Complejidad computacional

Joaquim Borges
Robert Clarisó
Ramon Masià
Jaume Pujol
Josep Rifà
Joan Vancells
Mercè Villanueva

PID_00215270

Universitat Oberta de Catalunya

Índice

Introducción

Este módulo es una introducción a la Teoría de la Complejidad Computacional. Esta rama de la Informática Teórica estudia cuáles son los límites prácticos de cálculo de los ordenadores.

Ante un problema, el objetivo será medir su dificultad inherente: la cantidad de recursos computacionales (tiempo de ejecución y espacio de memoria) que necesita un ordenador para calcular su solución. Esta complejidad temporal y espacial se estudia en términos del tamaño de los datos de entrada.

A grandes rasgos, queremos distinguir entre aquellos problemas donde el proceso de resolución es escalable incluso para grandes volúmenes de datos (problemas *tratables*) de aquellos donde el consumo de recursos se dispara rápidamente hasta llegar a ser prohibitivo (problemas *intratables*). Por ejemplo, ¡en problemas intratables es frecuente hablar de tiempos de ejecución superiores a la vida del universo! Se trata, pues, de problemas que nunca se podrán resolver en la práctica por mucho que avance la capacidad de cálculo de los ordenadores.

En este módulo, formalizaremos el concepto de problema y las técnicas para medir su complejidad computacional. Presentaremos diversos ejemplos de problemas "famosos" de diferentes ámbitos de la Informática, caracterizando su complejidad.

Además, para facilitar su clasificación, definiremos varias clases de problemas de una complejidad equivalente y estudiaremos las relaciones entre las clases. Por último, introduciremos una técnica llamada *reducción* que permite comparar la complejidad de dos problemas diferentes. De esta manera, ante un problema nuevo seremos capaces de establecer a qué clase pertenece y decidir si es "igual de complejo" o "más complejo" que otros problemas conocidos.

1. Concepto de problema

El punto de partida de la Teoría de la Complejidad es el concepto de *problema que se puede resolver con un ordenador*.

Definición 1

Un **problema computacional** es un enunciado donde, partiendo de una información de entrada (**instancia**), se pide una respuesta finita (**solución**) con unas características determinadas.

Formalmente, un problema Prob es una función $\text{Prob} : \mathcal{I} \to \mathcal{S}$ de un conjunto de instancias $\mathcal{I}$ a un conjunto de soluciones $\mathcal{S}$. $\text{Prob}(x)$ denotará la solución al problema Prob con entrada x.

$$x \in \mathcal{I} \to \boxed{\text{Prob}} \to \text{Prob}(x) \in \mathcal{S}$$

Ejemplo 1

RC: "Dado un número real no negativo x, calcular su raíz cuadrada."

Este problema corresponde a una función $\text{RC} : \mathbb{R}^+ \to \mathbb{R}^+$, concretamente la función $\text{RC}(x) = \sqrt{x}$. Por ejemplo, tendremos que $\text{RC}(4) = 2$ y $\text{RC}(5) = 2{,}2360679$.

Ejemplo 2

BUSQUEDA_ORD: "Dada una secuencia finita de enteros E ordenada en orden creciente y un entero k, decidir si el número k aparece en la secuencia."

En este caso, cada entrada del problema es un par de valores (una secuencia y un entero), mientras que la solución es sencillamente SÍ o NO. Formalmente, podríamos definir la función $\text{BUSQUEDA\_ORD} : \mathbb{Z}^* \times \mathbb{Z} \to \{\text{SÍ,NO}\}$. Tendríamos por ejemplo que $\text{BUSQUEDA\_ORD}(\langle -2,3,45,50\rangle,3) = \text{SÍ}$ mientras que con otra entrada tenemos que $\text{BUSQUEDA\_ORD}(\langle 5,11,29\rangle,24) = \text{NO}$.

Secuencia

Una secuencia es una serie de elementos donde el orden de los elementos es relevante y puede haber repeticiones: $\langle\,\rangle$, $\langle 3,-1\rangle$, $\langle 4,4,15,-2,5,6\rangle$, ...

Notación

Dado un conjunto C, llamaremos C^* al conjunto de todas las secuencias finitas de elementos de C, incluyendo la secuencia vacía $\langle\,\rangle$.

A lo largo de este módulo, veremos muchos ejemplos de problemas computacionales. Algunos de ellos son muy relevantes por sus aplicaciones prácticas y, por lo tanto, es interesante recordarlos y aprender a reconocerlos. Clasificaremos el grado de dificultad de

estos problemas a partir del análisis de los algoritmos que calculan su solución.

1.1. Problemas decisionales, de cálculo y de optimización

Empezaremos la clasificación de los problemas computacionales distinguiendo diferentes categorías según la solución esperada.

Definición 2

Un problema $\mathsf{Prob} : \mathcal{I} \to \mathcal{S}$ es un **problema decisional** o **de decisión** si el conjunto de soluciones $\mathcal{S}$ es {SÍ,NO}. Dicho de otra forma, el problema consiste en establecer si la entrada satisface una cierta propiedad.

Ejemplo 3

PAR: "Dado un número natural n, determinar si es par."

Este problema es una función $\mathsf{PAR} : \mathbb{N} \to$ {SÍ,NO}. Por ejemplo, $\mathsf{PAR}(16) =$ SÍ y $\mathsf{PAR}(7) =$ NO.

Ejemplo 4

CONEXO: "Dado un grafo $G = (V,A)$, determinar si es conexo."

Si llamamos $\mathcal{G}$ al conjunto de todos los grafos, podemos definir este problema como la función $\mathsf{CONEXO} : \mathcal{G} \to$ {SÍ,NO}. Por ejemplo, para los grafos completos K_n tendríamos que $\mathsf{CONEXO}(K_n) =$ SÍ para cualquier valor de n.

Definición 3

Dado un problema decisional $\mathsf{Prob} : \mathcal{I} \to$ {SÍ, NO}, denotamos por $\mathsf{Prob}^\bullet$ el conjunto de las entradas que tienen solución SÍ.

$$\mathsf{Prob}^\bullet = \{x \in \mathcal{I} \mid \mathsf{Prob}(x) = \text{SÍ}\}$$

Utilizando esta notación de conjuntos, la notación $x \in \mathsf{Prob}^\bullet$ es equivalente a $\mathsf{Prob}(x) =$ SÍ y $x \notin \mathsf{Prob}^\bullet$ es equivalente a $\mathsf{Prob}(x) =$ NO.

Ejemplo 5

Podemos definir el problema $\mathsf{PAR}^\bullet$ como el conjunto $\{0,2,4,6,8,10,\ldots\}$. Podemos expresar que $\mathsf{PAR}(8) = \text{SÍ}$ como $8 \in \mathsf{PAR}^\bullet$ y que $\mathsf{PAR}(11) = \text{NO}$ como $11 \notin \mathsf{PAR}^\bullet$.

Ejemplo 6

Llamaremos ProbSI a un problema decisional que tiene solución SÍ para cualquier entrada. Evidentemente, el conjunto $\mathsf{ProbSI}^\bullet = \mathcal{I}$, ya que todas las entradas pertenecen a $\mathsf{ProbSI}^\bullet$. En cambio, si un problema tiene solución NO para cualquier entrada se llama ProbNO y está claro que $\mathsf{ProbNO}^\bullet = \emptyset$. ProbSI y ProbNO nos serán útiles para obtener algunos resultados teóricos a lo largo de este módulo.

Definición 4

Un problema $\mathsf{Prob} : \mathcal{I} \to \mathcal{S}$ es un **problema de cálculo** si el conjunto de soluciones $\mathcal{S}$ es diferente de {SÍ, NO}. En estos problemas, la solución es un valor (un número, un conjunto, un grafo, ...) que se tiene que calcular a partir de la entrada de acuerdo con el enunciado del problema.

$$x \in \mathcal{I} \to \boxed{\text{CÁLCULO}} \to \text{Valor} \in \mathcal{S}$$

Ejemplo 7

SUMATORIO: "Dada una secuencia finita de enteros E, obtener su suma total."

Este problema se puede describir como $\mathsf{SUMATORIO} : \mathbb{Z}^* \to \mathbb{Z}$. Es un problema de cálculo, ya que la solución es un número entero ($\mathcal{S} = \mathbb{Z}$). Tendríamos por ejemplo que $\mathsf{SUMATORIO}(\langle 22, -35, -18, 47, 16\rangle) = 32$.

Ejemplo 8

AISLADOS: "Dado un grafo $G = (V,A)$, determinar el conjunto de vértices de grado cero (aislados)."

El problema $\mathsf{AISLADOS} : \mathcal{G} \to 2^V$ es de cálculo porque tiene como solución un subconjunto de los vértices V del grafo que no tienen ninguna arista incidente. Por ejemplo, para los grafos completos K_n tenemos que $\mathsf{AISLADOS}(K_n) = \emptyset$ para cualquier n. En cambio, en los grafos nulos N_n todos los vértices son aislados, es decir, $\mathsf{AISLADOS}(N_n) = V$.

Notación

Si C es un conjunto, 2^C representa el conjunto formado por todos los subconjuntos de C.

Definición 5

Un problema Prob : $\mathcal{I} \to \mathcal{S}$ es un **problema de optimización** si es un problema de cálculo ($\mathcal{S} \neq \{\text{SÍ, NO}\}$) y la solución tiene que ser óptima (mínima o máxima) de acuerdo con una función de valoración definida en el enunciado del problema. Dicho de otra manera, en un problema de optimización se pide la mejor solución según un cierto criterio.

$$x \in \mathcal{I} \to \boxed{\text{OPTIMIZACIÓN}} \to \text{Óptimo}(\mathcal{S})$$

Ejemplo 9

CPP: "Dado un conjunto de puntos en un espacio bidimensional, donde cada punto está definido por las coordenades (x,y), calcular el par de puntos que son más próximos entre sí."

Este problema puede verse como una función CPP : $2^{\mathbb{R}\times\mathbb{R}} \to (\mathbb{R} \times \mathbb{R}) \times (\mathbb{R} \times \mathbb{R})$. Un ejemplo de solución sería la que se muestra a continuación: $\text{CPP}\big(\{(0,10),(5,20),(-2,0),(30,-50)\}\big) = \big((0,10),(-2,0)\big)$.

El problema CPP es un problema de optimización. La función de valoración que se está optimizando es la distancia entre el par de puntos de la solución. En este caso, el objetivo es minimizar esta función.

Nomenclatura

En inglés, este problema se llama *closest pair problem*.

Ejemplo 10

LCS: "Dadas dos cadenas de caracteres a y b, encontrar la subcadena común más larga entre a y b."

Si Σ es el alfabeto de símbolos sobre el que hemos definido las cadenas de caracteres, podemos describir LCS como la función LCS : $\Sigma^* \times \Sigma^* \to \Sigma^*$. Un ejemplo de solución sería LCS("andromeda", "dromedary") = "dromeda".

LCS es un problema de optimización donde la función de valoración a optimizar es la longitud de la cadena común. El objetivo es maximizar esta función.

Nomenclatura

En inglés, este problema se llama *longest common substring*.

Ejemplo 11

MSP: "Dado un grafo conexo y ponderado, es decir, un grafo $G = (V,A)$ y la función de ponderación w, calcular un árbol generador minimal."

Este problema tiene como entrada un grafo $G = (V,A)$ y una función de ponderación, y como solución un grafo. Es decir, podemos describirlo como la función MSP : $\mathcal{G} \times (A \to \mathbb{R}) \to \mathcal{G}$. La función de valoración en este caso es el peso del árbol generador, que se pretende minimizar.

Nomenclatura

En inglés, este problema se llama *minimum spanning tree*.

Ejercicios

1. Definid los siguientes problemas en forma de función y clasificadlos como problemas de decisión, cálculo u optimización:

a) Dado un grafo conexo y ponderado (G,w) con $G = (V,A)$, y dos vértices $a,b \in V$, determinar el camino más corto entre a y b.

b) Dada una secuencia finita de enteros E y un valor k, contar el número de apariciones de k dentro de E.

c) Dado un grafo G, comprobar si es un grafo completo.

d) Dado un grafo G, encontrar el subgrafo completo de orden máximo.

e) Dada una cadena de caracteres c, indicar si contiene todos los dígitos del 0 al 9.

2. Justificad si son ciertas o falsas las afirmaciones siguientes (repasad el ejemplo 5):

a) Si un grafo G es un árbol, entonces $G \notin \mathsf{CONEXO}^\bullet$.

b) Si $x \notin \mathsf{PAR}^\bullet$, entonces $2 \cdot x \in \mathsf{PAR}^\bullet$.

Soluciones

1. a) $\mathsf{Prob} : \mathcal{G} \times (A \to \mathbb{R}) \times V \times V \to A^*$. Problema de optimización. La función de valoración es el peso del camino y se desea minimizarlo.

b) $\mathsf{Prob} : \mathbb{Z}^* \times \mathbb{Z} \to \mathbb{Z}$. Problema de cálculo.

c) $\mathsf{Prob} : \mathcal{G} \to \{\text{SÍ,NO}\}$. Problema de decisión.

d) $\mathsf{Prob} : \mathcal{G} \to \mathcal{G}$. Problema de optimización. La función de valoración es el orden (número de vértices) del subgrafo completo y se desea maximizarlo.

e) $\mathsf{Prob} : \Sigma^* \to \{\text{SÍ,NO}\}$. Problema de decisión.

2. a) Falsa. Por definición, todos los árboles son grafos conexos y por lo tanto $\mathsf{CONEXO}(G) = \text{SÍ}$. Para ser correcta, la afirmación debería decir $G \in \mathsf{CONEXO}^\bullet$.

b) Cierta. Aunque un número sea impar ($\mathsf{PAR}(x) = \text{NO}$), el doble de cualquier número siempre será par ($\mathsf{PAR}(2x) = \text{SÍ}$).

1.2. Algunos problemas decisionales importantes

En el resto del módulo, nos centraremos únicamente en los problemas decisionales. El motivo de esta elección aparentemente tan restrictiva es que cualquier problema de cálculo o de optimización se puede reformular en versión decisional:

Tipo	Enunciado original	Versión decisional
Cálculo	"Dada una entrada x, calcular el valor de salida $f(x)$"	"Dada una entrada x y un valor k, decidir si $f(x) = k$"
Optimización	"Dada una entrada x, calcular el valor de salida $f(x)$ que maximiza el criterio c"	"Dada una entrada x y un valor límite l, decidir si hay alguna salida $f(x)$ tal que $c(f(x)) \geq l$"

Podemos ver que la versión decisional es "más sencilla" que el problema original de cálculo o optimización: el algoritmo que resuelve el problema original puede solucionar también la versión decisio-

nal. Así pues, si demostramos que la versió decisional es "difícil", tendremos que las versiones de cálculo y optimización también lo son. Es decir, estudiar la complejidad de los problemas decisionales nos da información sobre la complejidad de los problemas de cálculo y optimización asociados. Pero fijémonos en que sólo tenemos información en un sentido: si la versión decisional de un problema es "sencilla", no sabemos si las versiones de cálculo o de optimización también serán "sencillas" o si serán "difíciles".

1.2.1. Satisfactibilidad de fórmulas booleanas

Definición 6

Una **variable booleana** es una variable que puede tomar los valores de verdad 0 (falso) o 1 (cierto).

Una **fórmula booleana** es una combinación de variables mediante operaciones lógicas como la conjunción *y-lógica* ($\wedge$), la disyunción *o-lógica* ($\vee$) o la negación *no* ($\overline{x}$).

Un **literal** es una variable booleana (x) o su negación ($\overline{x}$).

Una **cláusula** es una disyunción (o-lógica) de literales ($x_1 \vee \ldots \vee x_n$).

