Modelos Abstractos de Cálculo

Elvira Mayordomo Cámara

24 de septiembre de 2009

Capítulo 0

Presentación

La asignatura de Modelos Abstractos de Cálculo consta de dos partes: computabilidad (también llamada recursividad) y complejidad, cada una de ellas ocupa aproximadamente un 50 % de las clases de teoría y problemas.

Parte I: Computabilidad. Los contenidos fundamentales son los siguientes:

- Existen problemas que no se pueden resolver con ningún algoritmo.
- Veremos ejemplos importantes de estos problemas "irresolubles" y métodos para saber que un problema es de este tipo.
- Se conjetura que cualquier noción razonable de *algoritmo* da lugar al mismo conjunto de problemas resolubles.

EJERCICIOS

Escribir los siguientes procedimientos ada (y probarlos):

0.1. Function check(f:file_type; n:integer) return boolean;

{**Pre:** f es un fichero que contiene un programa ada que lee de teclado un único valor entero.

Post: devuelve true si el programa contenido en f con entrada n termina; devuelve false si se queda "colgado"}

0.2. Procedure fermat(n:in integer; x,y,z:out integer);

{**Pre:** $n \in \mathbb{N}$
Post: $x, y, z \in \mathbb{N}$ tales que $x^n + y^n = z^n$ }

Parte II: Complejidad. En esta parte veremos:

- Cómo medir la velocidad de un algoritmo en función del tamaño de la entrada.
- Existen problemas que se pueden resolver en tiempo razonable y otros no, se trata de los problemas tratables e intratables. Nos dedicaremos a los segundos.

EJERCICIO
Escribir un programa en ada que resuelva el siguiente problema (atención a la eficiencia):

0.3. Se trata de un viajante que necesita recorrer n ciudades en el menor tiempo posible. Disponemos de la distancia correspondiente a cada pareja de ciudades, $d(i, j)$ es la distancia de la ciudad i a la j, las ciudades están numeradas correlativamente de 1 a n.

Escribir un programa que dados n y $d(i, j)$ para todo i, j encuentre un camino (una ordenación de las n ciudades) de forma que la suma de las distancias recorridas en ese camino sea la mínima posible.

Bibliografía

[Jones] N. Jones: "Computability and Complexity From a Programming Perspective", MIT press, 1997.

[Cu80] N.J. Cutland: "Computability: An Introduction to Recursive Function Theory", Cambridge University Press, 1980.

[So87] R. Soare: "Recursively Enumerable Sets and Degrees", Springer-Verlag, 1987.

[LP81] H.R. Lewis, C.H. Papadimitriou: "Elements of the Theory of Computation", Prentice-Hall, 1981.

[GJ78] M. Garey, D. Johnson: "Computers and Intractability: A Guide to the Theory of NP-Completeness", Freeman, 1978.

[HMU02] J.E. Hopcroft, R. Motwani, J.D. Ullman, "Introducción a la Teoría de Autómatas, Lenguajes y Computación", Addison-Wesley, 2002.

[SACL01] M. Serna, C. Àlvarez, R. Cases, A. Lozano: "Els límits de la computació. Indecidibilitat i NP-completesa", Edicions UPC, 2001.

Capítulo 1

Preliminares. Numerabilidad y diagonalización

Referencia: Capítulo 1 de [LP81].

En el estudio de la calculabilidad y la complejidad, es imprescindible formular afirmaciones rigurosas y relacionarlas mediante deducciones rigurosas. En este contexto el lenguaje matemático nos permitirá expresarnos con precisión y agilidad.
Este capítulo contiene un repaso de la notación y conceptos principales de lógica, demostraciones, teoría de conjuntos, lenguajes y alfabetos, y funciones, además de una breve introducción a la teoría de cardinales y a la diagonalización.

1.1. Preliminares

1.1.1. Notación lógica: proposiciones

Una *proposición* o *enunciado* es una frase declarativa ó sentencia que podemos clasificar como cierta o falsa, por ejemplo "α es la primera letra del alfabeto griego", o "el sol se pone por el este". Si P y Q son dos proposiciones, podemos formar otras mediante las *conectivas* lógicas $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$:

Notación formal	Significado informal
$\neg P$	"no P"
$P \vee Q$	"P ó Q"
$P \wedge Q$	"P y Q"
$P \Rightarrow Q$	"P implica Q" ó "si P entonces Q" ó "P sólo si Q"
$P \Leftrightarrow Q$	"P si y sólo si Q" ó "P es equivalente a Q"

La asignación de valores de verdad a las proposiciones compuestas mediante las conectivas $\wedge, \vee, \neg$ corresponde a la interpretación usual de las palabras *y*, *o* y *no*, respectivamente.
El valor de verdad de la proposición $P \Rightarrow Q$ equivale al de $(\neg P) \vee Q$ de manera que la asignación de valores de verdad $P = cierto$ y $Q = falso$ es la única que hace falsa $P \Rightarrow Q$. La proposición $P \Rightarrow Q$ es equivalente a $(\neg Q) \Rightarrow (\neg P)$.
$P \Rightarrow Q$ NO es equivalente a $Q \Rightarrow P$. No es equivalente decir "Si Juan es inglés entonces Juan es europeo" a decir "Si Juan es europeo entonces Juan es inglés".
El valor de verdad de $P \Leftrightarrow Q$ corresponde al de $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$.
Para ahorrarnos paréntesis estableceremos precedencias entre las diferentes conectivas. En orden de precedencia decreciente tenemos:

$$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$$

1.1.2. Notación lógica: predicados

Un *predicado* es una sentencia que contiene una ó más variables. Cada variable puede tomar valor en un cierto universo. Por ejemplo, si x es un entero (variable en el universo de los enteros), el predicado

$$P(x) = (x \text{ es primo } \wedge (x \leq 100))$$

cumple que $P(19) = cierto$ y $P(20) = falso$. Cuando todas las variables se sustituyen por valores, el predicado se convierte en una proposición. Por ejemplo,

$$P(33) = (33 \text{ es primo } \wedge (33 \leq 100))$$

es una proposición falsa. Un predicado como el anterior, que contiene sólo una variable, se llama *propiedad*. Si $P(x)$ es cierto, decimos que x

tiene o cumple la propiedad P; si es falso, decimos que x no tiene o no cumple la propiedad P.
Otra forma de obtener proposiciones a partir de predicados consiste en *cuantificar* las variables. Si tenemos un predicado $P(x)$, podemos construir dos proposiciones nuevas mediante el *cuantificador existencial* $\exists$ y el *cuantificador universal* $\forall$ de la siguiente forma:

Notación formal	Significado informal
$\exists x P(x)$	Para algún x, $P(x)$
$\forall x P(x)$	Para todo x, $P(x)$

La proposición $\exists x P(x)$ es cierta si $P(x)$ es cierto para algún valor de x, mientras que $\forall x P(x)$ es cierta si $P(x)$ es cierto para todo posible valor de x. En el caso del cuantificador existencial, se introduce también el símbolo $\nexists$, que permite abreviar el enunciado $\neg(\exists P(x))$ con $\nexists P(x)$. El símbolo $\exists$ se puede considerar como la extensión de $\vee$ al caso infinito (y $\forall$ como extensión de $\wedge$).
El universo de valores que puede tomar una variable (los números naturales, las palabras sobre un determinado alfabeto, etc) depende del predicado concreto y es usual no especificarlo si se puede deducir fácilmente del contexto (por ejemplo, en el caso del predicado $P(x)$ del ejemplo anterior). Por otro lado, podemos escribir

$$\exists x, y, z \in \mathbb{N}^+ \; x^2 + y^2 = z^2$$

para establecer claramente que x, y, z toman valores naturales positivos. También podemos escribir

$$\forall n > 2 \; P(n)$$

para indicar que n toma valores naturales mayores que 2.
Existen toda una serie de equivalencias entre sentencias, por ejemplo las llamadas *leyes de Morgan*:

1. $\neg(P \wedge Q) \Leftrightarrow (\neg P) \vee (\neg Q)$
2. $\neg(\forall x P(x)) \Leftrightarrow \exists x \neg P(x)$

y las simétricas que se obtienen intercambiando las conectivas $\wedge$ y $\vee$, en 1, y los cuantificadores $\forall$ y $\exists$, en 2.

1.1.3. Demostraciones

Un *teorema* es cualquier proposición para la que existe una demostración.
A lo largo de esta asignatura utilizaremos algunos métodos de demostración:

1. *Demostración directa*: a partir de una serie de teoremas ya conocidos deducimos un nuevo teorema.

2. *Reducción al absurdo*: para demostrar P demostramos que su negación implica una contradicción, es decir $\neg P \Rightarrow falso$. Por ejemplo, para demostrar $X \Rightarrow Y$ demostramos que $X \wedge \neg Y$ implica que todos los naturales son pares.

Como hemos visto anteriormente, es equivalente demostrar $P \Rightarrow Q$ que demostrar $\neg Q \Rightarrow \neg P$, y para demostrar $P \Leftrightarrow Q$ hemos de demostrar que $P \Rightarrow Q$ y $Q \Rightarrow P$.

1.1.4. Notación de conjuntos

Dado un *universo* de elementos (por ejemplo, los números naturales), representaremos con

$$\{x \,|\, P(x)\}$$

el conjunto de los elementos que cumplen la propiedad P.
Utilizaremos las operaciones:

1. Unión: $A \cup B = \{x \,|\, x \in A \vee x \in B\}$.

2. Intersección: $A \cap B = \{x \,|\, x \in A \wedge x \in B\}$.

3. Diferencia: $A - B = \{x \,|\, x \in A \wedge x \notin B\}$.

4. Complementación: $\overline{A} = \{x \,|\, x \notin A\}$.

5. Producto cartesiano: $A \times B = \{(x, y) \,|\, x \in A \wedge y \in B\}$ (el conjunto de todos los pares ordenados de elementos de A y de B).

Obsérvese que la equivalencia entre las expresiones de conjuntos y los predicados correspondientes permite deducir las igualdades $\overline{\overline{A}} = A$, $A - B = A \cap \overline{B}$, $\overline{A \cap B} = \overline{A} \cup \overline{B}$.

Dados dos conjuntos A y B diremos que A *está incluído en* B (o A es un subconjunto de B), representado con $A \subseteq B$, si todo elemento de A pertenece a B, es decir, si es cierta la proposición

$$\forall x \; x \in A \Rightarrow x \in B$$

Para demostrar una igualdad de conjuntos $A = B$ demostraremos $A \subseteq B$ y $B \subseteq A$.
Representaremos con $A \not\subseteq B$ el hecho de que A no está incluído en B ($\neg(A \subseteq B)$). Cuando se cumple que $A \subseteq B$ pero $B \not\subseteq A$, diremos que A está *estrictamente incluído* en B, y lo representaremos con $A \subsetneq B$.
Representaremos por $\|A\|$ el cardinal de un conjunto A (en el caso de conjuntos finitos, el número de elementos que lo componen).
Dado un conjunto A cualquiera, el conjunto de todos los subconjuntos de A se llama *conjunto de las partes de* A y se denota $\mathcal{P}(A)$. Si $\|A\| = n$ entonces $\|\mathcal{P}(A)\| = 2^n$.

1.1.5. Lenguajes

Hacemos un repaso rápido de los primeros conceptos sobre lenguajes.

- Un *alfabeto* es un conjunto finito no vacío. Sus elementos se llaman *símbolos*.
- Una *palabra* sobre un alfabeto es una secuencia finita de símbolos. La secuencia que no contiene ningún símbolo se llama *palabra vacía* y se representa con λ.
- Un *lenguaje* sobre un alfabeto es un conjunto de palabras.
- Dado un alfabeto Σ, Σ^* es el lenguaje de todas las palabras sobre Σ.
- La *longitud* de una palabra x es el número de símbolos que tiene, denotado como $|x|$.
- Dado un alfabeto Σ y $n \in \mathbb{N}$, Σ^n es el conjunto de las palabras de longitud n, y $\Sigma^{\leq n}$ el conjunto de las palabras de longitud menor o igual a n.

Por ejemplo, si $\Sigma = \{0, 1\}$, $\Sigma^0 = \{\lambda\}$, $\Sigma^2 = \{00, 01, 10, 11\}$.
La siguiente proposición nos da el número de palabras de cada longitud y se puede demostrar fácilmente por inducción.

Proposición 1.1 Dado un alfabeto Σ *y un* $n \in \mathbb{N}$,

$$\|\Sigma^n\| = \|\Sigma\|^n$$

$$\|\Sigma^{\leq n}\| = \frac{\|\Sigma\|^{n+1}-1}{\|\Sigma\|-1}$$

En esta asignatura trataremos a menudo con conjuntos de lenguajes, que llamaremos *clases*.

1.1.6. Funciones

Nota muy importante: En esta asignatura utilizaremos la palabra *función* EXCLUSIVAMENTE con el significado matemático que se explica a continuación. Para evitar confusiones *nunca* la utilizaremos para denotar procedimientos o programas con un sólo parámetro de salida, que en muchos lenguajes de programación reciben también este nombre.
Dados dos conjuntos A y B, una función $f : A \to B$ es, informalmente, una forma de asociar a $x \in A$ un elemento de B que llamamos $f(x)$. Nos referiremos siempre a *funciones parciales*. Es decir, una función $f : A \to B$ no tiene que estar definida para todos los elementos de A.

Ejemplo 1.2 Sea $f : \mathbb{N} \to \mathbb{N}$ la función definida como: $f(2n) = n$, o equivalentemente,

$$f(x) = \begin{cases} x/2 & \text{si } x \text{ es par} \\ \text{indefinido} & \text{en otro caso} \end{cases}$$

Definición 1.3 El *dominio* de una función $f : A \to B$ es el conjunto $\mathrm{Dom}(f)$ definido como sigue

$$\mathrm{Dom}(f) = \{x \mid f(x) \text{ está definida } \}$$

Definición 1.4 El *rango* de una función f, también llamado *imagen* de f, es el conjunto $\mathrm{Im}(f)$ (también $\mathrm{Rang}(f)$)

$$\mathrm{Im}(f) = \{f(x) \mid x \in \mathrm{Dom}(f)\}$$

Así pues, $\mathrm{Im}(f)$ es el conjunto de las imágenes de la función f. Si $f : A \to B$ entonces $\mathrm{Dom}(f) \subseteq A$, $\mathrm{Im}(f) \subseteq B$.
Como convenio de notación, dadas dos funciones f y g, sólo escribiremos $f(x) = g(y)$ cuando $x \in \mathrm{Dom}(f)$ e $y \in \mathrm{Dom}(g)$, es decir, si $f(x)$ y $g(y)$ están ambas indefinidas diremos que $f(x) \neq g(y)$.
Existe una función de dominio vacío, que llamaremos *la función vacía*, y que está indefinida en todos los puntos.

Ejemplo 1.5 La función $f : \mathbb{N} \to \mathbb{N}$ que devuelve la raiz cuadrada exacta de un número la podemos definir como:

$$f(x) = \begin{cases} \sqrt{x} & \text{si } x \text{ es un cuadrado perfecto} \\ \text{indefinido} & \text{en otro caso} \end{cases}$$

En este caso $\mathrm{Dom}(f) = \{x \mid x \text{ es un cuadrado perfecto}\}$ y $\mathrm{Im}(f) = \mathbb{N}$.

Definición 1.6 f es una *función inyectiva* si se cumple que para cada $x, y \in \mathrm{Dom}(f)$ con $x \neq y$, $f(x) \neq f(y)$.

Es importante notar que la definición de inyectividad se refiere sólo a los puntos donde está definida la función. Por ejemplo la función vacía es trivialmente inyectiva.

Definición 1.7 Si f es una función inyectiva $f : A \to B$, definimos la *inversa* de f, $f^{-1} : A \to B$, de la siguiente forma:

$$\text{para cada } z \in \mathrm{Im}(f) \quad f^{-1}(z) = x \text{ tal que } f(x) = z.$$

Notar que $\mathrm{Dom}(f^{-1}) = \mathrm{Im}(f)$.

Definición 1.8 $f : A \to B$ es *suprayectiva* si $\mathrm{Im}(f) = B$.

Definición 1.9 $f : A \to B$ es *total* si $\mathrm{Dom}(f) = A$.

Definición 1.10 $f : A \to B$ es una *biyección* si f es total, inyectiva y suprayectiva.

Definición 1.11 Dadas dos funciones $f, g : A \to B$, se dice que f *extiende a* g si

1. $\mathrm{Dom}(g) \subseteq \mathrm{Dom}(f)$,

2. $\forall x \in \mathrm{Dom}(g), f(x) = g(x)$.

Definición 1.12 Dada $f : A \to B$ y $C \subseteq A$, la *restricción de f a C*, f/C, es la función

$$f/C : C \to B$$

de dominio $\mathrm{Dom}(f/C) = C \cap \mathrm{Dom}(f)$ y definida como $f/C(x) = f(x)$ para $x \in C \cap \mathrm{Dom}(f)$.

Ejemplo 1.13 Sea f la función definida en el ejemplo *1.5*. La función

$$g(x) = \begin{cases} \sqrt{x} & \text{si } x \text{ es un cuadrado perfecto} \\ 0 & \text{en otro caso} \end{cases}$$

es una extensión total de f.

Definición 1.14 Dadas $f : A \to B$ y $g : B \to C$ tales que $\mathrm{Im}(f) \subseteq \mathrm{Dom}(g)$, la *composición de f y g*, $g \circ f$, es la función

$$g \circ f : A \to C$$

definida como

$$g \circ f(x) = g(f(x))$$

para $x \in \mathrm{Dom}(f)$.

Definición 1.15 Dado un conjunto A, su *función característica* es:

$$\chi_A(x) = \begin{cases} 1 & \text{si } x \in A \\ 0 & \text{si } x \notin A \end{cases}$$

1.2. Numerabilidad

Una definición intuitiva de cardinal de un conjunto es el número de elementos que tiene. Si suponemos que número quiere decir número natural, entonces no podemos hablar de cardinal de conjuntos como $\mathbb{N}$. Si admitimos infinito como posible número de elementos, entonces tanto $\mathbb{N}$ como $\mathbb{R}$ tienen infinitos elementos. Sin embargo, nos gustaría poder expresar que hay más reales que naturales. En la búsqueda de una definición de cardinal, comparamos el conjunto de los pares y el de los impares:

pares $0, 2, 4, 6, \ldots, 2n, \ldots$
impares $1, 3, 5, 7, \ldots, 2n-1, \ldots$

Parece que hay el mismo número de pares que de impares, porque tenemos una biyección entre los dos conjuntos. Esta es exactamente la definición de cardinal.

Definición 1.16 Dos conjuntos A y B *tienen el mismo cardinal* si existe una biyección $f : A \to B$.

Nota: Es equivalente decir que existe una inyección total de A en B y otra de B en A.
Nota: Si A tiene el mismo cardinal que B y B tiene el mismo cardinal que C, entonces A tiene el mismo cardinal que C.
En el caso finito esta definición se corresponde con lo que decíamos al principio de este sección: dos conjuntos de tres elementos $\{A, B, C\}$, $\{X, Y, Z\}$ tienen el mismo cardinal, mientras que $\{A, B, C, D\}$, $\{X, Y, Z\}$ no lo tienen. En el caso infinito esto se corresponde con la idea de que hay tantos pares como naturales

0 1 2 3 4 ... $\quad f(n) = 2n$
0 2 4 6 8 ...

hay tantos naturales como potencias de dos

0 1 2 3 4 ... $\quad f(n) = 2^n$
1 2 4 8 16 ...

Por supuesto, con cardinales infinitos no se puede operar como con finitos:
¿Cuánto vale $\infty - \infty$?

Depende	$\mathbb{N} -$ pares $=$ impares	(cardinal infinito)
	$\mathbb{N} - \mathbb{N} = \emptyset$	(cardinal 0)
	$\mathbb{N} - \{1, 2, 3, 4, \ldots\} = \{0\}$	(cardinal 1)

Esto nos lleva a la observación de que los naturales tienen muchos subconjuntos con el mismo cardinal que $\mathbb{N}$. Esto sólo puede pasar para conjuntos infinitos, y de hecho podríamos definir "A es un conjunto infinito si existe $B \subseteq A$ con $B \neq A$ y B tiene el mismo cardinal que A".
A partir de ahora trataremos los infinitos más pequeños, llamados numerables.

Definición 1.17 A es *numerable* si A es finito ó A tiene el mismo cardinal que $\mathbb{N}$.

Es decir, si A es numerable infinito tenemos una biyección $f : \mathbb{N} \to A$, lo que nos permite enumerar los elementos de A y podernos referir a $f(0)$ como "ceroésimo elemento de A", a $f(1)$ como "primer elemento de A", ..., $f(x)$ como "x-ésimo elemento de A".
Otra forma intuitiva de verlo es: si A es numerable infinito tenemos una biyección $g : A \to \mathbb{N}$, lo que nos permite dar a cada elemento $x \in A$ su número de orden dentro de A, x es el elemento $g(x)$-ésimo de A.
Veamos algunas propiedades básicas de los numerables (algunas de las demostraciones se dejan como ejercicios):

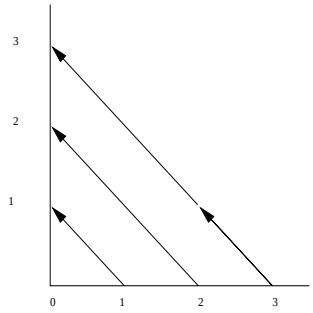
Propiedad 1.18
- $\mathbb{N}$ *es numerable.*
- *Si* $A \subseteq B$ *y* B *es numerable, entonces* A *es numerable.*

Propiedad 1.19
- *Si existe una inyección total* $f : A \to \mathbb{N}$*, entonces* A *es numerable.*
- *Si existe una función suprayectiva* $f : \mathbb{N} \to A$*, entonces* A *es numerable.*

Dem. Para la primera parte, si definimos $g : A \to \mathrm{Im}(f)$ como $g(x) = f(x)\ \forall x \in \mathrm{Dom}(f)$, entonces g es total e inyectiva, por serlo f, y además suprayectiva, por estar definida de A en $\mathrm{Im}(f)$, luego g es una biyección. Por tanto A tiene el mismo cardinal que $\mathrm{Im}(f)$.
Como $\mathrm{Im}(f) \subseteq \mathbb{N}$ y $\mathbb{N}$ es numerable, por la propiedad anterior $\mathrm{Im}(f)$ es numerable y por tanto también lo es A.
La segunda parte se demuestra de manera análoga. ∎

Ejemplos 1.20 de conjuntos numerables.

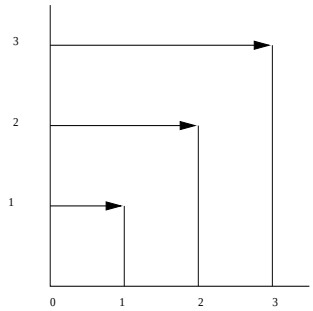
- $\mathbb{N} \times \mathbb{N}$ Definimos $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ que corresponde al siguiente recorrido de los puntos de $\mathbb{N} \times \mathbb{N}$



es decir, recorrido primero por diagonales: primero los puntos que suman 0, luego los que suman 1, etc.

$$f(n,m) = \frac{(1+n+m)(n+m)}{2} + m$$

Otros recorridos posibles:



- $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$

  Sea f biyección de $\mathbb{N} \times \mathbb{N}$ en $\mathbb{N}$. Entonces $g : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ definida como

  $$g(x,y,z) = f(f(x,y),z)$$

  es biyección:

  - inyectiva. Si $(x,y,z) \neq (x',y',z')$:
    si $(x,y) \neq (x',y')$ entonces $f(x,y) \neq f(x',y')$ y por tanto $f(f(x,y),z) \neq f(f(x',y'),z')$
    si $(x,y) = (x',y')$ entonces $z \neq z'$ (ya que $(x,y,z) \neq (x',y',z')$).
    Por tanto $f(f(x,y),z) \neq f(f(x,y),z')$.

- total: por serlo f.
- suprayectiva:

$$\mathrm{Im}(g) = \mathrm{Im}(f/_{\mathrm{Im}(f)\times\mathbb{N}}) = \mathrm{Im}(f/_{\mathbb{N}\times\mathbb{N}}) = \mathrm{Im}(f).$$

$\mathrm{Im}(f) = \mathbb{N}$ (por ser f suprayectiva).

- Z

$$f(x) = \begin{cases} 2x & \text{si } x \geq 0 \\ -2x-1 & \text{si } x < 0 \end{cases}$$

f es biyección de Z en $\mathbb{N}$. (Basta ver que los positivos van a los pares y los negativos a los impares).

En general, tenemos la siguiente propiedad para productos cartesianos:

Propiedad 1.21 Si A y B son numerables, entonces $A \times B$ es numerable.

- Q

  Como Z y $\mathbb{N}$ son numerables, también lo es $Z \times \mathbb{N}$ (propiedad *1.21*).

  Sea f la función total:

$$\begin{array}{rccl} f: & Q & \to & Z \times \mathbb{N} \\ & q & \mapsto & (n,m) \quad \text{con } \frac{n}{m} \text{ fracción irreducible de } q \end{array}$$

  f es inyectiva, ya que una fracción irreducible representa un único número racional. Por la propiedad *1.19*, Q es numerable.

- Dado Σ un alfabeto, Σ^* es el conjunto de todas las palabras sobre Σ.

  No podemos enumerar Σ^* por orden alfabético (ej: $\Sigma = \{0,1\}$, $\lambda, 0, 00, 000, \ldots$) ya que hay infinitas palabras empezando por la primera letra del alfabeto, y nunca llegaríamos a las que empiezan por la segunda.

  La forma de hacer una biyección de $\mathbb{N}$ en Σ^* es enumerar las palabras por orden lexicográfico por longitudes, es decir, por longitudes, y dentro de cada longitud por orden alfabético.

$$\begin{array}{rccl} f: & \Sigma^* & \to & \mathbb{N} \\ & w & \mapsto & \frac{\|\Sigma\|^{|w|}-1}{\|\Sigma\|-1} - 1 + \quad \text{“lugar de } w \text{ por orden alfabético en longitud } |w|\text{”} \end{array}$$

Nota: "Lexicográfico por longitudes" lo abreviaremos como lexicográfico.

- En capítulos sucesivos fijaremos un lenguaje de programación de alto nivel. Cada programa podemos codificarlo como una cadena de caracteres, es decir, una palabra sobre un alfabeto finito Σ (el alfabeto de los caracteres admisibles). Por tanto podemos establecer una biyección del conjunto de todos los programas en un subconjunto de Σ^*. Por esta razón *hay una cantidad numerable de programas*.

1.3. Diagonalización

En esta sección vamos a demostrar que algunos conjuntos son no numerables usando una técnica de Cantor llamada diagonalización. Esta técnica consiste en, dado un conjunto numerable A, construir un elemento $x \notin A$ por etapas.

Lema 1.22 Sea A un conjunto numerable, $A \subseteq [0,1]$ (donde $[0,1]$ es el intervalo de los números reales entre 0 y 1). Entonces existe $x \in [0,1]$ tal que $x \notin A$.

Dem. Por definición de numerable, existe una biyección $f : \mathbb{N} \to A$. Dado $n \in \mathbb{N}$, denoto $f(n)$ como r_n. Entonces

$$\mathrm{Im}(f) = A = \{r_0, r_1, r_2, r_3, \ldots, r_n, \ldots\}$$

Consideremos a los números de A escritos en binario. Dados $i, j \in \mathbb{N}$ denotamos como $r_i[j]$ al $(j+1)$-ésimo bit de la representación en binario de r_i, es decir, si

$$r_i = 0,00101 \qquad \begin{array}{l} r_i[2] = 1 \\ r_i[0] = 0 \end{array}$$

Vamos a construir $x \in [0,1]$ tal que $\forall i \in \mathbb{N}$, $x \neq r_i$ como sigue

$$x[i] = \begin{cases} 1 & \text{si } r_i[i] = 0 \\ 0 & \text{si } r_i[i] = 1 \end{cases}$$

de esta forma $\forall i\ x[i] \neq r_i[i]$, y por tanto $\forall i\ x \neq r_i$.

