Análisis de la eficiencia de algoritmos

Joaquín Fernández-Valdivia
Javier Abad
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada

DECSAI

Motivación

- Podemos diseñar diferentes algoritmos para resolver un problema.
- Cada uno de ellos puede consumir más o menos recursos, tiempo... ¿Cuál de ellos es el mejor?
- Ejemplo: asignar 5 trabajadores a 5 trabajos distintos, según sus capacidades. Fácil, ¿verdad?
- Pero, ¿y si tenemos que asignar 50 trabajadores a 50 puestos? ¡¡Serían 50! Posibilidades!!

¡¡Aproximadamente, 10^{64}!!

- ¡¡¡Si un ordenador evalúa 1 billón de posibilidades por segundo, necesitará más de 15000 millones de años!!!

Tamaño del problema

- Un algoritmo no tiene un tiempo de ejecución fijo. Éste depende del tamaño del problema.
- Si queremos comparar dos algoritmos, no podemos hacerlo para un único tamaño de problema.
- Debemos expresar el tiempo de ejecución como una función, f(n), del tamaño del problema, n.

$$f : N \to \mathbb{R}_0^+$$
$$n \to f(n)$$

- Para comparar algoritmos, comparamos sus tiempos de ejecución (compararemos sus funciones).

Tamaño del problema

Tamaño	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
10	3.3 ns	10 ns	33 ns	100 ns	1 µs	1 µs
20	4.3 ns	20 ns	86 ns	400 ns	8 µs	1 ms
30	4.9 ns	30 ns	147 ns	900 ns	27 µs	1 s
40	5.3 ns	40 ns	213 ns	2 µs	64 µs	18.3 min
50	5.6 ns	50 ns	282 ns	3 µs	125 µs	13 días
100	6.6 ns	100 ns	664 ns	10 µs	1 ms	40×10^{12} años
1000	10 ns	1 µs	10 µs	1 ms	1 s	
10000	13 ns	10 µs	133 µs	100 ms	16.7 min	
100000	17 ns	100 µs	2 ms	10 s	11.6 días	
1000000	20 ns	1 ms	20 ms	16.7 min	31.7 años	

Tiempos de ejecución en una máquina que realiza 10^9 pasos por segundo (1GHz), según el tamaño del problema y el coste del algoritmo

Algoritmos versus implementaciones

Debemos distinguir entre:

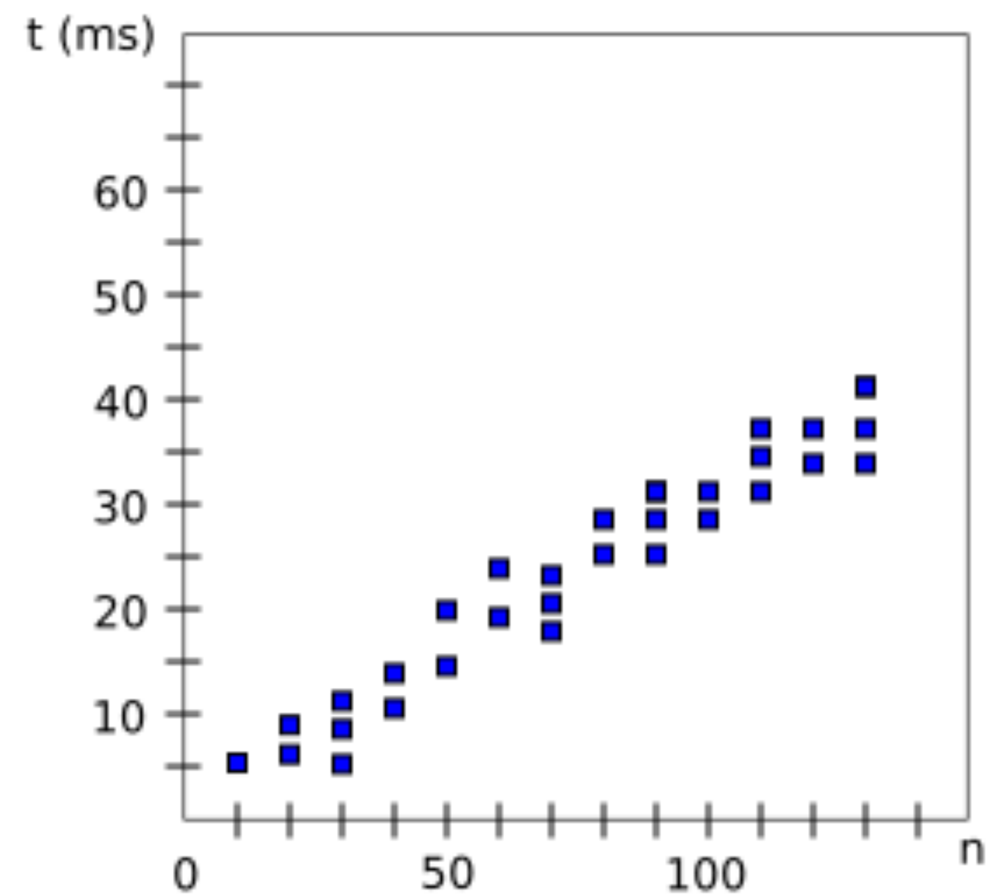
- **Algoritmo**: conjunto finito de pasos que nos llevan a resolver un problema.
- **Implementación**: realización de un algoritmo en un lenguaje de programación determinado

CENTRAREMOS NUESTRO ANÁLISIS EN ALGORITMOS, Y NO EN IMPLEMENTACIONES

Medición experimental del tiempo de ejecución

Estudio experimental:

- Escribir un programa que implemente el algoritmo
- Ejecutar el programa con conjuntos de datos de diferente tamaño y composición
- Usar algún método para tomar una medida adecuada del tiempo de ejecución



Más allá de los estudios experimentales

Limitaciones de los estudios experimentales:

- Es necesario implementar y testar el algoritmo
- Los experimentos se hacen sobre un conjunto limitado de entradas, que pueden no ser suficientemente representativas
- Para comparar dos algoritmos, deben usarse los mismos entornos hardware y software

