

Estructuras de Datos

Análisis asintótico y medición

O, Ω, Θ, leer código, y medir sin engañarse

Gerson Garrido

Semana 3

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Por qué contar operaciones y no segundos
2. Las clases que se repiten
3. O, Ω y Θ
4. Mejor caso, peor caso y caso promedio
5. Leer código: ciclos y recursión
6. Espacio además de tiempo
7. Medir sin engañarse
8. El experimento de duplicar
9. Lo que O no ve

> **Lectura**
>
> Apuntes de la semana 3. Código en `code/week-03`.

SECCIÓN 01

Contar, no cronometrar

01

¿Cuánto tarda este programa?

terminal, depende de la máquina

```
$ build/first_timer
una medicion: 359 us
```

Sumar un millón de enteros: unos 350 microsegundos. ¿Y en su laptop? Otro número.

¿Cuánto tarda este programa?

terminal, depende de la máquina

```
$ build/first_timer
una medicion: 359 us
```

Sumar un millón de enteros: unos 350 microsegundos. ¿Y en su laptop? Otro número.

- Depende del procesador, del compilador y de qué más está corriendo.

¿Cuánto tarda este programa?

terminal, depende de la máquina

```
$ build/first_timer
una medicion: 359 us
```

Sumar un millón de enteros: unos 350 microsegundos. ¿Y en su laptop? Otro número.

- Depende del procesador, del compilador y de qué más está corriendo.
- Un número en segundos describe una corrida, no un algoritmo.

¿Cuánto tarda este programa?

terminal, depende de la máquina

```
$ build/first_timer
una medicion: 359 us
```

Sumar un millón de enteros: unos 350 microsegundos. ¿Y en su laptop? Otro número.

- Depende del procesador, del compilador y de qué más está corriendo.
- Un número en segundos describe una corrida, no un algoritmo.
- Una vuelta por elemento da un millón de vueltas en cualquier computadora.

> $T(n)$
>
> Operaciones elementales con una entrada de tamaño n. Elemental: de costo acotado por una constante.

Lo que importa cuando n crece

$T(n) = 3n^2 + 10n$. ¿Cuánto aporta cada término?

n	$3n^2$	$10n$	parte de $3n^2$ en el total
10	300	100	75 %
100	30 000	1 000	96.8 %
1 000	3 000 000	10 000	99.7 %
1 000 000	3×10^{12}	10^7	99.9997 %

Análisis asintótico ***(asymptotic analysis)***

Cómo crece el costo cuando n crece, sin términos de menor orden ni factores constantes.

SECCIÓN 02

Las clases que se repiten

02



Seis clases

Clase	Dónde aparece
1	leer `values[i]`
$\log n$	búsqueda binaria
n	recorrer una vez
$n \log n$	ordenar bien
n^2	comparar cada par
2^n	todos los subconjuntos

log es $\log_2$: cuántas veces se divide n entre 2 hasta llegar a 1.

¿Qué *n* cabe en un segundo?

terminal

```
$ build/growth_table
con 1000000000 operaciones
          n     1000000000
  n log2 n       39620077
       n^2          31622
       n^3           1000
       2^n             29
```

- Supone 10^9 operaciones por segundo: una suposición redonda, no una medida.

¿Qué n cabe en un segundo?

terminal

```
$ build/growth_table
con 1000000000 operaciones
          n     1000000000
  n log2 n       39620077
       n^2          31622
       n^3           1000
       2^n             29
```

- Supone 10^9 operaciones por segundo: una suposición redonda, no una medida.
- Un millón de datos con n^2: 10^{12} operaciones, unos 17 minutos. Esa cuenta se hace antes de programar.

¿Qué n cabe en un segundo?

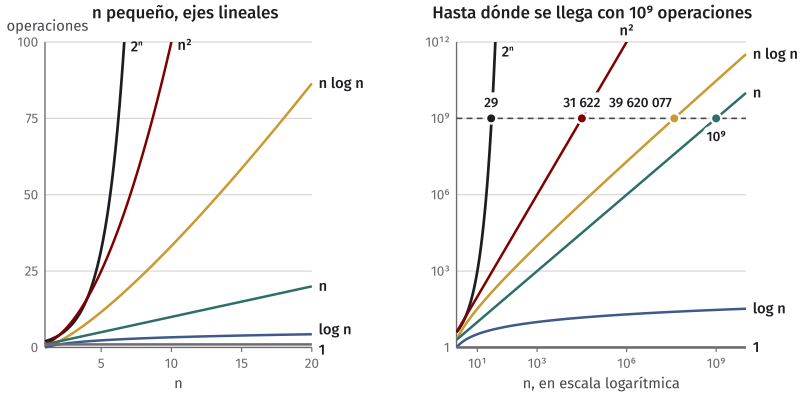


- Supone 10^9 operaciones por segundo: una suposición redonda, no una medida.
- Un millón de datos con n^2: 10^{12} operaciones, unos 17 minutos. Esa cuenta se hace antes de programar.
- Con una máquina mil veces más rápida: n^2 pasa de 31 622 a un millón. 2^n pasa de 29 a 39.

Idea clave

Un algoritmo mejor rinde más que una máquina mejor.

Las seis clases, dibujadas



Cada punto: el mayor n que alcanza un presupuesto de 10^9 operaciones

SECCIÓN 03

O, Ω y Θ

03

Las definiciones

f y g van de los naturales a los reales no negativos.

Cota superior: $O(g)$

$f \in O(g)$ si existen $c > 0$ y n_0 tales que $f(n) \leq c \cdot g(n)$ para todo $n \geq n_0$.

Cota inferior: $\Omega(g)$

$f \in \Omega(g)$ si existen $c > 0$ y n_0 tales que $f(n) \geq c \cdot g(n)$ para todo $n \geq n_0$.

Cota exacta: $\Theta(g)$

$f \in \Theta(g)$ si existen $c_1, c_2 > 0$ y n_0 tales que $c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0$.

Qué significa cada parte

$f \in \Theta(g)$ si y solo si $f \in O(g)$ y $f \in \Omega(g)$.

- c ignora los factores constantes: $300n^2$ y $3n^2$ quedan en la misma clase.

Qué significa cada parte

$f \in \Theta(g)$ si y solo si $f \in O(g)$ y $f \in \Omega(g)$.

- c ignora los factores constantes: $300n^2$ y $3n^2$ quedan en la misma clase.
- n_0 ignora los n pequeños: la desigualdad puede fallar al inicio.

Qué significa cada parte

$f \in \Theta(g)$ si y solo si $f \in O(g)$ y $f \in \Omega(g)$.

- c ignora los factores constantes: $300n^2$ y $3n^2$ quedan en la misma clase.
- n_0 ignora los n pequeños: la desigualdad puede fallar al inicio.
- **El orden importa:** primero se eligen c y n_0, una vez, y sirven para todos los $n \geq n_0$. La constante no puede depender de n.

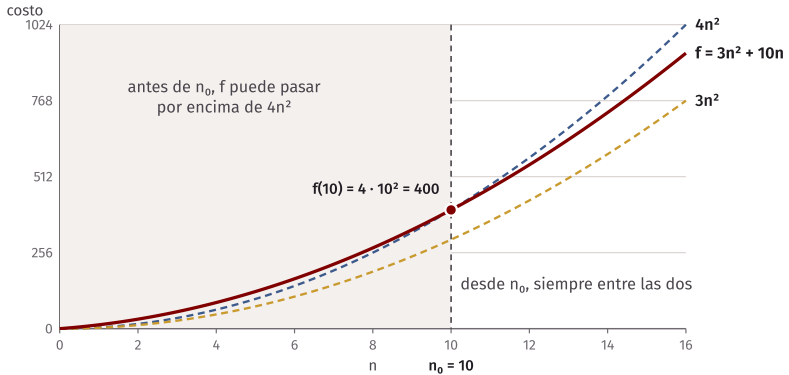
Qué significa cada parte

$f \in \Theta(g)$ si y solo si $f \in O(g)$ y $f \in \Omega(g)$.

- c ignora los factores constantes: $300n^2$ y $3n^2$ quedan en la misma clase.
- n_0 ignora los n pequeños: la desigualdad puede fallar al inicio.
- **El orden importa:** primero se eligen c y n_0, una vez, y sirven para todos los $n \geq n_0$. La constante no puede depender de n.

Desde n_0, f queda entre dos copias de g. Con O solo vale la de arriba, y con Ω la de abajo.

Un caso: $f = 3n^2 + 10n$ y $g = n^2$



Un ejemplo completo: $3n^2 + 10n \in \Theta(n^2)$

Solo hay que encontrar constantes.

Un ejemplo completo: $3n^2 + 10n \in \Theta(n^2)$

Solo hay que encontrar constantes.