(Esto es lo que se llama diagonalización, construir x que cumpla

$$\begin{array}{ll} x \neq r_0 & \text{usando } x[0] \\ x \neq r_1 & \text{usando } x[1] \\ \ldots & \ldots \\ x \neq r_n & \text{usando } x[n] \\ \ldots & \ldots \end{array}$$

es decir, cumplir una lista de objetivos $x \neq r_0$, $x \neq r_1$, ..., $x \neq r_n$, ... de forma constructiva).
Por construcción $x \notin \{r_0, r_1, \ldots\} = A$.
Pero x es la representación en binario de un número entre 0 y 1, luego $x \in [0, 1]$. ∎

Nota bene. Hay números (los decimales periódicos) que tienen doble representación en binario.

$$0,0\hat{1} = 0,1 \quad 0,010\hat{1} = 0,11 \quad \text{etc.}$$

Pero podemos considerar el conjunto de todas las representaciones binarias de los elementos de A en lugar de A en la demostración anterior. (Si A es numerable, también lo es el conjunto de todas sus representaciones en binario, que son como máximo dos por cada número).

Teorema 1.23 $[0, 1]$ *no es numerable.*

Dem. Reducción al absurdo. Supongamos que $[0, 1]$ es numerable, entonces por el lema anterior existe $x \in [0, 1]$ tal que $x \notin [0, 1]$. Esto es una contradicción, luego $[0, 1]$ no es numerable. ∎

Teorema 1.24 $\mathbb{R}$ *no es numerable.*

Dem. Como $[0, 1] \subseteq \mathbb{R}$ no es numerable, por la propiedad *1.18*, $\mathbb{R}$ no es numerable. ∎

Teorema 1.25 El conjunto de las funciones totales de $\mathbb{N}$ *en* $\mathbb{N}$ *no es numerable.*

Dem. Vamos a diagonalizar en $\mathcal{S}$, el conjunto de todas las funciones totales de $\mathbb{N}$ en $\mathbb{N}$.

Lema 1.26 Sea A un conjunto numerable, $A \subseteq \mathcal{S}$. Entonces existe $x \in \mathcal{S}$ tal que $x \notin A$.

Dem. Por definición de numerable, existe una biyección $F : \mathbb{N} \to A$. Dado $n \in \mathbb{N}$, denoto $F(n)$ como f_n. Entonces

$$\mathrm{Im}(F) = A = \{f_0, f_1, f_2, \ldots, f_n, \ldots\}$$

Voy a construir una función total que no está en A. Sea

$$\begin{array}{rl} g: & \mathbb{N} \to \mathbb{N} \\ & n \mapsto f_n(n) + 1 \end{array}$$

g es total porque todas las f_n lo son. Para todo n, $g(n) \neq f_n(n)$, por tanto para todo n, $g \neq f_n$. (Consigo

$$\begin{array}{ll} g \neq f_0 & \text{usando } g(0) \\ g \neq f_1 & \text{usando } g(1) \\ \ldots & \ldots \\ g \neq f_n & \text{usando } g(n) \\ \ldots & \ldots \end{array}$$

Luego $g \notin \{f_0, f_1, f_2, \ldots, f_n, \ldots\} = A$. ■ (lema)
El teorema se demuestra por reducción al absurdo como en el caso de $[0, 1]$. ■

Teorema 1.27 Dado un alfabeto Σ, el conjunto de las funciones totales de Σ^* en Σ^* no es numerable.

Dem. Análoga a la anterior. ■

Corolario 1.28 Dado un alfabeto Σ, el conjunto de las funciones de Σ^* en Σ^* no es numerable.

Hemos visto que el conjunto de todos los programas sí es numerable. En el capítulo siguiente formalizaremos la idea de que cada programa calcula una función (a cada entrada le hace corresponder una salida). De momento ya sabemos un hecho importante:

Corolario 1.29 Existen funciones que no se pueden calcular con ningún programa.

EJERCICIOS

1.1. Sea $f : Q \times Q \to Q$ una función definida por $f(a, b) = a/b$. ¿Es f total? ¿Cuál es el dominio de f? ¿Cuál es su imagen?

1.2. Demostrar la Proposición *1.1* usando inducción.

1.3. Demostrar la Propiedad *1.18*: si $A \subseteq B$ y B es numerable, entonces A es también numerable (utilizar la definición original de numerable, esta es una propiedad básica de la que se deducen otras definiciones equivalentes).

1.4. Dados dos conjuntos numerables A y B, demostrar que $A \times B$, el producto cartesiano de A y B, es numerable (Propiedad *1.21*).

1.5. Demostrar que el conjunto de los subconjuntos finitos de $\mathbb{N}$ es numerable.

1.6. Demostrar que el conjunto de las funciones de $\mathbb{N}$ en $\mathbb{N}$ con dominio finito es numerable.

1.7. Estudiar la demostración vista en clase de que R, el conjunto de los números reales, no es numerable. ¿Por qué no sirve una demostración similar para demostrar que Q, el conjunto de lo números racionales, no es numerable?

1.8. Demostrar que el conjunto de los subconjuntos de $\mathbb{N}$ no es numerable.

1.9. Demostrar que el conjunto de los lenguajes sobre un alfabeto Σ no es numerable.

1.10. Demostrar que el conjunto de las funciones totales de $\mathbb{N}$ en $\{0, 1\}$ no es numerable.

1.11. Demostrar que el conjunto de las funciones de $\mathbb{N}$ en $\mathbb{N}$ con imagen finita no es numerable.

1.12. Demostrar que si $\{A_1, A_2, A_3, \ldots\}$ es una colección numerable de conjuntos numerables disjuntos, entonces

$$\bigcup_{i \in \mathbb{N}} A_i$$

es un conjunto numerable.

Capítulo 2

Problemas y datos. Un modelo abstracto de cálculo: la máquina de registros

Referencia: Capítulo 1 de [Cu80].

Comenzaremos este tema modelizando problemas mediante objetos formales (lenguajes o funciones) para un tratamiento formal posterior. Después definiremos el modelo de cálculo que utilizaremos el resto del curso, la máquina de registros con sus programas.

2.1. Problemas, lenguajes y funciones

2.1.1. Problemas decisionales y funcionales

Quizá la manera más fácil de analizar las diferentes componentes de la definición de un problema es mediante ejemplos. Empezaremos con un problema clásico de la teoría de grafos.
(Recordemos que un grafo dirigido $G = (V, A)$ es un conjunto de vértices V y un conjunto de aristas $A \subseteq V \times V$. Para un grafo de n vértices tomaremos siempre como conjunto de vértices $V = \{1, 2, 3, \ldots, n\}$.)

Ejemplo 2.1 `Accesibilidad de grafos (GAP)`: Dado un grafo dirigido $G = (V, A)$ y dos vértices $u, v \in V$, determinar si existe un camino de u a v en G.

En el enunciado del problema se pregunta si se satisface o no una propiedad. A este tipo de problemas les denominaremos *problemas decisionales*. En un problema decisional se define un conjunto de datos, los *datos de entrada* y una propiedad.
Un segundo tipo de enunciado de problema pide la construcción de un objeto, o el cálculo de un valor.

Ejemplo 2.2 `true gates (TG)`: Dado un circuito booleano, y una asignación de valores a sus entradas, calcular el número de puertas que evalúan a uno.

Ejemplo 2.3 `Camino (PATH)`: Dado un grafo dirigido $G = (V, A)$ y dos vértices $u, v \in V$, calcular un camino de u a v en G.
Este problema no está especificado completamente, pues puede haber más de un camino. Podemos decir, por ejemplo, "calcular el primer camino, por orden alfabético".

A los problemas en cuyo enunciado se pide la construcción de un objeto (en caso de que exista) o el cálculo de un valor, les denominaremos *problemas funcionales*.
Un *problema decisional* se define a partir de un conjunto de datos E, que denominamos conjunto de entradas, y una propiedad R. Se expresa como: Dado $x \in E$, ¿se satisface $R(x)$?
Un *problema funcional* se define a partir de dos conjuntos de datos E, conjunto de entradas, y S, conjunto de salidas, junto con una propiedad Q y se expresa como: Dado $x \in E$, calcular $y \in S$ para el que se cumple $Q(x, y)$. Para una entrada x, el número de objetos y que verifican $Q(x, y)$ es 1 ó 0.
En nuestros ejemplos `GAP` y `PATH`, el conjunto de entradas E es el conjunto

$$\{(G, u, v) \mid G \text{ es un grafo dirigido y } u \text{ y } v \text{ son vértices de } G\}$$

2.1.2. Representación de datos, tamaño

Analicemos un poco más a fondo los conjuntos de datos. Nos interesan los problemas en los que cada dato es representable mediante una estructura de datos finita. En los problemas que hemos planteado hasta

ahora, se cumple este requisito, es fácil diseñar una estructura de datos para las entradas. Por supuesto que la estructura nunca será única, pero diseñarla es el primer paso en el diseño de un algoritmo.
Si a alto nivel requerimos que los conjuntos de datos sean representables mediante estructuras de datos, a bajo nivel queremos representarlos mediante una cadena de caracteres. A este proceso lo llamaremos *codificación*. A través de la codificación podemos asociar a cada dato un *tamaño*, la longitud de la cadena de caracteres que lo representa.
Dado un conjunto de datos D utilizaremos la siguiente notación:

- x es un elemento de D representado mediante una estructura de datos.
- $\langle x \rangle$ es la cadena de caracteres que codifica a x,
- $|x|$, el tamaño de x, es la longitud de $\langle x \rangle$.

Por ejemplo la representación usual de un número natural es en binario, con lo que si la tomamos como la codificación sobre el alfabeto $\Sigma = \{0, 1\}$ tenemos que para un número natural x, $|x| = \log x + 1$. Como $\langle x \rangle$ es x en binario en este caso no haremos distinción entre $\langle x \rangle$ y x.
Nota: Denotamos por log el logaritmo en base 2 por defecto, es decir $\lfloor \log_2 \rfloor$, con valor mínimo 1.
Para representar un grafo $G = (V, A)$ con vértices $V = \{1, \ldots, n\}$ podemos utilizar una estructura de datos consistente en una matriz booleana M definida como sigue:

$$M(i,j) = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{si } (i,j) \notin A \end{cases}$$

A bajo nivel la codificación de esta matriz puede ser escribir la matriz por filas, obteniendo así una cadena sobre el alfabeto $\{0, 1\}$. El tamaño de un grafo G será en este caso el cuadrado del número de vértices.
En general, a una codificación le pediremos que sea "razonable" queriendo expresar con ello que no debe engordar artificialmente el tamaño de los objetos que codifica. Además deben existir algoritmos eficientes que permitan pasar de la estructura de datos x a su codificación $\langle x \rangle$, reconocer si una cadena codifica una estructura de datos, y pasar de una cadena $\langle x \rangle$ a la estructura de datos que codifica x.

Podemos codificar una estructura de datos formada a partir de tipos de datos elementales a partir de las codificaciones de los mismos.
Por ejemplo la entrada de PATH formada por un grafo dirigido G y dos vértices u, v podemos codificarla a partir de la codificación de G, $X \in \{0,1\}^*$, y las codificaciones en binario de u y v, $Y, Z \in \{0,1\}^*$. Una forma simple de hacer esto es añadir un símbolo nuevo # para separar las componentes, por ejemplo si $X = 001110000$, $u = 1$, $v = 2$ la entrada G, u, v se codificaría como 001110000#1#10 sobre el alfabeto $\{0, 1, \#\}$. Con esta codificación el tamaño de la entrada (G, u, v) es $|G|+|u|+|v|+2$. En el mismo ejemplo, si queremos codificar entradas de la forma G, u, v sobre el alfabeto $\{0, 1\}$, podemos hacerlo a partir de la codificación anterior con tres símbolos, simplemente asociando al símbolo 0 la palabra 00, al 1 la palabra 01 y al # la palabra 11. Con la misma entrada que en el párrafo anterior obtenemos ahora 000001010100000000001101000100. Con esta codificación el tamaño de la entrada (G, u, v) es $2(|G|+|u|+|v|+2)$.
Ejercicio. Buscar otras posibles codificaciones de entradas formadas por varios datos elementales tanto con el alfabeto $\{0, 1\}$ como con alfabetos de más de dos símbolos.

2.1.3. Lenguajes y funciones

Fijado un alfabeto de codificación Σ (muy a menudo $\Sigma = \{0, 1\}$), asociaremos a cada problema decisional un lenguaje y a cada problema funcional una función.
Consideremos un *problema decisional* Π con un conjunto de entradas E, de manera que cada elemento $x \in E$ es representable mediante una cadena de caracteres $\langle x \rangle$. Supongamos que Π está definido mediante la propiedad R, asociaremos al problema Π el lenguaje:

$$L(\Pi) = \{\langle x \rangle \mid R(x)\}$$

Por ejemplo, si tenemos el problema:
`PRIMO`: Dado un número natural n, determinar si n es primo.
Asociaremos a `PRIMO` el lenguaje:

$$L(\texttt{PRIMO}) = \{n \mid n \text{ es un número primo}\}.$$

A cada lenguaje L le asociamos la función característica $\chi_L : \Sigma^* \rightarrow$

$\{0,1\}$ (ya definida en el capítulo 1):

$$\chi_L(x) = \begin{cases} 1 & \text{si } x \in L \\ 0 & \text{si } x \notin L \end{cases}$$

Luego cada problema decisional lo podemos representar mediante un lenguaje o bien mediante una función total de Σ^* en $\{0,1\}$.
Analicemos ahora un *problema funcional*. Supongamos que tenemos un problema funcional Π con conjunto de entradas E, conjunto de salidas S, ambos codificados sobre Σ, y Π definido mediante la propiedad Q. Asociamos a Π la función $f_\Pi : \Sigma^* \to \Sigma^*$ definida como:

$$f_\Pi(\langle x \rangle) = \langle y \rangle \quad \text{tal que se cumple } Q(x,y)$$

Para una entrada x sabemos que el número de soluciones (objetos y que verifican $Q(x,y)$) es 1 ó 0, así que la función f_Π puede no estar definida en algunos puntos.

2.2. La máquina de registros ó Random Access Machine

En el capítulo 1 hemos visto que existen funciones que ningún programa puede calcular. La siguiente pregunta a tratar es si dentro de estas funciones no calculables hay alguna interesante, de esto se ocupa la teoría de la calculabilidad.
Antes de empezar a tratar lo que se puede o no calcular con un programa, tenemos que fijar qué máquina estamos programando y qué lenguaje de programación usamos. Nos interesa un modelo de máquina y un lenguaje lo más generales posibles, es decir, que resuelvan tantos problemas como el computador *más potente*. De esta forma, si probamos que un problema no se puede resolver en nuestro modelo, no se podrá resolver en ningún computador.
Nuestro modelo va a ser un *modelo abstracto* o ideal, en el sentido de no real. Se trata de la "Random Access Machine" (RAM), que es una máquina de registros dotada de un número ilimitado de registros. Cada registro puede almacenar un número entero de cualquier tamaño. (Es la no limitación en el número de registros y en el tamaño de los mismos lo

que hace que la RAM sea un modelo no real, ya que es un modelo de memoria no acotada.)
Nosotros programaremos la RAM usando un lenguaje de alto nivel que representamos en una notación algorítmica convencional. Dispondremos de constantes y variables de tipos elementales (naturales, enteros, reales, booleanos y cadenas de caracteres) así como tipos no elementales de los que nos interesará a menudo la codificación a bajo nivel, de cara a medir el tamaño de los datos. Tendremos las instrucciones de asignación, condicional y bucles, así como las operaciones básicas de los tipos elementales.
Debemos recordar que disponemos de una cantidad de memoria ilimitada, es decir, podemos usar cualquier número de variables y no hay límite en el tamaño de los datos que estas variables pueden almacenar. Podemos utilizar la codificación de datos descrita en 2.1.2, y asumir que todos los datos de entrada de un programase codifican como una única cadena de caracteres. Es por ello que, siempre que nos convenga, asumimos que todos nuestros programas tienen un único parámetro de entrada, de tipo cadena de caracteres. Este parámetro corresponde a la codificación de la entrada según se haya fijado. De esta forma, el primer paso del programa será decodificar la entrada. Por ejemplo los dos programas siguientes son esencialmente el mismo:

```
Leer W: cadena
 decodificacion(W, X, Y);
 %% El procedimiento decodificacion decodifica W=⟨X, Y⟩
 Z:=X+Y;
 Devuelve Z;
```

```
Leer X, Y:natural
 Z:=X+Y;
 Devuelve Z;
```

Los programas tendrán un único parámetro de salida de tipo cadena de caracteres que corresponderá a la codificación de la salida según se haya fijado.

2.2.1. Codificación de programas

Algunos de los problemas que nos interesan tienen como entrada o parte de la entrada (o de la salida) un programa. Para resolverlos algorítmicamente tendremos que representar los programas mediante una estructura de datos y a bajo nivel establecer su codificación.
Para definir el tipo de datos programa simplemente utilizaremos que cada programa en nuestro lenguaje de alto nivel se puede escribir como una cadena de caracteres. Al escribir cada caracter en código ASCII tenemos la codificación del programa sobre $\{0,1\}$.
Dada $w \in \{0,1\}^*$ usaremos la notación P_w para el programa con codificación w, aunque a menudo identificaremos directamente w con el programa de codificación w. Aunque a menudo identificaremos cadenas y programas, usando w para el programa P_w y siendo muy laxos en la tipificación de datos.
También asociaremos a cada $i \in \mathbb{N}$ un programa P_i. P_i es el programa que ocupa la posición i-ésima entre todos los programas por orden lexicográfico (es decir, por longitudes y dentro de cada longitud por orden alfabético). También identificaremos directamente i con el programa P_i. Notemos que de esta forma P_i está definido para todo $i \in \mathbb{N}$ y para todo programa p existe un $i \in \mathbb{N}$ tal que $\mathrm{P}_i = p$. Además existe un programa que a partir de $i \in \mathbb{N}$ calcula una codificación de P_i, y existe otro programa que a partir de una codificación $w \in \{0,1\}^*$ calcula $i \in \mathbb{N}$ tal que $\mathrm{P}_i = \mathrm{P}_w$.
Ejercicio. Escribir un programa que a partir de $i \in \mathbb{N}$ calcula una codificación de P_i. Escribir un programa que a partir de una codificación $w \in \{0,1\}^*$ calcula $i \in \mathbb{N}$ tal que $\mathrm{P}_i = \mathrm{P}_w$.

2.2.2. Notación para programas

Dado un programa p y una entrada x utilizaremos la siguiente notación:

$p(x)\downarrow$	El programa p con entrada x termina su ejecución y da una salida.
$p(x)\uparrow$	El programa p con entrada x no acaba nunca o bien acaba pero no da salida.
φ_p	Función que calcula el programa p, es decir: $\varphi_p(x) =$ Salida del programa p con entrada x, si $p(x)\downarrow$.

Luego $\mathrm{Dom}(\varphi_p) = \{x \mid p(x)\downarrow\}$.
Nota: Diremos que "$p(x)$ para" (o termina, o acaba) si $p(x)\downarrow$. Si en algún momento queremos expresar el hecho "$p(x)$ acaba pero no da salida" lo indicaremos explícitamente.
Denominamos paso al tiempo de ejecución de una instrucción de alto nivel. Dado $t \in \mathbb{N}$, diremos que "$p(x)\downarrow$ en t pasos" si $p(x)$ para en t pasos o menos. Si en algún momento queremos expresar el hecho "$p(x)\downarrow$ en t pasos exactamente" lo indicaremos explícitamente.

2.2.3. Más sobre programas

Existe un programa intérprete o simulador, simular con el siguiente perfil:

> simular(Q:in programa; X:in cadena; ÉXITO:out booleano; RESULTADO:out cadena);

Dados un programa p y una entrada x:

- Si $p(x)\downarrow$ entonces $\mathrm{simular}(p, x, \text{ÉXITO}, \text{RESULTADO})\downarrow$ y da salida ÉXITO=TRUE, RESULTADO=$\varphi_p(x)$.
- Si $p(x)\uparrow$ entonces $\mathrm{simular}(p, x, \text{ÉXITO}, \text{RESULTADO})\uparrow$.

Este programa nos permite simular un programa como parte de otro programa:

```
Leer Q, X
  ...
  simular(Q, X, ÉXITO, RESULTADO);
   %% El resto sólo se ejecuta si Q(X)⇓
  ...
```

Existe un programa reloj, simularConReloj, que es una versión del intérprete con control de tiempo:

> simular(Q:in programa; X:in cadena; T:in natural; ÉXITO:out booleano; RESULTADO:out cadena);

Dados un programa p, una entrada x y un número natural t, simularConReloj$(p, x, t, \text{ÉXITO}, \text{RESULTADO})$ quiere decir "simular p con entrada x durante tiempo t". Para cualquier p, x, t, simularConReloj$(p, x, t, \text{ÉXITO}, \text{RESULTADO})\downarrow$ y se cumple:

- Si $p(x)\uparrow$ entonces $\forall t$ simularConReloj con entrada (p, x, t, ÉXITO, RESULTADO) da salida ÉXITO=FALSE.
- Si $p(x)\downarrow$ entonces $\exists t_0 \in \mathbb{N}$ tal que
  - si $t \geq t_0$ entonces simularConReloj con entrada (p, x, t, ÉXITO, RESULTADO) da salida ÉXITO=TRUE, RESULTADO= $\varphi_p(x)$
  - si $t < t_0$ entonces simularConReloj con entrada (p, x, t, ÉXITO, RESULTADO) da salida ÉXITO=FALSE.

Por tanto podemos utilizar este programa para simular un programa durante un tiempo:

```
Leer Q, X, T
  ...
  simularConReloj(Q, X, T, ÉXITO, RESULTADO);
   %% esto es simular Q(X) durante T pasos;
  ...
```

Es pues muy diferente utilizar los programas *simular* y *simularConReloj*, ya que el primero puede no parar. Compara la ejecución de estos dos programas, ¿los dos ejecutan el bloque de instrucciones S?

```
Leer Q, X                                  Leer Q, X
T:=30;                                     simular(Q, X, ÉX, RES);
simularConTiempo(Q, X, T, ÉX, RES);        S;
S;                                         ...
...
```

Compara estos otros dos:

```
Leer Q, X                                  Leer Q, X
T:=30;                                     simular(Q, X, ÉX, RES);
```

```
simularConTiempo(Q, X, T, ÉX, RES);   Si ÉX
Si ÉX                                   entonces A
  entonces A                            %% No sirve para nada poner un else.
  else B
```

Ejercicio. Implementar en un lenguaje de programación (por ejemplo ADA) un TAD programa que contenga los procedimientos simular y simularConReloj. Hay que empezar escribiendo dos procedimientos, el primero que devuelvan la configuración inicial (contenido de las variables al inicio de la ejecución del programa), y el segundo que a partir de una configuración (contenido de las variables y número de línea en que se encuentra la ejecución) calcule la siguiente configuración.

2.3. Definición de función calculable

Terminamos el capítulo definiendo lo que entendemos por función calculable.

Definición 2.4 Sea $f : \Sigma^* \to \Sigma^*$ una función. Decimos que f es *calculable* si existe un programa p tal que $f = \varphi_p$, es decir:
si $x \in \mathrm{Dom}(f)$ entonces p con entrada x da salida $f(x)$
si $x \notin \mathrm{Dom}(f)$ entonces $p(x)\uparrow$.

Es importante notar que *las funciones calculables pueden ser parciales*. Si $x \notin \mathrm{Dom}(f)$ entonces el programa que calcula f, con entrada x no para (lo que quiere decir que no termina o no da salida).

Ejemplos 2.5

- El producto de números naturales es una función calculable total.
- La división es una función calculable parcial.
- La función:
$$f(x) = 1 \quad \text{si } x(x)\downarrow$$
es calculable.

El hecho de que una función sea calculable parcial no quiere decir necesariamente que el programa que la calcule funcione "mal":

```
Leer X, Y
  Si Y≠ 0 entonces Devuelve X DIV Y;
```

El programa anterior calcula la división de naturales y termina siempre (pero no siempre da salida).

```
Leer X
  SUMA:=0; SUMANDO:=X; EPSILON:=0,01;
  Mientras que SUMANDO > EPSILON hacer
    SUMA:=SUMA+SUMANDO;
    SUMANDO:=SUMANDO * X;
  Fmq;
  Devuelve SUMA;
```

Este programa calcula una aproximación de $\sum_{n=1}^{\infty} X^n$, si $X < 1$. Si $X \geq 1$ el programa no para.

Nota: Clásicamente se utilizaba el término "función recursiva" en lugar de "función calculable". Nosotros evitaremos el primero porque está cayendo en desuso y porque la palabra recursiva tiene demasiadas connotaciones en informática.

Capítulo 3

Problemas decidibles y semidecidibles

Referencia: Capítulo 7 de [Cu80].

En el capítulo anterior hemos tratado el problema de si una función se puede calcular o no con un algoritmo, definiendo el concepto de función calculable. En este tema trataremos de problemas decisionales, es decir, con sólo dos respuestas posibles. En el capítulo anterior ya hemos identificado problemas decisionales con lenguajes o conjuntos de palabras.
Estudiaremos en este capítulo los conjuntos o problemas decidibles, que corresponden a problemas decisionales resolubles por medio de algoritmos, y los problemas o conjuntos semidecidibles, que representan un concepto menos restrictivo.
Nota: Existe una notación anterior que no utilizaremos, que habla de lenguages "recursivos" y "enumerables recursivamente".

3.1. Definición y primeros ejemplos de conjunto decidible

Definición 3.1 Sea $A \subseteq \Sigma^*$ un conjunto de cadenas o lenguaje. A es *decidible* si existe un programa p que cumple:

1. Para todo x, $p(x)\downarrow$

2. Si $x \in A$, entonces $\varphi_p(x) = 1$.
3. Si $x \notin A$, entonces $\varphi_p(x) = 0$.

Es decir, un programa que resuelve completamente el problema de pertenencia a A.

También se dice problema decidible refiriéndose al siguiente problema correspondiente a un conjunto decidible A:

Dada x, ¿$x \in A$?

Un problema o conjunto *indecidible* es un problema o conjunto que no es decidible.
Conocemos múltiples ejemplos de conjuntos decidibles:

- los lenguajes regulares, que se vieron el curso pasado y que se pueden resolver con programas que usan memoria constante,
- el problema de saber si un número natural es primo, resoluble con un algoritmo que pruebe exhaustivamente todos los posibles divisores.

Vamos a estar interesados en conjuntos relacionados con el comportamiento de los programas, especialmente con si paran o no. Durante este y los próximos capítulos estudiaremos muchos conjuntos de este tipo debido a que son los más sencillos de analizar.

Ejemplos 3.2 Los siguientes conjuntos son decidibles:

$$A = \{p, x, t \mid p(x)\downarrow \text{ en } t \text{ pasos o menos}\}$$

A es decidible, ya que el siguiente programa resuelve A:

```
Leer Q, X, T
 SimularConReloj(Q,X,T,ÉXITO)
 Si ÉXITO
   entonces Devuelve 1
   else Devuelve 0;
```

$$B = \{p, x, z, t \mid \varphi_p(x) = z, \text{ y la computación de } p \text{ con entrada } x \text{ tarda } t \text{ pasos o menos}\}$$

B es decidible, ya que el siguiente programa resuelve B:

```
Leer Q, X, Z, T
 SimularConReloj(Q,X,T,ÉXITO,RESULTADO)
 Si ÉXITO
 entonces
  Si RESULTADO=Z
    entonces Devuelve 1
    else Devuelve 0
  Fsi;
 else Devuelve 0;
```

3.2. El problema de parada

Existe un problema muy interesante para la teoría de la calculabilidad, es el problema de, dados un programa y una entrada, ¿para el programa con esta entrada? Se trata del *problema de parada* o "halting problem", estudiado por Turing en 1936.
El problema de parada se identifica con el conjunto:

$$H = \{p, x \mid p(x)\downarrow\}$$

Este será nuestro primer ejemplo de conjunto no decidible o indecidible, es decir, problema decisional que no resuelve ningún programa.
Para demostrar que H es indecidible estudiaremos primero *el problema diagonal de parada*, K:

$$K = \{p \mid p(p)\downarrow\}$$

Es decir, el conjunto de programas que, tomando como entrada su propia codificación, paran.

