Complejidad y estructuras de datos

Alex Blanchard

alex.blanchard@progcomp.cl

Requisitos

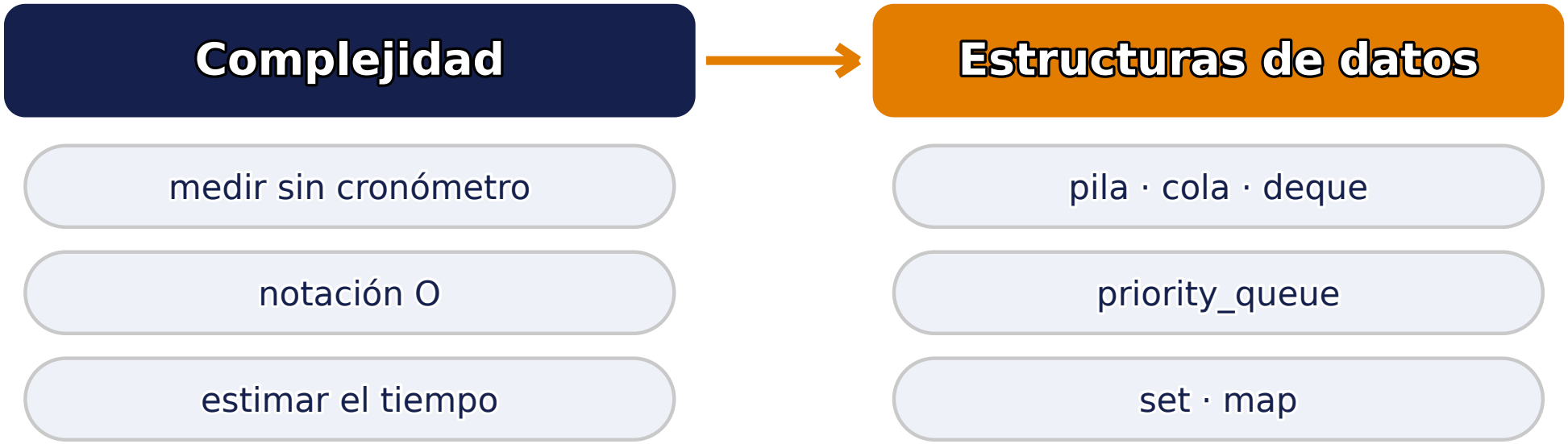
Se asume familiaridad básica con:

- **C++ básico**: variables, if/else, for y while.
- **Funciones** y el contenedor vector.
- Nada más: la complejidad y las estructuras las vemos **hoy desde cero**.

Si sabes escribir un **for** que recorre un vector, estás listo.

No necesitas conocer ningún algoritmo de antemano: todo lo demás lo construimos paso a paso.

Mapa de la clase



Primero aprendemos a **medir qué tan rápido** es un algoritmo, luego **herramientas listas para usar**.

Complejidad en programación competitiva

En programación competitiva hay **límites de tiempo estrictos**.

Un programa que exceda el límite recibe el veredicto **TLE** (Time Limit Exceeded).

Necesitamos estimar qué tan rápido correrá nuestro código **antes de enviarlo**.

¿Cuánto se demora mi programa?

Medir con **cronómetro** depende del entorno: hardware, lenguaje, compilador.

Buscamos un **modelo teórico** que anticipe el rendimiento.

Hoy aprenderemos a **estimar ese tiempo** de forma práctica.

Cronómetro

depende del entorno

hardware · lenguaje · compilador

vs

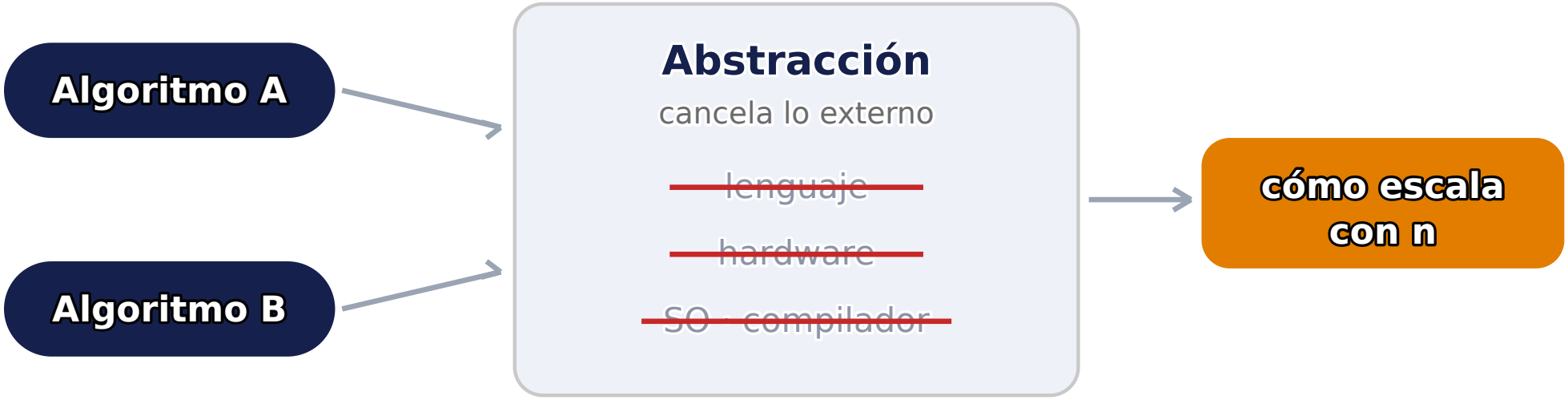
Modelo teórico

independiente del entorno

predice el rendimiento

¿Cuál algoritmo es más rápido?

Supongamos dos algoritmos que resuelven el **mismo problema**. ¿Cuál elijo?



