Tema 3:
Complejidad

GRADO EN INGENIERÍA DEL SOFTWARE

Salvador Sánchez y Sergio Cavero

Universidad
Rey Juan Carlos

Curso 2024-2025

Contenidos

- Definición
- Medidas de complejidad
- Notación O grande
- Cálculo de complejidades
- Ejercicios propuestos

Definición

- La **complejidad** de un algoritmo mide cómo de eficiente es para resolver el problema para el que ha sido diseñado.
- Mide cuánto tiempo y espacio (memoria) requiere para producir una solución.
- Estas medidas:
  - No son absolutas (segundos, bytes…), sino relativas al tamaño del problema
  - Son independientes de la máquina, la plataforma y el lenguaje utilizado

Medidas de la complejidad

- **Complejidad temporal**: consiste en contar el número de operaciones básicas (ej. sumas, multiplicaciones) que realiza el algoritmo.

- **Complejidad espacial**: consiste en determinar la cantidad de memoria (en bytes o bits) que emplea el algoritmo al ejecutarse.
  - Se estudiará con más detalle en cursos posteriores

Complejidad temporal

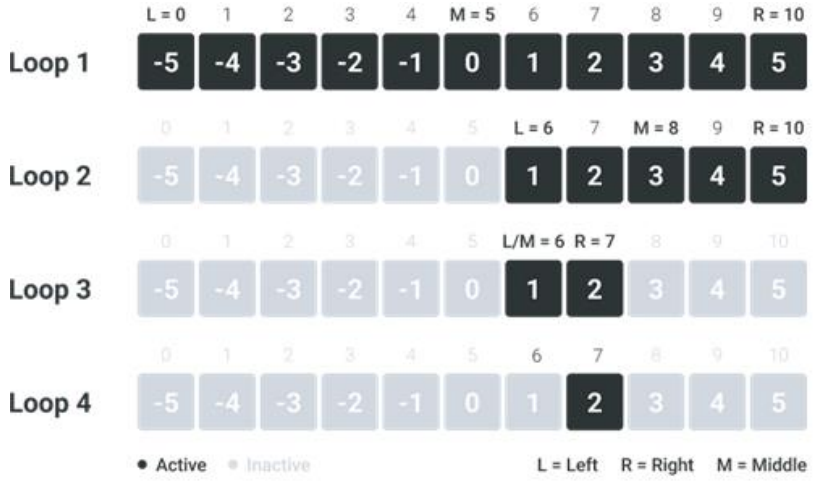
- La emplearemos para comparar diferentes algoritmos que resuelven una misma tarea:
  - Preferiremos siempre el algoritmo que realice la tarea en el menor número de operaciones (más eficiente).

- Dado que el tiempo de ejecución de un algoritmo puede variar entre diferentes entradas del mismo tamaño, consideraremos siempre la complejidad temporal del **caso más desfavorable**