1. **Por arriba.** Si $n \geq 10$, entonces $10n \leq n^2$. Así, $3n^2 + 10n \leq 4n^2$. Sirven $c = 4$ y $n_0 = 10$.

Un ejemplo completo: $3n^2 + 10n \in \Theta(n^2)$

Solo hay que encontrar constantes.

1. **Por arriba.** Si $n \geq 10$, entonces $10n \leq n^2$. Así, $3n^2 + 10n \leq 4n^2$.
   Sirven $c = 4$ y $n_0 = 10$.
2. **Por abajo.** Como $10n \geq 0$, $3n^2 + 10n \geq 3n^2$ para todo n.
   Sirven $c = 3$ y $n_0 = 0$.

Un ejemplo completo: $3n^2 + 10n \in \Theta(n^2)$

Solo hay que encontrar constantes.

1. **Por arriba.** Si $n \geq 10$, entonces $10n \leq n^2$. Así, $3n^2 + 10n \leq 4n^2$.
   Sirven $c = 4$ y $n_0 = 10$.
2. **Por abajo.** Como $10n \geq 0$, $3n^2 + 10n \geq 3n^2$ para todo n.
   Sirven $c = 3$ y $n_0 = 0$.
3. **Juntando,** $3n^2 \leq T(n) \leq 4n^2$ para todo $n \geq 10$.

Un ejemplo completo: $3n^2 + 10n \in \Theta(n^2)$

Solo hay que encontrar constantes.

1. **Por arriba.** Si $n \geq 10$, entonces $10n \leq n^2$. Así, $3n^2 + 10n \leq 4n^2$.
   Sirven $c = 4$ y $n_0 = 10$.
2. **Por abajo.** Como $10n \geq 0$, $3n^2 + 10n \geq 3n^2$ para todo n.
   Sirven $c = 3$ y $n_0 = 0$.
3. **Juntando,** $3n^2 \leq T(n) \leq 4n^2$ para todo $n \geq 10$.

Las constantes no son únicas: la definición pide que exista alguna.

Un ejemplo completo: $3n^2 + 10n \in \Theta(n^2)$

Solo hay que encontrar constantes.

1. **Por arriba.** Si $n \geq 10$, entonces $10n \leq n^2$. Así, $3n^2 + 10n \leq 4n^2$.
   Sirven $c = 4$ y $n_0 = 10$.
2. **Por abajo.** Como $10n \geq 0$, $3n^2 + 10n \geq 3n^2$ para todo n.
   Sirven $c = 3$ y $n_0 = 0$.
3. **Juntando,** $3n^2 \leq T(n) \leq 4n^2$ para todo $n \geq 10$.

Las constantes no son únicas: la definición pide que exista alguna.

Algo que no está: $n^2 \notin O(n)$

Si $n^2 \leq c\,n$ para todo $n \geq n_0$, sería $n \leq c$ para todo n grande. Falso: tomen un n mayor que c y que n_0. Ninguna constante alcanza.

O no significa “crece exactamente así”

O es solo un techo. Todas estas son verdaderas:

$$5n \in O(n) \qquad 5n \in O(n^2) \qquad 5n \in O(2^n) \qquad 5n \in \Omega(1)$$

O no significa “crece exactamente así”

O es solo un techo. Todas estas son verdaderas:

$$5n \in O(n) \qquad 5n \in O(n^2) \qquad 5n \in O(2^n) \qquad 5n \in \Omega(1)$$

Pero $5n \notin \Omega(n^2)$. Solo $\Theta(n)$ describe bien a $5n$.

O no significa “crece exactamente así”

O es solo un techo. Todas estas son verdaderas:

$$5n \in O(n) \qquad 5n \in O(n^2) \qquad 5n \in O(2^n) \qquad 5n \in \Omega(1)$$

Pero $5n \notin \Omega(n^2)$. Solo $\Theta(n)$ describe bien a $5n$.

- Θ cuando conocemos el crecimiento exacto.
- O cuando solo garantizamos un techo.
- Ω cuando solo garantizamos un piso.

O no significa “crece exactamente así”

O es solo un techo. Todas estas son verdaderas:

$$5n \in O(n) \qquad 5n \in O(n^2) \qquad 5n \in O(2^n) \qquad 5n \in \Omega(1)$$

Pero $5n \notin \Omega(n^2)$. Solo $\Theta(n)$ describe bien a $5n$.

- Θ cuando conocemos el crecimiento exacto.
- O cuando solo garantizamos un techo.
- Ω cuando solo garantizamos un piso.

Error común

“Es $O(n^2)$, así que es lento.” La búsqueda lineal también es $O(n^2)$. Para decir que es lento hace falta una cota inferior.

Reglas para no volver a las definiciones

- **Factores constantes:** si $a > 0$, $a \cdot f \in \Theta(f)$.

Reglas para no volver a las definiciones

- **Factores constantes:** si $a > 0$, $a \cdot f \in \Theta(f)$.
- **En una suma manda el mayor:** $\max(f, g) \leq f + g \leq 2 \max(f, g)$. Por eso, $3n^2 + 10n + 7 \in \Theta(n^2)$.

Reglas para no volver a las definiciones

- **Factores constantes:** si $a > 0$, $a \cdot f \in \Theta(f)$.
- **En una suma manda el mayor:** $\max(f, g) \leq f + g \leq 2 \max(f, g)$. Por eso, $3n^2 + 10n + 7 \in \Theta(n^2)$.
- **La base del logaritmo no importa:** $\log_a n = \log_b n / \log_b a$.

Reglas para no volver a las definiciones

- **Factores constantes:** si $a > 0$, $a \cdot f \in \Theta(f)$.
- **En una suma manda el mayor:** $\max(f, g) \leq f + g \leq 2\max(f, g)$. Por eso, $3n^2 + 10n + 7 \in \Theta(n^2)$.
- **La base del logaritmo no importa:** $\log_a n = \log_b n / \log_b a$.
- **La base de la exponencial sí importa:** $4^n/2^n = 2^n$ crece sin límite, así que $4^n \notin O(2^n)$.

Reglas para no volver a las definiciones

- **Factores constantes:** si $a > 0$, $a \cdot f \in \Theta(f)$.
- **En una suma manda el mayor:** $\max(f, g) \leq f + g \leq 2 \max(f, g)$. Por eso, $3n^2 + 10n + 7 \in \Theta(n^2)$.
- **La base del logaritmo no importa:** $\log_a n = \log_b n / \log_b a$.
- **La base de la exponencial sí importa:** $4^n / 2^n = 2^n$ crece sin límite, así que $4^n \notin O(2^n)$.

$$1 \;<\; \log n \;<\; n \;<\; n \log n \;<\; n^2 \;<\; n^3 \;<\; 2^n$$

Cada una está en la O de la siguiente, y no al revés. Sin demostrar aquí: desigualdades como $n^2 \leq 2^n$ (desde $n = 4$) se demuestran por inducción en Matemática Discreta.

SECCIÓN 04

Mejor, peor y promedio

04

Dos entradas del mismo tamaño

cases.cpp

```cpp
int linear_search(const std::vector<int>& values, int key, long long& comparisons) {
    for (std::size_t i = 0; i < values.size(); i++) {
        comparisons++;
        if (values[i] == key) {
            return static_cast<int>(i);
        }
    }
    return -1;
}
```

Si la key está al inicio, 1 comparación. Si no está, n. ¿Cuál es “el costo con tamaño n”?

Tres funciones de n

Con $t(x)$ el costo de la entrada x

Peor caso *(worst case)*: el **máximo** de $t(x)$ entre las entradas de tamaño n.
Mejor caso *(best case)*: el **mínimo**.
Caso promedio *(average case)*: el **promedio**, con una distribución que hay que decir.

terminal

```
$ build/cases
busqueda lineal: comparaciones
       n    mejor     promedio     peor
    1000        1        500.5     1000
    2000        1       1000.5     2000
```

El promedio supone que la key está, con posiciones igual de probables: $(n + 1)/2$.

Un algoritmo con casos muy separados

terminal

```
$ build/cases
ordenamiento por insercion: comparaciones
       n   ordenado      al azar  invertido
    1000        999       245621     499500
    2000       1999       989741    1999000
    4000       3999      3979189    7998000
```

Un algoritmo con casos muy separados

terminal

```
$ build/cases
ordenamiento por insercion: comparaciones
       n   ordenado      al azar  invertido
    1000        999       245621     499500
    2000       1999       989741    1999000
    4000       3999      3979189    7998000
```

- Ya ordenado, cada elemento se compara una vez y se queda: mejor caso $\Theta(n)$.