Más allá de los estudios experimentales

Por lo tanto,

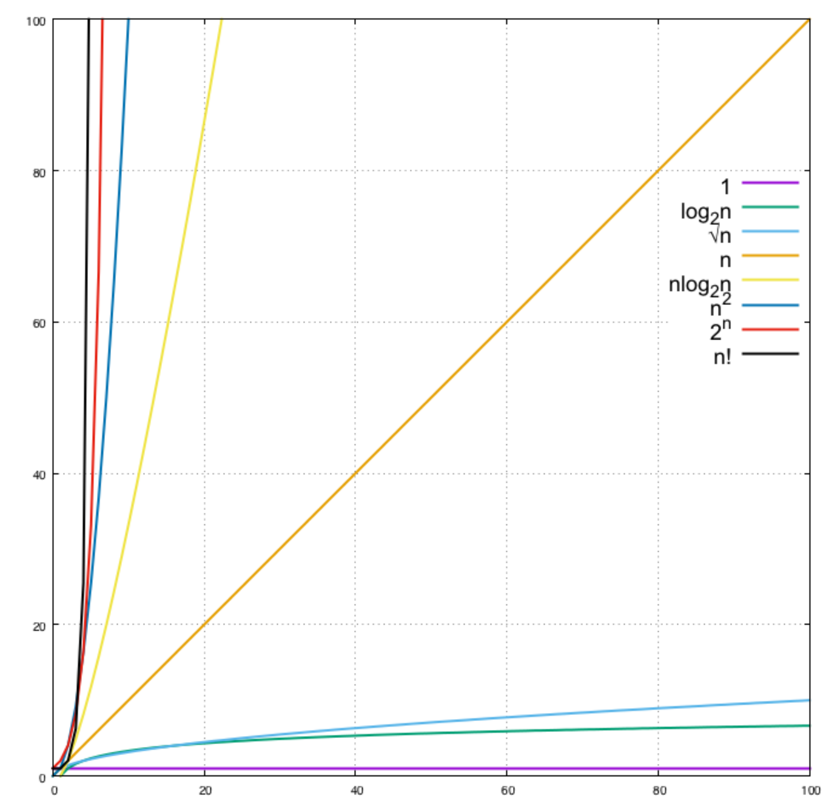
- Se desarrollará una metodología general para analizar el tiempo de ejecución de un algoritmo que:
  - Usa la descripción de alto nivel del algoritmo
  - Tiene en cuenta todas las posibles entradas
  - Permite evaluar la eficiencia de un algoritmo con independencia del hardware y del software

EFICIENCIA TEÓRICA vs. EXPERIMENTAL O EMPÍRICA

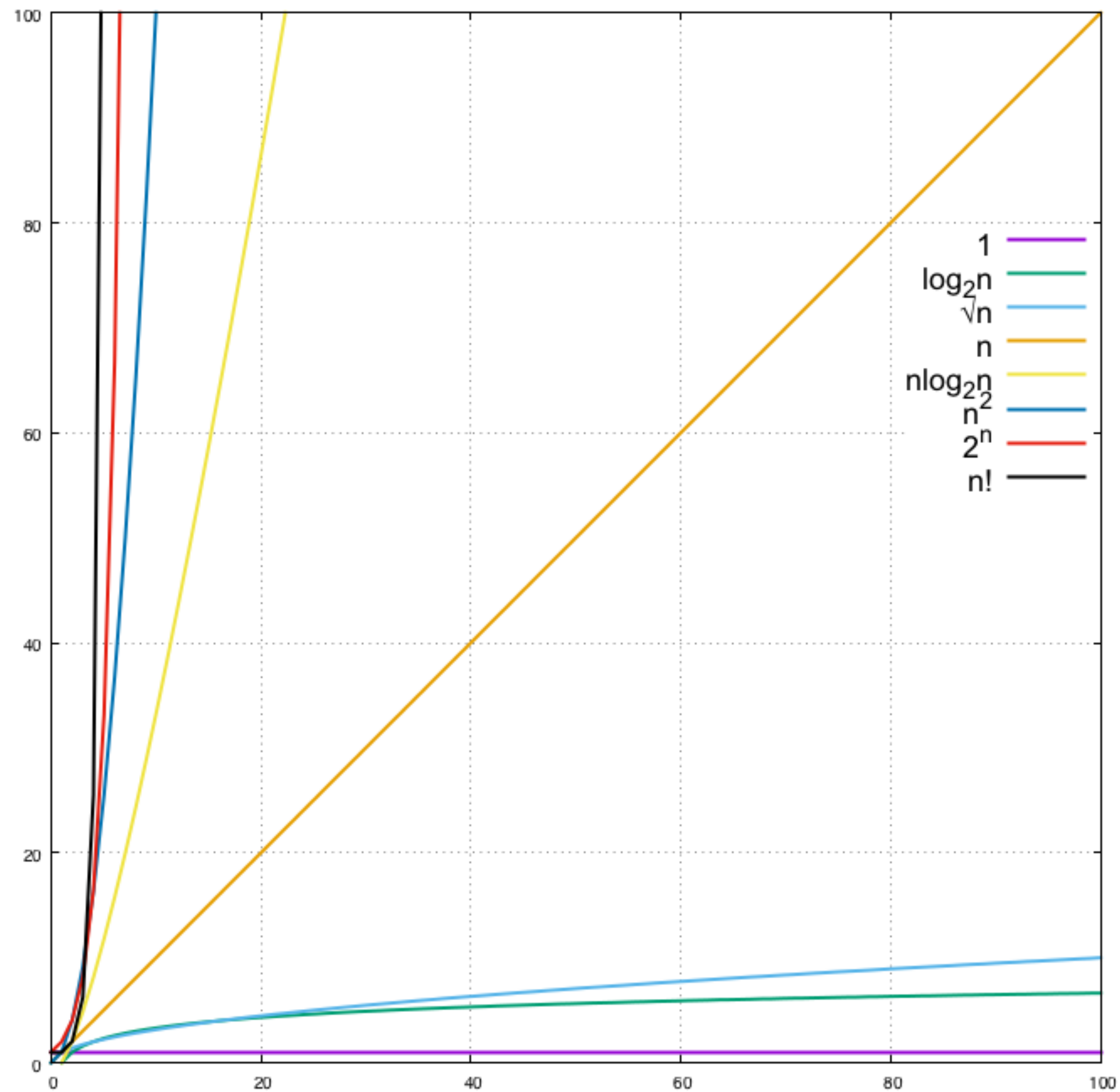
Familias de órdenes de eficiencia

- *Un algoritmo tiene un* ***tiempo de ejecución*** *t(n) si existe una constante positiva, c, y una implementación del algoritmo capaz de resolver cualquier ejemplo del problema en un tiempo acotado por ct(n), siendo n el tamaño del problema*

- Algunos tiempos de ejecución habituales:
  - n: lineal
  - n^2: cuadrático
  - n^k: polinómico (k natural)
  - log n: logarítmico
  - c^n: exponencial



