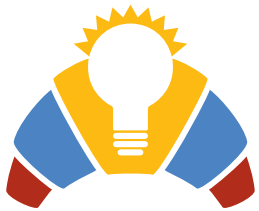
Hashing

Avanzado

Lautaro Lasorsa

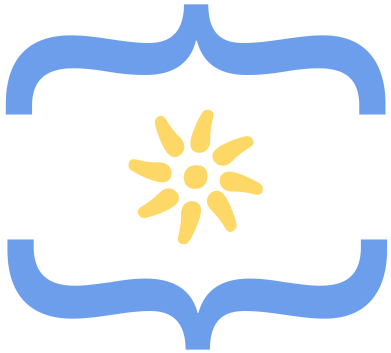
Universidad Nacional de La Plata – Facultad de Informática

Training Camp 2026



Gracias Sponsors!

Organiza



Diamond Sponsor

Outline

Outline

Problemas que veremos cómo resolver con Hashing

Todos son más difíciles de resolver con otras técnicas:

1. **Composición de subarreglos:** muchas queries "¿los subarreglos $[l_1, r_1]$ y $[l_2, r_2]$ tienen el mismo *multiconjunto* de valores?" en $\mathcal{O}(1)$.
   No es un string $\Rightarrow$ Suffix Array no aplica; ordenar cada query es $\mathcal{O}(n)$.
2. **Igualdad de substrings con updates:** comparar $s[a..a+L]$ vs $s[b..b+L]$ intercalado con "cambiar $s[i] = c$".
   Suffix Array/Automaton no soportan updates.
3. **Contar subárboles distintos** (isomorfismo de subárboles enraizados).
   No hay "suffix array de árboles"; hace falta una forma canónica de la estructura.

Outline

La primitiva: el monoide $\{h, b\}$

Definición (Hash polinomial de un string s)

$$H(s) = \sum_{i=0}^{|s|-1} s[i] \cdot P^i \pmod{M}, \qquad s[i] = c - \text{'a'} + 1 \ (\textit{nunca } 0).$$

Si 'a' $\to 0$: "a", "aa", "aaa" $\to 0$, todos colisionan.

Definición (Par-monoide $\{h, b\}$)

$$\{h, b\}: \quad h = \sum s[i] \cdot P^i, \qquad b = P^{|s|}.$$

Concatenación $a * c = \{\, a.h + a.b \cdot c.h, \quad a.b \cdot c.b \,\}$*: asociativa,* **no** *conmutativa, neutro* $\{0, 1\}$.

$\{h, b\}$: código

Módulo de Mersenne $2^{61} - 1$ y mulmod vía __int128

```
const ll B   = 131;                          // base
const ll MOD = 2305843009213693951LL; // modulo 2^61 - 1
inline ll mulmod(ll a, ll b) {
    return (ll)((__int128)a * b % MOD);
}
struct UnitHash {
    ll h, b;
    UnitHash() : h(0), b(1) {}              // neutro ""
    UnitHash(ll h_, ll b_) : h(h_), b(b_) {}
};
// concatenacion: s seguido de t
UnitHash operator*(const UnitHash& s, const UnitHash& t) {
    return UnitHash((s.h + mulmod(s.b, t.h)) % MOD,
                    mulmod(s.b, t.b));
}
```

Tabla aditiva de prefijos... y su trampa

Precomputamos $H[i] = \text{hash}(s[0..i))$ acumulando $s[i-1] \cdot P^{i-1}$: la "tabla aditiva" de S.

Para el hash de $s[i..j)$ hay que quitar el prefijo $s[0..i)$

La resta cruda $H[j] - H[i]$ queda **desalineada** por un factor P^i: ambos hashes pesan $s[k]$ por P^k (potencia absoluta), así que $H[j] - H[i]$ *no* es el hash de $s[i..j)$.

Necesitamos algo mejor: dividir, no restar.

Extensión sancionada: $\{h, b, binv\}$ y el operador /

Definición (Terna $\{h, b, binv\}$)

Extensión **mínima** *del* `UnitHash` *anterior con un tercer campo* $binv = P^{-|s|}$ *(inverso modular de b); neutro* $\{0, 1, 1\}$.

Definición (División: quitar un prefijo en $\mathcal{O}(1)$)

Si h_1 *es* **prefijo** *de* h_2*:*

$$\{h_2, b_2, bi_2\} / \{h_1, b_1, bi_1\} = \{ (h_2 - h_1) \cdot bi_1, \quad b_2 \cdot bi_1, \quad bi_2 \cdot b_1 \}$$

devuelve el hash del **sufijo** *que queda al remover el prefijo* h_1.

Propiedades: $x/x = \{0, 1, 1\}$ (neutro). Si h_1 *no* es prefijo de h_2, el resultado **no tiene valor** (no usarlo).

Hash de substring en $\mathcal{O}(1)$, sin inversos por consulta

Resultado

$$\text{hash}\big(s[i..j]\big) \;=\; H[j] \,/\, H[i]$$

- El *binv* ya viaja *dentro* del monoide: no calculamos inversos por consulta.
- Equivalente al enfoque "arrays planos + inverso modular", pero más limpio y **componible**.

BinPow sobre el monoide: S^k en $\mathcal{O}(\log k)$

Al ser monoide, concatenar un string consigo mismo k veces sale con exponenciación binaria (la operación `repeated`).

`repeated`: S^k por exponenciación binaria

```
UnitHash repeated(const UnitHash& s, long long n) {
    UnitHash res;              // neutro = ""
    UnitHash base = s;
    while (n > 0) {
        if (n & 1) res = res * base;
        base = base * base;
        n >>= 1;
    }
    return res;
}
```

Aplicación directa: decidir $S^p == T^q$ comparando `repeated(Hash(S), p) == repeated(Hash(T), q)`.

Aplicación: comparar A^p vs B^q lexicográficamente

Enunciado

Dados A, B y exponentes $p, q \geq 1$, decidir si $A^p < B^q$ (orden lexicográfico) **sin materializar** las repeticiones (hasta $\sim 10^{18}$ caracteres).

Aplicación: comparar A^p vs B^q lexicográficamente

Enunciado

Dados A, B y exponentes $p, q \geq 1$, decidir si $A^p < B^q$ (orden lexicográfico) **sin materializar** las repeticiones (hasta $\sim 10^{18}$ caracteres).

