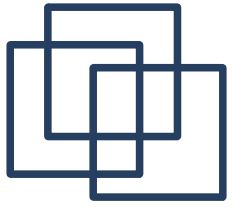


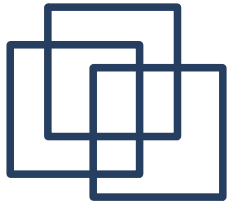
Tema 1

Introducción a la eficiencia de Algoritmos



Objetivos

- Concepto de eficiencia
- Concepto de operación elemental
- Notaciones asintóticas
- Saber calcular la eficiencia de un algoritmo



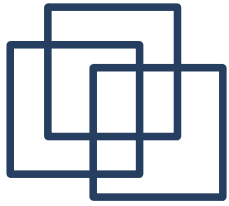
Motivación (I)

“ I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. ”

— Linus Torvalds (creator of Linux)

“ Algorithms + Data Structures = Programs. ”

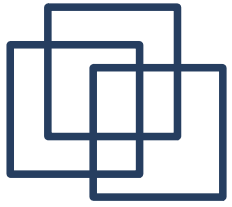
— Niklaus Wirth



Motivación (II)

- Todas las grandes empresas de internet se basan en estructuras de datos eficientes:
 - Búsqueda: Data mining
 - Selección: pagerank (Google: 200 millones de consultas al día)
- Ejemplo: suma de tres números igual a cero
tiempos en un PC con CPU 8 núcleos i7 2.8GHZ

	1000	2000	4000	5000	10000	100.000
t1(n)	0.03	0.12	0.50	0.82	3.42	???
t2(n)	0.68	5.35	42.79	80.36	671.17	???

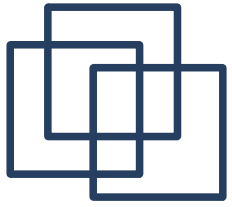


Comparación de algoritmos

- Es frecuente disponer de más de un posible algoritmo para resolver un problema dado:

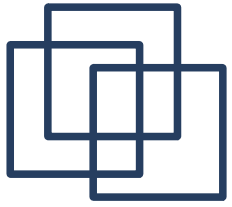
¿Con cuál nos quedamos?

- Estudio del uso de los recursos:
 - **tiempo**
 - espacio
- El uso de los recursos se distribuye en:
 - diseño
 - implementación
 - explotación



Factores en el uso de recursos

- Hardware: arquitectura, velocidad, etc.
- Diseñador: del algoritmo
- Codificador: Calidad del código creado
- Compilador: Calidad del código generado
- Tamaño de las entradas

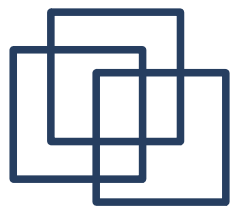


Principio de invarianza

Principio de Invarianza: dos implementaciones de un mismo algoritmo no diferirán más que en una constante multiplicativa.

Si $T_1(n)$ y $T_2(n)$ son los tiempos de dos implementaciones de un mismo algoritmo, se verifica:

$$\exists c, d \in \mathbb{R}, \quad T_1(n) \leq cT_2(n); \quad T_2(n) \leq dT_1(n)$$



Eficiencia

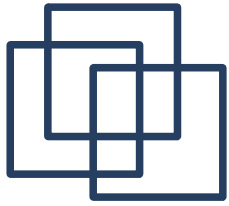
Eficiencia: medida del uso de los recursos en función del tamaño de las entradas

$T(n)$: Tiempo empleado para una entrada de tamaño n

Consideremos dos algoritmos distintos para un mismo problema:

- Algoritmo 1: $T(n) = 10^{-4} \times 2^n$ seg.

n	10	20	30	38
$T(n)$	0.1 s	2 m	> 1 día	1 año

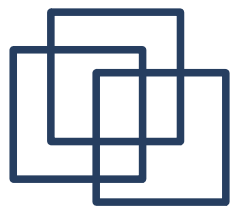


Eficiencia

- Algoritmo 2: $T(n) = 10^{-2} \times n^3$ seg.

