# Programación

## Tema 9. Algorítmica y Complejidad


**Javier González Villa**

**David Lázaro Urrutia**

Departamento de Matemática Aplicada
y Ciencias de la Computación

# Contenidos

1. Eficiencia
    1. Métodos de medición
2. Clases de complejidad
    1. Constante $O(k)$
    2. Lineal $O(n)$
    3. Polinomial $O(n^k)$
    4. Logarítmica $O(\log n)$
    5. Exponencial $O(k^n)$
3. Conclusiones

# 1. Eficiencia

- ¿Cómo podemos saber cuanto tiempo tardará en ejecutarse un algoritmo en particular?
- ¿Cómo saber el impacto de la implementación en el coste computacional?
- ¿Cómo podemos medir el impacto del equipo donde se esta ejecutando nuestro código en el tiempo requerido?
- ¿Qué aspectos he de tener en cuenta a la hora de realizar implementaciones en términos de coste computacional en espacio y tiempo?
- Dadas las capacidades actuales de los equipos de computo, ¿es realmente necesario tener en cuenta la eficiencia de nuestros algoritmos?

# 1. Eficiencia

- Simulación de evacuación masiva de personas en infraestructuras grandes.
- Análisis hidrodinámico de inundaciones urbanas a escala de ciudad.
- Optimización de trayectorias y horarios en redes ferroviarias extensas.
- Cálculo térmico transitorio de túneles largos o presas bajo escenarios extremos.
- Evaluación del impacto ambiental y estructural de grandes obras lineales, como carreteras o túneles.

# 1. Eficiencia

- La eficiencia trata de medir como de óptimo es un algoritmo, una implementación o la arquitectura de un ordenador.
- Existen diversas formas de medir la eficiencia dependiendo del propósito.
- En términos de eficiencia no solo es necesario entender el tiempo requerido para la ejecución de un determinado código en una máquina concreta sino también para entender el espacio de almacenamiento que este requiere.
- Existen principalmente tres formas de medir la eficiencia:
  - Tiempos de ejecución: orientada a análisis del rendimiento de diferentes ordenadores.
  - Conteo de operaciones: orientada a optimizar las implementaciones.
  - Orden de crecimiento: orientada a puramente la eficiencia de los algoritmos implementados.

# 1.1. Métodos de medición

***Tiempo:*** *medición del tiempo de ejecución requerido por el algoritmo en su peor ejecución.*

- Varía entre algoritmos.
- Varía entre implementaciones.
- Varía entre diferentes ordenadores.
- Se requieren múltiples entradas para observar la tendencia de los tiempos de ejecución.
- No se puede recoger de manera precisa los tiempos requeridos de ejecución en función de las entradas del algoritmo.

```
[35]: import time
      import itertools
      t1 = time.time()
      L = []
      suma = 0
      for i in range(100000000):
          L.append(5)
          suma = suma + 5

      t2 = time.time()
      suma = 0
      for v in itertools.repeat(5,100000000):
          suma = suma + 5

      t3 = time.time()

      print("Tiempo usando listas: ",t2-t1)
      print("Tiempo usando itertools: ",t3-t2)

      Tiempo usando listas:  8.695796966552734
      Tiempo usando itertools:  3.9877750873565674
```

# 1.1. Métodos de medición

***Operaciones:*** *conteo de las operaciones realizadas por el algoritmo en su peor ejecución.*

- Varía entre algoritmos.
- Varía entre implementaciones.
- No depende del ordenador donde se ejecute.
- No se define muy claramente qué operaciones requieren un coste tal que deben ser contadas.
- Se puede recoger de manera precisa el número de instrucciones ejecutadas en función de las entradas del algoritmo.

```
[19]: def sort(L):
          L2 = []
          lenObj = len(L)
          while len(L2) != lenObj:
              minV = float('inf')
              for v in L:
                  if (v < minV):
                      minV = v
              L2.append(minV)
              L.remove(minV)
          return L2

      sort([6,7,1,2,9,7,3,0,21,5,3])

[19]: [0, 1, 2, 3, 3, 5, 6, 7, 7, 9, 21]
```

# 1.1. Métodos de medición

***Ordenes:*** *estudio del código obviando el conteo de órdenes individuales y centrándose en la ejecución de grandes conjuntos de instrucciones para el peor caso del algoritmo.*