El carácter en la posición ℓ de X^e es $X[\ell \bmod |X|]$; el hash del prefijo de longitud ℓ sale del monoide $+$ binpow.

```
// hash del prefijo de longitud L de X^e (X con prefijos HX)
UnitHash pref(vector<UH>& HX, ll L) {
    ll m = (ll)HX.size() - 1;                // m = |X|
    return repeated(HX[m], L / m) * HX[L % m];
}
// LCP de A^p y B^q por busqueda binaria: O(log^2)
ll lo = 0, hi = min(p * (ll)A.size(), q * (ll)B.size()) + 1;
while (hi - lo > 1) {          // [lo, hi): ok(lo), !ok(hi)
    ll mid = (lo + hi) / 2;
    if (pref(HA, mid) == pref(HB, mid)) lo = mid;
    else hi = mid;
}
```

Con el LCP $= \ell$: si ℓ agota al más corto, gana el más corto; si no, comparar $A[\ell \bmod |A|]$ vs $B[\ell \bmod |B|]$. Total $\mathcal{O}(|A| + |B| + \log^2)$.

"Hashing mata todo" (showcase): LCP en $\mathcal{O}(\log N)$

Con el monoide solo: dos strings comparten un prefijo de longitud L sii $H_1[L] = H_2[L] \Rightarrow$ LCP por búsqueda binaria.

LCP por búsqueda binaria

```
using vh = vector<UnitHash>;   // h[k] = hash(s[0..k])
int lcp(vh& h1, vh& h2) {
    int lo = 0, hi = (int)min(h1.size(), h2.size());
    while (hi - lo > 1) { // [lo,hi): ok(lo), !ok(hi)
        int mid = (lo + hi) / 2;
        if (h1[mid] == h2[mid]) lo = mid;
        else hi = mid;
    }
    return lo;  // mayor L con h1[L]==h2[L]
}
```

Comparador $<$ por hashing

Con el LCP: comparador lexicográfico en $\mathcal{O}(\log N)$ (LCP + primer carácter distinto; centinela '\0' al terminar). Cada sufijo son dos vistas sin copiar.

```
using UH  = UnitHash;
using Suf = pair<span<UH>, string_view>;  // (hashes, texto)

// hash de los primeros L del sufijo = H[L] / H[0]
int lcp(const Suf& a, const Suf& b) {  // mayor L con pref. iguales
    int lo = 0, hi = (int)min(a.first.size(), b.first.size());
    while (hi - lo > 1) {          // [lo, hi): ok(lo), !ok(hi)
        int m = (lo + hi) / 2;
        if ((a.first[m]/a.first[0]).h ==
            (b.first[m]/b.first[0]).h) lo = m;
        else hi = m;
    }
    return lo;
}
bool comp(const Suf& a, const Suf& b) {    // a < b lexicografico
    int l = lcp(a, b);
    // '\0': innecesario aca, pero deja comp generica
    char ca = l < (int)a.second.size() ? a.second[l] : '\0';
    char cb = l < (int)b.second.size() ? b.second[l] : '\0';
    return ca < cb;
}
```

Suffix Array: ordenar los índices de los sufijos

Generación (cero copias) y ordenamiento de índices

```
// H[k] = hash(s[0..k)) como UnitHash (prefijos)
vector<UH> H = hashPrefijos(s);
int n = (int)s.size();
vector<Suf> suf(n);
for (int i = 0; i < n; i++)
    suf[i] = { span<UH>(H).subspan(i),
               string_view(s).substr(i) };  // vistas, no copian

// ordenamos los INDICES, no los pares (Suf)
vector<int> ord(n);
iota(ord.begin(), ord.end(), 0);
sort(ord.begin(), ord.end(), [&](int i, int j){
    return comp(suf[i], suf[j]);
});
```

Suffix Array en $\mathcal{O}(n \log^2 n)$ **sin** implementar la construcción específica (duplicación de órdenes + *counting sort*, $\mathcal{O}(n \log n)$).

Síntesis

Un solo monoide reemplaza Trie y Suffix Array

Con una fracción del código. Es una ventaja de **implementación**, no de complejidad: es probabilístico y paga un log extra.

Enlaza con §3: hay que hacerlo bien (colisiones).

Señal de reconocimiento

Comparar / igualar / ordenar segmentos de una secuencia, o potencias/repeticiones de un string $\rightarrow$ monoide de hash ($+$ binpow, $+$ búsqueda binaria para LCP).

Aplicación (para pensar): períodos de substrings

Dado s y q consultas (l, r, k): decidir si $s[l..r]$ es una **potencia** de su bloque inicial de longitud k, es decir $s[l..r] = \big(s[l..l+k-1]\big)^{(r-l+1)/k}$. Con $n, q \leq 2 \cdot 10^5$. ¿Cómo responder cada consulta en $\mathcal{O}(1)$?

Aplicación (para pensar): períodos de substrings

Dado s y q consultas (l, r, k): decidir si $s[l..r]$ es una **potencia** de su bloque inicial de longitud k, es decir $s[l..r] = \big(s[l..l+k-1]\big)^{(r-l+1)/k}$. Con $n, q \leq 2 \cdot 10^5$. ¿Cómo responder cada consulta en $\mathcal{O}(1)$?

Solución

$s[l..r] = \text{bloque}^m$ sii se cumplen **las dos**:

1. **Divisibilidad:** $k \mid (r - l + 1)$.
2. **Período k en el rango:** equivale a la igualdad de los substrings *solapados* $s[l..r-k] = s[l+k..r]$, que el hash decide en $\mathcal{O}(1)$ con la tabla de prefijos.

Ninguna sola alcanza: la divisibilidad no fuerza que los bloques coincidan; el solapamiento sin divisibilidad deja un bloque parcial.

Bonus: período, solapamiento y potencia

Definición (Período)

$s[l..r]$ tiene período k si $s[i] = s[i+k]$ para todo $i \in [l, r-k]$.

Período $\iff$ igualdad solapada. Escribir esas $r-k-l+1$ igualdades carácter a carácter, todas juntas, es exactamente

$$s[l..r-k] \;=\; s[l+k..r]$$

Es una reescritura, no un teorema: por eso el hash la decide de una.

Teorema (Período + divisibilidad $\Rightarrow$ potencia)

Si $s[l..r]$ tiene período k y $k \mid (r-l+1)$, entonces $s[l..r] = \big(s[l..l+k-1]\big)^m$ con $m = (r-l+1)/k$.