Un algoritmo con casos muy separados

terminal

```
$ build/cases
ordenamiento por insercion: comparaciones
       n   ordenado      al azar  invertido
    1000        999       245621     499500
    2000       1999       989741    1999000
    4000       3999      3979189    7998000
```

- Ya ordenado, cada elemento se compara una vez y se queda: mejor caso $\Theta(n)$.
- Invertido, cada uno pasa por encima de todos los anteriores: peor caso $n(n-1)/2$, $\Theta(n^2)$.

Un algoritmo con casos muy separados

terminal

```
$ build/cases
ordenamiento por insercion: comparaciones
       n   ordenado      al azar  invertido
    1000        999       245621     499500
    2000       1999       989741    1999000
    4000       3999      3979189    7998000
```

- Ya ordenado, cada elemento se compara una vez y se queda: mejor caso $\Theta(n)$.
- Invertido, cada uno pasa por encima de todos los anteriores: peor caso $n(n-1)/2$, $\Theta(n^2)$.
- "Al azar" es una sola permutación, cerca de $n^2/4$: cada par queda invertido con probabilidad $1/2$.

Los casos no son las notaciones

El caso dice qué función describimos. **La notación** dice qué afirmamos de ella.

Búsqueda lineal	O	Ω	Θ
$T_{\text{mejor}}(n) = 1$	$O(1)$, $O(n)$	$\Omega(1)$	$\Theta(1)$
$T_{\text{prom}}(n) = (n+1)/2$	$O(n)$, $O(n^2)$	$\Omega(1)$, $\Omega(n)$	$\Theta(n)$
$T_{\text{peor}}(n) = n$	$O(n)$, $O(n^2)$	$\Omega(1)$, $\Omega(n)$	$\Theta(n)$

Son dos preguntas independientes. Todas esas celdas son verdaderas.

Error común

Creer que O es el peor caso, Ω el mejor y Θ el promedio. No hay ninguna relación de ese tipo.

¿Qué dice un peor caso $\Theta(n)$?

$c_1 n \leq T_{\text{peor}}(n) \leq c_2 n$ para todo $n \geq n_0$. Desarmado:

¿Qué dice un peor caso $\Theta(n)$?

$c_1 n \leq T_{\text{peor}}(n) \leq c_2 n$ para todo $n \geq n_0$. Desarmado:

- **El techo habla de todas las entradas:** ninguna de tamaño n cuesta más que $c_2 n$.

¿Qué dice un peor caso $\Theta(n)$?

$c_1 n \leq T_{\text{peor}}(n) \leq c_2 n$ para todo $n \geq n_0$. Desarmado:

- **El techo habla de todas las entradas:** ninguna de tamaño n cuesta más que $c_2 n$.
- **El piso habla de alguna entrada:** para cada $n \geq n_0$ existe una que cuesta al menos $c_1 n$.

¿Qué dice un peor caso $\Theta(n)$?

$c_1 n \leq T_{\text{peor}}(n) \leq c_2 n$ para todo $n \geq n_0$. Desarmado:

- **El techo habla de todas las entradas:** ninguna de tamaño n cuesta más que $c_2 n$.
- **El piso habla de alguna entrada:** para cada $n \geq n_0$ existe una que cuesta al menos $c_1 n$.

¿Puede tardar menos en alguna entrada? Sí. Si la key está al inicio, una sola comparación.

¿Qué dice un peor caso $\Theta(n)$?

$c_1 n \leq T_{\text{peor}}(n) \leq c_2 n$ para todo $n \geq n_0$. Desarmado:

- **El techo habla de todas las entradas:** ninguna de tamaño n cuesta más que $c_2 n$.
- **El piso habla de alguna entrada:** para cada $n \geq n_0$ existe una que cuesta al menos $c_1 n$.

¿Puede tardar menos en alguna entrada? Sí. Si la key está al inicio, una sola comparación.

Convención

"Es $O(f)$", sin nombrar el caso, significa para toda entrada: $T_{\text{peor}} \in O(f)$. "Es $\Omega(g)$", también para toda entrada: $T_{\text{mejor}} \in \Omega(g)$. En este curso, nombramos el caso.

SECCIÓN 05

Leer código: ciclos

05



Las reglas

- Cosas **en secuencia** se suman. Manda la mayor.
- Un **ciclo** es la suma de lo que cuesta cada vuelta.
- Una **llamada** cuesta lo que cuesta la función: hay que mirar adentro.

Ojo con lo que un ciclo esconde

`values.insert(values.begin(), x)` mueve todo lo que ya hay.
Pasar un `std::vector` por valor copia todos sus elementos.
Con un `break` o un `return`, el ciclo ya tiene varios casos.

En secuencia, y anidados

count_loops.cpp

```cpp
for (long long i = 0; i < n; i++) {
    count++;
}
for (long long i = 0; i < n; i++) {
    count++;
}
```

$n + n = 2n$. Es $\Theta(n)$.
Lo que multiplica es anidar, no poner uno después del otro.

count_loops.cpp

```cpp
for (long long i = 0; i < n; i++) {
    for (long long j = 0; j < n; j
     ++) {
        count++;
    }
}
```

$n \cdot n = n^2$. Es $\Theta(n^2)$.

El de adentro depende del de afuera

count_loops.cpp

```cpp
for (long long i = 0; i < n; i++)
    {
    for (long long j = 0; j < i; j
    ++) {
        count++;
    }
}
```

El de adentro depende del de afuera

count_loops.cpp

```cpp
for (long long i = 0; i < n; i++)
    {
    for (long long j = 0; j < i; j
     ++) {
        count++;
    }
}
```

Sumen dos veces, una al revés:

$$\begin{array}{rccccc} S = & 0 & + & 1 & + \cdots + & (n-1) \\ S = & (n-1) & + & (n-2) & + \cdots + & 0 \\ \hline 2S = & (n-1) & + & (n-1) & + \cdots + & (n-1) \end{array}$$

Son n términos. Entonces:

$$S = \frac{n(n-1)}{2}$$

El de adentro depende del de afuera

count_loops.cpp

```cpp
for (long long i = 0; i < n; i++)
    {
    for (long long j = 0; j < i; j
     ++) {
        count++;
    }
}
```

$S \leq n^2/2$ para $n \geq 1$, y $S \geq n^2/4$ para $n \geq 2$: $\Theta(n^2)$.
Una fracción fija de n no cambia la clase.

Sumen dos veces, una al revés:

$$\begin{array}{rccccc} S = & 0 & + & 1 & + \cdots + & (n-1) \\ S = & (n-1) & + & (n-2) & + \cdots + & 0 \\ \hline 2S = & (n-1) & + & (n-1) & + \cdots + & (n-1) \end{array}$$

Son n términos. Entonces:

$$S = \frac{n(n-1)}{2}$$

Ciclos que dividen entre dos

count_loops.cpp

```cpp
for (long long j = n; j > 0; j /= 2) {
    count++;
}
```

- Después de k vueltas, $j \leq n/2^k$. Termina cuando $2^k > n$.

Ciclos que dividen entre dos

count_loops.cpp

```cpp
for (long long j = n; j > 0; j /= 2) {
    count++;
}
```

- Después de k vueltas, $j \leq n/2^k$. Termina cuando $2^k > n$.
- Son $\lfloor \log_2 n \rfloor + 1$ vueltas, para $n \geq 1$. Es $\Theta(\log n)$.

Ciclos que dividen entre dos

count_loops.cpp

```cpp
for (long long j = n; j > 0; j /= 2) {
    count++;
}
```

- Después de k vueltas, $j \leq n/2^k$. Termina cuando $2^k > n$.
- Son $\lfloor \log_2 n \rfloor + 1$ vueltas, para $n \geq 1$. Es $\Theta(\log n)$.
- Con $n = 1000$: 1000, 500, 250, 125, 62, 31, 15, 7, 3, 1. Son 10 vueltas.

Ciclos que dividen entre dos

count_loops.cpp

```cpp
for (long long j = n; j > 0; j /= 2) {
    count++;
}
```

- Después de k vueltas, $j \leq n/2^k$. Termina cuando $2^k > n$.
- Son $\lfloor \log_2 n \rfloor + 1$ vueltas, para $n \geq 1$. Es $\Theta(\log n)$.
- Con $n = 1000$: 1000, 500, 250, 125, 62, 31, 15, 7, 3, 1. Son 10 vueltas.
- La señal: la variable cambia por un **factor**, no por una cantidad fija.

Por fuera n, adentro uno que divide

count_loops.cpp

```cpp
for (long long i = 0; i < n; i++) {
    for (long long j = i; j > 0; j /= 2) {
        count++;
    }
}
```

El de adentro da $\lfloor \log_2 i \rfloor + 1$ vueltas si $i \geq 1$. Basta acotar.

