Análisis de algoritmos

Algoritmia y optimización
Grado en Ingeniería en Inteligencia Artificial

Análisis de algoritmos

Estudio de eficiencia

- El **análisis de algoritmos** permite evaluar y comparar algoritmos.
- Suele centrarse en medir la **eficiencia** en el uso de recursos:
  - Identificar si un algoritmo es eficiente.
  - Identificar si un algoritmo es más eficiente que otro.

Análisis de algoritmos

Complejidad

- Cuando hablamos de los recursos que consume un algoritmo, nos referimos a su **complejidad**.
- Un algoritmo es más complejo (menos eficiente) si consume más recursos.
- Recursos habituales a considerar:
  - Tiempo de ejecución (complejidad temporal).
  - Consumo de memoria (complejidad espacial).

Análisis de algoritmos

Tipos

- La complejidad puede verse afectada por factores tanto externos como internos.
  - **Externos**: potencia del hardware, el compilador o interprete, los datos de entrada.
  - **Internos**: número y tipo de instrucciones.
- Distinguiremos entre **análisis empírico** y **análisis teórico**.

Análisis de algoritmos

Análisis empírico

- **Ejecutar el algoritmo** y medir los recursos consumidos.
  - Por ejemplo, si nos referimos a recursos temporales, podemos cronometrar el tiempo de ejecución.
- **Ventaja**: medida real del comportamiento del algoritmo en un entorno concreto.
- **Desventaja**: puede verse afectado por cuestiones extrínsecas al propio algoritmo.

Análisis de algoritmos

Análisis teórico

- Consiste en obtener una **función matemática** que represente la complejidad del algoritmo.
- **Ventaja**: no es necesario ejecutar el algoritmo y el resultado depende exclusivamente del diseño del mismo.
- **Desventaja**: es difícil trasladar esta función a términos prácticos de una ejecución en un entorno real.

Complejidad teórica

Complejidad teórica

Nociones generales

- ¿Qué complejidad tienen los siguientes algoritmos?
  - Se suele considerar que el coste de las operaciones elementales es unitario.

```
función suma_uno(número)
    valor := número + 1
    devuelve valor
```

```
función suma_uno(número)
    devuelve número + 1
```

Complejidad teórica

Nociones generales

- ¿Qué complejidad tiene el siguiente algoritmo?

```
función acumulado(v)
    suma := 0
    para i := 1 hasta |v|
        suma += v[i]
    devuelve suma
```

- La complejidad depende del tamaño de `v`

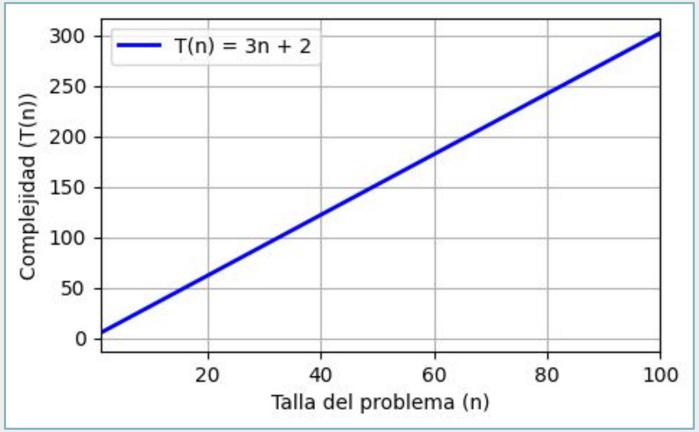
Complejidad teórica

Talla del problema

- La complejidad de un algoritmo se estima en función de su **talla**.
- La talla representa el **tamaño de la entrada**.

```
función acumulado(v)
    suma := 0
    para i := 1 hasta |v|
        suma += v[i]
    devuelve suma
```

$$T(n) = 1 + n(3) + 1 = 3n + 2$$



Complejidad teórica

Cotas de complejidad

- ¿Qué complejidad tiene el siguiente algoritmo?

```
función buscar(v, z)
    para i := 1 hasta |v|
        si v[i] = z
            devuelve i
    devuelve NO_ENCONTRADO
```

- La complejidad varía en función de los valores de la entrada.

Complejidad teórica

Cotas de complejidad

- ¿Cómo podemos calcular la complejidad cuando depende de algo **externo** al algoritmo?
  - Inferir **el mejor y el peor caso** y estimar los límites superiores e inferiores de su complejidad.
  - A esto se le llama **cotas de complejidad**.