El período dice que cada bloque de largo k es igual al anterior; la divisibilidad garantiza que el último está **completo**. Sin ella queda un bloque parcial: `"abcab"` tiene período 3 pero no es potencia de `"abc"`.

Aplicación (para pensar): substring común mayor

Dados N strings cuya longitud total es T, decidir la mayor longitud posible de un string L que es substring (contiguo) de todos los strings.

Aplicación (para pensar): substring común mayor

Dados N strings cuya longitud total es T, decidir la mayor longitud posible de un string L que es substring (contiguo) de todos los strings.

Solución

Búsqueda binaria sobre la longitud L del substring común.

- `ok(L)`: partir del conjunto de hashes de los substrings de largo L del primer string e intersecarlo con el de cada string siguiente, cortando si queda vacío. Cuesta $\mathcal{O}(T)$.
- **Monótono:** si hay uno común de largo L, sus substrings de largo $L-1$ también son comunes.
- Buscar sobre $[0, m+1)$ con m el largo mínimo, invariante `ok(lo)` / `!ok(hi)`: devuelve `lo` en $\mathcal{O}(T \log m)$.

Nota: También sale con Suffix Array + Union Find, en $\mathcal{O}(T \log T)$.

Outline

Motivación

Definición (Colisión)

Dos objetos distintos con el mismo hash.

Por qué importa

Una colisión produce un **WA silencioso** (no un TLE ni un RTE). El riesgo es mayor cuando se comparan muchos objetos entre sí (régimen *todos contra todos*).

Probabilidad de colisión: los tres regímenes

Lo que importa es Pr[al menos una colisión]. Se calcula bajo el modelo ideal: h uniforme sobre M valores e independiente para objetos distintos.

- **Un par** $x \neq y$: fijado $h(x)$, exactamente un valor de M es malo.

$$\Pr[\geq 1] = \frac{1}{M}$$

Probabilidad de colisión: los tres regímenes

Lo que importa es Pr[al menos una colisión]. Se calcula bajo el modelo ideal: h uniforme sobre M valores e independiente para objetos distintos.

- **Un par** $x \neq y$: fijado $h(x)$, exactamente un valor de M es malo.

$$\Pr[\geq 1] = \frac{1}{M}$$

- **Un objeto contra** k: condicionando en $h(x) = v$, los k eventos $h(y_i) = v$ sí son independientes y cada uno se evita con probabilidad $1 - 1/M$.

$$\Pr[\geq 1] = 1 - \left(1 - \frac{1}{M}\right)^k \leq \frac{k}{M}$$

Probabilidad de colisión: los tres regímenes

Lo que importa es Pr[al menos una colisión]. Se calcula bajo el modelo ideal: h uniforme sobre M valores e independiente para objetos distintos.

- **Un par** $x \neq y$: fijado $h(x)$, exactamente un valor de M es malo.

$$\Pr[\geq 1] = \frac{1}{M}$$

- **Un objeto contra** k: condicionando en $h(x) = v$, los k eventos $h(y_i) = v$ sí son independientes y cada uno se evita con probabilidad $1 - 1/M$.

$$\Pr[\geq 1] = 1 - \left(1 - \frac{1}{M}\right)^k \leq \frac{k}{M}$$

- **Todos contra todos** (k objetos): los eventos por par ya no son independientes, porque $h(x_1) = h(x_2)$ y $h(x_1) = h(x_3)$ fuerzan $h(x_2) = h(x_3)$. El complemento se cuenta insertando los objetos de a uno.

$$\Pr[\geq 1] = 1 - \prod_{i=1}^{k-1} \left(1 - \frac{i}{M}\right)$$

Todos contra todos: paradoja del cumpleaños

El producto sale de la regla de la **probabilidad compuesta**: al insertar el objeto $i+1$ hay exactamente i valores ocupados distintos (condicionado a que no hubo colisión antes), así que sobrevive con probabilidad $(M-i)/M$.

Todos contra todos: paradoja del cumpleaños

El producto sale de la regla de la **probabilidad compuesta**: al insertar el objeto $i+1$ hay exactamente i valores ocupados distintos (condicionado a que no hubo colisión antes), así que sobrevive con probabilidad $(M-i)/M$.

Teorema (Acotación por ambos lados)

Definiendo $C = \dfrac{k(k-1)}{2M} \approx \dfrac{k^2}{2M}$*:*

$$1 - e^{-C} \;\leq\; \Pr[\geq 1] \;\leq\; C$$

Bonus: de dónde sale la cota inferior $1 - e^{-C}$

Desigualdad elemental

Para todo x real vale $1 - x \leq e^{-x}$: la recta $1 - x$ es tangente en el origen a la convexa e^{-x}. Formalmente, $f(x) = e^{-x} - (1 - x)$ cumple $f(0) = 0$ y $f'(x) = 1 - e^{-x}$, negativa para $x < 0$ y positiva para $x > 0$, así que su mínimo es 0.

Paso 1. Aplicarla con $x = i/M$ a cada uno de los $k - 1$ factores del producto exacto.

$$1 - \frac{i}{M} \;\leq\; e^{-i/M} \qquad i = 1, \ldots, k-1$$

Paso 2. Multiplicar las $k - 1$ desigualdades, válido porque todos los factores son no negativos ($i < M$; si $k > M$ la colisión es segura por palomar).

$$\Pr[\text{sin colisión}] = \prod_{i=1}^{k-1} \left(1 - \frac{i}{M}\right) \;\leq\; \prod_{i=1}^{k-1} e^{-i/M}$$

Paso 3. Los exponentes se suman, y la suma de Gauss $\sum_{i=1}^{k-1} i = \binom{k}{2} = \frac{k(k-1)}{2}$ produce exactamente C.

$$\prod_{i=1}^{k-1} e^{-i/M} = \exp\left(-\frac{1}{M}\sum_{i=1}^{k-1} i\right) = \exp\left(-\frac{k(k-1)}{2M}\right) = e^{-C}$$

Paso 4. Complementar invierte el sentido de la desigualdad.

$$\Pr[\geq 1] = 1 - \Pr[\text{sin colisión}] \;\geq\; 1 - e^{-C}$$

Bonus: lectura de las dos cotas

La cota superior e^{-C} sobre Pr[sin colisión] es **optimista**: sobreestima cada factor del producto. La probabilidad real de no colisionar es todavía menor y, por lo tanto, la de colisionar es todavía mayor que $1 - e^{-C}$. Por eso sirve como garantía pesimista: si $1 - e^{-C}$ ya resulta alto, el riesgo real es peor.