Una fórmula está en **forma normal conjuntiva (FNC)** si es una conjunción (y-lógica) de cláusulas: $(x_1 \vee \ldots \vee x_{n_1}) \wedge \ldots \wedge (y_1 \vee \ldots \vee y_{n_k})$.

Ejemplo 12

La fórmula booleana siguiente sobre las variables a, b y c

$$(a \wedge \overline{b} \wedge c) \vee (a \wedge c) \vee b$$

contiene 5 literales sin negar y uno negado ($\overline{b}$). Podemos ver que no está en FNC: no es la conjunción de cláusulas, sino una disyunción de conjunciones. Para estar en FNC debería tener una forma como

$$(a \vee \overline{b} \vee c) \wedge (a \vee c) \wedge b.$$

Esta última fórmula tiene 3 cláusulas: una de 3 literales, una de 2 y una de 1 literal. Si se evalúa la fórmula, por ejemplo, para $a = 1$, $b = 1$ y $c = 0$ el resultado es 1 y para $a = 0$, $b = 1$ y $c = 0$ el resultado es 0.

Problema de decisión 7

SAT (**satisfactibilidad**): Dada una fórmula booleana $f : V \to \{0,1\}$ en FNC, decidir si hay alguna asignación de valores de verdad a las variables V que satisface f (es decir, hace que f evalúe a 1).

Si cada cláusula de f tiene exactamente 3 literales, estamos ante un caso particular de SAT llamado el **problema de la 3-satisfactibilidad** (3SAT).

Ejemplo 13

Dada la fórmula $f = (\overline{a} \vee b) \wedge (\overline{a} \vee \overline{c})$, tenemos que SAT($f$) = SÍ. De hecho, de las 2^3 asignaciones de valores de verdad a las variables $\{a,b,c\}$, hay 5 asignaciones que satisfacen la fórmula:

a	b	c
0	0	0
0	0	1
0	1	0
0	1	1
1	1	0

En cambio, ninguna de las $2^4 = 16$ asignaciones de valores de verdad satisface la fórmula siguiente:

$$(\overline{a} \vee \overline{b} \vee c) \wedge (a \vee c) \wedge b \wedge (a \vee \overline{c}) \wedge (a \vee d) \wedge (\overline{c}).$$

Comprobémoslo: las cláusulas b y $\overline{c}$ nos obligan a que $b = 1$ y $c = 0$; las cláusulas $(a \vee c)$ y $(a \vee \overline{c})$ nos obligan a que $a = 1$; pero entonces, la primera cláusula no se puede satisfacer. Como f es una *conjunción* de cláusulas, si la primera cláusula no se puede satisfacer entonces f es insatisfactible.

Se podría pensar que el problema SAT sólo tiene sentido en el contexto de la lógica o el diseño de circuitos digitales, pero es una impresión errónea. Cualquier problema sobre un dominio acotado se puede codificar mediante variables y fórmulas booleanas. En otras palabras, muchos problemas reales se pueden expresar como instancias de SAT.

Ejemplo 14

Tres profesores (Pilar, Ricardo y María) tienen que vigilar tres exámenes (1, 2 y 3). Cada examen debe tener al menos un vigilante, aunque también puede tener más de uno. A Ricardo no le va bién el horario del primer examen, y el segundo y el tercero están solapados y se tienen que asignar a personas diferentes. Finalmente, cada profesor debe vigilar al menos un examen.

Queremos decidir si hay alguna asignación de vigilancias de examen factible. Este problema es equivalente a decidir si la fórmula siguiente es satisfactible ($p_i = 1$ indica que Pilar vigila el examen i-ésimo y similarmente para Ricardo y María).

$$(\overline{r_1}) \wedge (\overline{p_2} \vee \overline{p_3}) \wedge (\overline{r_2} \vee \overline{r_3}) \wedge (\overline{m_2} \vee \overline{m_3}) \wedge$$
$$(p_1 \vee p_2 \vee p_3) \wedge (r_1 \vee r_2 \vee r_3) \wedge (m_1 \vee m_2 \vee m_3)$$

Esta fórmula es satisfactible si, por ejemplo, $p_1 = 1$, $m_2 = 1$, $r_3 = 1$ y el resto de variables están a 0.

1.2.2. Partición de conjuntos

Problema de decisión 8

PARTICION: Dado un multiconjunto de enteros C, decidir si existe un subconjunto $C' \subseteq C$ tal que la suma de todos los elementos de C' sea igual a la suma de los elementos fuera de C'.

$$\text{PARTICION}(C) \equiv \quad \exists C' \subseteq C \mid \Big(\sum_{x \in C'} x = \sum_{y \in C \setminus C'} y \Big)$$

Multiconjunto

Un multiconjunto es una colección no ordenada de elementos donde puede haber repeticiones. Se representan entre corchetes, por ejemplo, [7,3,7,24]. Un subconjunto de un multiconjunto también puede ser un multiconjunto.

Ejemplo 15

Podemos ver que PARTICION([2,3,3,34]) = NO. Para justificarlo, tenemos que comprobar que ninguna partición del conjunto tiene la misma suma. En un multiconjunto con n elementos, hay como máximo 2^{n-1} particiones (asumiendo que la mitad de particiones son simétricas, es decir, que de una partición a la otra sólo se intercambian C' y $C \setminus C'$):

C'	Suma de C'	$C \setminus C'$	Suma de $C \setminus C'$
$\emptyset$	0	[2,3,3,34]	42
[2]	2	[3,3,34]	40
[3]	3	[2,3,34]	39
[34]	34	[2,3,3]	8
[2,3]	5	[3,34]	37
[3,3]	6	[2,34]	36

En cambio PARTICION([3,4,12,17,18,22,28]) = SÍ, ya que tenemos dos subconjuntos con suma total 52, [3,4,28,17] y [22,18,12].

El nombre PARTICION proviene de que buscamos partir el multiconjunto en dos partes con la misma suma total. En la práctica, PARTICION aparece cuando se quiere repartir equitativamente entre dos entitades un conjunto de tareas o recursos independientes entre sí. Usualmente, al tratar problemas de planificación o logística se planteará la versión de optimización del problema (encontrar el reparto más equitativo).

Ejemplo 16

Un notario tiene que dividir un conjunto de obras de arte dejadas en herencia a dos herederos. En primer lugar, necesita decidir si hay algún reparto equitativo. Este problema se puede ver como una instancia de PARTICION, donde el valor monetario de cada obra sería un valor entero dentro del multiconjunto.

Nomenclatura

En inglés, el problema PARTICION se llama *partition*.

Ejemplo 17

Un array de discos debe almacenar una lista de ficheros en dos discos y es necesario decidir dónde se guarda cada fichero. Se desea que la ocupación de cada disco sea lo más equilibrada posible. Este problema vuelve a ser una instancia de la versión de optimización de PARTICION, donde el espacio ocupado por cada fichero es un valor entero dentro del multiconjunto a particionar.

Nomenclatura

En inglés, el problema SUMA_SUB se llama *subset sum*.

Problema de decisión 9

SUMA_SUB (**suma de subconjuntos**): Dado un conjunto de enteros C y un entero t, decidir si existe un subconjunto $C' \subseteq C$ tal que la suma de todos los elementos de C' sea igual a t.

$$\text{SUMA\_SUB}(C,t) \equiv \quad \exists C' \subseteq C : \left(\sum_{x \in C'} x = t \right)$$

Ejemplo 18

Tenemos que SUMA_SUB$(\{1,9,10,17,20\},48)$ = SÍ, ya que {1, 10, 17, 20} tiene como suma total 48. En cambio, SUMA_SUB$(\{1,9,10,17,20\},35)$ = NO, aunque para comprobarlo deberíamos estudiar los $2^5 = 32$ subconjuntos del conjunto original.

En este problema, t es el objetivo a alcanzar (capacidad a ocupar, beneficio a conseguir, ...) mientras que los valores del conjunto modelan los elementos a combinar para llegar al objetivo (su coste, tamaño, ...).

Ejemplo 19

Queremos comprar un producto de una máquina expendedora que sólo acepta el importe exacto. Para hacerlo, tenemos en nuestra cartera un puñado de monedas. Si todas las monedas tienen un valor diferente, decidir si podemos comprar el producto es una instancia de SUMA_SUB, donde C es el conjunto de valores de las monedas y t es el precio del producto.

Ejercicios

3. Indicad cuáles de las siguientes fórmulas están en FNC:

$$a \wedge b,\ a \vee \overline{b},\ \overline{(a \vee b)},\ (a \vee b) \wedge b \wedge \overline{a},\ a \vee (b \wedge c)$$

4. ¿Hay alguna fórmula booleana en FNC con exactamente dos cláusulas que sea insatisfactible? Y si además esta fórmula tuviera que ser una entrada de 3SAT, ¿cambiaría la respuesta?

5. Indicad cuáles de estas fórmulas booleanas son entradas correctas de 3SAT:

$$(a \vee b \vee \overline{c}),\ (\overline{a} \vee b \vee c) \wedge (a \vee b),\ (a \vee c) \vee (b \wedge c) \vee (b \wedge \overline{c})$$

6. Indicad si las siguientes fórmulas en FNC son satisfactibles y, si lo son, calculad una asignación de valores de verdad que las satisface:

a) $x \wedge (\overline{x} \vee y) \wedge \overline{z} \wedge (\overline{x} \vee \overline{y} \vee z)$

b) $(y \vee z) \wedge (\overline{x} \vee z) \wedge (\overline{x} \vee y)$

7. Dado un multiconjunto de enteros C, demostrad que si PARTICION(C) = SÍ entonces la suma de todos los elementos de C es un número par.

8. Dadas las siguientes entradas de PARTICION, indicad cuál es su solución. En caso de que la solución sea SÍ, indicad cuál sería la partición.

a) C_1 = [1,2,3,4,5,6,7,8,9,10,11].

b) C_2 = [18,175, – 64,14,13].

Soluciones

3. Las fórmulas en FNC son: $a \wedge b$, $a \vee \overline{b}$ y $(a \vee b) \wedge b \wedge \overline{a}$. Respecto a las otras, $\overline{a \vee b}$ tiene una o-lógica dentro de una negación y $a \vee (b \wedge c)$ es una disyunción de conjunciones.

4. Sí: $(a \wedge \overline{a})$. En cambio, si la fórmula es una entrada de 3SAT, todas las fórmulas con 2 cláusulas serán siempre satisfactibles.

5. Sólo $(a \vee b \vee \overline{c})$, ya que tiene 3 literales en todas las cláusulas (en este caso, su única cláusula). $(\overline{a} \vee b \vee c) \wedge (a \vee b)$ tiene una cláusula con 2 literales mientras que $(a \vee c) \vee (b \wedge c) \vee (b \wedge \overline{c})$ ni siquiera está en FNC.

6. a) No es satisfactible.

b) Es satisfactible, por ejemplo, para: $x = 0$, $y = 1$ y $z = 1$.

7. Si PARTICION(C) = SÍ, quiere decir que C se puede dividir en dos partes con la misma suma total. Si llamamos s a la suma de los elementos de cada una de estas partes, la suma de todos los elementos de C será $2s$ que es un número par.

8. a) PARTICION(C_1) = SÍ. Una posible partición sería C' = [3,9,10,11] y $C_1 \setminus C'$ = [1,2,4,5,6,7,8]. Una estrategia para calcular la partición en esta instancia tan pequeña es calcular primero la suma total (66) y entonces buscar un conjunto de elementos que sumen en total la mitad (33).

b) PARTICION(C_2) = NO. Fijémonos que 175 – 64 es más grande que la suma del resto de enteros positivos.

2. Medidas de complejidad

2.1. Tiempo y espacio

La **complejidad computacional** de un algoritmo es una métrica abstracta de la cantidad de recursos necesarios para calcular una solución. Consideraremos dos recursos de cálculo diferentes: el tiempo de ejecución y el espacio de memoria.

Definición 10

El **tiempo de ejecución** de un algoritmo es una medida del número de pasos de cálculo o operaciones elementales necesarias para obtener el resultado.

El **espacio de memoria** consumido por un algoritmo es una medida del número de posiciones de memoria (bits) necesarios para almacenar el resultado final y los cálculos intermedios.

En función del recurso a medir, hablaremos de **complejidad temporal** o **complejidad espacial**. En ambos casos, no se pretende cuantificar un valor concreto (segundos, instrucciones, bits o megabytes) sino la tasa de crecimiento respecto al tamaño de la entrada. Usaremos la notación O para describir esta tasa de crecimiento en el caso peor, es decir, la entrada que consume más recursos.

Ved también

En el módulo, "Conceptos previos: funciones y algoritmos", podéis repasar la notación O definida.

Ejemplo 20

Un *cuadrado mágico* es una matriz $n \times n$ de números naturales que contiene los números del 1 al n^2 sin repeticiones y de forma que la suma de cada fila, columna y diagonal principal es igual al mismo valor. Un ejemplo de cuadrado mágico 3×3 es la matriz

8	3	4
1	5	9
6	7	2

Estudiemos la complejidad espacial y temporal de un algoritmo que, dada una matriz $n \times n$, compruebe si es un cuadrado mágico.

Nuestro algoritmo debe calcular el sumatorio de n filas, n columnas y 2 diagonales principales. Cada sumatorio requiere $n - 1$ sumas. Por lo tanto,

nuestro algoritmo tiene que hacer $(2n + 2)(n - 1)$ sumas. Después es necesario comparar los resultados entre sí, es decir, es necesario hacer $2n + 1$ comparaciones. En resumen, la complejidad temporal es $O(n^2)$.

Respecto a la complejidad espacial, tenemos que es necesario almacenar el resultado parcial del sumatorio a medida que lo vamos calculando ($O(1)$) y el resultado del sumatorio previo para hacer la comparación ($O(1)$). La complejidad espacial del algoritmo es pues $O(1)$.

La complejidad temporal y la complejidad espacial de un algoritmo no tienen por qué ser iguales, pero sí que deben mantener una cierta relación: la complejidad temporal siempre será igual o superior a la complejidad espacial. El motivo es que si se utilizan N posiciones de memoria, el algoritmo deberá realizar al menos N pasos para leer o escribir sus contenidos. Por lo tanto, la complejidad temporal es una cota superior de la complejidad espacial.

Para analizar y comparar la complejidad de los algoritmos, agruparemos las funciones de coste en dos grandes categorías: complejidad **polinómica** y complejidad **exponencial**. Intuitivamente, las funciones de coste polinómicas tienen una tasa de crecimiento muy inferior y por lo tanto permiten resolver un problema para entradas de tamaño grande. En cambio, las funciones de coste exponenciales rápidamente se vuelven inasumibles a medida que crece la entrada.

Definición 11

Un algoritmo tiene **complejidad polinómica (espacial o temporal)** si la tasa de crecimiento de su coste respecto al tamaño de la entrada (n) es del orden de $O(n^k)$ para alguna constante k.

Un algoritmo tiene **complejidad exponencial (espacial o temporal)** si la tasa de crecimiento de su coste respecto al tamaño de la entrada (n) es del orden de $O(2^{O(n^k)})$ para alguna constante k.

Aproximación de Stirling

Para valores grandes de n, el factorial $n!$ se puede aproximar por $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$. Por este motivo, $O(n!)$ se considera también un coste exponencial.

