no evento por evento): dentro de un bloque no existe ninguna orientación real "intermedia", así que hay que aplicar todo el bloque antes de evaluar la respuesta.
4. Mantener el peso cortado por exactamente una recta y la cantidad de cables en estado 3; después de cada bloque, si esa cantidad es 0, la celda es válida y se actualiza el máximo.

```

1  i128 cross(Point a, Point b) { return (i128)a.x*b.y - (i128)a.y*b.x; }
2  i128 dot(Point a, Point b)   { return (i128)a.x*b.x + (i128)a.y*b.y; }
3  bool opposite(i128 a, i128 b) { return (a < 0) != (b < 0); }
4  // ...
5  while (r < events.size() && cross(events[l].direction, events[r].
    direction) == 0) {
6      int i = events[r].cable;
7      if (state[i] == 3) --invalid; else if (state[i]) score -= weight[i];
8      state[i] ^= events[r].bit;
9      if (state[i] == 3) ++invalid; else if (state[i]) score += weight[i];
10     ++r;
11 }
12 if (!invalid) answer = max(answer, score);

```

Toda la aritmética (**cross**, **dot**, orden, empates) se hace en `__int128_t`, nunca en `double`: los productos de coordenadas de hasta 10^9 alcanzan magnitud 10^{18} , que ya no entra en la mantisa de 53 bits de un `double`, y este problema está diseñado justamente con casos de direcciones casi (o exactamente) coincidentes, donde ese error de redondeo puede invertir un signo. Con enteros de 128 bits el signo calculado es siempre el verdadero.

Complejidad. Tiempo: $O(n \log n)$ por caso (dominado por el **sort** de eventos) Espacio: $O(n)$

Problema K — Señales que no esperan

Dificultad estimada: Muy difícil (aprox. 2500) • Temas: Programación dinámica sobre diferencias marginales (slope trick)

Pistas

Pista 1. Si la capacidad c_i fuera siempre infinita (es decir, si la batería nunca perdiera energía por desbordamiento), este problema es exactamente un clásico: en cada día se recibe una cantidad de energía y hay que decidir si "comprar" el envío de ese día pagando a_i , maximizando la cantidad de compras. Pensá primero esa versión simplificada.

Pista 2. Para esa versión simplificada, fijáte qué pasa si definís $dp[k]$ = la máxima energía posible al final del día i habiendo enviado *exactamente* k mensajes hasta ahora (entre todas las formas de elegir cuáles k enviar). Como enviar más mensajes siempre cuesta energía extra, $dp[k]$ es no creciente en k . Ésa es la función que hay que actualizar día a día.

Pista 3. La función dp no es arbitraria: es **cóncava**. Es decir, la "pérdida marginal" de enviar el k -ésimo mensaje (la diferencia $dp[k-1] - dp[k]$) nunca disminuye a medida que k crece – si elegís de forma óptima, siempre conviene haber comprado antes los envíos más baratos. Esto significa que dp queda totalmente determinada por el *multiconjunto ordenado* de esas diferencias, sin necesidad de guardar $dp[k]$ para cada k por separado.

Pista 4. Con esa representación, procesar un día es "insertar a_i como una nueva diferencia candidata, y si el costo total ya comprado supera la energía base disponible, descartar la diferencia *más cara*" (el envío menos conveniente de todos los comprados hasta ahora). Esto es, literalmente, el mismo truco que usa CF 1526C2 (<https://codeforces.com/problemset/problem/1526/C2>).

Pista 5. Falta la capacidad c_i . Cada día, antes de decidir si enviar, la energía se recorta a $\min(c_i, E + r_i)$. Ese recorte no afecta por igual a todos los valores de $dp[k]$: afecta más a los k chicos (que representan mayor energía retenida). Se puede simular exactamente descontando el "desbordamiento" ($\max(0, dp[0] + r_i - c_i)$) empezando por las diferencias **más chicas** del multiconjunto (no las más caras), hasta agotar ese desbordamiento.

Solución

Sea $dp_i[k]$ la máxima energía posible al terminar el día i habiendo enviado exactamente k mensajes entre los primeros i días (sobre todas las formas de elegir cuáles enviar), y $-\infty$ si no es alcanzable. Como argumentan las pistas, para i fijo esta función es cóncava y no creciente en k , así que en vez de guardar $dp_i[0], \dots, dp_i[i]$ explícitamente basta guardar:

- **base**: el valor $dp_i[0]$ (la energía máxima posible sin haber enviado nada), y
- un `multiset<long long>` con las diferencias $dp_i[k-1] - dp_i[k] > 0$ ya "compradas" (más un contador aparte de cuántas diferencias valen exactamente 0, para no re-escanearlas), de forma que $dp_i[k] = \text{base} - (\text{suma de las } k \text{ diferencias más chicas})$.

Cada día se procesa en dos pasos, en este orden:

1. Recarga y recorte por capacidad. Primero `base` pasa a ser $\min(c_i, \text{base} + r_i)$ (la recarga sube todos los valores por igual, y el recorte por capacidad afecta únicamente a los k para los que $dp[k]$ superó c_i , que son justamente los k chicos). Para simular ese recorte sobre las diferencias, se calcula $\text{overflow} = \max(0, \text{base}_{\text{anterior}} + r_i - c_i)$ y se descuenta esa cantidad empezando por las diferencias **más chicas** del multiset (sacando el mínimo, restando lo que se pueda, reinsertándolo si sobra algo, y si no pasa a contar como una diferencia cero), hasta agotar `overflow`.

2. Intento de envío. Se inserta a_i como una nueva diferencia en el multiset (el mensaje de hoy "se compra" tentativamente) y se suma al total acumulado de diferencias. Si ese total supera `base`, significa que ya no alcanza la energía para sostener todos los envíos elegidos, y se descarta la diferencia **más grande** (el envío menos rentable de todos los comprados hasta el momento, que puede ser el de hoy mismo o uno de un día anterior).

La respuesta final es la cantidad de diferencias que sobreviven al final (las nulas incluidas), es decir el tamaño final del multiset más el contador de ceros.

Un fragmento del núcleo del algoritmo (versión con `multiset`):

```
1  i64 overflow = max(0LL, base + recharge - capacity);
2  base = min(capacity, base + recharge);
3  while (overflow && !positive.empty()) {
4      auto it = positive.begin();
5      i64 value = *it;
6      positive.erase(it);
7      i64 removed = min(overflow, value);
8      total -= removed;
9      overflow -= removed;
10     value -= removed;
11     if (value) positive.insert(value);
12     else ++zeros;
13 }
14 positive.insert(cost);
15 total += cost;
16 if (total > base) {
17     auto it = prev(positive.end());
18     total -= *it;
19     positive.erase(it);
20 }
```

Complejidad. Tiempo: $O(n \log n)$ por caso de prueba (amortizado) Espacio: $O(n)$

Cada mensaje entra y sale del multiset $O(1)$ veces en amortizado, y cada operación de inserción/borrado cuesta $O(\log n)$.