



# Complejidad computacional


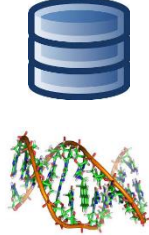
Informática I - 2547100

Departamento de Ingeniería Electrónica  
y de Telecomunicaciones

Facultad de Ingeniería

2016-2

# Efficient programs

- Lo más importante en un programa es que sus datos de salida sean **correctos**.
- Sin embargo, a veces la **velocidad** del programa es esencial para obtener datos de salida correctos.
- En otras ocasiones, la velocidad del programa incrementará su **utilidad**.
- En ocasiones es necesario entonces incrementar la **complejidad conceptual** del algoritmo para reducir su **complejidad computacional**.

# Execution time

¿Cuánto se demora `fact()` en ejecutarse?

```
def fact(n):  
    ''' Asume que n es un entero mayor que cero '''  
    ans = 1  
    while n>=1:  
        ans *= n  
        n -= 1  
    return ans
```

Dependerá de:

1. La velocidad del computador
2. La eficiencia del intérprete de Python
3. El valor de `n`

# Time vs. computational complexity

- Para deshacernos de las dependencias 1 y 2, no midamos el tiempo en segundos sino en **cantidad de operaciones primitivas**: copiar un dato, sumar, restar, comparar, etc.
- La dependencia 3 es ineludible y nos obliga siempre a expresar la complejidad computacional teniendo en cuenta los datos de entrada.
- El tiempo de ejecución depende del **tamaño** y **valor** de los datos de entrada:

```
def linearSearch(L, x):  
    for e in L:  
        if e==x:  
            return True  
    return False
```

*¿Qué pasa si ejecuto  
linearSearch(L, 3) donde L  
tiene un millón de elementos?*

# Execution time cases

Observemos que podemos dividir en tres casos las posibilidades de tiempo de ejecución de un programa:

- **Caso mejor:** es cuando las características de los datos de entrada (para un tamaño fijo) favorecen más el tiempo de ejecución.
- **Caso peor:** es cuando las características de los datos de entrada (para un tamaño fijo) generan el peor tiempo de ejecución.
- **Caso promedio:** es el generado por las condiciones promedio de las entradas.

Nos interesa el caso peor para poder dar garantías.

# Modeling execution time

```
def fact(n):  
    ''' Asume que n es un entero mayor que cero  
        Retorna el factorial de n '''
```

```
1 ← ans = 1  
1 ← while n >= 1:  
2 ←     ans *= n  
2 ←     n -= 1  
1 ← return ans
```

n veces

$2 + 5n$

→

Si  $n=1000$  entonces la función  
ejecutará 5002 operaciones básicas

Finalmente observemos que:

la constante aditiva  
es despreciable

→  $2 + 5n \approx 5n$  para  $n \gg 0$

¿y la constante  
multiplicativa?

# Multiplicative constants

```
def sqrtExhaustive(x, epsilon):
```

```
    step = epsilon**2
```

```
    ans = 0.0
```

```
    while abs(ans**2 - x) >= epsilon and ans**2 <= x: } ~ 9 operaciones  
        ans += step                                     básicas por ciclo
```

```
    if abs(ans**2 - x) < epsilon:
```

```
        return ans
```

```
>>> sqrtExhaustive(100, 0.0001)
```

→ ~ 10<sup>9</sup> iteraciones

```
def sqrtBisection(x, epsilon):
```

```
    low = 0.0
```

```
    high = max(1.0, x)
```

```
    ans = (high + low)/2.0
```

```
    while abs(ans**2 - x) >= epsilon:
```

```
        if ans**2 < x:
```

```
            low = ans
```

```
        else:
```

```
            high = ans
```

```
            ans = (high + low)/2.0
```

```
    return ans
```

la constante  
multiplicativa es  
despreciable

~ 11 operaciones  
básicas por ciclo

```
>>> sqrtBisection(100, 0.0001)
```

→ ~ 20 iteraciones

# Asymptotic notation

```
def f(x):  
    ans = 0  
    for i in range(1000):  
        ans += 1  
    print('Operaciones: ', ans)  
    for i in range(x):  
        ans += 1  
    print('Operaciones: ', ans)  
    for i in range(x):  
        for j in range(x):  
            ans += 1  
            ans += 1  
    print('Operaciones: ', ans)  
    return ans
```

1000 operaciones

x operaciones

$2x^2$  operaciones

```
>>> f(10)  
Operaciones : 1000  
Operaciones : 1010  
Operaciones : 1210  
  
>>> f(1000)  
Operaciones : 1000  
Operaciones : 2000  
Operaciones : 2002000  
  
>>> f(1000000)  
Operaciones : 1000  
Operaciones : 1001000  
Operaciones : 2000001001000
```

El tiempo de ejecución puede ser expresado por:

$$1000 + x + 2x^2 \approx x^2 \text{ para } x \gg 0$$

notación asintótica



# Big O notation

Para calcular la complejidad computacional asintótica de un algoritmo:

