Técnicas de diseño de algoritmos
Introducción

Luis Javier Rodríguez Fuentes
Amparo Varona Fernández

Departamento de Electricidad y Electrónica
Facultad de Ciencia y Tecnología, UPV/EHU
luisjavier.rodriguez@ehu.es
amparo.varona@ehu.es

OpenCourseWare 2015
Campus Virtual UPV/EHU

Indice

Algunos ejemplos de Python

- Entrada/Salida por pantalla:

```
nombre = input("Dime tu nombre: ")
edad = int(input("Dime tu edad: "))
print("Hola", nombre, "pronto cumplirás", edad+1)
```

- Decisiones condicionales:

```
Cs = float(input("Score: "))
if Cs>1.0 or Cs<0:
    print("No es correcto, prueba otra vez...")
elif Cs<0.6:
    print("D")
elif Cs>=0.6 and Cs<0.8:
    print("C")
elif Cs>=0.8 and Cs<0.9:
    print("B")
else:
    print("A")
```

Algunos ejemplos de Python

- Recorrer una lista e imprimir el número más grande:

```
# Recorriendo los elementos directamente
largest = -1
A=[3,41,12,9,74,15]
for value in A:
    if value>largest:
        largest = value
print(largest)

# Usando índices
largest = -1
A=[3,41,12,9,74,15]
for i in range(len(A)):
    if A[i]>largest:
        largest = A[i]
print(largest)
```

Algunos ejemplos de Python

- Crear una lista con datos:

```
# Usando el método append
A=[]
for i in range(10):
    x = int(input("Dame un dato: "))
    A.append(x)
# Usando el operador de concatenación (+)
A=[]
for i in range(10):
    x = int(input("Dame un dato: "))
    A=A+[x]
# Inicializando y escribiendo por encima
A=[0 for i in range(10)]
for i in range(10):
    x = int(input("Dame un dato: "))
    A[i] = x
```

Algunos ejemplos de Python

- Borrar datos de una lista:

```
# Usando el método remove
x = int(input("Dame un dato: "))
A.remove(x)

# Localizando la posición del dato
x = int(input("Dame un dato: "))
pos = L.index(x)
del L[pos]
```

Algunos ejemplos de Python

- Crear y usar una función:

```
def computar_pago(h, r):
    if h>40:
        h_extra=h-40
        total=40.0*r+h_extra*1.5*r
    else:
        total=h*r
    return total

nh = float(input("Número de horas: "))
ratio = float(input("Cantidad a pagar por hora: "))
pago = computar_pago(nh, ratio)
print("Cantidad total a pagar: ", pago)
```

Algunos ejemplos de Python

- Buscar el mínimo y el máximo de una serie de datos:

```
maximo=None
minimo=None
while True:
    num = input("Dame un número: ")
    if num=="":
        break
    try:
        n = float(num)
    except:
        continue
    if maximo==None or n>maximo:
        maximo=n
    if minimo==None or n<minimo:
        minimo=n
print ("El máximo es: ", maximo)
print ("El mínimo es: ", minimo)
```


Algunos ejemplos de Python

- Buscar en un archivo de texto:

```
fname = input("Nombre del archivo: ")
fp = open(fname,"r")
count = 0
total = 0
for line in fh:
    if not line.startswith("X-DSPAM-Confidence:"):
        continue
    pos = line.find(':')
    cadena = line[pos+1:]
    conf = float(cadena)
    count = count + 1
    total = total + conf
fp.close()
print("La confianza media es: ", total/count)
```

Motivación

- La Ciencia de la Computación (*Computer Science*) no sólo estudia la arquitectura y la programación de los ordenadores. Se ocupa también, sobre todo, de los métodos para calcular soluciones exactas o aproximadas a problemas: algoritmos.
- La ciencia de la computación existía antes que los ordenadores y se dedicaba al estudio y búsqueda de métodos sistemáticos de resolución de problemas algebraicos o simbólicos.
- La motivación principal de la computación durante muchos años fue la de desarrollar métodos de cómputo numérico más precisos.
- Los babilonios, egipcios y griegos desarrollaron una gran variedad de métodos para el cálculo de distintas magnitudes, como por ejemplo el área de un círculo o el máximo común divisor de dos números enteros (teorema de Euclides).
- En el siglo XIX, Charles Babbage describió la *máquina analítica*, que representaba un computador moderno de uso general, que podía liberar a los humanos del tedio de los cálculos y al mismo tiempo realizarlos de manera más precisa.

