

# TEMA V: NOCIONES DE COMPLEJIDAD COMPUTACIONAL



# OBJETIVOS GENERALES

- 1. Conocer el concepto de algoritmo.**
- 2. Complejidad de algoritmos.**



# OBJETIVOS ESPECÍFICOS

- ✓ Conocer el concepto de algoritmo.
- ✓ Reconocer algunos tipos de algoritmos.
- ✓ Conocer el significado de las funciones  $O$ ,  $\Omega$  y  $\Theta$ .
- ✓ Complejidad algorítmica
- ✓ Problemas  $P$  y  $NP$



# BIBLIOGRAFÍA

- “Matemática Discreta y sus aplicaciones”. K. H. Rosen. McGraw-Hill, 2004.
- “Matemática discreta y combinatoria”. R. P. Grimaldi. Addison-Wesley Iberoamericana, 1997.
- “Matemáticas discretas”, R. Johnsonbaugh. Ed. Pearson Prentice Hall, 1999.



# DESARROLLO TEÓRICO

IV.1 Introducción.

IV.2 Algoritmos.

IV.3 Crecimiento de funciones.

IV.4 Complejidad algorítmica.

# 1. INTRODUCCIÓN



Muchos problemas pueden resolverse considerándolos como casos especiales de un problema general. Muchos de estos problemas generales se pueden resolver mediante una secuencia de pasos conocida como **algoritmo**.

Una consideración importante en relación con un algoritmo es su **complejidad computacional**. Esto es, qué recursos computacionales se necesitan para usar este algoritmo en la resolución de un problema de un tamaño específico. Para medir la complejidad de un algoritmo utilizaremos las notaciones  $O$  y  $\Theta$ .

## 2. ALGORITMOS





En matemática discreta aparecen muchas clases de problemas genericos. Cuando se nos presentan tales problemas, lo primero que tenemos que hacer es construir un modelo que traduzca el problema a un contexto matemático. Sin embargo, fijar este modelo matemático sólo es parte de la solución, para completarla se necesita un procedimiento que siga una secuencia de pasos que conduzca a la respuesta deseada. Tal secuencia de pasos se le llama **algoritmo**. Por tanto,

Un **algoritmo** es un conjunto finito de instrucciones precisas que resuelven un problema o realizan un cálculo.



Ya conocemos algunos ejemplos en esta asignatura: algoritmo de la división, algoritmo de Euclides, algoritmo chino del resto,...; y otros que se han usado en clase de prácticas: cálculo de una tabla de verdad, producto cartesiano, partes de un conjunto, retículos, álgebras de Boole,...

No obstante, recurriremos en este capítulo a otros algoritmos tipo bien conocidos por el alumnado de Ingeniería Informática, como encontrar el mayor o menor elemento de una lista finita, búsqueda de un elemento en una lista finita de elementos (búsqueda lineal y búsqueda binaria), ordenación de una lista finita de elementos (la ordenación de burbuja y ordenación por inserción), etc.



El término algoritmo es una degeneración del nombre del astrólogo y matemático del siglo IX Mohamed Ben Musa Al-Jowarizmi, cuyo libro sobre numerales hindúes “Algoritmi de Numero Indorum” es la base de la notación decimal que hoy en día usamos. Los europeos aprendieron álgebra por primera vez a través de sus textos. La palabra álgebra proviene de vocablo al-jabr, parte del título del libro “Hisab al-jabr w'al muquabala”. Originalmente, una variante de la palabra algoritmo se uso para definir las reglas usadas para hacer aritmética usando notación decimal. El término evolucionó a algoritmo en el siglo XVIII, para describir la disciplina de la aritmética con numerales hindúes. Con el creciente interés por todo lo relativo a la computación, este concepto adquirió un significado más genérico, pasando a incluir procedimientos utilizados para resolver problemas.



Hay varias propiedades que generalmente comparten los algoritmos. Estas son:

- Entrada: valores de entrada que son elementos de un conjunto dado.
- Salida: para cada conjunto de valores de entrada, un algoritmo produce valores de salida que son la solución del problema.
- Definición: Los pasos de un algoritmo deben definirse con precisión.
- Corrección.
- Duración finita.
- Efectividad..
- Generalidad.