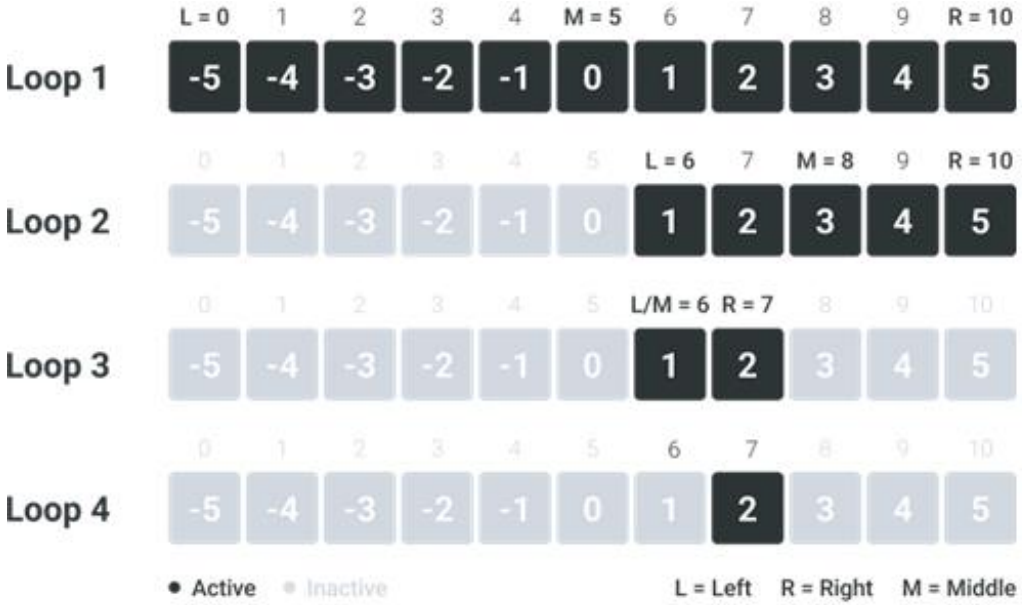
Complejidad temporal - Ejemplo

- Búsqueda lineal: En el peor caso (buscar el 5 o uno que no esté) hace tantos accesos como elementos tiene el array, es decir: n

- Búsqueda binaria: En el peor caso (elemento que no está) hace muchas menos operaciones, 4 en nuestro ejemplo
  - $\log_2(10)=4$
  - Se demuestra que el número de operaciones es log(n)



Complejidad temporal - Ejemplo



Notación “O grande” (Big-O)

- Notación para comparar la complejidad de diferentes algoritmos.
- Definiremos la complejidad temporal en función de n utilizando esta notación
- Estableceremos una cota superior asintótica, es decir, una función que hace de cota superior de la función “número de operaciones de nuestro algoritmo”

Comparativa de complejidades

O(1): Constante (la mejor)

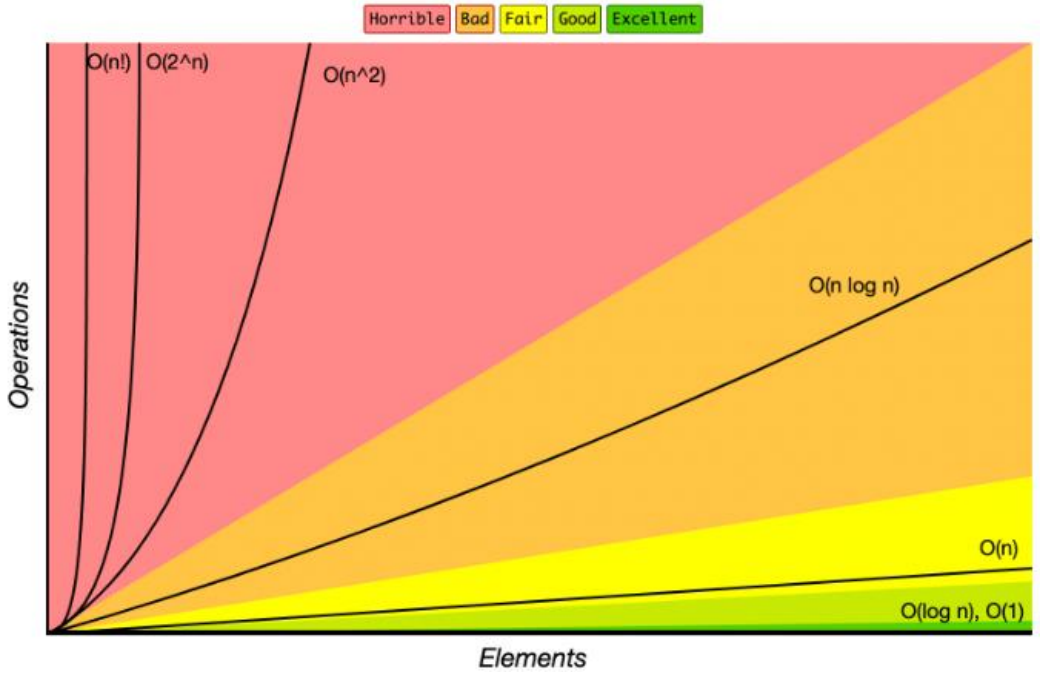
O(log n): Buena

O(n): Aceptable

O(n log n): Mala

O(n^2),O(2^n),O(n!): Muy mala

T(n)	n = 100	n = 200
log(n)	1 h.	1.15 h.
n	1 h.	2 h.
nlog(n)	1 h.	2.30 h.
n^2	1 h.	4 h.
n^3	1 h.	8 h.
2^n	1 h.	1.27*10^{30} h.