- Necesitamos considerar las entradas de nuestro algoritmo para obtener su coste.
- Generalmente al igual que con el conteo de operaciones tenderemos a considerar el caso peor de entre los posibles.
- Queremos obviar las operaciones triviales y centrarnos en el cuello de botella de nuestros algoritmos.
- Queremos eliminar todas las variables que no sean puramente algorítmicas.

```
[2]: def search(L, v):
         i = 0
         for e in L:
             if v == e:
                 return True, i
             i += 1
         return False, i

     L = range(1000)
     print(search(L, 10))
     print(search(L, 100))
     print(search(L, 1000))
```

```
(True, 10)
(True, 100)
(False, 1000)
```

# 1.1. Métodos de medición

***Ordenes:*** *estudio del código obviando el conteo de órdenes individuales y centrándose en la ejecución de grandes conjuntos de instrucciones para el peor caso del algoritmo.*

- Operaciones (search): 1 + 3n + 1
- Orden de crecimiento (search): O(n)

| Operaciones | O() |
|---|---|
| $2 + 5n + n^2$ | $O(n^2)$ |
| $2^n + 3n^{10}$ | $O(2^n)$ |
| $7 + n^3 + 1000n$ | $O(n^3)$ |

```
[2]: def search(L, v):
         i = 0
         for e in L:
             if v == e:
                 return True, i
             i += 1
         return False, i

     L = range(1000)
     print(search(L, 10))
     print(search(L, 100))
     print(search(L, 1000))

     (True, 10)
     (True, 100)
     (False, 1000)
```

# 1.1. Métodos de medición

***Ordenes:*** *estudio del código obviando el conteo de órdenes individuales y centrándose en la ejecución de grandes conjuntos de instrucciones para el peor caso del algoritmo.*

- Sumatorio de ordenes:

$$O(n) + O(n) = O(n + n) = O(2n) = O(n)$$

- Multiplicación de ordenes (anidación):

$$O(n) * O(n) = O(n * n) = O(n^2)$$

```
[19]: def sort(L):
          L2 = []
          lenObj = len(L)
          while len(L2) != lenObj:
              minV = float('inf')
              for v in L:
                  if (v < minV):
                      minV = v
              L2.append(minV)
              L.remove(minV)
          return L2

      sort([6,7,1,2,9,7,3,0,21,5,3])

[19]: [0, 1, 2, 3, 3, 5, 6, 7, 7, 9, 21]
```

# 1.1. Métodos de medición

***Ordenes:*** *estudio del código obviando el conteo de órdenes individuales y centrándose en la ejecución de grandes conjuntos de instrucciones para el peor caso del algoritmo.*

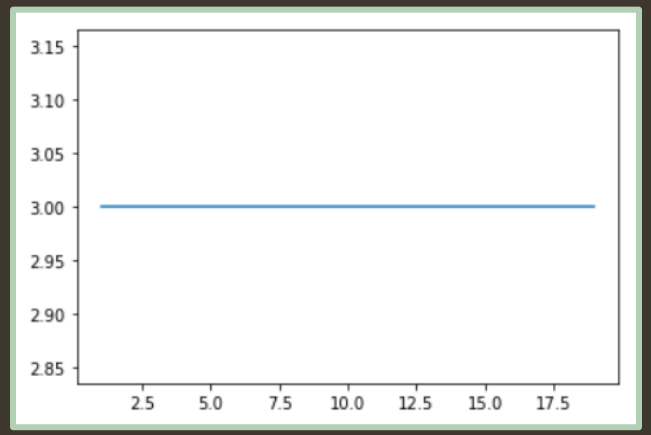
- Varía entre algoritmos.
- No varía entre implementaciones sino entre algoritmos.
- No depende del ordenador donde se ejecute.
- Obvia las instrucciones triviales centrándose solamente en los bloques que requieren gran carga computacional.
- Se puede recoger de manera precisa el número de instrucciones ejecutadas en función de las entradas del algoritmo.

```
[19]: def sort(L):
          L2 = []
          lenObj = len(L)
          while len(L2) != lenObj:
              minV = float('inf')
              for v in L:
                  if (v < minV):
                      minV = v
              L2.append(minV)
              L.remove(minV)
          return L2

      sort([6,7,1,2,9,7,3,0,21,5,3])

[19]: [0, 1, 2, 3, 3, 5, 6, 7, 7, 9, 21]
```