Ejemplo 21

Veamos a continuación algunos ejemplos de complejidades temporales polinómicas y exponenciales:

Complejidad	Nombre	Ejemplo de problema
$O(1)$	Constante	Determinar si un número es par.
$O(\log n)$	Logarítmica	Buscar un valor en un vector ordenado.
$O(n)$	Lineal	Buscar un valor en un vector desordenado.
$O(n \log n)$	Cuasilineal	Ordenar un vector (*merge-sort*).
$O(n^2)$	Cuadrática	Ordenar un vector (*bubble-sort*).
$O(n^3)$	Cúbica	Calcular el producto de dos matrices.
$O(2^n)$	Exponencial	Listar todos los subconjuntos de un conjunto de n elementos. Listar todas las asignaciones de valores de verdad a n variables booleanas.
$O(n!)$	Factorial	Listar todas las permutaciones de n elementos.

Un problema puede tener uno o más algoritmos que calculen su solución. La complejidad espacial y temporal de un problema se define a partir de estos algoritmos, concretamente la complejidad del algoritmo más eficiente que calcula la solución.

Definición 12

Una función f es **calculable en tiempo polinómico** si existe un algoritmo que calcula $f(x)$ en tiempo polinómico para cualquier entrada x.

Un problema Prob es **resoluble en tiempo polinómico** si Prob(x) es calculable en tiempo polinómico.

Similarmente, podemos dar las definiciones para la complejidad espacial (substituyendo *tiempo* por *espacio*) y para la complejidad exponencial (substituyendo *polinómico* por *exponencial*).

Definición 13

Un problema Prob es **tratable** si es resoluble en tiempo polinómico.

Un problema Prob es **intratable** si es resoluble en tiempo exponencial pero no es resoluble en tiempo polinómico.

2.1.1. El tamaño de la entrada

Antes de empezar a estudiar complejidades temporales o espaciales tenemos que hacer una reflexión sobre el tamaño de la entrada de un algoritmo. En los módulos anteriores, hemos expresado la complejidad de los algoritmos en función del tamaño de su entrada. La entrada podía ser un grafo, un conjunto, una lista, un número entero, ...

Ejemplo 22

En el análisis de la complejidad de la mayoría de los algoritmos sobre grafos (Dijkstra, Floyd, ...) la entrada es un grafo $G = (V,A)$, y analizamos la complejidad en términos del **orden** (el número n de vértices) del grafo y su **medida** (el número m de aristas).

En los algoritmos de ordenación, la entrada está formada por una lista o secuencia de números enteros y la complejidad se calcula en función de la **longitud** (número n de elementos) de la lista.

En el algoritmo de multiplicar y elevar, la entrada está formada por un número x y un exponente entero y positivo N. La complejidad se calcula en función del valor de N.

En caso que la entrada del algoritmo sea un número entero es importante fijar la base con la que representaremos el número. Un número como 101 puede estar en base $2,3,\ldots,10,\ldots$ y su representación interna en un ordenador es diferente según sea la base. Nosotros supondremos que todos los números enteros están representados en base 2, que es la forma como se representan en un ordenador.

Así, si N ($N > 0$) representa un número entero positivo cualquiera, el número de bits que necesitamos para representar N es $n = \lfloor \log_2 N \rfloor + 1$.

Definición 14

Cuando la entrada de un algoritmo dependa de un número entero N ($N > 0$), definiremos el **tamaño** de la entrada como el número de bits de la representación binaria de N.

Ejemplo 23

El número entero 101 en base 10 se representa como 1100101 en base 2. El número de bits de su representación binaria es $\lfloor \log_2 101 \rfloor + 1 = 7$. Por lo tanto, el tamaño de la entrada será $n = 7$.

Ejemplo 24

Supongamos que nos piden calcular la complejidad del algoritmo que suma los N ($N \geq 1$) primeros números naturales. Este problema ya se trató antes* y habíamos deducido que si sumábamos los N números consecutivamente entonces el algoritmo tenía una complejidad $O(N)$. Si N es pequeño, entonces este cálculo se puede considerar correcto.

¿Qué pasa si N es muy grande, por ejemplo, si N es un número de 1024 bits? Entonces es más adecuado expresar la complejidad del algoritmo en función de la longitud de la representación binaria de N y no directamente con el valor de N. Así pues, como la longitud binaria es $n = \lfloor \log_2 N \rfloor + 1$, este algoritmo pasaría a tener una complejidad $O(2^n)$, dado que hay 2^n números enteros diferentes de tamaño menor o igual que n.

Fijaos que este resultado pone en evidencia que este algoritmo es muy ineficiente. ¿Podemos pues afirmar que el problema de sumar los N primeros números naturales es un problema intratable? Esto tampoco es cierto ya que en el módulo "Conceptos previos: funciones y algoritmos", habíamos visto otro algoritmo para calcular esta suma. El algoritmo, propuesto por

Criptografía

En muchos protocolos criptográficos, la confidencialidad se garantiza a través del tamaño de una clave. Por ejemplo, con claves de 1024 bits se pueden requerir del orden de $2^{1024} \approx 10^{308}$ operaciones matemáticas para descifrar un mensaje sin tener la clave.

***Ved el subapartado 2.2 del módulo "Conceptos previos: funciones y algoritmos".**

Euler, es muy eficiente ya que tiene una complejidad polinómica (sólo realiza una suma, un producto y una división por 2). Por lo tanto, la existencia del algoritmo de Euler nos confirma que este problema es tratable.

Ejercicios

9. Tenemos un algoritmo con complejidad temporal $O(n^2)$. ¿Qué podemos decir sobre su complejidad espacial?

10. Tenemos un algoritmo con complejidad espacial $O(n)$. ¿Qué podemos decir sobre su complejidad temporal?

11. ¿Puede existir un algoritmo de complejidad temporal $O(2^n)$ y complejidad espacial $O(n)$?

12. ¿Puede existir un algoritmo de complejidad temporal $O(\log n)$ y complejidad espacial $O(n^3)$?

13. ¿Por qué en las definicions de problemas tratables e intratables no se limita la complejidad espacial?

14. Tenemos un problema resoluble en tiempo $O(n^{100})$. ¿Es tratable?

15. Tenemos un problema resoluble en tiempo $O(2^{n^2})$. ¿Es intratable?

16. Sea N ($N > 0$) un número entero y sea A un algoritmo que tiene como entrada el número entero N. Calculad, en función de la longitud de la representación binaria de N, la complejidad del algoritmo A para cada uno de los casos siguientes:

a) A tiene una complejidad $O(N)$.

b) A tiene una complejidad $O(N^2)$.

c) A tiene una complejidad $O(\log N)$.

d) A tiene una complejidad $O(N!)$.

Soluciones

9. La complejidad espacial será como mucho $O(n^2)$, pero podría ser inferior.

10. Es como mínimo $O(n)$, pero es posible que hagan falta más de $O(n)$ pasos de cálculo para llegar a la solución.

11. Sí.

12. No, porque este algoritmo no realiza suficientes pasos para poder leer o escribir toda la memoria que supuestamente consume.

13. No se impone explícitamente ninguna restricción sobre el espacio porque las restricciones ya vienen implícitas por la complejidad temporal: si un

problema es resoluble en tiempo polinómico, necesariamente es resoluble en espacio polinómico, y pasa lo mismo con tiempo exponencial.

14. Sí, ya que es resoluble en tiempo polinómico: aunque el exponente del polinomio sea una constante muy grande, el coste $O(n^{100})$ sigue siendo polinómico.

15. No lo sabemos. Sabemos que es resoluble en tiempo exponencial, pero para ser intratable, nos faltaría asegurar que no es resoluble en tiempo polinómico. Que haya una solución con complejidad exponencial, no significa que no pueda haber otra de complejidad polinómica.

16. Si N es un número entero, entonces la longitud de su representación binaria será $n = \lfloor \log_2 N \rfloor + 1$. Como hay 2^n números diferentes de tamaño menor o igual que n, obtenemos:

a) El algoritmo A tendría una complejidad $O(2^n)$.

b) El algoritmo A tendría una complejidad $O((2^n)^2) = O(2^{2n})$.

c) El algoritmo A tendría una complejidad $O(\log 2^n) = O(n \log 2) = O(n)$.

d) El algoritmo A tendría una complejidad $O(2^n!)$.

2.2. Las clases de complejidad P y EXP

Definición 15

Una **clase de complejidad** C es un conjunto de problemas con una característica común en su complejidad computacional.

Indicaremos que un problema Prob pertenece a una clase de complejidad C con $\text{Prob} \in C$.

El nombre de la clase de complejidad acostumbra a indicar si se refiere a complejidad temporal (con el sufijo TIME) o espacial (con el sufijo SPACE). Si no se hace una distinción explícita entre tiempo y espacio, entenderemos que se refiere al tiempo de cálculo.

Un primer criterio para definir clases de complejidad es la existencia de un algoritmo que resuelve un problema con un cierto consumo de recursos. Así pues, podemos definir las clases de problemas resolubles en tiempo y espacio polinómico/exponencial.

Definición 16

La **clase** PTIME es el conjunto de todos los problemas resolubles en tiempo polinómico. Para abreviar, nos referiremos a esta clase como P.

La **clase** EXPTIME es el conjunto de todos los problemas resolubles en tiempo exponencial. Para abreviar, nos referiremos a esta clase como EXP.

La **clase** PSPACE es el conjunto de todos los problemas resolubles en espacio polinómico.

La **clase** EXPSPACE es el conjunto de todos los problemas resolubles en espacio exponencial.

Ejemplo 25

Muchos de los problemas decisionales que conocemos pertenecen a las clases P y PSPACE. Algunos ejemplos de problemas en estas clases serían: decidir si un número es par, si un grafo es conexo o si un vector de enteros está ordenado en orden creciente.

Pondremos un ejemplo de problema que no pertenece a la clase P, el problema AJEDREZ: "Dada una configuración de un tablero de ajedrez, determinar si un jugador tiene una secuencia de jugadas ganadora a partir de esta configuración". Para resolver este problema es necesario explorar el árbol de posibles jugadas para los dos jugadores, que puede ser de un tamaño exponencial respecto al tamaño del tablero. A partir de este razonamiento, es posible demostrar que el problema AJEDREZ pertenece a EXP. Este problema también se puede definir para otros juegos como las damas o el go, con la misma complejidad temporal.

Explosión combinatoria

Explosión combinatoria es el término utilizado para referirse al rápido crecimiento del número de posibles soluciones respecto al tamaño de la entrada.

Ejemplo 26

Hay muchos otros problemas que se han clasificado dentro de las clases EXP o EXPSPACE, por ejemplo en el contexto de la lógica: comprobar si una propiedad expresada mediante una fórmula lógica es satisfactible puede requerir tiempo y/o espacio exponencial, en función de la expresividad del lenguaje lógico utilizado (operadores lógicos permitidos, existencia de cuantificadores, ...).

En el caso de la lógica también encontramos ejemplos de problemas que están **fuera** de las clases EXP y EXPSPACE. Por ejemplo, consideremos la lógica de primer orden, donde se permite utilizar los cuantificadores universal ($\forall$) y existencial ($\exists$). Definimos el problema LPO de la forma siguiente: "Dada una fórmula f escrita en lógica de primer orden, decidir si f es un teorema". Es decir, dada una fórmula f, nos planteamos si se puede demostrar a partir de los axiomas de la lógica de primer orden. Resulta que el problema LPO es **indecidible**: no hay ningún procedimiento que resuelva LPO en tiempo finito para cualquier entrada. Es decir, si proponemos un algoritmo para resolver este problema, necesariamente habrá algunas entradas para las que no podrá dar respuesta ni SÍ ni NO, porque requeriría un tiempo de cálculo infinito.

Entscheidungsproblem

Entscheidungsproblem (en castellano, *problema de decisión*) es el nombre alemán oficial para el problema LPO. En el año 1936, Alan Turing y Alonzo Church demostraron de forma independiente su indecidibilidad. Sus demostraciones definían un modelo abstracto de cálculo, la *máquina de Turing* y el λ*-cálculo* respectivamente.
Este resultado fue revolucionario: ¡algunos problemas no se pueden resolver en tiempo finito!

Otro problema indecidible famoso es el problema de la parada: "Dado un programa *Prog* y unos datos de entrada *x*, decidir si la ejecución de *Prog* con entrada *x* acaba en un tiempo finito". Este problema, pues, queda fuera del alcance de las clases de complejidad que estudiamos en este módulo. Los problemas indecidibles los estudia otra rama de la Informática Téorica, la *Teoría de la Calculabilidad*.

Ejemplo 27

Los problemas donde el tamaño de la solución es exponencial respecto al tamaño de la entrada nunca estarán en P ni en PSPACE. Por ejemplo, consideremos el problema PERMUTACION: "dada una cadena de caracteres *c* sin caracteres repetidos, calcular todas las posibles permutaciones de los caracteres de *c*". La solución de este problema tendrá un tamaño exponencial respecto a la entrada, concretamente $n!$, siendo n el tamaño de la entrada. Por lo tanto, este problema pertenece a EXP y a EXPSPACE pero no a P ni a PSPACE.

2.3. La clase de complejidad NP

Hasta ahora nos hemos planteado la complejidad temporal que representa resolver un problema. Un aspecto muy relacionado con éste es el tiempo necesario para comprobar que la solución a un problema es correcta. Comprobar una solución es una tarea más sencilla que calcularla y, por lo tanto, puede requerir menos recursos computacionales. Nos centraremos en los problemas decisionales y descubriremos algunos problemas donde se puede comprobar la solución de forma eficiente.

Definición 17

Sea $\mathcal{T}$ un conjunto, los elementos del cual se llaman **testigos**. Dado un problema decisional $\mathsf{Prob} : \mathcal{I} \to \{\text{SÍ,NO}\}$, una **función verificadora** es una función $v : \mathcal{I} \times \mathcal{T} \to \{\text{SÍ,NO}\}$ tal que:

- Si $\mathsf{Prob}(x) = \text{SÍ}$, entonces existe algún testigo $t \in \mathcal{T}$ para el que $v(x,t) = \text{SÍ}$.
- Si $\mathsf{Prob}(x) = \text{NO}$, entonces $v(x,t) = \text{NO}$ para cualquier $t \in \mathcal{T}$.

Definición 18

Un problema Prob es **verificable en tiempo polinómico** si tiene una función verificadora v calculable en tiempo polinómico.

De forma más coloquial, un testigo es un fragmento de información que, juntamente con la entrada, nos permite validar eficientemente la solución SÍ a un problema decisional. Fijémonos en que no nos planteamos el coste que puede tener calcular el testigo: sólo nos interesa el coste de validar la solución a partir del testigo.

Observación

El concepto de verificación es asimétrico: no intentamos certificar las instancias con solución NO, sólo las de solución SÍ.

Ejemplo 28

Consideremos el problema DUPL definido de la forma siguiente: "Dado un vector de enteros A, decidir si contiene algún número repetido". Definiremos una función verificadora para este problema decisional.

Cada entrada de este problema es una secuencia finita de enteros ($\mathcal{I} = \mathbb{Z}^*$). Para definir la función verificadora, necesitamos elegir un conjunto de testigos $\mathcal{T}$ apropiado. El testigo nos tendría que permitir comprobar que DUPL(A) = SÍ sin generar falsos positivos cuando DUPL(A) = NO.

Intuitivamente, para demostrar que hay algún valor repetido en un vector es suficiente indicar dos posiciones del vector con el mismo contenido. Por lo tanto, $\mathcal{T}$ sería el conjunto de pares de números naturales ($\mathcal{T} = \mathbb{N} \times \mathbb{N}$) y cada testigo sería un par de posiciones (i,j) del vector. La función de verificación $v(A,(i,j))$ se definiría de la forma siguiente:

$$v(A,(i,j)) = \begin{cases} \text{SÍ} & \text{Si } i \neq j \ \wedge \ A[i] = A[j] \\ \text{NO} & \text{En otro caso.} \end{cases}$$

Fijémonos que esta propuesta de función verificadora satisface las propiedades requeridas:

- Si DUPL(A) = SÍ (hay algún valor duplicado), existe un testigo (i,j) tal que $A[i] = A[j]$. Por ejemplo, para A = (3,4,17,4) el testigo (2,4) nos permitiría verificar que $A[2] = A[4]$.
- Si DUPL(A) = NO (no hay valores duplicados), no podremos encontrar ningún par (i,j) tal que $A[i] = A[j]$. Fijémonos que la definición de v controla el caso trivial en que $i = j$ para que no se produzcan falsos positivos. Sin este control, para (12,8,9) tendríamos que el testigo (1,1) verifica erróneamente la entrada.