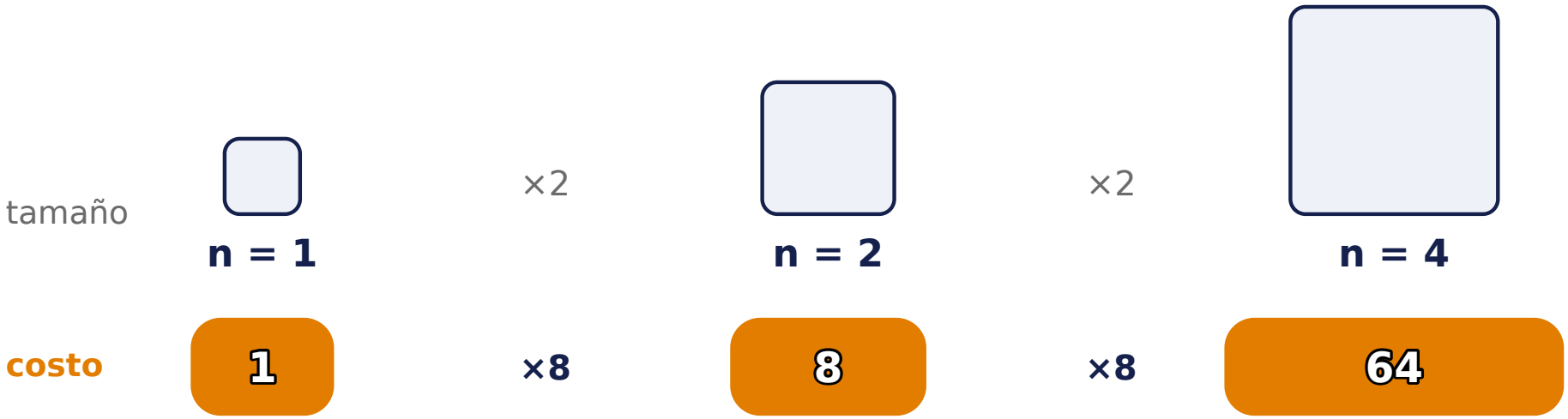
Factores externos que estorban la comparación: **lenguaje**, **potencia del computador**, **sistema operativo** y **compilador**.

Necesitamos una herramienta que **abstraiga** esos factores y deje solo cómo escala con n.

Estimar sin fórmula exacta

Analogía: para **estimar una masa** no siempre necesitas la fórmula exacta.

Basta saber **a qué ritmo crece** para predecir su valor cuando el tamaño aumenta.

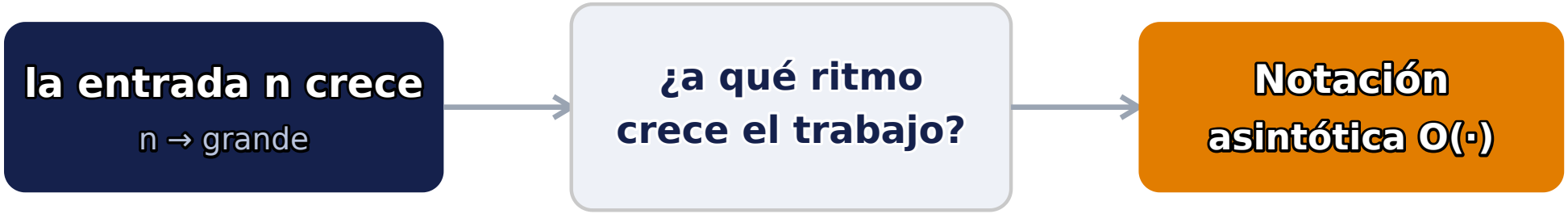


conociendo el ritmo, predecimos el costo futuro

Conocer la velocidad a la que escala una cantidad permite predecir su valor, aunque no tengamos la fórmula exacta.

¿Qué buscamos entonces?

Queremos decir **a qué ritmo crece el trabajo** de un algoritmo cuando la entrada n crece.



Esa forma es la **notación asintótica** (Big-O), que vemos ahora.

Una métrica independiente

Queremos una función $T(n)$ que mida el **trabajo** de un algoritmo según el tamaño de la entrada n.

Para comparar dos algoritmos, comparamos $T_1(n)$ contra $T_2(n)$.

Algoritmo 1
$T_1(n)$

vs

Algoritmo 2
$T_2(n)$

calcular T(n) exacto: casi imposible

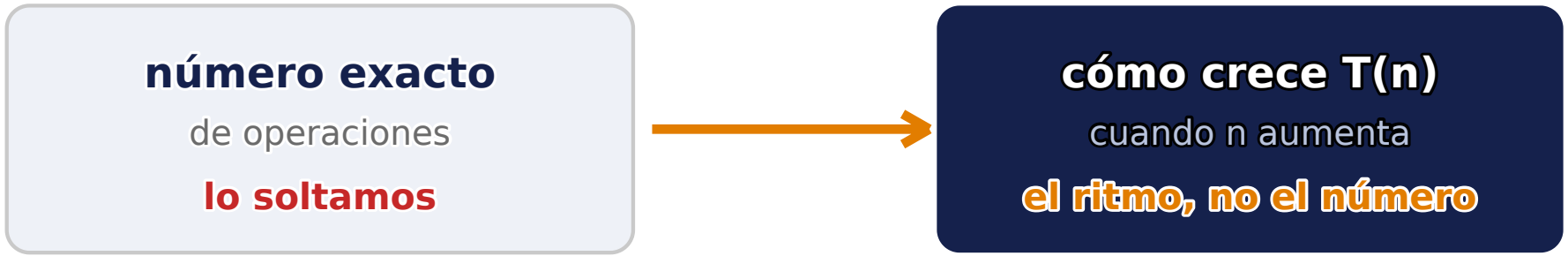
depende de constantes ocultas y del hardware

El **valor exacto** de $T(n)$ es casi imposible de conocer: cada máquina y cada detalle cambia el número.

La solución: mirar el crecimiento

Renunciamos a la **constante exacta**.

