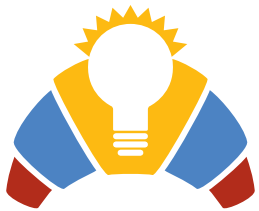
Árboles Recargados

(Consultas en subárboles y en caminos)

Sebastián Mestre

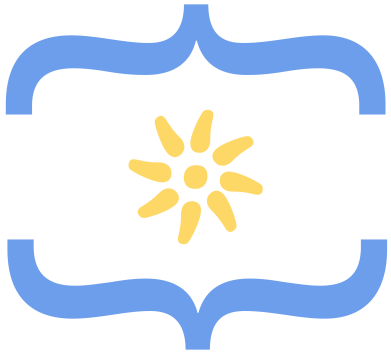
Universidad Nacional de Rosario - FCEIA

Training Camp 2026



Gracias Sponsors!

Organiza



Diamond Sponsor

Outline

Outline

Pregunta

- Sea N un número natural: ¿cuántos divisores tiene, en promedio, un número entre 1 y N?

Conteo doble

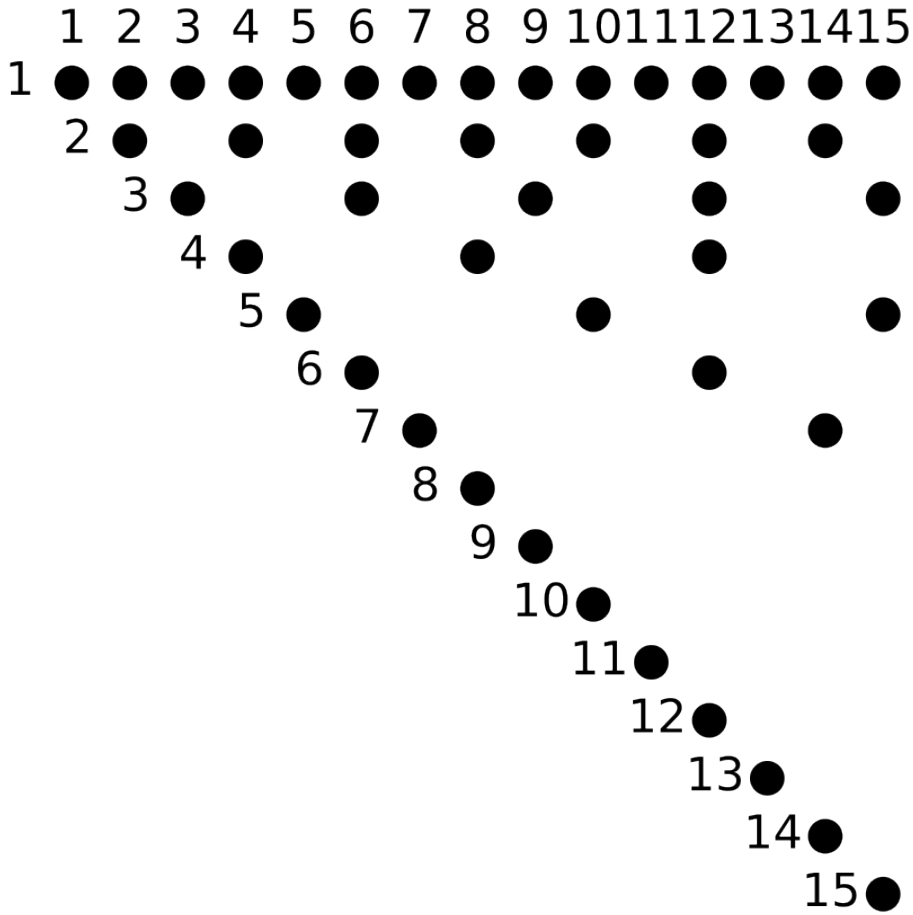
$$D(n) = \{\text{divisores de } n\}$$

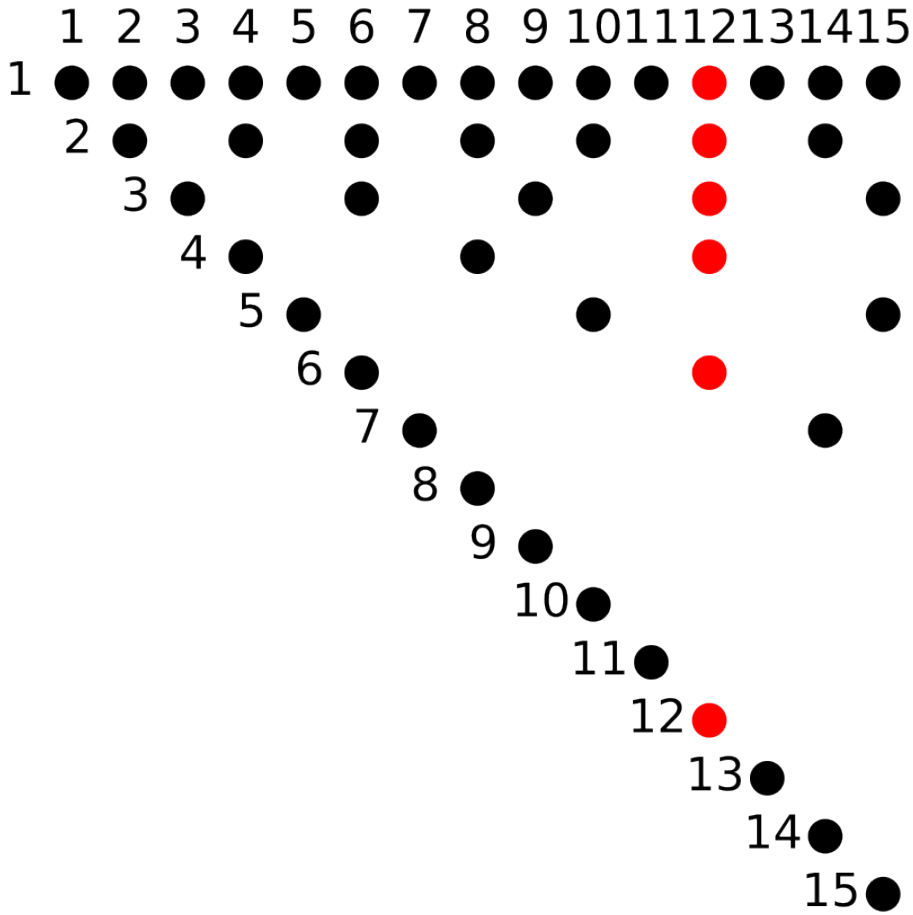
$$\frac{1}{N}\sum_{n=1}^{N} |D(n)|$$

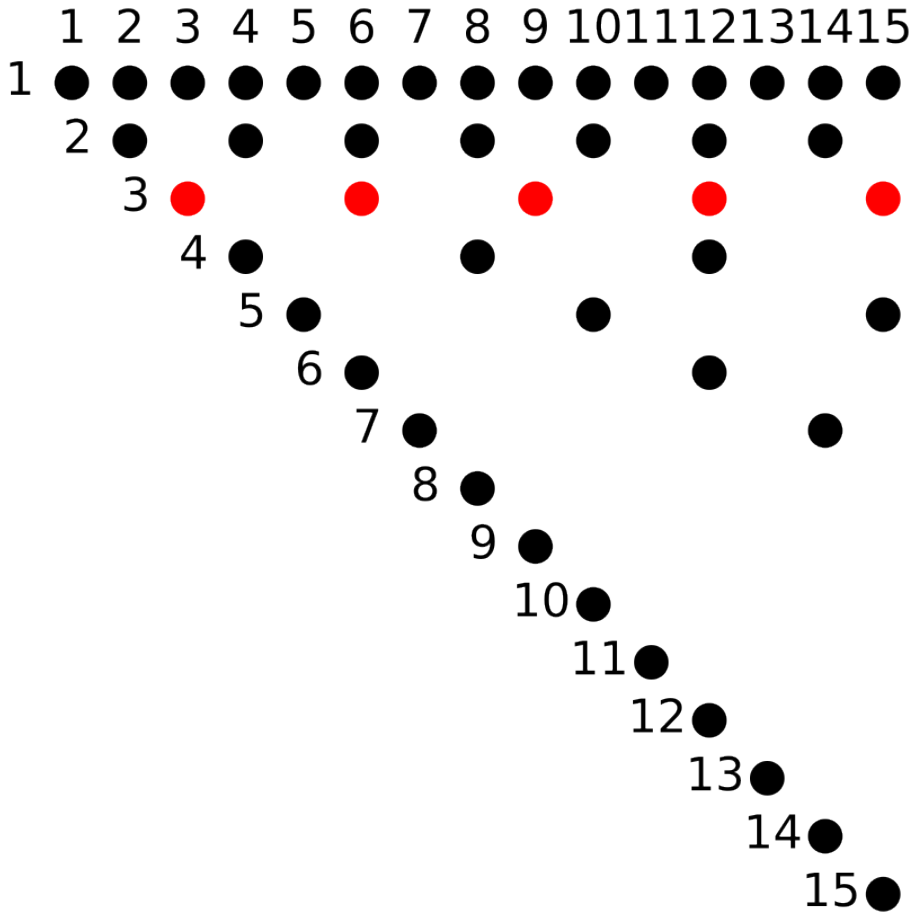
$$\left\{(m,n) \in \mathbb{N}^2 \;\middle|\; m \mid n \text{ y } n \leq N\right\}$$

$m \mid n$

- "m divide a n"
- "m es divisor de n"
- "n es múltiplo de m"
- $\exists k : n = km$