Ejemplo 29

Para el problema SAT (problema de decisión 7), cada entrada es una fórmula booleana en FNC. Un testigo sería una asignación de valores de verdad a las variables de la fórmula, concretamente la asignación que satisface la fórmula.

A partir de una fórmula de entrada satisfactible y este testigo, se puede verificar que la fórmula es satisfactible en tiempo polinómico: sólo es necesario comprobar que cada cláusula evalúa a 1 según la asignación de valores de verdad del testigo.

Fijaos también que si la fórmula de entrada no es satisfactible, ninguna asignación de valores de verdad a las variables conseguirá satisfacer todas las cláusulas. Por lo tanto, no hay ningún testigo que permita verificar erróneamente una entrada con solución NO.

Ejemplo 30

Para el problema PARTICION (problema de decisión 8), cada entrada es un multiconjunto de enteros C. Un testigo sería un subconjunto $C' \subseteq C$, concretamente el que cumple $\sum_{x \in C'} x = \sum_{x \in C \setminus C'} x$.

A partir de este subconjunto C', se puede verificar que PARTICION(C) = SÍ comprobando que la suma de los elementos de C' es igual a la suma de los elementos que no están en C'. Esta comprobación se puede hacer en tiempo lineal. También sería necesario validar que $C' \subseteq C$, es decir, que todos los elementos de C' estaban en el conjunto C original. Esta comprobación también se puede hacer en tiempo polinómico.

Además, si PARTICION(C) = NO tenemos que ningún subconjunto C' de C cumplirá $\sum_{x \in C'} x = \sum_{x \in C \setminus C'} x$. Por lo tanto, no hay ningún testigo que permita verificar erróneamente una entrada con solución NO.

Definición 19

La **clase** NPTIME es el conjunto de todos los problemas verificables en tiempo polinómico. Para abreviar, nos referiremos a esta clase como NP.

Nomenclatura

La N de NP significa "**N**o determinista" y proviene del nombre de un modelo abstracto de cálculo: la máquina de Turing no determinista. NP también se puede definir como el conjunto de problemas resolubles en tiempo polinómico sobre una máquina de Turing no determinista.

Ejemplo 31

Los problemas SAT y PARTICION pertenecen a NP, ya que como hemos visto previamente (ejemplos 29 y 30) se pueden verificar en tiempo polinómico. Iremos viendo más ejemplos de problemas NP a lo largo de este módulo.

Ejercicios

17. Demostrad que los siguientes problemas se pueden verificar en tiempo polinómico:

a) Dado un número entero positivo x, decidir si es un número compuesto (tiene dos divisores diferentes de 1 y él mismo).

b) Dado un grafo $G = (V,A)$ y un vértice v del grafo ($v \in V$), decidir si v forma parte de algún ciclo dentro del grafo.

c) Dado un grafo $G = (V,A)$, decidir si el grafo tiene al menos tres componentes conexas.

18. Si un problema decisional pertenece a P, entonces seguro que pertenece a NP. ¿Cierto o falso?

19. Consideremos el problema decisional "Dado un grafo ponderado (G,ω), decidir si contiene algún camino de longitud 100 o más". Un posible testigo es un par de vértices (v_1,v_2), correspondientes a la arista de más peso del grafo G. ¿Cierto o falso?

Ved también

En el subapartado 4.2 del módulo "Recorridos y connectividad", hemos visto que el problema de encontrar **el camino más corto** en un grafo es un problema tratable, resuelto por los algoritmos de Dijkstra y Floyd. Pero, sorprendentemente, el problema de encontrar **el camino más largo** en un grafo ponderado es un problema intratable.

Soluciones

17. a) Un posible testigo sería un par de números enteros positivos y y z. Para comprobar el testigo, sería necesario ver que $x = y \cdot z$ y que $y \neq 1$, $y \neq x$. Si el número de entrada x se puede representar en n bits, el producto de y y z se puede realizar en tiempo $O(n^2)$ y la comparación con 1 y x se puede hacer en tiempo $O(n)$.

b) Un posible testigo sería la lista de vértices de un circuito dentro de G que contiene el vértice v. Para comprobar el testigo, haría falta comprobar que v pertenece a la lista de vértices, que los vértices de la lista son adyacentes entre sí, que no hay vértices repetidos en la lista y que el último vértice de la lista es adyacente al primero. Dado que la lista puede tener, como mucho, todos los vértices del grafo (n), la comprobación se puede hacer en tiempo $O(n \log n)$ si el grafo se representa mediante una matriz de adyacencia.

c) Un posible testigo sería una terna de tres vértices (v_1,v_2,v_3), uno de cada componente conexa. Para comprobar el testigo, sería necesario comprobar que no hay ningún camino entre v_1 y v_2, entre v_1 y v_3 ni entre v_2 y v_3. Podemos descubrir que estos vértices no estan conectados asignando un peso unitario a cada arista del grafo G y aplicando el algoritmo de Floyd: el camino más corto entre los tres vértices debería tener longitud infinita, por lo tanto, la verificación del testigo se puede hacer en tiempo $O(n^3)$.

18. Cierto. El mismo algoritmo que resuelve el problema permite verificar el resultado SÍ a partir de la entrada en tiempo polinómico.

19. Falso. Este testigo sólo permite certificar la respuesta SÍ en los casos donde la arista de más peso del grafo tiene un peso más grande o igual que 100. Si el peso más grande de las aristas del grafo es menor que 100, puede haber un camino que pase por diversas aristas hasta superar el peso límite de 100.

2.4. Jerarquía de complejidad y la cuestión P $\stackrel{?}{=}$ NP

Un ámbito importante dentro de la Teoría de la Complejidad es el estudio de la relación entre diferentes clases de complejidad. Por ejemplo, se puede dar el caso que todos los problemas de una clase de complejidad pertenezcan también a otra, de forma que la primera clase esté incluida en la segunda.

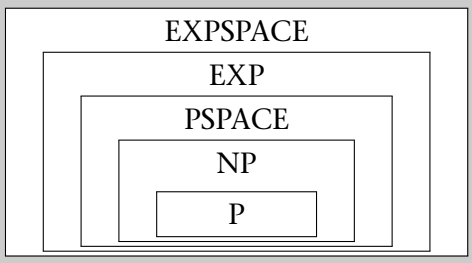
Propiedad 20

Las clases de complejidad satisfacen las siguientes relaciones de inclusión:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP} \subseteq \mathrm{EXPSPACE}$$

$$\mathrm{P} \subset \mathrm{EXP}$$

$$\mathrm{PSPACE} \subset \mathrm{EXPSPACE}$$



Demostración: Demostraremos algunas de estas inclusiones. En primer lugar, está claro que $P \subseteq EXP$ y $PSPACE \subseteq EXPSPACE$ por la propia definición de las clases: si un problema es resoluble en complejidad polinómica o inferior, también será posible resolverlo en complejidad exponencial o inferior. También hemos visto en el apartado anterior que $P \subseteq NP$.

Hemos mencionado previamente que la complejidad temporal es una cota superior de la complejidad espacial de un algoritmo. Por lo tanto, si un problema $\mathsf{Prob} \in P$ entonces $\mathsf{Prob} \in PSPACE$, de forma que hemos demostrado $P \subseteq PSPACE$. Similarmente, podemos demostrar $EXP \subseteq EXPSPACE$.

Por otro lado, la inclusión $P \subset EXP$ indica que hay problemas resolubles en tiempo exponencial y que no son resolubles en tiempo polinómico. Son los problemas que hemos llamado intratables. Un ejemplo sería el problema AJEDREZ que hemos descrito en el ejemplo 25.

No demostraremos las inclusiones $PSPACE \subset EXPSPACE$ y $PSPACE \subseteq EXP$. Sólo resaltaremos esta última propiedad, que nos indica una relación nada evidente: cualquier problema resoluble en espacio polinómico es resoluble en tiempo exponencial. ∎

Ejemplo 32

El problema PAR descrito en el ejemplo 3 pertenece a la clase P. Por lo tanto, sin necesidad de más información, ya sabemos que PAR también pertenece a las clases NP, PSPACE, EXP y EXPSPACE.

Ejemplo 33

En el ejemplo 25 hemos descrito un problema AJEDREZ que pertenece a EXP pero no a P. Automáticamente, esta información nos permite afirmar que $\text{AJEDREZ} \in EXPSPACE$. Sólo con estos datos, no podemos afirmar si AJEDREZ pertenece a NP o PSPACE.

En concreto, estudiaremos especialmente las relaciones entre las clases P y NP. Este especial interés está motivado por diversas razones:

Lectura complementaria

Revisad también el artículo de S. Cook (2003). *The Importance of the* P *versus* NP *Question.* Journal of the ACM (vol. 1, núm. 50, págs. 27-29)

- Primero, la clase NP contiene muchos problemas relevantes por su aplicación en diferentes ámbitos (logística, planificación, diseño asistido por computador, bioinformática, criptografía, ...).
- En segundo lugar, aún no se ha conseguido demostrar la relación entre la clase P y la clase NP. Es decir, hay problemas que se pueden *verificar* en tiempo polinómico pero no sabemos si se pueden *resolver* o no en tiempo polinómico.
- Finalmente, NP representa una frontera de la tratabilidad. Los problemas que no pertenecen a NP son *claramente intratables*: si hay un problema que ni siquiera se puede verificar eficientemente, nos podemos olvidar de resolverlo eficientemente. Por otro lado, los problemas de la clase NP son *potencialmente tratables*: como se pueden verificar eficientemente, existe la posibilidad de que también se puedan resolver eficientemente, aunque para algunos problemas nuestro conocimiento aún no permita hacerlo.

Ejemplo 34

En el ejemplo 31 hemos visto que los problemas SAT y PARTICION pertenecen a la clase NP ya que se pueden verificar eficientemente. Ahora bien, no se conoce ningún algoritmo eficiente que los resuelva: los mejores algoritmos conocidos tienen un tiempo de ejecución exponencial. Por otro lado, la teoría no impide que haya algoritmos más eficientes para SAT y PARTICION que los actuales. ¡Quizás existen y aún no se han encontrado!

Problema del milenio

El año 2000, el Clay Mathematics Institute denominó a la cuestión P = NP? uno de los "problemas del milenio" y ofrece 1 millón de dólares a quién consiga resolverlo.

Precisamente la relación entre las clases P y NP es uno de los problemas abiertos más importantes del campo de la Informática Teórica. Como ya sabemos que $P \subseteq NP$, la duda está en si $NP \subseteq P$. Dicho de otra forma, nos preguntamos si podemos **resolver** en tiempo polinómico todos los problemas que se pueden **verificar** en tiempo polinómico, es decir, si $P \stackrel{?}{=} NP$. Algunas formulaciones equivalentes de esta pregunta serían:

- ¿Son tratables todos los problemas de NP?
- ¿Hay algún problema que esté en NP pero no en P ($NP \setminus P \neq \emptyset$)?

Aunque a lo largo de los años se han hecho numerosas propuestas de demostración para esta cuestión, tanto afirmando que P = NP como que $P \neq NP$, actualmente no hay ninguna demostración aceptada sobre la relación entre P y NP. Las evidencias existentes parecen apuntar hacia $P \neq NP$, ya que una confirmación de que $P \neq NP$ no tendría consecuencias muy importantes. En cambio, si se demuestra que P = NP, este hecho tendría enormes repercusiones sobre la Informática: muchos problemas que hasta ahora se han considerado difíciles o que no se pueden resolver de forma práctica pasarían a poderse resolver de forma eficiente.

Por poner sólo un ejemplo, la criptografía de clave pública basa su seguridad en la existencia de problemas computacionalmente complejos. La existencia de algoritmos eficientes para resolver estos problemas reduciría enormemente el coste de descifrar cualquier mensaje. Esto podría comprometer la seguridad de algunos criptosistemas de clave pública utilizados en la actualidad.

Ejercicios

20. Para cada uno de los siguientes problemas, indicad a qué clases de complejidad debe pertenecer necesariamente, a cuáles puede pertenecer y a cuáles seguro que no pertenece:

a) Un problema que se puede resolver en espacio $O(n^3)$.

b) Un problema que se puede verificar en tiempo $O(n^5 \log n)$.

c) Un problema que se puede verificar en tiempo $O(3^n)$.

d) Un problema que no se puede resolver en tiempo polinómico.

21. Si P = NP, entonces P = PSPACE. ¿Cierto o falso?

22. Si P = PSPACE, entonces P = NP. ¿Cierto o falso?

23. Podemos afirmar que P $\neq$ EXPSPACE. ¿Cierto o falso?

24. Tenemos un problema de NP que se puede resolver en tiempo polinómico. ¿Quiere decir esto que P = NP?

Soluciones

20. Fijémonos que el hecho de que haya una determinada solución no impide que haya otras más o menos eficientes, a menos que el enunciado lo indique explícitamente. Otro aspecto relevante a destacar es que la verificación en tiempo exponencial no nos aporta información sobre la clase de complejidad de un problema.

Problema	P	NP	PSPACE	EXP	EXPSPACE
a	Puede	Puede	Sí	Sí	Sí
b	Puede	Sí	Sí	Sí	Sí
c	Puede	Puede	Puede	Puede	Puede
d	No	Puede sólo si P $\neq$ NP	Puede	Puede	Puede

21. Falso. P y NP son dos subclases de PSPACE y el hecho que sean iguales entre sí no nos indica nada respecto a la superclase PSPACE.

22. Cierto. La inclusión NP $\subseteq$ PSPACE hace que si P = PSPACE entonces P = NP.

23. Cierto. Sabemos que P $\subset$ EXP, y como EXP $\subseteq$ EXPSPACE podemos concluir que P $\neq$ EXPSPACE.

24. Esto no quiere decir que P = NP. Dado que P $\subseteq$ NP, es normal encontrar dentro de NP problemas que se pueden resolver en tiempo polinómico, es decir, problemas de P. Para demostrar que P = NP sería necesario demostrar que **todos** los problemas de NP se pueden resolver en tiempo polinómico.

3. Reducciones y completitud

En este apartado estudiaremos la **complejidad relativa** de dos problemas diferentes, entendida como la comparación entre los recursos necesarios para resolver los dos problemas. Dados dos problemas A y B, nos plantearemos cuestiones como "*¿es* A *más complejo que* B*?*" o "*¿es* A *igual de complejo que* B*?*". Estas preguntas nos permitirán clasificar la complejidad de un problema nuevo a partir de problemas conocidos, así como identificar los problemas más difíciles dentro de una clase de complejidad.

3.1. El concepto de reducción polinómica

Hay problemas decisionales que están estrechamente relacionados. En algunos de estos casos, el mismo algoritmo que resuelve un problema se puede utilizar para resolver otros problemas similares, realizando sólo un preproceso a la entrada.

Ejemplo 35

Si tenemos un algoritmo para decidir si un número entero es positivo, lo podemos reaprovechar para decidir si un número entero es negativo: sólo necesitamos multiplicar la entrada por –1 antes de ejecutar el algoritmo.

