



**Universidad Nacional Autónoma de México
Facultad de Ingeniería
Estructuras de datos y algoritmos**

**Tema 3:
ANÁLISIS BÁSICO DE ALGORITMOS**

4 Análisis básico de algoritmos

Objetivo: Describir los tipos de complejidad P, NP y NPC.

4 Complejidad de algoritmos

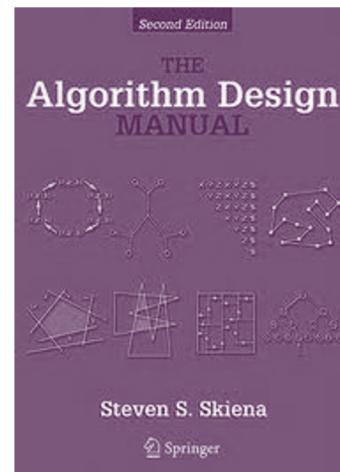
4.1 Complejidad.

4.1.1 P.

4.1.2 NP.

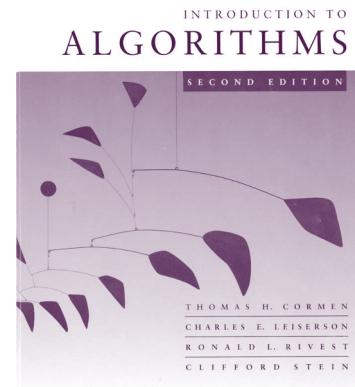
4.1.3 NP completos.

Bibliografía



The algorithm design manual.
Steven S. Skiena, Springer.

Bibliografía



Introduction to Algorithms.

***Thomas H. Cormen, Charles E. Leiserson,
Ronald L. Rivest, Clifford Stein, McGraw-Hill.***



4.1 Complejidad.

4.1 Complejidad.

La complejidad de un algoritmo se puede determinar a través de la notación O y los límites asintóticos que éste dibuja.

Dependiendo del número máximo de instancias de un algoritmo y los límites asintóticos del mismo, existen 3 tipos de complejidades que puede describir un algoritmo:

- P
- NP
- NP completos

P

Se conoce como clase de problema P a aquellos algoritmos que son resolubles en un tiempo polinomial.

Más específicamente, los algoritmos clase P son problemas que pueden resolverse en un tiempo $O(n^k)$ para alguna constante 'k', donde 'n' es el tamaño de la entrada del problema.

Una función f es una función correcta de complejidad si es no decreciente y existe una máquina Turing M_f de k cintas con entrada y salida tal que para toda entrada x aplica que $M_f(x) = f(|x|)$, y además M_f para después de $O(|x| + f(|x|))$ pasos utiliza $O(f(|x|))$ espacio en memoria.

tiempo	determinista $\text{TIME}(f)$ no determinista $\text{NTIME}(f)$
espacio	determinista $\text{SPACE}(f)$ no determinista $\text{NSPACE}(f)$

La clase P abarca las siguientes funciones de complejidad computacional:

Tiempo	Clase	Ejemplo
Problemas con algoritmos eficientes		
$\mathcal{O}(1)$	P	si un número es par o impar
$\mathcal{O}(n)$	P	búsqueda de un elemento entre n elementos
$\mathcal{O}(n \log n)$	P	ordenación de n elementos
$\mathcal{O}(n^3)$	P	multiplicación de matrices
$\mathcal{O}(n^k)$	P	programación lineal

NP

La clase NP consiste de aquellos problemas que son verificables en tiempo polinomial. Esto es, si de alguna manera se obtiene un certificado de solución para un algoritmo, ese certificado se puede comprobar en tiempo polinomial en el tamaño de entrada del problema.

NP es un nombre compuesto por No-determinista y por Polinómico. Lo que significa que su solución no es polinómica porque no se puede definir una ecuación que lo resuelva, es decir, para un valor de entrada existen muchos posibles resultados.