n	200	1000
$T(n)$	1 día	1 año

- La mejora del hardware **NO** es suficiente
- Es necesario un **análisis asintótico**
- La elección de una buena estructura de datos es fundamental para crear un buen algoritmo

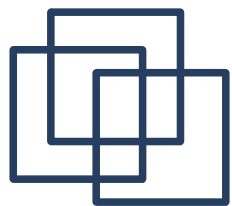


Órdenes de eficiencia ^(I)

Orden de Eficiencia: Un algoritmo tiene un *tiempo de ejecución de orden* $f(n)$, para una función dada f , si existe una constante positiva c y una implementación del algoritmo capaz de resolver cada caso del problema en un tiempo acotado superiormente por $cf(n)$, donde n es el tamaño del problema considerado.

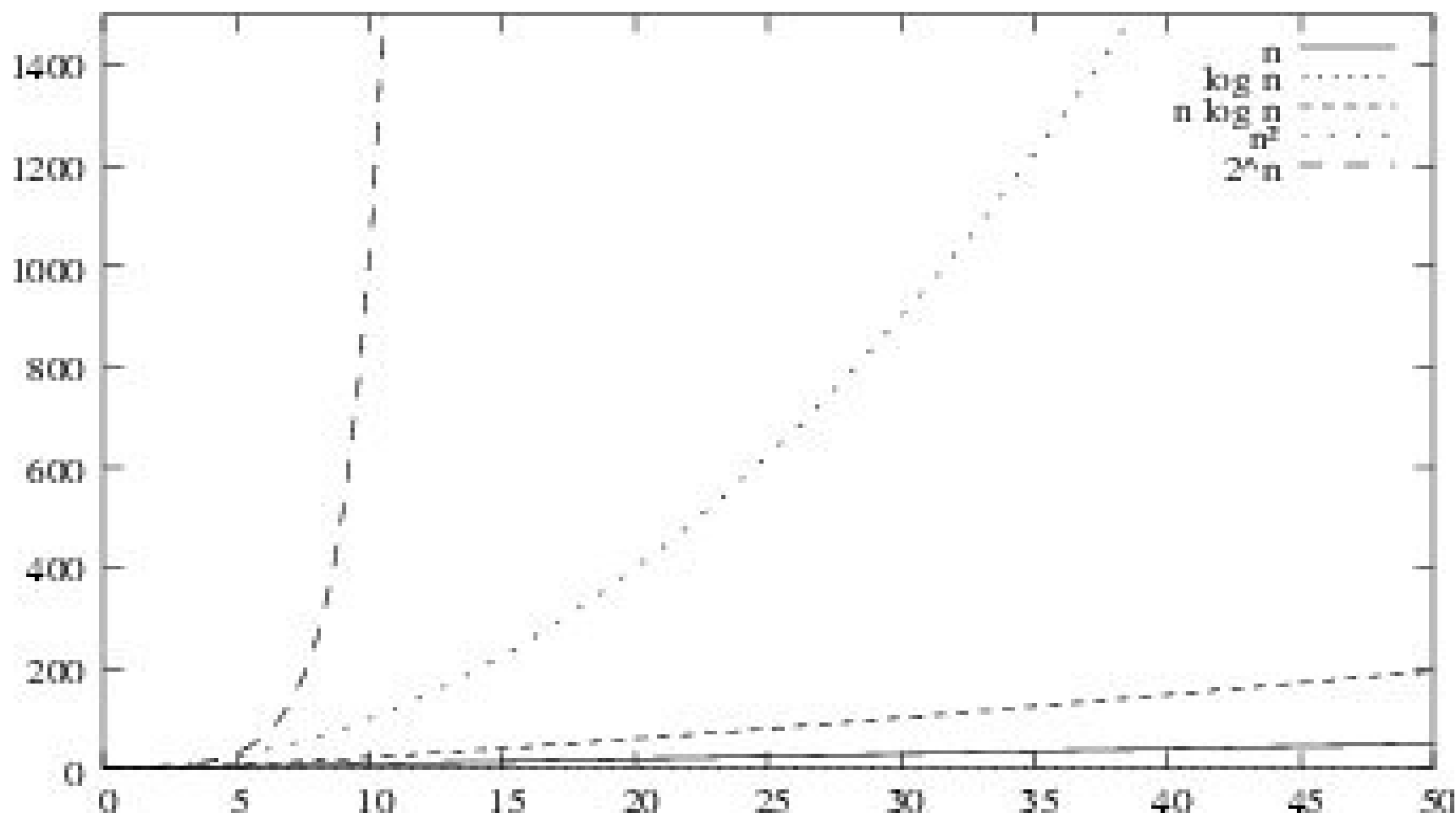
Órdenes más habituales:

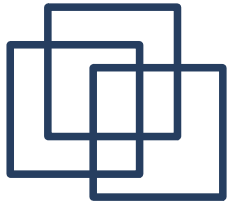
- lineal: n
- cuadrático: n^2
- Polinómico: n^k con $k \in \mathbb{N}$
- logarítmico: $\log(n)$
- exponencial: c^n , $c > 1$



Órdenes de eficiencia (II)

Comparación gráfica de algunos órdenes de eficiencia



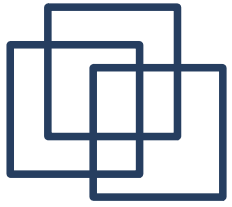


Medidas de la eficiencia

- Enfoque práctico (*a posteriori*)
- Enfoque teórico (*a priori*)
- Enfoque híbrido

Situaciones:

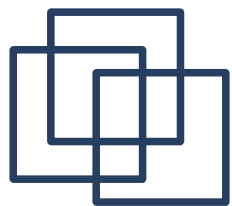
- Peor caso
- Caso promedio
- Mejor caso
- Análisis amortizado



Operación elemental

Definición: Operación de un algoritmo cuyo tiempo de ejecución se puede acotar superiormente por una constante.

Para nuestro análisis sólo contará el número de operaciones elementales ejecutadas y no el tiempo exacto necesario para cada una de ellas.



Operación elemental: Consideraciones

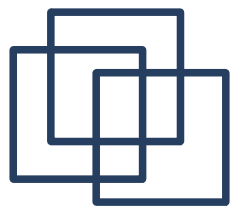
1. Línea de código = un número variable de operaciones elementales. P.e.: sea A un vector con n elementos y

$$x = \max\{A[k], 0 \leq k < n\}$$

el tiempo para calcular x depende de n , no es constante.

2. Algunas operaciones matemáticas no son operaciones elementales. P.e.: sumas y productos dependen de la longitud de los operandos. Sin embargo, en la práctica se consideran elementales siempre que los datos que se usen tengan un tamaño razonable.

En la práctica, consideraremos las operaciones suma, diferencia, multiplicación, división, módulo, operaciones booleanas, comparativas y asignaciones como elementales, (salvo que se indique otra cosa).



Notación O Mayúscula

Decimos que una función $T(n)$ es $O(f(n))$ si existen constantes n_0 y c tales que $T(n) \leq cf(n)$ para $n \geq n_0$:

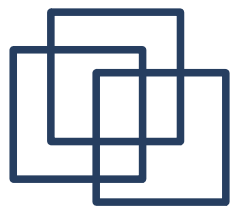
$$T(n) \text{ es } O(f(n)) \iff \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \text{ t.q.}$$

$$\forall n \geq n_0 \in \mathbb{N}, T(n) \leq cf(n)$$

Se trata de una *cota superior*.

Ejemplos:

- $T(n) = (n + 1)^2$ es $O(n^2)$
- $T(n) = 3n^3 + 2n^2$ es $O(n^3)$
- $T(n) = 3^n$ no es $O(2^n)$

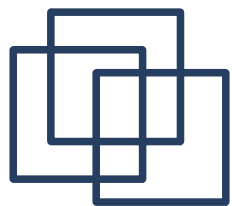


Notación O Mayúscula (II)

Flexibilidad en la notación:

- Emplearemos $O(f(n))$ aun cuando en un número finito de valores de n , $f(n)$ sea negativa o no esté definida.