Motivación

- La resolución de problemas en el mundo real ha requerido un estudio más profundo de la ciencia de la computación, que a su vez ha ampliado la gama de problemas a los que se puede aplicar.
- Por otro lado, la construcción de algoritmos es una tarea eminentemente práctica. La existencia de ordenadores cada vez más poderosos no elimina la necesidad de desarrollar algoritmos más eficientes. En la mayoría de las aplicaciones, el así llamado *cuello de botella* no es el hardware sino más bien el uso de un software poco eficiente.
- Hay tres preguntas centrales que aparecen cuando se considera la posibilidad de usar un ordenador para realizar un determinado cálculo:
  1. ¿Es posible hacerlo?
  2. ¿Cómo se hace?
  3. ¿Cómo de rápido puede hacerse?

Objetivos Generales

1. Introducir el análisis de la complejidad de algoritmos.

Objetivos Generales

1. Introducir el análisis de la complejidad de algoritmos.
2. Presentar y estudiar los esquemas de diseño de algoritmos más comunes.

Objetivos Generales

1. Introducir el análisis de la complejidad de algoritmos.
2. Presentar y estudiar los esquemas de diseño de algoritmos más comunes.
3. Desarrollar habilidades en la aplicación práctica de las técnicas de análisis y diseño de algoritmos.

Objetivos Generales

1. Introducir el análisis de la complejidad de algoritmos.
2. Presentar y estudiar los esquemas de diseño de algoritmos más comunes.
3. Desarrollar habilidades en la aplicación práctica de las técnicas de análisis y diseño de algoritmos.
4. Analizar la eficiencia de diversas técnicas para resolver una serie de problemas de referencia.

Objetivos Generales

1. Introducir el análisis de la complejidad de algoritmos.
2. Presentar y estudiar los esquemas de diseño de algoritmos más comunes.
3. Desarrollar habilidades en la aplicación práctica de las técnicas de análisis y diseño de algoritmos.
4. Analizar la eficiencia de diversas técnicas para resolver una serie de problemas de referencia.
5. Diseñar y analizar nuevos algoritmos.

Datos históricos

- El término **algoritmo** proviene del nombre del matemático árabe Al'Khwarizmi, que escribió un tratado sobre los números. Este texto se perdió, pero su versión latina, *Algoritmi de Numero Indorum*, sí se conoce.
- Al'Khwarizmi une al rigor de los griegos la simplicidad de los indios. Sus libros son intuitivos y prácticos y su principal contribución fue simplificar las matemáticas a un nivel entendible por no expertos. En particular, muestra las ventajas de usar el sistema decimal indio, un atrevimiento para su época, dado lo tradicional de la cultura árabe.
- La exposición clara de cómo calcular de una manera sistemática a través de algoritmos diseñados para ser usados con algún tipo de dispositivo mecánico similar a un ábaco, más que con lápiz y papel, muestra la intuición y el poder de abstracción de Al'Khwarizmi. Hasta se preocupaba de reducir el número de operaciones necesarias en cada cálculo. Por esta razón, aunque no inventara el primer algoritmo, merece que el concepto esté asociado a su nombre.

Datos históricos

- Los babilonios empleaban unas pequeñas bolas hechas de semillas o pequeñas piedras, a manera de *cuentas*, agrupadas en carriles de caña. En 1.800 A.C. un matemático babilonio inventó varios algoritmos que le permitieron resolver problemas de cálculo numérico.
- En la misma época, los egipcios usan un algoritmo de multiplicación similar al de expansión binaria.
- La ciencia de la computación trata con objetos que pueden representarse de manera abstracta mediante modelos. La investigación en modelos formales de computación se inició en los años 30 del siglo XX por Turing, Post, Kleene, Church y otros.
- Los computadores construidos en los años 40 y 50 del siglo XX disponían de muy poca memoria y unidades de procesamiento muy lentas. Era importante, por tanto, desarrollar teorías (modelos, análisis y algoritmos) para hacer un uso eficiente de tan escasos recursos. Esto dio origen al área que ahora se conoce como Algoritmos y Estructuras de Datos. También se estudió la complejidad inherente de algunos problemas, dando lugar a una jerarquía de problemas computacionales y al área hoy conocida como Complejidad Computacional.

Definición de algoritmo

- Un **algoritmo** es una serie finita de pasos para resolver un problema.
  1. **El número de pasos debe ser finito:** el algoritmo debe terminar en un tiempo finito.
  2. **La especificación debe estar libre de ambigüedad:** dados un estado inicial y unos datos de entrada, el algoritmo debe conducir en todos los casos a un estado final (salida o solución).
- Definiciones alternativas
  - Algoritmo: conjunto de reglas operacionales inherentes a un cómputo.
  - Algoritmo: método sistemático, susceptible de ser realizado mecánicamente, para resolver un problema.
- Sería un error pensar que los algoritmos tratan únicamente sobre problemas de computación. Cualquier problema en cualquier contexto requiere plantearse cómo resolverlo, es decir, qué algoritmo utilizar. La respuesta puede depender de numerosos factores, a saber: el tamaño del problema, el modo en que está planteado (es decir, de qué datos se dispone) y el tipo y la potencia del equipo disponible para su resolución.