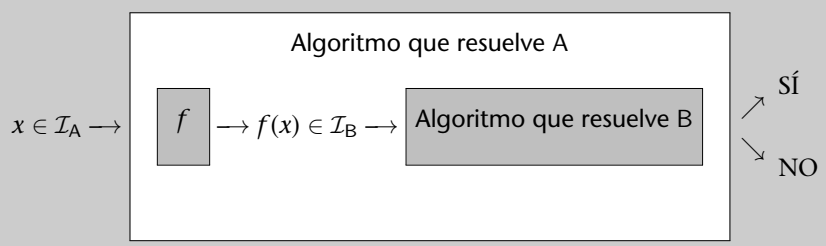
Atención!

El orden de los problemas A y B en una reducción es relevante: $\mathsf{A} \leq_p \mathsf{B}$ no es lo mismo que $\mathsf{B} \leq_p \mathsf{A}$.

Definición 21

Una **reducción** del problema A al problema B es una función $f : \mathcal{I}_\mathsf{A} \to \mathcal{I}_\mathsf{B}$ que transforma cada entrada del problema A ($x \in \mathcal{I}_\mathsf{A}$) en una entrada del problema B ($f(x) \in \mathcal{I}_\mathsf{B}$) de forma que se cumple $\mathsf{A}(x) = \mathsf{B}(f(x))$.

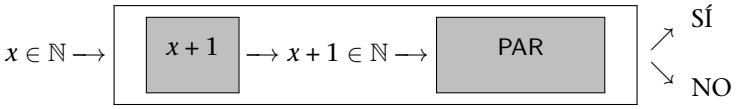
Si además hay un algoritmo que calcula f en tiempo polinómico, diremos que f es una **reducción polinómica** del problema A al problema B y lo denotaremos como $\mathsf{A} \leq_p \mathsf{B}$.



Intuitivamente, la reducción es un preproceso que nos permite reaprovechar el algoritmo que resuelve el problema B para resolver el problema A sin ninguna otra modificación al algoritmo. Veamos algunos ejemplos de funciones que son reducciones y de funciones que no lo son.

Ejemplo 36

Disponemos de un algoritmo muy eficiente para resolver el problema PAR y queremos reaprovecharlo para resolver el problema IMPAR: "Dado un número natural x, decidir si es impar". Un posible algoritmo que calcularía la solución de IMPAR podría ser el siguiente: calcular la solución de PAR con entrada $x+1$. Gráficamente, representaríamos esta reducción así:



Fijémonos que este algoritmo calculará la solución correcta para todas las entradas del problema IMPAR, es decir, que IMPAR(x) = PAR($x+1$) para cualquier x. De cara a demostrar esta igualdad, es necesario considerar dos tipos de entradas: las que tienen solución SÍ y las que tienen solución NO.

- Caso IMPAR(x) = SÍ: Hay que demostrar que PAR($x+1$) = SÍ.

  Cierto. Como x es un número impar, entonces $x+1$ será par (PAR($x+1$) = SÍ). Tenemos pues que IMPAR(x) = PAR($x+1$).

- Caso IMPAR(x) = NO: Hay que demostrar que PAR($x+1$) = NO.

  Cierto. Como x es un número par, entonces $x+1$ será impar (PAR($x+1$) = NO). Volvemos a tener que IMPAR(x) = PAR($x+1$).

En este caso, la función de reducción utilizada ha sido $f(x) = x+1$. Es una función que se puede calcular en tiempo polinómico y, por lo tanto, podemos decir que IMPAR $\leq_p$ PAR.

Ejemplo 37

Supongamos que queremos hacer una reducción de PRIMO ("dado un número natural x, decidir si tiene exactamente dos divisores: 1 y x") al problema IMPAR. Proponemos como función de reducción $f(x) = x+2$. Comprobemos si es una reducción correcta:

- Caso PRIMO(x) = SÍ: Hay que demostrar que IMPAR($x+2$) = SÍ.

  Falso. Por ejemplo $x = 2$ es un número primo (PRIMO(2) = SÍ) pero $x+2 = 4$ no es impar (IMPAR(4) = NO).

- Caso PRIMO(x) = NO: Hay que demostrar que IMPAR($x+2$) = NO.

  Falso. Por ejemplo $x = 1$ no es un número primo (PRIMO(1) = NO) pero $x+2 = 3$ sí es impar (IMPAR(3) = SÍ).

Vemos pues que f no es una reducción. Intentemos definir una función más complicada para hacer la reducción de PRIMO a IMPAR que resuelva estos problemas detectados:

$$g(x) = \begin{cases} 2 & \text{Si } x = 1 \\ 3 & \text{Si } x = 2 \\ x & \text{En otro caso.} \end{cases}$$

Comprobemos ahora si g es una reducción correcta:

- Caso PRIMO(x) = SÍ: Hay que demostrar que IMPAR($g(x)$) = SÍ.

  Cierto. Si un número es primo, entonces tiene que ser impar, con la excepción del 2, pero la función g ya lo trata como un caso especial: PRIMO(2) = IMPAR(3) = SÍ.

- Caso PRIMO(x) = NO: Hay que demostrar que IMPAR($g(x)$) = NO.

  Falso. La igualdad se cumple para $x = 4$ pero hay números compuestos que son impares, como por ejemplo $15 = 3 \cdot 5$: PRIMO(15) = NO, IMPAR(15) = SÍ.

La función g tampoco nos sirve como reducción, ya que no preserva la solución para las entradas con solución NO (aunque sí se comporta correctamente para entradas con solución SÍ). Fijémonos pues que, al definir una reducción, siempre tenemos que comprobar tanto la solución SÍ como la solución NO por separado.

Ejercicios

25. Sea MULT4 el problema "Dado un número natural x, decidir si es un múltiplo de 4." Proponed reducciones polinómicas para IMPAR $\leq_p$ MULT4, PAR $\leq_p$ MULT4, MULT4 $\leq_p$ PAR y MULT4 $\leq_p$ IMPAR y demostrad que son correctas.

26. Sea Prob un problema cualquiera. ¿Podemos definir una reducción Prob $\leq_p$ Prob? ¿Qué función de reducción tendríamos que utilizar?

27. Ahora recuperemos los problemas SUMA_SUB (problema de decisión 9) y PARTICION (problema de decisión 8). Proponemos la siguiente reducción SUMA_SUB $\leq_p$ PARTICION: $f(C,t) = C \cup \{s - 2t\}$, donde s es la suma de todos los elementos de C. ¿Es f una reducción polinómica? Justificad la respuesta.

28. Queremos hacer la reducción inversa, PARTICION $\leq_p$ SUMA_SUB, y proponemos como función de reducción: $f(C) = (C, s \text{ div } 2)$, donde s es la suma de todos los elementos de C y div es la división entera. ¿Es f una reducción polinómica? Justificad la respuesta.

Soluciones

25. Proponemos las reducciones siguientes:

- PAR $\leq_p$ MULT4: $f(x) = 2x$.
  Si x es par, entonces $2x$ será múltiplo de 4. En cambio, si x es impar, $2x$ será par pero no será múltiplo de 4.
- IMPAR $\leq_p$ MULT4: $g(x) = 2(x + 1)$.
  Si x es impar, $x + 1$ será par y por lo tanto $2(x + 1)$ será múltiplo de 4. Igual que antes, si x no es impar, $x + 1$ será impar y $2(x + 1)$ será par pero no múltiplo de 4.
- MULT4 $\leq_p$ PAR: $h(x) = (x \text{ div } 2)^{(1-(x \bmod 2))}$, donde div es la división entera y mod es el resto de la división.
  Si x es múltiplo de 4, el cociente será un número par y el resto será 0, y por lo tanto $h(x)$ será par. En cambio, si x no es múltiplo de 4 puede pasar que el resto sea 1 (por ejemplo, $x = 13$) o que el cociente sea impar (por ejemplo, para $x = 30$). Si el resto es 1, independientemente de como sea el cociente, tendremos $h(x) = (x \text{ div } 2)^0 = 1$ y por lo tanto $h(x)$ no es un número par. Si el resto es 0 y el cociente es impar, tenemos que

$h(x) = (x \text{ div } 2)^1 = x \text{ div } 2$ que como hemos dicho es un número impar. Por lo tanto, si x no es múltiplo de 4, tenemos que $h(x)$ no es par.

- MULT4 $\leq_p$ IMPAR: $m(x) = h(x) + 1$.
  La demostración es equivalente a la que hemos hecho en la reducción anterior.

Todas las funciones propuestas son calculables en tiempo polinómico.

26. La reducción Prob $\leq_p$ Prob siempre se puede definir, independientemente de quién sea el problema Prob. La función de reducción que utilizaríamos sería la función identidad $f(x) = x$, que tiene coste constante y garantiza que $A(x) = A(f(x))$.

27. En primer lugar, podemos ver que f es calculable en tiempo polinómico: se calcularía haciendo un recorrido sobre el conjunto C, haciendo una suma para cada elemento y, al final, una resta y un producto.

Ahora nos falta demostrar que SUMA_SUB(C,t) = PARTICION($f(C,t)$) para aquellas entradas con soluciones SÍ y aquellas con solución NO:

- Caso SUMA_SUB(C,t) = SÍ: Hay que demostrar que PARTICION($f(C,t)$) = SÍ.

  SUMA_SUB(C,t) = SÍ significa que tenemos un subconjunto $X \subseteq C$ tal que la suma de sus elementos es t. Como s está definido como la suma de todos los elementos de C, el complementario de X, $\overline{X}$, tendrá como sumatorio de sus elementos $s - t$. Entonces, $C \cup \{s - 2t\}$ se puede particionar de la forma siguiente: $\overline{X}$ y $X \cup \{s - 2t\}$. Fijémonos que las dos particiones tienen suma total $s - t$, de manera que PARTICION($f(C,t)$) = SÍ.
- Caso SUMA_SUB(C,t) = NO: Hay que demostrar que PARTICION($f(C,t)$) = NO.

  Suponemos que PARTICION($f(C,t)$) = SÍ y SUMA_SUB(C,t) = NO y llegaremos a una contradicción. La suma total de los elementos de $f(C,t)$ es $2s-2t$ y, por lo tanto, si hay una partición de $C \cup \{s-2t\}$, la suma de los elementos de cada partición debe ser $s - t$. Si nos fijamos en la partición que contiene el número $s - 2t$, el resto de elementos de la partición deben sumar t. Pero esto significa que hay un subconjunto de C que tiene como suma total t, contradiciendo la hipótesis que SUMA_SUB(C,t) = NO.

28. La reducción propuesta va bien encaminada pero tiene dos errores. En primer lugar, f no preserva la solución cuando la suma s tiene un valor impar. Por ejemplo, si C = [3,4,8], tenemos que PARTICION(C) = NO, pero en cambio SUMA_SUB({3,4,8},7) = SÍ.

Finalmente, la reducción tiene otro error más sutil: el problema PARTICION tiene como entrada un **multiconjunto** de enteros (donde se admiten repeticiones) mientras que SUMA_SUB espera recibir como entrada un **conjunto** de enteros (y por lo tanto, no admite repeticiones). Por lo tanto, la función $f(C) = (C,s \text{ div } 2)$ no siempre nos retorna una entrada correcta de SUMA_SUB para cualquier entrada de PARTICION.

3.2. Propiedades de las reducciones, transitividad y equivalencia

El concepto de reducción es muy importante por dos motivos, uno más práctico y otro más teórico.

En primer lugar, las reducciones nos permiten *reaprovechar* algoritmos conocidos para la resolución de nuevos problemas. Esta reutilización es especialmente ventajosa cuando el algoritmo es muy eficiente o cuando su implementación es muy costosa. Por ejemplo, en la actualidad, gran cantidad de problemas complejos se resuelven traduciéndolos a SAT. Los *solvers* de SAT están muy optimizados, de forma que es más práctico traducir un problema a SAT que desarrollar soluciones a medida para cada nuevo problema.

¿El mejor *solver*?

Desde el año 2002, se realiza una competición científica anual donde se compara la eficiencia de los *solvers* de SAT más recientes: http://www.satcompetition.org.

En segundo lugar, una reducción establece una relación entre las complejidades de los problemas involucrados. Con $A \leq_p B$, estamos diciendo que una forma de resolver A es calcular la función de reducción f y después resolver B.

- En una dirección, esto fija una *cota superior* para la complejidad del problema A: como máximo, A se puede resolver en el tiempo que se tarda en calcular la función de reducción más el tiempo de resolver B. No estamos diciendo que esta sea la forma más eficiente de resolver A, pero como mínimo sabemos que tenemos esta forma de resolverlo.
- En la direcció opuesta, $A \leq_p B$ también nos fija una *cota inferior* a la complejidad de B: ¡calcular $B(f(x))$ no puede ser más eficiente que calcular $A(x)$, como mínimo será igual de eficiente! Por lo tanto, si sabemos que A es un problema muy complejo, no es posible que B se pueda resolver de forma muy eficiente.

Utilizando estas propiedades de las reducciones es posible identificar la clase de complejidad de un problema comparándolo con problemas conocidos. A continuación, enunciamos formalmente estas dependencias que aparecen al hacer una reducción.

Observación

Notad que la relación $A \leq_p B$ denota que "el problema B es al menos tan difícil como el problema A" o, enunciado de otra forma, "el problema A no es más difícil que el problema B".

Propiedad 22

Si $A \leq_p B$, entonces:

1) $B \in P \rightarrow A \in P$ (Si B pertenece a P, entonces A también)

2) $B \in NP \rightarrow A \in NP$ (Si B pertenece a NP, entonces A también)

3) $A \notin P \rightarrow B \notin P$ (Si A no pertenece a P, entonces B tampoco)

4) $A \notin NP \rightarrow B \notin NP$ (Si A no pertenece a NP, entonces B tampoco)

Demostración: Demostremos que si $A \leq_p B$ y $A \notin P$ entonces $B \notin P$ (el resto de propiedades se demostrarían de forma similar). Procedemos por reducción al absurdo: supondremos que $A \notin P$ y $B \in P$ y llegaremos a una contradicción.

Como $B \in P$, sabemos que $B(x)$ es calculable en tiempo polinómico. Además, como $A \leq_p B$, sabemos que hay una función f calculable en tiempo polinómico que permite expresar A en términos de B: $A(x) = B(f(x))$. Pero como B y f se pueden calcular en tiempo polinómico, tenemos que $B(f(x))$ también se puede calcular en tiempo polinómico. Esto querría decir que $A \in P$, contradiciendo la hipótesis inicial que $A \notin P$. ∎

Ejemplo 38

Supongamos que tenemos una reducción SUMA_SUB $\leq_p$ PARTICION como la del ejercicio 27. Como sabemos que PARTICION $\in$ NP, automáticamente SUMA_SUB $\in$ NP. Si descubriéramos que SUMA_SUB $\notin$ P, entonces sabríamos que PARTICION $\notin$ P.

Las conclusiones sólo se aplican en una dirección: si averiguamos que PARTICION $\notin$ P, esta reducción no nos aporta ninguna información sobre SUMA_SUB. Igualmente, si averiguamos que SUMA_SUB $\in$ NP, esta reducción no nos informa sobre la complejidad de PARTICION.

Propiedad 23

La reducción polinómica es **transitiva**: dados tres problemas A, B y C, si $A \leq_p B$ y $B \leq_p C$, entonces tenemos que $A \leq_p C$.

Demostración: Para demostrar que existe la reducción $A \leq_p C$, tenemos que construir una función f calculable en tiempo polinómico tal que $A(x) = C(f(x))$.

La reducción $A \leq_p B$ nos asegura que existe una función g calculable en tiempo polinómico tal que $A(x) = B(g(x))$. Igualmente, con la reducción $B \leq_p C$, sabemos que hay una función h calculable en tiempo polinómico tal que $B(x) = C(h(x))$. La función f que estamos buscando es precisamente la composición de estas funciones g y h: $A(x) = B(g(x)) = C(h(g(x)))$.