Ej.: $n / \log_2 n$ no está definida para $n = 0$ ó $n = 1$.



Notación Ω Omega

Decimos que una función $T(n)$ es $\Omega(f(n))$ si existen constantes n_0 y c tales que $T(n) \geq cf(n)$ para $n \geq n_0$:

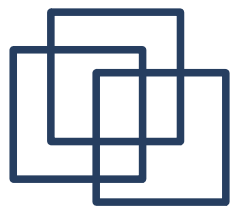
$$T(n) \text{ es } \Omega(f(n)) \iff \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \text{ t.q.}$$

$$\forall n \geq n_0 \in \mathbb{N}, T(n) \geq cf(n)$$

Es una *cota inferior*.

Ejemplos:

- $T(n) = (n + 1)^2$ es $\Omega(n^2)$
- $T(n) = 3n^3 + 2n^2$ es $\Omega(n^3)$
- $T(n) = 3^n$ es $\Omega(2^n)$

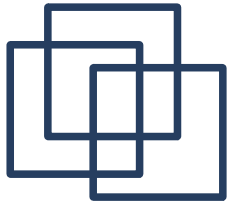


Notación Ω Omega _(II)

Flexibilidad en la notación:

- Emplearemos $\Omega(f(n))$ aun cuando en un número finito de valores de n , $f(n)$ sea negativa o no esté definida.

Ej.: $n / \log_2 n$ no está definida para $n = 0$ ó $n = 1$.



Notación Θ Theta

Cuando se puede acotar el tiempo de ejecución de un algoritmo tanto superior como inferiormente por la misma función, estamos ante un caso especial. Entonces se usa la notación Theta, Θ .

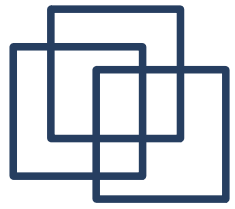
Si se cumple

$$T(n) \text{ es } O(f(n))$$

$$T(n) \text{ es } \Omega(f(n))$$

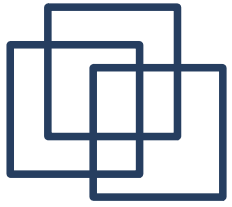
entonces se dice que $T(n)$ es orden exacto de $f(n)$ y se escribe:

$$T(n) \text{ es } \Theta(f(n))$$



Reglas de cálculo del tiempo de ejecución

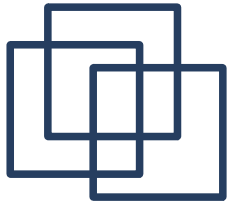
- Sentencias simples
- Bucles
- Sentencias condicionales
- Bloques de sentencias
- Llamadas a funciones



Sentencias simples

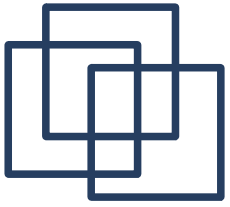
Consideraremos que cualquier sentencia simple (lectura, escritura, asignación,...) va a consumir un tiempo constante, $O(1)$, salvo que la sentencia contenga una llamada a función.

```
1:      for (i = 0; i < n-1; i++) {
2:          indice = i;
3:          for (j = i+1; j < n; j++) {
4:              if (A[j] < A[indice])
5:                  indice = j;
6:              temporal = A[indice];
7:              A[indice] = A[i];
8:              A[i] = temporal;
9:          };
10:     }
```



Bucles

Tiempo para un bucle: **la suma del tiempo invertido en cada iteración**. Debería incluir: el tiempo propio del cuerpo y el asociado a la evaluación de la condición y su actualización, en su caso. Si se trata de una condición sencilla (sin llamadas a función) el tiempo es $O(1)$. Cuando se trata de un bucle donde todas las iteraciones son iguales, entonces el tiempo total será el producto del número de iteraciones por el tiempo que requiere cada una.