NP completo

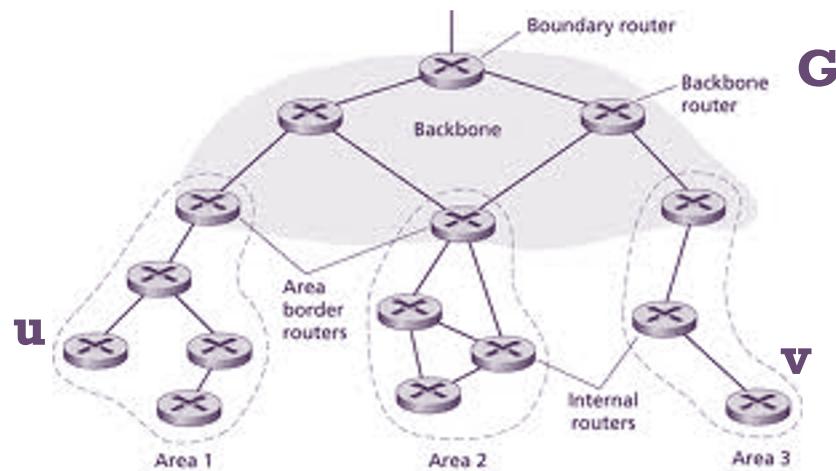
Un problema pertenece a la clase NPC (NP Completo) si está en NP y es tan difícil como cualquier problema en NP.

La mayoría de los teóricos científicos en computación creen que los problemas NP-Completos no se pueden resolver, debido a que de todos los problemas que han estudiado hasta el momento, no hay alguno que tenga una solución en tiempo polinomial.

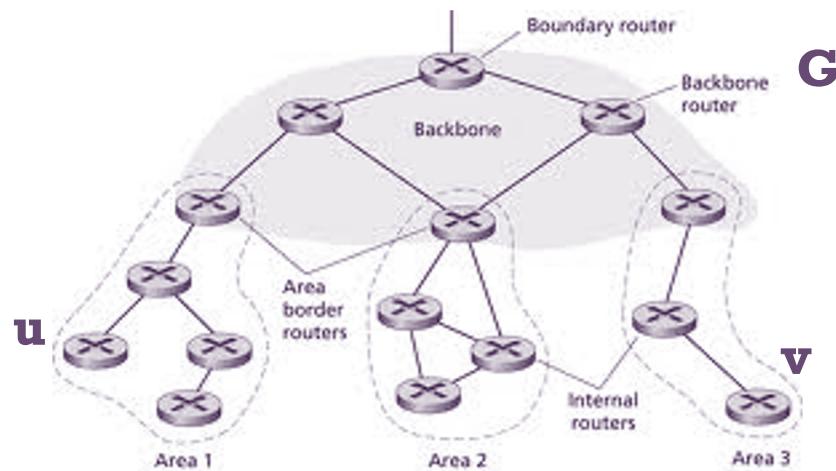
Un buen diseñador de algoritmos necesita entender los primeros conceptos de la teoría de NP-completos.

Si se puede demostrar que un problema es NP-completo, se cuenta con pruebas suficientes de que dicho problema es intratable. Como ingeniero es mejor gastar tiempo en desarrollar un algoritmo aproximado o resolver un caso particular del problema, antes que buscar una solución exacta al problema cuando éste pertenece a la clase NPC.

En la creación de algoritmos se busca optimizar el tiempo de ejecución. Por ejemplo, al tratar de encontrar la ruta más corta entre dos elementos, se tiene:



Los algoritmos NPC no ven un problema de optimización de tiempo sino un problema de decisión: ¿Existe una ruta de u a v ? Donde la respuesta puede ser sí (1) o no (0).



Existe una conveniente relación entre los problemas de optimización y los problemas de decisión. Usualmente, es posible moldear un problema de optimización a uno de decisión limitando los valores a ser optimizados.

El problema de la ruta más corta, se puede moldear como un problema de decisión si, dados como datos una gráfica G , un par de vértices u y v y un entero k , existe una ruta de u a v que tenga como máximo k pasos.