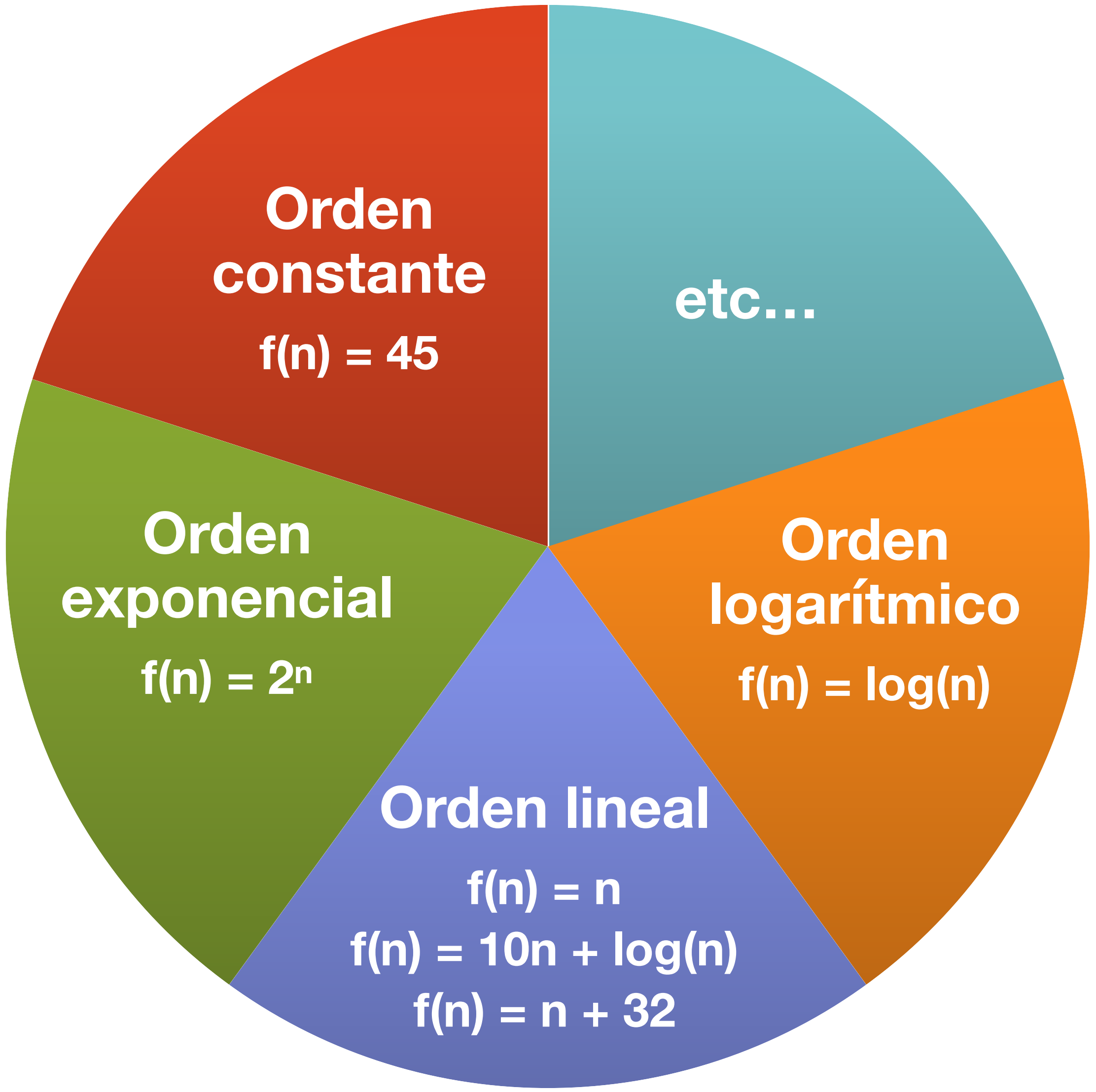
Familias de órdenes de eficiencia



Familias de órdenes de eficiencia

Cuando queramos estudiar el tiempo de ejecución de un algoritmo independientemente de la implementación, tendremos que determinar la clase de equivalencia a la que pertenece la función de su tiempo de ejecución, esto es, determinar su

ORDEN DE EFICIENCIA



Comparación de órdenes de eficiencia

- La determinación de la eficiencia de un algoritmo dependerá del comportamiento de su orden de eficiencia cuando n crece, esto es, cuando $n \longrightarrow \infty$

- Comparamos **perfiles de crecimiento**: un algoritmo es más eficiente que otro si su perfil de crecimiento tiene un crecimiento menor

- Condiciones para comparar dos órdenes de eficiencia:
  - El resultado no puede depender del resultado para un número finito de valores de la función
  - El resultado no puede depender de las funciones concretas que representan las correspondientes clases

Comparación de órdenes de eficiencia

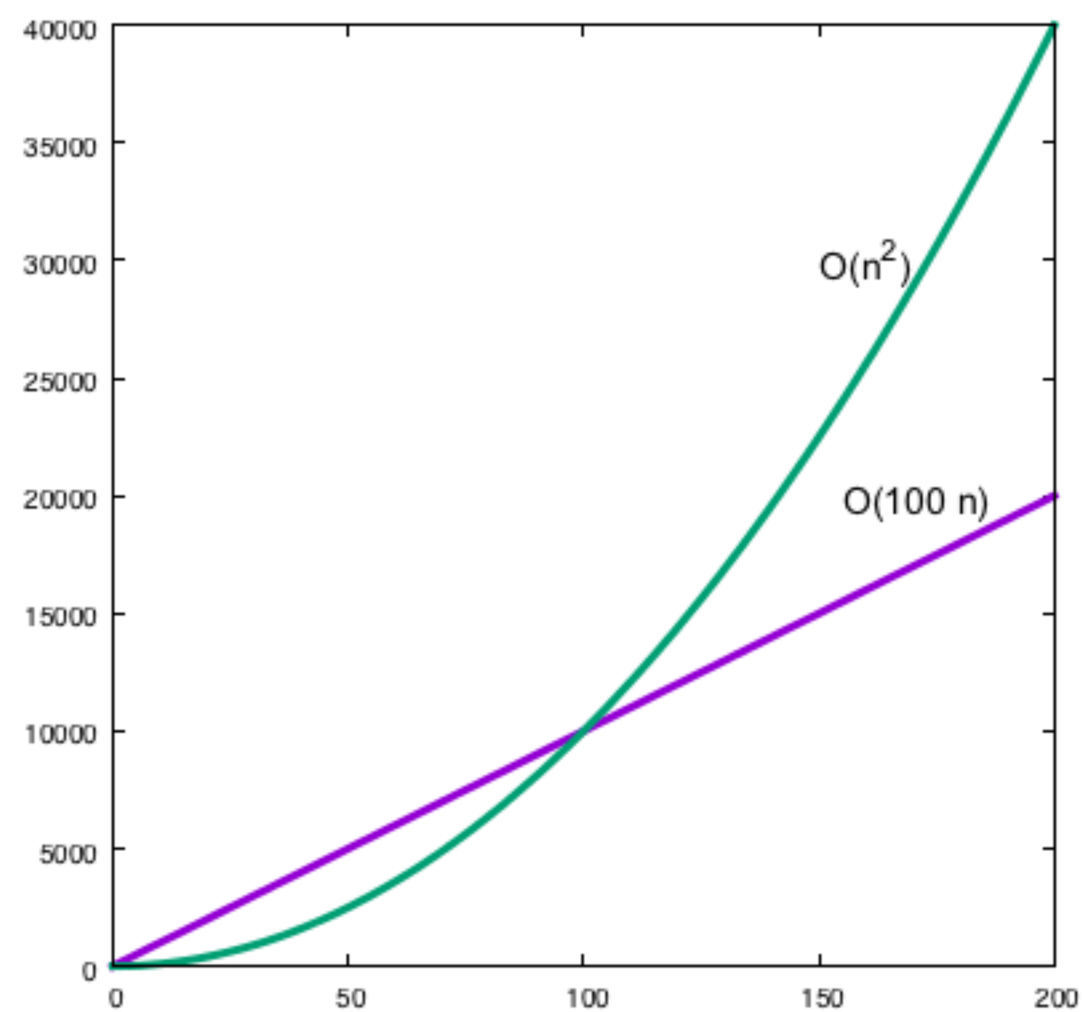
- Dadas f, g: N → R^+, diremos que g(n) es menor o igual que f(n) si