Características de un algoritmo

1. **Entrada:** qué necesita el algoritmo (datos de entrada).
2. **Salida:** qué produce el algoritmo (formato de la solución).
3. **No ambiguo:** los pasos deben ser explícitos, en cualquier situación debe estar claro cuál es la siguiente instrucción a ejecutar.
4. **Finito:** el algoritmo termina en un número finito de pasos.
5. **Correcto:** el algoritmo hace lo que se supone que debe hacer (la solución es correcta).
6. **Eficaz:** cada instrucción debe ser lo suficientemente básica como para que pueda ser ejecutada usando papel y lápiz.
7. **General:** debe ser lo suficientemente general como para contemplar todos los casos de entrada.

Diseño de algoritmos: consideraciones generales

- Si no se conoce un método para solucionar el problema en cuestión, debemos hacer un análisis previo del problema, evaluando alternativas, exigencias y excepciones.

Diseño de algoritmos: consideraciones generales

- Si no se conoce un método para solucionar el problema en cuestión, debemos hacer un análisis previo del problema, evaluando alternativas, exigencias y excepciones.
- Si se conoce un *buen* método de resolución del problema, se debe especificar el método exacta y completamente en un lenguaje no ambiguo.

Diseño de algoritmos: consideraciones generales

- Si no se conoce un método para solucionar el problema en cuestión, debemos hacer un análisis previo del problema, evaluando alternativas, exigencias y excepciones.
- Si se conoce un *buen* método de resolución del problema, se debe especificar el método exacta y completamente en un lenguaje no ambiguo.
- Los problemas pueden agruparse en conjuntos de problemas semejantes: la solución a uno de ellos puede dar pistas sobre cómo resolver los demás.

Diseño de algoritmos: consideraciones generales

- Si no se conoce un método para solucionar el problema en cuestión, debemos hacer un análisis previo del problema, evaluando alternativas, exigencias y excepciones.
- Si se conoce un *buen* método de resolución del problema, se debe especificar el método exacta y completamente en un lenguaje no ambiguo.
- Los problemas pueden agruparse en conjuntos de problemas semejantes: la solución a uno de ellos puede dar pistas sobre cómo resolver los demás.
- En general, un método que resuelve todos los problemas de un conjunto se considera superior a otro método que sólo es capaz de resolver una parte de ellos. Sin embargo, este último podría ser la mejor opción si es capaz de resolver nuestro problema de un modo más eficiente.

Diseño de algoritmos: consideraciones generales

- Si no se conoce un método para solucionar el problema en cuestión, debemos hacer un análisis previo del problema, evaluando alternativas, exigencias y excepciones.
- Si se conoce un *buen* método de resolución del problema, se debe especificar el método exacta y completamente en un lenguaje no ambiguo.
- Los problemas pueden agruparse en conjuntos de problemas semejantes: la solución a uno de ellos puede dar pistas sobre cómo resolver los demás.
- En general, un método que resuelve todos los problemas de un conjunto se considera superior a otro método que sólo es capaz de resolver una parte de ellos. Sin embargo, este último podría ser la mejor opción si es capaz de resolver nuestro problema de un modo más eficiente.
- Criterios de bondad: corrección, generalidad, eficiencia, elegancia, etc.

Análisis de la eficiencia computacional

- Uno de los objetivos prioritarios en el diseño de algoritmos es mantener tan bajo como sea posible el consumo de recursos (tiempo y memoria).
- Siempre que se trata de resolver un problema, se suelen considerar distintos algoritmos, con el fin de utilizar el más eficiente.
- Pero **¿cómo determinar qué algoritmo es el más eficiente?**
- La **estrategia empírica** consiste en escribir los algoritmos en un lenguaje de programación y ejecutarlos en un ordenador sobre algunos ejemplares de prueba, midiendo experimentalmente el tiempo y la memoria consumidos.
- La **estrategia teórica** consiste en determinar matemáticamente la cantidad de recursos (tiempo y memoria) que necesitará el algoritmo en función del tamaño del ejemplar considerado.

Complejidad: conceptos básicos

- **Orden de magnitud** de:
  - el tiempo de procesamiento, y
  - la memoria necesaria,

  en términos de un parámetro (n) denominado **tamaño del problema**
- **Formulación**:
  - independiente de la implementación (lenguaje de programación, máquina, condiciones instantáneas de uso de la CPU, memoria disponible en placa, etc.)
  - dependiente **SÓLO** del tamaño del problema (n)

Complejidad temporal

Notación asintótica	**Perfil de ejecución**
a priori	*a posteriori*
independiente de la implementación	dependiente de la implementación
$n \to \infty$	$n = 1, 2, 3, \ldots, N$

PASO Segmento de un algoritmo con significado completo cuyo tiempo de ejecución es independiente del tamaño del problema