Dos caminos independientes

La otra mitad del sándwich, $\Pr[\geq 1] \leq C$, se obtiene por un argumento distinto: la cota de la unión, que suma $1/M$ sobre los $\binom{k}{2}$ pares. No toca el producto ni supone independencia entre los eventos, y por eso es la que sobrevive fuera del modelo ideal.

$$1 - e^{-C} \;\leq\; \Pr[\geq 1] \;\leq\; C$$

Ambos extremos usan la misma cantidad C, y como $1 - e^{-C} \leq C$ para todo $C \geq 0$, el intervalo nunca es vacío. Cuando $C \ll 1$ se cierra: $1 - e^{-C} \approx C$.

Cuántos objetos tolera cada módulo

El quiebre está en $\sqrt{M}$, no en M: en eso consiste la paradoja. Despejando k del sándwich, $\Pr[\geq 1] = 1/2$ pide $C = \ln 2$ y una confianza $\Pr[\geq 1] = \varepsilon \ll 1$ pide $C \approx \varepsilon$:

$$k_{1/2} \approx \sqrt{2 \ln 2 \cdot M} \approx 1{,}18\sqrt{M}, \qquad k_\varepsilon \approx \sqrt{2\varepsilon M}$$

Cuántos objetos tolera cada módulo

El quiebre está en $\sqrt{M}$, no en M: en eso consiste la paradoja. Despejando k del sándwich, $\Pr[\geq 1] = 1/2$ pide $C = \ln 2$ y una confianza $\Pr[\geq 1] = \varepsilon \ll 1$ pide $C \approx \varepsilon$:

$$k_{1/2} \approx \sqrt{2 \ln 2 \cdot M} \approx 1{,}18\sqrt{M}, \qquad k_\varepsilon \approx \sqrt{2\varepsilon M}$$

Máximo k admisible para cada nivel de riesgo:

Módulo	$\Pr[\geq 1] = 1/2$	$\Pr[\geq 1] = 10^{-4}$
$M \approx 10^9$	$\approx 3{,}7 \times 10^4$	$\approx 4{,}5 \times 10^2$
$M = 2^{61} - 1$	$\approx 1{,}8 \times 10^9$	$\approx 2{,}1 \times 10^7$
Doble hash, $M_1 M_2 \approx 10^{18}$	$\approx 1{,}2 \times 10^9$	$\approx 1{,}4 \times 10^7$

Cuántos objetos tolera cada módulo

El quiebre está en $\sqrt{M}$, no en M: en eso consiste la paradoja. Despejando k del sándwich, $\Pr[\geq 1] = 1/2$ pide $C = \ln 2$ y una confianza $\Pr[\geq 1] = \varepsilon \ll 1$ pide $C \approx \varepsilon$:

$$k_{1/2} \approx \sqrt{2 \ln 2 \cdot M} \approx 1{,}18\sqrt{M}, \qquad k_\varepsilon \approx \sqrt{2\varepsilon M}$$

Máximo k admisible para cada nivel de riesgo:

Módulo	$\Pr[\geq 1] = 1/2$	$\Pr[\geq 1] = 10^{-4}$
$M \approx 10^9$	$\approx 3{,}7 \times 10^4$	$\approx 4{,}5 \times 10^2$
$M = 2^{61} - 1$	$\approx 1{,}8 \times 10^9$	$\approx 2{,}1 \times 10^7$
Doble hash, $M_1 M_2 \approx 10^{18}$	$\approx 1{,}2 \times 10^9$	$\approx 1{,}4 \times 10^7$

- La columna de 10^{-4} es la que corresponde a **enviar** una solución.
- Exigirla en lugar de una moneda al aire solo divide el umbral por ≈ 83: ambos escalan con $\sqrt{M}$.
- Con $M \approx 10^9$ y $k = 10^6$ resulta $C = 500$: la colisión es **prácticamente segura**.
- Comparar muchos conjuntos, estados o subárboles entre sí cae **siempre** en este régimen.

Elección de módulo: nunca 2^{64}

- Usar un **primo grande**, no potencias de 2.
- **Nunca** 2^{64} (overflow natural de `unsigned long long`): el ataque de **Thue–Morse** rompe cualquier base impar de forma *determinista* (sin conocer la base elegida).

Dos soluciones seguras

1. **Mersenne** $2^{61} - 1$ con `mulmod` vía `__int128` (ya visto).
2. **Doble hashing:** `pair<ll,ll>` con dos (*base*, *módulo*) independientes $\rightarrow$ colisión $\approx 1/(M_1 M_2) \approx 10^{-18}$.

Anti-hash tests: aleatorizar la base

El único cambio sobre el código ya visto: **base aleatoria en runtime**.

Semilla de alta resolución

```
mt19937_64 rng(
    chrono::steady_clock::now().time_since_epoch().count());
// base aleatoria: distinta en cada corrida del binario
ll B = 131 + rng() % (MOD - 200);
```

- El hacker no conoce los parámetros antes de que corra tu binario → no puede construir colisiones a propósito.
- **Evitar** `std::rand()` y `time(0)` (adivinable).

Señal de reconocimiento

Insertar muchos hashes en un `set`/`map`, o comparar de a pares muchísimas veces, implica estar en $k^2/2M$: hace falta $2^{61}-1$ o doble hash, y base aleatoria si hay hacking.

Aplicación (para pensar): elegir parámetros seguros

Un problema pide contar cuántos de los $\approx 10^6$ substrings de longitud fija L de un texto son **distintos**, insertando sus hashes en un `set`. El juez tiene **fase de hacking**. ¿Qué esquema de parámetros es adecuado?

Aplicación (para pensar): elegir parámetros seguros

Un problema pide contar cuántos de los $\approx 10^6$ substrings de longitud fija L de un texto son **distintos**, insertando sus hashes en un `set`. El juez tiene **fase de hacking**. ¿Qué esquema de parámetros es adecuado?