Además, como $f(x) = h(g(x))$ y g y h tienen complejidad polinómica, f también tiene complejidad polinómica. ∎

Definición 24

Dos problemas decisionales A y B son **polinómicamente equivalentes** cuando $A \leq_p B$ y $B \leq_p A$.

Ejemplo 39

Los problemas IMPAR y PAR son polinómicamente equivalentes entre sí. Curiosamente, la misma función $f(x) = x+1$ que hace la reducción IMPAR $\leq_p$ PAR permite hacer la reducción PAR $\leq_p$ IMPAR. Normalmente no tendremos esta suerte y necesitaremos definir una función de reducción diferente para cada dirección de la reducción.

Ejemplo 40

Todos los problemas de la clase P (con dos excepciones que mencionaremos a continuación) son polinómicamente equivalentes entre sí. Una función de reducción genérica $\mathsf{A} \leq_p \mathsf{B}$ entre cualquier par de problemas A y B de la clase P se construiría de la forma siguiente.

Primero, elegimos dos entradas del problema B, una con solución SÍ ($y_s \in \mathcal{I}_\mathsf{B}$,B($y_s$) = SÍ) y la otra con solución NO ($y_n \in \mathcal{I}_\mathsf{B}$,B($y_n$) = NO). Qué entradas elijamos es indiferente, sólo nos preocupa cuál es su solución. Entonces, la función de reducción $f(x)$ se definiría como:

$$f(x) = \begin{cases} y_s & \text{Cuando A}(x) = \text{SÍ} \\ y_n & \text{Cuando A}(x) = \text{NO} \end{cases}$$

Es decir, en primer lugar resolvemos el problema A y devolvemos una entrada precalculada del problema B que sabemos que tiene la misma solución. Esta función f es calculable en tiempo polinómico porque $\mathsf{A} \in$ P.

Para poder definir esta función f necesitamos que el problema B tenga alguna entrada con solución SÍ y alguna entrada con solución NO. Precisamente, los dos problemas de P donde no se puede establecer esta reducción son dos problemas definidos en el ejemplo 6: ProbSI, el problema con solución SÍ para todas las entradas y ProbNO, el problema con solución NO para todas las entradas.

Ejercicios

29. Recordad el problema AJEDREZ definido en el ejemplo 25. ¿Es posible definir una reducción AJEDREZ $\leq_p$ PAR? ¿Y PAR $\leq_p$ AJEDREZ?

30. En el ejercicio 27 hemos visto cómo hay que hacer la reducción polinómica SUMA_SUB $\leq_p$ PARTICION. ¿Qué información nos aporta sobre la complejidad de estos problemas?

31. Todo problema es polinómicamente equivalente a sí mismo. ¿Cierto o falso?

Soluciones

29. La reducción AJEDREZ $\leq_p$ PAR es imposible. Como PAR $\in$ P, las propiedades de las reducciones nos llevarían a afirmar que AJEDREZ $\in$ P. Pero hemos visto previamente que AJEDREZ no se puede resolver en tiempo polinómico y, por lo tanto, no es posible hacer la reducción.

En cambio, la reducción contraria PAR $\leq_p$ AJEDREZ es factible según las propiedades de las reducciones (PAR $\in$ P, AJEDREZ $\in$ EXP). La reducción $f(x)$ se podría hacer de la forma siguiente: si x es par, retorna un tablero de ajedrez donde el rey negro está rodeado de 9 reinas blancas (jaque mate en un movimiento); si x es impar, entonces retorna un tablero de ajedrez donde sólo queda el rey blanco y el rey negro (es imposible ganar la partida).

30. Sabemos que PARTICION $\in$ NP y, por lo tanto, podemos afirmar que SUMA_SUB $\in$ NP (dato que también podemos obtener definiendo el algoritmo verificador para SUMA_SUB).

Por otro lado, si consiguiéramos demostrar que SUMA_SUB $\notin$ P, entonces sabríamos que PARTICION $\notin$ P. Pero de momento no se ha conseguido averiguar si SUMA_SUB pertenece a P.

31. Cierto, ya que para cualquier problema Prob tenemos Prob $\leq_p$ Prob utilizando como función de reducción la función identidad $f(x) = x$.

3.3. Completitud

Definición 25

Sea C una clase de complejidad y sea Prob un problema. Prob es un **problema C-Difícil** si cualquier problema de la clase C se puede reducir polinómicamente a Prob, es decir:

$$\text{Prob es } C\text{-Difícil} \quad \leftrightarrow \quad \forall \text{A} \in C : \text{A} \leq_p \text{Prob}$$

Si, además de ser C-Difícil, Prob $\in C$, diremos que Prob es un **problema C-Completo**.

Intuitivamente, un problema C-Difícil es tan o más complejo que cualquiera de la clase C, ya que todos ellos se pueden resolver en base a éste. En el caso de un problema C-Completo, como además pertenece a la clase C, es uno de los problemas que comparten la complejidad máxima dentro de esta clase.

Aunque el concepto de C-Difícil y C-Completo se puede aplicar a cualquier clase de complejidad, lo utilizaremos especialmente para referirnos a las clases NP y EXP. Así pues, hablaremos de problemas NP-Difíciles, NP-Completos, EXP-Difíciles o EXP-Completos.

Nomenclatura

En inglés, NP-Completo se llama NP-*Complete* y NP-Difícil se llama NP-*Hard*.

Teorema 26

Teorema de Cook: El problema SAT es NP-Completo.

Lectura complementaria

La demostración del teorema de Cook se puede encontrar en los libros de Garey y Johnson (1979) y Papadimitriou (1994).

Demostración: Este teorema, demostrado en 1971, es uno de los resultados fundamentales de la Teoría de la Complejidad. Aparte de justificar que SAT $\in$ NP como ya hemos visto previamente, Cook demostró que cualquier problema NP se puede codificar de forma equivalente como una fórmula booleana en FNC, y que este proceso de codificación tenía complejidad polinómica. La demostración es larga y compleja y no la veremos aquí. ∎

Ejemplo 41

Los problemas PARTICION y SUMA_SUB también son problemas NP-Completos (no veremos la demostración).

Las propiedades de las reducciones, como por ejemplo la transitividad, permiten extraer algunas propiedades interesantes de los problemas completos y difíciles. Por ejemplo, nos proporcionan mecanismos para identificar nuevos problemas completos y difíciles a partir de problemas difíciles conocidos.

Propiedad 27

Todos los problemas C-Completos de una clase de complejidad C son polinómicamente equivalentes entre sí.

Demostración: Sean A y B dos problemas C-Completos. Sabemos que todos los problemas de la clase C se tienen que reducir polinómicamente tanto a A como a B. Además, al ser A y B completos, ambos problemas pertenecen a C (A,B $\in C$). Esto significa que existen las reducciones A $\leq_p$ B (ya que A $\in C$ y B es C-Completo) y B $\leq_p$ A (ya que B $\in C$ y A es C-Completo). Como existen A $\leq_p$ B y B $\leq_p$ A, hemos demostrado que A y B son polinómicamente equivalentes.

Fijaos que este razonamiento no permitiría demostrar que todos los problemas C-Difíciles son polinómicamente equivalentes entre sí (propiedad totalmente falsa), ya que no podríamos asegurar que A,B $\in C$. ∎

Propiedad 28

Sea C una clase de complejidad y Prob un problema C-Difícil. Si tenemos un problema A tal que Prob $\leq_p$ A, entonces A también es C-Difícil.

Demostración: Necesitamos demostrar que, dado cualquier problema B $\in C$, podemos hacer una reducción B $\leq_p$ A. Para hacer la demostración, utilizaremos la propiedad transitiva de las reducciones.

En primer lugar, sabemos que Prob es C-Difícil. Esto quiere decir que para cualquier problema B $\in C$, podemos hacer una reducción B $\leq_p$ Prob. Como según la hipótesis tenemos la reducción Prob $\leq_p$ A, por transitividad podemos asegurar que existe la reducción B $\leq_p$ A. ∎

Esta propiedad nos ofrece una forma sencilla de demostrar que un problema es difícil respecto a una clase: en lugar de tener que demostrar que todos los problemas de la clase se pueden reducir a él, es suficiente construir una sola reducción a partir de un problema difícil previamente conocido.

Ejemplo 42

A partir del teorema de Cook se puede demostrar que el problema 3SAT es NP-Difícil si definimos una reducción SAT $\leq_p$ 3SAT. En SAT las cláusulas pueden tener cualquier número de literales, mientras que 3SAT es una ver-

sión restringida del problema donde todas las cláusulas tienen exactamente tres literales. Por lo tanto, cada cláusula de SAT se tiene que transformar en un conjunto equivalente de cláusulas de tres literales. Parece lógico que el procedimiento de traducción distinga cuatro casos de cláusulas de SAT: las de un literal, las de dos literales, las de tres literales y las de cuatro o más literales.

Utilizamos las letras a,b,a_i,... para indicar los literales de las cláusulas SAT. Al hacer la traducción a 3SAT tendremos que introducir nuevas variables en la fórmula, que denominaremos mediante las letras α,β,α_i,...

Literales	**Cláusula de SAT**	**Cláusulas en 3SAT**
1	(a)	$(a \vee \alpha \vee \beta) \wedge (a \vee \overline{\alpha} \vee \beta) \wedge (a \vee \alpha \vee \overline{\beta}) \wedge (a \vee \overline{\alpha} \vee \overline{\beta})$
2	$(a \vee b)$	$(a \vee b \vee \alpha) \wedge (a \vee b \vee \overline{\alpha})$
3	$(a \vee b \vee c)$	$(a \vee b \vee c)$
4 o más	$(a_1 \vee a_2 \vee \ldots \vee a_k)$	$(a_1 \vee a_2 \vee \alpha_1) \wedge (a_3 \vee \overline{\alpha_1} \vee \alpha_2) \wedge \ldots \wedge (a_{k-2} \vee \overline{\alpha_{k-4}} \vee \alpha_{k-3}) \wedge (a_{k-1} \vee a_k \vee \overline{\alpha_{k-3}})$

La fórmula 3SAT resultante sería la conjunción de las traducciones de todas las cláusulas SAT. Una vez propuesta la función f de reducción SAT $\leq_p$ 3SAT, nos hace falta comprobar que es una reducción correcta. Los dos pasos para hacerlo son (1) demostrar que f se puede calcular en tiempo polinómico y (2) demostrar que SAT(x) = 3SAT($f(x)$).

El tiempo de cálculo polinómico se puede justificar fácilmente. Supongamos que la fórmula SAT de partida tiene n cláusulas y un máximo de k literales por cláusula. La traducción hace un recorrido de las cláusulas (tiempo $O(n)$) y en cada cláusula hace una traducción que genera, según el número de literales de la cláusula, 4, 2, 1 o k nuevas cláusulas (tiempo $O(k)$). Por lo tanto, la traducción se realiza en tiempo polinómico.

Pasemos pues a demostrar la equivalencia. La idea de la traducción es utilizar variables auxiliares para crear cláusulas de tres literales equivalentes a las de partida en SAT. Cuando hay 1 o 2 literales en la cláusula, la equivalencia es clara: introducimos tantas variables nuevas como literales falten para llegar a 3 y generamos tantas cláusulas como hagan falta para generar las posibles combinaciones de estas nuevas variables (4 y 2 respectivamente). Las cláusulas ya inicialmente de tres literales en SAT sirven tal y como están en el problema 3SAT y no es necesario modificarlas.

En el último caso con 4 o más literales, hacemos la demostración detalladamente:

- Caso SAT(x) = SÍ: Algún literal a_i tiene que evaluar a 1. Si fijamos todas las variables α_i a 0 excepto la que aparezca sin negar en la misma cláusula que a_i podremos satisfacer todas las cláusulas de $f(x)$ y, por lo tanto, 3SAT($f(x)$) = SÍ.
- Caso SAT(x) = NO: No hay ningún literal a_i que se satisfaga y, por lo tanto, $f(x)$ sólo podría satisfacerse usando los literales α_i. Tenemos suficiente con ver que siempre queda alguna cláusula en 3SAT sin satisfacer. El razonamiento sería el siguiente: para satisfacer la primera cláusula, α_1 tiene que ser 1 ya que los literales a_i son todos 0; entonces, para satisfacer la segunda cláusula, α_2 tiene que ser 1; seguiendo así, llegamos a la última cláusula teniendo que α_{k-3} tiene que ser 1 y, por lo tanto, no se satisface la última cláusula. Así pues, hemos demostrado que 3SAT($f(x)$) = NO.

En definitiva, hemos construido la reducción SAT $\leq_p$ 3SAT. Como sabemos que SAT es NP-Difícil, podemos concluir que 3SAT es NP-Difícil.

Se podría ir más allá y demostrar que 3SAT es NP-Completo además de NP-Difícil. Para demostrarlo sólo nos faltaría ver que el problema 3SAT pertenece a NP, es decir, que se puede verificar en tiempo polinómico (repasad el ejercicio de autoevaluación 4).

Corolario 29

Si Prob es un problema NP-Completo cualquiera, entonces Prob $\in$ P si y sólo si P = NP.

Demostración: Tenemos que demostrar dos cosas: si Prob $\notin$ P entonces P $\neq$ NP y si Prob $\in$ P entonces P = NP.

La primera demostración es directa: como Prob es un problema NP-Completo, Prob $\in$ NP. Como Prob pertenece a NP pero no pertenece a P, tenemos que P $\neq$ NP.

La segunda demostración es más interesante: si Prob $\in$ P, entonces podemos demostrar que cualquier problema NP es resoluble en tiempo polinómico. Sea A un problema NP cualquiera, entonces describiremos el algoritmo que lo calcularía:

- Como Prob $\in$ P (según la hipótesis), Prob(x) se puede calcular en tiempo polinómico.
- Como Prob es NP-Completo y A $\in$ NP, existe una reducción polinómica A $\leq_p$ Prob. Es decir, hay una función f calculable en tiempo polinómico tal que A(x) = Prob($f(x)$).
- Como tanto Prob como f se pueden calcular en tiempo polinómico, tenemos que A(x) = Prob($f(x)$) se puede calcular en tiempo polinómico y, por lo tanto, A $\in$ P.

∎

Este último corolario nos aporta un posible método para decidir la cuestión P $\stackrel{?}{=}$ NP: sólo necesitaríamos demostrar la pertenencia de un problema NP-Completo en la clase P (o su no pertenencia) para concluir que P = NP (respectivamente, P $\neq$ NP). Es decir, si encontráramos un algoritmo que resolviera un problema NP-Completo en tiempo polinómico o bién demostráramos que no hay ninguno, habríamos resuelto uno de los problemas abiertos más importantes del campo de la Informática Teórica.

Ejercicios

32. Demostrad que la propiedad "ser polinómicamente equivalente" es transitiva, es decir, que si tenemos tres problemas A, B y C con A polinómicamente equivalente a B y B polinómicamente equivalente a C, entonces A y C son polinómicamente equivalentes.

33. Según hemos definido las reducciones polinómicas, ¿qué problemas de la clase P son P-Completos? ¿Y qué problemas de la clase NP son P-Completos?

34. ¿Una reducción PAR $\leq_p$ SAT nos permite demostrar que P = NP? ¿Y una reducción SAT $\leq_p$ PAR?