Teorema 3.3 K es indecidible.

Dem. Por diagonalización. Para cada conjunto decidible A, construiremos x_0 testigo de que $A \neq K$.
Sea A decidible. Sea pA un programa que resuelve A. Definimos x_0 como el siguiente programa:

```
Leer Z
Simular(pA, Z, ÉXITO, RESULTADO);
Si RESULTADO=0 entonces Devuelve 1;
```

Veamos que $x_0 \in K \Leftrightarrow x_0 \notin A$:

- Si $x_0 \in K$ entonces $x_0(x_0)\downarrow$, luego $\varphi_{x_0}(x_0) = 1$ y $\varphi_{pA}(x_0) = 0$. Por tanto $x_0 \notin A$.
- Si $x_0 \notin K$ entonces $x_0(x_0)\uparrow$, luego $\varphi_{pA}(x_0) = 1$. Por tanto $x_0 \in A$.

Luego $A \neq K$. Como esto es cierto para cualquier A decidible, entonces K no es decidible. ∎

Corolario 3.4 H es indecidible.

Dem. Por reducción al absurdo. Si H fuera decidible, sea pH un programa que resuelve H. El siguiente programa resuelve K:

```
Leer X
  Simular(ph,⟨X,X⟩,ÉXITO,RESULTADO);
  Devuelve RESULTADO;
```

Pero esto es imposible porque K es indecidible, luego pH no puede existir. ∎

3.3. Definición y primeros ejemplos de conjunto semidecidible

Definición 3.5 Un conjunto L es *semidecidible* si existe un programa p que cumple:

1. Si $x \in A$, entonces $\varphi_p(x) = 1$.
2. Si $x \notin A$, entonces $\varphi_p(x) = 0$ ó bien $p(x)\uparrow$

Luego para que un conjunto A sea semidecidible es suficiente que haya un programa que conteste bien en el caso $x \in A$, aunque pueda no contestar (o incluso colgarse) para alguna entrada $x \notin A$.
También se dice problema semidecidible refiriéndose al problema de pertenencia a un conjunto semidecidible.
De las definiciones anteriores se sigue:

Propiedad 3.6 Si un conjunto A es decidible entonces A es semidecidible.

Tenemos pues como ejemplos de semidecidibles todos los decidibles. Veamos algunos otros.

Ejemplo 3.7 El siguiente conjunto es semidecidible:

$$C = \{p, x, z \mid \varphi_p(x) = z\}$$

C cumple la definición de semidecidible, ya que tenemos el siguiente programa:

```
paraC: Leer Q, X, Z
        Simular(Q,X,ÉXITO,RESULTADO)
        Si ÉXITO AND (RESULTADO=Z) entonces Devuelve 1.
```

Este programa no para siempre. Si p, x cumplen que $p(x)\uparrow$, entonces el programa paraC con entrada p, x, z (para cualquier z) no para.

Un ejemplo importante es el problema de parada.

Teorema 3.8 H es semidecidible.

Dem. El siguiente programa demuestra que H es semidecidible:

```
Leer Q, X
 Simular(Q,X,ÉXITO)
 Si ÉXITO entonces Devuelve 1;
```

■

El programa anterior da salida 1 si la entrada está en H, y no da salida o incluso no termina si la entrada no está en H. Intuitivamente este algoritmo resuelve el problema de parada "en el caso positivo" aunque hemos demostrado que no existe ningún programa que lo resuelva completamente. Veremos otros muchos problemas decisionales en el mismo caso, son semidecidible pero no son decidibles.
Como ejercicio se puede ver la propiedad análoga para K:

Teorema 3.9 K es semidecidible.

3.4. Caracterizaciones

Vamos a estudiar a continuación caracterizaciones de los dos conceptos anteriores que utilizarán la función característica y los programas generadores.
Nos centraremos sólo en los conjuntos infinitos, ya que los finitos son casos triviales que trataremos en el siguiente apartado.

Definición 3.10 Un programa p *para siempre* si para cualquier entrada x, $p(x)\downarrow$

Definición 3.11 Un programa p genera un conjunto L si

- p para siempre y
- $L = \{\varphi_p(n) \mid n \in \mathbb{N}\}$.

Es decir, un programa p genera un conjunto L si las salidas del programa son exactamente las cadenas de L.

Teorema 3.12 Dado un conjunto A infinito, son equivalentes:

1. *A es semidecidible.*
2. *La función Π_A es calculable, donde*
$$\Pi_A(x) = 1 \quad \textit{si } x \in A.$$
3. *Existe un programa que genera A.*

4. *Existe un programa p que genera A sin repeticiones (es decir, para todo n, $\varphi_p(n) \notin \{\varphi_p(0), \varphi_p(1), \ldots, \varphi_p(n-1)\}$).*
5. *Existe una función calculable y biyectiva $f : \mathbb{N} \to A$.*
6. *A es el dominio de una función calculable.*
7. *A es el conjunto imagen de una función calculable.*
8. *A es el conjunto imagen de una función calculable total.*

Dem. Demostraremos primero la equivalencia de 1., 2., 6. y 7., después la equivalencia de 3., 4., 5. y 8., y por último 1.⇒3. y 8.⇒7.
1. ⇒ 2. Por definición de semidecidible tenemos un programa p que resuelve A al menos en el caso positivo. A partir de él construímos el siguiente programa que calcula Π_A:

```
Leer X
  Simular(p,X,ÉXITO, RESULTADO);
  Si ÉXITO AND (RESULTADO=1) entonces Devuelve 1;
```

2. ⇒ 6. $A = \mathrm{Dom}(\Pi_A)$.
6. ⇒ 7. Sea f una función calculable que cumple 6. y p un programa que calcula f. Definimos la función g:

$$g(x) = x \quad \text{si } x \in \mathrm{Dom}(f)$$

Tenemos que $\mathrm{Im}(g) = \mathrm{Dom}(f) = A$. Además g es calculable ya que la calcula el siguiente programa:

```
Leer X
  Simular(p,X,ÉXITO);
  Si ÉXITO entonces Devuelve X;
```

7. ⇒ 1. Sea f una función calculable tal que $A = \mathrm{Im}(f)$, sea p un programa que calcula f. El siguiente programa demuestra que A es semidecidible. (Como realiza varias simulaciones de p con distintas entradas hay que hacer simulaciones controladas.)

```
Leer X
  T:=1;
```

```
TERMINADO:=FALSE;
Mientras que NOT TERMINADO hacer
  Para Y:=0 hasta T hacer
    SimularConTiempo(p,Y,T,ÉXITO,RESULTADO);
    Si ÉXITO AND (RESULTADO=X) entonces TERMINADO:=TRUE;
  Fpara;
  T:=T+1;
Fmq;
Devuelve 1;
```

3. $\Rightarrow$ 4. Sea p un programa que genera A. Vamos a eliminar las posibles repeticiones con el siguiente programa que con entrada n da como salida la n-ésima salida diferente que produce p.

```
Leer N
NDISTINTOS:=0;
DISTINTOS:=vacio;
M:=0;
Mientras que (NDISTINTOS < N) hacer
  Simular(p,M,ÉXITO,RESULTADO);
  Si NOT esta(RESULTADO,DISTINTOS)
    entonces
      NDISTINTOS:=NDISTINTOS+1;
      añadir(DISTINTOS, RESULTADO);
  Fsi;
  M:=M+1;
Fmq;
Devuelve RESULTADO;
```

4. $\Rightarrow$ 5. Si p genera A sin repeticiones, φ_p es una función calculable, total e inyectiva, $\mathrm{Im}(\varphi_p) = A$. Luego φ_p es la biyección buscada de $\mathbb{N}$ en A.
5. $\Rightarrow$ 8. En 5., por ser f biyectiva es total, y $\mathrm{Im}(f) = A$.
8. $\Rightarrow$ 3. Sea $f : \mathbb{N} \to \Sigma^*$ una función que cumple 8., sea p un programa que calcula f. Por 8. p para siempre y $A = \mathrm{Im}(f) = \{\varphi_p(n) \mid n \in \mathbb{N}\}$.
1. $\Rightarrow$ 3. Por definición de semidecidible tenemos un programa p que resuelve A al menos en el caso positivo. El siguiente programa genera A:

```
Leer N
 NGENERADOS:=0;
 T:=1;
 Mientras que (NGENERADOS < N) hacer
  Para Y:=0 hasta T hacer
   SimularConTiempo(p,Y,T,ÉXITO,RESULTADO);
   Si (ÉXITO) AND (RESULTADO=1)
    entonces
     NGENERADOS:=NGENERADOS+1;
     Si NGENERADOS=N entonces ULTIMO:=Y;
   Fsi;
  Fpara;
  T:=T+1;
 Fmq;
 Devuelve ULTIMO;
```

8. $\Rightarrow$ 7. Inmediato. ∎

Teorema 3.13 Dado un conjunto A infinito, son equivalentes:

1. *A es decidible.*
2. *La función χ_A es calculable.*
3. *Existe un programa p que genera A en orden y sin repeticiones (es decir, para todo $n > 0$, $\varphi_p(n-1) < \varphi_p(n)$).*

Dem. 1. $\Rightarrow$ 2. Al ser A decidible tenemos un programa p que resuelve A. Recordemos que la función χ_A está definida como:

$$\chi_A(x) = \begin{cases} 1 & \text{si } x \in A \\ 0 & \text{si } x \notin A \end{cases}$$

Luego el mismo programa p calcula χ_A.
2. $\Rightarrow$ 3.
Sea p un programa que calcula χ_A. Al ser una función total el programa para siempre y por tanto podemos ir enumerando en orden las entradas cuya salida es 1:

```
Leer N
  NELEMENTOS:=0;
  M:=0;
  Mientras que (NELEMENTOS < N) hacer
    Simular(p,M,ÉXITO,RESULTADO);
    Si RESULTADO=1
      entonces
        NELEMENTOS:=NELEMENTOS+1;
    Fsi;
    M:=M+1;
  Fmq;
  Devuelve RESULTADO;
```

3. $\Rightarrow$ 1. Sea p un programa que genera A en orden y sin repeticiones. Para resolver A sólo tenemos que ir generando elementos hasta que sepamos que está el que buscamos o que ya no va a aparecer:

```
Leer X
  Simular(p,0,ÉXITO,RESULTADO);
  N:=1;
  Mientras que (RESULTADO < X) hacer
    Simular(p,N,ÉXITO,RESULTADO);
    N:=N+1;
  Fmq;
  Si RESULTADO=X
    entonces Devuelve 1
    else Devuelve 0;
  Fsi;
```

El programa anterior para siempre. ∎

3.5. Propiedades elementales de los conjuntos decidibles y semidecidibles

Propiedad 3.14 Todo conjunto finito es decidible.

Dem. Sea $L = \{a_1, \ldots, a_k\}$ un conjunto finito. Para resolver el problema

sólo necesitamos un programa que compare la entrada con k constantes, las k palabras de L:

```
Leer X
  Case X of
    a_1   Devuelve 1;
    a_2   Devuelve 1;
    ...
    a_k   Devuelve 1;
    else Devuelve 0;
```

■

Propiedad 3.15 Un conjunto L es decidible si y sólo si $\overline{L}$ es decidible. ($\overline{L}$ es el complementario de L, $\overline{L} = \{x \mid x \in \Sigma^* \wedge x \notin L\}$.)

Dem. Si p es un programa que resuelve L, el siguiente programa resuelve $\overline{L}$:

```
Leer X
  simular(p,X,ÉXITO,RESULTADO);
  Si RESULTADO=0
    entonces Devuelve 1
    else Devuelve 0;
```

La otra implicación es idéntica, ya que $\overline{\overline{L}} = L$. ■

Propiedad 3.16 Si A y B son conjuntos decidibles entonces $A \cup B$ es decidible y $A \cap B$ es decidible.

Dem. Tenemos p y q programas que resuelven A y B respectivamente. Los siguientes programas resuelven $A \cup B$ y $A \cap B$ respectivamente.

```
Leer X
  simular(p,X,ÉXITO,RESULTADO);
  simular(q,X,ÉXITO2,RESULTADO2);
  Si RESULTADO=1 OR RESULTADO2=1
    entonces Devuelve 1
    else Devuelve 0;
```

```
Leer X
  simular(p,X,ÉXITO,RESULTADO);
  simular(q,X,ÉXITO2,RESULTADO2);
  Si RESULTADO=1 AND RESULTADO2=1
    entonces Devuelve 1
    else Devuelve 0;
```

■

Propiedad 3.17 Un conjunto A es decidible si y sólo si A y $\overline{A}$ son ambos semidecidibles.

Dem. $\Rightarrow$) Si A es decidible entonces $\overline{A}$ es decidible (propiedad *3.15*). Si A y $\overline{A}$ son decidibles entonces A y $\overline{A}$ son ambos semidecidibles (propiedad *3.6*).
$\Leftarrow$) Por ser A y $\overline{A}$ semidecidibles tenemos p y q dos programas que resuelven al menos el caso positivo de A y $\overline{A}$, respectivamente. El siguiente programa resuelve A:

```
Leer X
  T:=1;
  TERMINADO:=FALSE;
  Mientras que NOT TERMINADO hacer
    SimularConReloj(p,X,T,ÉXITO,RESULTADO);
    Si ÉXITO
      entonces
        Devuelve RESULTADO;
        TERMINADO:=TRUE;
      else
        SimularConReloj(q,X,T,ÉXITO2,RESULTADO2);
        Si ÉXITO2
        entonces
          Devuelve 1-RESULTADO2;
          TERMINADO:=TRUE;
    Fsi; Fsi;
    T:=T+1;
  Fmq;
```

Si $x \in A$ entonces $p(x)\downarrow$, luego entrará en el primer entonces (si no ha entrado antes en el segundo). Si $x \notin A$ entonces $q(x)\downarrow$, luego entrará en el segundo entonces (si no ha entrado antes en el primero). Luego el programa para siempre, y la respuesta que da es siempre correcta. ∎

Esta última propiedad confirma la intuición de que A es semidecidible si se puede resolver ¿$x \in A$? en el caso afirmativo.
Como consecuencia tenemos el siguiente resultado para H (y para cualquier otro conjunto que sea semidecidible y no sea decidible).

Corolario 3.18 $\overline{H}$ no es semidecidible

Dem. Sabemos que H es semidecidible pero no decidible. Si $\overline{H}$ fuera semidecidible entonces, por la propiedad anterior H sería decidible. ∎

Propiedad 3.19 Si A y B son conjuntos semidecidibles entonces $A \cup B$ es semidecidible y $A \cap B$ es semidecidible

Dem. Tenemos p y q programas que resuelven al menos el caso positivo de A y B respectivamente. Los siguientes programas resuelven al menos el caso positivo de $A \cup B$ y $A \cap B$ respectivamente.

```
Leer X
 T:=1;
 TERMINADO:=FALSE;
 Mientras que NOT TERMINADO hacer
  SimularConTiempo(p,X,T,ÉXITO,RESULTADO);
  Si (ÉXITO) AND (RESULTADO=1)
   entonces
   TERMINADO:=TRUE;
  Fsi;
  SimularConTiempo(q,X,T,ÉXITO2,RESULTADO2);
  Si (ÉXITO2) AND (RESULTADO2=1)
   entonces
   TERMINADO:=TRUE;
  Fsi;
  T:=T+1;
 Fmq;
```

```
  Devuelve 1;


Leer X
  Simular(p,X,ÉXITO,RESULTADO);
  Simular(q,X,ÉXITO2,RESULTADO2);
Si ÉXITO AND ÉXITO2
    entonces Si RESULTADO=1 AND RESULTADO2=1 entonces
      Devuelve 1;
```

En ambos casos los programas anteriores dan salida en los casos que nos interesan ($A \cup B$ y $A \cap B$ respectivamente), a pesar de que los programas p y q no necesariamente paran siempre. ∎

EJERCICIOS

3.1. Dadas dos funciones calculables f y g, sea h la función

$$h(x) = 0, \quad \text{si } x \in \mathrm{Dom}(f) \cup \mathrm{Dom}(g).$$

Demostrar que h es calculable.

3.2. Lo mismo que el anterior, con h definida como

$$h(x) = 0, \quad \text{si } x \in \mathrm{Dom}(f) \cap \mathrm{Dom}(g).$$

3.3. Demostrar el Teorema *3.9*.

3.4. Demostrar que el conjunto de los conjuntos semidecidibles es numerable. Hacer lo mismo para el conjunto de los conjuntos decidibles.

3.5. 1. ¿Existe algún conjunto que no sea decidible y que contenga un subconjunto infinito decidible?

2. Demostrar que todo conjunto semidecidible e infinito tiene un subconjunto decidible infinito. (Idea: utilizar las caracterizaciones de las propiedades *3.12* y *3.13*.)

3.6. 1. Demostrar por diagonalización que existe un conjunto que no es semidecidible.

2. Demostrar por diagonalización que no existe un programa que genere T, el conjunto de programas que para para todas las entradas. (Idea: Para cada programa p que genera $A \subseteq T$, definimos un programa q con $\varphi_q(n) = \varphi_{\varphi_p(n)}(n) + 1$, que no está en A pero sí en T.)

3.7. Si definimos $\phi : \Sigma^* \times \Sigma^* \to \mathbb{N}$ como

$$\phi(x, y) = 1, \quad \text{si } x(z)\downarrow \text{ para algún } z \leq y.$$

¿Es ϕ calculable? ¿Es total?

3.8. Si definimos ϕ como

$$\phi(x, y) = 1, \quad \text{si } x(k)\downarrow \text{ para algún } k > y.$$

¿Es ϕ calculable? ¿Es total?

3.9. Demostrar que toda función de dominio finito es calculable.

3.10. Demostrar que no toda función de imagen finita es calculable.

3.11. Sea $f : \mathbb{N} \to \mathbb{N}$ la función definida como

$$f(n) = \sum_{i=0}^{n} \varphi_i(n), \quad \text{si } 0(n)\downarrow, 1(n)\downarrow, \ldots, n(n)\downarrow .$$

1. Demostrar que f tiene dominio finito y por tanto es calculable.
2. Demostrar que el conjunto $\{n, m \mid f(n) = m\}$ es decidible.

3.12. Sea ψ una función calculable y A un conjunto semidecidible. Demostrar que

$$\psi^{-1}(A) = \{x \mid \psi(x) \in A\}$$

es semidecidible.

3.13. Sea A un conjunto semidecidible Demostrar que el conjunto

$$B = \{x \mid \exists y \; x, y \in A\}$$

también es semidecidible.

3.14. Sea A un conjunto semidecidible Demostrar que

$$\bigcup_{i \in A} \mathrm{Dom}(\varphi_i)$$

también es semidecidible.

3.15. Demostrar que las funciones f y g definidas a continuación son calculables pero no tienen ninguna extensión calculable total.

1. $f(x,y) = \varphi_x(y) + 1$ si $x(y)\downarrow$. (Idea, por diagonalización, para cada h calculable total tomamos M un programa que calcula $v(x) = h(x,x)$. Estudiar $f(M,M)$ y $h(M,M)$.)
2. $g(x,y) = t$ si $x(y)$ para en exactamente t pasos. (Idea, usar reducción al absurdo, si existe tal extensión entonces H es decidible.)

3.16. Sea f la función que con entrada x devuelve la codificación del siguiente programa

Leer N
  Si N= x entonces Devuelve 1
    else Devuelve 0;

1. ¿Es f calculable? ¿Es f total? ¿Es f inyectiva?
2. Para cada x, ¿cuánto vale la función $\varphi_{f(x)}$? ¿Es calculable? ¿Es inyectiva?
3. ¿Qué se puede decir del conjunto

$$\{x, y \,|\, \varphi_{f(x)}(y) = \varphi_{f(y)}(x)\}?$$

4. ¿Es $H \cap \{f(x), x \,|\, x \in \Sigma^*\}$ un conjunto decidible?

3.17. Sea h la función que con entrada x devuelve la codificación del siguiente programa

Leer N
  Si N= x entonces Devuelve 1
    else repetir hasta que $1 > 2$;

1. ¿Es h calculable? ¿Es h total? ¿Es h inyectiva?
2. Para cada x, ¿cuánto vale la función $\varphi_{h(x)}$? ¿Es total? ¿Es calculable? ¿Es inyectiva?
3. ¿Qué se puede decir del conjunto

$$\{x, y \mid \varphi_{h(x)}(y) = \varphi_{h(y)}(x)\}?$$

4. ¿Es $H \cap \{h(x), x \mid x \in \Sigma^*\}$ un conjunto decidible?

Capítulo 4

Reducciones. El teorema de Rice

Referencia: Capítulos 9.1 y 6.1 de [Cu80]. Sección 10.1 de [Jones].

4.1. Reducciones

En este apartado formalizaremos la idea de reducir un conjunto (o problema decisional) a otro, como medio de de mostrar que un conjunto es no decidible, o bien que no es semidecidible.
La idea informal de que el conjunto A se puede reducir al conjunto B se puede expresar de varias formas:

1. "Resolver A se reduce a resolver B": a partir de un programa que resuelve B podemos construir otro que resuelve A.
2. Resolver A no es más difícil que resolver B.
3. Resolver A es tan fácil o más que resolver B.

Nosotros usaremos la siguiente formalización de reducibilidad, que corresponde al tipo más sencillo:

Definición 4.1 Un conjunto A es *reducible* a un conjunto B si existe una función f calculable y total tal que, para cada $x \in \Sigma^*$

$$x \in A \Leftrightarrow f(x) \in B$$

Es decir, f transforma la pregunta ¿$x \in A$? a ¿$f(x) \in B$? de forma que si puedo resolver ¿$f(x) \in B$? tengo resuelto ¿$x \in A$? porque sé que la respuesta es la misma.
Notación: A es reducible a B lo denotaremos:

$$A \leq_{\mathrm{m}} B$$

La función f de la definición anterior es una *reducción* de A a B.

Ejemplo 4.2 Sean H y K los conjuntos definidos en el capítulo anterior (el problema de parada y el problema diagonal de parada):

$$K = \{p \,|\, p(p)\downarrow\}$$
$$H = \{p, x \,|\, p(x)\downarrow\}$$

Veamos que $K \leq_{\mathrm{m}} H$.
Sea $f : \Sigma^* \to \Sigma^*$ la siguiente función:

$$\forall p \quad f(z) = \langle p, p\rangle$$

f es total y claramente calculable.
Veamos que f es una reducción de K en H.
Si $p \in K$ entonces $p(p)\downarrow$ y por tanto $p, p \in H$.
Si $p \notin K$ entonces $p(p)\uparrow$ y por tanto $p, p \notin H$.
Luego $p \in K \Leftrightarrow f(p) \in H$ y por tanto $K \leq_{\mathrm{m}} H$.

Ejemplo 4.3 Dados dos conjuntos A y B, sea

$$A \oplus B = \{0w \,|\, w \in A\} \cup \{1w \,|\, w \in B\}$$

es decir, las palabras de A con 0 delante y las palabras de B con 1 delante. Esto se denomina *unión marcada de A y B.*
Entonces

$$\begin{aligned} K \oplus \overline{K} &= \{0w \,|\, w \in K\} \cup \{1w \,|\, w \in \overline{K}\} \\ &= \{0w \,|\, w \in K\} \cup \{1w \,|\, w \notin K\} \end{aligned}$$

Veamos que

$$K \leq_{\mathrm{m}} K \oplus \overline{K}$$
$$\text{y} \quad \overline{K} \leq_{\mathrm{m}} K \oplus \overline{K}$$

Esto se demuestra utilizando las reducciones:

$$f(x) = 0x \qquad (\text{para } K \leq_{\mathrm{m}} K \oplus \overline{K})$$
$$g(x) = 1x \qquad (\text{para } \overline{K} \leq_{\mathrm{m}} K \oplus \overline{K}).$$

Ejemplo 4.4 Veamos que $H \leq_{\mathrm{m}} K$. Sea $f : \Sigma^* \to \Sigma^*$ la función:

$$f(p,x) = \begin{array}{l} \text{“Leer N} \\ \text{constantes CP:=}p\text{; CX:=}x\text{;} \\ \text{Simular(CP, CX, ÉXITO);} \\ \text{Si ÉXITO entonces Devuelve 5.”} \end{array}$$

f es una función total y calculable calculada por el programa:

```
Leer Q,X
  RESULTADO:=CONCATENAR("Leer N; constantes CP:=",Q);
  RESULTADO:=CONCATENAR(RESULTADO, "; CX:=");
  RESULTADO:=CONCATENAR(RESULTADO, X);
  RESULTADO:=CONCATENAR(RESULTADO, "; Simular(CP, CX, ÉXITO); Si ÉXITO e
  Devuelve RESULTADO;
```

Veamos que f es una reducción de H en K.
Si $p, x \in H$ entonces $p(x)\downarrow$ y por tanto $\forall n\ f(p,x)(n)\downarrow$, es decir, para cualquier n el programa $f(p,x)$ para con entrada n. Por tanto $f(p,x)$ con entrada $f(p,x)$ para.
Si $p, x \notin H$ entonces $p(x)\uparrow$ y por tanto $\forall n\ f(p,x)(n)\uparrow$, es decir, para cualquier n el programa $f(p,x)$ con entrada n no para. Por tanto $f(p,x)(f(p,x))\uparrow$ ($f(p,x)$ con entrada $f(p,x)$ no para).
Luego

$$p, x \in H \Rightarrow f(p,x) \in K$$
$$p, x \notin H \Rightarrow f(p,x) \notin K$$

y por tanto $H \leq_{\mathrm{m}} K$.

4.2. Propiedades elementales de las reducciones

El siguiente teorema explica el interés de las reducciones. En él se formaliza la idea de que si $A \leq_{\mathrm{m}} B$ entonces A es tanto o más fácil que B.

Teorema 4.5 Sean A y B dos conjuntos tales que $A \leq_{\mathrm{m}} B$. Entonces se cumple que:

1. *Si B es decidible entonces A es decidible.*

2. Si B es semidecidible entonces A es semidecidible.

Dem. Sea f una reducción de A a B.
1. Como B es decidible, tenemos un programa p que resuelve B. Veamos que A es decidible dando un programa que resuelve A:

```
Leer X
  Y:=f(X);
  Simular(p,Y,ÉXITO,RESULTADO);
  Si RESULTADO=1
    entonces Devuelve 1
    else Devuelve 0.
```

El algoritmo anterior para siempre ya que f es calculable total y p para siempre. Como $x \in A \Leftrightarrow f(x) \in B$, el algoritmo anterior resuelve A.
2. Como B es semidecidible, tenemos un programa p que resuelve al menos el caso positivo de B. Veamos que A es semidecidible dando un programa que resuelve al menos el caso positivo de A:

```
Leer X
  Y:=f(X);
  Simular(p,Y,ÉXITO,RESULTADO);
  Si ÉXITO
    Si RESULTADO=1 entonces Devuelve 1.
```

El algoritmo anterior para cuando $x \in A$ y resuelve al menos el caso positivo de B ya que f es calculable total, p para cuando $z \in B$ y $x \in A \Leftrightarrow f(x) \in B$. ∎

El teorema anterior se puede enunciar equivalentemente como:

Teorema 4.6 Sean A y B dos conjuntos tales que $A \leq_{\mathrm{m}} B$. Entonces se cumple que:

1. *Si A no es decidible entonces B no es decidible.*
2. *Si A no es semidecidible entonces B no es semidecidible.*

lo que nos servirá para demostrar que algunos conjuntos son indecidibles o no son semidecidibles.