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Conteo doble

$$M_N(m) = \{\text{múltiplos de } m \text{ menores o iguales a } N\}$$

$$\frac{1}{N}\sum_{m=1}^{N} |M_N(m)| = \frac{1}{N}\sum_{m=1}^{N} \left\lfloor \frac{N}{m} \right\rfloor \leq \frac{1}{N}\sum_{m=1}^{N} \frac{N}{m} = \sum_{m=1}^{N} \frac{1}{m} \in O(\log N)$$

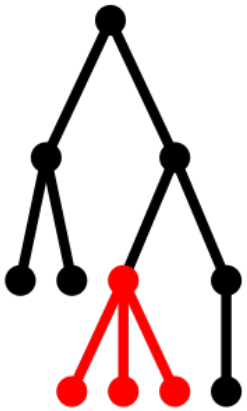
Árboles?

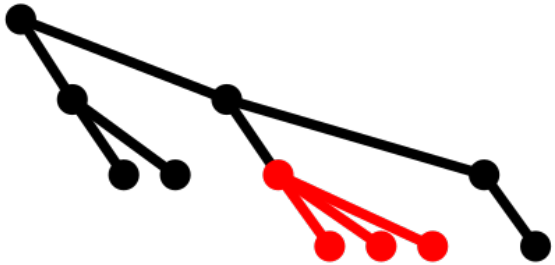
Outline

Consultas de subárboles

Supongamos un árbol con raíz donde cada nodo tiene un valor numérico. Queremos soportar dos operaciones:

- `suma(u)`: devuelve la suma de todos los valores del subárbol enraizado en u.
- `cambiar(u, x)`: reemplaza el valor del nodo u por x.





Consultas de subárboles: idea

Podemos construir el **preorden** del árbol.