Complejidad teórica

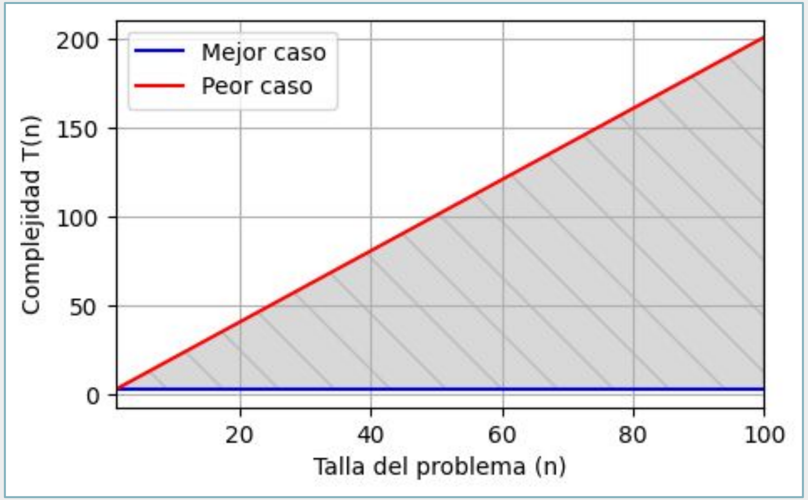
Cotas de complejidad

- ¿Qué complejidad tiene el siguiente algoritmo?

```
función buscar(v, z)
    para i := 1 hasta |v|
        si v[i] = z
            devuelve i
    devuelve NO_ENCONTRADO
```

Mejor caso: $T(n) = 3$

Peor caso: $T(n) = 2n + 1$



Notación asintótica

Notación asintótica

Concepto

- No es (tan) interesante medir la complejidad exacta de un algoritmo sino **cómo crece** cuando la talla es cada vez mayor.
- Para esto se puede utilizar la **notación asintótica *Big O***:
  - Proporciona una estimación de la complejidad cuando la talla del problema **tiende a infinito**.

Notación asintótica

Big O

- Formalmente:

$$T(n) \in \mathcal{O}(g(n)) \iff \exists n_0, c > 0 : T(n) \leq c \cdot g(n) \, \forall n \geq n_0$$

- Se centra en cómo **escala** el algoritmo.
- **Factores constantes** y **términos de menor orden** se ignoran.
- Es una manera **eficaz** de analizar y comparar algoritmos.

Notación asintótica

Big O

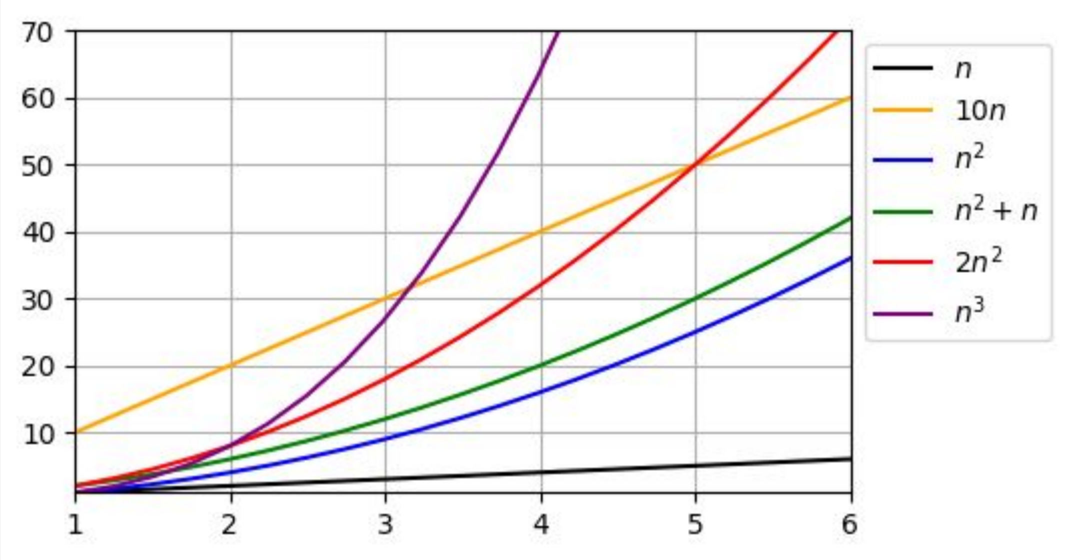
$$n^2 + 3 \in \mathcal{O}(n^2)$$

$$n^2 + n + 3 \in \mathcal{O}(n^2)$$

$$4n^2 \in \mathcal{O}(n^2)$$

$$n^2 + 3 \in \mathcal{O}(n^3)$$

$$n^2 + 3 \notin \mathcal{O}(n)$$



Notación asintótica

Órdenes de complejidad

- Complejidad constante: $O(1)$
- Complejidad logarítmica: $O(\log n)$
- Complejidad lineal: $O(n)$
- Complejidad lineal-logarítmica: $O(n \log n)$
- Complejidad cuadrática: $O(n^2)$
- Complejidad exponencial: $O(2^n)$

$$\mathcal{O}(1) \subset \mathcal{O}(\log n) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(2^n)$$

Notación asintótica

Otras notaciones

- La notación *Big O* forma parte de un conjunto más amplio de notaciones.
  - Notación Ω: límites inferiores.
  - Notación Θ: funciones en la intersección de O y Ω.