$t(n)$ Número de pasos necesarios para la ejecución de un algoritmo, en función del tamaño del problema (n)

Notación asintótica

- Notación *está en el orden de* (COTA SUPERIOR)

$$O(f(n)) = \{t : \mathcal{N} \to \mathcal{R}^+ : \quad \exists c \in \mathcal{R}^+ \wedge \exists n_0 \in \mathcal{N} : \quad \forall n \geq n_0 \quad t(n) \leq c \cdot f(n)\}$$

- Notación *está en Omega de* (COTA INFERIOR)

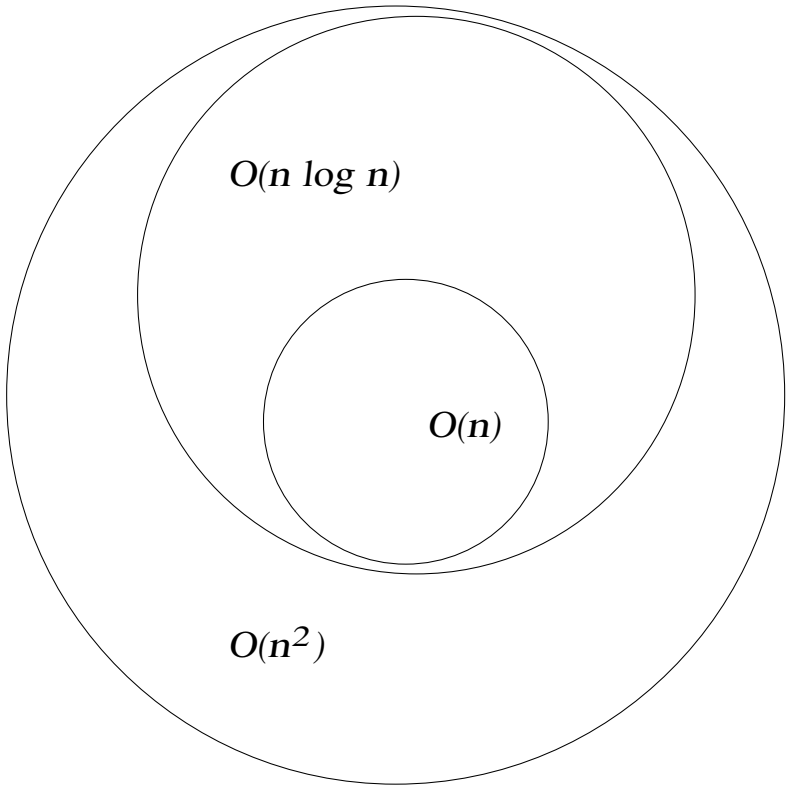
$$\Omega(f(n)) = \{t : \mathcal{N} \to \mathcal{R}^+ : \quad \exists c \in \mathcal{R}^+ \wedge \exists n_0 \in \mathcal{N} : \quad \forall n \geq n_0 \quad t(n) \geq c \cdot f(n)\}$$

- Notación *está en Theta de* (ORDEN DE MAGNITUD EXACTO)

$$\Theta(f(n)) = \{t : \mathcal{N} \to \mathcal{R}^+ : \quad \exists c, d \in \mathcal{R}^+ \wedge \exists n_0 \in \mathcal{N} : \quad \forall n \geq n_0 \quad c \cdot f(n) \leq t(n) \leq d \cdot f(n)\}$$