**Ejemplo 1.** Encontrar el mayor o menor elemento, en un conjunto o sucesión de elementos finitos para un cierto orden en éste: elegimos uno cualquiera como el mayor (o menor) provisional, por ejemplo el primero, lo comparamos con todos uno a uno y cambiamos el provisional siempre que este cambie, al terminar, el provisional será mayor (o menor) definitivo.

En Mathematica:

```
In[ ]:= Conjunto={3, 6, 2, 1, 0, 7, 8, 5};  
Provisional=Conjunto[[1]];  
Do[  
  If[Provisional<Conjunto[[i]],Provisional=Conjunto[[i]]];  
  ,{i,2,Length[Conjunto]};  
Print["El mayor elemento es: ", Provisional];
```



El problema de localizar un elemento de una lista ordenada se puede encontrar en muchos contextos. Los problemas de este tipo se llaman **problemas de búsqueda** y se puede describir como sigue: localizar un elemento  $x$  en una lista de elementos distintos  $a_1, a_2, \dots, a_n$  o determinar que no está en la lista. La solución es localizar el término  $x$  en la lista (esto es, la solución es  $i$  si  $x = a_i$ ) o es  $0$  si  $x$  no está en la lista.

**Ejemplo 2.** Búsqueda lineal: comienza por comparar  $x$  y  $a_1$ , cuando  $x = a_1$  la solución es el índice de  $a_1$ , es decir,  $1$ , en otro caso  $x$  se compara con el siguiente elemento y así sucesivamente con cada término de la lista hasta que se encuentra una coincidencia. Si tras recorrer toda la lista no se localiza  $x$ , la solución es  $0$ .



En Mathematica:

```
In[]:= Elemento=0;  
Conjunto={3, 6, 2, 1, 0, 7, 8, 5};  
i=1;  
While[Elemento!=Conjunto[[i]] && i!=Length[Conjunto],  
    i++;  
];  
If[Elemento==Conjunto[[i]],  
    Print["El elemento está en la posición: ", i];  
    ,Print["El elemento no está"];  
];
```

### 3. CRECIMIENTO DE FUNCIONES





Para resolver un mismo problema puede que encontremos varios algoritmos, nos interesará determinar cuáles son los mejores (más efectivos) en términos del tiempo o memoria empleadas para resolver el problema, para ello tendremos que analizarlos independientemente del hardware o software donde se implementen, es por ello que nos interesa conocer cómo se comportan dichos algoritmos según la cantidad de datos que manejen, cuántas instrucciones tiene que ejecutar el ordenador o cuánta memoria precisa, tanto en el mejor de los casos, como en el peor. Este análisis lo realizaremos buscando una función que para un tamaño concreto (cantidad de datos) del problema nos dé el número de instrucciones a ejecutar o la memoria usada, y después también estudiaremos cómo se comporta y cómo crece dependiendo del tamaño del problema.



Dadas dos funciones  $f$  y  $g$  de  $\mathbb{R}$  o  $\mathbb{Z}$  en  $\mathbb{R}$ , diremos que  $f(x)$  es  $O(g(x))$  si existen dos constantes  $C$  y  $k$  tales que

$$|f(x)| \leq C|g(x)|, \quad \forall x > k.$$

A  $C$  y  $k$  los llamaremos **testigos** (hay infinitos) y escribiremos:  
 $f(x) = O(g(x))$ .

Dadas dos funciones  $f$  y  $g$  de  $\mathbb{R}$  o  $\mathbb{Z}$  en  $\mathbb{R}$ , diremos que  $f(x)$  es  $\Omega(g(x))$  si existen dos constantes  $C$  y  $k$  tales que

$$|f(x)| \geq C|g(x)|, \quad \forall x > k.$$

Escribiremos:  $f(x) = \Omega(g(x))$ .

Es obvio que si  $C \neq 0$ , entonces  $g(x) = O(f(x)) \Leftrightarrow f(x) = \Omega(g(x))$ .