35. Todo problema NP-Difícil tiene que ser NP-Completo. ¿Cierto o falso?

Soluciones

32. Tenemos que demostrar que existen dos reducciones polinómicas $\mathsf{A} \leq_p \mathsf{C}$ y $\mathsf{C} \leq_p \mathsf{A}$. Para hacerlo, nos aprovecharemos de la propiedad transitiva de las reducciones:

- Como A y B son polinómicamente equivalentes, tenemos $\mathsf{A} \leq_p \mathsf{B}$ y $\mathsf{B} \leq_p \mathsf{A}$.
- Como B y C son polinómicamente equivalentes, tenemos $\mathsf{B} \leq_p \mathsf{C}$ y $\mathsf{C} \leq_p \mathsf{B}$.
- Como tenemos las reducciones $\mathsf{C} \leq_p \mathsf{B}$ y $\mathsf{B} \leq_p \mathsf{A}$, por transitividad tenemos la reducción $\mathsf{C} \leq_p \mathsf{A}$.
- Como tenemos las reducciones $\mathsf{A} \leq_p \mathsf{B}$ y $\mathsf{B} \leq_p \mathsf{C}$, por transitividad tenemos la reducción $\mathsf{C} \leq_p \mathsf{A}$.
- Como tenemos las reducciones $\mathsf{A} \leq_p \mathsf{C}$ y $\mathsf{C} \leq_p \mathsf{A}$, entonces A y C son polinómicamente equivalentes.

33. Todos los problemas de la clase P son P-Completos, excepto dos problemas: ProbSI, el que tiene solución SÍ para todas las entradas y ProbNO, el que tiene solución NO para todas las entradas. La demostración es la misma que hemos utilizado para ver que todos los problemas de P (excepto estos dos) son polinómicamente equivalentes.

Respecto a la clase NP, es una pregunta con trampa: sólo los problemas de P pueden ser P-Completos.

34. Sabemos que $\mathsf{PAR} \in$ P y $\mathsf{SAT} \in$ NP-Completo. Una reducción $\mathsf{PAR} \leq_p \mathsf{SAT}$ no nos aporta más información sobre la complejidad de PAR y SAT. En cambio, una reducción $\mathsf{SAT} \leq_p \mathsf{PAR}$ nos indicaría que $\mathsf{SAT} \in$ P y como SAT es NP-Completo podríamos concluir que P = NP.

35. Falso. La definición nos garantiza justamente lo contrario: que todo problema NP-Completo es NP-Difícil.

Ejercicios de autoevaluación

1. Enunciad los problemas de cálculo SUMATORIO y AISLADOS (ejemplos 7 y 8) como problemas de decisión.

2. Enunciad los problemas de optimización CPP, LCS y MCP (ejemplos 9, 10 y 11) como problemas de decisión.

3. Hemos encontrado en Internet una definición alternativa del problema de la suma de subconjuntos (SUMA_SUB). La nueva definición es la siguiente:

SUMA_SUB_ALT: "Dado un conjunto de enteros C, decidir si algún subconjunto de C tiene suma total igual a 0"

a) Describid los problemas de decisión SUMA_SUB y SUMA_SUB_ALT en forma de función.

b) Demostrad que el problema SUMA_SUB_ALT está en NP.

c) Describid una reducción polinómica SUMA_SUB_ALT $\leq_p$ SUMA_SUB.

d) Si SUMA_SUB es NP-Completo, la reducción anterior nos permite demostrar que el problema SUMA_SUB_ALT es NP-Difícil. ¿Cierto o falso?

4. Demostrad que el problema 3SAT pertenece a la clase NP.

5. Dados dos problemas A y B, justificad si son ciertas o falsas las afirmaciones siguientes:

a) Si A $\leq_p$ B y B $\in$ P, entonces A $\in$ P.

b) Si A $\leq_p$ B y A $\in$ P, entonces B $\in$ P.

c) Si A $\leq_p$ B y B $\in$ NP, entonces A $\in$ NP.

d) Si A $\leq_p$ B y A $\in$ NP, entonces B $\in$ NP.

e) Si A $\leq_p$ B y A es NP-Difícil, entonces B es NP-Difícil.

f) Si A $\leq_p$ B y A es NP-Completo, entonces B es NP-Completo.

g) Si A $\leq_p$ B y A es NP-Completo, entonces B es NP-Difícil.

h) Si S(A) $\leq_p$ B, donde S(A)$^\bullet$ es un subconjunto de A$^\bullet$ y A es NP-Difícil, entonces B es NP-Difícil.

i) Si A $\leq_p$ S(B), donde S(B)$^\bullet$ es un subconjunto de B$^\bullet$, y A es NP-Difícil, entonces B es NP-Difícil.

6. Dado un número natural N, considerad el problema SUMA_N: "Calculad la suma de los N primeros números naturales" (repasad el ejemplo 24).

a) Reformulad el problema SUMA_N como un problema de decisión.

b) Proponemos el algoritmo siguiente para resolver el problema SUMA_N: "sumamos consecutivamente los números $1,2,3,\ldots,N-1,N$". Calculad, en función de N, la complejidad de este algoritmo. ¿Podemos afirmar que SUMA_N $\in$ P?

c) Calculad, en función del número de bits necesarios para representar N, la complejidad del algoritmo propuesto. ¿Podemos afirmar que SUMA_N $\in$ NP?

d) ¿Podemos afirmar que SUMA_N $\notin$ P? Justificad la respuesta.

7. Considerad un conjunto finito X de N elementos. Queremos encontrar todas las funciones biyectivas de X en X*.

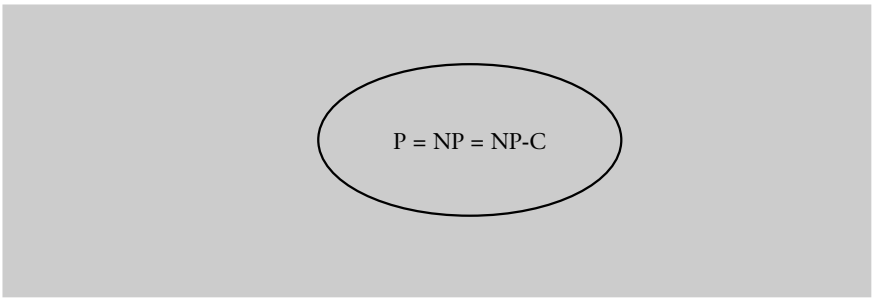
***Repasad el apartado 1 sobre funciones del módulo "Conceptos previos: funciones y algoritmos".**

a) Definimos el problema BIYECCIONES: "Listad todas las funciones biyectivas del conjunto *X*". Cada función biyectiva puede describirse mediante una lista de pares (valor, imagen) para cada valor del dominio. ¿A qué clases de complejidad (tiempo y espacio) pertenece este problema? ¿Puede pertenecer a la clase NP? Justificad las respuestas.

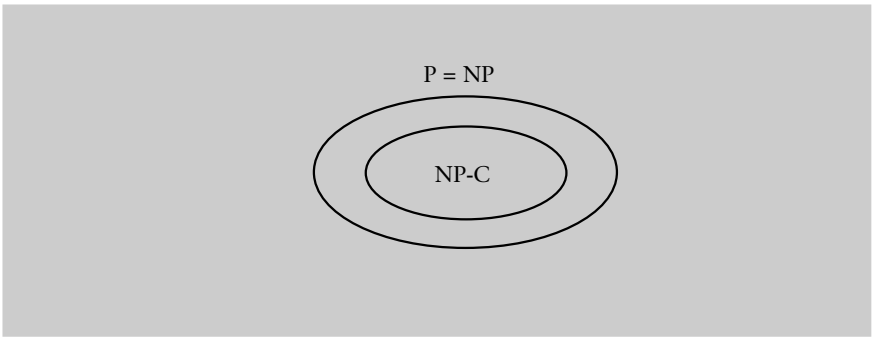
b) Considerad a continuación el problema NUMERO_BIYECCIONES: "Calculad el número de funciones biyectivas del conjunto *X*". ¿A qué clases de complejidad (tiempo y espacio) pertenece este problema? ¿Es equivalente al anterior?

8. Actualmente, las relaciones entre las clases de complejidad P, NP y NP-C (NP-Completo) no están claramente delimitadas. Para cada uno de los diagramas siguientes, indicad si el escenario que se plantea es factible y si se está realizando alguna suposición actualmente no demostrada sobre la relación entre estas 3 clases de complejidad.

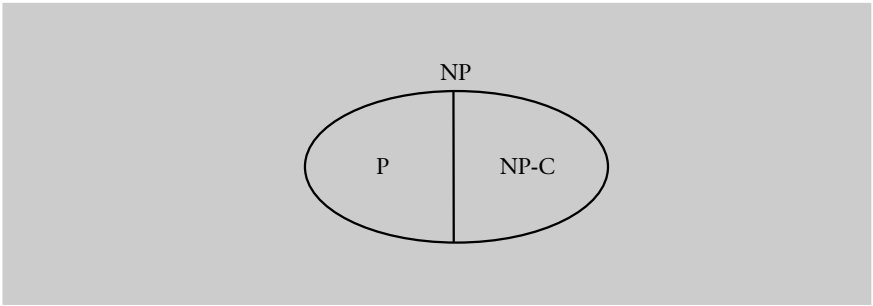
a)



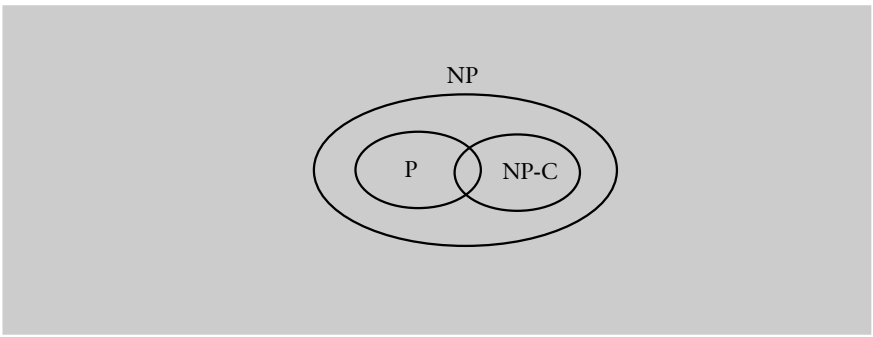
b)



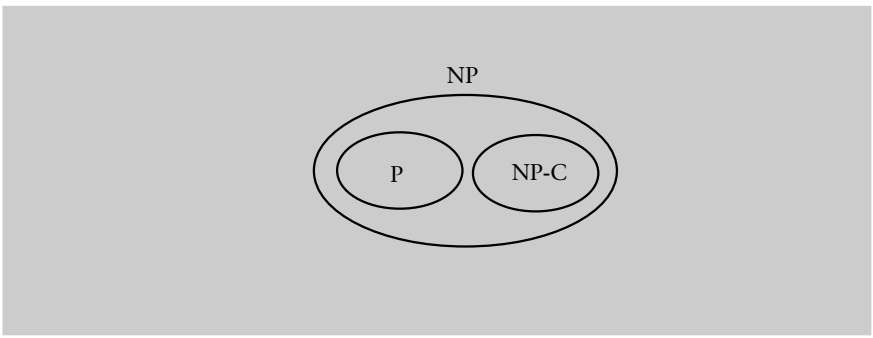
c)



d)



e)



9. Algunos textos sobre complejidad computacional definen PARTICION sobre un multiconjunto de números naturales en lugar de un multiconjunto de números enteros. Es decir, en el multiconjunto de números a particionar no se aceptan números negativos. Llamamos PARTICION_NAT a esta versión del problema.

a) Describid una reducción polinómica PARTICION_NAT $\leq_p$ PARTICION.

b) Describid una reducción polinómica PARTICION $\leq_p$ PARTICION_NAT.

c) Indicad qué conclusiones se pueden llegar a extraer sobre la complejidad de PARTICION_NAT a partir de los apartados anteriores.

10. El problema de la planificación de un multiprocesador (MPSCHED, *multiprocessor scheduling*) estudia la asignación de un conjunto de tareas a un grupo de procesadores de forma que su ejecución acabe antes de un tiempo límite. Formalmente, el problema se puede definir en versión decisional de la forma siguiente:

Entrada: un conjunto finito A de tareas, una duración $d(a) \in \mathbb{N}$ para cada $a \in A$, un número $m \in \mathbb{N}$ de procesadores y una fecha límite $D \in \mathbb{N}$.

Pregunta: existe una partición $A = A_1 \cup A_2 \cup \cdots \cup A_m$ de A en m conjuntos disjuntos tal que $\max(\sum_{a \in A_i} d(a) \mid 1 \leq i \leq m) \leq D$?

a) Indicad si MPSCHED se puede verificar en tiempo polinómico.

b) Partiendo del problema de decisión PARTICION_NAT definido en el ejercicio anterior, proponed una reducción polinómica PARTICION_NAT $\leq_p$ MPSCHED (Indicación: considerad $m = 2$).

c) ¿Sería posible hacer la reducción PARTICION $\leq_p$ MPSCHED, llegando a utilizar PARTICION en lugar de PARTICION_NAT? (Indicación: centraos en la diferencia entre PARTICION y PARTICION_NAT).

d) Indicad qué conclusiones se pueden extraer de los apartados anteriores sobre la complejidad del problema MPSCHED.

Solucionario

1. SUMATORIO: "Dada una secuencia finita de enteros E y un entero k, decidir si $k = \sum_{x \in E} x$."

AISLADOS: "Dado un grafo $G = (V,A)$ y un conjunto de vértices V', decidir si el conjunto de vértices de grado 0 de V es igual a V'."

2. CPP: "Dado un conjunto de puntos en un espacio bidimensional, donde cada punto está definido por unas coordenadas (x,y), y un número real d, decidir si hay algún par de puntos que estén a distancia d o inferior."

LCS: "Dadas dos cadenas de caracteres a y b y un entero k, decidir si la subcadena común más larga entre a y b tiene k caracteres o más."

MSP: "Dado un grafo ponderado y conexo (G,w) y un número real k, decidir si el árbol generador minimal tiene peso k o menos."

3. **a)** SUMA_SUB : $2^{\mathbb{Z}} \times \mathbb{Z} \to \{\text{SÍ, NO}\}$
SUMA_SUB_ALT : $2^{\mathbb{Z}} \to \{\text{SÍ, NO}\}$

b) La entrada del problema es un conjunto C. Un testigo sería un subconjunto $C' \subseteq C$, concretamente el subconjunto con suma 0.

A partir del conjunto de entrada y el testigo, es posible validar que todos los elementos de C' estaban en el conjunto original C y que la suma de todos los elementos de C' es igual a 0. Todo esto se puede comprobar en tiempo polinómico.

También es directo ver que si SUMA_SUB_ALT(C) = NO, cualquier subconjunto que propongamos como testigo no pasará la validación al no tener una suma total igual a 0 (si hubiera alguno, la respuesta al problema sería SÍ).

c) La reducción SUMA_SUB_ALT $\leq_p$ SUMA_SUB es bastante directa: usamos como función de reducción $f(C) = (C,0)$. Esta reducción fija como valor objetivo $t = 0$ sin modificar el conjunto C. Entonces SUMA_SUB_ALT(C) = SUMA_SUB(C,0) para cualquier C, independientemente de si el resultado es SÍ o NO. La función f es calculable en tiempo polinómico y, por lo tanto, queda demostrado que la reducción SUMA_SUB_ALT $\leq_p$ SUMA_SUB es posible.

d) Falso. La reducción que permitiría llegar a esta conclusión es SUMA_SUB $\leq_p$ SUMA_SUB_ALT.

4. Para demostrar que 3SAT $\in$ NP tenemos que justificar que 3SAT se puede verificar en tiempo polinómico. En el ejemplo 29, ya hemos visto que hay un procedimiento para verificar SAT en tiempo polinómico. Dado que 3SAT es un caso particular de SAT, el procedimiento del ejemplo 29 también nos permite verificar 3SAT en tiempo polinómico.