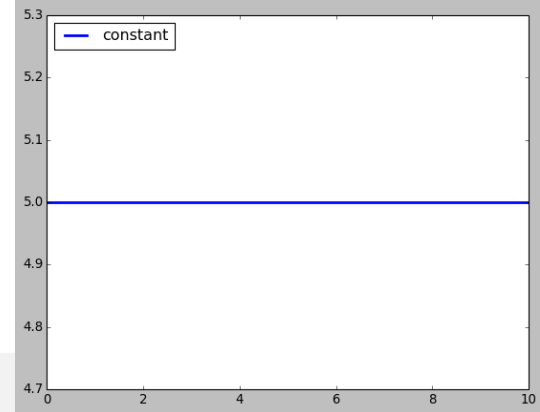
- Si el modelo de tiempo de ejecución se compone de la suma de varios términos, entonces conserve solo el que crece más rápido.
- Si ese término es un producto, elimine las constantes.

$$\begin{array}{l} \cancel{1000} + \cancel{x} + 2x^2 \\ \cancel{2}x^2 \\ x^2 \end{array}$$

Notación de **O** grande:

$$O(x^2)$$

# Constant complexity



```
def f():  
    ans = 0  
    for i in range(1000):  
        ans += 1  
    print('Iteraciones acumuladas: ', ans)
```

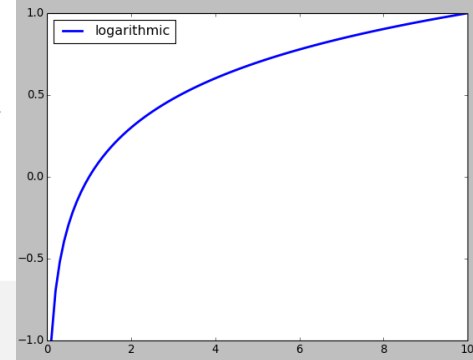
$O(1)$

```
def genderGuesser(name, age):  
    print('¡Hola '+name+'!')  
    if name[len(name)-1]=='a' or name[len(name)-1]=='A':  
        print('Ya sé que tienes '+age+' años y creo que eres una mujer...')  
    else:  
        print('Ya sé que tienes '+age+' años y creo que eres un hombre...')  
    ans = input('¿Adiviné? ')  
    if ans.lower()=='si':  
        print('¡Cómo te quedó el ojo!')  
    else:  
        print('Mmmm ok, lo siento')  
    print('Chao...')
```

$O(1)$

*Asumir que print y len son  $O(1)$*

# Logarithmic complexity



```
def intToStr(i):  
    '''Asume que i es un entero no negativo  
    Retorna la representación decimal de i en un string'''  
    digits = '0123456789'  
    if i==0:  
        return 0  
    result = ''  
    while i > 0:  
        result = digits[i%10] + result  
        i = i//10  
    return result
```

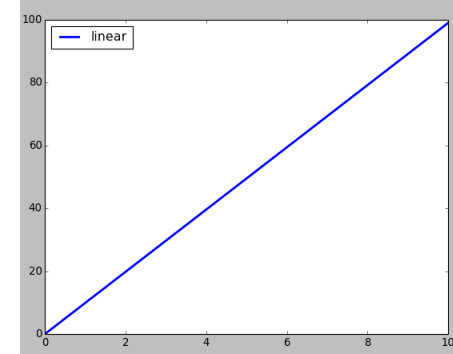
$O(\log(i))$

```
def addDigits(n):  
    '''Asume que n es un entero no negativo  
    Retorna la suma de los dígitos de n'''  
    stringRep = intToStr(n)  
    val = 0  
    for c in stringRep:  
        val += int(c)  
    return val
```

$O(\log(n))$   
+  
 $O(\underbrace{\text{len}(\text{stringRep})}_{\log(n)})$   
 $O(\log(n))$

Asumir que int es  $O(1)$

# Linear complexity



```
def addDigits(s):  
    '''Asume que s es un string compuesto por caracteres decimales  
    Retorna un entero que es la suma de los dígitos en s'''  
    val = 0  
    for c in s:  
        val += int(c)  
    return val
```

**$O(\text{len}(s))$**

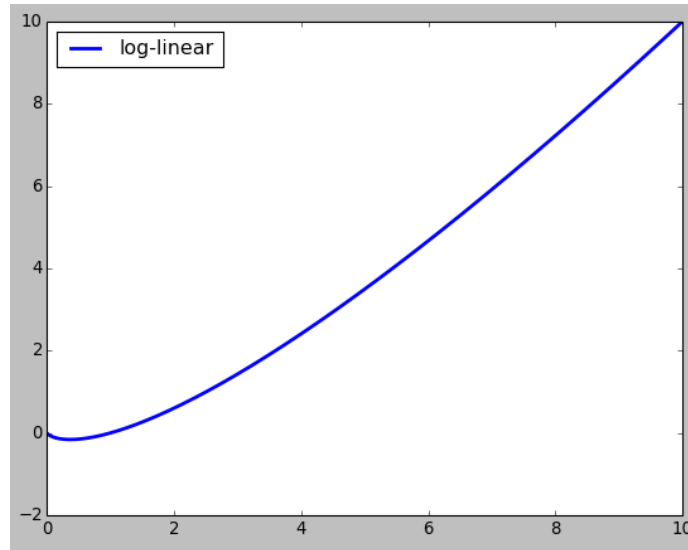
```
def factorial(x):  
    '''Asume que x es un entero positivo  
    Retorna el factorial de x'''  
    if x == 1:  
        return 1  
    else:  
        return x * factorial(x-1)
```