Por fuera n, adentro uno que divide

count_loops.cpp

```cpp
for (long long i = 0; i < n; i++) {
    for (long long j = i; j > 0; j /= 2) {
        count++;
    }
}
```

El de adentro da $\lfloor \log_2 i \rfloor + 1$ vueltas si $i \geq 1$. Basta acotar.

- **Por arriba:** a lo más n vueltas por $\log_2 n + 1$: $n(\log_2 n + 1) \leq 2n \log_2 n$, para $n \geq 2$.

Por fuera n, adentro uno que divide

count_loops.cpp

```cpp
for (long long i = 0; i < n; i++) {
    for (long long j = i; j > 0; j /= 2) {
        count++;
    }
}
```

El de adentro da $\lfloor \log_2 i \rfloor + 1$ vueltas si $i \geq 1$. Basta acotar.

- **Por arriba:** a lo más n vueltas por $\log_2 n + 1$: $n(\log_2 n + 1) \leq 2n \log_2 n$, para $n \geq 2$.
- **Por abajo:** solo la mitad con $i \geq n/2$: $\frac{n}{2}(\log_2 n - 1) \geq \frac{1}{4} n \log_2 n$, para $n \geq 4$.

Por fuera n, adentro uno que divide

count_loops.cpp

```cpp
for (long long i = 0; i < n; i++) {
    for (long long j = i; j > 0; j /= 2) {
        count++;
    }
}
```

El de adentro da $\lfloor \log_2 i \rfloor + 1$ vueltas si $i \geq 1$. Basta acotar.

- **Por arriba:** a lo más n vueltas por $\log_2 n + 1$: $n(\log_2 n + 1) \leq 2n \log_2 n$, para $n \geq 2$.
- **Por abajo:** solo la mitad con $i \geq n/2$: $\frac{n}{2}(\log_2 n - 1) \geq \frac{1}{4} n \log_2 n$, para $n \geq 4$.

Es $\Theta(n \log n)$. Tirar la mitad de la suma sirve muy seguido para las cotas inferiores.

Los cinco, contados

terminal

```
$ build/count_loops
patron                n=1000     n=2000     n=4000     n=8000   razon
secuencia               2000       4000       8000      16000    2.00
anidados             1000000    4000000   16000000   64000000    4.00
triangulo             499500    1999000    7998000   31996000    4.00
divide                    10         11         12         13    1.08
triangulo_divide        8977      19953      43905      95809    2.18
```

- `razon` es la cuenta con $n = 8000$ entre la cuenta con $n = 4000$.

Los cinco, contados

terminal

```
$ build/count_loops
patron                n=1000     n=2000     n=4000     n=8000   razon
secuencia               2000       4000       8000      16000    2.00
anidados             1000000    4000000   16000000   64000000    4.00
triangulo             499500    1999000    7998000   31996000    4.00
divide                    10         11         12         13    1.08
triangulo_divide        8977      19953      43905      95809    2.18
```

- `razon` es la cuenta con $n = 8000$ entre la cuenta con $n = 4000$.
- Fila `divide`: cada vez que n se duplica, la cuenta suma 1. Esa es la marca de un logaritmo.

Los cinco, contados

terminal

```
$ build/count_loops
patron                n=1000     n=2000     n=4000     n=8000   razon
secuencia               2000       4000       8000      16000    2.00
anidados             1000000    4000000   16000000   64000000    4.00
triangulo             499500    1999000    7998000   31996000    4.00
divide                    10         11         12         13    1.08
triangulo_divide        8977      19953      43905      95809    2.18
```

- `razon` es la cuenta con $n = 8000$ entre la cuenta con $n = 4000$.
- Fila `divide`: cada vez que n se duplica, la cuenta suma 1. Esa es la marca de un logaritmo.
- Son cuentas exactas, no tiempos. Dan lo mismo en cualquier máquina.

SECCIÓN 06

Recursión

06

Dos llamadas con $n/2$: el árbol

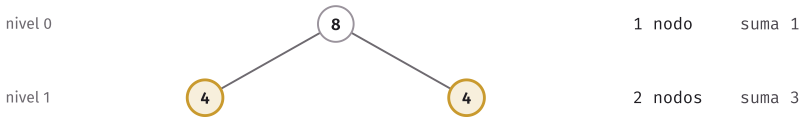
$T(n) = 2\,T(n/2) + 1$. El trabajo propio de cada llamada es constante.



La llamada de n = 8 es un solo nodo, y hace trabajo constante

Dos llamadas con $n/2$: el árbol

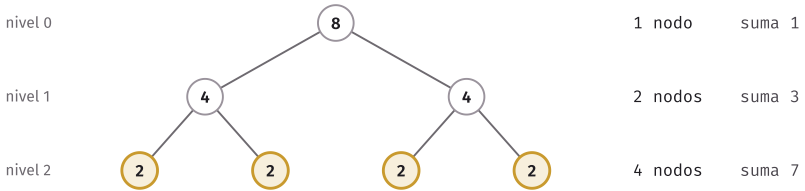
$T(n) = 2\,T(n/2) + 1$. El trabajo propio de cada llamada es constante.



Hace dos llamadas con n = 4: el nivel 1 tiene 2 nodos

Dos llamadas con $n/2$: el árbol

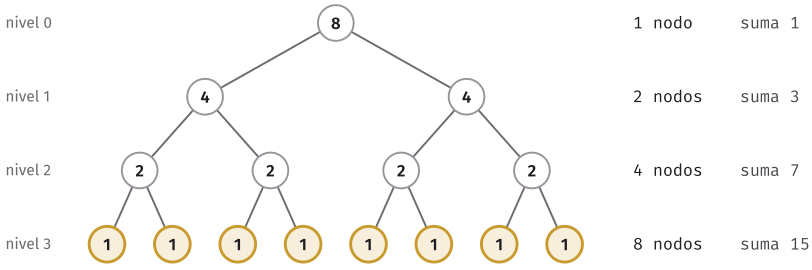
$T(n) = 2\,T(n/2) + 1$. El trabajo propio de cada llamada es constante.



Cada uno hace dos más. El nivel 2 tiene 4 nodos

Dos llamadas con $n/2$: el árbol

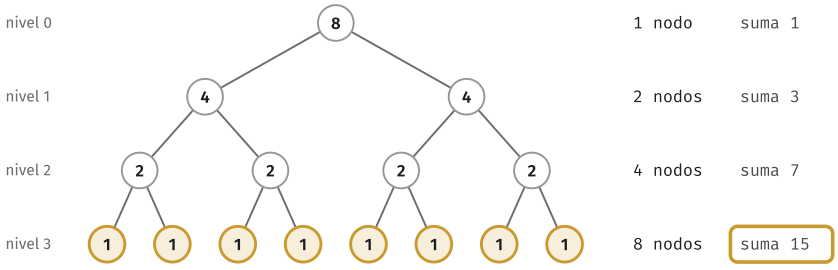
$T(n) = 2\,T(n/2) + 1$. El trabajo propio de cada llamada es constante.



Con n = 1, caso base. El nivel 3 tiene 8 nodos: cada nivel duplica al anterior

Dos llamadas con $n/2$: el árbol

$T(n) = 2\,T(n/2) + 1$. El trabajo propio de cada llamada es constante.



1 + 2 + 4 + 8 = 15 llamadas, o sea 2n − 1. Más de la mitad son las 8 hojas: Θ(n)

El nivel i tiene 2^i nodos, y hay $\log_2 n + 1$ niveles. El total es $1 + 2 + 4 + \cdots + 2^k$, con $2^k = n$.

Sumar sin inducción

Multipliquen por 2 y resten:

$$\begin{array}{rl} 2S = & 2 + 4 + \cdots + 2^{k+1} \\ S = & 1 + 2 + 4 + \cdots + 2^{k} \\ \hline S = & 2^{k+1} - 1 \end{array}$$

terminal

```
$ build/call_tree
T(n) = 2 T(n/2) + 1
         n      llamadas      trabajo profund.
         8            15            15        3
        16            31            31        4
      1024          2047          2047       10
      2048          4095          4095       11
```

Sumar sin inducción

Multipliquen por 2 y resten:

$$\begin{array}{rl} 2S = & 2 + 4 + \cdots + 2^{k+1} \\ S = & 1 + 2 + 4 + \cdots + 2^{k} \\ \hline S = & 2^{k+1} - 1 \end{array}$$

Como $2^k = n$: $S = 2n - 1$ llamadas, y es $\Theta(n)$.

terminal

```
$ build/call_tree
T(n) = 2 T(n/2) + 1
         n      llamadas      trabajo profund.
         8            15            15        3
        16            31            31        4
      1024          2047          2047       10
      2048          4095          4095       11
```

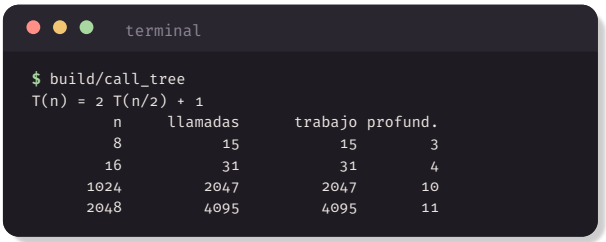
Sumar sin inducción

Multipliquen por 2 y resten:

$$\begin{array}{rl} 2S = & 2 + 4 + \cdots + 2^{k+1} \\ S = & 1 + 2 + 4 + \cdots + 2^{k} \\ \hline S = & 2^{k+1} - 1 \end{array}$$

Como $2^k = n$: $S = 2n - 1$ llamadas, y es $\Theta(n)$.