Solución

- Régimen **todos-contra-todos**: $\approx 10^6$ hashes comparados entre sí $\Rightarrow$ vale $k^2/2M$.
- Hace falta ~ 61 bits $(2^{61}-1)$ o **doble hash**: un solo primo $\approx 10^9+7$ da ≈ 500 colisiones esperadas.
- Hay hacking $\Rightarrow$ **base aleatoria en runtime**. Base fija es atacable, y 2^{64} está roto de forma determinista (Thue–Morse, sin conocer la base).
- Materializar y ordenar los substrings cuesta $\mathcal{O}(nL)$: demasiado lento.

Outline

Aplicación 1: colecciones invariantes al orden

Enunciado

Decidir si dos colecciones de elementos son **iguales** (mismos elementos, ignorando el orden) en $\mathcal{O}(1)$, soportando además operaciones entre ellas: **agregar**/**quitar** un elemento, **unión**, comparación de sub-rangos, etc.

Aplicación 1: colecciones invariantes al orden

Enunciado

Decidir si dos colecciones de elementos son **iguales** (mismos elementos, ignorando el orden) en $\mathcal{O}(1)$, soportando además operaciones entre ellas: **agregar**/**quitar** un elemento, **unión**, comparación de sub-rangos, etc.

Por qué hashing gana: no hay estructura de strings que sirva; comparar composición de colecciones es $\mathcal{O}(\text{tamaño})$ salvo que colapses la colección a un número **invariante al orden**.

Aplicación 1: colecciones invariantes al orden

Enunciado

Decidir si dos colecciones de elementos son **iguales** (mismos elementos, ignorando el orden) en $\mathcal{O}(1)$, soportando además operaciones entre ellas: **agregar**/**quitar** un elemento, **unión**, comparación de sub-rangos, etc.

Por qué hashing gana: no hay estructura de strings que sirva; comparar composición de colecciones es $\mathcal{O}(\text{tamaño})$ salvo que colapses la colección a un número **invariante al orden**.

Definición (Fingerprint de conjunto y de multiconjunto)

Se asigna un valor aleatorio de 64 bits $r(x)$ por elemento (`splitmix64`) y se combinan los de la colección:

- **Conjunto** *(importa presencia): con* **XOR**.
- **Multiconjunto** *(importa multiplicidad): con* **suma** *(mód 2^{64}).*

XOR y suma son conmutativas y asociativas $\Rightarrow$ el fingerprint no depende del orden.

splitmix64: id aleatorio determinista por elemento

Adaptado de neal, CF blog 62393 (splitmix64: S. Vigna)

```
uint64_t splitmix64(uint64_t x) {
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
}
// semilla fija por corrida (no time(0): adivinable)
const uint64_t SEED =
    chrono::steady_clock::now().time_since_epoch().count();
// id de x: mezcla aplicada directo, sin map/cache
uint64_t getRand(uint64_t x) {
    return splitmix64(x + SEED);
}
```

Fingerprint de la colección: **XOR** (conjuntos) o **suma** (multiconjuntos) de getRand sobre sus elementos.

Prefijos XOR / suma: subarreglos en $\mathcal{O}(1)$

```
// Conjunto (presencia): prefijos XOR
vector<uint64_t> pxor(n + 1, 0);
for (int i = 0; i < n; i++)
    pxor[i + 1] = pxor[i] ^ getRand(a[i]);

// Multiconjunto (multiplicidad): prefijos suma
vector<uint64_t> psum(n + 1, 0);
for (int i = 0; i < n; i++)
    psum[i + 1] = psum[i] + getRand(a[i]);
// multiconjunto de [l, r) = psum[r] - psum[l]
```

Cuidado con XOR y paridad: $r(x) \oplus r(x) = 0$; un valor con multiplicidad *par* se cancela. Para multiconjuntos usar **suma** (o un set auxiliar). Esto resuelve el primer problema de la introducción.

Outline

Aplicación 2: strings módulo rotación

Enunciado

Decidir si dos substrings son **rotación** uno del otro ("abcab" $\sim$ "cabab"), en $\mathcal{O}(1)$ por consulta.

Aplicación 2: strings módulo rotación

Enunciado

Decidir si dos substrings son **rotación** uno del otro (`"abcab"` $\sim$ `"cabab"`), en $\mathcal{O}(1)$ por consulta.

Falta un invariante ciego a la rotación y sensible a todo lo demás.
Las dos que ya tenemos fallan por extremos opuestos:

- Hash **polinomial** (§2): sensible al orden $\rightarrow$ separa rotaciones que deberían ser iguales.
- Fingerprint de **multiconjunto** (§4): ignora el orden *por completo* $\rightarrow$ identifica anagramas que no son rotación.

Aplicación 2: strings módulo rotación

Enunciado

Decidir si dos substrings son **rotación** uno del otro ("abcab" $\sim$ "cabab"), en $\mathcal{O}(1)$ por consulta.

Falta un invariante ciego a la rotación y sensible a todo lo demás.
Las dos que ya tenemos fallan por extremos opuestos:

- Hash **polinomial** (§2): sensible al orden $\rightarrow$ separa rotaciones que deberían ser iguales.
- Fingerprint de **multiconjunto** (§4): ignora el orden *por completo* $\rightarrow$ identifica anagramas que no son rotación.

Definición (Hash rotacional por traza)

Una matriz aleatoria $R_c \in \mathbb{Z}_M^{d\times d}$ *por carácter* ($d = 3$). *Para* $s = s_0 \cdots s_{n-1}$:

$$R(s) = R_{s_0} R_{s_1} \cdots R_{s_{n-1}}, \qquad h(s) = \operatorname{tr} R(s).$$

Por qué la traza no ve la rotación

- **Rotar = conjugar.** Mover s_0 al final transforma $R \mapsto R_{s_0}^{-1} R\, R_{s_0}$: el mismo producto leído desde otro punto del ciclo.
- **La traza no distingue conjugados:** $\operatorname{tr}(AB) = \operatorname{tr}(BA) \Rightarrow \operatorname{tr}(P^{-1}RP) = \operatorname{tr}(R)$.
- **Colisión** $\leq n/M$: mismo argumento de Schwartz–Zippel de §3.

Por qué la traza no ve la rotación

- **Rotar = conjugar.** Mover s_0 al final transforma $R \mapsto R_{s_0}^{-1} R\, R_{s_0}$: el mismo producto leído desde otro punto del ciclo.
- **La traza no distingue conjugados:** $\operatorname{tr}(AB) = \operatorname{tr}(BA) \Rightarrow \operatorname{tr}(P^{-1}RP) = \operatorname{tr}(R)$.
- **Colisión** $\leq n/M$: mismo argumento de Schwartz–Zippel de §3.