Cálculo de complejidades

Cálculo de complejidades

Pasos

1. Determinar la talla del problema.
2. Determinar los casos mejor o peor (si los hubiera).
3. Calcular la complejidad asintótica de cada caso.

Cálculo de complejidades

Algoritmos iterativos

```
función producto_matrices_cuadradas(A, B)
    n := dimensión(A)
    C := ceros(A)
    para i := 1 hasta n
        para j := 1 hasta n
            suma := 0
            para k := 1 hasta n
                suma := suma + A[i][k] * B[k][j]
            C[i][j] := suma
    devuelve C
```

Cálculo de complejidades

Algoritmos iterativos

```
función ordenación_por_inserción(v)
    para i := 2 hasta |v|
        valor := v[i]
        j := i - 1
        mientras j > 0 y v[j] > valor
            v[j + 1] := v[j]
            j := j - 1
        v[j + 1] := valor
```

Cálculo de complejidades

Tipos

En la práctica, los algoritmos se categorizan como:

- **Iterativos**: análisis directo basado en la cantidad de veces que se ejecutan los distintos bloques del algoritmo.
- **Recursivos**: cuya complejidad se calcula recursivamente.

Cálculo de complejidades

Algoritmos recursivos

- La complejidad de un algoritmo recursivo depende del número y la naturaleza de sus llamadas recursivas.
- **Relación de recurrencia:** describe cómo el problema original se descompone en subproblemas más pequeños.
- Proporcionan un marco para analizar la complejidad de un algoritmo recursivo.

Cálculo de complejidades

Búsqueda binaria

```
función búsqueda_binaria(v, z)
    si |v| = 0
        devuelve NO_ENCONTRADO
    m := |v| / 2
    si v[m] = z
        devuelve m
    si v[m] > z
        devuelve búsqueda_binaria(v[:m], z)
    si_no
        devuelve búsqueda_binaria(v[m:], z)
```

Mejor caso: $T(n) \in \mathcal{O}(1)$

Peor caso: $T(n) \in \mathcal{O}(\log n)$

- ¿Qué complejidad tendría una **búsqueda secuencial**?

Cálculo de complejidades

Ordenación por selección

```
función ordenación_por_selección(v):
    si |v| = 1:
        devuelve

    índice_mínimo := 1
    para i=1 hasta |v|:
        si v[i] < v[índice_mínimo]:
            índice_mínimo = i

    intercambiar(v[1], v[índice_mínimo])

    ordenación_por_selección(v[2:])
```

$$T(n) \in \mathcal{O}(n^2)$$

- ¿Es más eficiente que la **ordenación por inserción**?

Ejercicios

Ejercicios

Pertenencias de complejidad

$$n^3 \in \mathcal{O}(n^2)?$$

$$n^3 + n \in \mathcal{O}(n^3)?$$

$$n^2 \in \mathcal{O}(2^n)\,?$$

$$2^{n+1} \in \mathcal{O}(2^n)\,?$$

$$\log(2n) \in \mathcal{O}(\log n)\,?$$

$$\log n \in \mathcal{O}(n^{\frac{1}{2}})?$$

$$f(n) \notin \mathcal{O}(g(n)) \implies g(n) \in \mathcal{O}(f(n))?$$

Ejercicios

Cálculo de complejidad iterativa (I)

```
función calculo(n)
    resultado := 0
    para i := 1 hasta n
        j := n
        mientras j > 1
            resultado := resultado + (i * j)
            j := j / 2
    devuelve resultado
```

Ejercicios

Cálculo de complejidad iterativa (II)

```
función conteo(n, a)
    count := 0
    i := 1
    mientras i < n
        j := n
        mientras j > 1
            count := count + 1
            j := j / 2
        si mod(a,2) = 0
            i := i * 2
        si no
            i := i + 1
    devuelve count
```

Ejercicios

Cálculo de complejidad recursiva (I)

Calcula el orden de complejidad de la siguiente relación de recurrencia:

$$T(n) = 4T\left(\frac{n}{2}\right) \qquad T(1) = n$$

Ejercicios

Cálculo de complejidad recursiva (II)

```
función recursiva(v)
    si |v| <= 1
        devuelve v[1]
    si v[1] < v[2]
        devuelve recursiva(v[2:])
    si no
        x := 0
        para i := 1 hasta |v|
            para j := i hasta |v|
                    x := v[i] + v[j]
        devuelve x + recursiva(v[:-1])
```