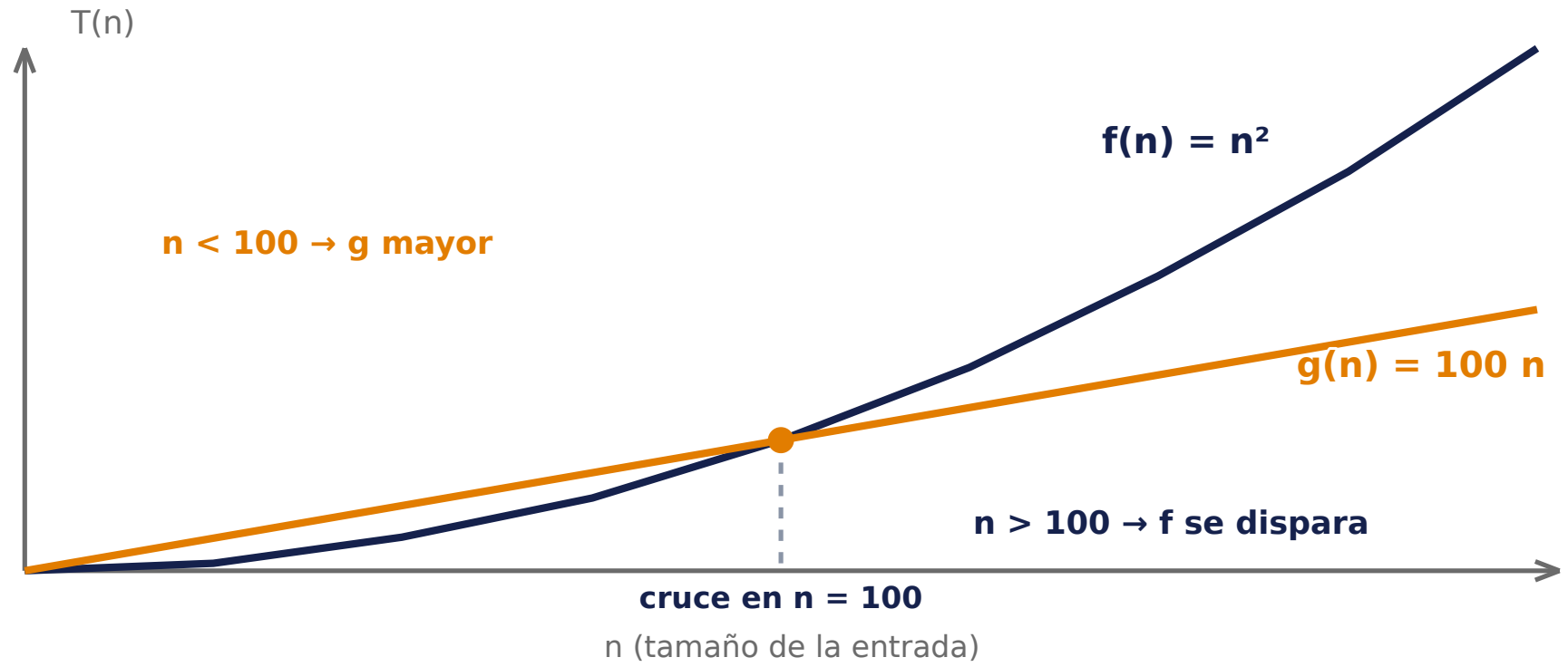
Nos fijamos en **cómo crece** $T(n)$ cuando n aumenta.



Frase clave: nos importa el **ritmo de crecimiento**, no el número exacto.

Crecimiento de funciones

Comparemos $f(n) = n^2$ (azul) contra $g(n) = 100\,n$ (naranjo).

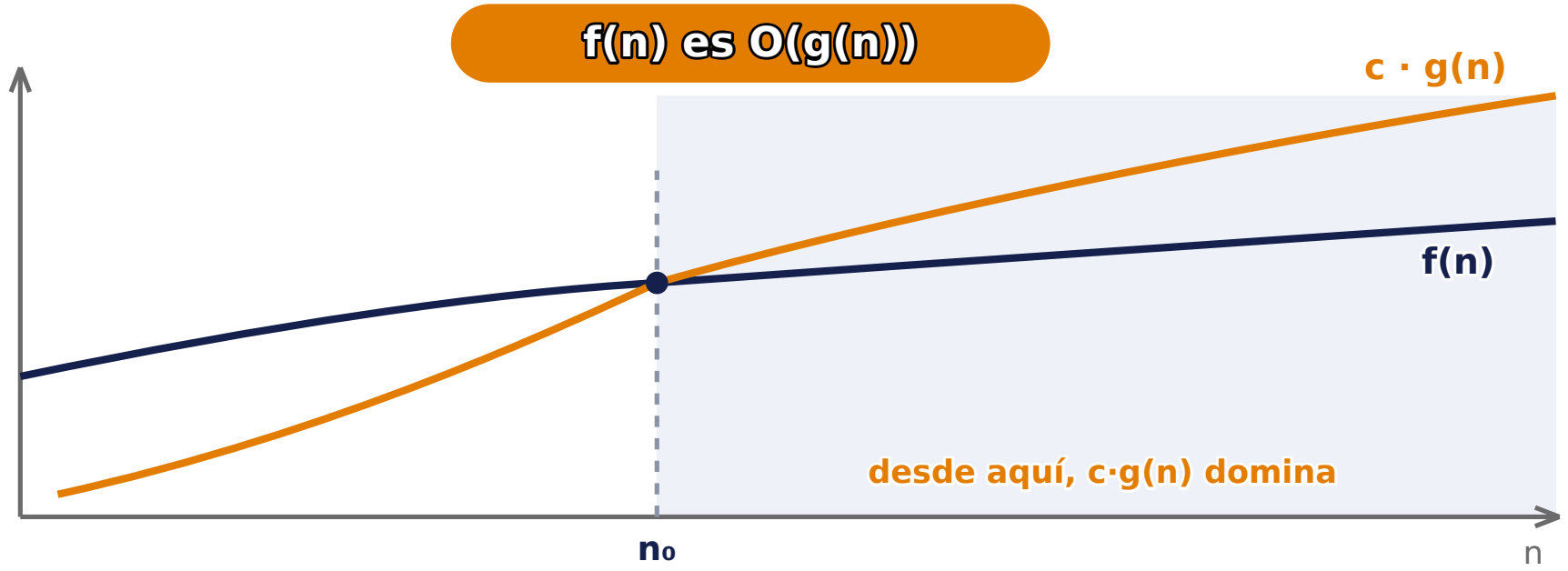


Para n chico, g es mayor. Pero desde $n > 100$, $f(n)$ **domina** y la diferencia se dispara.

Notación O (Big-O)

$O(f)$ describe cómo crece $T(n)$ **ignorando** constantes y detalles de hardware.

Idea: $f(n)$ es $O(g(n))$ si a partir de cierto n_0, la curva $c \cdot g(n)$ (con una constante $c > 0$) siempre queda **por encima** de $f(n)$.



No nos preocupamos por el valor exacto de c ni de n_0: basta con que **existan**.

Ignoramos las constantes

El área de un cuadrado y la de un círculo crecen al **mismo ritmo**: proporcional al **cuadrado** del tamaño.

Cuadrado

lado L

área = L²

Círculo

radio r

área = π · r²

el π es una constante: se ignora

El área del cuadrado es $O(L^2)$. El área del círculo es πr^2, y también es $O(r^2)$.

Moraleja: en Big-O botamos las constantes multiplicativas y los términos de menor orden.