Requisitos: $d \geq 3$ y matrices densas

- $d = 1$: el producto conmuta $\rightarrow$ colapsa al fingerprint de multiconjunto.
- $d = 2$: la traza de un producto no cambia al invertir el orden $\rightarrow$ colisionan los **reversos**.
- Entradas **diagonales o triangulares**: la traza depende solo del conteo por carácter. Las d^2 entradas van independientes.

Traza de matrices: código

Reusa MOD y mulmod (§2), getRand (§4).

```
const int D = 3;                    // D >= 3 (ver requisitos)
struct Mat { ll a[D][D]; };         // DxD sobre Z_MOD
Mat operator*(const Mat& x, const Mat& y) {
    Mat z{};
    for (int i = 0; i < D; i++) for (int k = 0; k < D; k++) {
        ll v = x.a[i][k];
        for (int j = 0; j < D; j++)
            z.a[i][j] = (z.a[i][j] + mulmod(v, y.a[k][j])) % MOD;
    }
    return z;
}
ll tr(const Mat& m) {               // la traza es el hash
    ll s = 0;
    for (int i = 0; i < D; i++) s = (s + m.a[i][i]) % MOD;
    return s;
}
Mat R[256];                         // matriz aleatoria por caracter
void initMats() {
    for (int c = 0, k = 0; c < 256; c++)
        for (int i = 0; i < D; i++) for (int j = 0; j < D; j++)
            R[c].a[i][j] = getRand(k++) % MOD;   // k inyectivo
}
```

El monoide $\{R, R^{-1}\}$ y el operador /

Mismo empaquetado que en §2, con matrices.

Definición (Par-monoide $\{R, R^{-1}\}$)

$$\{R,\ R^{-1}\}(s) = \{\, R_{s_0} \cdots R_{s_{n-1}},\quad R^{-1}_{s_{n-1}} \cdots R^{-1}_{s_0} \,\}$$

Concatenación $a * c = \{\, a.R \cdot c.R,\quad c.R^{-1} \cdot a.R^{-1} \,\}$*: asociativa,* **no** *conmutativa, neutro* $\{I, I\}$*.*

El monoide $\{R, R^{-1}\}$ y el operador /

Mismo empaquetado que en §2, con matrices.

Definición (Par-monoide $\{R, R^{-1}\}$)

$$\{R,\ R^{-1}\}(s) = \{\, R_{s_0} \cdots R_{s_{n-1}},\quad R^{-1}_{s_{n-1}} \cdots R^{-1}_{s_0} \,\}$$

Concatenación $a * c = \{\, a.R \cdot c.R,\quad c.R^{-1} \cdot a.R^{-1} \,\}$*: asociativa,* **no** *conmutativa, neutro* $\{I, I\}$*.*

Definición (División: quitar un prefijo en $\mathcal{O}(1)$)

Si a *es* **prefijo** *de* b*:*

$$b \,/\, a = \{\, a.R^{-1} \cdot b.R,\quad b.R^{-1} \cdot a.R \,\}$$

es el par del **sufijo** *restante.*

El hash es tr del primer campo. A diferencia de *binv*, acá R^{-1} invierte al hash mismo: es un **grupo**.

Substrings módulo rotación en $\mathcal{O}(1)$

Resultado

Con la tabla de prefijos $H[i] = \{R, R^{-1}\}(s[0..i))$:

$$h(s[i..j)) = \text{tr}\left((H[j] \,/\, H[i]).R\right)$$

- Los inversos por carácter se precomputan una vez (adj / det, con $\det^{-1}$ por Fermat): el monoide no invierte nada por consulta.
- Una matriz aleatoria es singular con probabilidad $\sim d/M$; si $\det = 0$, resortear ese carácter.
- **Complejidad:** $\mathcal{O}(nd^3)$ de preproceso, $\mathcal{O}(d^3)$ por consulta.
- Lo que **no** da la alternativa clásica: duplicar y buscar ($t \subseteq ss$) cuesta $\mathcal{O}(L)$ por substring.

RotHash: código

Mismo esqueleto que `UnitHash` (§2), con matrices.

```
Mat ID, Rinv[256];        // Rinv[c] = adj(R[c]) / det (Fermat)
struct RotHash {
    Mat m, mi;                          // producto y su inverso
    RotHash() : m(ID), mi(ID) {}        // neutro ""
    RotHash(Mat m_, Mat mi_) : m(m_), mi(mi_) {}
};
// concatenacion: s seguido de t
RotHash operator*(const RotHash& s, const RotHash& t) {
    return RotHash(s.m * t.m, t.mi * s.mi);
}
// quitar el prefijo s de t (s prefijo de t)
RotHash operator/(const RotHash& t, const RotHash& s) {
    return RotHash(s.mi * t.m, t.mi * s.m);
}
ll rotHash(const RotHash& x) { return tr(x.m); }
```

Prefijos H como en §2: el hash de $s[i..j)$ es `rotHash(H[j] / H[i])`.

Refuerzo: la tupla de trazas

- La traza no es el único invariante de conjugación: $\mathrm{tr}(R), \mathrm{tr}(R^2), \ldots, \mathrm{tr}(R^{d-1})$ también lo son, y son **independientes** entre sí.
- Juntos determinan el **polinomio característico** de R: son toda la información de R que sobrevive a la rotación.
- Emitir la tupla en vez de una sola traza cuesta lo mismo asintóticamente (d es constante) y hace las colisiones mucho más raras.
- Con $d = 3$ y hash de 61 bits, el par $(\mathrm{tr}(R), \mathrm{tr}(R^2))$ da 122 bits de aleatoriedad. Suficiente para casi cualquier escenario.

Bonus: ¿cuándo 61 bits no alcanzan?