- Propiedad clave: a cada subárbol le corresponde un **subarray** en el preorden.
- Reducimos el problema a consultas en subarrays.

Consultas de subárboles: código

```
int lp[maxn], rp[maxn];
vector<int> pre;
void dfs(int u, int p) {
    lp[u] = sz(pre);
    pre.push_back(val[u]);
    for (int v : g[u]) if (v != p) {
        dfs(v, u);
    }
    rp[u] = sz(pre);
}

int suma(int u) {
    return suma_subarray(lp[u], rp[u]);
}
```

Consultas de subárboles: observaciones

- La misma solución sirve para máximos, mínimos, xor, gcd, etc.
- Sin embargo, **no funciona** para cualquier consulta que se nos ocurra.
- Por ejemplo: cantidad de números distintos, `mex` de los números, etc.

Outline

Aristas livianas y pesadas

Ehad and the Big Finale (Codeforces 1174F)

- Problema interactivo.
- Hay un árbol con raíz que tiene un nodo especial secreto $\star$.
- $N \leq 10^5$.
- Te dan el árbol pero no te dicen cuál es el nodo secreto.
- Objetivo: descubrir el nodo secreto.

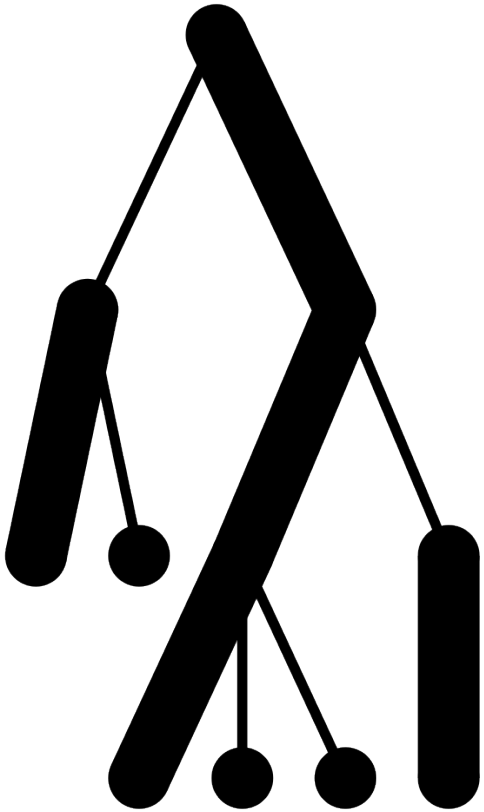
Ehad and the Big Finale (Codeforces 1174F)

- Para esto se pueden hacer dos tipos de consultas:
  - $d(u, \star)$: distancia de u al nodo especial.
  - $s(u)$: siguiente nodo en el camino de u a $\star$.
- Solo se pueden hacer 36 consultas.
- Solo se puede hacer la consulta s si u es ancestro de $\star$.
- (Sin esta restricción se puede resolver haciendo consultas s en los centroides.)

Aristas livianas y pesadas

Consideramos un árbol de N nodos.

- Por cada nodo con hijos, designamos como **hijo pesado** a aquel cuyo subárbol tiene la máxima cantidad de nodos (si hay más de uno, elegimos uno arbitrariamente).
- A las aristas que conectan nodos con su hijo pesado las llamamos **pesadas**.
- Si una arista no es pesada, la llamamos **liviana**.
- Terminología: a los caminos maximales formados enteramente por aristas pesadas los llamamos **caminos pesados**.

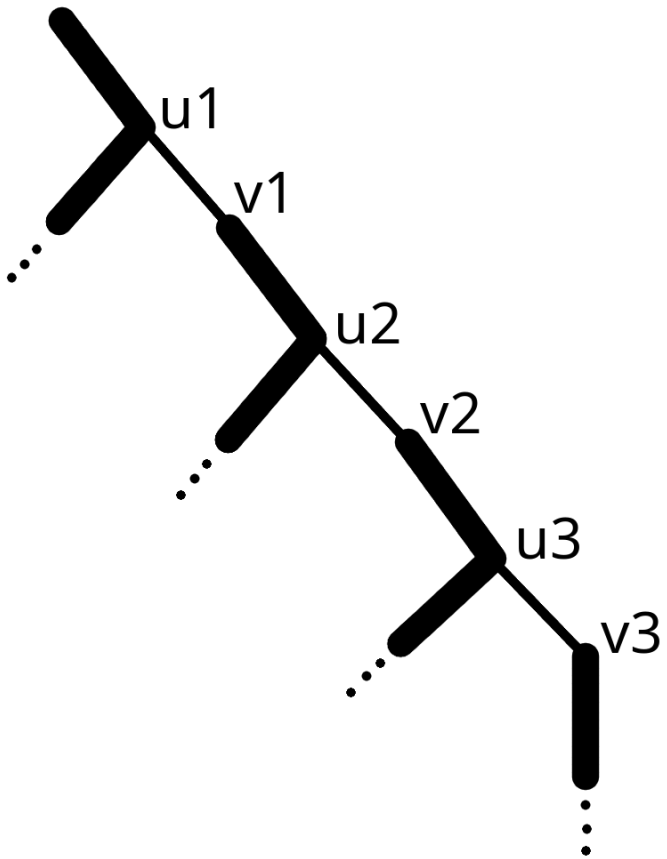


Aristas livianas y pesadas: observaciones

Observación clave

- El subarbol que cuelga de una arista liviana tiene tamaño menor a la mitad del subarbol del nodo padre.
- En un camino nodo–raíz puede haber a lo sumo $\log_2 N$ aristas livianas.