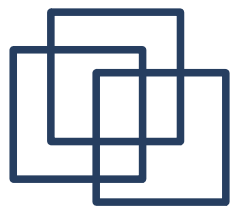
Bucles

Ejemplo 1:

```
1:      for (i = 0; i < n; i++)
2:          for (j = 0; j < n; j++)
3:              A[i][j] = 0;
```

Ejemplo 2:

```
1:      k = 0;
2:      while (k < n && A[k] != Z)
3:          k++;
```

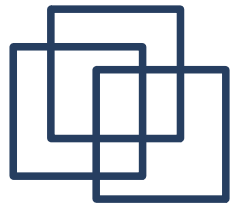


Sentencias condicionales

El tiempo de ejecución es el máximo tiempo de la parte *if* y de la parte *else*, de forma que si son, respectivamente, $O(f(n))$ y $O(g(n))$ será $O(\max(f(n), g(n)))$

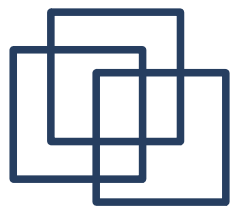
Ejemplo:

```
1:      if (A[0][0] == 0)
2:          for (i = 0; i < n; i++)
3:              for (j = 0; j < n; j++)
4:                  A[i][j] = 0;
5:      else
6:          for (k = 0; k < n; k++)
7:              A[k][k] = 1;
```

Bloques de sentencias

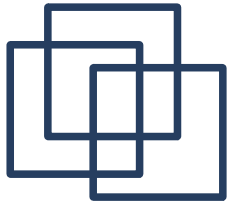
- Se aplica la regla de la suma, de forma que se calcula el tiempo de ejecución tomando el máximo de los tiempos de ejecución de cada una de las partes (sentencias individuales, bucles o condicionales) en que puede dividirse.



Llamadas a funciones _(I)

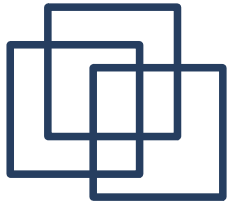
Si una determinada función P tiene una eficiencia de $O(f(n))$ con n la medida del tamaño de los argumentos, cualquier función o procedimiento que llame a P tiene en la llamada una cota superior de eficiencia de $O(f(n))$.

- Las asignaciones con diversas llamadas a función deben sumar las cotas del tiempo de ejecución de cada llamada.
- La misma consideración es válida para las condiciones de bucles y condicionales.



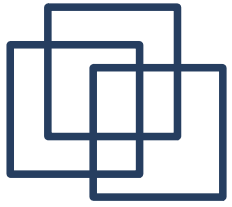
Llamadas a funciones (II)

```
1:      int funcion1(int n) {
2:          int i, x;
3:          for (i = 0, x = 0; i < n; i++)
4:              x += funcion2(i, n);
5:          return x;
6:      }
7:
8:      int funcion2(int x, int n) {
9:          int i;
10:         for (i = 0; i < n; i++)
11:             x += i;
12:         return x;
13:     }
```



Llamadas a funciones (III)

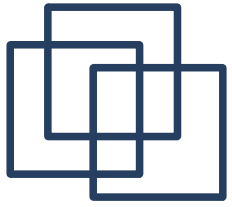
```
20:
21:     int main()
22:     {
23:         int control, n;
24:
25:         cin >> n;
26:         control = funcion1(n);
27:         cout << control << endl;
28:
29:         return 1;
30:     }
```



Ejemplo: Potenciación

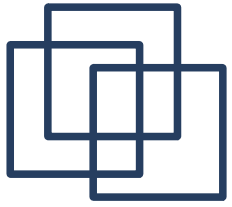
```
1: void ejemplo1 (int n) {  
2:     int x, contador;  
3:     contador = 0;  
4:     x = 2;  
5:     while (x <= n) {  
6:         x = 2 * x;  
7:         contador++;  
8:     }  
9:     cout << contador;  
10: }
```

- Las líneas 3, 4, 6, 7 y 9 son operaciones elementales: $O(1)$.
- Las líneas 5-8 componen un bucle que se realiza $\log_2(n)$ veces y su cuerpo es $O(1)$, con lo que el bloque completo es $O(\log_2(n))$.