Ejemplo 4.7 Sea A el conjunto:

$$A = \{x \mid \varphi_x \text{ es inyectiva y } \mathrm{Dom}(\varphi_x) \neq \emptyset\}$$

Veamos que $K \leq_{\mathrm{m}} A$.
Sea $f : \Sigma^* \to \Sigma^*$ la función:

$$f(p) = \begin{array}{l} \text{“Leer N} \\ \text{constante CP:=}p\text{;} \\ \text{Simular(CP, CP, ÉXITO);} \\ \text{Si ÉXITO entonces Devuelve N.”} \end{array}$$

f es claramente total y calculable. Veamos que es la reducción que buscamos:
$p \in K \Rightarrow p(p) \downarrow \Rightarrow f(p)$ es un programa que calcula la identidad, $\forall n\ \varphi_{f(p)}(n) = n \Rightarrow \varphi_{f(p)}$ es inyectiva, $\mathrm{Dom}(\varphi_{f(p)}) = \Sigma^* \Rightarrow f(p) \in A$.
$p \notin K \Rightarrow p(p) \uparrow \Rightarrow f(p)$ es un programa que no para para ninguna entrada, $\varphi_{f(p)}$ tiene dominio vacío $\Rightarrow f(p) \notin A$.
Luego $K \leq_{\mathrm{m}} A$.
Utilizando el teorema *4.6* (que es el mismo que el *4.5*), como sabemos que K no es decidible (teorema *3.3*) sabemos que A no es decidible.

El siguiente teorema demuestra que los conjuntos decidibles son reducibles a todos los conjuntos. Intuitivamente esto quiere decir que según el orden marcado por las reducciones $\leq_{\mathrm{m}}$ los conjuntos decidibles son los más fáciles.

Teorema 4.8 Sea A un conjunto decidible. Sea B un conjunto cualquiera tal que $B \neq \emptyset$ y $B \neq \Sigma^*$. Entonces $A \leq_{\mathrm{m}} B$

Dem. Por ser $B \neq \emptyset$ y $B \neq \Sigma^*$ sabemos que existen palabras dentro y fuera de B. *Fijamos* $b_0 \in B$ y $b_1 \notin B$. La función

$$f(x) = \begin{cases} b_0 & \text{si } x \in A \\ b_1 & \text{si } x \notin A \end{cases}$$

es total y calculable ya que la calcula el siguiente algoritmo, donde p es un programa que resuelve A:

Leer X
  SIMULAR(p, X, ÉXITO, RESULTADO);
  Si RESULTADO=1
    entonces Devuelve b_0
    else Devuelve b_1.

(En este algoritmo b_0 y b_1 son constantes.)
f es claramente una reducción de A en B. ∎

La reducción $\leq_{\mathrm{m}}$ es transitiva:

Teorema 4.9 Si $A \leq_{\mathrm{m}} B$ y $B \leq_{\mathrm{m}} C$ entonces $A \leq_{\mathrm{m}} C$.

Dem. Sea f una reducción de A en B y g una reducción de B en C. Entonces la composición de f y g, $g \circ f(x) = g(f(x))$ es una reducción de A a C, ya que es calculable total (por serlo f y g) y cumple:

$$x \in A \Leftrightarrow f(x) \in B \Leftrightarrow g(f(x)) = g \circ f(x) \in C.$$

∎

Vamos a ver a continuación que H y K están entre los más difíciles de los conjuntos semidecidibles.

Definición 4.10 Dado un conjunto o clase de conjuntos C, un conjunto $X \in$ C es *completo para C* si para todo $A \in$ C, $A \leq_{\mathrm{m}} X$.

Teorema 4.11 H es completo para la clase de los conjuntos semidecidibles, es decir, si A es un conjunto semidecidible entonces $A \leq_{\mathrm{m}} H$.

Dem. Sea A un conjunto semidecidible. Sea p un programa que resuelve al menos el caso positivo de A. Existe un programa que resuelve al menos el caso positivo de A y sólo da salida para las entradas en A:

x_0: Leer X
    Simular(p, X, ÉXITO, RESULTADO);
Si ÉXITO entonces
    Si RESULTADO=1 entonces Devuelve 1.

Sea $f : \Sigma^* \to \Sigma^*$ la función:

$$f(z) = x_0, z$$

f es claramente una función total y calculable calculada por el programa:

Leer Z
  Devuelve x_0, Z.

(x_0 es una constante del programa.)
Veamos que f es una reducción de A en H.

$$z \in A \Rightarrow \varphi_{x_0}(z) = 1 \Rightarrow x_0(z)\downarrow \Rightarrow f(z) = x_0, z \in H$$
$$z \notin A \Rightarrow x_0(z)\uparrow \Rightarrow f(z) = x_0, z \notin H$$

■

Por la transitividad tenemos:

Corolario 4.12 Sea X un conjunto semidecidible y sea C un conjunto completo para los semidecidibles. Si $C \leq_{\mathrm{m}} X$ entonces X es completo para la clase de los conjuntos semidecidibles.

Dem. Sea A un conjunto semidecidible. Como sabemos que $A \leq_{\mathrm{m}} C$ y por hipótesis $C \leq_{\mathrm{m}} X$, aplicando transitividad (teorema *4.9*) tenemos que $A \leq_{\mathrm{m}} X$. ■

Con los dos últimos resultados tenemos:

Corolario 4.13 K es completo para la clase de los conjuntos semidecidibles.

Dem. $H \leq_{\mathrm{m}} K$ (ejemplo *4.4*) y H es completo para los semidecidibles (teorema *4.11*), luego por el corolario anterior K es completo para los semidecidibles. ■

Nota: Las reducciones que utilizaremos en los problemas serán a menudo reducciones desde H o desde $\overline{H}$. Además serán casi todas de unos de los dos tipos que incluímos a continuación. Definimos las funciones f_1 y f_2 como:

$f_1(p, x) =$ “Leer N
constante CP:=p; CX:=x;
Simular(CP, CX, ÉXITO);
Si ÉXITO entonces
CÓDIGO1.”

$f_2(p,x) =$ "Leer N
constante CP:=p; CX:=x;
SimularConTiempo(CP, CX, N, ÉXITO);
Si ÉXITO entonces
CÓDIGO1
else
CÓDIGO2."

Veamos cómo funcionaran f_1 y f_2 como reducciones desde H:
$p, x \in H \Rightarrow p(x)\downarrow \Rightarrow f_1(p,x)$ es un programa que ejecuta CÓDIGO1.
$p, x \notin H \Rightarrow p(x)\uparrow \Rightarrow f_1(p,x)$ es un programa que no para para ninguna entrada, $\varphi_{f_1(p,x)}$ tiene dominio vacío.
$p, x \in H \Rightarrow p(x)\downarrow \Rightarrow \exists n_0$ tal que p con entrada x tarda exactamente tiempo n_0, luego $f_2(p,x)$ es un programa que con entrada $n < n_0$ ejecuta CÓDIGO2 y con entrada $n \geq n_0$ ejecuta CÓDIGO1.
$p, x \notin H \Rightarrow p(x)\uparrow \Rightarrow f_2(p,x)$ es un programa que ejecuta CÓDIGO2.
Eligiendo adecuadamente CÓDIGO1 y CÓDIGO1 podemos tener gran variedad de reducciones, como veremos en los problemas.

4.3. Conjuntos de índices, teorema de Rice

Hasta ahora hemos visto dos tipos de demostraciones de que un conjunto no es decidible: la que utilizamos para el problema de parada, basada en diagonalización, y las demostraciones del apartado anterior basadas en reducciones desde un conjunto indecidible utilizando el teorema *4.5*. Vamos a ver ahora un tercer método que sirve exclusivamente para los conjuntos que llamaremos *conjuntos de índices*, que nos servirá para ahorrarnos unas cuantas reducciones.

Definición 4.14 Sea f una función. Llamaremos *conjunto de índices de* f al siguiente conjunto:

$$\mathrm{IND}(f) = \{p \,|\, \varphi_p \equiv f\}$$

Es decir, el conjunto de índices de una función f está formado por todos los programas que calculan f.

Por supuesto, si f no es calculable entonces $\mathrm{IND}(f) = \emptyset$.
Ejercicio: Demostrar que si f es una función calculable entonces $\mathrm{IND}(f)$ es infinito.

Definición 4.15 Sea F un conjunto de funciones. Llamaremos *conjunto de índices de* F a:

$$\mathrm{IND}(F) = \bigcup_{f \in F} \mathrm{IND}(f) = \{p \mid \varphi_p \in F\}$$

es decir, el conjunto de los programas que calculan una función de F.

Definición 4.16 Sea A un conjunto. Diremos que A es un *conjunto de índices* si existe un conjunto de funciones F tal que $A = \mathrm{IND}(F)$.

En otras palabras, si un conjunto es un conjunto de índices y contiene un programa que calcula una función contiene todos los programas que la calculan, y viceversa, si no contiene uno no contiene ninguno.
La siguiente propiedad se deja como ejercicio:

Propiedad 4.17 Sea A un conjunto. A es un conjunto de índices si y sólo si

$$\bigcup_{x \in A} IND_{\varphi_x} \subseteq A$$

A continuación veremos que ningún conjunto de índices no trivial es decidible.

Teorema 4.18 Teorema de Rice. Sea A un conjunto de índices con $A \neq \emptyset$ y $A \neq \Sigma^*$. Entonces A es indecidible.

Dem. Vamos a ver que se cumple al menos una de las dos siguientes afirmaciones:

1. $H \leq_{\mathrm{m}} A$
2. $\overline{H} \leq_{\mathrm{m}} A$

Si demostramos 1., como H no es decidible, A no puede serlo tampoco.
Si demostramos 2., como $\overline{H}$ no es decidible, A tampoco.
Sea v la función de dominio vacío, función que calculan los programas que no paran para ninguna entrada. Como A es un conjunto de índices, tenemos dos casos posibles:

1. $\mathrm{IND}(v) \cap A = \emptyset$. En este caso demostraremos que $H \leq_{\mathrm{m}} A$.
2. $\mathrm{IND}(v) \subseteq A$. Aquí demostraremos que $\overline{H} \leq_{\mathrm{m}} A$.

Caso 1: Si $\mathrm{IND}(v) \cap A = \emptyset$, sea g una función calculable tal que $\mathrm{IND}(g) \subseteq A$ (g existe porque $A \neq \emptyset$ y es un conjunto de índices), sea w_0 un programa que calcula g. Definimos α la siguiente función:

$$\alpha(p,x) = \begin{array}{l} \text{“Leer N} \\ \text{constante CP:=}p\text{; CX:=}x\text{;} \\ \text{Simular(CP,CX,ÉXITO);} \\ \text{Si ÉXITO entonces} \\ \quad \text{Simular}(w_0\text{,N,ÉXITO2,RESULTADO2);} \\ \text{Si ÉXITO2 entonces Devuelve RESULTADO2.”} \end{array}$$

Veamos que α es una reducción de H en A. Si $p(x)\downarrow$, $\alpha(p,x)$ es un programa que calcula g. Si $p(x)\uparrow$ entonces $\alpha(p,x)$ es un programa que no para con ninguna entrada, luego calcula v. Por tanto

$$p,x \in H \Rightarrow \alpha(p,x) \in \mathrm{IND}(g) \Rightarrow \alpha(p,x) \in A$$
$$p,x \notin H \Rightarrow \alpha(p,x) \in \mathrm{IND}(v) \Rightarrow \alpha(p,x) \notin A$$

Luego en este caso $H \leq_{\mathrm{m}} A$.
Caso 2: Si $\mathrm{IND}(v) \subseteq A$, sea f una función calculable tal que $\mathrm{IND}(f) \cap A = \emptyset$ (f existe porque $A \neq \Sigma^*$ y es un conjunto de índices), sea w_1 la codificación de un programa que calcula f. Definimos β la siguiente función:

$$\beta(p,x) = \begin{array}{l} \text{“Leer N} \\ \text{constante CP:=}p\text{; CX:=}x\text{;} \\ \text{Simular(CP,CX,ÉXITO);} \\ \text{Si ÉXITO entonces} \\ \quad \text{Simular}(w_1\text{,N,ÉXITO2,RESULTADO2);} \\ \text{Si ÉXITO2 entonces Devuelve RESULTADO2.”} \end{array}$$

Veamos que β es una reducción de $\overline{H}$ en A. Si $p(x)\uparrow$ entonces $\beta(p,x)$ es un programa que no para con ninguna entrada, luego calcula v. Si $p(x)\downarrow$, $\beta(p,x)$ es un programa que calcula f. Por tanto

$$p,x \in \overline{H} \Rightarrow \beta(p,x) \in \mathrm{IND}(v) \Rightarrow \beta(p,x) \in A$$
$$p,x \notin \overline{H} \Rightarrow \beta(p,x) \in \mathrm{IND}(f) \Rightarrow \beta(p,x) \notin A$$

Luego en este caso $\overline{H} \leq_{\mathrm{m}} A$. ∎

Como corolario al caso 2 de la demostración anterior tenemos:

Corolario 4.19 Sea A un conjunto de índices con $A \neq \emptyset$ y $A \neq \Sigma^*$ y tal que A contiene un programa que calcula la función vacía. Entonces A no es semidecidible.

Dem. Esto es consecuencia de la demostración anterior, ya que este caso, $\mathrm{IND}(v) \subseteq A$, demostramos que $\overline{H} \leq_{\mathrm{m}} A$. ∎

¿Cuándo podemos utilizar el teorema de Rice para demostrar que un conjunto no es decidible? Cuando dicho conjunto es un conjunto de índices, es decir, cuando el hecho de que p esté o no en el conjunto sólo depende de quién es φ_p.

Ejemplos 4.20

- Sea $L = \{x \mid \varphi_x \text{ es inyectiva}\}$. L es el conjunto de índices de $F = \{f \mid f \text{ es inyectiva}\}$. $L \neq \emptyset$ ya que existen funciones calculables e inyectivas, por ejemplo la función identidad $ident(x) = x\ \forall x$. $L \neq \Sigma^*$ ya que existen funciones calculables no inyectivas, por ejemplo una función constante $f(x) = 0\ \forall x$. Luego por el teorema de Rice, L no es decidible.

- No se puede usar Rice para el siguiente conjunto

  $$L = \{x \mid \varphi_x(x) = 1\}$$

  ya que no es un conjunto de índices, porque pueden existir dos programas $x \neq y$ que calculan la misma función f y tales que $f(x) = 1$, $f(y) \neq 1$. Tanto x como y están en $\mathrm{IND}(f)$, pero $x \in L$ y $y \notin L$. Luego L no es un conjunto de índices.

- Tampoco se puede usar para

  $$K = \{p \mid p(p)\downarrow\}$$

  ya que pueden existir dos programas x, y que calculen la misma función f, con $f(x)$ definida y $f(y)$ indefinida. En este caso $x \in K$, $y \notin K$, $x, y \in \mathrm{IND}(f)$, luego K no es un conjunto de índices.

En los dos casos anteriores el hecho de que p esté o no en el conjunto NO depende sólo de quién es φ_p, sino también de p tomado como entrada. Por tanto no se puede aplicar Rice.
Cuando no es posible utilizar Rice, otro método para demostrar que un conjunto no es decidible son las reducciones vistas en este mismo capítulo.

Ejemplo 4.21 Veamos que $H \leq_{\mathrm{m}} L$, donde

$$L = \{x \mid \varphi_x(x) = 1\}$$

Como sabemos que H no es decidible, entonces L no lo es.
La reducción es

$$f(p,x) = \begin{array}{l} \text{“Leer N} \\ \text{constante CP:=}p\text{; CX:=}x\text{;} \\ \text{Simular(CP,CX,ÉXITO);} \\ \text{Si ÉXITO entonces Devuelve 1.”} \end{array}$$

Si $p, x \in H \Rightarrow p(x)\downarrow \Rightarrow f(p,x)$ es un programa que calcula la función constante 1 $\Rightarrow \varphi_{f(p,x)}(f(p,x)) = 1 \Rightarrow f(p,x) \in L$. Si $p, x \notin H \Rightarrow p(x)\uparrow \Rightarrow f(p,x)$ es un programa que no para nunca $\Rightarrow f(p,x)(f(p,x))\uparrow$ $\Rightarrow f(p,x) \notin L$. ∎

EJERCICIOS

Decir si los siguientes conjuntos son o no decidibles, y si son o no semidecidibles.

4.1. $\{x \mid \varphi_x(x) = x\}$.

4.2. $\{x, y, z \mid \varphi_x(y) = z\}$.

4.3. $\{x, y, z \mid \varphi_x(z) = \varphi_y(z)\}$.

4.4. $\{x \mid \varphi_x$ es suprayectiva$\}$.

4.5. $\{x \mid \varphi_x$ no es suprayectiva$\}$.

4.6. $\{x \mid \varphi_x$ es inyectiva$\}$.

4.7. $\{x \mid \varphi_x$ no es inyectiva$\}$.

4.8. $\{x \mid \varphi_x$ es biyectiva$\}$.

4.9. $\{x \mid \varphi_x$ no es biyectiva$\}$.

4.10. $\{x \mid \varphi_x$ es total$\}$.

4.11. $\{x \mid x$ no da salida para ninguna entrada$\}$.

4.12. $\{x \mid \mathrm{Dom}(\varphi_x)$ es indecidible$\}$.

4.13. $\{x \mid \mathrm{Dom}(\varphi_x)$ es decidible$\}$.

4.14. $\{x \mid \mathrm{Dom}(\varphi_x)$ no es semidecidible$\}$.

4.15. $\{x \mid \mathrm{Dom}(\varphi_x)$ es semidecidible $\}$.

4.16. $\{x \mid \mathrm{Im}(\varphi_x) = \emptyset\}$.

4.17. $\{x \mid \mathrm{Im}(\varphi_x)$ es indecidible $\}$.

4.18. $\{x \mid \mathrm{Im}(\varphi_x)$ es decidible $\}$.

4.19. $\{x \mid \mathrm{Im}(\varphi_x)$ no es semidecidible $\}$.

4.20. $\{x \mid \mathrm{Im}(\varphi_x)$ es semidecidible $\}$.

4.21. $\{x, y \mid y \in \mathrm{Im}(\varphi_x)\}$.

4.22. $\{x \mid \varphi_x$ es constante$\}$.

4.23. $\{x \mid \varphi_x$ tiene una extensión calculable total$\}$. Pista: usar el ejercicio 3.15.

4.24. $\{x, y \mid \mathrm{Dom}(\varphi_x) = \mathrm{Dom}(\varphi_y)\}$.

4.25. $\{x \mid \exists t(x(t)\downarrow$ y $\varphi_x(t) = \varphi_t(t))\}$.

4.26. $\{x \mid \mathrm{Im}(\varphi_x) \subseteq \{0, 1\}\}$.

4.27. $\{x \mid \mathrm{Im}(\varphi_x) \subseteq \{0, 1\}$ y $(\forall y \varphi_x(y) = 1$ si y sólo si $y(y)\downarrow)\}$.

4.28. $\{x \mid \mathrm{Im}(\varphi_x) \subseteq \{0, 1\}$ y $(\forall y \varphi_x(y) = 1 \Rightarrow y(y)\downarrow)\}$.

4.29. $\{x, y \mid \varphi_x = \varphi_y\}$.

4.30. Demostrar la Propiedad *4.17*.

Capítulo 5

Otros problemas indecidibles

Referencia: Capítulos 6.3 de [Cu80] y 9.4 de [HMU02].

De los capítulos anteriores sabemos que los siguientes problemas son indecidibles:

- Dado un programa p, ¿termina p con cualquier entrada?
- Dados dos programas p y q, ¿calculan p y q la misma función?
- Dado un programa p y una entrada x, ¿para p con entrada x?

A continuación enunciamos algunos problemas indecidibles históricos. Por falta de tiempo no demostraremos la indecidibilidad.
Ecuaciones diofánticas. Dada una ecuación de cualquier grado con coeficientes enteros, ¿tiene dicha ecuación alguna solución entera?
El problema de correspondencia de Post. Dadas dos listas de palabras de Σ^*, $A = (x_1, \ldots, x_n)$ y $B = (y_1, \ldots, y_n)$, ¿existe una secuencia no vacía de enteros $(i_1, \ldots, i_r)$ con $1 \leq i_j \leq n$ para $1 \leq j \leq r$ tal que $x_{i_1} \ldots x_{i_r} = y_{i_1} \ldots y_{i_r}$.
Ejemplo de entrada del problema de Post:

$$\begin{array}{ll} x_1 = 111 & y_1 = 1 \\ x_2 = 10 & y_2 = 10111 \\ x_3 = 0 & y_3 = 10 \end{array}$$

Para esta entrada la solución es SI, ya que

$$y_2 y_1 y_1 y_3 = x_2 x_1 x_1 x_3 = 101111110$$

Capítulo 6

Otros modelos de cálculo: la tesis de Turing-Church

Referencia: Capítulos 2 y 3 de [Cu80].

Hemos definido en los capítulos anteriores el concepto de función calculable por un programa de nuestro modelo RAM (así como el concepto de conjunto decidible que se puede caracterizar a partir del de función calculable). En los últimos 50 años ha habido muchas propuestas distintas para una caracterización formal precisa de la idea informal de "lo que se puede calcular de manera automática".
La propuesta más reciente es la que hemos presentado en el capítulo 2, la máquina de registros o RAM. En este capítulo vamos a considerar algunas otras formalizaciones:

- Las funciones recursivas de Gödel y Kleene (1936).
- Las máquinas de Turing (1936).
- El λ-cálculo de Church (1930).

De estos modelos nos interesan dos cuestiones:

1. ¿Cómo se relacionan las distintas formalizaciones entre sí, y en particular con la RAM?
2. ¿Con qué exactitud caracterizan estos modelos (y en particular la RAM) la idea intuitiva de calculable?

6.1. Las funciones recursivas de Gödel y Kleene

Gödel y Kleene definen explícitamente qué funciones son calculables, es decir, dan una definición independiente de un modelo de cálculo concreto. Veremos que su definición coincide con la nuestra de función calculable.
En esta sección trabajamos con funciones de $\mathbb{N}^n$ en $\mathbb{N}$ ($n \geq 1$).
Veamos primero tres formas de construir unas funciones a partir de otras. Sea $x = (x_1, \ldots, x_n)$.

1. Dadas las funciones $f(y_1, \ldots, y_k), g_1(x), g_2(x), \ldots g_k(x)$ la *sustitución* de $g_1, \ldots, g_k$ en f es la función h:

$$h(x) = f(g_1(x), g_2(x), \ldots g_k(x))$$

2. Dadas las funciones $f(x)$, $g(x, y, z)$ la *recursión* de f y g es la función h:

$$\begin{aligned} h(x, 0) &= f(x) \\ h(x, y+1) &= g(x, y, h(x, y)) \end{aligned}$$

3. Dada la función $f(x, y)$, la *minimalización* de f es la función h tal que $h(x) =$

   - el mínimo y tal que
     (i) $f(x, y) = 0$,
     (ii) $f(x, z)$ está definido, para todo $z \leq y$,
     si existe tal y.
   - indefinida, en otro caso

   (h es una versión fuerte de "el menor y tal que $f(x, y) = 0$").

Ejemplos 6.1

- La función h

$$\begin{aligned} h(x, 0) &= x + 1 \\ h(x, y+1) &= x + h(x, y) \end{aligned}$$

es la recursión de $f(x) = x + 1$, $g(x, y, z) = x + z$.

- La función h

$$\begin{aligned} h(0) = &\quad 1 \\ h(y+1) = &\quad y \cdot h(y) \end{aligned}$$

  es la recursión de $f \equiv 1$, $g(y,z) = y \cdot z$.

- Dado $p(x)$ un polinomio de coeficientes enteros, la función

$$h(z) = \begin{array}{l} \text{“la menor raiz entera de } p(x) - z \\ \text{(si existe tal raiz)”} \end{array}$$

  es la minimalización de $f(z,x) = p(x) - z$.

A continuación damos la definición de funciones recursivas de Gödel y Kleene.

Definición 6.2 El conjunto de las *funciones recursivas de Gödel y Kleene* es el menor conjunto que *contiene* las siguientes funciones:

1. La función constante 0: $0(n) = 0\ \forall n$.
2. La función sucesor $s(n) = n + 1\ \forall n$.
3. Para cada $n \in \mathbb{N}$, $i \leq n$, la función de proyección i-ésima U_i^n definida como:
$$U_i^n(x_1, \ldots, x_n) = x_i$$

y es cerrado por las operaciones de sustitución, recursión y minimalización.

Nota: Un conjunto A es cerrado por una operación $*$ si para todo $f, g \in A$ se cumple que $f * g \in A$.
Esta definición de función recursiva de Gödel y Kleene dada es formalista, no usa ningún modelo de cálculo sino que a partir de unas funciones elementales (1., 2. y 3.) se construyen todas las funciones recursivas de Gödel y Kleene usando sustitución, recursión y minimalización.

Ejemplos 6.3 Algunas funciones recursivas de Gödel y Kleene:

- La función suma, obtenida como recursión de U_1^1 y la función sucesor:

$$\begin{aligned} suma(n,0) &= U_1^1(n) \\ suma(n,m+1) &= s(U_3^3(n,m,suma(n,m))) \end{aligned}$$

- La función suma de 3, obtenida aplicando la sustitución a la anterior:

$$h(x,y,z) = suma(x, suma(y,z)) = \\ suma(U_1^3(x,y,z), suma(U_2^3(x,y,z), U_3^3(x,y,z)))$$

- La función predecesor $p(m+1) = m,\ p(0) = 0$ puede obtenerse como $p(m) = h(0,m)$ con h:

$$\begin{aligned} h(n,0) &= 0(n) \\ h(n,m+1) &= U_2^3(n,m,h(n,m)) \end{aligned}$$

- La función $x \dot{-} y = \text{máx}(x-y, 0)$ obtenida como recursión de U_1^1 y la función predecesor:

$$\begin{aligned} n \dot{-} 0 &= U_1^1(n) \\ n \dot{-} (m+1) &= p(U_3^3(n,m,n \dot{-} m)) \end{aligned}$$

- Si f es recursiva de Gödel y Kleene, inyectiva y total, la función $f^{-1}(y) = x$ con $f(x) = y$ obtenida por minimalización de h:

$$h(y,x) = (y \dot{-} f(x)) + (f(x) \dot{-} y)$$

Veamos que la definición de función recursiva de Gödel y Kleene coincide con nuestra definición de función calculable, dada en el capítulo 2.

Teorema 6.4 Una función f es recursiva de Gödel y Kleene si y sólo si es calculable.

Dem. (esquema)
(Parte I) Demostramos que las funciones recursivas de Gödel y Kleene son calculables:
Una función h recursiva de Gödel y Kleene se obtiene aplicando a funciones básicas ($0(x)$, $s(x)$, $U_i^n(x_1, \ldots, x_n)$) un número finito de veces t, operaciones de sustitución, recursión y minimalización.
Por inducción sobre t, el número de operaciones aplicadas:

1. $t = 0$. Se trata de una de las funciones básicas, para cualquiera de ellas podemos construir fácilmente un programa que la calcula.