Dos reglas prácticas

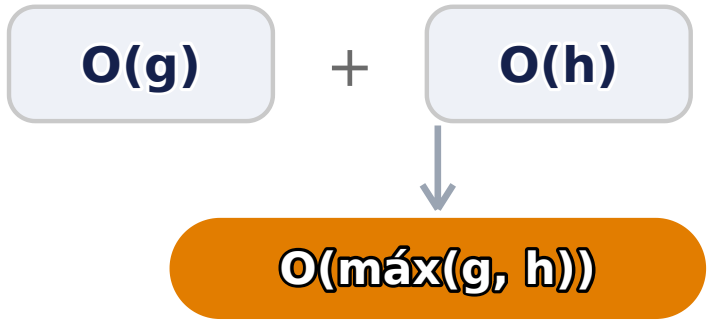
Regla de la suma · dos partes en secuencia: nos quedamos con la más grande.

$$f_1 \in O(g),\ f_2 \in O(h) \ \Rightarrow\ f_1 + f_2 \in O(\max(g, h))$$

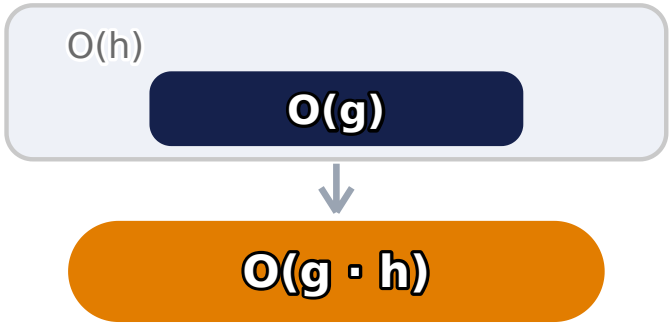
Regla del producto · algo $O(g)$ anidado dentro de algo $O(h)$: se multiplican.

$$f_1 \in O(g),\ f_2 \in O(h) \ \Rightarrow\ f_1 \cdot f_2 \in O(g \cdot h)$$

Suma → gana la mayor



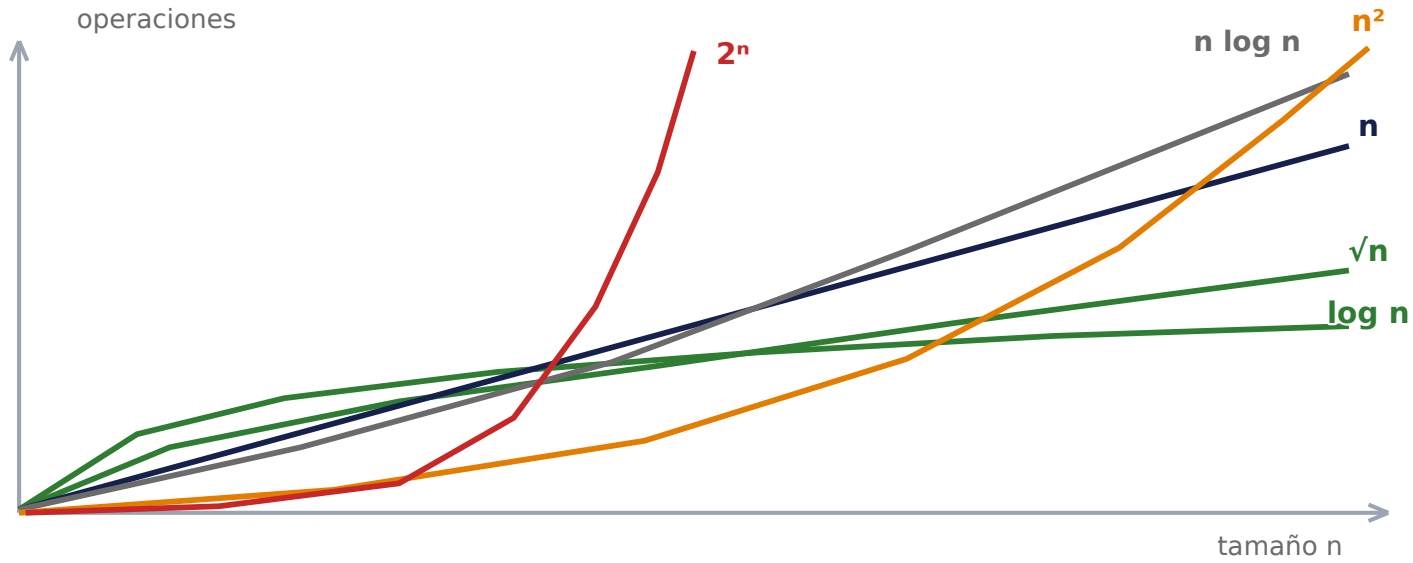
Producto → se anidan



Jerarquía de complejidades

De más rápida a más lenta:

$$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

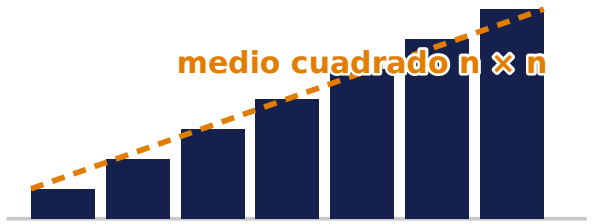


Curvas planas escalan bien; las **empinadas explotan** al crecer n.

Formulario útil

Suma de los primeros n enteros:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$



cada barra i aporta i · el total llena la mitad → O(n²)

Cambio de base de logaritmos:

$$\log_b a = \frac{\log_c a}{\log_c b}$$

Como cambiar de base solo multiplica por una constante, en Big-O todos los logaritmos son equivalentes: escribimos $O(\log n)$ sin base.

¿Cómo la leo en el código?

Una **operación elemental** cuesta $O(1)$: una operación aritmética o lógica, comparar, asignar, imprimir un dato.

Contamos **cuántas operaciones elementales** se ejecutan en función de n y nos quedamos con el **orden de crecimiento**.

Un bucle de n pasos	Bucles anidados	No depende de n
multiplica por n	**se multiplican**	**queda en O(1)**

Regla mental: cuenta los bucles y de qué depende cada uno.

Ejemplo 1: un bucle

```
int suma = 0;
for (int i = 0; i < n; i++)
    suma += 5;
```

El cuerpo del bucle se ejecuta n veces, y cada vez hace una operación elemental.

Respuesta: $O(n)$.

Ejemplo 2: bucles anidados

```
int suma = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        suma += 2;
```

Por cada uno de los n pasos del bucle externo, el interno hace n pasos: $n \cdot n = n^2$.

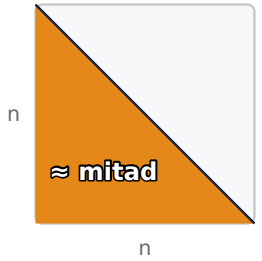
Respuesta: $O(n^2)$.