Aristas livianas y pesadas: observaciones



$$N \geq |u_1| > 2|v_1| > 2|u_2| > 4|v_2| > \cdots > 2^k|v_k| \qquad |v_k| \geq 1$$

$$\Rightarrow k \leq \log_2 N$$

Heavy-light decompose: código

```
int p[maxn]; // padre (-1 si no tiene)
int s[maxn]; // tamano del subarbol
int h[maxn]; // hijo pesado (-1 si no tiene)

void decompose() {
    forn(u, n) {
        h[u] = -1;
        for (int v : g[u]) if (v != p[u]) {
            if (h[v] == -1 || s[h[v]] < s[v]) {
                h[u] = v;
            }
        }
    }
}
```

Ehad and the Big Finale – solución

Nos paramos en la raíz y vamos bajando hasta llegar a $\star$, sin salirnos del camino de los ancestros de $\star$ (para poder usar la consulta $\mathtt{s}$).

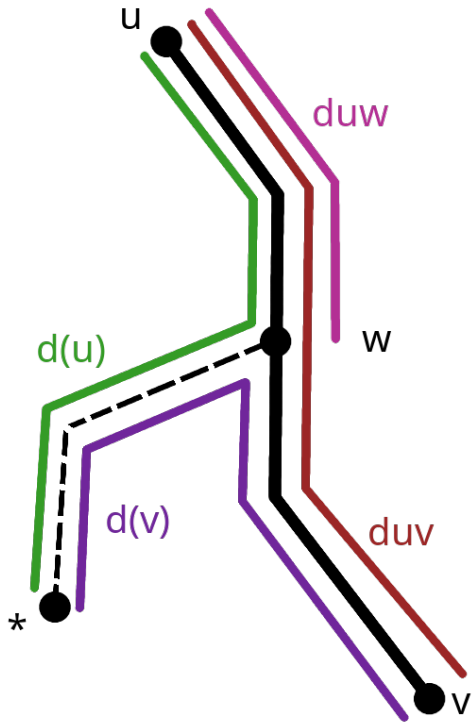
La idea del algoritmo es ir caminando de la raíz hasta $\star$, pero saltando de un camino pesado al siguiente en cada paso: si estamos en un camino pesado, saltar rápido a la posición en la que el camino hacia $\star$ se sale del camino pesado.

Al estar en un camino pesado hacemos dos consultas $\mathtt{d}$(-):

- una en el nodo de arriba de todo (u),
- otra en el de abajo de todo (v).

Al saber las distancias u-$\star$ y v-$\star$, y la distancia u-v, podemos calcular el punto en el que el camino raíz-$\star$ se sale del camino pesado u-v.

Localizar la salida con d(u) y d(v)



$$\text{duw} = \frac{duv + d(u) - d(v)}{2}$$

Cruzar la arista liviana

Una vez que sabemos el punto en el que se sale, sabemos que ahí hay que cruzar una arista liviana, pero no sabemos cuál.

Para resolver eso hacemos una consulta $\mathtt{s}(\text{-})$.

Como nos mantenemos siempre en un camino nodo-raíz, la cantidad de aristas livianas y caminos pesados que tocamos es a lo sumo $\log_2(N)$.

Ahí tenemos una solución en $3\log(N)$ consultas. Bajarlo a $2\log(N)+1$ es bastante directo, y queda como ejercicio.

Ehad and the Big Finale – código

```
int special() {
    int u = 0;
    int du = d(u);
    while (true) {
        if (du == 0) return u;
        int duv = 0;
        int v = u;
        while (h[v] != -1) v = h[v], duv += 1;
        int dv = d(v);
        int steps = (du - dv + duv) / 2;
        forn(_, steps) u = h[u], du -= 1;
        if (du == 0) return u;
        u = s(u);
        du -= 1;
    }
}
```

Distinct Colors (CSES 1139)

- Hay un árbol enraizado en el nodo 1, con numeritos c_i en cada nodo i.
- $N \leq 2 \cdot 10^5$.
- Por cada subárbol queremos saber la cantidad de valores distintos que contiene.

Distinct Colors

- Solucion Naive: Por cada nodo, metemos todos los valores del subárbol en un `set` y devolvemos `size()`.
- Costo: $O(N^2)$.
- Para construir el conjunto de colores de un subarbol, podemos reutilizar el conjunto de colores de alguno de sus hijos. En particular, el del hijo pesado, ya que es el más grande.
- Esto seguro es más rápido, pero ¿cambia el costo asintótico? Resulta que sí, veamos por qué.

Detalle del algoritmo

En vez de pensarlo como una optimización, plantemos el algoritmo desde cero con esta idea de ir pasando el conjunto de colores de un hijo al padre. Bajo esta perspectiva, contruimos un conjunto a medida que recorremos un camino pesado de abajo hacia arriba, agregando los colores de los descendientes faltantes de cada nodo. (los que no estan en el subarbol pesado)

```
for p : caminos_pesados
  s = {}
  for u : ascendente(p)
    for v : hijo liviano de u
      for w : subarbol(v)
        agregar color(w) a s
    respuesta[u] = tamano de s
```

Distinct Colors: doble conteo

Más allá del orden de iteración, nuestro algoritmo recorre los mismos valores que este otro:

```
for u = 1..n
  for v : g[u]
    if u-v es liviana
      for w : subarbol(v)
        print(w, v, u)
```

O sea, itera por los pares $\{(u, w) \mid u \in V,\ w \in D(u)\}$, donde

$$D(u) = \{\text{descendientes de } u \text{ que no están en el subárbol pesado}\}$$

Pero este es el mismo conjunto de pares que $\{(u, w) \mid w \in V,\ u \in A(w)\}$, donde