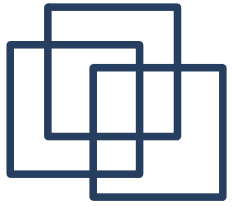
Ejemplo: Selección _(I)

```
1: void seleccion(int A[], int n) {
2:     int i, j, min, t;
3:     for (i = 0; i < n - 1; i++) {
4:         min = i;
5:         for (j = i + 1; j < n; j++)
6:             if (A[j] < A[min])
7:                 min = j;
8:         t = A[min];
9:         A[min] = A[i];
10:        A[i] = t;
11:    }
12: }
```



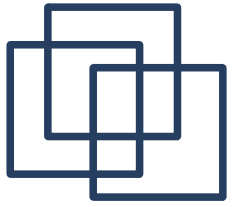
Ejemplo: Selección (II)

$$\begin{aligned}\sum_{i=0}^{n-2} c(n-i) &= c \sum_{i=0}^{n-2} n - c \sum_{i=0}^{n-2} i = n(n-1) - \frac{n}{2}(n-1) = \\ &= \frac{n(n-1)}{2} \implies \boxed{O(n^2)}\end{aligned}$$



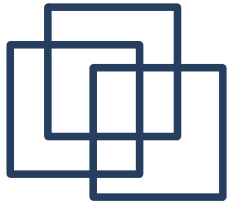
Ejemplo: Inserción _(I)

```
1: void insercion(int A[], int n) {
2:     int i, j, valor;
3:     for (i = 1; i < n; i++) {
4:         valor = A[i];
5:         j = i;
6:         while ((j > 0) && (A[j-1] > valor)) {
7:             A[j] = A[j-1];
8:             j--;
9:         }
10:        A[j] = valor;
11:    }
12: }
```

Ejemplo: Inserción_(II)

$$\begin{aligned}\sum_{i=1}^{n-1} (i - 1) &= \sum_{i=1}^{n-1} i - \sum_{i=1}^{n-1} 1 = \frac{n}{2}(n - 1) - (n - 1) = \\ &= \frac{(n - 2)(n - 1)}{2} \implies \boxed{O(n^2)}\end{aligned}$$



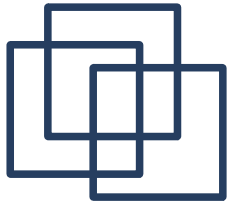
Ejemplo _(I)

```
1: void ejemplo2(int n) {  
2:     int i, j, k;  
3:     for (i = 1; i < n; i++)  
4:         for (j = i+1; j < n + 1; j++)  
5:             for (k = 1; k < j + 1; k++)  
6:                 /* Alguna sentencia O(1) */  
7: }
```

Líneas 5-6: j veces

Líneas 4-6:

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

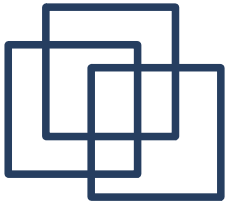


Ejemplo _(II)

Líneas 5-8:

$$\begin{aligned} \frac{1}{2} \sum_{i=1}^{n-1} [n^2 + n - i^2 - i] &= \frac{1}{2} \left(\sum_{i=1}^{n-1} n^2 + \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i^2 - \sum_{i=1}^{n-1} i \right) = \\ &= \frac{1}{2} \left(n^2(n-1) + n(n-1) - \frac{(n-1)n(2n-1)}{6} - \frac{n(n-1)}{2} \right) \end{aligned}$$

$$\boxed{O(n^3)}$$



Otro ejemplo

```
1: void ejemplo3(int n) {
2:     int i, j, x, y;
3:     for (i = 1; i < n+1; i++)
4:         if (i % 2 == 0) {
5:             for (j = i; j < n+1; j++)
6:                 x++;
7:             for (j = 1; j < i+1; j++)
8:                 y++;
9:         }
10: }
```