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \text{ tales que } \forall n \geq n_0, \quad g(n) \leq c \cdot f(n)$$

g(n) = 100n es menor que f(n) = n^2, ya que para c=100 y n_0=1 se cumple que

$$\forall n \geq 1, 100 \cdot n \leq 100 \cdot n^2$$

Este orden es sólo parcial, ya que habrá parejas de funciones que no podrán ordenarse



Notación O

- Objetivo: poder indicar de forma clara y precisa el grado de eficiencia obtenido en el análisis del algoritmo

- Pasos:

  1. Análisis del tiempo de ejecución: estudiar la función que indica el tiempo necesario para cada n

  2. Análisis de eficiencia: clasificar ese tiempo de ejecución en una familia de funciones

Si tenemos ordenadas las familias de funciones, podremos comparar algoritmos

Notación O

- Objetivo: eliminar información innecesaria, simplificando el análisis. Ej.: $3n^2 + 5 \simeq n^2$

NOTACIÓN O GRANDE (Big O)

- Dadas dos funciones, f(n) y g(n), decimos que f(n) es O(g(n)) sii $f(n) \leq cg(n)$ para $n \geq n_0$, con c y n_0 constantes

  Ej.: $(n+1)^2$ es $O(n^2)$ [$n_0=1$, $c=4$]

  Ej.: $3n^3+2n^2$ es $O(n^3)$ [$n_0=0$, $c=5$]

  Ej.: 3^n no es $O(2^n)$

- Aunque es cierto que, por ejemplo, $7n+5$ es $O(n^5)$, la idea es buscar el menor orden posible, esto es, la menor cota superior posible

Notación O

- Una primera regla muy sencilla: *eliminar los términos de orden menor y los factores constantes*. Así:
  - $7n-3$ es $O(n)$
  - $8n^2\log_2 n+5n^2+n$ es $O(n^2\log_2 n)$
  - $3n^2+2n$ es $O(n^2)$
  - $n/2 + \log_2 n$ es $O(n)$
  - 225 es $O(1)$

- Algunos casos especiales:
  - Logarítmico $O(\log n)$
  - Lineal $O(n)$
  - Cuadrático $O(n^2)$
  - Polinomial $O(n^k)$, $k>1$
  - Exponencial $O(a^n)$

En resumen...

- La notación O nos permite acotar y caracterizar el tiempo de ejecución de un algoritmo de forma independiente de la máquina, la implementación, etc.
- El signo $\leq$ implica la idea de eficiencia en el peor caso
- Aunque, p.ej., $2n^2+3$ es $O(n^2)$ y también es $O(n^3)$, la primera es mejor, ya que caracteriza mejor la función
- La notación O nos permite expresar el número de operaciones primitivas en función del tamaño del problema
- La notación O nos permitirá comparar tiempos de ejecución: $O(n)$ es mejor que $O(n^2)$, ó $O(\log n)$ es mejor que $O(n)$…
  ➔ JERARQUÍA DE FUNCIONES

Jerarquía de funciones

Logarítmicas

Sublineales

Constantes

Lineales

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset \cdots$$

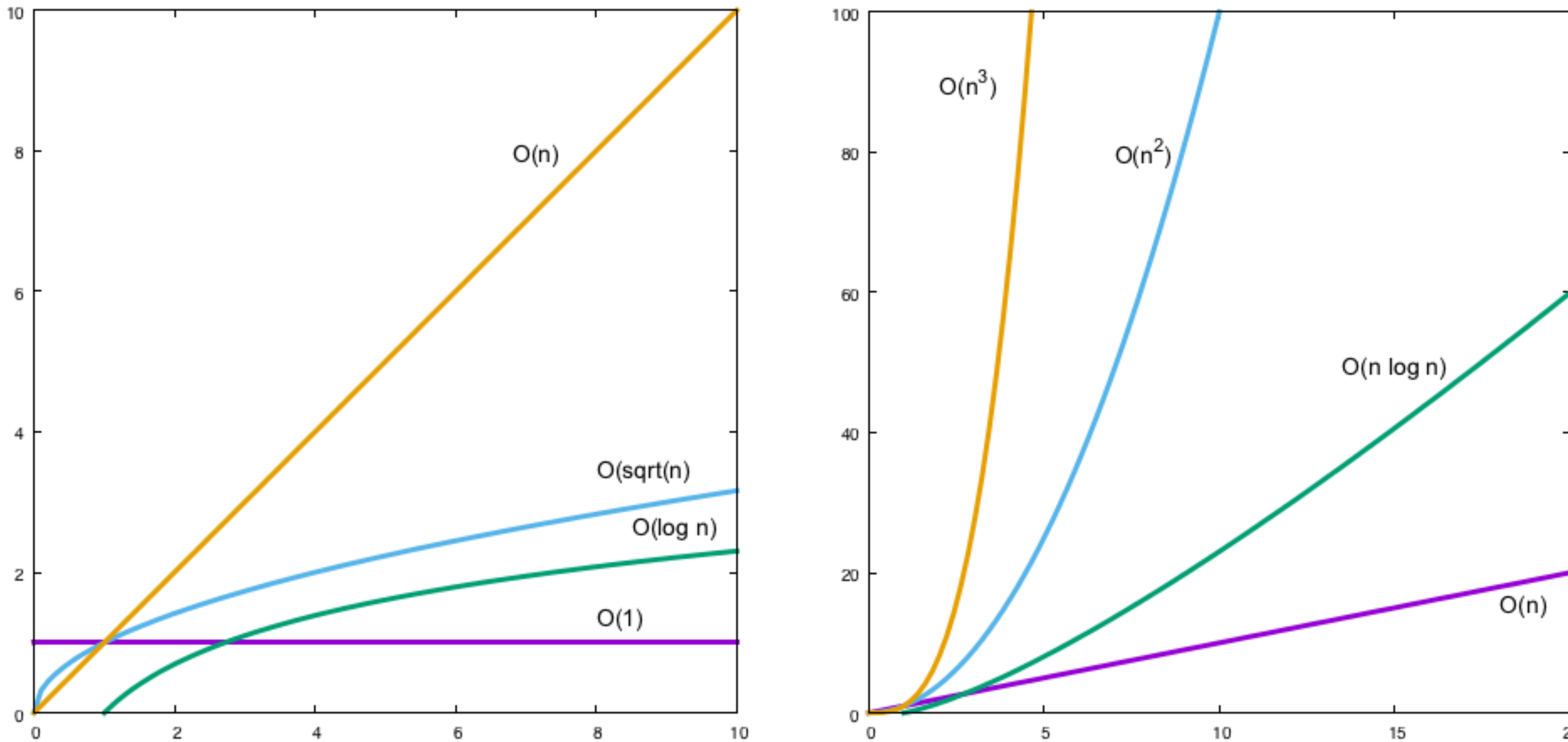
$$\cdots \subset O(n \cdot \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n^n)$$