2. Paso de inducción, $t > 0$. Hay tres casos:

   (a) h es la sustitución de $g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n)$ en f, y cada una de las funciones $g_1, \ldots, g_k, f$ se obtiene aplicando menos de t operaciones. Por hipótesis de inducción, las funciones $g_1, \ldots, g_k, f$ son calculables, y existen programas $p_1, \ldots, p_k, q$ que calculan respectivamente $g_1, \ldots, g_k, f$. Utilizando estos programas y siguiendo la definición de h, tenemos un programa para h (nótese que n es un valor fijo):

   Leer $\mathrm{X}_1, \ldots, \mathrm{X}_n$
   Para I:=1 hasta k hacer
   $\mathrm{A}_\mathrm{I} := \varphi_{p_\mathrm{I}}(\mathrm{X}_1, \ldots, \mathrm{X}_n)$;
   Devuelve $\varphi_q(\mathrm{A}_1, \ldots, \mathrm{A}_k)$.

   Por tanto h es calculable.

   (b) Los casos en que h es la recursión de dos funciones f y g, ó h es es la minimalización de una función f se demuestran análogamente al caso (a), utilizando la hipótesis de inducción y la programación de dichas operaciones.

(Parte II) Esquema de la demostración de que las funciones calculables son recursivas de Gödel y Kleene:
Sea h una función calculable calculada por un programa p. Definimos dos funciones auxiliares

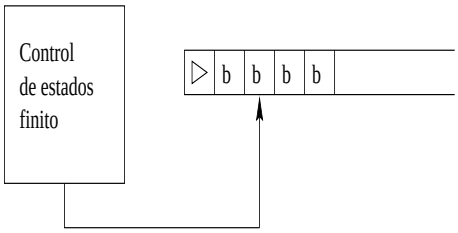
- $c(x, t) =$ "contenido del registro de salida depués de t pasos de p con entrada x".
- $j(x, t) =$ "número de la instrucción ejecutada en el paso t de p con entrada x (vale 0 si p con entrada x termina antes)"

Notemos que la minimalización de $j(x, t)$ es el tiempo que tarda el programa p con entrada x.
Si llamamos $f(x)$ a esa minimalización, $h(x) = c(x, f(x))$.
La demostración se completa demostrando que tanto c como j son funciones recursivas de Gödel y Kleene. ∎

Debido a la equivalencia de esta definición con la de función calculable basada en el modelo RAM, podemos asegurar que todo programa es equivalente a un número finito de aplicaciones de los operadores sustitución, recursión y minimalización.

6.2. Las máquinas de Turing

Una máquina de Turing, abreviadamente TM, es un autómata finito con una cinta infinita adicional. La cinta está dividida en celdas, la máquina accede a la información de la cinta a través de una cabeza lectora/escritora que puede leer/escribir sobre una única celda cada vez. La cabeza se puede mover a derecha o izquierda, pero sólo una posición cada vez.



Una transición de una máquina de Turing depende del estado en que está la máquina y del contenido de la cabeza lectora, según estos se realizarán las siguientes tres acciones:

- cambio de estado,
- escritura en la celda sobre la que está la cabeza,
- movimiento de la cabeza.

Inicialmente, la posición más a la izquierda de la cinta contiene un carácter especial $\triangleright$. A partir de esta posición contiene la palabra de entrada. El resto de la cinta contiene en todas las celdas un carácter especial denominado *blanco,* al que representaremos con el signo b.
Admitiremos tres tipos de movimientos para la cabeza: celda a la derecha, celda a la izquierda y no moverse. Si la cabeza está en la posición más a la izquierda de la cinta y se pide un movimiento a la izquierda, la cabeza no se moverá.
La definición formal de una máquina de Turing es:

Definición 6.5 Una *máquina de Turing* es una estructura $(Q, \Sigma, \Gamma, \delta, q_0)$, donde

- Q es un conjunto finito de estados,

- Σ es un alfabeto, el alfabeto de entrada,
- $\Gamma = \Sigma \cup \{\triangleright, \mathrm{b}\}$ es el alfabeto de cinta (b,$\triangleright \notin \Sigma$),
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{i}, \mathrm{d}, \mathrm{n}\}$ es una función de transición,
- q_0 es el estado inicial ($q_0 \in Q$),

La función de transición especifica, dado el estado actual y el símbolo que lee la cabeza, el nuevo estado, el nuevo símbolo de dicha posición y un movimiento de la cabeza.
Inicialmente la máquina se encuentra en el estado q_0, con una palabra $w \in \Sigma^*$ escrita en la parte izquierda de la cinta, precedida por el símbolo $\triangleright$, y la cabeza sobre la segunda celda de la cinta (primer símbolo de w). La máquina ejecuta transiciones mientras pueda aplicar la función de transición. Si en algún momento la máquina se encuentra en un estado q con carácter a en la cabeza y no hay transición definida para el par (q, a), entonces la máquina para.
Nota: Existen varias definiciones equivalentes de máquina de Turing que pueden encontrarse en la literatura. Por ejemplo se pueden definir máquinas de Turing con dos (o más) cintas, en ese caso la entrada se escribe únicamente en la primera cinta, inicialmente todas las cabezas de cinta están sobre la segunda celda, y la función de transición es de la forma:

$$\delta : Q \times \Gamma \times \Gamma \to Q \times \Gamma \times \{\mathrm{i}, \mathrm{d}, \mathrm{n}\} \times \Gamma \times \{\mathrm{i}, \mathrm{d}, \mathrm{n}\}$$

Utilizaremos la siguiente notación:

- $M(w)\downarrow$ representa que la TM M con entrada w para.
- $M(w)\uparrow$ representa que la TM M con entrada w no para.
- $M(w)$ representa el contenido de la cinta desde el carácter $\triangleright$ hasta el primer blanco, después de que M con entrada w ha parado (si $M(w)\downarrow$).

Definición 6.6 La *función calculada por una TM* M, denotada por φ_M, se define como:

$$\varphi_M(w) = M(w) \text{ si } M(w)\downarrow$$

Dada una función $f : \Sigma^* \to \Sigma^*$ diremos que M calcula f si $f = \varphi_M$.

Ejercicio. Diseñar máquinas de Turing que calculen la función identidad y la función sucesor para números naturales codificados en binario.

Cada máquina de Turing calcula una función. ¿Cómo se comparan estas funciones con las calculables? La respuesta es el siguiente teorema.

Teorema 6.7 Una función es calculable si y sólo si existe una máquina de Turing que la calcula.

Dem. (idea)
Para la implicación de derecha a izquierda, dada una máquina de Turing M es fácil construir un programa que la simule. Dicho programa calculará la misma función que M.
Para la otra implicación se utiliza la caracterización de las funciones calculables como funciones recursivas de Gödel y Kleene. ∎

6.3. El λ-cálculo de Church

El λ-cálculo es un modelo en el que se trabaja con expresiones que se transforman mediante una única operación, la reescritura.

Definición 6.8 Sea A un conjunto finito, el conjunto de las variables. Llamamos *λ-expresión* a e, una palabra sobre $A \cup \{\lambda, [,]\}$ que cumple una de las siguientes condiciones:

- $e \in A$,
- $e = [f\ g]$, donde f y g son λ-expresiones,
- $e = \lambda a.f$, donde $a \in A$ y f es una λ-expresión.

Definición 6.9 Dada una λ-expresión $e = [\lambda x.P\ \ Q]$, *e se reescribe en R* si R es el resultado de sustituir x por Q cada vez que aparezca x en la expresión P. Lo denotaremos

$$[\lambda x.P\ \ Q] \ \rightarrow\ R$$

Ejemplos 6.10
- $[\lambda x.x + 2\ 5] \ \rightarrow\ 5 + 2$
- $[\lambda x.y\ 3] \ \rightarrow\ y$

- $[\lambda x.[x\ x]\ \lambda x.[x\ x]] \ \rightarrow\ [\lambda x.[x\ x]\ \lambda x.[x\ x]]$
- $[\lambda x.[\lambda y.x + y\ 3]\ 2] \ \rightarrow\ [\lambda y{,}2 + y\ 3] \ \rightarrow\ 2 + 3$

Intuitivamente, interpretamos una expresión M como un programa, y ejecutar M con una entrada N es reescribir iteradamente la expresión $[M\ N]$. La ejecución termina cuando no se puede seguir reescribiendo, es decir, cuando no aparece [λ
Algunas expresiones nunca terminan de reescribirse, como la del ejemplo, $[\lambda x.[x\ x]\ \lambda x.[x\ x]]$
Nota: (Sobre los nombres de las variables) Una λ-expresión será *ilegal* si contiene una subexpresión de la forma $[P\ Q]$ de forma que tanto P como Q contienen una misma variable x pero P contiene λx mientras que Q no (o viceversa). Sólo admitiremos expresiones donde esto no ocurre, las que llamaremos *legales*.
Un ejemplo de expresión ilegal es $[\lambda x.y\ \lambda u.[x\ u]]$, la x aparece precedida de λ en $\lambda x.y$ y "libre" en $\lambda u.[x\ u]$.
Podemos transformar expresiones ilegales en otras legales "similares" renombrando en la subexpresión correspondiente variables que aparecen precedidas de λ. (El ejemplo anterior pasa a $[\lambda w.y\ \lambda u.[x\ u]]$).
Tampoco admitiremos λ-expresiones que contengan una subexpresión de la forma $[\lambda x.P\ Q]$ de forma que P contiene λx. Por ejemplo no admitiremos $[\lambda x.[\lambda x.ya]\ [uv]]$

Para hablar de funciones que pueden calcularse en este modelo, primero hemos de definir las expresiones que representan los naturales, que se denominan *naturales de Church:*
$0 \ \equiv\ \lambda f.\lambda x.x$
$1 \ \equiv\ \lambda f.\lambda x.[f\ x]$
$2 \ \equiv\ \lambda f.\lambda x.[f\ [f\ x]]$
$3 \ \equiv\ \lambda f.\lambda x.[f\ [f\ [f\ x]]]$
$n \ \equiv\ \lambda f.\lambda x.[f\ [f\ [f\ [f \ldots\ [f\ x] \ldots]$ ([f aparece n veces)

Definición 6.11 Dada una λ-expresión M, $\varphi_M : \mathbb{N} \rightarrow \mathbb{N}$ es la función definida como

$$\varphi_M(n) = m \text{ si } [M\ n] \text{ se reescribe a } m$$

donde dentro de las expresiones, m y n denotan el correspondiente natural de Church.

Definición 6.12 Una función $f : \mathbb{N} \to \mathbb{N}$ es λ-calculable si existe una λ-expresión M tal que $\varphi_M = f$.

Por ejemplo, la siguiente expresión calcula la función sucesor:
$succ \equiv \lambda n.\lambda y.\lambda z.[y\ [[n\ y]\ z]]$
$[succ\ 1] \equiv$
$[\lambda n.\lambda y.\lambda z.[y\ [[n\ y]\ z]]\ \lambda f.\lambda x.[f\ x]] \to$
$\lambda y.\lambda z.[y\ [[\lambda f.\lambda x.[f\ x]\ y]\ z]] \to$
$\lambda y.\lambda z.[y\ [\lambda x.[y\ x]\ z]] \to$
$\lambda y.\lambda z.[y\ [y\ z]] \equiv 2$

Un programa que no para nunca:
$M \equiv \lambda y.[\lambda x.[x\ x]\ \lambda x.[x\ x]]$
$[M\ a] \to [\lambda x.[x\ x]\ \lambda x.[x\ x]] \to [\lambda x.[x\ x]\ \lambda x.[x\ x]]\ \ldots$

Las siguientes expresiones representan los valores true y false.
$true \equiv \lambda x.\lambda y.x$
$false \equiv \lambda x.\lambda y.y$
$[[true\ P]Q] \equiv [[\lambda x.\lambda y.x\ P]Q] \to [\lambda y.P\ Q] \to P$
$[[false\ P]Q] \equiv [[\lambda x.\lambda y.y\ P]Q] \to [\lambda y.y\ Q] \to Q$

La función not:
$not \equiv \lambda x.[[x\ false]\ true]$
$[not\ true] \equiv [\lambda x.[[x\ false]\ true]\ true] \to$
$[[true\ false]\ true] \to false$
$[not\ false] \equiv [\lambda x.[[x\ false]\ true]\ false] \to$
$[[false\ false]\ true] \to true$

La función if:
$if \equiv \lambda c.\lambda p.\lambda q.[[c\ p]\ q]$

Ejercicio. Reescribir $[[[if\ A]\ B]\ C]$ donde A es la expresión true o la expresión false.
Ejercicio. Averiguar qué funciones calculan las siguientes expresiones:
$zerop \equiv \lambda n.[[n\ [true\ false]]\ true]$
$mifuncion \equiv \lambda x.[[[if\ [zerop\ x]]\ [succ\ x]]\ [[*\ x]\ x]]$

Cada λ-expresión calcula una función. El siguiente teorema nos dice que se trata de las funciones calculables.

Teorema 6.13 Una función es calculable si y sólo si es λ-calculable.

Dem. (idea)
Para la implicación de derecha a izquierda, dada una expresión M es fácil construir un programa que con entrada n haga la reescritura iterada de $[M\ n]$. Dicho programa calculará la misma función que M.
Para la otra implicación se utiliza la caracterización de las funciones calculables como funciones recursivas de Gödel y Kleene. ∎

6.4. La tesis de Turing-Church

En este capítulo hemos visto el siguiente resultado general:

Teorema 6.14 Los tres modelos vistos en este capítulo dan como funciones calculables las funciones calculables definidas con la máquina de registros RAM.

Hay otros modelos que se han ido proponiendo para caracterizar lo que se puede "computar". Todos ellos han resultado equivalentes. Esto ha dado lugar a la siguiente tesis o conjetura:

Tesis de Turing-Church. Cualquier modelo razonable de computación calcula exactamente las funciones calculables.

La palabra "razonable" hace que esta tesis no sea precisa. El significado concreto es: la comunidad científica, con los conocimientos actuales, opina que no hay ningún modelo de computación más potente que los conocidos (potente en el sentido de calcular más funciones).
En la segunda parte del curso veremos que la situación puede variar ligeramente cuando comparamos la eficiencia de los distintos modelos.

EJERCICIOS

6.1. Para cada una de las siguientes funciones, dar una máquina de Turing que la calcule.

1. Dado $A = \{w \mid w = \#y,\ y \in \{0,1\}^*,\ y = y^R\}$,

$$\begin{aligned} f:\ \{0,1\#\}^* &\to \{0,1,\#\}^* \\ w &\mapsto \ \# \text{ si } w \in A \\ w &\mapsto \ \text{indefinido, en otro caso} \end{aligned}$$

2. Dado $A = \{0^n 1^n \mid n \in \mathbb{N}\}$,

$$\begin{aligned} f : \{0,1\}^* \to & \quad \{0,1\}^* \\ w \mapsto & \quad 1 \text{ si } w \in A \\ w \mapsto & \quad 0 \text{ en otro caso} \end{aligned}$$

3. Dado $A = \{0^n 1^n 0^n \mid n \in \mathbb{N}\}$,

$$\begin{aligned} f : \{0,1\}^* \to & \quad \{0,1\}^* \\ w \mapsto & \quad 1 \text{ si } w \in A \\ w \mapsto & \quad 0 \text{ en otro caso} \end{aligned}$$

4. Dado $A = \{w \mid w \in \{0,1\}^*, |w|_0 = |w|_1\}$,

$$\begin{aligned} f : \{0,1\}^* \to & \quad \{0,1\}^* \\ w \mapsto & \quad 1 \text{ si } w \in A \\ w \mapsto & \quad 0 \text{ en otro caso} \end{aligned}$$

6.2. Demostrar que las siguientes funciones son funciones recursivas de Gödel y Kleene (f.r.g.)

1. $f(n,m) = \text{mín}(n,m)$ (Usar la función $x \dot{-} y$ de los Ejemplos *6.3.*)
2. $f(n,m) = \text{máx}(n,m)$
3. $f(n) = n!$
4. $f(n,m) = \text{m.c.d.}(n,m)$. (Usar el algoritmo de Euclides.)
5. La función f definida como

$$\begin{aligned} f(0) = & \quad 1 \\ f(1) = & \quad 1 \\ f(n+2) = & \quad f(n) + f(n+1) \end{aligned}$$

Capítulo 7

Complejidad y codificación

Referencia: Capítulos 1 y 2 de [GJ78]

En la primera parte del curso nos hemos ocupado de saber si un problema dado se puede resolver o no con un algoritmo. En ningún momento nos hemos preocupado de si un algoritmo que resuelve un problema concreto es eficiente o no. De esto se va a ocupar el resto de este curso: queremos saber qué problemas se pueden resolver con un algoritmo eficiente, lo que en la actualidad se identifica con un algoritmo que tarda tiempo polinómico en el tamaño de la entrada.

7.1. El problema del viajante

Continuamos la presentación del tema utilizando el siguiente problema concreto.

> Dado un número $n \in \mathbb{N}$ que representa el número de ciudades, y $n \times n$ números $d(i,j) \in \mathbb{N}$ para $1 \leq i, j \leq n$ que representan las distancias entre cada dos ciudades (y tales que $d(i,j) = d(j,i)$ y $d(i,i) = 0$ para todo i, j).
>
> ¿Cuánto mide el camino más corto que pasa por todas las ciudades una sola vez?

Este problema se llama problema del viajante. Puede resultar sorpendente saber que no se conoce ningún algoritmo que resuelva completamente el problema y que sea sustancialmente mejor que la búsqueda

exhaustiva, es decir, probar todos los caminos y quedarse con el más corto. Como para n ciudades hay $n!$ caminos, y $n! \approx n^n$, este método es bastante lento.
El alumno puede intentar encontrar un algoritmo para el problema del viajante que trabaje en tiempo menor que $n!$ para todas las entradas.
Vamos a estudiar este problema y muchos otros del mismo tipo para los que no se conocen algoritmos mínimamente eficientes.
Nos vamos a centrar en problemas decisionales porque son más sencillos de formalizar. Una versión decisional del problema del viajante es la siguiente, que llamaremos TSP:

> Dados $n \in \mathbb{N}$, $d(i,j) \in \mathbb{N}$ para $1 \leq i,j \leq n$ (tales que $d(i,j) = d(j,i)$ y $d(i,i) = 0$ para todo i,j), y $k \in \mathbb{N}$.
>
> ¿Existe un camino que pasa por todas las ciudades una sola vez y que tiene una longitud total menor o igual que k?

El pasar de la versión funcional a la versión decisional en este caso puede parecer una gran simplificación pero no lo es tanto si tenemos en cuenta que tampoco se conocen algoritmos eficientes para TSP, y que en realidad la complejidad de TSP es similar a la del problema general, ya que a partir de un algoritmo q que resuelva TSP podemos resolver el problema del viajante usando búsqueda dicotómica:

```
Leer N, D
  L_SUP:=∑_{I=1}^{N-1} D(I,I+1);
  L_INF:=0;
  Mientras que L_INF<L_SUP hacer
    Si q(N,D,(L_SUP+L_INF)/2)=SI
      entonces L_SUP:=(L_SUP+L_INF)/2
      else L_INF:=(L_SUP+L_INF)/2
    Fsi;
  Fmq;
  Devuelve L_INF.
```

El tiempo que tarda este algoritmo con entrada n, d es esencialmente el tiempo que tarda q multiplicado por el logaritmo de $\sum_{i=1}^{n-1} \mathrm{d(i,i+1)}$.
Para muchos problemas de búsqueda existe una versión decisional cuya complejidad es similar a la del problema original, igual que en el caso del problema del viajante.

7.2. Complejidad en tiempo

Definición 7.1 Dado un algoritmo p y una entrada x, $\mathrm{t}_p(x)$ es el número de pasos que tarda p con entrada x.

Falta concretar qué consideramos un paso. Un paso es una instrucción de alto nivel, excluídas las multiplicaciones. Multiplicar dos enteros a y b consideraremos que cuesta $\log(a) + \log(b)$ pasos.
De esta forma podemos asegurar que si y es el resultado de p con entrada x, entonces $|y| \leq |x| \cdot \mathrm{t}_p(x)$.
En la sección 2.1.2 hemos definido qué es el tamaño de una entrada. Vamos a medir la complejidad en tiempo de un programa p en función de la longitud de las entradas, según la siguiente definición de $\mathrm{T}_p(m)$:

Definición 7.2 Dado un algoritmo p y $m \in \mathbb{N}$, $\mathrm{T}_p(m)$ es el número de pasos que tarda p con una entrada de tamaño m en el caso peor:

$$\mathrm{T}_p(m) = \max_{|x|=m} \mathrm{t}_p(x)$$

Tanto para t_p como para T_p nos interesan las cotas superiores, es decir,

$$\mathrm{T}_p(m) \leq m^2$$

que quiere decir que para cualquier entrada de tamaño m, p tarda tiempo como mucho m^2.

7.3. Cómo codificamos las entradas

En esta segunda parte del curso trataremos con datos de diferentes tipos, entre ellos grafos. Vamos a detallar algunas de las codificaciones que utilizaremos y ver cómo podemos acotar el tiempo de un algoritmo en función del tamaño de la entrada.
Un grafo $G = (V, A)$ es un conjunto de vértices o nodos V y un conjunto de aristas A. Si G es un grafo dirigido, las aristas son pares de vértices ($A \subseteq V \times V$), es decir, la arista (u, v) es distinta de la (v, u). Si G es un grafo no dirigido, cada arista es un conjunto de dos vértices $\{u, v\}$, es decir, las aristas no tienen dirección ($\{u, v\} = \{v, u\}$). Para un grafo de n vértices tomaremos siempre como conjunto de vértices $V = \{1, 2, 3, \ldots, n\}$.

Codificaremos los grafos (dirigidos y no dirigidos) de 3 formas distintas:

1. Con la matriz de adyacencia.
2. Con listas de adyacencia.
3. Con la lista de aristas.

1. Dado un grafo de n vértices, la matriz de adyacencia es una matriz M $n \times n$ con valores en $\{0, 1\}$ definida como sigue:

   para grafos dirigidos

   $$M(i,j) = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{si } (i,j) \notin A \end{cases}$$

   para grafos no dirigidos

   $$M(i,j) = \begin{cases} 1 & \text{si } \{i,j\} \in A \\ 0 & \text{si } \{i,j\} \notin A \end{cases}$$

   La codificación de cada grafo G se realizará sobre $\{0, 1\}$ y consistirá en la matriz de adyacencia de G escrita por filas, es decir, la codificación de un grafo de n vértices será:

   $$M[1,1]M[1,2]\dots M[1,n]\dots M[n,n]$$

   y por tanto $|G| = n^2$.

   Dado un grafo G, el tamaño de G con esta primera codificación sólo depende del número de vértices.

   Notemos que en el caso de grafos no dirigidos la matriz de adyacencia tiene información redundante, ya que siempre $M[i,j] = M[j,i]$.

2. Dado un grafo G y un vértice i, la lista de adyacencia de i está formada por los vértices j tales que $(i,j) \in A$ (en el caso dirigido) ó $\{i,j\} \in A$ (en el caso no dirigido). Vamos a codificar los grafos utilizando las listas de adyacencia.

   Los codificamos sobre $\{0, 1, \#, ;\}$. Para cada vértice incluímos el vértice en binario, su lista de adyacencia con los vértices en binario separados por # y por último ; para separarlo de la siguiente lista, es decir:

   $$1\#b_1^1\#b_1^2\#\dots\#b_1^{x_1};2\#b_2^1\#\dots\#b_n^{x_n};$$

donde $b_i^1, \ldots, b_i^{x_i}$ es la lista de adyacencia de vértice i.

El tamaño de G depende ahora del tamaño de cada lista de adyacencia y del número de vértices. Si tenemos un grafo G con n vértices y k aristas, podemos acotar superiormente $|G|$ con

$$|G| \leq n(\log(n) + 2 + k(\log(n) + 2))$$

(para cada vértice aparece como máximo su número ($\leq \log(n) + 1$ bits) y hasta k elementos en su lista de adyacencia, cada uno un máximo de $\log(n) + 2$ símbolos).

Podemos acotar inferiormente $|G|$ con

$$|G| \geq 2n + 2k$$

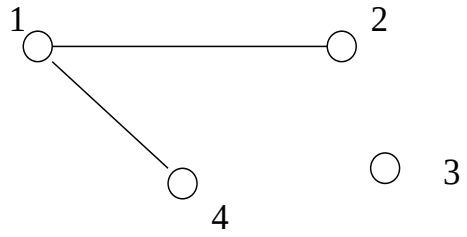
(para cada vértice aparece al menos un bit con su número y el separador ; y por cada arista aparece un elemento de una lista de adyacencia lo cual es al menos un bit con un número y el separador #).

3. Podemos codificar un grafo dirigido G con las aristas (a_1, b_1) $(a_2, b_2) \ldots (a_k, b_k)$ (o un grafo no dirigido G con las aristas $\{a_1, b_1\}$ $\{a_2, b_2\} \ldots \{a_k, b_k\}$) sobre $\{0, 1, \#, (,)\}$ simplemente dando el número de vértices en binario, seguido de la lista de aristas, escribiendo los números de los vértices en binario:

$$n(a_1 \# b_1)(a_2 \# b_2) \ldots (a_k \# b_k)$$

De esta forma $|G| \leq \log(n) + 1 + k(2\log(n) + 5)$ y $|G| \geq \log(n) + 1 + 5k$.

Ejemplo 7.3 Sea G el siguiente grafo no dirigido:



Su codificación según 1. será

$$0101100000001000$$

según 2.

$$1\#10\#100; 10\#1; 11; 100\#1;$$

y según 3.

$$100(1\#10)(1\#100)$$

7.4. Transformación de cotas de tiempo

Hemos visto que la codificación concreta que se elige influye de forma importante sobre el tamaño de las entradas. Veamos ahora cómo podemos pasar de una cota dada a un algoritmo en función de una parte de la entrada a una en función del tamaño de la entrada.
Sean p y q dos algoritmos que resuelven un problema con entrada un grafo. Dado un grafo G, llamamos n al número de vértices y k al número de aristas. Supongamos que tenemos las siguientes dos cotas para la complejidad en tiempo de p y q:

$$\begin{aligned} \mathrm{t}_p(G) &\leq 2n^3 + 6n \\ \mathrm{t}_q(G) &\leq 3k^2 \end{aligned}$$

Veamos cómo transformar las cotas anteriores en otras cotas que dependen sólo del tamaño de la entrada. Para ello necesitamos desigualdades de la forma $n \leq f_1(|G|)$ ó $k \leq f_2(|G|)$ para combinarlas con las dos desigualdades anteriores.

1. Primera codificación. Sabemos que $|G| \geq n^2$, luego $n \leq \sqrt{|G|}$ y por tanto

$$\begin{aligned} \mathrm{T}_p(m) &= \text{máx}_{|G|=m}\, \mathrm{t}_p(G) \leq \text{máx}_{|G|=m}\, 2n^3 + 6n \leq \\ &\leq \text{máx}_{|G|=m}\, 2|G|^{3/2} + 6|G|^{1/2} \leq 2m^{3/2} + 6m^{1/2} \end{aligned}$$

   Para el caso de q, cuyo tiempo tenemos acotado en función del número de aristas, sabemos que $k \leq n^2$, luego $|G| \geq n^2 \geq k$, por tanto

$$\mathrm{T}_q(m) = \max_{|G|=m} \mathrm{t}_q(G) \leq \max_{|G|=m} 3k^2 \leq \max_{|G|=m} 3|G|^2 = 3m^2$$

2. Segunda codificación. Sabemos que $|G| \geq 2n + 2k$ luego $|G| \geq k$ y $|G| \geq n$. Por tanto

   $$\mathrm{T}_p(m) = \max_{|G|=m} \mathrm{t}_p(G) \leq \max_{|G|=m} 2n^3{+}6n \leq \max_{|G|=m} 2|G|^3{+}6|G| \leq 2m^3{+}6m$$

   De la misma forma

   $$\mathrm{T}_q(m) \leq 3m^2$$

3. Tercera codificación. Sabemos que $|G| \geq \log(n) + 1 + 5k \geq \log(n)$. Por tanto $n \leq 2^{|G|}$. No podemos dar una cota inferior de $|G|$ en función de n mucho mejor, porque por ejemplo en el caso de un grafo de n vértices y ninguna arista, $|G| = \log(n) + 1$ y $n = 2^{|G|-1}$. Por tanto

   $$\begin{aligned}\mathrm{T}_p(m) = \quad & \max_{|G|=m} \mathrm{t}_p(G) \leq \max_{|G|=m} 2n^3 + 6n \\ & \leq \max_{|G|=m} 2^{3|G|+1} + 6 \cdot 2^{|G|} \leq 2^{3m+1} + 6 \cdot 2^m\end{aligned}$$

   Para el caso de q, sabemos que $|G| \geq k$, luego

   $$\mathrm{T}_q(m) \leq 3m^2$$

Como vemos, para pasar de una cota superior de $\mathrm{t}_p(x)$ que depende de una parte de la entrada z a una cota superior de $\mathrm{T}_p(m)$ en función del tamaño de la entrada $|x|$ el método es:

- Hacer una cota de la forma $|x| \geq g(z)$.
- Pasar a $f(|x|) \geq z$.
- Acotar $\mathrm{T}_p(m)$ usando la cota de $\mathrm{t}_p(x)$ que depende de z y el hecho de que $z \leq f(|x|)$.