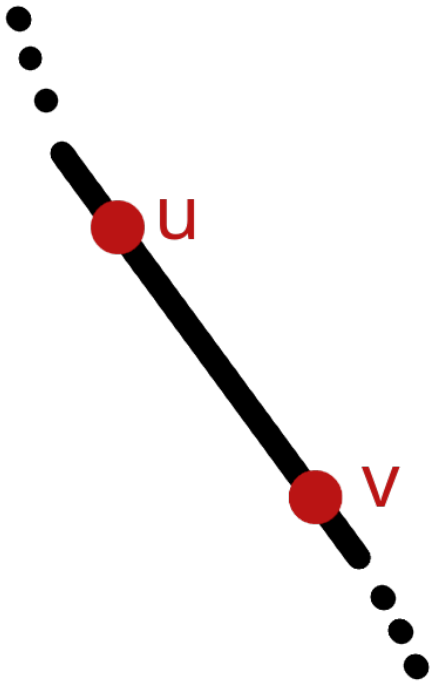
$$A(w) = \{\text{ancestros de } w \text{ con arista liviana hacia } w\}$$

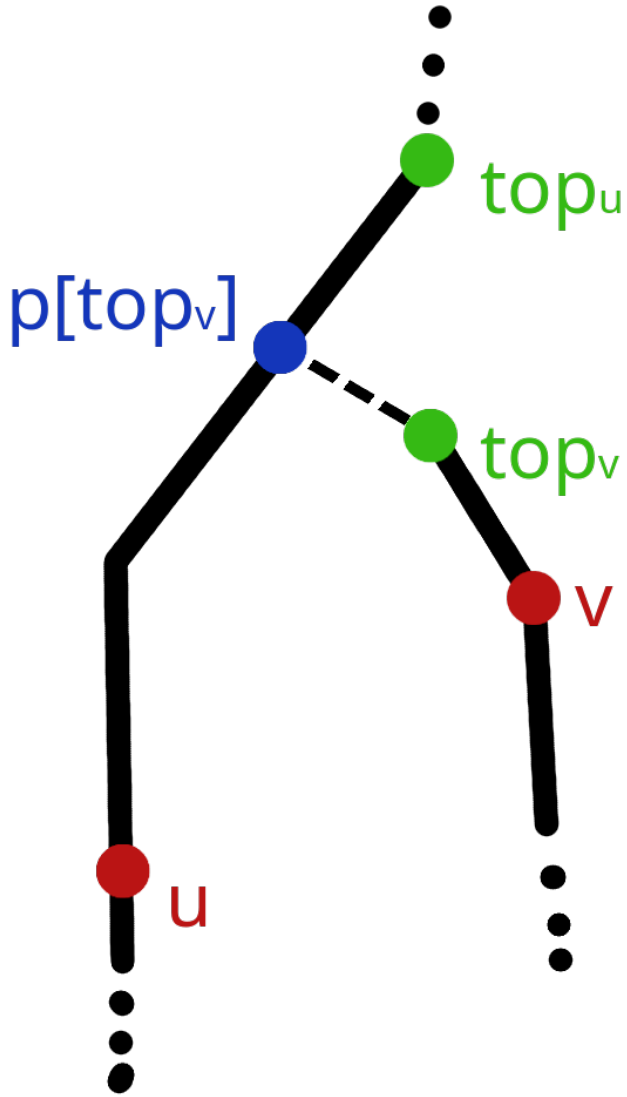
El cual es claramente $O(N \log N)$, porque por cada w hay solo $\log_2(N)$ aristas livianas en su camino a la raíz.

Distinct Colors: código

```
vector<int> distintos() {
    vector<int> ans(n);
    unordered_set<int> s;
    forn(f, n) if (h[f] == -1) {
        int u = f;
        while (true) {
            s.insert(u);
            for (int v : g[u]) if (v != p[u] && v != h[u])
                meter_todo(v, u, s);
            ans[u] = s.size();
            int w = p[u];
            // corto al llegar al tope del camino pesado
            if (w == -1 || h[w] != u) break;
            u = w;
        }
        s.clear();
    }
    return ans;
}
```

Mandatory LCA slide (again)





Mandatory LCA slide (again)

```
int lca(int u, int v) {
    int tu = htop[u], tv = htop[v];
    if (tu == tv) {
        if (dep[u] > dep[v]) swap(u, v);
        return u;
    } else {
        if (dep[tu] < dep[tv]) swap(u, v), swap(tu, tv);
        return lca(p[tu], v)
    }
}
```

Outline

Consultas de caminos

Consultas de caminos usando la descomposición

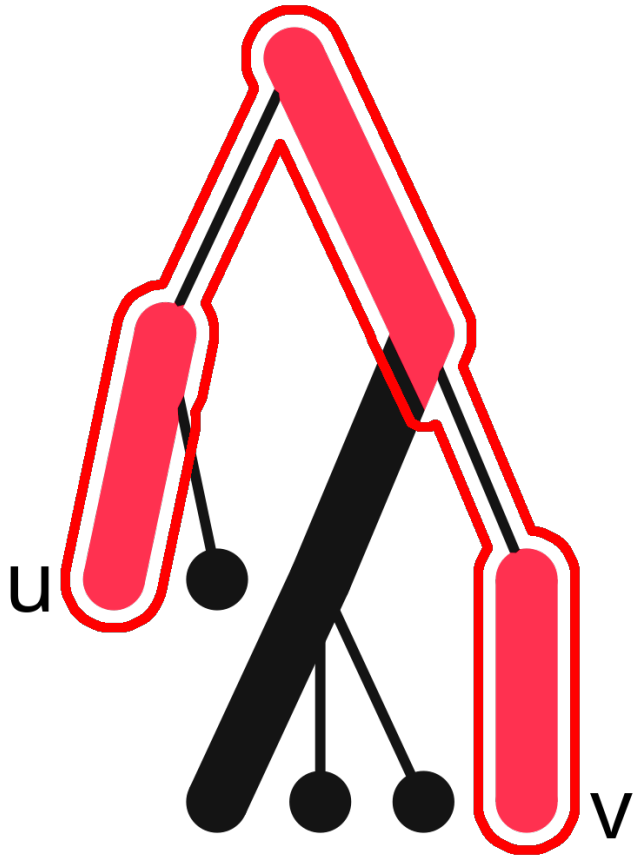
Tenemos un árbol con números en los nodos. Queremos soportar dos operaciones:

- `suma(u, v)`: devuelve la suma de todos los números del camino u–v
- `cambiar(u, x)`: reemplaza el número del nodo u con x
- $N \leq 2 \cdot 10^5$
- $Q \leq 2 \cdot 10^5$

Consultas de caminos: descomposición

La idea es nuevamente aprovechar que cualquier camino se puede descomponer en varios pedazos, cada uno parte de un camino pesado distinto, y que esta descomposición tiene pocas partes ($O(\log N)$).

Consultas de caminos: descomposición



Consultas de caminos: consultas por pedazo

Para resolver la suma dentro de cada pedazo necesitamos poder hacer consultas de rango en el array de elementos del camino pesado.

La forma clásica es construir un **segment tree** sobre cada camino pesado.

Consultas de caminos: idea del algoritmo

- Si el camino está completamente contenido en un camino pesado, hacemos la consulta directamente.
- Si no, el camino u–v necesariamente llega hasta la cima de un camino pesado y se sale hacia arriba.
- Detectamos cuál de las dos puntas es la que se sale por la cima, y hacemos la consulta en ese camino pesado.
- De esta forma resolvemos la punta del camino u–v y achicamos el camino que falta por resolver.

Consultas de caminos: consulta

```
int query(int u, int v) {
  int tu = htop[u], tv = htop[v];
  if (tu == tv) {
    if (dep[u] > dep[v]) swap(u, v);
    return tree[tu].query(dep[u]-dep[tu], dep[v]-dep[tu]+1);
  } else {
    if (dep[tu] < dep[tv]) swap(u, v), swap(tu, tv);
    return tree[tu].query(0, dep[u]-dep[tu]+1) + query(p[tu], v);
  }
}
```

Consultas no conmutativas

El algoritmo de consultas en caminos funciona también para operaciones **no conmutativas** (por ejemplo, producto de matrices).

- Guardamos pares con la suma de izquierda a derecha y la suma de derecha a izquierda.
- En cada paso del algoritmo elegimos la dirección correcta según cuál punta del camino estamos resolviendo.

Consultas no conmutativas: estructuras

```
struct Mono {
    static Mono neut() { /* ... */ }
};
Mono operator + (Mono x, Mono y);

struct Conm {
    Mono ltr, rtl;
    static Conm make(Mono x) { return {x, x}; }
    static Conm neut() { return {Mono::neut(), Mono::neut()}; }
};
Conm operator + (Conm x, Conm y) {
    return {x.ltr + y.ltr, y.rtl + x.rtl};
}
```

Consultas no conmutativas: query

```
Mono query(int u, int v) {
  int tu = htop[u], tv = htop[v];
  if (tu == tv) {
    if (dep[u] < dep[v]) {
      return tree[tu].query(dep[u]-dep[tu],dep[v]-dep[tu]+1).ltr;
    } else {
      return tree[tv].query(dep[v]-dep[tv],dep[u]-dep[tv]+1).rtl;
    }
  } else {
    if (dep[tu] > dep[tv]) {
      return tree[tu].query(0, dep[u]-dep[tu]+1).rtl
           + query(p[tu], v);
    } else {
      return query(u, p[tv])
           + tree[tv].query(0, dep[v]-dep[tv]+1).ltr;
    }
  }
}
```

Outline

Euler Tour

Consultas de xor en caminos (sin updates)

- Te dan un arbol con numeros en las aristas
- Querés soportar consultas de xor en caminos
- $N \leq 2 \cdot 10^5$
- $Q \leq 2 \cdot 10^5$

Esto se puede resolver descomponiendo en caminos pesados.

Pero también hay una solución muchísimo más simple.

Consultas de xor en caminos: idea

La idea es construir el **Euler tour**: un array de los valores de aristas en el orden en que te los encontrás si hacés un DFS del árbol.

- Se inserta el valor tanto al **ir** como al **volver** de cada arista.
- Para responder una consulta de xor en un camino u–v, consultamos en ese array. El rango corresponde a las posiciones de u y v en el recorrido.

Consultas de xor en caminos: por qué funciona

Si mirás el intervalo de valores entre una aparición de u y una de v, cualquier arista por la que "fuimos y volvimos" entre medio en el DFS se cancela, porque el xor es su propia inversa.

Y coincidentemente esas aristas son las que **no** están en el camino u–v. O sea, las que no se cancelan son exactamente las del camino u–v.

Por lo tanto, la respuesta es simplemente el xor de los valores de las aristas en el camino u–v.

Consultas de xor en caminos: código

```
struct edge {
    int u, v, x;
    int opp(int w) { return u + v - w; }
}
edge edges[maxn];
vector<int> g[maxn];
vector<int> euler_tour;
int position[maxn];
void dfs(int u, int p) {
    for (int i : g[u]) {
        edge e = edges[i];
        int v = e.opp(u);
        if (v == p) continue;

        position[u] = sz(euler_tour);
        euler_tour.push_back(e.x);
        position[v] = sz(euler_tour);
        dfs(v, u);
        euler_tour.push_back(e.x);
```

Consultas de caminos usando el truco de Mo

Esta idea es muy buena pero depende de propiedades particulares de la operación `xor`:

- es asociativa
- tiene elemento neutro
- tiene inverso para cada elemento
- es conmutativa
- **cada elemento es su propio inverso**

Consultas de caminos usando el truco de Mo: generalización

En realidad se puede generalizar a cualquier operación si cumple las primeras 4.