Cálculo de complejidades

- **Regla de la Suma**: si ejecutamos dos fragmentos de código la complejidad total será la del fragmento dominante: O(max(A,B)))
- **Regla del Producto**: cuando un algoritmo se ejecuta como parte de otro, la complejidad final es el producto de complejidades: O(A)·O(B)
- **Regla del producto por constante**: si dos fragmentos de código tienen complejidades O(N) y O(kN) para alguna constante k, en complejidad asintótica el valor es el mismo: O(kN) = O(N)
  - Es decir, se ignoran las constantes ya que importa el orden del algoritmo, no el cálculo exacto:
    - 5 x O(1) → O(1)
    - O(n/3) → O(n)

Cálculo de complejidades

- Asignaciones y expresiones simples, accesos a elementos de arrays, lectura y escritura de datos simples, llamadas a funciones → O(1)
- Instrucciones en secuencia: el máximo de todas ellas
- Instrucciones selectivas: complejidad máxima de las alternativas
- Cuando el tamaño de entrada se reduce cada vez a la mitad por causa de una iteración o el uso de la recursividad, etc. complejidad logarítmica: O(log n)

Cálculo de complejidades

- Complejidad constante:

```
x:= x+1          → O(1)
y:= 5*100        → O(1)
writeln(x,y)     → O(1)
```

- Complejidad lineal:

```
x:= 10+1             → O(1)
for i:=1 to n do     → n
  writeln(i,i*i)     → O(1)
```

→ $O(1)+n \cdot O(1) \rightarrow O(1)+O(n) \rightarrow O(n)$

Cálculo de complejidades: bucles

- Si el bucle hace un número fijo de iteraciones: O(1):
  - `for i:= 1 to K do sentencia_simple` → K * O (1) = O (1)
- Cuando n es parte de los límites del bucle, complejidad lineal: O(n)
  - `for i:= 1 to n do sentencia_simple` → n * O (1) = O (n)
- Bucles anidados dentro del algoritmo… si hay dos, complejidad cuadrática: $O(n^2)$, etc.
- Bucles anidados no estrictamente n:
  - `for i:= 1 to n do for j:= 1 to i do sentencia_simple`
  - 1 + 2 + 3 + … + n → n * (1+n)/2 = $O(n^2)$

Cálculo de complejidades: bucles

- Complejidad cuadrática:

```
for i:=1 to n do          → n
  for j:=1 to n do        → n
    writeln(i+j)          → O(1)
```

$$n \cdot n \cdot O(1) \rightarrow n^2 \cdot O(1) \rightarrow O(n^2)$$

Cálculo de complejidades: bucles

- Complejidad cuadrática:

```
for i:=1 to n do          → O(n)
  writeln(i)
for j:=1 to n do
  for k:=1 to j do        → O (n²)
    writeln(i+j)
```

$$O(n) + O(n^2) \rightarrow O(n^2)$$

Cálculo de complejidades: recursividad

- Recursividad: expandir recurrencias y estudiar caso base y general
- Ejemplo: Factorial(n)
  - Factorial(0) = 1 → O(1)
  - Factorial (n) = n * Factorial (n-1) → O(1 + 1 + 1 + .. 1) n+1 veces

O(n+1) → O(n)

Ejercicios propuestos

Ejercicios

1. Algoritmo iterativo para hallar el factorial de un número natural n:

```
fact := 1;
for i:= 1 to n do
  fact := fact * i;
```

2. Algoritmo para buscar secuencialmente un elemento en un vector v con n elementos.

```
function Buscar(v:TipoVecor; buscado:integer):boolean;
    i := 1;
    while (i<=n) and (v[i]<>buscado) do
      i := i + 1;
    Buscar:= i<=n;
```

Ejercicios

3. Bucle anidado:

```
x := 0;
for i:= 1 to n do
  for j:= 1 to n do
    for k:= 1 to n do
      x := x + 1;
```

4. Bucle anidado:

```
x := 0;
for i:= 1 to n do
  for j:= 1 to i do
    for k:= j to n do
      x := x + 1;
```