Ejemplo 3: triángulo

```
int suma = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < i; j++)
        suma += 1;
```

El interno hace $0, 1, 2, \ldots, n-1$ pasos:

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$



celdas pintadas = n(n-1)/2

casi la mitad del cuadrado → **O(n²)**

Respuesta: $O(n^2)$.

Ejemplo 4: dos variables

```
int suma = 0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        suma += n;
```

El externo hace n pasos, el interno hace m pasos. *No asumas que* $m = n$.

Respuesta: $O(n \cdot m)$.

Ejemplo 5: condición con i*i

```
int suma = 0;
for (int i = 0; (long long)i*i < n; i++)
    suma += 1;
```

El bucle corre mientras $i^2 < n$, es decir mientras $i < \sqrt{n}$.

Respuesta: $O(\sqrt{n})$.

Ejemplo 6: se divide a la mitad

```
for (int i = n; i > 0; i /= 2)
    cout << "hola" << endl;
```

i se divide entre 2 cada vez: $n \to n/2 \to n/4 \to \cdots \to 1$, en total $\log_2 n$ pasos.

Respuesta: $O(\log n)$.

Ejemplo 7: cuidado con la condición

```
for (int i = 0; i < (n % 10); i++)
    cout << "cuidado" << endl;
```

$n\%10$ es a lo más 9: el bucle da **como mucho 9 pasos**, sin importar cuán grande sea n.

Respuesta: $O(1)$.

Contar primos: primera versión

Contamos cuántos primos hay entre 2 y n.

```
bool es_primo(int x) {
    for (int i = 2; i < x; i++)
        if (x % i == 0) return false;
    return true;
}
int main() {
    int n; cin >> n;
    int contador = 0;
    for (int i = 2; i <= n; i++)
        if (es_primo(i)) contador++;
    cout << contador << endl;
}
```

- El bucle externo recorre ~n valores y `es_primo(x)` prueba divisores hasta x.
- Costo total: $\sum_{i=2}^{n} i = \frac{n(n+1)}{2} \Rightarrow O(n^2)$.

Optimización: saltar los pares

El único par primo es el 2: los demás pares no sirven, los saltamos.

```
int main() {
    int n; cin >> n;
    int contador = 1; // cuenta el 2
    for (int i = 3; i <= n; i += 2)
        if (es_primo(i)) contador++;
    cout << contador << endl;
}
```

La función `es_primo` es la misma de la versión anterior.

- El bucle externo da solo $n/2$ pasos.
- Pero `es_primo` **sigue costando hasta** n por número.
- La constante $1/2$ no cambia el orden de crecimiento.
- $O(n^2)$ *(más rápido en la práctica, misma clase)*

Optimización: probar hasta √x

Si x tiene un divisor $d > \sqrt{x}$, entonces $x/d < \sqrt{x}$ también lo divide.

$$d \cdot \frac{x}{d} = x \quad \Rightarrow \quad \text{basta llegar a } \lfloor \sqrt{x} \rfloor$$

```
bool es_primo(int x) {
    for (int i = 2; (long long)i*i <= x; i++)
        if (x % i == 0) return false;
    return x >= 2;
}
```

- Ahora `es_primo(x)` cuesta $O(\sqrt{x})$.
- Total: $\sum_{i=2}^{n} \sqrt{i} = O(n\sqrt{n})$.
- $O(n\sqrt{n})$

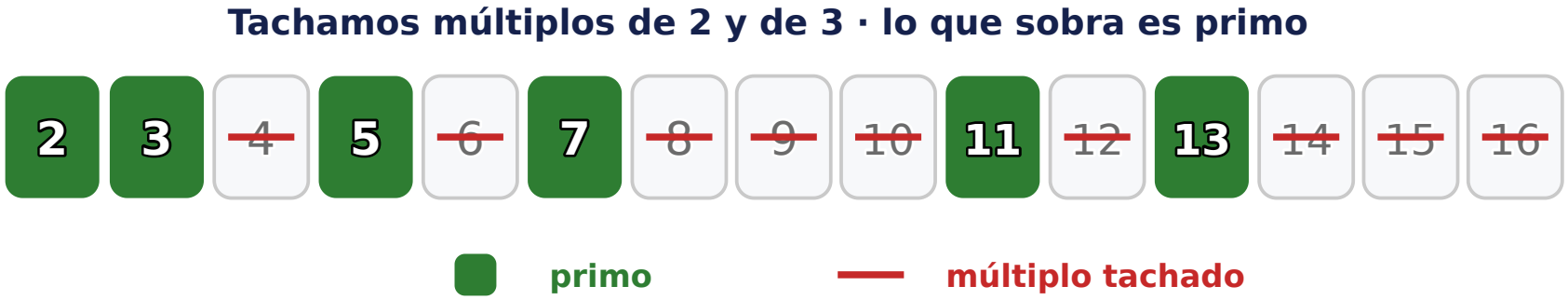
Criba de Eratóstenes

En vez de preguntar por cada número, **tachamos los múltiplos**.

```
int main() {
    int n; cin >> n;
    vector<bool> es_primo(n + 1, true);
    es_primo[0] = es_primo[1] = false;
    for (int i = 2; i <= n; i++)
        if (es_primo[i])
            for (int j = 2*i; j <= n; j += i)
                es_primo[j] = false;
    int contador = 0;
    for (int i = 2; i <= n; i++)
        if (es_primo[i]) contador++;
    cout << contador << endl;
}
```

Cada número se tacha desde sus múltiplos: no se prueba uno por uno.

Criba: por qué es casi lineal



- El bucle interno solo corre para primos p, tachando n/p números.
- Suma sobre primos: $\sum_{p \le n} \frac{n}{p} = O(n \log \log n)$.
- $O(n \log \log n)$, casi lineal.

De la complejidad al tiempo

Con la complejidad $O(f(n))$ estimamos cuánto **demora** el programa.

Regla de dedo en competitiva: una máquina hace $\approx 10^8$ operaciones por segundo.

$$\text{tiempo (s)} \approx \frac{f(n)}{10^8}$$

Dos ejemplos

mismo n = 10^6

$f(n) = n^2$

$= 10^{12}$ operaciones

$\approx 10^4$ s ($\approx$ 2.8 horas)

TLE seguro

$f(n) = n \cdot \log_2 n$

$\approx 2 \cdot 10^7$ operaciones

$\approx$ 0.2 s

cómodo

Moraleja: el mismo n pasa o no según la complejidad.