Dos llamadas, y da **lineal**. Más de la mitad de las llamadas son hojas.



Sumar sin inducción

Multipliquen por 2 y resten:

$$\begin{array}{rl} 2S = & 2 + 4 + \cdots + 2^{k+1} \\ S = & 1 + 2 + 4 + \cdots + 2^{k} \\ \hline S = & 2^{k+1} - 1 \end{array}$$

Como $2^k = n$: $S = 2n - 1$ llamadas, y es $\Theta(n)$.

Dos llamadas, y da **lineal**. Más de la mitad de las llamadas son hojas.

terminal

```
$ build/call_tree
T(n) = 2 T(n/2) + 1
         n      llamadas      trabajo profund.
         8            15            15        3
        16            31            31        4
      1024          2047          2047       10
      2048          4095          4095       11
```

Si n no es potencia de 2: con $k = \lfloor \log_2 n \rfloor$ son $2^{k+1} - 1$ llamadas, entre n y $2n - 1$. Sigue siendo $\Theta(n)$.

Dos llamadas con $n/2$ y trabajo lineal

$T(n) = 2\,T(n/2) + n$. Es la forma de mergesort. Por niveles, con $n = 8$:

nivel	nodos	tamaño	trabajo del nivel
0	1	8	$1 \times 8 = 8$
1	2	4	$2 \times 4 = 8$
2	4	2	$4 \times 2 = 8$
3	8	1	$8 \times 1 = 8$

Dos llamadas con $n/2$ y trabajo lineal

$T(n) = 2\,T(n/2) + n$. Es la forma de mergesort. Por niveles, con $n = 8$:

nivel	nodos	tamaño	trabajo del nivel
0	1	8	$1 \times 8 = 8$
1	2	4	$2 \times 4 = 8$
2	4	2	$4 \times 2 = 8$
3	8	1	$8 \times 1 = 8$

Doble de nodos, mitad de tamaño: cada nivel suma n. Hay $\log_2 n + 1$ niveles.

Dos llamadas con $n/2$ y trabajo lineal

$T(n) = 2\,T(n/2) + n$. Es la forma de mergesort. Por niveles, con $n = 8$:

nivel	nodos	tamaño	trabajo del nivel
0	1	8	$1 \times 8 = 8$
1	2	4	$2 \times 4 = 8$
2	4	2	$4 \times 2 = 8$
3	8	1	$8 \times 1 = 8$

Doble de nodos, mitad de tamaño: cada nivel suma n. Hay $\log_2 n + 1$ niveles.

Total: $n(\log_2 n + 1)$**, que es** $\Theta(n \log n)$

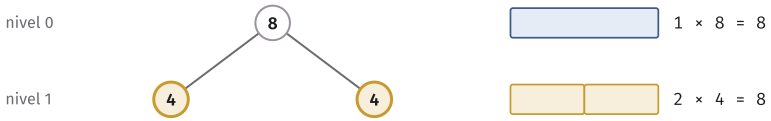
Con $n = 1024$: $1024 \times 11 = 11\,264$, y es lo que cuenta `call_tree`.

Cada nivel suma *n*



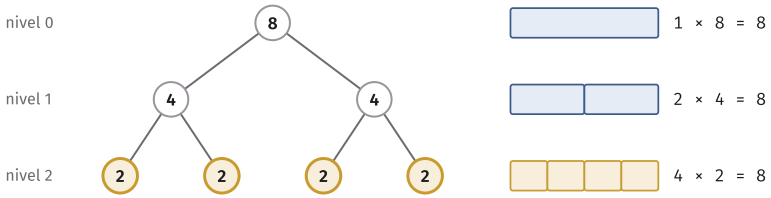
La raíz trabaja su tamaño: 8

Cada nivel suma *n*



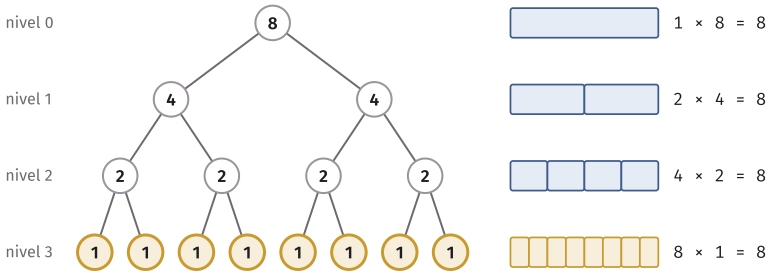
En el nivel 1 hay 2 nodos de tamaño 4. Trabajan 2 × 4 = 8

Cada nivel suma *n*



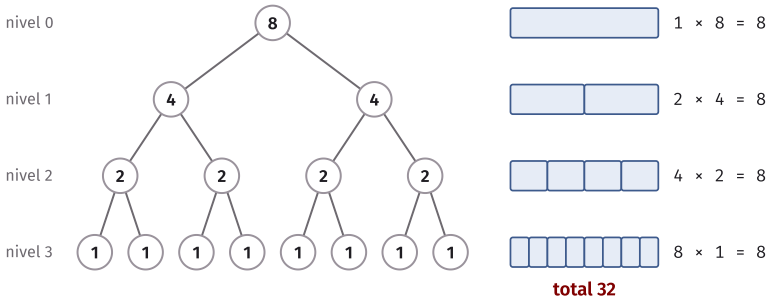
El doble de nodos, de la mitad de tamaño: 4 × 2 = 8

Cada nivel suma *n*



Las 8 hojas de tamaño 1 también suman 8. Cada nivel suma n

Cada nivel suma *n*



Cuatro niveles de 8: 32 en total. Con n general, n ($\log_2 n + 1$), que es $\Theta(n \log n)$

Dos llamadas con $n - 1$

$T(n) = 2\,T(n-1) + 1$. Parece igual a la de $n/2$, y es otro mundo.

Dos llamadas con $n - 1$

$T(n) = 2\,T(n-1) + 1$. Parece igual a la de $n/2$, y es otro mundo.

- El tamaño baja de 1 en 1: $n + 1$ niveles.

Dos llamadas con $n - 1$

$T(n) = 2\,T(n-1) + 1$. Parece igual a la de $n/2$, y es otro mundo.

- El tamaño baja de 1 en 1: $n + 1$ niveles.
- Los niveles se siguen duplicando: $1 + 2 + \cdots + 2^n = 2^{n+1} - 1$. Es $\Theta(2^n)$.

terminal

```
$ build/call_tree
T(n) = 2 T(n-1) + 1
         n     llamadas      trabajo profund.
         5           63           63        5
        10         2047         2047       10
        20      2097151      2097151       20
```

Dos llamadas con $n - 1$

$T(n) = 2\,T(n-1) + 1$. Parece igual a la de $n/2$, y es otro mundo.

- El tamaño baja de 1 en 1: $n + 1$ niveles.
- Los niveles se siguen duplicando: $1 + 2 + \cdots + 2^n = 2^{n+1} - 1$. Es $\Theta(2^n)$.

terminal

```
$ build/call_tree
T(n) = 2 T(n-1) + 1
         n     llamadas      trabajo profund.
         5           63           63        5
        10         2047         2047       10
        20      2097151      2097151       20
```

Para analizar una recursión: dibujar el árbol, contar niveles y nodos por nivel, calcular el trabajo de un nivel, y sumar.

SECCIÓN 07

Espacio

07

Espacio además de tiempo

Espacio: la memoria que se usa además de la entrada, como función de n. Invertir creando otro arreglo usa $\Theta(n)$. Hacerlo en el lugar *(in place)*, $\Theta(1)$.

Espacio además de tiempo

Espacio: la memoria que se usa además de la entrada, como función de n. Invertir creando otro arreglo usa $\Theta(n)$. Hacerlo en el lugar *(in place)*, $\Theta(1)$.