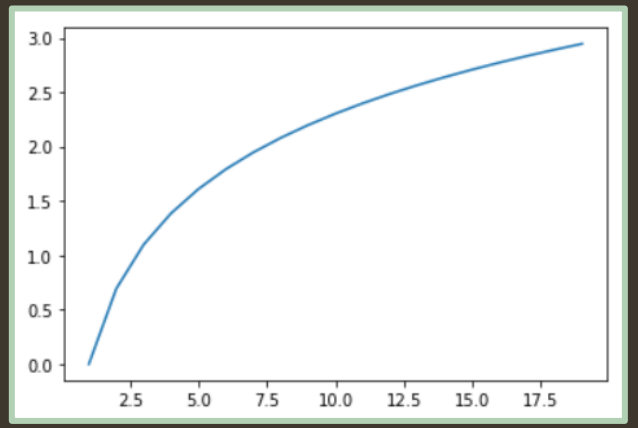
# 2. Clases de complejidad

- Describen el orden de crecimiento del algoritmo en función de las entradas del mismo obviando las instrucciones de bajo coste computacional.
- Permiten estudiar la escalabilidad de nuestros algoritmos mediante la clasificación directa en una clase concreta, permitiendo extrapolarlo a los tiempos de ejecución.
- Permite evaluar cuando un algoritmo puede utilizarse y cuando los límites computacionales harán imposible su uso.
- Para clasificar un algoritmo generalmente se utiliza el algoritmo con su peor entrada, es decir por ejemplo en el caso de búsquedas en listas que el elementos buscado este en la última posición, haciendo necesario recorrer toda la lista.

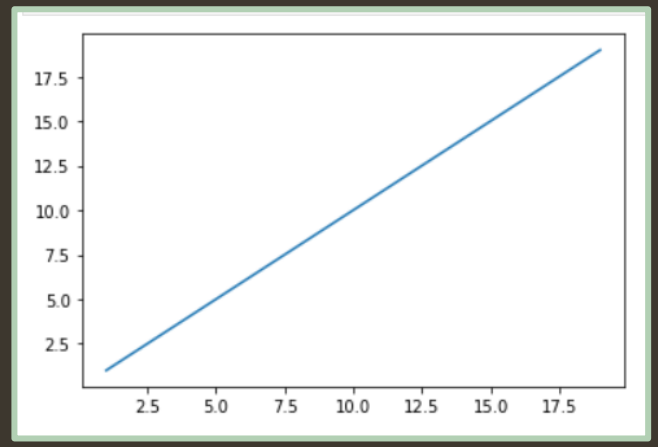
# 2. Clases de complejidad



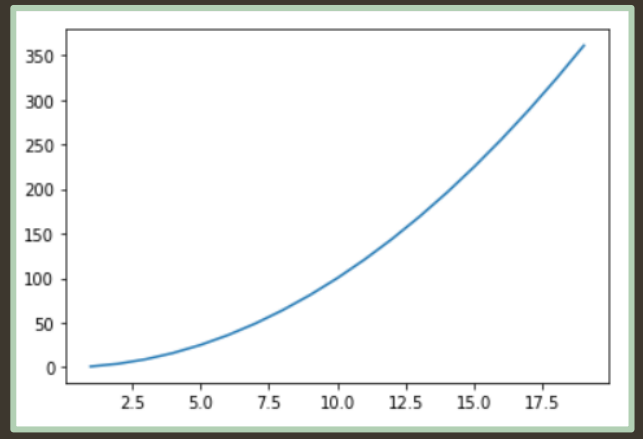

$O(3)$



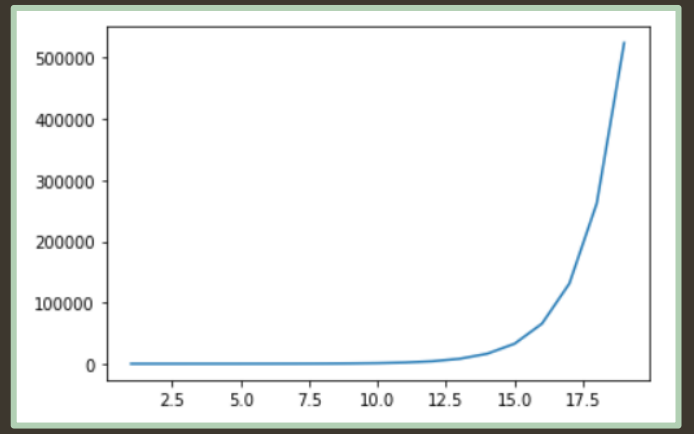

$O(\log n)$




$O(n)$




$O(n^2)$




$O(2^n)$

# 2. Clases de complejidad

| Clase | n = 1 | n=10 | n=100 | n=1000 |
|---|---|---|---|---|
| $O(k)$ | 2 | 2 | 2 | 2 |
| $O(n)$ | 1 | 10 | 100 | 1.000 |
| $O(n^k)$ | 1 | 100 | 10.000 | 1.000.000 |
| $O(\log n)$ | 0 | 1 | 2 | 3 |
| $O(n \log n)$ | 0 | 10 | 200 | 3000 |
| $O(k^n)$ | 2 | 1024 | 1.267.650.600.228.229.4<br>01.496.703.205.376 | … |