Superlineales

Exponenciales

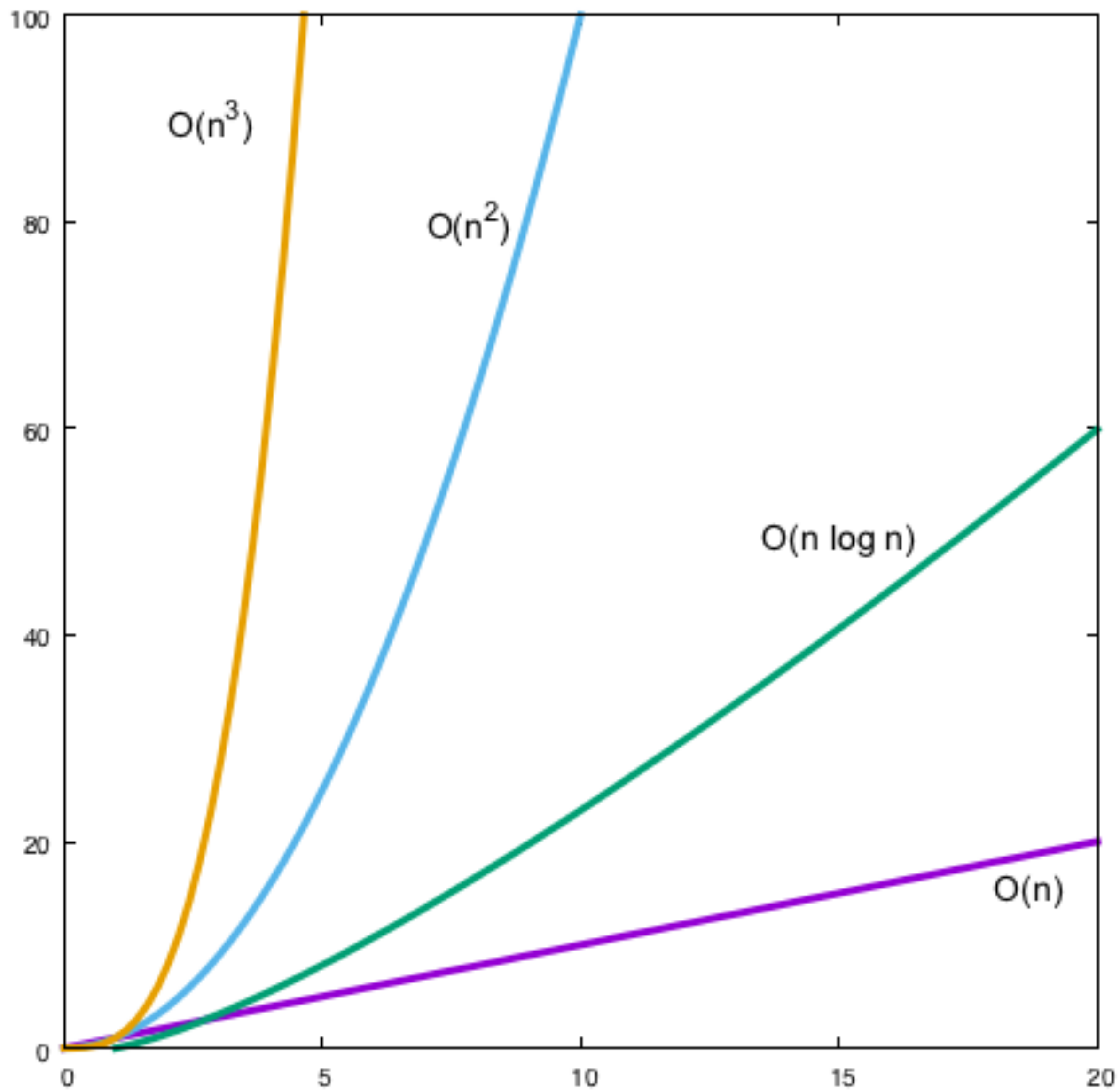
Polinómicas

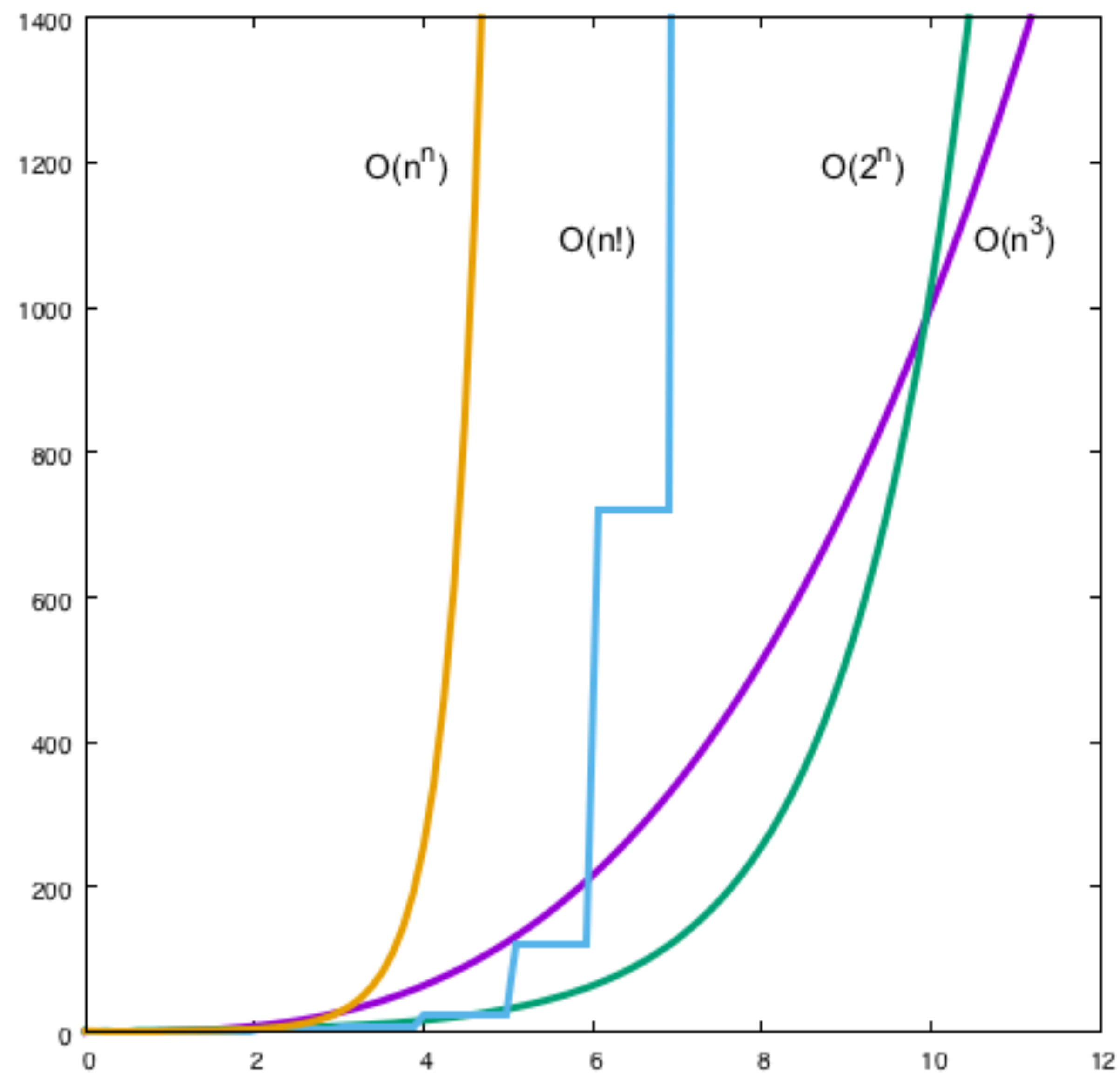
Jerarquía de funciones



Sublineales, lineales y superlineales

Jerarquía de funciones





Superlineales, polinómicas y exponenciales

Algunas reglas simples...