Recursión	Tiempo	Pila (la profundidad)
$T(n) = T(n/2) + 1$	$\Theta(\log n)$	$\Theta(\log n)$
$T(n) = 2\,T(n/2) + 1$	$\Theta(n)$	$\Theta(\log n)$
$T(n) = 2\,T(n/2) + n$	$\Theta(n \log n)$	$\Theta(\log n)$
$T(n) = 2\,T(n-1) + 1$	$\Theta(2^n)$	$\Theta(n)$

Espacio además de tiempo

Espacio: la memoria que se usa además de la entrada, como función de n. Invertir creando otro arreglo usa $\Theta(n)$. Hacerlo en el lugar *(in place)*, $\Theta(1)$.

Recursión	Tiempo	Pila (la profundidad)
$T(n) = T(n/2) + 1$	$\Theta(\log n)$	$\Theta(\log n)$
$T(n) = 2\,T(n/2) + 1$	$\Theta(n)$	$\Theta(\log n)$
$T(n) = 2\,T(n/2) + n$	$\Theta(n \log n)$	$\Theta(\log n)$
$T(n) = 2\,T(n-1) + 1$	$\Theta(2^n)$	$\Theta(n)$

Error común

"No reservé con `new`, así que es $\Theta(1)$." La recursión usa pila, y una copia de `std::vector` usa heap.

SECCIÓN 08

Medir sin engañarse

08

El reloj

first_timer.cpp

```cpp
auto start = std::chrono::steady_clock::now();
long long total = sum_values(values);
auto stop = std::chrono::steady_clock::now();
keep(total);
double seconds = std::chrono::duration<double>(stop - start).count();
```

- `steady_clock` nunca retrocede. `system_clock` es la hora del calendario, y se puede ajustar mientras el programa corre.

El reloj

first_timer.cpp

```cpp
auto start = std::chrono::steady_clock::now();
long long total = sum_values(values);
auto stop = std::chrono::steady_clock::now();
keep(total);
double seconds = std::chrono::duration<double>(stop - start).count();
```

- `steady_clock` nunca retrocede. `system_clock` es la hora del calendario, y se puede ajustar mientras el programa corre.
- El reloj no es infinitamente fino: aquí, su salto mínimo es de unos 41 ns.

El reloj

first_timer.cpp

```cpp
auto start = std::chrono::steady_clock::now();
long long total = sum_values(values);
auto stop = std::chrono::steady_clock::now();
keep(total);
double seconds = std::chrono::duration<double>(stop - start).count();
```

- `steady_clock` nunca retrocede. `system_clock` es la hora del calendario, y se puede ajustar mientras el programa corre.
- El reloj no es infinitamente fino: aquí, su salto mínimo es de unos 41 ns.
- Lo que dura menos que eso se mide en tandas: muchas llamadas seguidas, y se divide.

Con -O2 y sin sanitizers

terminal, depende de la máquina

```
$ build/sanitizer_cost
suma de 1000000 enteros:       355 us
100000 bloques pequenos:       978 us
bloques / suma:               2.76
$ build/sanitizer_cost_san
suma de 1000000 enteros:      1277 us
100000 bloques pequenos:     11995 us
bloques / suma:               9.40
```

Con -O2 y sin sanitizers

terminal, depende de la máquina

```
$ build/sanitizer_cost
suma de 1000000 enteros:       355 us
100000 bloques pequenos:       978 us
bloques / suma:               2.76
$ build/sanitizer_cost_san
suma de 1000000 enteros:      1277 us
100000 bloques pequenos:     11995 us
bloques / suma:               9.40
```

- Las dos con `-O2`. La suma, 3.6 veces más lenta. Los bloques, unas 12.

Con -O2 y sin sanitizers

terminal, depende de la máquina

```
$ build/sanitizer_cost
suma de 1000000 enteros:       355 us
100000 bloques pequenos:       978 us
bloques / suma:               2.76
$ build/sanitizer_cost_san
suma de 1000000 enteros:      1277 us
100000 bloques pequenos:     11995 us
bloques / suma:               9.40
```

- Las dos con `-O2`. La suma, 3.6 veces más lenta. Los bloques, unas 12.
- No es un factor parejo: la razón entre las dos pasó de 2.8 a 9.4.

Con -O2 y sin sanitizers

terminal, depende de la máquina

```
$ build/sanitizer_cost
suma de 1000000 enteros:       355 us
100000 bloques pequenos:       978 us
bloques / suma:               2.76
$ build/sanitizer_cost_san
suma de 1000000 enteros:      1277 us
100000 bloques pequenos:     11995 us
bloques / suma:               9.40
```

- Las dos con `-O2`. La suma, 3.6 veces más lenta. Los bloques, unas 12.
- No es un factor parejo: la razón entre las dos pasó de 2.8 a 9.4.
- **El flujo:** primero, comprobar que es correcto, con sanitizers. Después, medirlo, sin ellos.

Repetir y usar la mediana

Nueve mediciones de la misma suma, en microsegundos:

```
349  356  907  349  349  364  349  349  360
```

Media: 415. Mediana: 349.

Repetir y usar la mediana

Nueve mediciones de la misma suma, en microsegundos:

```
349 356 907 349 349 364 349 349 360
```

Media: 415. Mediana: 349.

- Casi todo el ruido hace la medición más lenta: interrupciones, otro proceso, la cache ocupada.

Repetir y usar la mediana

Nueve mediciones de la misma suma, en microsegundos:

```
349 356 907 349 349 364 349 349 360
```

Media: 415. Mediana: 349.

- Casi todo el ruido hace la medición más lenta: interrupciones, otro proceso, la cache ocupada.
- Los valores atípicos están de un solo lado. La media se deja arrastrar. La mediana los ignora.

Repetir y usar la mediana

Nueve mediciones de la misma suma, en microsegundos:

```
349  356  907  349  349  364  349  349  360
```

Media: 415. Mediana: 349.

- Casi todo el ruido hace la medición más lenta: interrupciones, otro proceso, la cache ocupada.
- Los valores atípicos están de un solo lado. La media se deja arrastrar. La mediana los ignora.

> **Depende de la máquina**
>
> Estos tiempos son de una corrida, en una máquina. En la suya serán otros.

`median_seconds`, en `bench.h`

bench.h

```cpp
template <typename Fn>
double median_seconds(int reps, Fn fn) {
    assert(reps >= 1);
    std::vector<double> times;
    times.reserve(static_cast<std::size_t>(reps));
    for (int i = 0; i < reps; i++) {
        auto start = std::chrono::steady_clock::now();
        fn();
        auto stop = std::chrono::steady_clock::now();
        times.push_back(std::chrono::duration<double>(stop - start).count());
    }
    return median_of(times);
}
```

Error común: preparar los datos dentro de `fn`. Todo lo que pasa adentro se mide.

El compilador puede borrar lo que miden

terminal, depende de la máquina

```
$ build/no_keep
n =  1000000   sin keep:          0 ns
n =  1000000   con keep:     351958 ns
n =  2000000   sin keep:          0 ns
n =  2000000   con keep:     714750 ns
n =  4000000   sin keep:          0 ns
n =  4000000   con keep:    1421750 ns
```

El compilador puede borrar lo que miden

terminal, depende de la máquina

```
$ build/no_keep
n =  1000000   sin keep:          0 ns
n =  1000000   con keep:     351958 ns
n =  2000000   sin keep:          0 ns
n =  2000000   con keep:     714750 ns
n =  4000000   sin keep:          0 ns
n =  4000000   con keep:    1421750 ns
```

Sumar cuatro millones de enteros en 0 ns no es creíble. Si el resultado no se usa, el ciclo no tiene efecto visible, y el compilador lo puede borrar.

El compilador puede borrar lo que miden

terminal, depende de la máquina

```
$ build/no_keep
n =  1000000   sin keep:          0 ns
n =  1000000   con keep:     351958 ns
n =  2000000   sin keep:          0 ns
n =  2000000   con keep:     714750 ns
n =  4000000   sin keep:          0 ns
n =  4000000   con keep:    1421750 ns
```

Sumar cuatro millones de enteros en 0 ns no es creíble. Si el resultado no se usa, el ciclo no tiene efecto visible, y el compilador lo puede borrar.