Capítulo 8

Tiempo polinómico versus tiempo exponencial

Referencia: Capítulos 1 y 2 de [GJ78].

En este capítulo estudiaremos P y EXP, dos clases o conjuntos de problemas decisionales.
La clase EXP contiene casi todos los problemas que intentaréis resolver con un algoritmo. P es una parte de EXP formada por los problemas que se pueden resolver eficientemente o resolubles en la práctica.

8.1. Definiciones

Definición 8.1 Dada una función $f : \mathbb{N} \to \mathbb{N}$, llamamos $O(f)$ al conjunto

$$O(f) = \{h : \mathbb{N} \to \mathbb{N} \mid \exists c > 0 \text{ tal que } h(m) \leq c \cdot f(m) \ \forall m\}$$

(es decir, $O(f)$ es el conjunto de funciones acotadas por $c \cdot f$, para alguna constante c).

Vamos a clasificar los problemas decisionales según el tiempo que se tarda en resolverlos en función del tamaño de la entrada y en el caso peor, es decir, según T_p.

Definición 8.2 Dada una función $f : \mathbb{N} \to \mathbb{N}$ llamamos
DTIME($f(m)$) a la clase de problemas:

$$\mathrm{DTIME}(f(m)) = \{\, \Pi \mid \begin{array}{l} \Pi \text{ es un problema decisional} \\ \text{y existe un algoritmo } q \text{ que lo} \\ \text{resuelve y cumple } \mathrm{T}_q \in \mathrm{O}(f) \,\}. \end{array}$$

Es decir, DTIME($f(m)$) es el conjunto de problemas resolubles en tiempo menor o igual que $c \cdot f$, para alguna constante c.
Es importante notar que f es una cota superior, un problema que está en DTIME($f(m)$) puede tener un programa que lo resuelva en tiempo mucho menor que $f(m)$.

Definición 8.3 P es el conjunto de problemas resolubles en tiempo (menor o igual que) polinómico, es decir

$$\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{DTIME}(m^k)$$

EXP es el conjunto de problemas resolubles en tiempo (menor o igual que) exponencial, es decir

$$\mathrm{EXP} = \bigcup_{k \in \mathbb{N}} \mathrm{DTIME}(2^{m^k})$$

Insistimos en que en DTIME(2^{m^k}), 2^{m^k} es cota *superior*, no tiene porque ser igualdad, el tiempo de un problema en DTIME(2^{m^k}) puede ser mucho menor que 2^{m^k}.

Teorema 8.4 P $\subseteq$ *EXP*

Dem. Para cualquier k, sabemos que $m^k \in \mathrm{O}(2^m)$ (ya que $\lim_m m^k/2^m = 0$), luego DTIME(m^k) $\subseteq$ DTIME(2^m) y por tanto P $\subseteq$ EXP. ∎

Se sabe que P $\neq$ EXP (luego P $\subset$ EXP). Esto quiere decir que hay más problemas en EXP que los de P, es decir, hay algún problema que se puede resolver en tiempo acotado por una exponencial pero no por un polinomio.

8.2. Problemas resolubles en la práctica

P es el conjunto de problemas considerados resolubles en la práctica. Esto quiere decir que si un problema Π no está en P se considera no resoluble de forma eficiente.
Las razones de P se considere el límite de lo resoluble de forma eficiente son de índole práctico y se resumen en:

1. La mayoría de los algoritmos q que se implementan cumplen $\mathrm{T}_q(m) \in \mathrm{O}(m^k)$ para algún k.

2. Los problemas naturales que se sabe que están en P tienen algoritmos "rápidos" (que cumplen $\mathrm{T}_q(m) \leq c \cdot m^k$ para $k \leq 3$ y c pequeña), luego se pueden resolver en tiempo polinómico pero para polinomios muy pequeños.

3. Los problemas naturales para los que no se conocen algoritmos polinómicos, tampoco tienen algoritmos conocidos con tiempo muy por debajo de una exponencial 2^{cm}, es decir, no se conocen algoritmos para problemas interesantes con tiempos intermedios como $m^{\log(m)}, m^{\log(m)^2}$, etc. Por tanto en la práctica se trata de comparar algoritmos que tardan tiempo m^k con algoritmos que tardan tiempo al menos 2^m. Basta examinar las tablas siguientes para ver que los algoritmos que tardan tiempo 2^m ó más son inútiles en la práctica.

Considerando la velocidad de un pentium 4 (cada instrucción tarda alrededor de 1,5 $\times 10^{-9}$ segundos, suponiendo una velocidad de 2,4 GHz y 4 ciclos por instrucción) veamos en la siguiente tabla cuánto valen las funciones de tiempo m, m^2, m^3, m^5, 2^m y 3^m para distintos valores de m.

$m =$	10	20	30	40	50	60
m	$1{,}5\times10^{-8}$ seg.	3×10^{-8} seg.	$4{,}5\times10^{-8}$ seg.	6×10^{-8} seg.	$7{,}5\times10^{-8}$ seg.	9×10^{-8} seg.
m^2	$1{,}5\times10^{-7}$ seg.	6×10^{-7} seg.	$1{,}35\times10^{-6}$ seg.	$2{,}4\times10^{-6}$ seg.	$3{,}75\times10^{-6}$ seg.	$5{,}4\times10^{-6}$ seg.
m^3	$1{,}5\times10^{-6}$ seg.	$1{,}2\times10^{-5}$ seg.	4×10^{-5} seg.	$9{,}6\times10^{-5}$ seg.	$1{,}9\times10^{-4}$ seg.	$3{,}2\times10^{-4}$ seg.
m^5	$1{,}5\times10^{-4}$ seg.	0,0048 seg.	0,036 seg.	0,15 seg.	0,47 seg.	1,17 seg.
2^m	$1{,}5\times10^{-6}$ seg.	0,0015 seg.	1,611 seg.	27,6 minutos	19 días	59,4 años
3^m	$8{,}8\times10^{-5}$ seg.	5,2 seg.	3 días	5,8 siglos	3×10^5 siglos	2×10^{10} siglos

A continuación vemos el mayor tamaño de entrada resoluble en una hora:

tiempo	computador de hoy	100 veces más rápido	1000 veces más rápido
m	$N_1 = 24 \cdot 10^{11}$	$100N_1$	$1000N_1$
m^2	$N_2 = 1550000$	$10N_2$	$31{,}6N_2$
m^3	$N_3 = 13200$	$4{,}64N_3$	$10N_3$
m^5	$N_4 = 300$	$2{,}5N_4$	$3{,}98N_4$
2^m	$N_5 = 41,1$	N_5+6,64	N_5+9,97
3^m	$N_6 = 25,9$	N_6+4,19	N_6+6,29

Esta tabla presenta el efecto de la mejora tecnológica el varios algoritmos de tiempo polinómico y exponencial.

En los siguientes capítulos nos ocuparemos de muchos problemas que es importante resolver eficientemente, ya que aparecen en todo tipo de aplicaciones, pero para los cuales no se conocen algoritmos eficientes, es decir, no se sabe que estén en P.

Terminamos señalando que existen algunos (pocos) problemas para los que el mejor algoritmo conocido tarda tiempo exponencial en el caso peor ($\mathrm{T}_p(m) \approx 2^m$) pero que funcionan bien en la práctica, por ejemplo el problema de la mochila. Esto es debido a que la definición de $\mathrm{T}_p(m)$ es en caso peor, pero para las entradas más frecuentes $\mathrm{t}_p(x)$ se mantiene bajo y son entradas menos usadas las que hacen $\mathrm{T}_p(m)$ exponencial. Este comportamiento anómalo se da para muy pocos problemas.

8.3. Tesis extendida de Turing-Church

Hemos estudiado distintos modelos de cálculo, que son equivalentes respecto a la definición de *función calculable* (y por tanto conjunto decidible, que corresponde a problema decisional resoluble). Vamos a comparar ahora estos modelos respecto a la definición de las clases P y EXP.
Tomando el modelo de las máquinas de Turing, cada máquina calcula una función a base de acciones elementales o transiciones, cada una de ellas consistente en cambiar de estado, escribir un símbolo y moverse una casilla. Llamamos paso a cada una de estas transiciones.

Definición 8.5 Dada una máquina de Turing M defino t_M y T_M como:

$$\begin{aligned} \mathrm{t}_M(x) &= \text{Número de pasos de } M \text{ con entrada } x \text{ desde que empieza hasta que se para.} \\ \mathrm{T}_M(m) &= \max_{|x|=m} \mathrm{t}_M(x). \end{aligned}$$

Ahora podemos definir clases de problemas decisionales clasificándolos a partir de T_M:

Definición 8.6 Dada una función $f : \mathbb{N} \to \mathbb{N}$, llamamos $\mathrm{DTIME}_1(f(m))$ a la clase:

$$\mathrm{DTIME}_1(f(m)) = \{\, \Pi \mid \begin{array}{c} \Pi \text{ es un problema decisional} \\ \text{y existe una máquina } M \text{ que lo} \\ \text{resuelve y cumple } \mathrm{T}_M \in \mathrm{O}(f) \,\}. \end{array}$$

El siguiente resultado técnico relaciona las clases DTIME y DTIME_1:

Lema 8.7 Dada una función $f : \mathbb{N} \to \mathbb{N}$ que sea total, creciente y recursiva (por ejemplo $f(m) = m^k$ y $f(m) = 2^{m^k}$)

$$\begin{aligned} DTIME_1(f(m)) &\subseteq DTIME(f^3(m)) \\ DTIME(f(m)) &\subseteq DTIME_1(f^3(m)) \end{aligned}$$

Omitimos la demostración, que requiere manejo avanzado de las máquinas de Turing.
Por tanto podemos definir P y EXP utilizando máquinas de Turing:

Corolario 8.8

$$\begin{aligned} \mathrm{P} &= \bigcup_{k\in\mathbb{N}} DTIME_1(m^k) \\ EXP &= \bigcup_{k\in\mathbb{N}} DTIME_1(2^{m^k}) \end{aligned}$$

Dem. Por el lema anterior, para cada $k \in \mathbb{N}$,

$$\mathrm{DTIME}_1(m^k) \subseteq \mathrm{DTIME}(m^{3k}) \subseteq \mathrm{P}$$

luego

$$\bigcup_{k\in\mathbb{N}} \mathrm{DTIME}_1(m^k) \subseteq \mathrm{P}$$

También para cada $k \in \mathbb{N}$,

$$\mathrm{DTIME}(m^k) \subseteq \mathrm{DTIME}_1(m^{3k})$$

luego

$$\mathrm{P} = \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}(m^k) \subseteq \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}_1(m^k)$$

■

En general, para cada uno de los modelos conocidos con una definición natural de paso, P y EXP corresponden a tiempo polinómico y exponencial respectivamente. Por ejemplo, en el λ-cálculo, un paso es una aplicación de la regla de reescritura.
Esta situación ha motivado la siguiente tesis o conjetura:

Tesis extendida de Turing-Church: Para cualquier modelo *razonable* (y secuencial) de cálculo, P y EXP corresponden a número de pasos polinómico y exponencial respectivamente.

La conjetura anterior está siendo replanteada a la luz de los recientes estudios sobre *el computador cuántico*.
Este modelo, formulado en 1982 por Deutsch, Lloyd y Feynman es de naturaleza muy distinta a los otros modelos secuenciales.
En 1994, Nishino resolvió con este modelo en tiempo polinómico problemas para los que no se conocen algoritmos polinómicos. También Shor

en el 94 provó que es posible factorizar números en tiempo medio polinómico con un computador cuántico, lo cual hasta el momento no se ha conseguido con los computadores tradicionales.
Si embargo, a pesar de los múltiples intentos de varios grupos de investigadores, no se ha construído todavía un computador cuántico, y todavía no está claro si será viable construirlo en el futuro.

EJERCICIOS

8.1. Demostrar que los siguientes problemas están en la clase P.

1. Mochila-fácil:
   Datos: $n, p_1, \ldots, p_n, k, C \in \mathbb{N}$
   Salida: ¿Existe $A \subseteq \{1, \ldots, n\}$ con $\#A = k$ y

   $$\Sigma_{i \in A} p_i \leq C \ ?$$

   Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, los $n+3$ números naturales que componen una entrada se escriben en binario y se separan por comas.
   Pista: Ordenar $p_1, \ldots, p_n$.
2. 2-color:
   Datos: $G = (V, A)$ grafo dirigido.
   Salida: ¿Pueden etiquetarse los vértices de G con dos colores de manera que los dos vértices de cada arista tengan colores diferentes?
   Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $u \in \{0,1\}^*$ es el número de vértices escrito en binario y $v \in \{0,1\}^*$ es la matriz de adyacencia de G escrita por filas, entonces la codificación de la entrada G es la palabra u, v.
   Pista: Es equivalente a dividir V en dos conjuntos V_1 y V_2 de manera que no haya ninguna arista de un vértice de V_1 a otro de V_1, ni de uno de V_2 a otro de V_2.
3. Path-bet-two-vertices:
   Datos: $G = (V, A)$ grafo dirigido, $u, v \in V$
   Salida: ¿Existe un camino de u a v?

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $a \in \{0,1\}^*$ es el número de vértices escrito en binario, $b \in \{0,1\}^*$ es la matriz de adyacencia de G escrita por filas, y x, y son u, v en binario, entonces la codificación de la entrada G, u, v es la palabra a, b, x, y.

Pista: Calcular el conjunto de los vértices que están a distancia menor o igual que n de u de forma incremental.

4. Shortest-Path-bet-two-vertices:
   Datos: $G = (V, A)$ grafo dirigido, $u, v \in V$, $k \in \mathbb{N}$
   Salida: ¿Existe un camino de u a v de longitud menor o igual a k?
   Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $a \in \{0,1\}^*$ es el número de vértices escrito en binario, $b \in \{0,1\}^*$ es la matriz de adyacencia de G escrita por filas, x, y son u, v en binario, y z es k en binario, entonces la codificación de la entrada G, u, v, k es la palabra a, b, x, y, z.
5. Modulo:
   Datos: $a, b, c \in \mathbb{N}$
   Salida: ¿Existe un $x \in \mathbb{N}$ tal que $x < c$ y $x \equiv a (\mathrm{mod} b)$?
   Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, los 3 números naturales que componen una entrada se escriben en binario y se separan por comas.

Capítulo 9

Estudio de algunos problemas importantes; la clase NP

Referencia: Capítulos 2.6 y 3.1 de [GJ78].

En este capítulo estudiaremos tres problemas importantes: SAT, MOCHILA y CLIQUE, y definiremos la clase NP a la que pertenencen esos tres problemas.

9.1. SAT

Sea $X = \{x_1, \ldots, x_n\}$ un conjunto de variables booleanas.

Definición 9.1 Un *literal* sobre X es una variable $x \in X$ o su negación $\neg x$.

Definición 9.2 Una *cláusula* sobre X es una disyunción de literales.

Ejemplos 9.3 Sea $n = 8$. Son cláusulas sobre X:

- $x_2 \vee \neg x_3 \vee x_8$
- $\neg x_2$

Definición 9.4 Una *fórmula CNF* sobre X es una conjunción de cláusulas.

Ejemplos 9.5 Sea $n = 15$. Son fórmulas CNF las siguientes:

- $(x_7 \vee x_{15}) \wedge (x_2 \vee \neg x_3 \vee x_8)$
- $(\neg x_2) \wedge (x_4 \vee \neg x_2)$

Definición 9.6 Una *asignación de verdad* de X es una función $\alpha : X \to \{T, F\}$, es decir, una función que asigna a cada variable el valor T (cierto) o el F (falso).

Definición 9.7 Dada una fórmula L y una asignación de verdad α, α *satisface* L si al sustituir cada variable x por $\alpha(x)$ en L y operar según la definición habitual de los operadores booleanos $\neg, \vee, \wedge$, el resultado es T.

Ejemplos 9.8 $\alpha(x_1) = T, \alpha(x_2) = F, \alpha(x_3) = T, \alpha(x_4) = T$ satisface la fórmula $(\neg x_2) \wedge (x_4 \vee \neg x_2)$.

Ya podemos definir el problema SAT. Restringiremos las entradas a fórmulas en las que aparecen todas las variables de X.

Datos: Un conjunto de variables X y una fórmula CNF sobre X, L (que cumplen que todas las variables de X aparecen al menos una vez en L).

Salida: ¿Existe una asignación de verdad que satisface L?

Es decir, el problema se trata de saber si una cierta fórmula CNF se puede hacer cierta o si por el contrario es falsa para cualquier asignación. Por ejemplo, cualquiera de los ejemplos anteriores son fórmulas para las que existen asignaciones que las satisfacen. No existe ninguna asignación que satisfaga la fórmula $(x_2 \vee x_1) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_2 \vee \neg x_1) \wedge (x_2 \vee \neg x_1) \wedge (x_2 \vee \neg x_2)$.
Tenemos que especificar la codificación de las entradas del problema anterior. Vamos a utilizar el alfabeto $\Sigma = \{0, 1, \neg, (,), \#, \vee, \wedge\}$ y codificar una entrada X, L como el número de variables n en binario, seguido de la fórmula escribiendo los números de variable en binario y los símbolos de las operaciones lógicas necesarios.
Por ejemplo, $X = \{x_1, x_2, x_3\}$, $L = (x_2 \vee \neg x_1) \wedge x_3$ se codifica como

$$11\#(10 \vee \neg 1) \wedge 11$$

Vamos a relacionar el tamaño de una entrada X, L con el número de variables n y el número de cláusulas de la fórmula k:

- Como hemos restringido a las fórmulas en las que aparecen todas las variables de X, y para cada una de las variables que aparece en L hay que escribir al menos un bit con el número de variable, $|X, L| \geq n$.
- Para cada cláusula hay que escribir al menos un símbolo, luego $|X, L| \geq k$.

El algoritmo de búsqueda exhaustiva, es decir, el que con entrada X, L evalúa la fórmula L con cada una de las 2^n asignaciones posibles, tarda tiempo $\mathrm{t}_p(X, L) \leq 2^n \cdot$ ("tiempo de evaluar$''$), veremos más adelante que podemos evaluar L con una determinada asignación en tiempo menor o igual que una constante c por kn. Como $|X, L| \geq n$, $|X, L| \geq k$, $\mathrm{T}_p(m) \leq 2^m(cm^2) \in \mathrm{O}(2^{m^2})$. Por tanto SAT $\in$ DTIME(2^{m^2}) y SAT $\in$ EXP.
Pero ya hemos hablado de la inutilidad práctica de las cotas de tiempo exponenciales. Lo que ocurre es que no se conoce ningún algortimo que resuelva todas las entradas de SAT y tarde tiempo (en caso peor) sustancialmente menor que la búsqueda exhaustiva.
Estudiamos a continuación el problema de evaluación de fórmulas booleanas (EVAL), mucho más simple de resolver que SAT:

Datos: Un conjunto de variables X, una fórmula CNF sobre X, L y una asignación de verdad α (que cumplen que todas las variables de X aparecen al menos una vez en L).

Salida: ¿α satisface L?

La codificación de las entradas de EVAL, sobre el alfabeto
$\Sigma = \{0, 1, \neg, (,), \#, \vee, \wedge\}$, se basa en la codificación de las entradas de SAT, pero tenemos que especificar la codificación de las asignaciones de verdad. Una asignación α sobre n variables la codificaremos con n bits $w_1 \ldots w_n$ de forma que $w_i = 1$ si $\alpha(x_i) = T$ y $w_i = 0$ si $\alpha(x_i) = F$.
La codificación de una entrada X, L, α será la codificación de X, L como entrada de SAT, seguida de $\#$, seguida de la codificación de α descrita anteriormente, luego $|X, L, \alpha| = |X, L| + n + 1$.
Un simple algoritmo para EVAL es el que primero sustituye cada literal de L por su valor correspondiente según α y después va simplificando los operadores booleanos:

Leer X, L, α
/* X= $\{x_1, \ldots, x_n\}$, L= $c_1 \wedge c_2 \wedge \ldots \wedge c_k$,
con c_i clausula para $1 \leq i \leq k$ */
Para I=1 hasta k hacer
Para cada H literal de c_{I} hacer
Si H=x sustituir H por $\alpha(x)$
else Si H=$\neg x$ sustituir H por $\neg\alpha(x)$
/* Sustituyo por T si $\alpha(x)$=F, por F si $\alpha(x)$=T */
Fpara; Fpara;
RESULTADO:=TRUE;
I:=1;
Mientras que (I<= k) AND RESULTADO hacer
ESTACLAUSULA:=FALSE;
Mientras que (NOT ESTACLAUSULA) AND ($c_i \neq \emptyset$) hacer
Quitar CTE la siguiente constante de c_i;
Si CTE=T entonces ESTACLAUSULA:=TRUE;
Fmq;
RESULTADO:=RESULTADO AND ESTACLAUSULA;
I:=I+1;
Si RESULTADO entonces Devuelve SI
else Devuelve NO.

Si p es el algoritmo anterior, como el número de literales por cláusula es como máximo $2n$,

$$\mathrm{t}_p(X, L, \alpha) \leq k \cdot 2n + 2 + k(3 + 4n) + 1 \leq 12kn$$

Como $|X, L, \alpha| \geq |X, L|$, entonces $|X, L, \alpha| \geq n$ y $|X, L, \alpha| \geq k$. Por tanto $\mathrm{T}_p(m) \leq 12m^2$, EVAL $\in$ DTIME$(m^2) \subseteq$ P.
La diferencia de tamaño entre las entradas de SAT y las de EVAL no es muy grande, ya que $|\alpha| = n \leq |X, L|$.
Observemos que

$$\begin{aligned} & X, L \text{ tiene solución Sí para SAT} \\ \Leftrightarrow\ & \exists \alpha, |\alpha| \leq |X, L|\ X, L, \alpha \text{ tiene solución Sí para EVAL} \end{aligned}$$

Como conjuntos,

$$\mathrm{SAT} = \{X, L \mid \exists \alpha, |\alpha| \leq |X, L|\ X, L, \alpha \in \mathrm{EVAL}\}.$$

Esto se puede expresar informalmente como "podemos COMPROBAR SAT en tiempo polinómico", ya que EVAL es el problema de comprobar que un determinado α hace que X, L tenga solución Sí para SAT.

9.2. MOCHILA

Sea MOCHILA el siguiente problema:

Datos: $n \in \mathbb{N}$ el número de objetos,

$p_1, \ldots, p_n \in \mathbb{N}$ los pesos de los objetos,

$P \in \mathbb{N}$ el peso máximo que admite la mochila, y

$d \in \mathbb{N}$ el hueco máximo permitido ($d \leq P$).

Salida: ¿Existe un conjunto de objetos $A \subseteq \{1, \ldots, n\}$ que cumpla:

$$P - d \leq \sum_{i \in A} p_i \leq P \ ?$$

Es decir, ¿existe una forma de llenar la mochila pesando como mínimo d menos que el máximo permitido?

Codificamos la entrada con el alfabeto $\Sigma = \{0, 1, \#\}$, cada natural en binario y separados por #: $n\#p_1\# \ldots \#p_n\#P\#d$
Vamos a relacionar el tamaño de una entrada $n, p_1, \ldots, p_n, P, d$ con el número de objetos n y con $M = \text{máx}\{p_1, \ldots, p_n, P\}$:

- Como hay que escribir $n + 3$ números en binario, $|n, p_1, \ldots, p_n, P, d| \geq n$.
- Como hay que escribir al menos una vez en binario el número M, $|n, p_1, \ldots, p_n, P, d| \geq \log(M)$.

El algoritmo de búsqueda exhaustiva, es decir, el que con entrada $n, p_1, \ldots, p_n, P, d$ calcula el valor de $\sum_{i \in A} p_i$ para cada uno de los 2^n subconjuntos de $\{1, \ldots, n\}$, tarda tiempo $\mathrm{t}_p(n, p_1, \ldots, p_n, P, d) \leq 2^n \cdot n$, ya que podemos calcular una suma de como máximo n naturales en tiempo menor o igual que n. Como $|n, p_1, \ldots, p_n, P, d| \geq n$, $\mathrm{T}_p(m) \leq 2^m \cdot m \in \mathrm{O}(2^{m^2})$. Por tanto MOCHILA $\in$ DTIME$(2^{m^2}) \subseteq$ EXP.

Para MOCHILA tampoco se conoce ningún algortimo que resuelva todas las entradas y tarde tiempo (en caso peor) sustancialmente menor que la búsqueda exhaustiva.
Estudiamos a continuación una versión muy simplificada de MOCHILA, que llamamos compMOCHILA:

Datos: $n \in \mathbb{N}$, $p_1, \ldots, p_n \in \mathbb{N}$, $P \in \mathbb{N}$, $d \in \mathbb{N}$ $(d \leq P)$ y A un subconjunto de $\{1, \ldots, n\}$.

Salida: ¿Se cumple:

$$P - d \leq \sum_{i \in A} p_i \leq P \ ?$$

La codificación de las entradas de compMOCHILA, sobre el alfabeto $\Sigma = \{0, 1, \#\}$, se basa en la codificación de las entradas de MOCHILA, pero tenemos que especificar la codificación de los subconjuntos de $\{1, \ldots, n\}$. Un subconjunto de $\{1, \ldots, n\}$ lo codificaremos con n bits $w_1 \ldots w_n$ de forma que $w_i = 1$ si $i \in A$ y $w_i = 0$ si $i \notin A$.
La codificación de una entrada $n, \ldots, d, A$ será la codificación de $n, \ldots, d$ como entrada de MOCHILA, seguida de #, seguida de la codificación de A descrita anteriormente, luego $|n, \ldots, d, A| =$
$= |n, \ldots, d| + n + 1$.
Un algoritmo para compMOCHILA es un único bucle que para una entrada $n, p_1, \ldots, p_n, P, d, A$ calcula $\sum_{i \in A} p_i$. Si q es este algoritmo,

$$\mathrm{t}_q(n, p_1, \ldots, p_n, P, d, A) \leq n$$

Como $|n, p_1, \ldots, p_n, P, d, A| \geq |n, p_1, \ldots, p_n, P, d| \geq n$, $\mathrm{T}_q(m) \leq m$, compMOCHILA $\in$ DTIME$(m) \subseteq$ P.
La diferencia de tamaño entre las entradas de MOCHILA y las de compMOCHILA no es muy grande, ya que $|A| =$
$= n \leq |n, \ldots, d|$.
Observemos que

$$n, p_1, \ldots, p_n, P, d \text{ tiene solución Sí para MOCHILA}$$

$$\Leftrightarrow \exists A, |A| \leq |n, p_1, \ldots, p_n, P, d|$$

$$n, p_1, \ldots, p_n, P, d, A \text{ tiene solución Sí para compMOCHILA}$$

Esto se puede expresar informalmente como "podemos COMPROBAR MOCHILA en tiempo polinómico".
También podemos escribir MOCHILA como:

$$\text{MOCHILA} = \{n, p_1, \ldots, p_n, P, d \,|$$

$$\exists A, |A| \leq |n, p_1, \ldots, p_n, P, d| \; n, p_1, \ldots, p_n, P, d, A \in \text{compMOCHILA}\}.$$

9.3. CLIQUE

Sea $G = (V, A)$ un grafo no dirigido.