Recuerda: $\log_2(10^6) \approx 20$, por eso $n \log_2 n \approx 10^6 \cdot 20 = 2 \times 10^7$.

¿Qué complejidad me pide el límite de n ?

Guía rápida: el mayor n que aguanta cada complejidad en ~1 segundo.

Tamaño de n	Complejidad (≈ 1 s)
n ≤ 11	O(n!)
n ≤ 25	O(2ⁿ)
n ≤ 500	O(n³)
n ≤ 5 000	O(n²)
n ≤ 10⁶	O(n log n)
n ≤ 10⁸	O(n)

Es una guía aproximada: la constante oculta y la constante del lenguaje pueden mover el límite.

Estructuras de datos básicas

Pilas, colas, deque y colas de prioridad

¿Por qué son importantes?

- Permiten resolver problemas de forma **rápida y eficiente**.
- Ayudan a **organizar y manejar** la información con operaciones ya optimizadas.
- Vienen **listas en la STL** de C++.

Rápidas y eficientes

resuelven problemas

Organizan datos

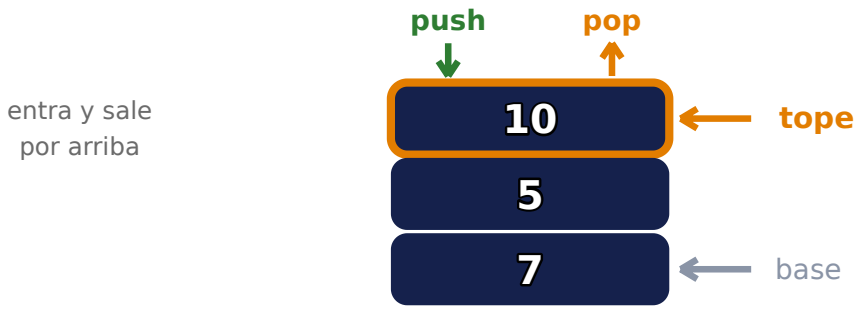
operaciones ya optimizadas

Ya en la STL

vienen listas en C++

Pila (stack)

- **LIFO**: el último en entrar es el primero en salir.
- `push` agrega arriba · `pop` quita arriba · `top` mira el tope · `empty`.



LIFO · el último en entrar sale primero

```
stack<int> pila;
pila.push(7);
pila.push(5);
pila.push(10);
cout << pila.top() << endl; // 10
pila.pop();
cout << pila.top() << endl; // 5
```

Cola (queue)

- **FIFO**: el primero en entrar es el primero en salir.
- `push` atrás · `pop` adelante · `front` · `back` · `empty`.

FIFO · el primero en entrar sale primero

```
queue<int> cola;
cola.push(1);
cola.push(2);
cout << cola.front() << endl; // 1
cola.pop();
cout << cola.front() << endl; // 2
```

Doble cola (deque)

- Inserta y elimina por **ambos extremos** en $O(1)$; combina pila y cola.
- Además permite acceso por índice: `dq[i]`.
- `push_back` · `push_front` · `pop_back` · `pop_front`.

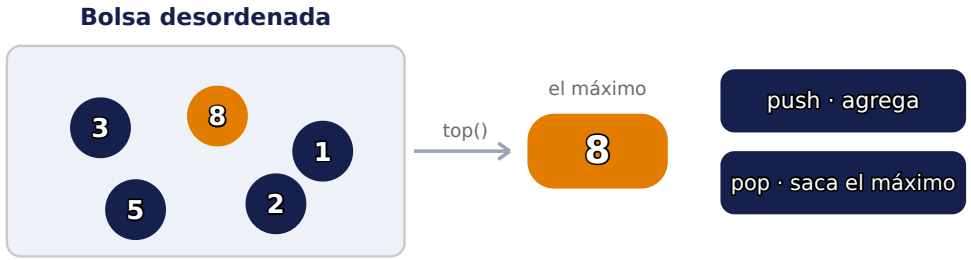
ambos extremos en O(1) · combina pila y cola

```
deque<int> dq;
dq.push_back(1);
dq.push_front(2);
cout << dq.front() << endl; // 2
cout << dq.back() << endl;  // 1
```

Cola de prioridad (priority_queue)

Como una cola, pero **no respeta el orden de llegada**: siempre entrega el **mayor** (o el menor, si la configuras así).

Operaciones **push**, **pop**, **top**, **empty**, cada una en $O(\log n)$.




```cpp
priority_queue<int> pq;                    // máximo arriba (por defecto)
pq.push(3); pq.push(8); pq.push(1);
cout << pq.top() << endl; // 8
pq.pop();
cout << pq.top() << endl; // 3
priority_queue<int, vector<int>, greater<int>> pqmin; // mínimo arriba
```

Ejercicio: paréntesis balanceados

Te dan una cadena con solo `(` y `)`.

Determina si están **correctamente balanceados**.

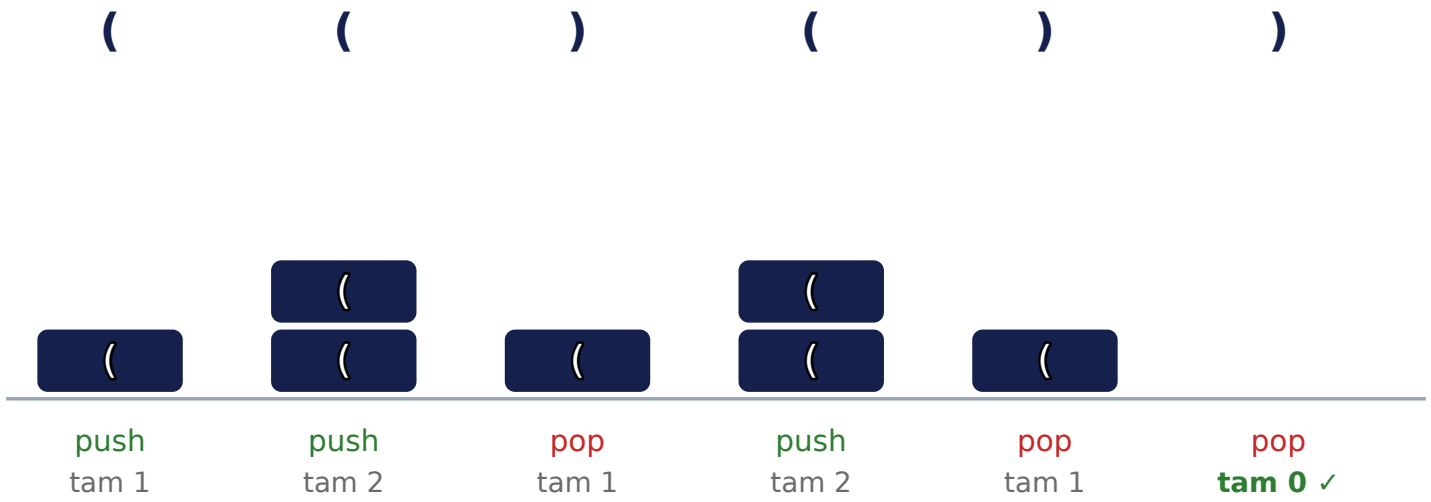
Cada `(` debe cerrarse con un `)` posterior, en el orden correcto.

cadena	¿balanceada?
`(()())`	YES
`)()(`	NO

Paréntesis: la idea

Recorremos la cadena con una **pila**: cada `(` hace **push**, cada `)` hace **pop**.