Union bound sobre pares

Con Q strings dos a dos no equivalentes y de largo $\leq n$:

$$\Pr[\exists \text{ colisión}] \leq \binom{Q}{2} \cdot \frac{n}{M} \approx \frac{Q^2 n}{2M} \implies Q_{\max} = \sqrt{\frac{2M\varepsilon}{n}}$$

- **De dónde sale el factor n:** la cota por par no es $1/M$ sino n/M, porque $\operatorname{tr} R(s) - \operatorname{tr} R(t)$ es homogéneo de **grado** n (un factor por carácter). d cambia la cantidad de variables, no el grado. Es el mismo n del hash polinomial, no un costo de esta técnica.
- $Q = n = 10^6$ con $M = 2^{61} - 1$ da cota $\approx 0{,}22$; para $\varepsilon = 10^{-6}$, $Q_{\max} \approx 2100$.
- **En un problema normal no pasa:** si el input mide N en total, $Qn \leq N$ y la cota cae a $N^2/2M \approx 2 \cdot 10^{-7}$. Muchas strings $\Rightarrow$ cortas; largas $\Rightarrow$ pocas.
- **El régimen peligroso es el que motivó la técnica:** con consultas de substrings Q se desacopla del input (10^6 consultas dan 10^6 strings distintas de largo 10^6). Ídem strings generadas: todas las rotaciones, DP sobre concatenaciones.
- Con el par de 122 bits, $Q_{\max} = \frac{M}{n}\sqrt{2\varepsilon} \approx 3 \cdot 10^9$. Siempre **como par**: colapsarlo a un u64 tira el refuerzo a la basura.

Bonus: dos familias 3×3, no una 4×4

esquema	**mult/car.**	**comp.**	**bits**	**exponente**
$d = 3$, solo $\operatorname{tr} R$	27	1	61	1
$d = 3$, $(\operatorname{tr} R, \operatorname{tr} R^2)$	27	2	122	2 heurístico
$d = 4$, $(\operatorname{tr} R^k)_{k \le 3}$	64	3	183	3 heurístico
$2 \times (d = 3)$, el par en cada una	54	4	244	2 probado + 2 heur.

- **El argumento decisivo es la independencia.** Dentro de una familia, $\operatorname{tr} R - \operatorname{tr} R'$ y $\operatorname{tr} R^2 - \operatorname{tr} R'^2$ son polinomios en **las mismas** variables: que se anulen a la vez es $\approx (n/M)^2$ solo si no comparten componente irreducible (plausible, no demostrado). Con dos familias las variables son disjuntas y $\Pr[f(X) = 0 \wedge g(Y) = 0] = \Pr[f = 0] \Pr[g = 0]$ sale por independencia estadística, sin hipótesis.
- **Y encima es más barato:** $54 < 64$ productos por carácter, con más componentes. Además $\operatorname{tr}(R^k)$ tiene grado kn, o sea cota kn/M: las invariantes altas de $d = 4$ son individualmente más débiles.
- **Regla: nunca subas la dimensión, duplicá.**

Bonus: la excepción es la memoria

Los prefijos de una matriz 3×3 ocupan 9 u64 $= 72$ B por posición y por familia. Con $n = 10^6$:

- Una familia, solo A: **72 MB**.
- Una familia, A y B: **144 MB**.
- Dos familias, A y B: **288 MB** $\Rightarrow$ te pasás de los 256 MB habituales.

Mitigación: guardar solo A e invertir en la consulta

$A_i^{-1} = \text{adj}(A_i) \cdot \det(A_i)^{M-2}$: la exponenciación son ~ 60 productos, más la adjunta. La query pasa de $\sim$27 a $\sim$100 operaciones (sigue siendo $\mathcal{O}(1)$) y la memoria baja a la mitad.

Regla práctica: si es *memory-bound*, una familia $d = 3$ con el par de trazas y el heurístico. Si sobran los 288 MB o las strings son cortas, dos familias.

Outline

Aplicación 3: fingerprint de estados

Por qué hashing gana

Serializar el estado completo (tablero, conjunto dinámico) para meterlo en un `set` de visitados es demasiado caro. Zobrist da un fingerprint de 64 bits **actualizable en** $\mathcal{O}(1)$ por movimiento.

Definición (Hash de Zobrist)

Valor aleatorio de 64 bits por par `(posición, estado)`; el hash del estado es el **XOR** *de los presentes. Es el caso particular de §4 aplicado a un tablero/estado.*

Actualización incremental: XOR-out lo que sale, XOR-in lo que entra, en $\mathcal{O}(1)$ por movimiento.

Zobrist: código

Esquema estándar (Wikipedia / Chess Programming Wiki)

```
uint64_t zob[NUM_PIEZAS][NUM_CASILLAS];  // precomputado

uint64_t boardHash = 0;
for (auto [pieza, casilla] : piezasEnTablero)
    boardHash ^= zob[pieza][casilla];

// mover una pieza de 'from' a 'to' (sin captura):
boardHash ^= zob[pieza][from];   // sacar  (O(1))
boardHash ^= zob[pieza][to];     // poner
```

Aplicaciones: dedup de estados visitados en DFS/backtracking, tablas de transposición en juegos, conjuntos dinámicos con inserción/borrado.

Aplicación (para pensar): anagramas por consulta

Dadas dos cadenas s, t y q consultas (i, j, L): decidir si $s[i..i+L)$ y $t[j..j+L)$ son **anagramas** (mismo multiconjunto de caracteres), en $\mathcal{O}(1)$ por consulta. Con $n, q \leq 2 \cdot 10^5$. ¿Cuál es el enfoque?

Aplicación (para pensar): anagramas por consulta

Dadas dos cadenas s, t y q consultas (i, j, L): decidir si $s[i..i+L)$ y $t[j..j+L)$ son **anagramas** (mismo multiconjunto de caracteres), en $\mathcal{O}(1)$ por consulta. Con $n, q \leq 2 \cdot 10^5$. ¿Cuál es el enfoque?

Solución

Valor aleatorio de 64 bits por carácter y prefijos **suma**: dos rangos son anagramas sii coinciden sus sumas (mód 2^{64}). La suma ignora el orden y preserva la multiplicidad, o sea que es el fingerprint de multiconjunto.

Trampas:

- **XOR**: se autocancela de a pares $\rightarrow$ una letra con multiplicidad par desaparece.
- Hash **polinomial**: sensible al orden, distinguiría anagramas.
- **Ordenar**: $\mathcal{O}(L \log L)$ por consulta, viola el $\mathcal{O}(1)$.
- **Histogramas**: $\mathcal{O}(\Sigma) \neq \mathcal{O}(1)$

Outline

Aplicación 4: hashing de árboles enraizados

Definición (Id canónico de un árbol enraizado)

La **firma** *de un nodo es la lista de los ids de sus hijos* **ordenados**; *una hoja recibe una constante. El* **id** *del nodo es el que un diccionario global asigna a esa firma.*

Ordenar los ids de los hijos hace la firma **invariante a permutar hijos**: dos subárboles isomorfos reciben el mismo id.