Relación entre distintos conjuntos de funciones

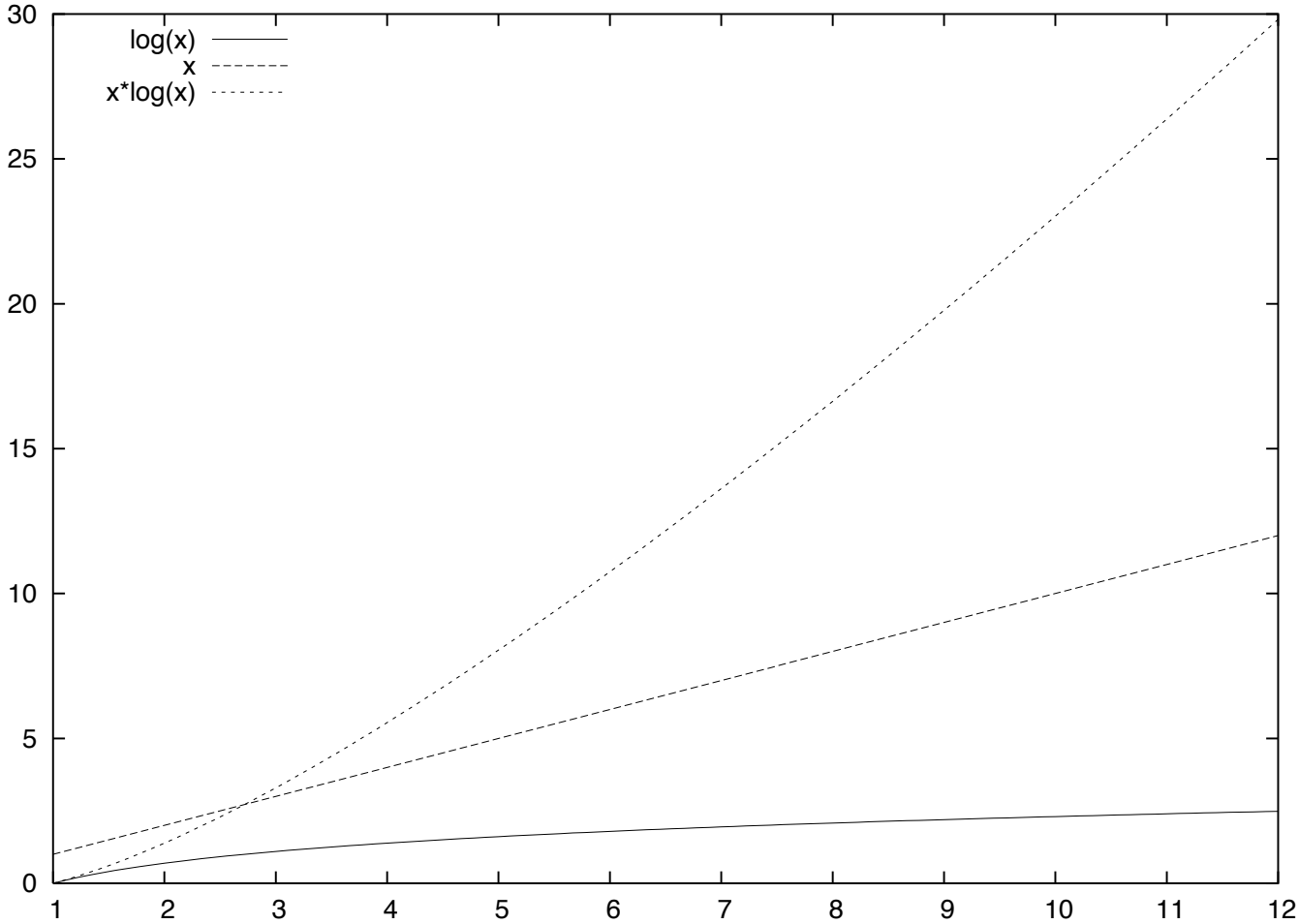
$$O(1) < O(\log(n)) < O(n) < O(n\log(n)) < O(n^2) < O(n^2\log(n)) < O(n^3) < O(2^n) < O(n!)$$