Un `)` con la pila **vacía** → **NO**. Si al final quedan `(` sin cerrar → **NO**. Si no → **YES**.



Termina **vacía** → la cadena está balanceada → **YES**.

Paréntesis: código

```
string s; cin >> s;
stack<char> st;
bool ok = true;
for (char c : s) {
    if (c == '(') st.push(c);
    else {
        if (st.empty()) { ok = false; break; }
        st.pop();
    }
}
if (!st.empty()) ok = false;
cout << (ok ? "YES" : "NO") << endl;
```

`st.empty()` al ver un `)` → sobra un cierre → **NO**.

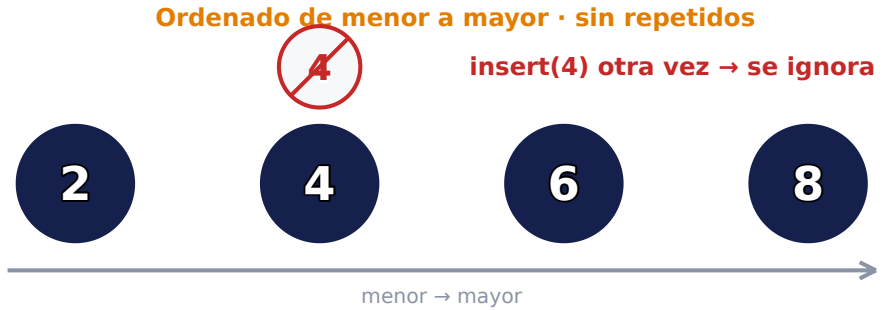
`!st.empty()` al final → quedaron `(` sin cerrar → **NO**.

En cualquier otro caso → **YES**.

Conjunto (set)

Colección **ordenada** y **sin repetidos**.

- Todo en $O(\log n)$: `insert`, `erase`, `count` (0/1), `find`.
- `lower_bound(x)`: primer elemento $\geq x$.
- `upper_bound(x)`: primer elemento $> x$.
- ¿Quieres permitir repetidos? Usa **multiset**.

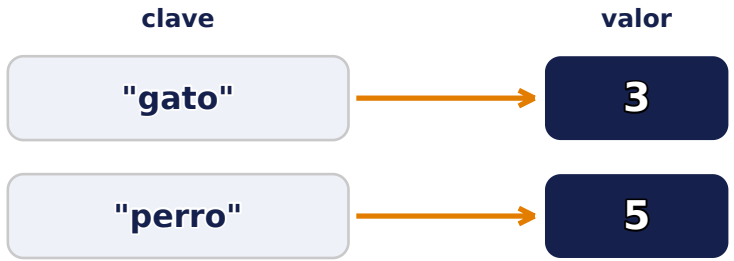



```cpp
set<int> s;
s.insert(4); s.insert(2); s.insert(4); // el 4 no se repite
if (s.count(2)) cout << "esta el 2" << endl;
s.erase(2);
if (!s.count(2)) cout << "el 2 ya no esta" << endl;
```

Diccionario (map)

Asocia **clave** → **valor**, con las **claves ordenadas**.

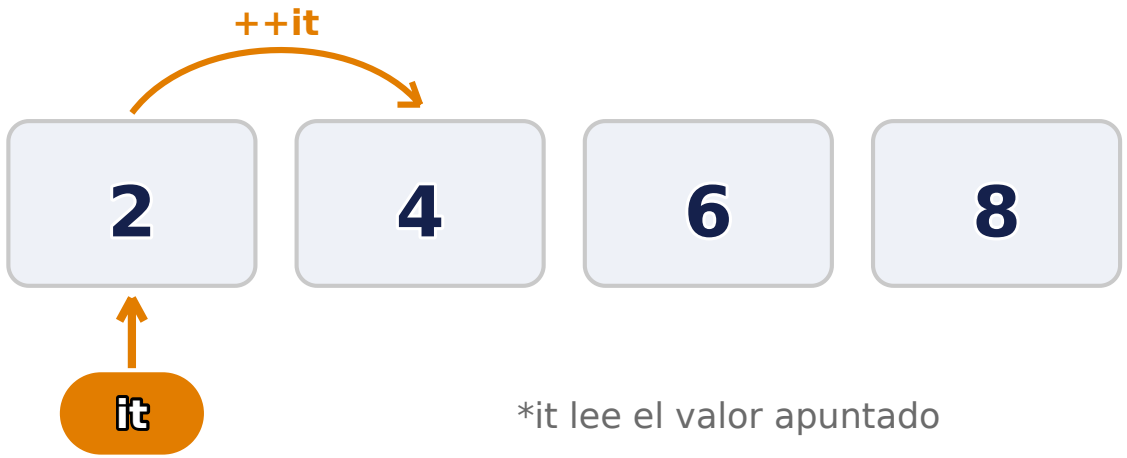
- `insert`, `erase`, `find`, `lower_bound` / `upper_bound` por clave, en $O(\log n)$.
- Operador `[]`: accede y **crea la clave si no existe**.
- Uso típico: **contar frecuencias**, asociar datos a una clave.




```cpp
map<string,int> mp;
mp["gato"] = 3;
mp["perro"] = 5;
cout << mp["gato"] << endl; // 3
mp.erase("gato");
if (!mp.count("gato")) cout << "gato ya no esta" << endl;
```

Un iterador es como un **puntero** a un elemento del contenedor.

- Avanzar `++it`, retroceder `--it`, leer `*it`.
- `find` devuelve un iterador al elemento, o `end()` si no está.
- Se compara con `s.end()` para saber si el elemento existe.