Hashing de árboles enraizados: etiquetado canónico (AHU)

Diccionario global firma $\rightarrow$ id (algoritmo AHU)

```
struct TreeCanonizer {
    map<vector<int>, int> idOf;   // hijos ordenados -> id

    int canonicalId(int u, int p,
                    const vector<vector<int>>& adj) {
        vector<int> childIds;
        for (int v : adj[u]) if (v != p)
            childIds.push_back(canonicalId(v, u, adj));
        sort(childIds.begin(), childIds.end());// normalizar
        return idOf.try_emplace(
            childIds, (int)idOf.size()).first->second;
    }
};
// isomorfos sii, con UNA instancia compartida:
//   c.canonicalId(r1,-1,adj1) == c.canonicalId(r2,-1,adj2)
```

Árboles no enraizados y aplicaciones

- **Sin raíz:** enraizar en el/los **centroide(s)** (invariantes bajo isomorfismo). Si hay dos, comparar el par ordenado $\{id(c_1), id(c_2)\}$.
- Este etiquetado canónico es el algoritmo **AHU**: al ser determinista, dos árboles reciben el mismo id **si y solo si** son isomorfos.
- Aplicaciones: isomorfismo, **contar subárboles distintos**, detectar simetrías/automorfismos.

Señal de reconocimiento

El problema compara formas de (sub)árbol / cuenta estructuras arbóreas distintas $\rightarrow$ id canónico con hijos ordenados; si no hay raíz, centroide.

Aplicación (para pensar): pares de subárboles isomorfos

Dado un árbol enraizado de $n \leq 2 \cdot 10^5$ nodos, contar cuántos **pares** de nodos $\{u, v\}$ tienen subárboles (colgando de u y de v) isomorfos entre sí. ¿Cuál es el enfoque?

Aplicación (para pensar): pares de subárboles isomorfos

Dado un árbol enraizado de $n \leq 2 \cdot 10^5$ nodos, contar cuántos **pares** de nodos $\{u, v\}$ tienen subárboles (colgando de u y de v) isomorfos entre sí. ¿Cuál es el enfoque?

Solución

- **Id canónico** por subárbol (firma = ids de los hijos **ordenados**, hoja = constante): son isomorfos sii comparten id.
- Agrupar por id y sumar $\binom{c_k}{2}$ por clase. Total $\mathcal{O}(n \log n)$: las firmas suman $\mathcal{O}(n)$ y cada consulta al diccionario cuesta $\log n$ comparaciones.
- **Clave: ordenar** los ids. Sin ordenar, subárboles isomorfos reciben ids distintos y se sobrecontarían las clases.
- Comparar nodo a nodo es $\mathcal{O}(n^2 \cdot \text{tamaño})$; un hash de los grados en preorden no es invariante bajo isomorfismo.

Outline

Aplicación 5: strings mutables

Por qué hashing gana

Es el caso donde *incluso sobre strings* hashing es lo único viable: Suffix Array/Automaton **no soportan updates**. Resuelve el segundo problema de la introducción.

- El hash es un **monoide** (§2) $\rightarrow$ se enchufa a un `SegTree<UnitHash>` genérico.
- `Query(i,j)` = concatenación en el rango (hash del substring, **sin inverso**).
- `Update(i,c)` = point-update de un carácter en $\mathcal{O}(\log n)$.

La clave: usar la concatenación (**no** conmutativa); sumar haría colisionar `"ab"` y `"ba"`.

Segment Tree de hashes

El neutro y la operación son los del monoide

```
struct StringEq {
    SegTree<UnitHash> st;
    explicit StringEq(const string& s) : st(build(s)) {}
    static SegTree<UnitHash> build(const string& s) {
        vector<UnitHash> hojas;
        for (char c : s) hojas.emplace_back(string(1, c));
        return SegTree<UnitHash>(hojas, UnitHash(),
            [](UnitHash a, UnitHash b) { return a * b; });
    }
    void Update(int i, char c) {
        st.update(i, UnitHash(string(1, c)));
    }
    bool Query(int i, int j, int l) {// S[i..i+l)==S[j..j+l)
        return st.query(i, i + l) == st.query(j, j + l);
    }
};
```

Aplicación (para pensar): palíndromos con updates

Se procesan operaciones sobre un string: (1) cambiar $s[i] = c$; (2) dado (l, r), decidir si $s[l..r]$ es **palíndromo**. Con $n, q \leq 2 \cdot 10^5$. ¿Cuál es el enfoque?

Aplicación (para pensar): palíndromos con updates

Se procesan operaciones sobre un string: (1) cambiar $s[i] = c$; (2) dado (l, r), decidir si $s[l..r]$ es **palíndromo**. Con $n, q \leq 2 \cdot 10^5$. ¿Cuál es el enfoque?

Solución

Dos **Segment Trees de hashes**: uno del string directo y otro del reverso. $s[l..r]$ es palíndromo sii el hash directo del rango coincide con el del rango espejado en el reverso. Consulta y update, $\mathcal{O}(\log n)$.

Trampas:

- **Manacher** no soporta updates: recomputar es $\mathcal{O}(n)$.
- Combinar con **suma** es simétrico $\rightarrow$ diría "palíndromo" siempre.
- Comparar carácter por carácter es $\mathcal{O}(\text{longitud})$ por consulta.

Outline

Cierre: lo que se llevan

Señales de reconocimiento

- **Segmentos/potencias de string** $\rightarrow$ monoide de hash (+ LCP, + binpow).
- **Conjuntos/multiconjuntos** (orden no importa) $\rightarrow$ XOR / suma de aleatorios.
- **Comparar módulo rotación** $\rightarrow$ traza de un producto de matrices aleatorias.
- **Estados que cambian de a poco** $\rightarrow$ Zobrist incremental.
- **Formas de (sub)árbol** $\rightarrow$ id canónico con hijos ordenados / centroide.
- **Substrings con updates** $\rightarrow$ Segment Tree de `UnitHash`.

Y siempre: **hacerlo bien** ($2^{61}-1$ o doble hash, base aleatoria) porque comparar de a muchos vive en $k^2/2M$.