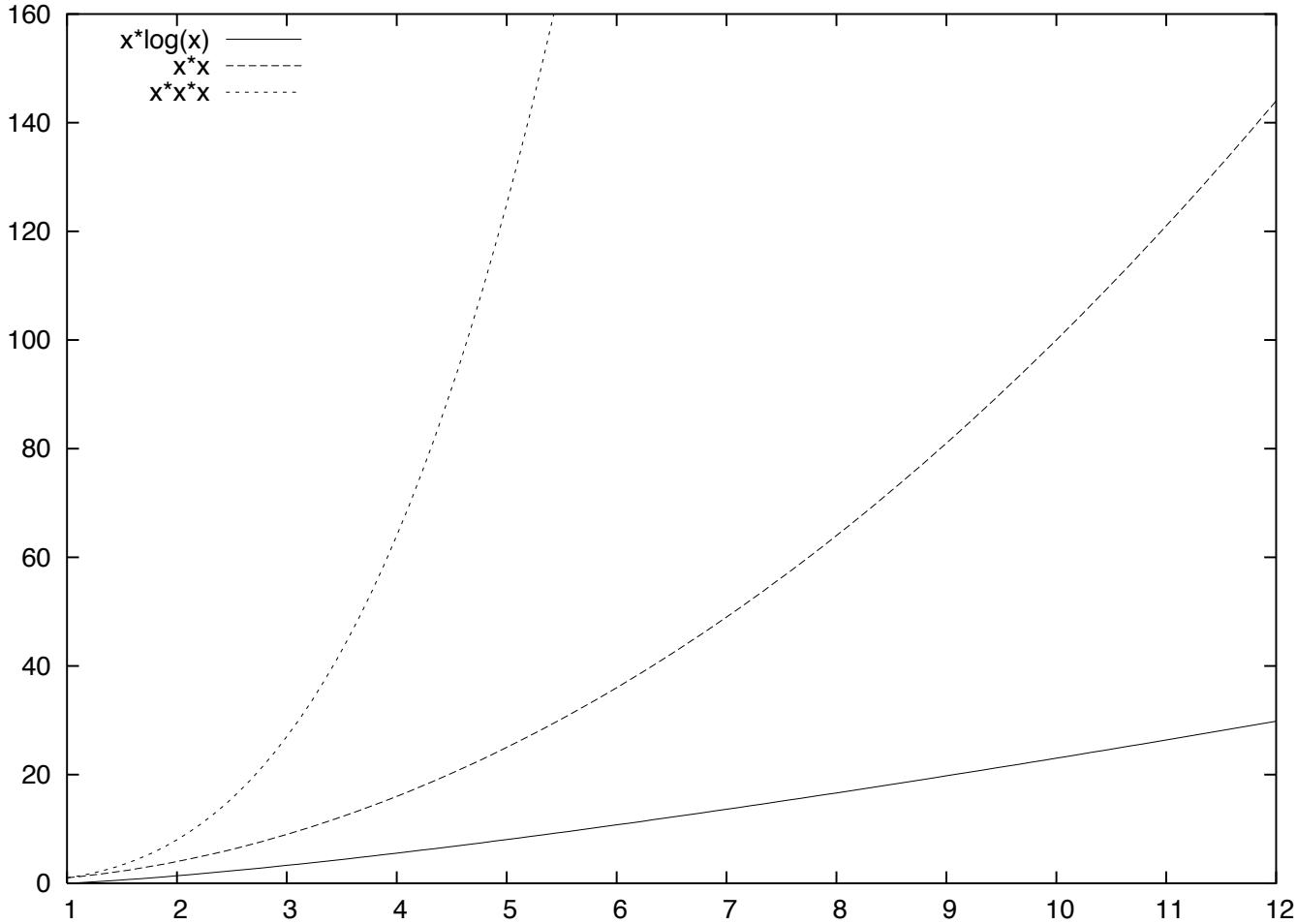
Órdenes de magnitud

$\log(n)$	n	$n\log(n)$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296
6	64	384	4096	262144	$1.844 \cdot 10^{19}$
7	128	896	16384	2097152	$3.4 \cdot 10^{38}$