Iteradores: ejemplos

Recorrer un **set** con un iterador:

```
set<int> s = {2, 4, 6};
auto it = s.begin();   // apunta al 2
cout << *it << endl;   // 2
++it;                  // ahora al 4
cout << *it << endl;   // 4
```

Recorrer un **map** completo (`it->first` clave, `it->second` valor):

```
map<string,int> m = {{"a",1}, {"b",2}};
for (auto it = m.begin(); it != m.end(); ++it)
    cout << it->first << " -> " << it->second << endl;
```

Ejercicio: ¿pasan todos los niveles?

Un juego tiene n niveles. **Dos jugadores** pueden pasar ciertos niveles.

Colaborando, ¿logran pasar **todos** los niveles del 1 al n?

- Entrada: n; luego los niveles del **jugador 1** y los del **jugador 2**.
- Salida: `I become the guy.` si cubren todos, o `Oh, my keyboard!` si falta alguno.

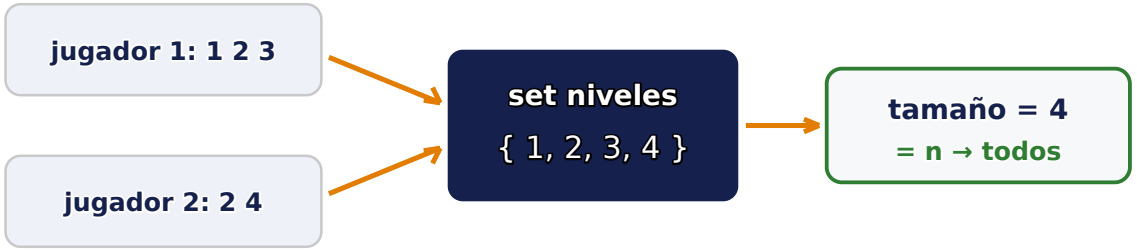
n	**Jugador 1**	**Jugador 2**	**Salida**
4	1 2 3	2 4	I become the guy.
4	1 2 3	2 3	Oh, my keyboard!

1 **2** **3** **4**

Niveles: solución

Metemos **todos** los niveles que alguien puede pasar en un **set** (así no contamos repetidos).

Si el tamaño del set es n, entonces **están todos**.



tamaño del set = n significa que no falta ningún nivel

```cpp
int n; cin >> n;
set<int> niveles;
int p; cin >> p;
for (int i = 0; i < p; i++) { int x; cin >> x; niveles.insert(x); }
int q; cin >> q;
for (int i = 0; i < q; i++) { int x; cin >> x; niveles.insert(x); }
if ((int)niveles.size() == n) cout << "I become the guy." << endl;
else cout << "Oh, my keyboard!" << endl;
```

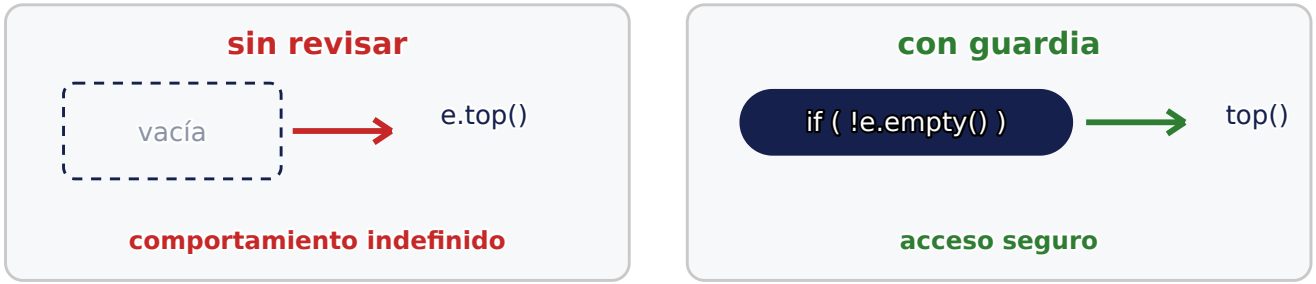
Resumen de complejidades

Estructura	Inserción	Borrado	Búsqueda	Acceso al extremo
stack · queue · deque	$O(1)$	$O(1)$	—	$O(1)$
priority_queue	$O(\log n)$	$O(\log n)$	—	$O(1)$
set · map	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

extremos · **O(1)**

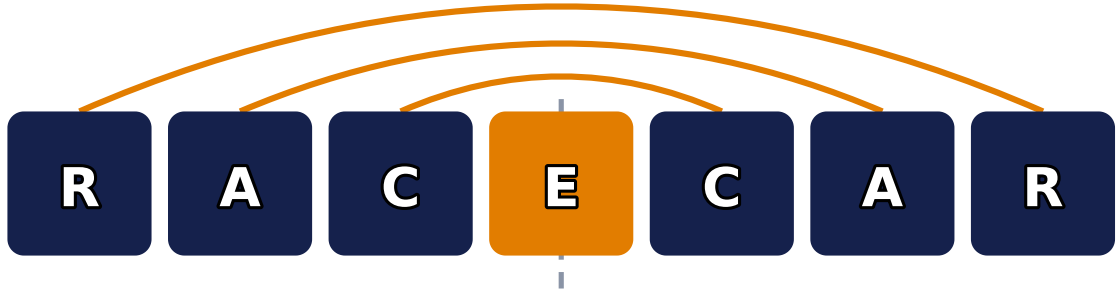
ordenados por clave · **O(log n)**

Consejos para competir



- Antes de `top()`, `front()` o `pop()`: revisa `!e.empty()`. Tocar una estructura **vacía** es comportamiento indefinido.
- `unordered_map` y `unordered_set` dan $O(1)$ **en promedio**, pero $O(n)$ en el peor caso por colisiones. Útiles con entradas **no adversarias**.
- Con `pair`, `tuple` o `struct` como clave en un contenedor *unordered*, define un **hash propio**.
- `find` y `lower_bound` rinden más cuando cada elemento **guarda más de un dato**, por ejemplo un `pair<int,int>`.

Ejercicio final: Palindrome Reorder



Reordena las letras de un string (mayúsculas **A–Z**) para formar un **palíndromo**.
Si hay varias soluciones, **cualquiera** sirve. Si es imposible, imprime `NO SOLUTION`.

Pista suave: piensa en **cuántas letras** pueden tener frecuencia **impar**.

Enlace: https://cses.fi/problemset/task/1755

¿Dudas?

Gracias por acompañarnos en el track. Ahora a practicar: nos vemos en el contest.

¡Mucho éxito!