*k constante de ejemplo igual a 2 y n tamaño de la entrada del algoritmo*

# 2.1. Constante $O(k)$

***Constante:*** *la complejidad del algoritmo es independiente del tamaño de las entradas.*

- Pueden contener estructuras iterativas o recursivas siempre y cuando estas no dependan directamente del tamaño de la entrada del algoritmo.
- La mayoría de algoritmos o funciones sencillas suelen tener esta complejidad.
- A pesar de poder tener constantes k diversas, todas se agrupan como de igual coste computacional y complejidad.

```
[13]: def k_x_10(k):
          suma = 0
          for i in range(10):
              suma = suma + k
          return k

      k_x_10(10)

[13]: 10
```

# 2.2. Lineal $O(n)$

***Lineal:*** *la complejidad depende directamente del tamaño de la entrada del algoritmo.*

- Generalmente son algoritmos sencillos que simplemente recorren mediante elementos iterativos y recursivos una lista con una longitud determinada.
- Algunos algoritmos con esta complejidad son por ejemplo búsquedas en listas, cálculo de sumatorios u otra aplicación matemática a todos los elementos de una lista.

```
[15]: def search(e,L):
          i = -1
          while i < len(L)-1:
              i = i + 1
              if e == L[i]:
                  return i

      search(4, [1,3,4,1,8,5,7,9])

[15]: 2
```

# 2.3. Polinomial $O(n^k)$

***Polinomial:*** *la complejidad depende generalmente del número de bucles anidados dependientes de las variables de entrada del algoritmo.*

- Dentro de esta familia de algoritmos, los más comunes son los de orden cuadrático $O(n^k)$ ya que es muy común el uso de dos bucles anidados.
- Pueden darse complejidades polinómicas de mayor orden, pero generalmente viene asociadas a una mala producción de código.
- Es un tipo de complejidad que si alcanza un alto orden de magnitud ha de ser controlada para que sea viable su ejecución en tiempos razonables.

[19]:
```
def sort(L):
    L2 = []
    lenObj = len(L)
    while len(L2) != lenObj:
        minV = float('inf')
        for v in L:
            if (v < minV):
                minV = v
        L2.append(minV)
        L.remove(minV)
    return L2

sort([6,7,1,2,9,7,3,0,21,5,3])
```

[19]:
```
[0, 1, 2, 3, 3, 5, 6, 7, 7, 9, 21]
```

# 2.4. Logarítmica $O(\log n)$

***Logarítmica:*** *esta complejidad crece de manera logarítmica requiriendo menos tiempo que la complejidad polinómica.*

- Muchos de los algoritmos optimizados para búsquedas y ordenación de listas tienen una complejidad logarítmica.
- Algunos ejemplos de algoritmos con complejidad logarítmica son por ejemplo la búsqueda binaria o la búsqueda por bisección.
- Los algoritmos que impliquen la aplicación de  técnicas de divide y vencerás suelen tener este tipo de complejidad.

```
[1]: def busqueda_binaria_recursiva(a,L,menor,mayor):
         print("Ejecución!")
         if mayor >= menor:
             mitad = (mayor + menor) // 2
             if L[mitad] == a:
                 return mitad
             elif L[mitad] > a:
                 return busqueda_binaria_recursiva(a,L,menor, mitad-1)
             else:
                 return busqueda_binaria_recursiva(a,L,mitad+1,mayor)
         else:
             return -1

     lista = [6,7,13,17,87,121,123,255]
     print("El valor 13 esta en la posición: "+str(busqueda_binaria_recursiva(13,lista,0,7)))

     Ejecución!
     Ejecución!
     Ejecución!
     El valor 13 esta en la posición: 2
```

# 2.4. Logarítmica $O(\log n)$

***Logarítmica:*** *esta complejidad crece de manera logarítmica requiriendo menos tiempo que la complejidad polinómica.*

Ejecución 1

| 6 | 7 | 13 | 17 | 87 | 121 | 123 | 255 |
|---|---|---|---|---|---|---|---|

Ejecución 2

| 6 | 7 | 13 | 17 |
|---|---|---|---|

Ejecución 3

| 13 | 17 |
|---|---|

Tamaño de lista = 8

Complejidad: O(log n)