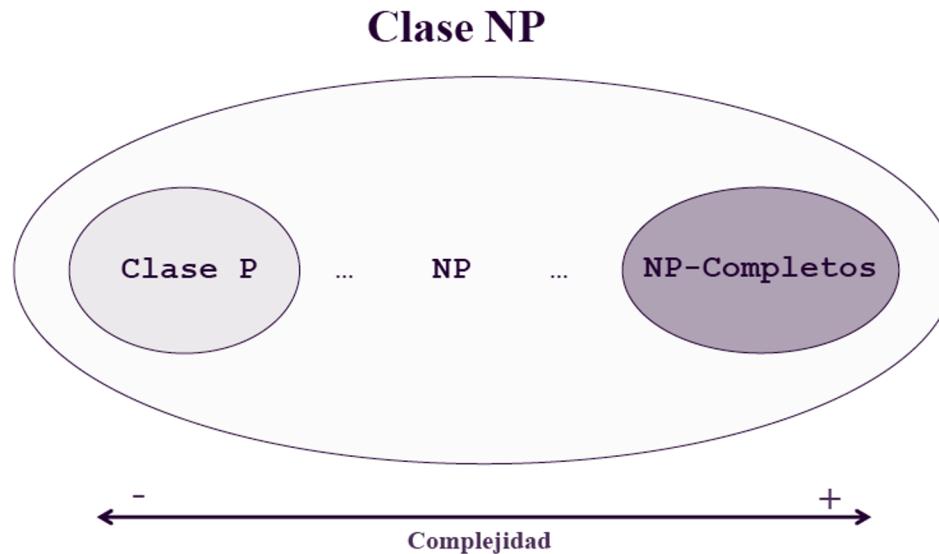
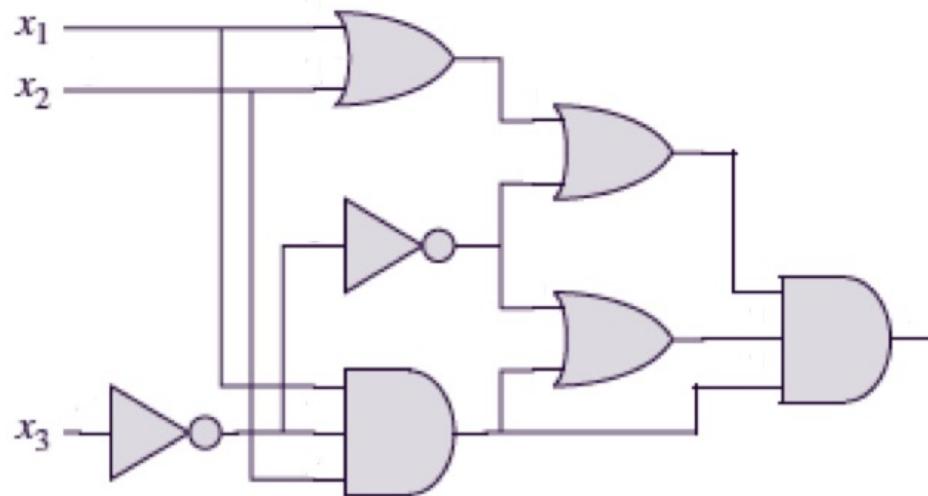
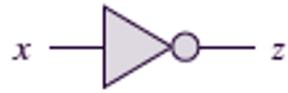


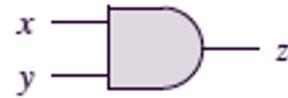
Figura 1. Relación entre las clases P, NP y NPC.

Dado el siguiente circuito booleano combinacional formado por compuertas lógicas NOT, AND y OR, ¿es posible satisfacer el circuito, es decir, ¿existe un conjunto de entradas que generen como salida 1?

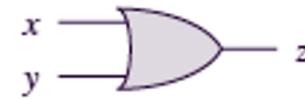




x	$\neg x$
0	1
1	0



x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1



x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

NOT

AND

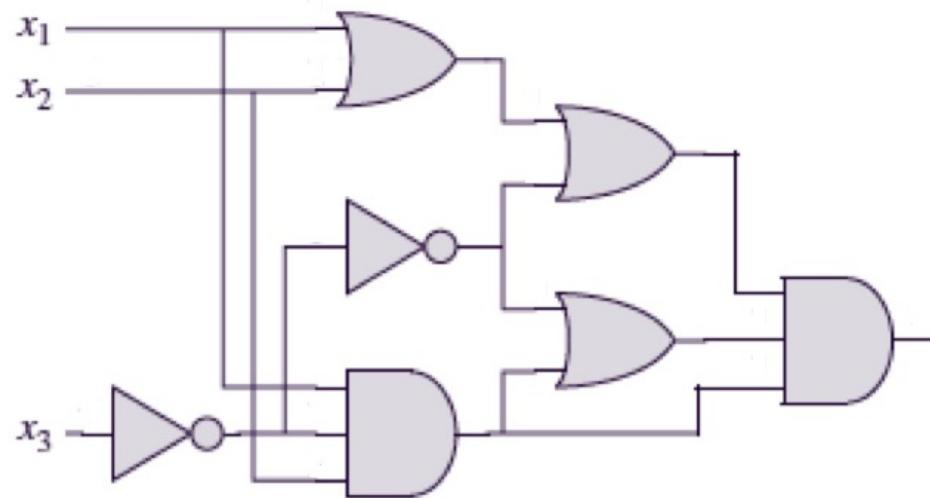
OR

Se parte de que el circuito booleano combinacional no contiene ciclos, es decir, se posee una gráfica G , con n nodos con cada elemento combinacional y k vértices a la salida de los circuitos, de tal manera que existe un vértice (u,v) que conecta la salida del elemento u con la entrada del elemento v . Por lo tanto, se puede afirmar que G no es cíclica.