Dadas dos funciones  $f$  y  $g$  de  $\mathbb{R}$  o  $\mathbb{Z}$  en  $\mathbb{R}$ , diremos que  $f(x)$  es  $\Theta(g(x))$  si  $f(x)$  es  $O(g(x))$  y  $\Omega(g(x))$ . En tal caso diremos que  $f(x)$  es zeta de  $g(x)$  o que  $f(x)$  es del orden de  $g(x)$ .

Se puede ver que  $f(x)$  es  $\Theta(g(x))$  si podemos encontrar dos números reales  $C_1$  y  $C_2$  y un número positivo  $k$  tales que:

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|, \quad \forall x > k.$$

Habitualmente cuando decimos que una función  $f(x)$  es  $\Theta(g(x))$ , la función  $g(x)$  es simple:  $1$ ,  $\log(x)$ ,  $x$ ,  $x \cdot \log(x)$ ,  $x^n$ ,  $a^x$ ,  $x!$ .

Equivalen:

1.  $f(x)$  es de orden  $g(x)$ .
2.  $f(x) = O(g(x))$  y  $g(x) = O(f(x))$ .
3.  $f(x) = \Omega(g(x))$  y  $g(x) = \Omega(f(x))$ .



## Propiedades

1. Si  $f(x) = O(g(x))$  y  $g(x) = O(h(x))$  entonces  $f(x) = O(h(x))$ . (Idem para  $\Omega$  y  $\Theta$ ).

2. Si  $f(x) = a_n x^n + \dots + a_1 x + a_0$  (polinomio con coeficientes reales), entonces  $f(x) = O(x^n)$ .

3. Si  $f_1(x) = O(g_1)$  y  $f_2(x) = O(g_2)$ , entonces:

3.1  $(f_1 + f_2)(x) = O(\max\{|g_1(x)|, |g_2(x)|\})$ .

3.2  $(f_1 f_2)(x) = O(g_1(x) g_2(x))$ .

4. Si  $f_1(x) = O(g(x))$  y  $f_2(x) = O(g(x))$  entonces  $(f_1 + f_2)(x) = O(g(x))$ .

5. Si  $f(x) = a_n x^n + \dots + a_1 x + a_0$  (polinomio con coeficientes reales), entonces  $f(x)$  es del orden de  $x^n$ .

## 4. COMPLEJIDAD ALGORÍTMICA



¿Cuándo un algoritmo soluciona de forma satisfactoria un problema? Primero, debe darnos siempre la respuesta correcta y en segundo lugar, debería ser eficiente. ¿Cómo podemos analizar la eficiencia de los algoritmos? Una medida de eficiencia es el tiempo que requiere un ordenador para resolver un problema usando un algoritmo para valores de entrada de un tamaño determinado. Una segunda medida es la cantidad de memoria necesaria para valores de entrada de un tamaño dado. Ambas preguntas están ligadas a la **complejidad computacional**.



Distinguimos entre **complejidad en tiempo**: número de operaciones que tiene que realizar el algoritmo; o **complejidad en espacio**: memoria necesaria. Esta última depende de la estructura de los datos usada en la implementación del algoritmo, por lo que escapa de nuestros objetivos. Nos limitaremos a estudiar la primera.

La complejidad en tiempo se describe en términos del número de operaciones requeridas, en lugar del tiempo de cálculo real, ya que distintos ordenadores necesitan tiempos diferentes para realizar la misma operación básica.



**Ejemplo 1: (Mayor elemento de un conjunto)** Usamos como medida el número de comparaciones (operación básica usada). Fijado como máximo preliminar el elemento inicial de la lista, se realiza una comparación para determinar si se ha llegado al final de la lista y otra comparando el preliminar  $k$  y el segundo elemento de la lista, actualizando el valor del máximo si el segundo término es mayor que el primero. Como se hacen dos comparaciones desde el segundo hasta el  $n$ -ésimo, y una más para salir del bucle cuando  $i = n + 1$ , se realizan exactamente  $2(n - 1) + 1 = 2n - 1$  comparaciones.

Por tanto, este algoritmo es  $\Theta(n)$ , donde  $n$  es el número de elementos de “Conjunto”.