- Transitividad

$$\left.\begin{array}{l} f(n) \text{ es } O(g(n)) \\ g(n) \text{ es } O(h(n)) \end{array}\right\} f(n) \text{ es } O(h(n))$$

- Polinomios $a_d n^d + \cdots + a_1 n + a_0$ es $O(n^d)$

- Jerarquía de funciones

$n + log_2 n$ es $O(n)$ $\qquad$ $2^n + n^3$ es $O(2^n)$

- Bases y potencias de logaritmos pueden ignorarse

$\log_a n$ es $O(\log_b n)$ $\qquad$ $\log(n^2)$ es $O(\log n)$

- Bases y potencias de exponentes no pueden ignorarse

3^n no es $O(2^n)$ $\qquad$ a^{n^2} no es $O(a^n)$

Elección del mejor algoritmo

- En la práctica, debemos tener en cuenta otros factores, además de la eficiencia:
  - Tamaño de los problemas a resolver
  - Requisitos de espacio y tiempo del sistema
  - Complejidad de implantación y mantenimiento de los algoritmos
- Ejemplo:
  - Algoritmo 1 con $t_1(n) = 100n$ Lineal $[O(n)]$
  - Algoritmo 2 con $t_2(n) = n^2/5$ Cuadrático $[O(n^2)]$

En teoría está claro, pero ¿y en la práctica?

Elección del mejor algoritmo

- El segundo algoritmo, cuadrático, puede ser preferible si:
  - El tamaño de los problemas a resolver no va a pasar de 100 [¡constante multiplicativa!]
  - El algoritmo 1 tiene unos requerimientos de memoria muy superiores. P.ej., si el algoritmo 1 requiere una cantidad de espacio cuadrática respecto al tamaño, mientras que el algoritmo 2 tiene requerimiento constante
  - El algoritmo 1 tiene costes de implementación y mantenimiento muy superiores al algoritmo 2

Notación O

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

$$T(n) \text{ es } O(f(n)) \iff \exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N} \text{ tq } \forall n \geq n_0, T(n) \leq c \cdot f(n)$$

- Seremos flexibles en la notación: Usaremos O(f(n)) aun cuando en un número finito de valores de n, f(n) no esté definida o sea negativa. Ej: logaritmos

Notación O

- **Regla de la suma**

Si $T_1(n)$ y $T_2(n)$ son los tiempos de ejecución de dos segmentos de código tales que $T_1(n)$ es $O(f(n))$ y $T_2(n)$ es $O(g(n))$, entonces

$$T_1(n) + T_2(n) \text{ es } O(\max(f(n),g(n)))$$

- **Regla del producto**

Si $T_1(n)$ y $T_2(n)$ son los tiempos de ejecución de dos segmentos de código tales que $T_1(n)$ es $O(f(n))$ y $T_2(n)$ es $O(g(n))$ (y no son negativos para ningún n), entonces

$$T_1(n)\, T_2(n) \text{ es } O(f(n)g(n))$$

Cálculo de la eficiencia

- **Operación elemental:** operación de un algoritmo cuyo tiempo de ejecución se puede acotar superiormente por una constante

- **Nos interesa** el número de operaciones elementales, no el tiempo concreto requerido para ejecutarlas

- **No nos interesa** saber la función exacta que indica el tiempo de ejecución, sino cualquiera que corresponda a la misma eficiencia, al mismo orden. Esto simplificará enormemente nuestro análisis.

Cálculo de la eficiencia

- Ejemplo: un algoritmo con
  - n_s sumas (t_s)
  - n_a asignaciones (t_a)
  - n_m multiplicaciones (t_m)

Podemos ver que

$$t \geq \min(t_s, t_a, t_m) \cdot (n_s + n_a + n_m)$$

y

$$t \leq \max(t_s, t_a, t_m) \cdot (n_s + n_a + n_m)$$

Por lo tanto, $c_1 \cdot n \leq t \leq c_2 \cdot n$

Pero, ¡si las constantes no importan!
Por lo tanto, el algoritmo es $O(n)$

Cálculo de la eficiencia

- **Estructura secuencial:** Si S_1 y S_2 son dos segmentos de código con eficiencias $O(f_1(n))$ y $O(f_2(n))$, la eficiencia de su unión secuencial es $O(f_1(n) + f_2(n))$.

  Por la regla de la suma, tenemos $O(\text{máx}(f_1(n), f_2(n)))$.

- **Estructura condicional:** Si la evaluación de la condición requiere un tiempo $O(E(n))$ y el camino más costoso requiere un tiempo $O(f(n))$, la eficiencia total de la estructura condicional es la suma de ambas.

  Por la regla de la suma, tenemos $O(\text{máx}(E(n), f(n)))$.

Cálculo de la eficiencia

- **Estructura iterativa:** Debemos tener en cuenta el tiempo invertido en la inicialización, la evaluación de la condición y en la actualización, si procede. En muchos casos, son O(1).
  - Bucle for: O(Ini(n)) + O(Con(n)) + O(Ite(n)) · [O(Cu(n)) + O(Inc(n)) + O(Con(n))]
  - Bucle while: O(Con(n)) + O(Ite(n)) · [O(Cu(n)) + O(Con(n))]
  - Bucle do-while: O(Ite(n)) · [O(Cu(n)) + O(Con(n))]

donde:

O(Ini(n)): Inicialización

O(Con(n)): Condición

O(Ite(n)): Iteración

O(Cu(n)): Cuerpo

O(Inc(n)): Incremento

Cálculo de la eficiencia

- Ejemplo:

```
for (i=0; i<n; i++)
  A[i][j] = 0;
```

1+1+ 4n operaciones

1+ 1n operaciones

- Antes de entrar en el bucle:
  - 1 asignación
  - 1 evaluación de condición
- En cada iteración
  - 1 indexación
  - 1 asignación
  - 1 incremento
  - 1 evaluación de condición

A efectos prácticos, en la evaluación de la complejidad, nos da igual que se ejecuten 4 operaciones en cada iteración o que se ejecute sólo 1 ➢ O(n)

Cálculo de la eficiencia

- Ejemplo:

```
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    A[i][j] = 0;
```

$$\left] \sum_{i=0}^{n} 1 = n = n \times 1 \right] \sum_{i=0}^{n} n = n^2 = n \times n$$

- En muchos casos, la inicialización, la condición y el incremento son operaciones simples, cuyo tiempo es O(1), lo que simplifica mucho el análisis
- Cuando se trata de bucles simples (todas las iteraciones son iguales), el tiempo total es el producto del número de iteraciones por el tiempo del cuerpo del bucle.