Nota

conmutativa es opcional pero complica la implementación

Consultas de caminos usando el truco de Mo: idea

Básicamente la idea es hacer un `for` desde la posición de u hasta la de v, e ir acumulando los valores.

A medida que hacemos esto, hay que chequear si estamos yendo o volviendo por una arista. En caso de estar yendo, acumulamos el valor de la arista. En caso de estar volviendo, acumulamos el inverso del valor de la arista.

De esta forma, tenemos un algoritmo que funciona para una operación donde no necesariamente cada elemento es su propio inverso.

Consultas de caminos usando el truco de Mo: implementación

Para chequear esto, simplemente guardamos los índices de las aristas en el Euler tour en vez de los valores.

Aparte, podemos mantener un array de booleanos que nos diga si ya sumamos el valor de la arista o no.

Consultas de caminos usando el truco de Mo: código

```
bool tengo[maxn];
int query(int u, int v) {
    int ans = 0;
    int l = position[u], r = position[v];
    if (l > r) swap(l, r);
    for (int i = l; i < r; i++) {
        int j = euler_tour[i];
        if (tengo[j]) ans -= edges[j].x;
        else          ans += edges[j].x;
        tengo[j] = !tengo[j];
    }
    return ans;
}
```

Consultas de caminos usando el truco de Mo: complejidad

Esto funciona pero es horrendamente lento. Claramente hace un `for` $O(N)$ por consulta.

El truco de Mo

Fijate que realmente podemos extender y achicar el rango de la consulta:

```
void metesaca(int j) {
    if (tengo[j]) {
        <saca>
    } else {
        <mete>
    }
    tengo[j] = !tengo[j];
}
```

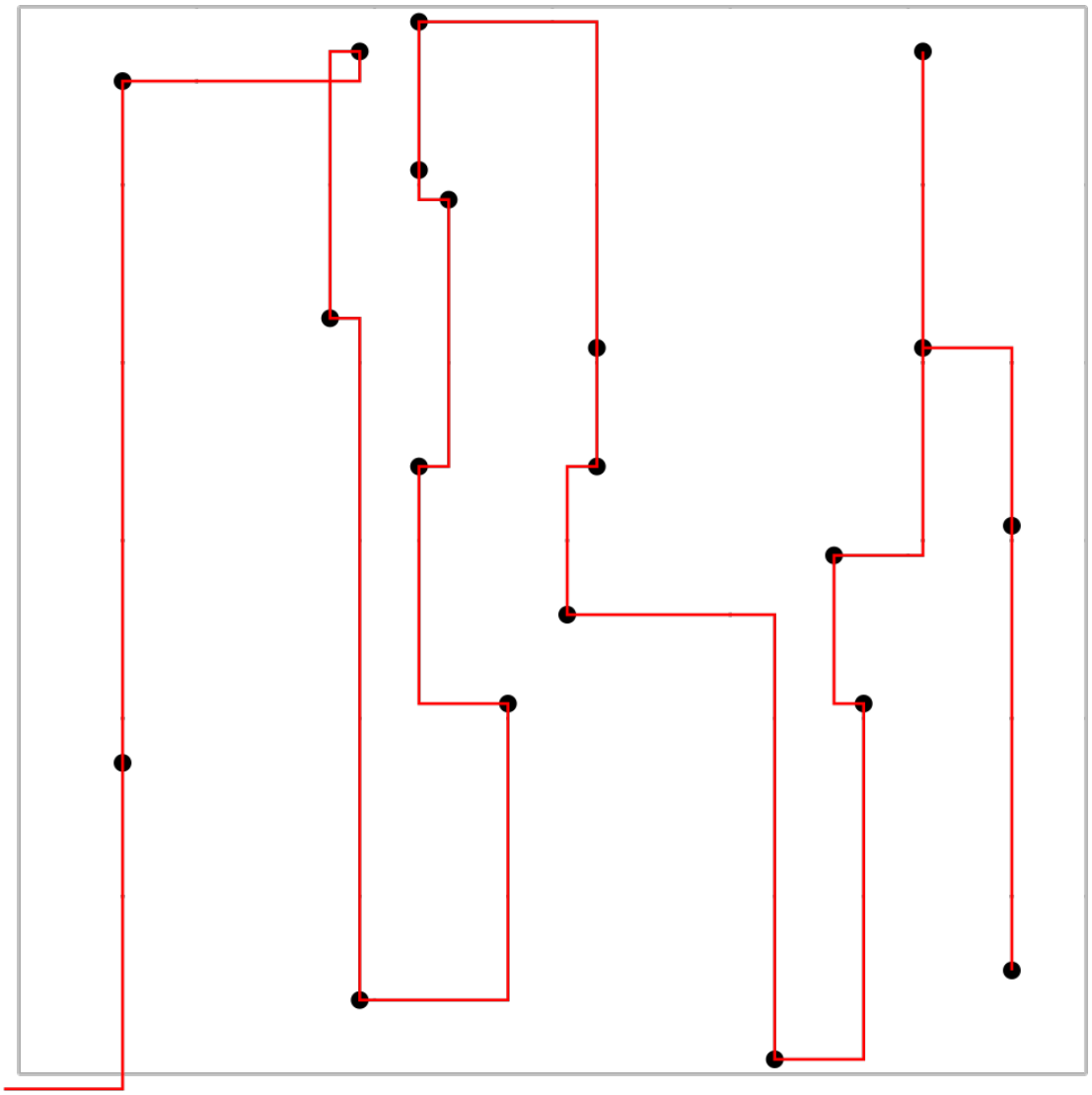
O sea, si tenemos en algún momento acumulado el intervalo $[l, r)$, podemos *metesacar* ($l-1$, l, $r-1$ o r) para obtener los intervalos $[l-1, r)$, $[l+1, r)$, $[l, r-1)$ o $[l, r+1)$.