**Ejemplo 2: (Búsqueda lineal)** De nuevo, la operación básica usada es la comparación. En cada paso del bucle se llevan a cabo dos comparaciones: una para ver si se ha llegado al final de la lista y otra para comparar  $x$  y el elemento  $a_i$ . Finalmente, fuera del bucle se hace una comparación más. Por tanto, si  $x = a_i$ , se hacen  $2i + 1$  comparaciones. El número máximo de comparaciones lo hacemos si  $x$  no está en la lista,  $2n + 2$  ( $2n$  para determinar que  $x \neq a_i$  para todo  $i$ , una para salir del bucle y otra fuera del bucle).

Por tanto, una búsqueda lineal requiere a lo más  $O(n)$  comparaciones.



El tipo de análisis que hemos realizado en el ejemplo anterior se denomina **peor caso**: el mayor número de operaciones que hace falta para resolver el problema dado usando el algoritmo para unos datos de entrada de un tamaño fijado.

Otro tipo importante de análisis de complejidad, es el llamado análisis del **caso promedio**. En este tipo se busca el número promedio de operaciones realizadas para resolver un problema considerando todas las posibles entradas de un tamaño fijo.



**Ejemplo 2: (Búsqueda lineal)** Describimos ahora como se comporta este algoritmo suponiendo que  $x$  está en la lista: Si  $x$  está hay  $n$  posibles soluciones. Si  $x$  es el primer elemento, necesitamos tres comparaciones (una para comprobar si se ha alcanzado el final, otra para compara  $x$  con el elemento y otra fuera del bucle). Si  $x$  es el segundo se necesitan dos más y así sucesivamente. Por tanto el número promedio es:

$$\frac{3 + 5 + 7 + \dots + (2n + 1)}{n} = \frac{2(1 + 2 + 3 + \dots + n) + n}{n} = \frac{2\left(\frac{n(n+1)}{2}\right) + n}{n} = n + 2$$

Por tanto, es  $\Theta(n)$ .



La terminología que comúnmente se usa para describir la complejidad de los algoritmos viene dada en la siguiente tabla:

Complejidad	Terminología
$O(1)$	Constante
$O(\log n)$	Logarítmica
$O(n)$	Lineal
$O(n \log n)$	$n \log n$
$O(n^k)$	Polinómica
$O(a^n), a > 1$	Exponencial
$O(n!)$	Factorial



Un problema que se puede resolver utilizando un algoritmo con complejidad polinómica en el peor caso se dice **tratable**, pues se espera que el algoritmo nos de la solución para una entrada de tamaño razonable en un tiempo relativamente corto. Sin embargo, si el polinomio de estimación en notación  $O$  tiene grado alto o los coeficientes son excesivamente grandes, el algoritmo puede necesitar demasiado tiempo para su utilización.

Diremos que un problema es **intratable** si no se puede resolver usando un algoritmo con complejidad polinómica en el peor caso.



En la práctica, hay situaciones en las que un algoritmo puede resolver un problema mucho más rápido en la mayoría de los casos que en el peor caso. En tales casos la complejidad en tiempo en el caso promedio es una medida más acertada que el peor caso. Muchos problemas en la industria son intratables, pero pueden resolverse para todos los conjuntos de datos que aparecen en la vida real. Otra forma de manejar este tipo de problemas es buscar soluciones aproximadas del problema en lugar de las soluciones exactas.

Existe algunos problemas para los cuales incluso se puede probar que no existen algoritmos que puedan resolverlos, estos problemas se les llama **irresolubles** o **no resolubles**, en oposición a los problemas **resolubles**, para los que existen algoritmos que los resuelven.



Se dice que un problema pertenece a la **clase NP** si se puede comprobar en tiempo polinómico que una solución efectivamente lo es. Los problemas tratables se dicen que pertenecen a la **clase P**. Por último, una clase importante de problemas son los llamados **problemas NP-completos**, con la propiedad de que si alguno de estos problemas se puede resolver haciendo uso de un algoritmo con complejidad polinómica en el peor caso, entonces todos ellos se pueden resolver por medio de algoritmos con complejidad polinómica en el peor caso.