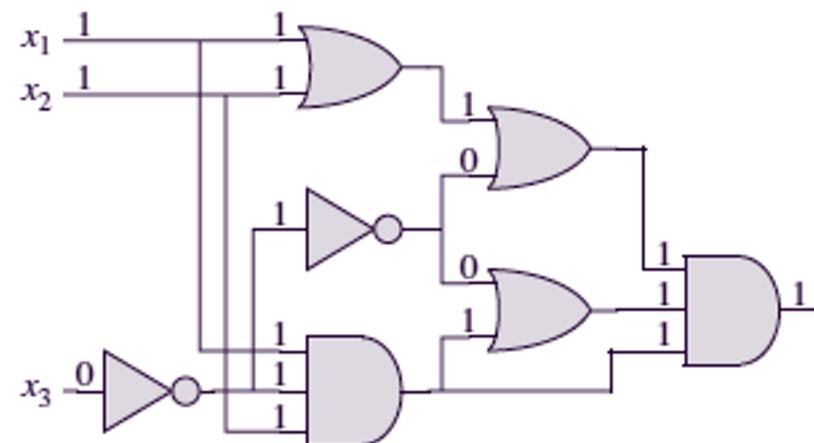
Para un circuito booleano combinacional una asignación válida es aquel conjunto de valores de entrada booleanos.

Se dice que un circuito se puede satisfacer si existe una asignación válida (valor de entrada) que provoca que la salida del circuito sea 1.

¿Qué combinaciones de entrada satisfacen al circuito?



¿Qué combinaciones de entrada satisfacen al circuito?

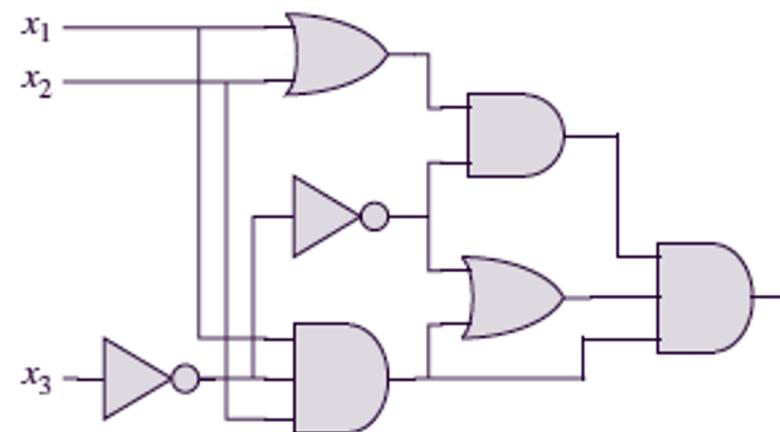


El tamaño de un circuito booleano combinacional está determinado por el número de valores booleanos de entrada y el número vértices (enlaces) del circuito.

Dado un circuito C, se puede determinar si éste se puede satisfacer únicamente revisando las posibles asignaciones de entrada.

El problema surge cuando se tienen k entradas, porque existen 2^k asignaciones posibles, lo que hace el algoritmo super-polinomial. Con base en lo anterior, se puede afirmar que un circuito combinacional booleano pertenece a la clase NPC.

Dado el siguiente circuito booleano combinacional formado por compuertas lógicas NOT, AND y OR, ¿es posible satisfacer el circuito, es decir, ¿existe un conjunto de entradas que generen como salida 1?



En la práctica, la mayoría de los problemas son de la clase NP-Completos, sin embargo, esa no es una razón suficiente para declinar la implementación de un algoritmo.

De manera empírica existen, por lo menos, tres maneras diferentes para tratar de implementar algoritmos del tipo NPC.

- Dado un algoritmo con un tiempo de ejecución exponencial, si la entrada del problema es pequeña (o se puede acotar), el algoritmo se puede implementar.

n	$f(n)$	2^n
10		1 μ s
20		1 ms
30		1 sec
40		18.3 min
50		13 days
100		4×10^{13} yrs
1,000		
10,000		
100,000		
1,000,000		
10,000,000		
100,000,000		
1,000,000,000		

- Es posible acotar algunos problemas que tienen un tiempo de ejecución polinomial.

n	$f(n)$	n^2
10		0.1 μ s
20		0.4 μ s
30		0.9 μ s
40		1.6 μ s
50		2.5 μ s
100		10 μ s
1,000		1 ms
10,000		100 ms
100,000		10 sec
1,000,000		16.7 min
10,000,000		1.16 days
100,000,000		115.7 days
1,000,000,000		31.7 years

- **Partiendo de una solución en tiempo polinomial, es posible encontrar una solución óptima cercana, es decir, encontrar un algoritmo aproximado para llegar a la solución del problema que no sea tan costosa en tiempo (o espacio).**

Hardware y software

Supóngase que se quiere comparar el procesamiento de una computadora rápida A corriendo un algoritmo de inserción ordenada contra una computadora más lenta B ejecutando una unión ordenada de elementos.