Cálculo de la eficiencia

- Ejemplo:

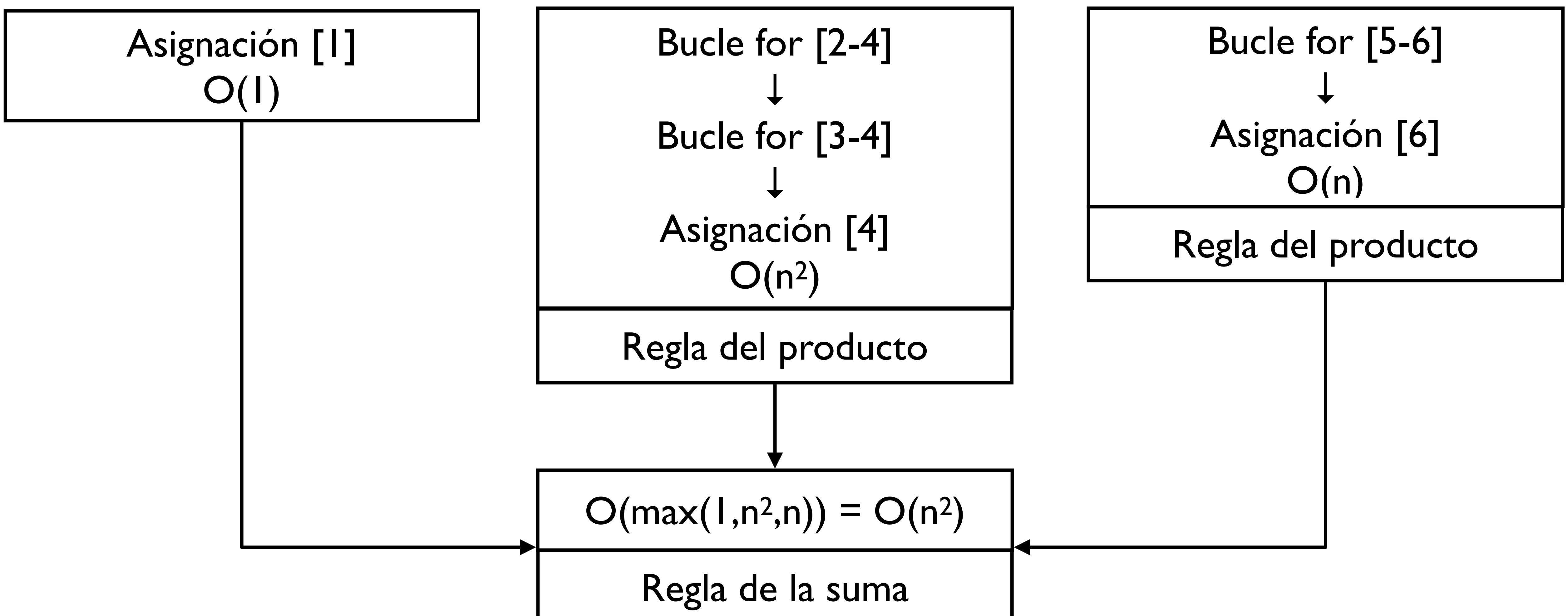
```
k=0;
while (k<n && A[k]!= z)
  k++;
```

- **¡Importante!** No olvidemos que estamos buscando una cota superior. Por ello, siempre analizaremos el peor caso
- En este ejemplo, consideraremos que la condición A[k]!=z es siempre verdadera (en la práctica, esa condición actúa como un "acortador" del bucle).

Cálculo de la eficiencia

```
1.  cin >> n;                       O(1)
2.  for (int i=0; i<n; i++)
3.    for (int j=0; j<n; j++)
4.      A[i][j] = 0;                O(1)
5.  for (int k=0; k<n; k++)
6.    A[k][k] = 1;                  O(1)
```

Líneas 3-4: $O(n)$; líneas 2-4: $O(n^2)$; líneas 5-6: $O(n)$; líneas 1-6: $O(n^2)$



Cálculo de la eficiencia

- Ejemplo:

```
if (A[0][0] == 0)
  for(i=0; i<n; i++)
    for(j=0; j<n; j++)
      A[i][j] = 0;
else
  for(i=0; i<n; i++)
    A[i][i] = 0;
```

$O(n)$ $O(n^2)$ $O(n^2)$

$O(n)$

Cálculo de la eficiencia

- **Funciones:** El orden de eficiencia de una función es el orden de las sentencias que la componen
- La llamada a la función y el paso de parámetros por referencia se realizan en un tiempo constante
- El paso de parámetros por valor implica una operación de copia. Debemos calcular su coste.
- Allí donde aparezca la llamada a la función (en una asignación, en la condición, inicialización o cuerpo de un bucle, etc.… debe tenerse en cuenta su orden de eficiencia y contabilizarlo

Cálculo de la eficiencia

- En particular:
  - Las asignaciones con llamadas a función deben sumar el tiempo de ejecución de cada llamada
  - En las condiciones e incrementos de bucles con llamada se deberá multiplicar el tiempo de la llamada por las iteraciones del bucle
  - La inicialización de bucles o la condición de sentencias condicionales con llamadas deben sumar el tiempo de ejecución de la llamada al tiempo total del bucle o del condicional

Ejemplos

```
int main() {
  double a, b, c;
  double r1, r2;

  cout << "Coeficiente de 2º grado: ";        O(1)
  cin >> a;    O(1)
  cout << "Coeficiente de 1º grado: ";        O(1)
  cin >> b;        O(1)
  cout << "Término independiente: ";        O(1)
  cin >> c;    O(1)
                                          O(1)
  if (a!=0){     O(1)
    r1 = (-b + sqrt(b*b-4*a*c))/2*a;          O(1)
    r2 = (-b - sqrt(b*b-4*a*c))/2*a;          O(1)
    cout << "Las raíces son: " << r1 << " y " << r2 << endl;  O(1)
  }
  else{
    r1 = c/b;      O(1)
    cout << "La única raíz es " << r1 << endl;       O(1)
  }
  return 0;
}
```

O(1)

Ejemplos

```
int BusquedaLineal(const int v[], const int n, const int elemento){
  int i, posicion;
  bool encontrado;

  i=0;                                O(1)
  encontrado = false;                 O(1)
  while(i<n && !encontrado)           O(1)
    if (v[i] == elemento){            O(1)
      posicion = i;                   O(1)
      encontrado = true;              O(1)
    }
    else
      i++;                            O(1)
  if (encontrado)                     O(1)
    return posicion;                  O(1)
  else
    return -1;                        O(1)
}
```

O(1)

O(1×n)=O(n)

O(n)

Debemos realizar siempre el análisis del peor caso: en nuestro ejemplo, el peor caso es que no se encuentre el elemento, lo que supone recorrer todo el vector