**Ejemplo 3: (Búsqueda binaria)** sólo es válido para conjunto finitos donde suponemos ordenados los elementos por alguna relación de orden total: partimos el conjunto en dos subconjuntos con igual, si es posible, número de elementos (los pequeños y los grandes) y comparamos con el mayor de los pequeños y el menor de los grandes, después de concluir en cuál de los dos subconjuntos podría estar, repetimos el proceso para este subconjunto.





En Mathematica:

```
In[]:= Elemento=7;  
Conjunto={1,2,3,4,5,6,7,8,9,10};  
Principio=1;  
Final=Length[Conjunto];  
While[Principio!=Final,  
    Mitad=Quotient[Final-Principio,2];  
    If[Conjunto[[Principio+Mitad]]>=Elemento,  
        Final=Principio+Mitad;;  
    If[Conjunto[[Principio+Mitad+1]]<=Elemento,  
        Principio=Principio+Mitad+1;;Principio=Final];  
];  
If[Elemento==Conjunto[[Principio]],  
    Print["El elemento está en la posición: ", Principio];,  
    Print["El elemento no está"];  
];
```



Pasos:

(1) Principio=1, Final=10, Mitad=4

$\{1,2,3,4,5\}$        $\{5,6,7,8,9,10\}$

(2) Principio=6, Final=10, Mitad=2

$\{6,7,8\}$        $\{9,10\}$

(3) Principio=6, Final=8, Mitad=1

$\{6,7\}$        $\{8\}$

(4) Principio=6, Final=7, Mitad=1

$\{6\}$        $\{7\}$

(5) Principio=7, Final=7

Se compara “Elemento” y 7



Si suponemos que  $n = 2^k$  ( $k = \log_2 n$ ), en otro caso podemos considerar que la lista de elementos es parte de otra más larga con un número de elementos potencia de 2.

En cada paso del algoritmo,  $i$  y  $j$ , las posiciones del primero y último término de la lista restringida considerada se comparan para ver que la nueva lista tiene más de un término. Si  $i < j$ , se realiza una comparación para determinar si  $x$  es mayor que el término central de la lista restringida.



En el primer paso, la búsqueda se restringe a  $2^{k-1}$  elementos (hasta aquí se han hecho dos comparaciones). Este proceso continua realizando dos comparaciones en cada paso. Por tanto, tenemos que realizar  $2k + 2 = 2 \log n + 2$  comparaciones. Así, que una búsqueda binaria requiere como máximo  $\Theta(\log n)$  comparaciones.



Otro tipo de algoritmos muy comunes son los dedicados a ordenar los elementos de una lista. Si tenemos una lista de elementos de un conjunto para los que conocemos una forma de ordenar. Una **ordenación** es colocar estos elementos en una lista en la cual los elementos se dispongan en orden creciente.


**Ejemplo 4. Ordenación de burbuja**, es uno de los más simple, pero no de los más eficientes. Damos sucesivas pasadas sobre la sucesión, intercambiando las posiciones de los elementos contiguos cuando no estén ordenados hasta que demos una pasada en la que no intercambiamos ninguna posición.

Es  $\Theta(n^2)$ .



**Ejemplo 5. *Ordenación por inserción.*** Se añaden los elementos de uno en uno, colocándolos en su posición correcta (ordenados).

**Ejemplo 6. *Algoritmos voraces.*** Son algoritmos que hacen la que aparentemente es la mejor elección a cada paso. Por ejemplo, si queremos devolver el cambio exacto de cierta cantidad usando una determinada clase de monedas y siempre elegimos la moneda más grande que podamos a cada paso.



```
In[]:= Cantidad=5.35;  
Entregado=50.0;  
Monedasybilletes={10,5,2,1,0.5,0.25,0.10,0.05,0.02,0.01};  
  
Cambio={};  
If[Entregado>Cantidad,  
While[Entregado>Cantidad,  
i=1;  
While[Entregado-Monedasybilletes[[i]]<Cantidad,  
i++;  
];  
AppendTo[Cambio,Monedasybilletes[[i]]];  
Entregado=Entregado-Monedasybilletes[[i]];  
];  
Print["Cambio: ",Cambio];  
,  
Print["No ha entregado suficiente dinero"]  
]
```