Cada algoritmo debe ordenar un arreglo de un millón de elementos. La computadora A ejecuta 1000 millones de instrucciones por segundo y la computadora B ejecuta 10 millones de instrucciones por segundo (A es 100 veces más rápida que B).

Para hacer más dramático el asunto, supóngase que el programador más astuto del mundo codificó el algoritmo de inserción en lenguaje máquina para la computadora A y el código resultante para requiere $2n^2$ instrucciones para ordenar n números ($c_1 = 2$).

Por otro lado, el código de la unión ordenada fue realizado por un programador promedio usando un lenguaje de alto nivel con un compilador ineficiente y toma $50n \log n$ instrucciones ($c_2 = 50$).

Si se ordenan 1 millon de elementos, los tiempos que tardan A y B son, respectivamente:

$$\frac{2*(10^6)^2 \text{ [instrucciones]}}{10^9 \text{ [instrucciones/s]}} = 2000 \text{ [s]}$$

$$\frac{50 * 10^6 \lg(10^6) \text{ [instrucciones]}}{10^7 \text{ [instrucciones/s]}} \approx 100 \text{ [s]}$$

Debido a que se utilizó un algoritmo cuyo tiempo de ejecución crece poco a poco, incluso con un compilador débil, el proceso en B se ejecutó 100 veces más rápido que el proceso en A.

Análisis y diseño avanzado de algoritmos

Regresando al problema del producto de dos números $a*b$, ¿es posible mejorar su tiempo de ejecución, es decir, transformar el comportamiento $O(n)$ a uno más eficiente similar a $O(\log n)$ sin utilizar el algoritmo de los campesinos rusos?

Si se realiza un análisis matemático del problema se tiene que el producto de dos números está dado por la suma de uno las veces del otro, es decir:

a veces

$$a * b = \overbrace{(b + b + \dots + b)}^{a \text{ veces}}$$

a/2 veces a/2 veces

$$a * b = \overbrace{(b + b + \dots + b)}^{a/2 \text{ veces}} + \overbrace{(b + b + \dots + b)}^{a/2 \text{ veces}}$$

Por lo tanto, el producto de dos números se puede representar como:

$$a * b = \overbrace{b + b + \dots + b}^{a/2 \text{ veces}}$$

La fórmula anterior aplica si el número es par. Si el número es impar, se puede omitir un elemento de b, de tal manera que las sumas se realicen un número par de veces, para al final sumar el elemento faltante, es decir:

$$a * b = \overbrace{2(b + b + \dots + b)}^{a/2 \text{ veces}} + b$$

Utilizando un algoritmo del tipo divide y vencerás es posible realizar el cálculo iterativo (o recursivo) del problema anterior. Lo último que hay que tener en cuenta es el caso base o caso final, en este caso el número de iteraciones debe ser mayor a cero.

A continuación se presenta un algoritmo más eficiente para realizar la multiplicación de manera recursiva.

```
FUNC multRec(x: ENTERO, y ENTERO) DEV ENTERO
    SI (x = 0) ENTONCES
        DEV 0;
    FIN_SI
    DE_LO CONTRARIO
        SI ((x%2) = 0) ENTONCES
            DEV 2*(multRec(x/2,y))
        FIN_SI
        DE_LO CONTRARIO
            DEV y+2*multRec((x-1)/2,y)
        FIN_DLC
    FIN_DLC
FIN_FUNC
```

“Beware of bugs in the above code; I have only proved it correct, not tried it.”

Donald Knuth

(American computer scientist, mathematician, and professor emeritus at Stanford University.)

Para cada función en la siguiente tabla, determinar el tamaño más grande que puede tomar n para que el problema pueda ser resuelto en un tiempo t , asumiendo que el algoritmo para resolver el problema toma $f(n)$ microsegundos.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$							
\sqrt{n}							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

4 Análisis básico de algoritmos

Objetivo: Describir los tipos de complejidad P, NP y NPC.

4.1 Complejidad.

4.1.1 P.

4.1.2 NP.

4.1.3 NP completos.