$\log_2 8 = 3$

# 2.5. Exponencial $O(k^n)$

***Exponencial:*** *complejidad generalmente no abordable ya que el coste computacional es demasiado alto.*

- Algunas funciones recursivas o iterativas generalmente que implican combinatoria, pueden dar lugar a complejidades de orden exponencial.
- Costes computacionales tan altos pueden dar lugar a la imposibilidad de resolver el problema mediante ese algoritmo.
- Se suelen usar algoritmos de aproximación o probabilistas en lugar de algoritmos deterministas de tan alto coste.

```
[1]: #2.4
def Hanoi(n, Origen, Auxiliar, Destino):
    '''
    Argumentos: n número de discos a jugar.
                Origen cadena que identifique la torre origen.
                Auxiliar cadena que idenfifique la torre auxiliar.
                Destino cadena que identifique la torre destino.
    Pinta por pantalla la secuencia de movimientos necesaria para resolver el juego.
    '''
    if n == 1:
        print('Mueve disco de la torre ', Origen, ' a la torre ', Destino)
    else:
        Hanoi(n-1, Origen, Destino, Auxiliar)
        print('Mueve disco de la torre ', Origen, ' a la torre ', Destino)
        Hanoi(n-1, Auxiliar, Origen, Destino)
    return

Hanoi(3, 'Origen', 'Auxiliar', 'Destino')

Mueve disco de la torre  Origen  a la torre  Destino
Mueve disco de la torre  Origen  a la torre  Auxiliar
Mueve disco de la torre  Destino  a la torre  Auxiliar
Mueve disco de la torre  Origen  a la torre  Destino
Mueve disco de la torre  Auxiliar  a la torre  Origen
Mueve disco de la torre  Auxiliar  a la torre  Destino
Mueve disco de la torre  Origen  a la torre  Destino
```

# 2.5. Exponencial $O(k^n)$

***Exponencial:*** *complejidad generalmente no abordable ya que el coste computacional es demasiado alto.*

- Permutaciones completas.
- Algoritmos recursivos como Fibonacci.

$$F(n) = F(n-1) + F(n-2)$$

- Generación de subconjuntos.
- Algunos algoritmos de fuerza bruta que exploran todas las posibles combinaciones.

```
[10]: def fibonacci(n):
          print("Ejecución!")
          if n == 0:
              return 0
          elif n == 1:
              return 1
          else:
              return fibonacci(n-1) + fibonacci(n-2)

      print(fibonacci(2))

      Ejecución!
      Ejecución!
      Ejecución!
      1
```

```
[3]: def fibonacci_iterativo(n):
         a, b = 0, 1
         for _ in range(n):
             print("Ejecución!")
             a, b = b, a + b
         return a

     print(fibonacci_iterativo(2))

     Ejecución!
     Ejecución!
     1
```

```
[11]: def generar_subconjuntos(lista):
          subconjuntos = [[]]
          for elemento in lista:
              nuevos_subconjuntos = []
              for subconjunto in subconjuntos:
                  nuevos_subconjuntos.append(subconjunto + [elemento])
              subconjuntos += nuevos_subconjuntos
          return subconjuntos

      generar_subconjuntos([1,2,3])

[11]: [[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
```

# 2. Clases de complejidad – Operaciones comunes

| Clase | Listas | Tuplas | Cadenas | Diccionarios | Conjuntos |
|---|---|---|---|---|---|
| Añadir | *O(1)* | --- | --- | *O(1)* | *O(1)* |
| Longitud | *O(1)* | *O(1)* | *O(1)* | *O(1)* | *O(1)* |
| Comparar | *O(n)* | *O(n)* | *O(n)* | *O(n)* | *O(n)* |
| Eliminar | *O(n)* | --- | --- | ***O(1)*** | ***O(1)*** |
| Copiar | *O(n)* | *O(n)* | *O(n)* | *O(n)* | *O(n)* |
| Buscar | *O(n)* | *O(n)* | *O(n)* | ***O(1)*** | ***O(1)*** |

*Algunas estructuras estan optimizadas por la unicidad de sus elementos mediante funciones llamadas Hash*

# 3. Conclusiones

- La eficiencia mide qué tan óptimo es un algoritmo o implementación, considerando tanto el tiempo de ejecución como el espacio de almacenamiento.
- Existen tres métodos principales para medir eficiencia: tiempos de ejecución prácticos, conteo de operaciones algorítmicas y estudio de órdenes de crecimiento.
- El análisis se centra habitualmente en el peor caso para garantizar que el algoritmo funcionará razonablemente en todas las circunstancias.
- Las clases de complejidad (constante, lineal, polinomial, logarítmica y exponencial) permiten comparar la escalabilidad de algoritmos y decidir su aplicabilidad según el tamaño de entrada.
- Un buen análisis de eficiencia ayuda a elegir y diseñar algoritmos que sean prácticos y viables para problemas reales.