Si un tiempo no crece con *n*, o es sospechosamente pequeño, primero duden de la medición.

keep

bench.h

```
template <typename T>
void keep(const T& value) {
    asm volatile("" : : "r"(&value) : "memory");
}
```

errors/no_keep.cpp

```
seconds = median_seconds(9, [&] { keep(sum_values(values)); });
```

- Un bloque de ensamblador vacío: no genera ninguna instrucción.
- El compilador debe suponer que alguien lee `value`, así que tiene que calcularlo.
- Es una extensión de g++ y clang, no parte del estándar de C++.

keep no alcanza siempre

errors/closed_form.cpp

```cpp
for (long long i = 0; i < n; i++) {
    for (long long j = 0; j < n; j++) {
        count++;
    }
}
```

terminal, depende de la máquina

```
$ build/closed_form
n =  1000   vueltas =   1000000         41 ns
n =  8000   vueltas =  64000000          0 ns
$ build/closed_form_O0
n =  1000   vueltas =   1000000    1471000 ns
n =  8000   vueltas =  64000000   93936667 ns
```

`keep` no alcanza siempre

errors/closed_form.cpp

```cpp
for (long long i = 0; i < n; i++) {
    for (long long j = 0; j < n; j++) {
        count++;
    }
}
```

terminal, depende de la máquina

```
$ build/closed_form
n =  1000   vueltas =   1000000         41 ns
n =  8000   vueltas =  64000000          0 ns
$ build/closed_form_O0
n =  1000   vueltas =   1000000    1471000 ns
n =  8000   vueltas =  64000000   93936667 ns
```

- `keep` protege el resultado, no el camino. Con `-O2`, g++ reemplaza el ciclo por una multiplicación: $\Theta(1)$.

`keep` no alcanza siempre

errors/closed_form.cpp

```cpp
for (long long i = 0; i < n; i++) {
    for (long long j = 0; j < n; j++) {
        count++;
    }
}
```

terminal, depende de la máquina

```
$ build/closed_form
n =  1000   vueltas =   1000000         41 ns
n =  8000   vueltas =  64000000          0 ns
$ build/closed_form_O0
n =  1000   vueltas =   1000000    1471000 ns
n =  8000   vueltas =  64000000   93936667 ns
```

- `keep` protege el resultado, no el camino. Con `-O2`, g++ reemplaza el ciclo por una multiplicación: $\Theta(1)$.
- Para medir, el trabajo debe depender de los datos: que no se pueda resumir en una fórmula.

Tampoco repitan los mismos datos

terminal, depende de la máquina

```
$ build/same_array
       n     mismo (us) distintos (us)
    1000            7.5           33.6
    4000           37.8          160.3
   16000          467.2          744.5
```

Copias del mismo arreglo salen unas cuatro veces más rápido con n pequeño. Lo más probable: el predictor de saltos aprende el recorrido.

Tampoco repitan los mismos datos

terminal, depende de la máquina

```
$ build/same_array
       n     mismo (us) distintos (us)
    1000            7.5           33.6
    4000           37.8          160.3
   16000          467.2          744.5
```

Copias del mismo arreglo salen unas cuatro veces más rápido con n pequeño. Lo más probable: el predictor de saltos aprende el recorrido.

Una medición honesta

`-O2` y sin sanitizers. Repeticiones y mediana. `keep`. Trabajo y datos que no se puedan adivinar. Y si el resultado parece demasiado bueno, sospechen.

SECCIÓN 09

El experimento de duplicar

09

La razón cancela la constante

Si $T(n) \approx c \cdot n^k$:

$$\frac{T(2n)}{T(n)} \approx \frac{c\,(2n)^k}{c\,n^k} = 2^k$$

El exponente se lee en la razón, y no depende de la máquina.

Clase	$T(2n)/T(n)$	con n grande
1	1	1
$\log n$	$(\log_2 n + 1)/\log_2 n$	tiende a 1
n	2	2
$n \log n$	$2(\log_2 n + 1)/\log_2 n$	tiende a 2
n^2	4	4
n^3	8	8
2^n	2^n	sin límite

La razón cancela la constante

Si $T(n) \approx c \cdot n^k$:

$$\frac{T(2n)}{T(n)} \approx \frac{c\,(2n)^k}{c\,n^k} = 2^k$$

El exponente se lee en la razón, y no depende de la máquina.

Práctica. 1, 2, 4, 8 segundos: razón 2, lineal o casi. 1, 4, 16, 64 segundos: razón 4, cuadrático.

Clase	$T(2n)/T(n)$	con n grande
1	1	1
$\log n$	$(\log_2 n + 1)/\log_2 n$	tiende a 1
n	2	2
$n \log n$	$2(\log_2 n + 1)/\log_2 n$	tiende a 2
n^2	4	4
n^3	8	8
2^n	2^n	sin límite

Un experimento de verdad: n y $n \log n$

terminal, depende de la máquina

```
$ build/doubling
suma
         n   T(n) en us   razon  T/n (ns)
      8000         2.89       -      0.36
     16000         5.72    1.98      0.36
     32000        11.54    2.02      0.36
     64000        23.41    2.03      0.37
    128000        46.95    2.01      0.37
    256000        93.81    2.00      0.37
```

terminal, depende de la máquina

```
$ build/doubling
std::sort
         n   T(n) en us   razon  T/n (ns)
      8000       348.96       -     43.62
     16000       737.62    2.11     46.10
     32000      1597.29    2.17     49.92
     64000      3387.62    2.12     52.93
    128000      7159.25    2.11     55.93
    256000     15253.46    2.13     59.58
```

Compilado con `-O2`. La suma, con tandas de al menos 2 ms. `std::sort`, con 15 arreglos distintos. En ambos, la mediana. Sus números van a ser otros: comparen las razones.

Un experimento de verdad: n^2

terminal, depende de la máquina

```
$ build/doubling
inversiones
         n   T(n) en us    razon  T/n (ns)
      1000       186.79        -    186.79
      2000       727.36     3.89    363.68
      4000      2862.75     3.94    715.69
      8000     11379.79     3.98   1422.47
     16000     45618.62     4.01   2851.16
```

- Sube hacia 4: con n pequeño todavía pesan los términos de menor orden.

Un experimento de verdad: n^2

terminal, depende de la máquina

```
$ build/doubling
inversiones
         n   T(n) en us    razon  T/n (ns)
      1000       186.79        -    186.79
      2000       727.36     3.89    363.68
      4000      2862.75     3.94    715.69
      8000     11379.79     3.98   1422.47
     16000     45618.62     4.01   2851.16
```

- Sube hacia 4: con n pequeño todavía pesan los términos de menor orden.
- Las predicciones se cumplen en las tres tablas.

Cómo distinguir clases parecidas

Con $n \log n$ la razón es 2.1: muy cerca de 2. Una sola razón no alcanza.

Razón cerca de	Miren	Para decidir
1	$T(2n) - T(n)$	cerca de 0: constante; una cantidad fija: $\log n$
2	T/n	plano: n; sube una cantidad fija por fila: $n \log n$
sigue creciendo	los n	exponencial, o peor que cualquier potencia

Cómo distinguir clases parecidas

Con $n \log n$ la razón es 2.1: muy cerca de 2. Una sola razón no alcanza.

Razón cerca de	Miren	Para decidir
1	$T(2n) - T(n)$	cerca de 0: constante; una cantidad fija: $\log n$
2	T/n	plano: n; sube una cantidad fija por fila: $n \log n$
sigue creciendo	los n	exponencial, o peor que cualquier potencia

En la tabla: T/n en la suma, plano en 0.36. En `std::sort`, sube de 43.62 a 59.58.

Cómo distinguir clases parecidas

Con $n \log n$ la razón es 2.1: muy cerca de 2. Una sola razón no alcanza.

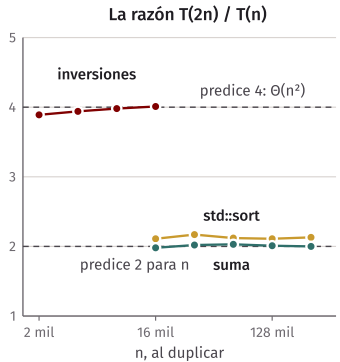
Razón cerca de	Miren	Para decidir
1	$T(2n) - T(n)$	cerca de 0: constante; una cantidad fija: $\log n$
2	T/n	plano: n; sube una cantidad fija por fila: $n \log n$
sigue creciendo	los n	exponencial, o peor que cualquier potencia

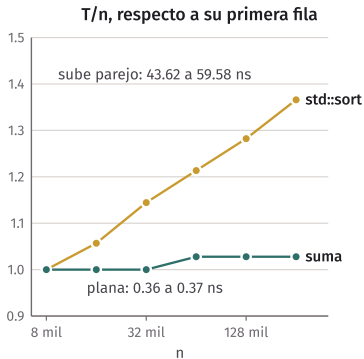
En la tabla: T/n en la suma, plano en 0.36. En `std::sort`, sube de 43.62 a 59.58.

Error común

Mirar solo la última fila. Una razón que cambia de un valor a otro dice que el programa cambió de régimen.

La razón y T/n, dibujadas





Lo que el experimento no prueba

Una medición es evidencia, no una demostración.

Lo que el experimento no prueba

Una medición es evidencia, no una demostración.