El truco de Mo: movimiento en el plano

Si pensamos en el intervalo $[l, r)$ como un punto (l, r) en el plano, podemos movernos hacia arriba, abajo, izquierda o derecha.

A medida que nos movemos por el plano, podemos llegar a posiciones que se corresponden con las consultas que tenemos que hacer.
En esos puntos, podemos guardar el valor acumulado para obtener la respuesta de la consulta.

Resulta posible elegir un orden de movimiento tal que el costo total sea $O((N + Q)\sqrt{N})$.



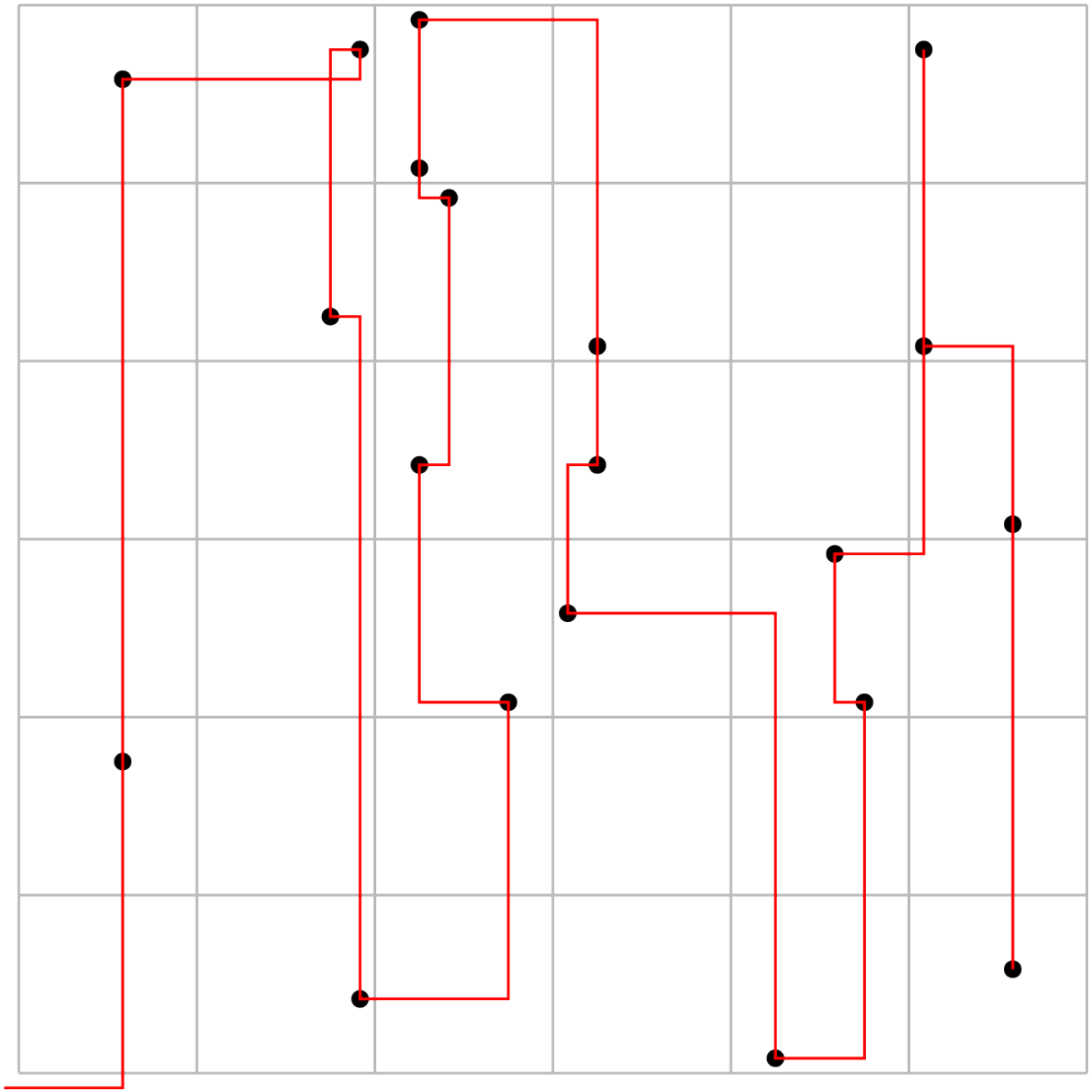
El truco de Mo: complejidad

Partimos el plano en cuadrados de $\sqrt{N} \times \sqrt{N}$ y recorremos los cuadrados en zig zag.

Pasar de cada punto al siguiente cuesta $O(\sqrt{N})$ si está en el mismo bloque. En total sobre todos los puntos es $O(Q\sqrt{N})$.

Podemos considerar por separado el caso de cambiar de bloque.
En total hay $O(N)$ bloques, y moverse de uno a otro cuesta $O(\sqrt{N})$. En total es $O(N\sqrt{N})$.

Entonces, el costo total es $O((Q+N)\sqrt{N})$.



```
vector<int> perm = orden_mo(queries);
int l = 0, r = 0;
for (int i : perm) {
    auto [ql, qr] = queries[i];
    while (ql < l) metesaca(--l);
    while (r < qr) metesaca(r++);
    while (l < ql) metesaca(l++);
    while (qr < r) metesaca(--r);
    ans[i] = <respuesta>
}
```

Gracias