Ejemplos

```
int BusquedaBinaria(const int v[], const int n, const int elemento){
  int izquierda, derecha, centro;

  izquierda=0;                                        O(1)  ]
  derecha = n-1;                                      O(1)  ] O(1)
  centro = (izquierda + derecha)/2;                   O(1)  ]

  while(izquierda<=derecha && v[centro]!=elemento){   O(1)  ]
    if (v[centro] > elemento)                         O(1)  ]
      derecha = centro - 1;                           O(1)  ] O(log_2 n)
    else                                                    ]     *           O(log_2 n)
      izquierda = centro + 1;                         O(1)  ]
    centro = (izquierda + derecha)/2;                 O(1)  ]
  }

  if (izquierda > derecha)                            O(1)  ]
    return -1;                                        O(1)  ] O(1)
  else                                                      ]
    return centro;                                    O(1)  ]
}
```

*Análisis del peor caso: si no se encuentra el elemento, vamos dividiendo el espacio de búsqueda a la mitad en cada iteración

Algunas "recetas" matemáticas útiles

- Propiedades de logaritmos:

$$\begin{aligned} \log_b(xy) &= \log_b x + \log_b y \\ \log_b(x/y) &= \log_b x - \log_b y \end{aligned}$$

$$\begin{aligned} \log x^a &= a \cdot \log x \\ \log_b x &= \frac{\log_a x}{\log_a b} \end{aligned}$$

- Propiedades de exponenciales:

$$\begin{aligned} a^{b+c} &= a^b \cdot a^c \\ a^{b \cdot c} &= \left(a^b\right)^c \\ a^{b-c} &= \frac{a^b}{a^c} \end{aligned}$$

$$\begin{aligned} b &= a^{\log_a b} \\ b^c &= a^{c \cdot \log_a b} \end{aligned}$$

Algunas "recetas" matemáticas útiles

- Sumatorias: definición general

$$\sum_{s}^{t} f(i) = f(s) + f(s+1) + f(s+2) + \cdots + f(t)$$

- Progresión geométrica: $f(i) = a^i$

Dado un entero $n \geq 0$ y un número real a $(0 < a \leq 1)$

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}$$

- Progresión aritmética: $f(i) = a_i$ $\qquad a_n - a_{n-1} = d$

Dado un entero $n > 1$

$$\sum_{i=1}^{n} a_i = n \cdot \frac{a_n + a_1}{2} \qquad \sum_{i=0}^{n} i = 1 + 2 + 3 + \cdots + n = n \cdot \frac{n+1}{2}$$

- Suma de cuadrados:

$$\forall n \geq 1 \;\; \sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \cdots = \frac{n(n+1)(2n+1)}{6}$$

Ejemplos

```
void intercambiar(int &a, int &b){
  int aux = a;                    O(1)
  a = b;                          O(1)   O(1)
  b = aux;                        O(1)
}

void OrdenacionSeleccion (int v[], int n){
  int minimo, aux;

  for(int i=0; i<n; i++){             O(1)
    minimo = i;                       O(1)

    for(int j=i+1; j<n; j++)          O(1)
      if (v[j]<v[minimo])             O(1)   O(n-i)   O(n-i)   O(n^2)
        minimo = j;                   O(1)

    intercambiar(v[minimo],v[i]);     O(1)
  }
}
```

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

Ejemplos

```
for(int i=1; i<n; i++)
  for(int j=i+1; j<n+1; j++)
    for(int k=1; k<j+1; k++)
      // Sentencia O(1)
```

O(n-1)
O(n-i)
O(j)

j iteraciones

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

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

$$\left.\begin{array}{r} \sum_{i=1}^{n-1} n^2 = n^2(n-1) \\ \sum_{i=1}^{n-1} n = n(n-1) \\ \sum_{i=1}^{n-1} i^2 = \frac{n(n-1)(2n-1)}{6} \\ \sum_{i=1}^{n-1} i = (n-1)\cdot\frac{1+(n-1)}{2} = \frac{n(n-1)}{2} \end{array}\right\} n^2(n-1)+n(n-1)-\frac{(n-1)n(2n-1)}{6}-\frac{n(n-1)}{2}$$

$O(n^3)$

Ejemplos

```
for(int i=1; i<n+1; i++)
  if(i%2==0){
    for(int j=i; j<n+1; j++)
      x++;
    for(int j=1; j<i+1; j++)
      y++;

  }
```

$x{+}{+}$ → $O(1)$] $O(n-i+1)$

$y{+}{+}$ → $O(1)$] $O(i)$

$$O(n-i+1)+O(i) = O(n+1)$$

Puesto que se ejecuta n/2 veces, tenemos

$$O(n/2(n+1)) \succ O(n^2)$$

Ejemplos

```
void mi_funcion(const int n){
  int x, contador;
  contador = 0;                        O(1)
  x = 2;                               O(1)
  while (x<=n){                        O(1)
    x = 2 * x;                         O(1)      O(?)      O(?)
    contador++;                        O(1)
  }
  cout << contador << endl;            O(1)
}
```

Ejemplos

Producto Matricial:

```
for(int i=0; i<n; i++)
  for(int j=0; j<n; j++){
    C[i][j] = 0;
    for( int k=0; k<n; k++)
      C[i][j] += A[i][k]*B[k][j];
  }
```

$O(1)$

$O(1)$ $O(n)$ $O(n)$ $O(n^2)$ $O(n^3)$

Ejemplos

$$\left.\begin{array}{c} M_1 \rightarrow 10 \times 20 \\ M_2 \rightarrow 20 \times 50 \\ M_3 \rightarrow 50 \times 1 \\ M_4 \rightarrow 1 \times 100 \end{array}\right\} M_1 \times M_2 \times M_3 \times M_4$$

$$\left.\begin{array}{c} \underbrace{\underbrace{\underbrace{\underset{(10 \times 20)}{M_1} \times \underset{(20 \times 50)}{M_2}}_{\substack{M_{12}[10000] \\ 10 \times 50}} \times \underset{(50 \times 1)}{M_3}}_{\substack{M_{123}[500] \\ 10 \times 1}} \times \underset{(1 \times 100)}{M_4}}_{\substack{M_{1234}[1000] \\ 10 \times 100}} \end{array}\right\} [11500]$$

Ejemplos

$$\left.\begin{array}{c} M_1 \to 10 \times 20 \\ M_2 \to 20 \times 50 \\ M_3 \to 50 \times 1 \\ M_4 \to 1 \times 100 \end{array}\right\} M_1 \times M_2 \times M_3 \times M_4$$

$$\left.\begin{array}{l}
\underbrace{\underset{(10 \times 20)}{M_1} \times \underbrace{\underset{(20 \times 50)}{M_2} \times \underset{(50 \times 1)}{M_3}}_{\substack{M_{23}[1000] \\ 20 \times 1}}}_{\substack{M_{123}[200] \\ 10 \times 1}} \times \underset{(1 \times 100)}{M_4} \\
\underbrace{M_{123} \times M_4}_{\substack{M_{1234}[1000] \\ 10 \times 100}}
\end{array}\right\} [2200]$$