1. Solo ve los n que midieron. Lo que pase después de la última fila no está en la tabla.

Lo que el experimento no prueba

Una medición es evidencia, no una demostración.

1. Solo ve los n que midieron. Lo que pase después de la última fila no está en la tabla.
2. Solo ve las entradas que probaron. Si midieron el mejor caso, midieron el mejor caso.

Lo que el experimento no prueba

Una medición es evidencia, no una demostración.

1. Solo ve los n que midieron. Lo que pase después de la última fila no está en la tabla.
2. Solo ve las entradas que probaron. Si midieron el mejor caso, midieron el mejor caso.
3. Supone $T(n) \approx c\, n^k$. Estar en $\Theta(n^k)$ no basta: si $T(n) = n$ cuando $\lfloor \log_2 n \rfloor$ es par, y $T(n) = 2n$ cuando es impar, entonces $T \in \Theta(n)$, y las razones en potencias de 2 se alternan entre 4 y 1.

Lo que el experimento no prueba

Una medición es evidencia, no una demostración.

1. Solo ve los n que midieron. Lo que pase después de la última fila no está en la tabla.
2. Solo ve las entradas que probaron. Si midieron el mejor caso, midieron el mejor caso.
3. Supone $T(n) \approx c\, n^k$. Estar en $\Theta(n^k)$ no basta: si $T(n) = n$ cuando $\lfloor \log_2 n \rfloor$ es par, y $T(n) = 2n$ cuando es impar, entonces $T \in \Theta(n)$, y las razones en potencias de 2 se alternan entre 4 y 1.

Predecir y medir

Cuando coinciden, confíen. Cuando no coinciden, hay algo que aprender: un error en la cuenta, un error en la medición, o algo que el modelo no estaba viendo.

SECCIÓN 10

Lo que *O* no ve

10

$O(n)$ **no siempre gana a** $O(n^2)$

¿Hay repetidos, entre valores de 0 a $2^{20} - 1$? Con $U = 2^{20}$ y valores distintos ($n \leq U$).
Pares: dos ciclos, $\Theta(n^2)$. **Tabla:** limpia U casillas, siempre, y hace una pasada, $\Theta(U + n)$.

terminal, depende de la máquina

```
$ build/small_n
       n     pares (us)     tabla (us)  pares/tabla
      10           0.04           5.92        0.007
     100           2.38           6.54        0.363
    1000         185.46          11.79       15.727
    4000        2868.33          17.38      165.084
   16000       45519.33          33.21     1370.734
   32000      181941.17          45.92     3962.392
```

$O(n)$ **no siempre gana a** $O(n^2)$

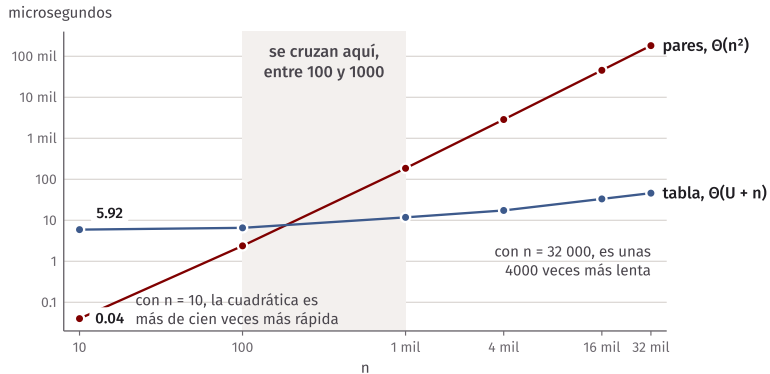
¿Hay repetidos, entre valores de 0 a $2^{20} - 1$? Con $U = 2^{20}$ y valores distintos ($n \leq U$).
Pares: dos ciclos, $\Theta(n^2)$. **Tabla:** limpia U casillas, siempre, y hace una pasada, $\Theta(U + n)$.

terminal, depende de la máquina

```
$ build/small_n
       n     pares (us)     tabla (us)  pares/tabla
      10           0.04           5.92        0.007
     100           2.38           6.54        0.363
    1000         185.46          11.79       15.727
    4000        2868.33          17.38      165.084
   16000       45519.33          33.21     1370.734
   32000      181941.17          45.92     3962.392
```

Se cruzan entre 100 y 1000, en esta máquina. “n grande” es parte de la afirmación.

Dónde se cruzan



Error común: "$O(n)$ siempre gana"

Error común

Decir que $O(n)$ siempre gana a $O(n^2)$. Solo lo garantiza desde algún n, y antes de eso puede perder.

- O es solo un techo: un $O(n^2)$ puede ser en realidad mucho mejor.
- Si sus datos son siempre pequeños, usen lo más simple.

En la biblioteca

`std::sort`, en g++, deja de partir el arreglo cuando quedan pedazos de 16 elementos o menos, y los termina con ordenamiento por inserción: $\Theta(n^2)$ en el peor caso, y más rápido con n tan pequeño.

La cache: no todas las operaciones cuestan lo mismo

Misma cantidad de lecturas, mismo $\Theta(n)$. Solo cambia el orden de los saltos.

terminal, depende de la máquina

```
$ build/cache_walk
         n       KiB      ordenado      al azar   razon
      4096        16       1.39 ns       1.40 ns     1.0
     65536       256       0.39 ns       5.68 ns    14.6
   1048576      4096       0.39 ns      15.13 ns    38.9
   4194304     16384       0.37 ns      22.55 ns    61.1
```

La cache: no todas las operaciones cuestan lo mismo

Misma cantidad de lecturas, mismo $\Theta(n)$. Solo cambia el orden de los saltos.

terminal, depende de la máquina

```
$ build/cache_walk
         n      KiB     ordenado      al azar   razon
      4096       16      1.39 ns      1.40 ns     1.0
     65536      256      0.39 ns      5.68 ns    14.6
   1048576     4096      0.39 ns     15.13 ns    38.9
   4194304    16384      0.37 ns     22.55 ns    61.1
```

- Si cabe en la cache, el orden da lo mismo.

La cache: no todas las operaciones cuestan lo mismo

Misma cantidad de lecturas, mismo $\Theta(n)$. Solo cambia el orden de los saltos.

terminal, depende de la máquina

```
$ build/cache_walk
         n      KiB     ordenado      al azar   razon
      4096       16      1.39 ns      1.40 ns     1.0
     65536      256      0.39 ns      5.68 ns    14.6
   1048576     4096      0.39 ns     15.13 ns    38.9
   4194304    16384      0.37 ns     22.55 ns    61.1
```

- Si cabe en la cache, el orden da lo mismo.
- Con 16 MiB, ir al azar cuesta decenas de veces más por salto, según la corrida.

La cache: no todas las operaciones cuestan lo mismo

Misma cantidad de lecturas, mismo $\Theta(n)$. Solo cambia el orden de los saltos.

terminal, depende de la máquina

```
$ build/cache_walk
         n      KiB     ordenado      al azar   razon
      4096       16      1.39 ns      1.40 ns     1.0
     65536      256      0.39 ns      5.68 ns    14.6
   1048576     4096      0.39 ns     15.13 ns    38.9
   4194304    16384      0.37 ns     22.55 ns    61.1
```

- Si cabe en la cache, el orden da lo mismo.
- Con 16 MiB, ir al azar cuesta decenas de veces más por salto, según la corrida.
- *O* predice cómo crece el costo. El reloj dice cuánto cuesta aquí y ahora.

SECCIÓN 11

Cierre

11

En el laboratorio

Sesión 1.

- Seis funciones misteriosas: predecir su clase leyendo el código, antes de correr nada.
- Escribir un medidor sobre `bench.h`: varios n, mediana de repeticiones, un CSV.
- El experimento de duplicar, y comparar contra lo que anotaron.

Sesión 2.

- `push_back` frente a `insert` al inicio.
- Búsqueda lineal frente a binaria sobre el mismo arreglo ordenado.
- El mismo experimento con `-O0` y con `-O2`: qué cambió y qué no.
- Una conclusión corta, con sus números.

> **Código**
>
> `code/week-03/lab/README.md`. Con `make lab` y `make test-lab`.

Para la próxima clase

- Terminen la sesión 1 del laboratorio y anoten sus predicciones antes de medir.
- Lean los apuntes de la semana 4: arreglos dinámicos y costo amortizado.
- Piensen: `push_back` salió barato en sus mediciones. Pero a veces el arreglo tiene que crecer y mover todo. ¿Cómo puede ser?
- Repaso: si un peor caso es $\Theta(n)$, ¿puede tardar menos en alguna entrada? Una función con dos llamadas a $n/2$ y trabajo constante, ¿cuántas llamadas hace?

Para recordar

Predecir con O, Ω y Θ. Medir con `-O2`, mediana y `keep`. Contrastar con las razones.