Definición 9.9 Un clique de G es un conjunto de vértices $U \subseteq V$ que forma un subgrafo completo, es decir, que cumple

$$\forall u, v \in U, u \neq v \; \{u, v\} \in A$$

Esto es, los vértices de U están unidos por todas las aristas posibles.

El problema que tratamos aquí es la búsqueda de cliques lo más grandes posibles. En versión decisional aparece el problema que sigue.
Sea CLIQUE el siguiente problema:

Datos: $G = (V, A)$ un grafo no dirigido con n vértices,

$k \in \mathbb{N}$ con $k \leq n$.

Salida: ¿Existe un clique de G con k vértices?

Codificamos la entrada con el alfabeto $\Sigma = \{0, 1, \#\}$, utilizando la codificación del grafo con matriz de adyacencia, después # seguida de la codificación de k en binario.
Vamos a relacionar el tamaño de una entrada G, k con el número de vértices n, sabemos que $|G| = n^2$, luego $|G, k| \geq n^2$.
El algoritmo de búsqueda exhaustiva, es decir, el que con entrada G, k prueba cada subconjunto de k vértices de G como posible clique, tiene que probar $\binom{n}{k}$ subconjuntos y para cada uno de ellos chequear la existencia de $k(k-1)/2$ aristas. Por tanto tarda tiempo $\mathrm{t}_p(G, k) \leq \binom{n}{k} \cdot k^2 \leq$

$2^n \cdot n^2$. Como $|G,k| \geq n^2$ $\mathrm{T}_p(m) \leq 2^{m^{1/2}} \cdot m \in \mathrm{O}(2^m)$. Por tanto CLIQUE $\in$ DTIME$(2^m) \subseteq$ EXP.
Para CLIQUE tampoco se conoce ningún algoritmo que resuelva todas las entradas y tarde tiempo (en caso peor) sustancialmente menor que la búsqueda exhaustiva.
Estudiamos a continuación una versión comprobación que llamamos compCLIQUE:

Datos: $G = (V, A)$ grafo no dirigido, $k \in \mathbb{N}$, U subconjunto de V de k elementos.

Salida: ¿Es U un clique de G ?

La codificación de las entradas de compCLIQUE, sobre el alfabeto $\Sigma = \{0, 1, \#\}$, se basa en la codificación de las entradas de CLIQUE, pero tenemos que especificar la codificación de los subconjuntos de k elementos de $\{1, \ldots, n\}$. Un subconjunto de k elementos de $\{1, \ldots, n\}$ lo codificaremos listando sus elementos en binario, por orden y separados por #. Por ejemplo $U = \{2, 6, 3\}$ lo codificaremos como $10, 11, 110$.
La codificación de una entrada G, k, U será la codificación de G, k como entrada de CLIQUE, seguida de #, seguida de la codificación de U descrita anteriormente, luego $|G, k, U| \geq |G, k| \geq n^2$.
Un algoritmo para compCLIQUE es un único bucle que para una entrada G, k, U comprueba si todas las parejas u, v con $u, v \in U$ son aristas de G. Si q es el algoritmo anterior,

$$\mathrm{t}_q(G, k, U) \leq k^2 \leq n^2$$

Como $|G, k, U| \geq n^2$, $\mathrm{T}_q(m) \leq m$, compCLIQUE $\in$ DTIME$(m) \subseteq$ P.
La diferencia de tamaño entre las entradas de CLIQUE y las de compCLIQUE no es muy grande, ya que

$$|U| \leq k \cdot (\log(n) + 2) \leq n \cdot (\log(n) + 2) \leq 3 \cdot n^2,$$

$$|G, k| \geq n^2,$$

luego $|U| \leq 3 \cdot |G, k|$.
Observemos que

$$G, k \text{ tiene solución Sí para CLIQUE}$$
$$\Leftrightarrow \exists U, |U| \leq 3 \cdot |G, k| \; G, k, U \text{ tiene solución Sí para compCLIQUE}$$

Esto se puede expresar informalmente como "podemos COMPROBAR CLIQUE en tiempo polinómico".

$$\text{CLIQUE} = \{G, k \mid \exists U, |U| \leq 3 \cdot |G, k| \ G, k, U \in \text{compCLIQUE}\}.$$

9.4. La clase NP

Hemos visto que los tres problemas anteriores, SAT, MOCHILA y CLIQUE, tienen una propiedad en común, son comprobables en tiempo polinómico. Existen muchos problemas que cumplen esta propiedad, son los que forman la clase NP.

Definición 9.10 Dado un problema decisional Π, Π es *comprobable en tiempo polinómico* si existe un problema $\Lambda \in \text{P}$ y una constante c que cumplen, para cualquier x entrada de Π:

$$x \text{ tiene solución Sí para } \Pi \Leftrightarrow$$
$$\exists y, |y| \in \text{O}(|x|^c), \text{ con } x, y \text{ una entrada con solución Sí para } \Lambda.$$

Es decir, como conjunto,

$$\Pi = \{x \mid \exists y, |y| \in \text{O}(|x|^c) \ x, y \in \Lambda\}.$$

Definición 9.11 NP es el conjunto de problemas comprobables en tiempo polinómico.

Para demostrar que un problema está en NP tenemos que encontrar un problema Λ (versión comprobación de Π) y dos constantes c, c' que cumplan:

1. $\Lambda \in \text{P}$.
2. Una entrada x, y de Λ, con x entrada de Π, cumple $|y| \leq c' \cdot |x|^c$.
3. x tiene solución Sí para $\Pi \Leftrightarrow \exists y$ tal que x, y tiene solución Sí para Λ.

Ejemplos 9.12 Los siguientes problemas están en NP:

- SAT, ya que EVAL $\in$ P, las entradas X, L, α de EVAL cumplen

  $$|\alpha| \leq |X, L|$$

  y por definición de EVAL se cumple 3.

- MOCHILA, ya que compMOCHILA $\in$ P, las entradas $n, p_1, \ldots, p_n, d, A$ de compMOCHILA cumplen

  $$|A| \leq |n, p_1, \ldots, p_n, d|$$

  y por definición de compMOCHILA se cumple 3.

- CLIQUE, ya que compCLIQUE $\in$ P, las entradas G, k, U de compCLIQUE cumplen

  $$|U| \leq 3|G, k|$$

  y por definición de compCLIQUE se cumple 3.

Resumiendo, los problemas de NP son aquellos que tienen una versión comprobación que está en P, y de manera que las entradas de la versión comprobación no tengan tamaño mucho mayor que el problema original. Veamos dos propiedades importantes de NP.

Propiedad 9.13 P $\subseteq$ *NP.*

Dem. Sea $\Pi \in$ P. Considero una versión comprobación trivial Λ:

Datos: x entrada de Π, $w \in \{0,1\}^*$ con $|w| = 1$.

Salida: ¿x tiene respuesta Sí para Π ?

Si Σ es el alfabeto para codificar las entradas de Π, codifico las entradas de Λ sobre $\Sigma \cup \{0, 1, \|\}$. Para codificar x, w añado a la codicación de x $\|w$, luego $|w| \leq |x|$.
Como $|x, w| \geq |x|$, un algoritmo q para Λ que ignore w y utilice p un algoritmo para Π tarda tiempo $\mathrm{T}_q(m) \leq T_p(m)$ luego $\Lambda \in$ P.
Es trivial que x tiene solución Sí para $\Pi \Leftrightarrow \exists w, |w| \leq |x|$ tal que x, w tiene solución Sí para Λ. ∎

Propiedad 9.14 NP ⊆ EXP.

Dem. Sea $\Pi \in$ NP. Sea $\Lambda \in$ P el que cumple la definición *9.10*. Sean c, c' las constantes tales que si x, y es entrada de Λ, x entrada de Π, entonces $|y| \leq c' \cdot |x|^c$.
Sea p un algoritmo en tiempo polinómico para Λ. Sean k, k' constantes tales que $\mathrm{T}_p(m) \leq k' \cdot m^k$.
Podemos construir un algoritmo q para Π que con entrada x prueba todas las posibles y con $|y| \leq c' \cdot |x|^c$ para ver si x, y da solución sí para Λ:

```
Leer X
  Para cada Y con |Y| ≤ c'|X|^c hacer
    RESPUESTA:= p(X,Y);
    Si RESPUESTA = SI entonces Devuelve SI; Fin;
  Fpara;
  Devuelve NO.
```

Sea a el número de símbolos del alfabeto que codifica las entradas de Λ. El número de y que cumplen $|y| \leq c'|x|^c$ es

$$\frac{a^{c'|x|^c+1} - 1}{a - 1}$$

Luego

$$\begin{aligned} \mathrm{t}_q(x) &\leq \tfrac{a^{c'|x|^c+1}-1}{a-1} \cdot \mathrm{T}_p(c'|x|^c) \\ &\leq \tfrac{a^{c'|x|^c}}{a-1} \cdot k' c'^k \cdot (|x|^c)^k \end{aligned}$$

Como $a^{c'|x|^c} = 2^{c' \log(a)|x|^c}$

$$\mathrm{t}_q(x) \leq k' c'^k/(a-1) \cdot 2^{c' \log(a)|x|^c} \cdot (|x|^c)^k \in \mathrm{O}(2^{|x|^{c+1}})$$

Luego $\mathrm{T}_q(m) \in \mathrm{O}(2^{m^{c+1}})$ y $\Pi \in$ EXP. ∎

Así pues sabemos que P ⊆ NP ⊆ EXP. Estas son todas las relaciones conocidas entre NP y las clases P y EXP.
Se sospecha que P ≠ NP, es decir, que existan problemas comprobables en tiempo polinómico pero no resolubles en tiempo polinómico, pero no

existe ninguna prueba de ello. Lo que tenemos son muchos problemas en NP, los llamados NP-completos, que no se saben resolver en tiempo polinómico. Los estudiaremos en los siguientes capítulos.

EJERCICIOS

9.1. Demostrar que los siguientes problemas están en la clase NP.

1. TSP.
2. Compuesto:
   Datos: $n \in \mathbb{N}$
   Salida: ¿Existen $x, y \in \mathbb{N}$ tal que $x > 1, y > 1$ y $n = x \cdot y$?
   Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1\}$, en binario.
3. Linear Divisibility:
   Datos: $a, c \in \mathbb{N}$
   Salida: ¿Existe un $x \in \mathbb{N}$ tal que $ax + 1$ divide a c?
   Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, los 2 números naturales que componen una entrada se escriben en binario y se separan por comas.
4. Hitting-Set:
   Datos: $n \in \mathbb{N}$, $A_1, \ldots, A_l$ subconjuntos de $\{1, \ldots, n\}$, $k \in \mathbb{N}$
   Salida: ¿Existe $A \subseteq \{1, \ldots, n\}$ con $\#A \leq k$ y para todo $i \leq l$, $A_i \cap A \neq \emptyset$?
   Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, n y k se escriben en binario, cada subconjunto de $\{1, \ldots, n\}$ se escribe con n bits (utilizando la secuencia característica) y los $l + 2$ datos se separan por comas.
5. Multiprocessor Scheduling

Datos: $n \in \mathbb{N}$ el numero de tareas,
$l_1, \ldots, l_n$ el tiempo de cada tarea,
$M \in \mathbb{N}$ el número de procesadores
$C \in \mathbb{N}$ el tiempo máximo permitido

Salida: ¿Podemos repartir la n tareas entre los M procesadores de manera que cada procesador tarde un tiempo menor o

igual a C?, es decir,
¿Existen $A_1, \ldots, A_M$ subconjuntos de $\{1, \ldots, n\}$ tales que $A_1 \cup A_2 \ldots \cup A_M = \{1, \ldots, n\}$ y para cada $i \leq M$

$$\Sigma_{j \in A_i} l_j \leq C \ ?$$

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, los $n+3$ números naturales que componen una entrada se escriben en binario y se separan por comas.

6. Subgrafo:
Datos: $G = (V, A)$, $H = (V', A')$ dos grafos no dirigidos.
Salida: ¿Es H un subgrafo de G?, es decir, ¿existe $V_1 \subseteq V$ y $f : V_1 \to V'$ biyectiva tal que para cada $u, v \in V_1$, $\{u, v\} \in A$ si y sólo si $\{f(u), f(v)\} \in A'$?
Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $x \in \{0, 1\}^*$ es el número de vértices de G escrito en binario, $y \in \{0, 1\}^*$ es la matriz de adyacencia de G escrita por filas y z, t corresponden al número de vértices y matriz de adyacencia de H, entonces la codificación de la entrada G, H es la palabra x, y, z, t.

¿Están en EXP?

Capítulo 10

Reducciones en tiempo polinómico

Referencia: Capítulo 2.5 de [GJ78].

En el capítulo 4 estudiamos las reducciones recursivas entre lenguajes, $\leq_{\mathrm{m}}$. Aquí vamos a estudiar una parte de estas reducciones, las calculables en tiempo polinómico. En este caso daremos las definiciones para problemas decisionales codificados sobre un alfabeto cualquiera.

10.1. Definición

Definición 10.1 Dados dos alfabetos Σ y Γ, una función $f : \Sigma^* \to \Gamma^*$ es *calculable en tiempo polinómico* si existe un programa p que calcula f y una constante $k \in \mathbb{N}$ tales que $\mathrm{T}_p(m) \in \mathrm{O}(m^k)$.

Nota: Si f es calculable en tiempo polinómico, entonces existen c, k tal que $|f(x)| \leq c|x|^k$ para toda x.

Definición 10.2 Sean A y B problemas decisionales con entradas codificadas sobre los alfabetos Σ y Γ respectivamente. Una *reducción en tiempo polinómico* de A en B es una función $f : \Sigma^* \to \Gamma^*$ que cumple:

1. f es calculable en tiempo polinómico.
2. Para todo x entrada de A:

x tiene respuesta sí para $A \Leftrightarrow f(x)$ tiene respuesta sí para B.

Es decir, una reducción en tiempo polinómico es una reducción de $L(A)$ en $L(B)$ en el sentido definido en el capítulo 4, pero exigiendo que la reducción se pueda calcular en tiempo polinómico. Recordemos que $L(\Pi)$ es el lenguaje asociado al problema decisional Π (capítulo 2).
De hecho, la mayoría de las reducciones vistas en el capítulo 4 son calculables en tiempo polinómico.

Definición 10.3 Dados dos problemas decisionales A y B, *A es reducible en tiempo polinómico a B* si existe una reducción en tiempo polinómico de A en B. Lo denotamos con $A \leq_{\mathrm{m}}^{\mathrm{p}} B$.

10.2. Primer ejemplo

Recordemos TSP, el problema del viajante, definido en el capítulo 7.
Vamos a ver una primera reducción desde un problema de grafos, el problema del hamiltoniano (HAM), en el problema del viajante (TSP).
Comenzamos dando unas definiciones sobre caminos para poder enunciar HAM.

Definición 10.4 Dado un grafo no dirigido $G = (V, A)$ con $V = \{1, \ldots, n\}$, un *camino* es una secuencia de vértices $C = (c_1, \ldots, c_k)$ que cumple que para todo i desde 1 a $k-1$, $\{c_i, c_{i+1}\} \in A$.
La *longitud* de un camino $C = (c_1, \ldots, c_k)$ es $k-1$.
Un camino $C = (c_1, \ldots, c_k)$ es *simple* si no tiene vértices repetidos, es decir, para todo i desde 1 hasta k, $c_i \notin \{c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_k\}$.
Un *camino hamiltoniano* es un camino simple de longitud $n-1$, es decir, un camino simple que pasa por todos los vértices.
Un *circuito* es un camino $C = (c_1, \ldots, c_k)$ tal que $\{c_k, c_1\} \in A$.

Sea HAM el siguiente problema:

Datos: $G = (V, A)$ un grafo no dirigido con n vértices,

Salida: ¿Tiene G un camino hamiltoniano?

Codificamos la entrada con el alfabeto $\Sigma = \{0, 1, ,\}$, utilizando la codificación del grafo con matriz de adyacencia. Luego $|G| = n^2$.

Ejercicio. Demostrar que HAM $\in$ NP.

Para reducir HAM a TSP en tiempo polinómico queremos traducir cada entrada de HAM (un grafo no dirigido) en una entrada de TSP, es decir, un mapa de ciudades con sus distancias, de manera que el grafo original tenga un camino hamiltoniano si y sólo si el mapa tiene un camino corto que pasa por todas las ciudades. Para eso pondremos el mismo número de ciudades que vértices tiene el grafo y haremos la distancia entre dos ciudades pequeña si los correspondientes vértices están unidos con una arista, y grande en caso contrario, es decir, para cada $G = (V, A)$ un grafo de n vértices, la reducción f se define como:

$$f(G) = (n, d(i,j)(1 \leq i \leq n, 1 \leq j \leq n), n-1)$$

con

$$d(i,j) = \begin{cases} 1 & \text{si } \{i,j\} \in A \\ 2n & \text{en otro caso.} \end{cases}$$

Veamos que se trata de una reducción de HAM a TSP y que se puede calcular en tiempo polinómico.

Para ver que es una reducción, sea $G = (V, A)$ con $V = \{1, \ldots, n\}$ una entrada a HAM.

Si G tiene solución sí para HAM, entonces G tiene un camino hamiltoniano $C = (c_1, \ldots, c_n)$. Luego para cada i desde 1 hasta $n-1$, $\{c_i, c_{i+1}\} \in A$. Por tanto para cada i desde 1 hasta $n-1$, $d(c_i, c_{i+1}) = 1$ luego existe un camino, C, que pasa por las n ciudades una sola vez con longitud total $\sum_{i=1}^{n-1} d(c_i, c_{i+1}) = n-1$ y por tanto $f(G) = (n, d(i,j)(1 \leq i \leq n, 1 \leq j \leq n), n-1)$ tiene solución sí para TSP.

En la otra dirección, si $f(G)$ tiene solución sí para TSP, existe un camino $C = (c_1, \ldots, c_n)$ que pasa por las n ciudades una sola vez con longitud total menor o igual que $n-1$. Como $d(i,j)$ es siempre mayor que 0, para que $\sum_{i=1}^{n-1} d(c_i, c_{i+1}) \leq n-1$ tiene que ser $d(c_i, c_{i+1}) = 1$ para cada i desde 1 hasta $n-1$. Luego para cada i $\{c_i, c_{i+1}\} \in A$, C es un camino de G y por no repetir vértices y ser de logitud $n-1$ es un camino hamiltoniano, luego G tiene solución sí para HAM.

El tiempo que tarda un programa en calcular f con una entrada G es n^2 para calcular d más la instrucción de Devuelve, luego $\mathrm{t}_p(G) \leq n^2 + 1$. Como $|G| \geq n^2$, $\mathrm{T}_p(m) \leq m + 1$ y por tanto f es calculable en tiempo polinómico.

10.3. Propiedades elementales

Vamos a demostrar primero la propiedad que más nos interesará de las reducciones en tiempo polinómico, que es que si $A \leq_{\mathrm{m}}^{\mathrm{p}} B$ y $B \in \mathrm{P}$ entonces $A \in \mathrm{P}$. Esta propiedad es similar a la propiedad 4.5 y formaliza la idea de que si $A \leq_{\mathrm{m}}^{\mathrm{p}} B$ entonces A es tanto o más fácil que B.
Empezaremos con el siguiente lema.

Lema 10.5 Sea $h : \mathbb{N} \to \mathbb{N}$ *una función total creciente con* $h(m) \geq m$ *para todo* m. *Sean* A *y* B *dos problemas decisionales que cumplen*

1. $A \leq_{\mathrm{m}}^{\mathrm{p}} B$ *y*
2. $B \in DTIME(h(m))$.

Entonces existen dos constantes $c, k \in \mathbb{N}$ *tales que* $A \in DTIME(h(cm^k))$.

Dem. Sea p un programa que resuelve B en tiempo $\mathrm{T}_p(m) \leq c_1 \cdot h(m)$ con c_1 constante.
Sea f una reducción de A en B calculable en tiempo polinómico. Sea q un programa que calcula f con $\mathrm{T}_q(m) \leq c_2 \cdot m^{c_3}$, con c_2 y c_3 constantes. Por tanto para cada x entrada de A, $|f(x)| \leq |x| \cdot c_2 \cdot |x|^{c_3}$.
El siguiente algoritmo p' resuelve A:

Leer X
  Y:=φ_q(X);
  Z:=φ_p(Y);
  Devuelve Z.

Vamos a acotar el tiempo de p' con una entrada x:

$$\begin{aligned} \mathrm{t}_{p'}(x) &\leq \mathrm{t}_q(x) + \mathrm{t}_p(f(x)) \leq c_2 \cdot |x|^{c_3} + c_1 \cdot h(|f(x)|) \leq \\ &\leq h(c_2 \cdot |x|^{c_3}) + c_1 \cdot h(c_2 \cdot |x|^{c_3+1}) \end{aligned}$$

(ya que $h(m) \geq m$ para todo m).
Luego $\mathrm{T}_{p'}(m) \leq (c_1 + 1)h(cm^k)$ con $c = c_2$ y $k = c_3 + 1$. Por tanto $A \in \mathrm{DTIME}(h(cm^k))$. ∎

Luego si $B \in \mathrm{P}$ tenemos:

Teorema 10.6 Sean A y B dos problemas decisionales tales que $A \leq_{\mathrm{m}}^{\mathrm{p}} B$. Entonces se cumple que:

$$\textit{Si } B \in \mathrm{P} \textit{ entonces } A \in \mathrm{P}.$$

Dem. Si $B \in \mathrm{DTIME}(m^k)$, aplicamos el lema anterior para $h(m) = m^k$. ∎

El teorema anterior se puede enunciar equivalentemente como:

Teorema 10.7 Sean A y B dos problemas decisionales tales que $A \leq_{\mathrm{m}}^{\mathrm{p}} B$. Entonces se cumple que:

$$\textit{Si } A \notin \mathrm{P} \textit{ entonces } B \notin \mathrm{P}.$$

lo que nos servirá para tratar con los problemas NP-completos (en el próximo capítulo).
De manera análoga se demuestra el siguiente teorema:

Teorema 10.8 Sean A y B dos problemas decisionales tales que $A \leq_{\mathrm{m}}^{\mathrm{p}} B$. Si $B \in EXP$ entonces $A \in EXP$.

El siguiente teorema demuestra que los problemas que están en P son reducibles en tiempo polinómico a todos los problemas. Intuitivamente esto quiere decir que según el orden marcado por las reducciones $\leq_{\mathrm{m}}^{\mathrm{p}}$ los problemas en P son los más fáciles.

Teorema 10.9 Sea $A \in \mathrm{P}$. Sea B un problema decisional cualquiera no trivial, es decir, que no tiene la misma solución para todas las entradas. Entonces $A \leq_{\mathrm{m}}^{\mathrm{p}} B$

Dem. Por ser B no trivial, tomo dos entradas fijas y_1, y_2 tales que y_1 tiene respuesta sí para B e y_2 tiene respuesta no para B.
Sean Σ y Γ los alfabetos que codifican las entradas de A y B respectivamente.
Defino la función $f : \Sigma^* \to \Gamma^*$ como

$$f(x) = \begin{cases} y_1 & \text{si } x \text{ tiene solución sí para } A \\ y_2 & \text{si } x \text{ tiene solución no para } A \end{cases}$$

f es calculable en tiempo polinómico ya que la calcula el siguiente algoritmo, donde p es un programa para A con $\mathrm{t}_p(x) \leq c|x|^k$:

Leer X
  Simular(p,X,EXITO, Y);
  Si Y=SI
    entonces Devuelve y_1
    else Devuelve y_2.

(En este algoritmo y_1 e y_2 son constantes.)
f es claramente una reducción de A en B. ∎

La reducción $\leq_{\mathrm{m}}^{\mathrm{p}}$ es transitiva, lo que utilizaremos en el siguiente capítulo:

Teorema 10.10 Si $A \leq_{\mathrm{m}}^{\mathrm{p}} B$ *y* $B \leq_{\mathrm{m}}^{\mathrm{p}} C$ *entonces* $A \leq_{\mathrm{m}}^{\mathrm{p}} C$.

Dem. Sea f una reducción en tiempo polinómico de A en B y g una reducción en tiempo polinómico de B en C. Entonces la composición de f y g, $g \circ f(x) = g(f(x))$ es una reducción de A a C, ya que es cumple:

$$\begin{aligned} x \text{ tiene solución sí para } A &\Leftrightarrow f(x) \text{ tiene solución sí para } B \\ &\Leftrightarrow g(f(x)) = g \circ f(x) \text{ tiene solución sí para } C. \end{aligned}$$

Veamos que $g \circ f$ es calculable en tiempo polinómico. Si g es calculable en tiempo $\mathrm{T}_p(m) \leq cm^k$ y f es calculable en tiempo $\mathrm{T}_q(m) \leq c'm^{k'}$, entonces el algoritmo r:

Leer X
  Y:=f(X);
  Z:=g(Y);
  Devuelve Z.

funciona en tiempo

$$\begin{aligned} \mathrm{t}_r(x) \quad &\leq c'|x|^{k'} + c|f(x)|^k \leq c'|x|^{k'} + c(c'|x|^{k'+1})^k \leq \\ &\leq c''|x|^{(k'+1)k} \end{aligned}$$

∎

Definimos a continuación el concepto de $\leq_{\mathrm{m}}^{\mathrm{p}}$-difícil y $\leq_{\mathrm{m}}^{\mathrm{p}}$-completo para una clase:

Definición 10.11 Dada una clase o conjunto de problemas decisionales C, un problema decisional X es $\leq_{\mathrm{m}}^{\mathrm{p}}$-*difícil para* C si para todo $A \in$ C, $A \leq_{\mathrm{m}}^{\mathrm{p}} X$.

Definición 10.12 Dada una clase o conjunto de problemas decisionales C, un problema decisional X es $\leq^{\mathrm{p}}_{\mathrm{m}}$-*completo para* C si X es $\leq^{\mathrm{p}}_{\mathrm{m}}$-difícil para C y además $X \in$C.

Por la transitividad tenemos:

Corolario 10.13 Sea X *un problema decisional y sea* V *un problema* $\leq^{\mathrm{p}}_{\mathrm{m}}$-*difícil para C. Si* $V \leq^{\mathrm{p}}_{\mathrm{m}} X$ *entonces* X *es* $\leq^{\mathrm{p}}_{\mathrm{m}}$-*difícil para C.*

Dem. Sea $A \in$C. Como sabemos que $A \leq^{\mathrm{p}}_{\mathrm{m}} V$ y por hipótesis $V \leq^{\mathrm{p}}_{\mathrm{m}} X$, aplicando transitividad (teorema *10.10*) tenemos que $A \leq^{\mathrm{p}}_{\mathrm{m}} X$. ∎

EJERCICIOS

10.1. Demostrar el Teorema *10.8*.

Capítulo 11

Los problemas NP-completos

Referencia: Capítulo 2 de [GJ78].

En este capítulo estudiaremos una serie de problemas para los cuales no se conocen algoritmos eficientes, es decir, no se sabe si están en P. Estos problemas están fuertemente relacionados entre sí, de manera que si uno de ellos estuviera en P entonces lo estarían todos.

11.1. El concepto de NP-completo

Empezaremos enunciando sin demostración el teorema de Cook, que dice que cualquier problema de la clase NP se puede reducir a SAT.