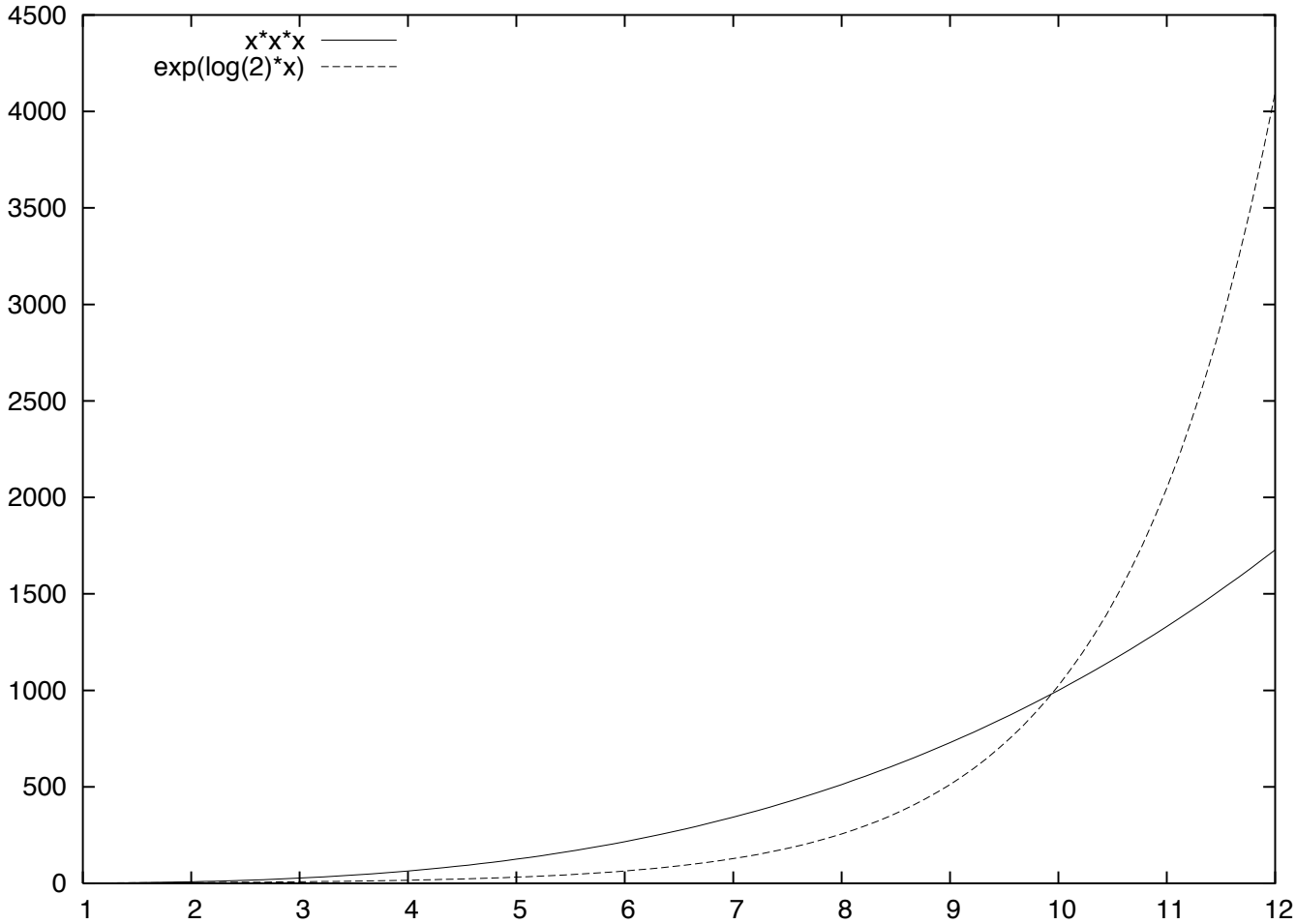
Órdenes de magnitud



Órdenes de magnitud



Órdenes de magnitud



Propiedades de la notación $O(\cdot)$

- Si $t(n) = \sum_{i=1}^{m} g_i(n)$, donde $g_i(n) \in O\left(f_i(n)\right)$, entonces:

$$t(n) \in \max_{i=1,\ldots,m} \left\{O\left(f_i(n)\right)\right\} = O\left(\max_{i=1,\ldots,m} \left\{f_i(n)\right\}\right)$$

  En particular:

$$t(n) = \left(\sum_{k=1}^{m} a_k n^k\right) \in O\left(n^m\right)$$

- Si $t(n) = \prod_{i=1}^{m} g_i(n)$, donde $g_i(n) \in O\left(f_i(n)\right)$, entonces:

$$t(n) \in O\left(\prod_{i=1}^{m} f_i(n)\right)$$

Las mismas propiedades pueden deducirse para la notación $\Omega(\cdot)$

Otras propiedades

- *Regla de dualidad*

$$f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$$

- *Definición 1*

$$O(f(n)) < O(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge g(n) \notin O(f(n))$$

- *Definición 2*

$$\Omega(f(n)) > \Omega(g(n)) \Leftrightarrow f(n) \in \Omega(g(n)) \wedge g(n) \notin \Omega(f(n))$$

- *Propiedad de absorción*

$$f_1(n) \in O(g(n)) \wedge f_2(n) \in \Theta(g(n)) \Rightarrow f_1(n) + f_2(n) \in \Theta(g(n))$$

Identidades útiles

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2)$
- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \in \Theta(n^3)$
- $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1}), \quad k > 0$
- $\sum_{i=1}^{n} r^i = \frac{r^{n+1}-1}{r-1} \in \Theta(r^n), \quad r > 1$
- $\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log(n))$
- $\left.\begin{array}{l} \sum_{i=1}^{n} \frac{1}{r^i} \\ \sum_{i=1}^{n} \frac{i}{r^i} \\ \sum_{i=1}^{n} \frac{1}{i!} \end{array}\right\} \in \Theta(1), \quad r > 1$
- $\sum_{i=1}^{n} i2^{i-1} = n2^n - 2^n + 1 \in \Theta(n2^n)$
- $\sum_{i=1}^{n} \binom{n}{i} = 2^n \in \Theta(2^n)$
- $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) \in \Theta\left(\sqrt{n}\left(\frac{n}{e}\right)^n\right)$

Búsqueda: formulación del problema

- **Problema de la búsqueda**: dado un vector v (de longitud n) y un valor x, deberá devolverse una posición (entre 0 y $n-1$) donde aparezca x; en caso de no encontrarla, se devolverá `None`.
- Se considerarán problemas de búsqueda sobre estructuras lineales.
- Se supondrá que el acceso a los elementos es aleatorio y de coste fijo (independiente del tamaño del vector).
- Además de mostrar con ejemplos que puede haber distintas soluciones a un problema, se tratará de evaluar cualitativamente el coste temporal de cada solución y analizar la idoneidad de una u otra en función del tamaño del problema.