5. Justificamos cada afirmación:

a) Cierta, por las propiedades de las reducciones polinómicas.

b) Falsa. La reducción no nos aporta ninguna información sobre la complejidad de B.

c) Cierta, por las propiedades de las reducciones.

d) Falsa. La reducción no nos aporta ninguna información sobre la complejidad de B.

e) Cierta. Si A es NP-Difícil, entonces C $\leq_p$ A para todo C $\in$ NP. Por la propiedad de transitividad, C $\leq_p$ B para todo C $\in$ NP y esto quiere decir que B es NP-Difícil.

f) Falsa. Si A es NP-Completo, entonces $\mathsf{C} \leq_p \mathsf{A}$ para todo $\mathsf{C} \in$ NP y $\mathsf{A} \in$ NP. Por la propiedad de transitividad, $\mathsf{C} \leq_p \mathsf{B}$ para todo $\mathsf{C} \in$ NP, pero no podemos afirmar que $\mathsf{B} \in$ NP.

g) Cierta. Si A es NP-Completo, entonces $\mathsf{C} \leq_p \mathsf{A}$ para todo $\mathsf{C} \in$ NP y $\mathsf{A} \in$ NP. Por la propiedad de transitividad, $\mathsf{C} \leq_p \mathsf{B}$ para todo $\mathsf{C} \in$ NP y esto quiere decir que B es NP-Difícil.

h) Falsa. Un subconjunto de un problema NP-Difícil puede no ser NP-Difícil. Por ejemplo, podríamos elegir S(A) = ProbNO, que no es NP-Difícil (recordemos que los problemas con alguna entrada con solución SÍ no se pueden reducir a este problema y, por lo tanto, no es posible que todos los problemas de NP se puedan reducir a él). Por lo tanto, la reducción $\mathsf{ProbNO} \leq_p \mathsf{B}$ no nos aporta suficiente información para asegurar que B es NP-Difícil.

i) Falsa. Aunque un problema S(B) sea NP-Difícil, esto no significa que los problemas B que lo incluyen ($\mathsf{S(B)}^\bullet \subseteq \mathsf{B}^\bullet$) sean NP-Difíciles. Por ejemplo, podríamos escoger B = ProbSI, que sabemos que no es NP-Difícil (los problemas que tienen alguna entrada con solución NO no se pueden reducir a este problema).

6. a) El problema de decisión asociado sería, SUMA_N: "Dado un natural N y un natural k, decidir si $\sum_{i=1}^{N} i = k$".

b) En el algoritmo propuesto tenemos que hacer $N-1$ sumas por lo cual tendrá una complejidad $O(N)$, en función de N. Aunque la complejidad que hemos calculado es polinómica, no podemos afirmar que SUMA_N $\in$ P dado que no hemos tenido en cuenta el tamaño de la entrada del algoritmo.

c) El número de bits necesarios para representar N es $\lfloor \log_2 N \rfloor + 1$. La complejidad será ahora $O(2^n)$. Con la información que hemos obtenido, podemos afirmar que SUMA_N $\in$ EXP.

Para decidir si SUMA_N $\in$ NP tenemos que ver si el problema de decisión SUMA_N es verificable en tiempo polinómico, es decir, si podemos verificar en tiempo polinómico que $\sum_{i=1}^{N} i = k$ donde k es el valor de la suma de los N primeros números naturales. Como función verificadora no podemos elegir el algoritmo propuesto dado que tiene una complejidad exponencial en función del tamaño de la entrada.

Por suerte, la fórmula de Euler $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$ se puede utilizar como función verificadora, $v : \mathbb{N} \times \mathbb{N} \to \{\text{SÍ,NO}\}$ definida por, si SUMA_N = k entonces $v(N,k)$ = SÍ si $\frac{N(N+1)}{2} = k$ y $v(N,k)$ = NO, en otro caso. Esta función verificadora tiene una complejidad polinómica y, por lo tanto, podemos concluir que SUMA_N $\in$ NP.

d) Para poder afirmar que SUMA_N $\notin$ P tendríamos que estar seguros de que no hay ningún algoritmo de complejidad polinómica para calcular la suma de los N primeros números naturales. Pero en el apartado anterior hemos visto que la fórmula de Euler resuelve el problema con una complejidad polinómica. Por lo tanto, podemos afirmar que SUMA_N $\in$ P.

Observad que la fórmula de Euler sirve al mismo tiempo para demostrar que SUMA_N $\in$ P y también como función verificadora (revisad el ejercicio 18).

7. El número de funciones biyectivas de un conjunto finito X de N elementos coincide con el número de N-muestras ordenadas sin repetición de elementos de X que coincide con el número de permutaciones $P(N)$ de los elementos de X*. Así, el número de permutaciones es $P(N) = N!$.

***Proposición 4 del módulo "Conceptos previos: funciones y algoritmos".**

a) La solución al problema BIYECCIONES consistirá en dar las $N!$ muestras ordenadas sin repetición de N elementos de X. Como en el ejemplo 27, la salida de este problema tendrá un tamaño exponencial respecto a la entrada, concretamente, $N!$ veces el tamaño de la entrada. Así pues, este problema pertenece a EXP y a EXPSPACE pero no a P ni a PSPACE.

Como BIYECCIONES $\notin$ PSPACE y NP $\subseteq$ PSPACE, podemos afirmar como conclusión que BIYECCIONES $\notin$ NP.

b) La solución al problema NUMERO_BIYECCIONES consistirá en calcular $N!$. La diferencia con el apartado anterior es que no es necesario dar todas las permutaciones, sino sólo saber cuántas hay.

Calcular $N!$ requiere $N-1$ multiplicaciones y, por lo tanto, se puede calcular en tiempo lineal. Esto quiere decir que NUMERO_BIYECCIONES $\in$ P y como consecuencia NUMERO_BIYECCIONES $\in$ PSPACE. Como conclusión, NUMERO_BIYECCIONES no es equivalente a BIYECCIONES.

Fijémonos que en este problema la entrada no es un valor del que tenemos que medir su longitud en representación binaria, sino un conjunto de N elementos y, por lo tanto, el tamaño de la entrada es N (el número de elementos del conjunto). Para poner un ejemplo de esta diferencia, no es lo mismo un problema donde la entrada es un conjunto de 75000 números que otro problema donde la entrada es el valor "75000": en el primer caso, el tamaño de la entrada es 75000, mientras que en el segundo caso el tamaño es $\lfloor \log_2 75000 \rfloor + 1 = 17$.

8. En la situación actual del conocimiento, P, NP-C son subconjuntos de NP.

a) Factible, pero asume que P = NP.

b) No factible. Si P = NP, entonces P = NP = NP-C. Por lo tanto, esta situación no puede suceder.

c) Factible, pero asume que P $\neq$ NP y P $\neq$ NP-C.

d) Factible, pero en este diagrama se supone que hay problemas que están al mismo tiempo en P y en NP-C. Por la definición de NP-C, entonces estamos asumiendo que P = NP = NP-C.

e) Factible, pero asume que P $\neq$ NP y P $\neq$ NP-C.

9. a) La reducción PARTICION_NAT $\leq_p$ PARTICION es directa, utilizando como función de reducción $f(x) = x$, la función identidad. Es decir, como todo número natural es también un número entero, toda entrada de PARTICION_NAT es una entrada correcta de PARTICION. La solución calculada por PARTICION para esta entrada será igual a la de PARTICION_NAT.

b) La reducción PARTICION $\leq_p$ PARTICION_NAT es más interesante: partiendo de un multiconjunto de enteros que puede contener números negativos, tenemos que construir un multiconjunto de números naturales que preserve la solución. La función de reducción que utilizaremos será $g(C)$ donde $g(C)$ contiene el multiconjunto resultante de aplicar la función "valor absoluto" a cada elemento de C. Es decir, para $C = [2,4,0,-4]$, $g(C) = [2,4,0,4]$.

Esta función se puede calcular en tiempo polinómico: sólo es necesario hacer un recorrido del multiconjunto, cambiando el signo a los valores negativos.

Nos queda demostrar que PARTICION(C) = PARTICION_NAT($g(C)$), tanto para los casos donde PARTICION(C) = SÍ como cuando PARTICION(C) = NO. Procedemos a hacerlo a continuación:

- Caso PARTICION(C) = SÍ: En este caso, tenemos una partición de C en dos multiconjuntos C_1 y C_2 de forma que $\sum_{x \in C_1} x = \sum_{x \in C_2} x$. Veremos que es posible construir una partición equivalente en $g(C)$ obteniendo C'_1 y C'_2.

  La idea es la siguiente: si no hay números negativos en C, entonces $g(C) = C$ y la misma partición nos sirve para $g(C)$ ($C'_1 = C_1, C'_2 = C_2$). Si no, construimos C'_1 y C'_2 de la forma siguiente: todos los valores no negativos de C_1 van a C'_1 y todos los valores no negativos de C_2 van a C'_2; si hay algún valor negativo en C_1, su valor absoluto pasa a C'_2 y viceversa.

Por ejemplo, si $C_1 = [7, -1, 0, 2, 2]$ y $C_2 = [8, 4, 6, -3, -5]$ tendríamos que $C'_1 = [7, 0, 2, 2, 3, 5]$ y $C'_2 = [8, 4, 6, 1]$. Intuitivamente, es lo mismo sumar $-x$ a C_1 que sumar $+x$ a C_2, de forma que esta transformación preserva la suma total de los elementos de la partición.

- Caso PARTICION(C) = NO: Si no hay ninguna partición en C donde la suma total de cada parte sea igual, tenemos que demostrar que tampoco hay ninguna en $g(C)$. Haremos la demostración por reducción al absurdo: supondremos que hay una partición para $g(C)$ cuando PARTICION(C) = NO y llegaremos a una contradicción.

  Supongamos que hay una partición para $g(C)$ en dos multiconjuntos C'_1 y C'_2 con la misma suma total. A partir de aquí, construiremos una partición de C en C_1 y C_2 que también tiene la misma suma total. La partición se construye de forma equivalente al apartado anterior: para cada $x \in C$, si $abs(x)$ estaba en C'_1, entonces lo añadimos a C_1 si $x \geq 0$ y a C_2 si $x < 0$; similarmente, si $abs(x)$ estaba en C'_2, entonces lo añadimos a C_2 si $x \geq 0$ y a C_1 si $x < 0$.

  Esta partición C_1 y C_2 preserva la suma total de cada parte, por el mismo razonamiento utilizado previamente: sumar $-x$ a una parte o sumar $+x$ a la otra son equivalentes. Esto quiere decir que si $\sum_{x \in C'_1} x = \sum_{x \in C'_2} x$, también se cumplirá $\sum_{x \in C_1} x = \sum_{x \in C_2} x$.

  La existencia de la partición C_1 y C_2 indica que PARTICION(C) = SÍ, contradiciendo nuestra hipótesis. Esto nos permite concluir que la hipótesis de que hay una partición para $g(C)$ cuando PARTICION(C) = NO es incorrecta, es decir, que si PARTICION(C) = NO entonces PARTICION_NAT($g(C)$) = NO.

c) La reducción PARTICION_NAT $\leq_p$ PARTICION y el hecho que PARTICION sea NP indica que PARTICION_NAT pertenece a NP. La reducción PARTICION $\leq_p$ PARTICION_NAT y el hecho que PARTICION sea NP-Difícil nos indica que el problema PARTICION_NAT es también un problema NP-Difícil. Considerando las dos reducciones al mismo tiempo, podemos concluir que PARTICION y PARTICION_NAT son polinómicamente equivalentes (por la existencia de las dos reducciones) y que PARTICION_NAT es un problema NP-Completo (al ser NP y NP-Difícil).

10. a) Empezamos proponiendo un testigo para MPSCHED: una partición del conjunto A en m conjuntos disjuntos.

La función de verificación consiste en sumar, para cada procesador j, la duración $d(a)$ de las tareas asignadas a j y comparar cada duración total con la fecha límite D. Una entrada se considera verificada si el testigo consigue que, en cada procesador, la duración total sea inferior a D.

Si hay n tareas y m procesadores, para este cálculo tendremos que hacer $n-m$ sumas y m comparaciones. Esto significa que la función de verificación es calculable en tiempo polinómico.

Si la solución de MPSCHED es SÍ, habrá algún testigo que nos permita verificar esta solución (por la propia definición del problema, que nos pide que exista esta partición). Claramente, si la solución para una entrada es NO, ningún testigo nos permitirá verificarla.

b) La reducción $f(x)$ ha de partir de una entrada del problema PARTICION_NAT y construir una entrada equivalente del problema MPSCHED. Es decir, que partiendo de un multiconjunto de naturales, tenemos que llegar a un conjunto de tareas, una duración para cada tarea, un número de procesadores y una fecha límite.

Intuitivamente, dado que PARTICION_NAT intenta dividir un multiconjunto en dos partes, tendríamos que plantejar un problema de MPSCHED con $m = 2$ procesadores. La duración de cada tarea debería estar relacionada con el valor de los enteros originales, y la fecha límite, con la suma total de los números naturales.

Concretamente, la propuesta de función de reducción $f(C)$ es: hay una tarea para cada número x de C, con duración d igual a x y D es la mitad de la suma total de los valores de C, es decir, $D = (\sum_{x \in C} x)/2$. Esta función de reducción se puede calcular en tiempo polinómico: sólo es necesario hacer un recorrido de C, calcular la suma total de los valores y hacer una división por 2.

Podemos demostrar la equivalencia entre las entradas de PARTICION_NAT y MPSCHED según esta reducción. La duración total de las tareas es $2D$ y, por lo tanto, sólo se puede conseguir que ambos procesadores completen el cálculo antes de la fecha límite D repartiendo las tareas de forma perfectamente equilibrada entre los procesadores (duración total D en cada procesador). Si no es así, uno de los procesadores tardaría menos de D y el otro tardaría más, incumpliendo la fecha límite. Esto es equivalente a conseguir particionar un conjunto de números naturales en dos partes con la misma suma total. Es decir, PARTICION_NAT(C) = MPSCHED($f(C)$).

c) La reducción desde PARTICION es posible, pero no se puede utilizar la misma función de reducción que se utiliza para PARTICION_NAT $\leq_p$ MPSCHED. En concreto, haría falta definir el mapeo para los números negativos del multiconjunto C, teniendo en cuenta que las duraciones de las tareas no pueden ser negativas. Por esto nos hemos basado en PARTICION_NAT en lugar de PARTICION para hacer la reducción.

La forma más sencilla de definir una función de reducción PARTICION $\leq_p$ MPSCHED podría hacerse básicamente componiendo la función que reduce PARTICION $\leq_p$ PARTICION_NAT (g, definida en un ejercicio de autoevaluación previo) con la que reduce PARTICION_NAT $\leq_p$ MPSCHED (f, definida en el apartado anterior). La función de reducción h para PARTICION $\leq_p$ MPSCHED se definiría como $h(C) = f(g(C))$.

d) A partir del apartado *a*, podemos decir que MPSCHED pertenece a NP. A partir del apartado *b*, podemos decir que MPSCHED es NP-Difícil. Combinando ambas informaciones, podemos decir que MPSCHED es NP-Completo.

Bibliografía

Cook, S. (2003). *The Importance of the* P *versus* NP *Question*. [s.l.]: Journal of the ACM (vol. 1, núm. 50, págs. 27-29).

Garey, M. R.; Johnson, D. S. (1979). *Computers and Intractability. A Guide to the Theory of* NP-*Completeness*. Nueva York: Freeman and Co.

Papadimitriou, C. H. (1994). *Computational Complexity*. Reading (Massachusetts): Addison Wesley.