**$O(\text{len}(x))$**

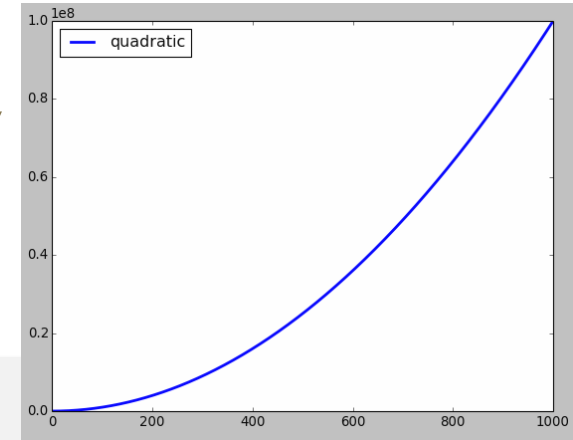
*Asumir que int es  $O(1)$*

# Log-linear complexity

- Muchos algoritmos prácticos pertenecen a esta clase
- Un ejemplo típico es el algoritmo de ordenamiento **merge sort**



# Polynomial complexity



```
def intersec(L1, L2):  
    '''Asume que L1 y L2 son listas  
    Retorna una lista con la intersección de L1 y L2'''  
  
    # Construcción de lista con elementos comunes  
    tmp = []  
    for e1 in L1:  
        for e2 in L2:  
            if e1 == e2:  
                tmp.append(e1)  
  
    # Eliminación de duplicados  
    result = []  
    for e in tmp:  
        if e not in result:  
            result.append(e)  
    return result
```

$O(\text{len}(L1) * \text{len}(L2))$

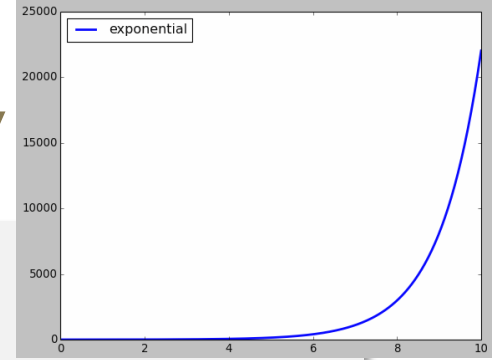
+

$O(\text{len}(tmp) * \text{len}(result))$

$$2 * O(\text{len}(L1) * \text{len}(L2)) = O(\text{len}(L1) * \text{len}(L2))$$

Asumir que append es  $O(1)$

# Exponential complexity



```
def getBinaryRep(n, numDigits):
```

```
    '''Asume que n y numDigits son enteros no negativos
    Retorna un string que es la representación binaria
    de n en numDigits bits'''
```

```
    result = ''
```

```
    while n > 0: →  $O(\log(n))$ 
```

```
        result = str(n%2) + result
```

```
        n = n//2
```

```
    for i in range(numDigits-len(result)): →  $O(\text{numDigits})$ 
```

```
        result = '0' + result
```

```
    return result
```

**$O(\text{numDigits})$**

```
def genPowerSet(L):
```

```
    '''Asume que L es una lista
    Retorna una lista de listas que contiene todas las posibles
    combinaciones de los elementos de L'''
```

```
    powerset = []
```

```
    for i in range(2**len(L)): →  $O(2^{\text{len}(L)})$ 
```

```
        binStr = getBinaryRep(i, len(L)) →  $O(\text{len}(L))$ 
```

```
        subset = []
```

```
        for j in range(len(L)): →  $O(\text{len}(L))$ 
```

```
            if binStr[j] == '1':
```

```
                subset.append(L[j])
```

```
        powerset.append(subset)
```

```
    return powerset
```

**$O(2^{\text{len}(L)} * 2^{\text{len}(L)}) \approx O(2^{\text{len}(L)})$**

# Main complexity classes

$O(1)$ : tiempo de ejecución **constante**

$O(\log n)$ : tiempo de ejecución **logarítmico**

$O(n)$ : tiempo de ejecución **lineal**

$O(n \log n)$ : tiempo de ejecución **log-lineal**

$O(n^k)$ : tiempo de ejecución **polinomial**

$O(k^n)$ : tiempo de ejecución **exponencial**