Búsqueda secuencial

- La solución más sencilla en Python (sin usar el método `index`) es la búsqueda secuencial:

```
def buscar(v,x):
    n=len(v)
    for i in range(n):
        if v[i]==x:
            return i
    return None
```

- Usando el método index:

```
v.index(x)
```

- **Coste temporal:** t(n) = número de pasos a realizar en función del tamaño del problema (n)
- **Paso:** conjunto de instrucciones cuyo coste no depende de n
- En este ejemplo, $n = len(v)$
- **Mejor caso:** 1 paso (el valor se encuentra en la posición 0)
- **Peor caso:** n pasos (el valor no se encuentra en el vector)

Búsqueda secuencial

- Si el vector está ordenado, la búsqueda secuencial puede acabar en cuanto detecta que el valor buscado es menor que el actual:

```
def buscar2(v,x):
    n=len(v)
    for i in range(n):
        if v[i]==x:
            return i
        elif v[i]>x:
            return None
    return None
```

- Curiosamente, dependiendo del valor de x, el coste temporal de `buscar2()` puede ser menor (x pequeños) o mayor (x grandes) que el de `buscar()`
- Sin embargo, el orden de magnitud del coste es idéntico: en el mejor caso se realiza un único paso y en el peor caso n pasos

Ordenación: formulación del problema

- El objetivo de la ordenación es únicamente poder aplicar métodos eficientes de búsqueda.
- **Problema de la ordenación**: sea v un vector de valores entre los que se establece una relación de orden; el objetivo es reorganizar los valores en el propio vector, de modo que $v[i] \leq v[j] \quad \forall i < j$.
- Si el vector v se ha representado en Python mediante una lista, el método `sort` realiza precisamente esa tarea: `v.sort()`.
- Existen métodos sencillos de ordenación que en el peor caso implican un número de pasos proporcional a n^2, y métodos más eficientes (*quicksort*, *mergesort*, etc.) que en un caso promedio implican del orden de $n * log_2(n)$ pasos.

Ordenación por inserción

- Ordenación por inserción

```
def sort_ins(v):
    n=len(v)
    for i in range(1,n):
        x=v[i]
        j=i-1
        while j>=0 and x<v[j]:
            v[j+1]=v[j]
            j=j-1
        v[j+1]=x
```

- Ejemplo (evolución del vector):

```
i=0   3  1  7  -1  0  5
i=1   1  3  7  -1  0  5
i=2   1  3  7  -1  0  5
i=3  -1  1  3   7  0  5
i=4  -1  0  1   3  7  5
i=5  -1  0  1   3  5  7
```

- Mejor caso: $n-1$ pasos (vector ordenado).
- Peor caso: del orden de n^2 pasos (vector en orden inverso).

Ordenación por inserción

- Coste temporal en el peor caso:

$$\begin{aligned} t_p(n) &= 1 + \sum_{i=1}^{n-1}(1 + \sum_{j=0}^{i-1} 1) \\ &= n + \sum_{i=1}^{n-1} i \\ &= n + \frac{1}{2}n(n-1) \\ &= \frac{1}{2}n^2 + \frac{1}{2}n \quad \text{(del orden de } n^2 \text{ pasos)} \end{aligned}$$

Ordenación por selección

- Ordenación por selección:

```
def sort_sel(v):
    n=len(v)
    for i in range(0,n-1):
        minimo=v[i]
        argmin=i
        for j in range(i+1,n):
            if v[j]<minimo:
                minimo=v[j]
                argmin=j
        v[argmin]=v[i]
        v[i]=minimo
```

- Ejemplo (evolución del vector):

	3	1	7	-1	0	5
i=0	-1	1	7	3	0	5
i=1	-1	0	7	3	1	5
i=2	-1	0	1	3	7	5
i=3	-1	0	1	3	7	5
i=4	-1	0	1	3	5	7

- Coste temporal: del orden de n^2 pasos (en todos los casos).

Ordenación por selección

- Coste temporal:

$$
\begin{aligned}
t(n) &= 1 + \sum_{i=0}^{n-2}(1 + \sum_{j=i+1}^{n-1} 1) \\
&= n + \sum_{i=0}^{n-2}(n - i - 1) \\
&= n + \sum_{i=1}^{n-1} i \\
&= n + \frac{1}{2}n(n-1) \\
&= \frac{1}{2}n^2 + \frac{1}{2}n \quad \text{(del orden de } n^2 \text{ pasos)}
\end{aligned}
$$

Ordenación por intercambio

- Ordenación por intercambio:

```
def sort_int(v):
    n=len(v)
    for i in range(0,n-1):
        for j in range(n-1,i,-1):
            if v[j]<v[j-1]:
                tmp=v[j]
                v[j]=v[j-1]
                v[j-1]=tmp
```

- Ejemplo (evolución del vector):

	3	1	7	-1	0	5
i=0	-1	3	1	7	0	5
i=1	-1	0	3	1	7	5
i=2	-1	0	1	3	5	7
i=3	-1	0	1	3	5	7
i=4	-1	0	1	3	5	7

- Coste temporal: del orden de n^2 pasos (en todos los casos).

Ordenación por intercambio

- Coste temporal:

$$\begin{aligned} t(n) &= 1 + \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \\ &= 1 + \sum_{i=0}^{n-2} (n - i - 1) \\ &= 1 + \sum_{i=1}^{n-1} i \\ &= \frac{1}{2} n^2 - \frac{1}{2} n + 1 \quad \text{(del orden de } n^2 \text{ pasos)} \end{aligned}$$