Teorema 11.1 Para todo $A \in NP$ se cumple que $A \leq_{\mathrm{m}}^{\mathrm{p}} SAT$.

La demostración se puede ver en [HMU02] y en [GJ78].
Utilizando el teorema *10.6* tenemos la siguiente propiedad:

Corolario 11.2 Si $SAT \in$ P entonces cualquier $A \in NP$ está en P, luego $NP \subseteq$ P.

Como sabemos que P $\subseteq$ NP (propiedad *9.13*):

Corolario 11.3 Si $SAT \in$ P entonces P $=$ *NP.*

De otra forma:

Corolario 11.4 Si P $\neq$ *NP entonces $SAT \notin$ P.*

Como SAT$\in$ NP, si P = NP entonces SAT$\in$ P y por tanto:

Corolario 11.5 Son equivalentes:

a. P=NP

b. SAT$\in$P

De otra forma:

Corolario 11.6 Son equivalentes:

a. P$\neq$NP

b. SAT$\notin$P

Podemos interpretar estas propiedades como "SAT tiene dificultad máxima en NP", es decir, si existe un algoritmo eficiente para SAT entonces existen algoritmos eficientes para todos los problemas de NP.
Existen muchos otros problemas en NP que comparten estas propiedades de SAT, son los NP-completos.

Definición 11.7 Un problema se llama *NP-difícil* si es $\leq_{\mathrm{m}}^{\mathrm{p}}$-difícil para NP.
Un problema se llama *NP-completo* si es $\leq_{\mathrm{m}}^{\mathrm{p}}$-completo para NP.

Todos los NP-completos cumplen las anteriores propiedades de SAT (las demostraciones son análogas).

Propiedad 11.8 Sea A un NP-completo. Si P $\neq$ *NP entonces A* $\notin$ P.

Corolario 11.9 Sea A un NP-completo. Son equivalentes:

a. P$\neq$NP

b. A$\notin$P

Por tanto dados A y B NP-completos, son equivalentes:

a. A$\notin$P

b. B$\notin$P

ya que ambas afirmaciones son equivalentes a P≠NP.
Sabemos pues de los problemas NP-completos que:

- Si un NP-completo no está P entonces ninguno está en P.
- No se conocen algorimos que funcionen en tiempo polinómico para ningún NP-completo.

Por todo lo anterior, el que un problema A sea NP-completo se toma como una prueba de intratabilidad, en ese caso hay sospechas fundadas de que no existe un algoritmo eficiente que resuelva A.
Para demostrar que un problema es NP-completo utilizaremos la siguiente propiedad, que se sigue del corolario *10.13*:

Propiedad 11.10 Sea A un problema NP-completo. Si $B \in$NP y $A \leq^{\mathrm{p}}_{\mathrm{m}} B$ entonces B es NP-completo.

Luego como SAT es NP-completo, si demostramos que un problema B cumple:

- $B \in$NP
- SAT$\leq^{\mathrm{p}}_{\mathrm{m}} B$

entonces ya sabemos que B es NP-completo.
En adelante utilizaremos la última propiedad para demostrar que nuevos problemas son NP-completos a partir de los que ya sepamos que lo son.

11.2. Una reducción complicada

En esta sección vamos a demostrar que 3-SAT, el problema que definimos a continuación, es NP-completo.

Datos: Un conjunto de variables X y una fórmula CNF sobre X, L *con exactamente 3 literales en cada cláusula* (y que cumple que todas las variables de X aparecen al menos una vez en los literales de L, no hay cláusulas repetidas en L y dentro de cada cláusula no hay literales repetidos).

Salida: ¿Existe una asignación de verdad que satisface L?

3-SAT tiene como entradas un subconjunto de las entradas de SAT, la fórmulas CNF que tienen exactamente tres literales por cláusula, y dada una entrada de 3-SAT, la solución para SAT y para 3-SAT es la misma. Codificaremos las entradas de 3-SAT de la misma forma que codificábamos las de SAT.

Ejemplo 11.11 Una entrada de 3-SAT: $X = \{x_1, x_2, x_3\}$, $L = (\neg x_2 \vee x_3 \vee \neg x_1) \wedge (x_2 \vee \neg x_2 \vee x_1)$.

Podría pensarse que al haber restringido las fórmulas, 3-SAT es más fácil que SAT. Veamos que no, ya que 3-SAT es también NP-completo. Para ello vemos primero que 3-SAT $\in$ NP y después que SAT $\leq^{\mathrm{p}}_{\mathrm{m}}$ 3-SAT.

Teorema 11.12 3-SAT $\in$ NP.

Dem.
Si tomamos el problema EVAL definido en el capítulo 9, sabemos que EVAL $\in$ P y que si X, L es una entrada a 3-SAT y X, L, α es una entrada a EVAL entonces $|X, L, \alpha| \leq 3|X, L|$.
Además como dada una entrada a 3-SAT X, L, su solución para SAT y para 3-SAT es la misma, entonces

$$\begin{aligned} & X, L \text{ tiene solución Sí para 3-SAT} \\ \Leftrightarrow\ & \exists \alpha\ X, L, \alpha \text{ tiene solución Sí para EVAL} \end{aligned}$$

■

Teorema 11.13 SAT $\leq^{\mathrm{p}}_{\mathrm{m}}$ 3-SAT.

Dem. Para ver que SAT $\leq^{\mathrm{p}}_{\mathrm{m}}$ 3-SAT, tenemos que definir una reducción computable en tiempo polinómico de SAT a 3-SAT, es decir, una función calculable en tiempo polinómico que transforme cada entrada de SAT X, L en $f(X, L) = X', L'$ una entrada a 3-SAT, de forma que existe una asignación que satisface L si y sólo si existe una asignación que satisface L'.
Para definir f usamos la siguiente notación para los literales de una fórmula: Si $X = \{x_1, \ldots, x_n\}$, $L = c_1 \wedge \ldots \wedge c_k$, donde $c_1, \ldots c_k$ son cláusulas, entonces c_i tiene r_i literales $z^i_1, \ldots, z^i_{r_i}$, o sea, $c_i = (z^i_1 \vee \ldots \vee z^i_{r_i})$.

Ejemplo 11.14 $X = \{x_1, x_2, x_3, x_4, x_5\}$,

$$L = (\neg x_1) \wedge (x_3 \vee \neg x_1 \vee \neg x_4 \vee \neg x_5 \vee x_2)$$

Luego $r_1 = 1$, $r_2 = 5$, $z_1^1 = \neg x_1$, $z_1^2 = x_3$, $z_2^2 = \neg x_1$, $z_3^2 = \neg x_4$, $z_4^2 = \neg x_5$, $z_5^2 = x_2$.

Vamos a transformar X, L en X', L' con $X' = X \cup (\bigcup_{i=1}^{k} Y_i)$ con Y_i nuevas variables añadidas para modificar la cláusula c_i, $L' = \wedge_{i=1}^{k} D_i$ donde cada D_i es una fórmula CNF con tres literales por cláusula que sustituye a la cláusula c_i de L.
Definimos a continuación Y_i, D_i para cada i desde 1 hasta k:
Si $r_i = 1$ entonces $Y_i = \{y_1^i, y_2^i\}$ y

$$D_i = (z_1^i \vee y_1^i \vee y_2^i) \wedge (z_1^i \vee y_1^i \vee \neg y_2^i) \wedge (z_1^i \vee \neg y_1^i \vee y_2^i) \wedge (z_1^i \vee \neg y_1^i \vee \neg y_2^i)$$

Si $r_i = 2$ entonces $Y_i = \{y_1^i\}$ y

$$D_i = (z_1^i \vee z_2^i \vee y_1^i) \wedge (z_1^i \vee z_2^i \vee \neg y_1^i)$$

Si $r_i = 3$ entonces $Y_i = \emptyset$ y $D_i = c_i$.
Si $r_i > 3$ entonces $Y_i = \{y_1^i, \ldots, y_{r_i-3}^i\}$ y

$$\begin{aligned} D_i = \quad & (z_1^i \vee z_2^i \vee y_1^i) \wedge (\neg y_1^i \vee z_3^i \vee y_2^i) \wedge (\neg y_2^i \vee z_4^i \vee y_3^i) \wedge \ldots \\ & \ldots \wedge (\neg y_{s-2}^i \vee z_s^i \vee y_{s-1}^i) \wedge \ldots \wedge (\neg y_{r_i-4}^i \vee z_{r_i-2}^i \vee y_{r_i-3}^i) \wedge \\ & \wedge (\neg y_{r_i-3}^i \vee z_{r_i-1}^i \vee z_{r_i}^i) \end{aligned}$$

es decir,

$$D_i = (z_1^i \vee z_2^i \vee y_1^i) \wedge (\neg y_{r_i-3}^i \vee z_{r_i-1}^i \vee z_{r_i}^i) \wedge_{s=3}^{r_i-2} (\neg y_{s-2}^i \vee z_s^i \vee y_{s-1}^i)$$

Ejemplo 11.15 El ejemplo anterior queda transformado en
$X' = \{x_1, x_2, x_3, x_4, x_5, y_1^1, y_2^1, y_1^2, y_2^2\}$,

$$L' = (\neg x_1 \vee y_1^1 \vee y_2^1) \wedge (\neg x_1 \vee y_1^1 \vee \neg y_2^1) \wedge (\neg x_1 \vee \neg y_1^1 \vee y_2^1)$$

$$\wedge (\neg x_1 \vee \neg y_1^1 \vee \neg y_2^1) \wedge (x_3 \vee \neg x_1 \vee y_1^2) \wedge (\neg y_1^2 \vee \neg x_4 \vee y_2^2) \wedge (\neg y_2^2 \vee \neg x_5 \vee x_2)$$

Veamos que existe una asignación que satisface L si y sólo si existe una asignación que satisface L'.

$\Rightarrow$) Supongamos que existe una asignación $\alpha : X \to \{T, F\}$ que satisface L. Definimos $\beta : X' \to \{T, F\}$ una asignación que satisface L' como sigue:
Si $x \in X$, $\beta(x) = \alpha(x)$.
Si $x \in Y_i$ distinguimos tres casos:

- Si $r_i = 1$ entonces $\beta(y_1^i) = \beta(y_2^i) = T$.
- Si $r_i = 2$ entonces $\beta(y_1^i) = T$.
- Si $r_i > 3$ entonces como α satisface L, α satisface cada c_i, luego existen uno o más literales de c_i satisfechos por α. Sea j el primero tal que α satisface z_j^i.
  - Si $j \leq 2$ entonces $\beta(y_s^i) = F$ para todo s.
  - Si $2 < j < r_i - 1$ entonces:
$$\beta(y_s^i) = \begin{cases} T & \text{para } 1 \leq s \leq j-2 \\ F & \text{para } j-1 \leq s \leq r_i - 3 \end{cases}$$
  - Si $j \geq r_i - 1$ entonces $\beta(y_s^i) = T$ para todo s.

Veamos que β satisface L' viendo que satisface todas las D_i para i desde 1 hasta k:

a. Si $r_i = 1$ entonces α satisface z_1^i, luego β satisface z_1^i y también D_i
$$D_i = (z_1^i \vee y_1^i \vee y_2^i) \wedge (z_1^i \vee y_1^i \vee \neg y_2^i) \wedge (z_1^i \vee \neg y_1^i \vee y_2^i) \wedge (z_1^i \vee \neg y_1^i \vee \neg y_2^i)$$

b. Si $r_i = 2$ entonces α satisface $(z_1^i \vee z_2^i)$, luego β también satisface
$$D_i = (z_1^i \vee z_2^i \vee y_1^i) \wedge (z_1^i \vee z_2^i \vee \neg y_1^i)$$

c. Si $r_i > 3$ entonces
$$D_i = (z_1^i \vee z_2^i \vee y_1^i) \wedge (\neg y_{r_i-3}^i \vee z_{r_i-1}^i \vee z_{r_i}^i) \wedge_{s=3}^{r_i-2} (\neg y_{s-2}^i \vee z_s^i \vee y_{s-1}^i)$$
sea j el primero tal que α satisface z_j^i.

   a) Si $2 < j < r_i - 1$ entonces β satisface la cláusula $(\neg y_{j-2}^i \vee z_j^i \vee y_{j-1}^i)$ de D_i. Son ciertas las y_s^i para $1 \leq s \leq j-2$ y falsas el resto, luego son ciertas las cláusulas $(\neg y_{s-1}^i \vee z_{s+1}^i \vee y_s^i)$ para $2 \leq s \leq j-2$, las $(\neg y_s^i \vee z_{s+2}^i \vee y_{s+1}^i)$ para $j-1 \leq s \leq r_i-4$, y las cláusulas primera y última $(z_1^i \vee z_2^i \vee y_1^i)$, $(\neg y_{r_i-3}^i \vee z_{r_i-1}^i \vee z_{r_i}^i)$.

b) Si $j \leq 2$ β satisface la cláusula $(z_1^i \vee z_2^i \vee y_1^i)$ de D_i y hemos hecho todas las y_s^i falsas, luego todas las cláusulas de la forma $(\neg y_{s-2}^i \vee z_s^i \vee y_{s-1}^i)$ son ciertas y también la última $(\neg y_{r_i-3}^i \vee z_{r_i-1}^i \vee z_{r_i}^i)$.

c) Si $j \geq r_i - 1$ β satisface la cláusula $(\neg y_{r_i-3}^i \vee z_{r_i-1}^i \vee z_{r_i}^i)$ de D_i. Todas las y_s^i son ciertas, luego son ciertas todas las cláusulas de la forma $(\neg y_{s-2}^i \vee z_s^i \vee y_{s-1}^i)$ y también la primera $(z_1^i \vee z_2^i \vee y_1^i)$.

$\Leftarrow$) Supongamos que existe una asignación $\beta : X' \to \{T, F\}$ que satisface L', tomamos $\alpha : X \to \{T, F\}$ como $\alpha(x) = \beta(x)$ para cada $x \in X$. Veamos que α satisface L, es decir, que satisface c_i para todo i desde 1 a k.
Si $r_i = 1$ entonces β satisface las cuatro cláusulas $(z_1^i \vee y_1^i \vee y_2^i)$, $(z_1^i \vee y_1^i \vee \neg y_2^i)$, $(z_1^i \vee \neg y_1^i \vee y_2^i)$ y $(z_1^i \vee \neg y_1^i \vee \neg y_2^i)$. Como una de las cuatro cláusulas $(y_1^i \vee y_2^i)$, $(y_1^i \vee \neg y_2^i)$, $(\neg y_1^i \vee y_2^i)$, $(\neg y_1^i \vee \neg y_2^i)$ es falsa, necesariamente β hace cierto z_1^i y por tanto c_i.
Si $r_i = 2$ entonces β satisface las dos cláusulas $(z_1^i \vee z_2^i \vee y_1^i)$ y $(z_1^i \vee z_2^i \vee \neg y_1^i)$, luego tiene que satisfacer $(z_1^i \vee z_2^i) = c_i$.
Si $r_i > 3$ sea j el primero tal que $\beta(y_j^i) = T$, si no existe tal j entonces $j = r_i$. Sea s el último tal que $\beta(y_s^i) = F$, si no existe tal s entonces $s = 0$.

- Si $j > 1$ entonces $\beta(y_1^i) = F$ y por ser cierta la cláusula $(z_1^i \vee z_2^i \vee y_1^i)$ es cierta $(z_1^i \vee z_2^i)$ y por tanto c_i.
- Si $s < r_i - 3$ entonces $\beta(y_{r_i-3}^i) = T$ y por ser cierta la cláusula $(\neg y_{r_i-3}^i \vee z_{r_i-1}^i \vee z_{r_i}^i)$ es cierta $(z_{r_i-1}^i \vee z_{r_i}^i)$ y por tanto c_i.
- Si $j = 1$ y $s = r_i - 3$ entonces existe un l con $1 \leq l \leq r_i - 4$ tal que $\beta(y_l^i) = T$, $\beta(y_{l+1}^i) = F$. Por tanto la cláusula $(\neg y_l^i \vee z_{l+2}^i \vee y_{l+1}^i)$ es cierta por ser cierto z_{l+2}^i, luego β satisface c_i.

Para terminar falta ver que f es calculable en tiempo polinómico. Para calcular f sólo hace falta contar cláusulas y varibles de X, L y recorrer las cláusulas c_i transformándolas en D_i.
Todo esto se puede hacer en tiempo

$$\mathrm{t}_p(X, L) \leq \sum_{i=1}^{k} 4r_i$$

como cada $r_i \leq 2n$,

$$\mathrm{t}_p(X, L) \leq 8kn$$

Como $|X, L| \geq k$ y $|X, L| \geq n$, $\mathrm{T}_p(m) \leq 8m^2$. ∎

11.3. Algunos problemas NP-completos

Ya sabemos que SAT y 3-SAT son NP-completos. Vamos a ver aquí cinco problemas más que también son NP-completos, lo cual nos permitirá demostrar que un nuevo problema es NP-completo utilizando la propiedad *11.10* y uno de los NP-completos ya conocidos.
Comenzaremos enunciando los problemas VC y PARTICION.

11.3.1. El problema del Vertex Cover

Este problema trata de cubrimientos de un grafo, que pasamos a definir.

Definición 11.16 Dado un grafo no dirigido $G = (V, A)$, un *cubrimiento* por vértices de G es un conjunto $X \subseteq V$ tal que, para toda arista $\{u, v\} \in A$, $u \in X$ ó $v \in X$.

El problema VC o vertex cover trata de la existencia de cubrimientos pequeños de un grafo:

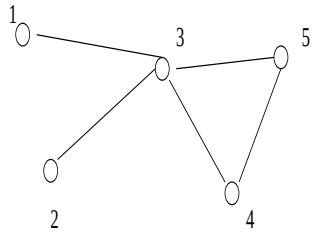
Datos: $G = (V, A)$ un grafo no dirigido con n vértices,

$k \in \mathbb{N}$ con $k \leq n$.

Salida: ¿Existe un cubrimiento de G con k vértices?

Codificamos la entrada como en el problema CLIQUE.

Ejemplo 11.17 Sea G:



Este grafo tiene un cubrimiento de dos vértices, $X = \{3, 4\}$.

El problema VC tiene una estrecha relación con CLIQUE, ya que se cumple la siguiente propiedad:

Propiedad 11.18 Sea $G = (V, A)$ un grafo no dirigido, y sea $X \subseteq V$. Son equivalentes:

1. *X es un cubrimiento de G.*
2. *$V - X$ es un clique de G^c, donde $G^c = (V, A^c)$ con*
$$A^c = \{\{u, v\} \mid \{u, v\} \notin A\}$$

Dem.
1.⇒2. Sea X un cubrimiento de G. Veamos que $V - X$ es un clique de G^c. Sean $u, v \in V - X$, $u \neq v$. Entonces $\{u, v\} \notin A$, ya que ni u no v están en X, que es un cubrimiento de G. Por tanto $\{u, v\} \in A^c$.
Esto se cumple para todo $u, v \in V - X$, $u \neq v$, luego $V - X$ es clique de G^c.
2.⇒1. Sea X tal que $V - X$ es clique de G^c. Veamos que X es un cubrimiento de G. Sea $\{u, v\} \in A$, $u \neq v$. Como $\{u, v\} \notin A^c$, al menos uno de los dos u, v no está en $V - X$ que es un clique de G^c, luego al menos uno de los dos u, v está en X.
Como eso se cumple para cada $\{u, v\} \in A$, X es un cubrimiento de G. ∎

Notemos que $(G^c)^c = G$, luego tenemos el siguiente corolario que relaciona los problemas CLIQUE y VC.

Corolario 11.19 Sea G un grafo no dirigido de n vértices.

1. *G, k tiene solución sí para VC si y sólo si $G^c, n - k$ tiene solución sí para CLIQUE.*
2. *G, k tiene solución sí para CLIQUE si y sólo si $G^c, n - k$ tiene solución sí para VC.*

Luego tenemos el siguiente resultado

Teorema 11.20 VC $\leq_{\mathrm{m}}^{\mathrm{p}}$ CLIQUE y CLIQUE $\leq_{\mathrm{m}}^{\mathrm{p}}$ VC.

Dem. La función $f(G, k) = G^c, n - k$ es reducción en los dos casos por la propiedad anterior.
Para calcular f, sólo es necesario intercambiar ceros y unos en la matriz de adyacencia, lo cual se puede hacer en tiempo n^2, luego $\mathrm{T}_p(m) \leq m$ ($|G, k| \geq n^2$) y f es calculable en tiempo polinómico. ∎

También demostramos que VC $\in$ NP.

Teorema 11.21 VC $\in$ NP.

Dem. Sea compVC el siguiente problema:

Datos: $G = (V, A)$ grafo no dirigido, $k \in \mathbb{N}$, U subconjunto de V de k elementos.

Salida: ¿Es U un cubrimiento de G ?

Con las entradas codificadas como en compCLIQUE. Por tanto $|G, k, U| \geq n^2$ y $|G, k, U| \leq 5 \cdot |G, k|$.
Un algoritmo para compVC es un único bucle que para una entrada G, k, U comprueba si todas las parejas u, v con $u, v \in V$, $u \neq v$ cumplen que si $\{u, v\} \in A$ entonces $u \in U$ ó $v \in U$. Si q es el algoritmo anterior,

$$\mathrm{t}_q(G, k, U) \leq 2n^2$$

Como $|G, k, U| \geq n^2$, $\mathrm{T}_q(m) \leq 2m$, compVC $\in$ DTIME$(m) \subseteq$ P.
Además por definición de compVC

$$\begin{aligned} & G, k \text{ tiene solución Sí para VC} \\ \Leftrightarrow\ & \exists U\ G, k, U \text{ tiene solución Sí para compVC} \end{aligned}$$

Luego VC $\in$ NP. ∎

11.3.2. El problema PARTICION

El problema PARTICION se define como:

Datos: $n \in \mathbb{N}$ el número de objetos,

$p_1, \ldots, p_n \in \mathbb{N}$ los pesos de los objetos.

Salida: ¿Existe un conjunto de objetos $A \subseteq \{1, \ldots, n\}$ que cumpla:

$$\sum_{i \in A} p_i = \sum_{i \notin A} p_i \ ?$$

Esto es equivalente a decir, ¿existe un A tal que $\sum_{i \in A} p_i = \frac{p_1 + \ldots + p_n}{2}$?

Codificamos la entrada con el alfabeto $\Sigma = \{0, 1, ,\}$, cada natural en binario y separados por comas: $n, p_1, \ldots, p_n$
Dejamos como ejercicio la demostración del siguiente teorema:

Teorema 11.22 PARTICION $\in$ *NP.*

11.3.3. Siete NP-completos

Ya sabemos que SAT y 3-SAT son NP-completos. Vamos a ver que CLIQUE, VC, HAM, TSP y PARTICION también lo son partiendo del siguiente resultado, que no demostraremos.

Teorema 11.23

- *3-SAT* $\leq_{\mathrm{m}}^{\mathrm{p}}$ *PARTICION.*
- *3-SAT* $\leq_{\mathrm{m}}^{\mathrm{p}}$ *VC.*
- *VC* $\leq_{\mathrm{m}}^{\mathrm{p}}$ *HAM.*

Corolario 11.24 PARTICION, VC y HAM son NP-completos.

Dem. Sabemos que PARTICION, VC y HAM están en NP.
A partir de que 3-SAT es NP-completo y de las dos primeras partes del teorema anterior tenemos que PARTICION y VC son NP-completos. Utilizando esto y la tercera parte del teorema anterior, HAM es NP-completo. ∎

Utilizando dos reducciones ya conocidas

Teorema 11.25 CLIQUE y TSP son NP-completos.

Dem. Sabemos que CLIQUE y TSP están en NP (capítulo 9), y que HAM $\leq_{\mathrm{m}}^{\mathrm{p}}$ TSP (capítulo 10) y VC $\leq_{\mathrm{m}}^{\mathrm{p}}$ CLIQUE. Luego por el corolario anterior, CLIQUE y TSP son NP-completos. ∎

Nota: También son NP-completos los problemas dCLIQUE, dVC y dHAM cuyos enunciados son exactamente iguales a los de CLIQUE, VC y HAM excepto que el grafo de entrada es *DIRIGIDO*. El lector puede comprobar que estos tres nuevos problemas pertenecen a la clase NP. La demostración de que son NP-difíciles no se tratará.

EJERCICIOS

11.1. Demostrar que los siguientes problemas son NP-completos, sabiendo que lo son SAT, 3-SAT, VC, CLIQUE, HAM, TSP y PARTICION.

1. MOCHILA:
   Datos: $n, p_1, \ldots, p_n, k, d \in \mathbb{N}$
   Salida: ¿Existe $A \subseteq \{1, \ldots, n\}$ con

   $$k - d \leq \Sigma_{i \in A} p_i \leq k \ ?$$

   Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, los $n+3$ números naturales que componen una entrada se escriben en binario y se separan por comas.
   Pista: Reducir PARTICION.

2. Hitting-Set:
   Datos: $n \in \mathbb{N}$, $A_1, \ldots, A_l$ subconjuntos de $\{1, \ldots, n\}$, $k \in \mathbb{N}$
   Salida: ¿Existe $A \subseteq \{1, \ldots, n\}$ con $\#A \leq k$ y para todo $i \leq l$, $A_i \cap A \neq \emptyset$?
   Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, n y k se escriben en binario, cada subconjunto de $\{1, \ldots, n\}$ se escribe con n bits (utilizando la secuencia característica) y los $l + 2$ datos se separan por comas.
   Pista: Reducir VC.

3. Multiprocessor Scheduling

   Datos: $n \in \mathbb{N}$ el numero de tareas,
   $l_1, \ldots, l_n$ el tiempo de cada tarea,
   $M \in \mathbb{N}$ el número de procesadores
   $C \in \mathbb{N}$ el tiempo máximo permitido
   Salida: ¿Podemos repartir la n tareas entre los M procesadores de manera que cada procesador tarde un tiempo menor o igual a C?, es decir,
   ¿Existen $A_1, \ldots, A_M$ subconjuntos de $\{1, \ldots, n\}$ tales que $A_1 \cup A_2 \ldots \cup A_M = \{1, \ldots, n\}$ y para cada $i \leq M$

   $$\Sigma_{j \in A_i} l_j \leq C \ ?$$

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, los $n+3$ números naturales que componen una entrada se escriben en binario y se separan por comas.

Pista: Reducir PARTICION.

4. Partición en Hamiltonianos:
   Datos: $G = (V, A)$ grafo no dirigido, $k \in \mathbb{N}$
   Salida: ¿Existen $V_1, \ldots, V_k$ tales que $V_1 \cup \ldots V_k = V$ y para cada $i \leq k$ el subgrafo de vértices V_i tiene un camino hamiltoniano?
   Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $x \in \{0, 1\}^*$ es el número de vértices de G escrito en binario, $y \in \{0, 1\}^*$ es la matriz de adyacencia de G escrita por filas y z es k en binario, entonces la codificación de la entrada G, k es la palabra x, y, z.
   Pista: Reducir HAM.

5. Subgrafo:
   Datos: $G = (V, A)$, $H = (V', A')$ dos grafos no dirigidos.
   Salida: ¿Es H un subgrafo de G?, es decir, ¿existe $V_1 \subseteq V$ y $f : V_1 \to V'$ biyectiva tal que para cada $u, v \in V_1$, $\{u, v\} \in A$ si y sólo si $\{f(u), f(v)\} \in A'$?
   Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $x \in \{0, 1\}^*$ es el número de vértices de G escrito en binario, $y \in \{0, 1\}^*$ es la matriz de adyacencia de G escrita por filas y z, t corresponden al número de vértices y matriz de adyacencia de H, entonces la codificación de la entrada G, H es la palabra x, y, z, t.
   Pista: Reducir CLIQUE. G tiene un clique de k vértices si y sólo si el grafo completo de k vértices es subgrafo de G.

Capítulo 12

Ejercicios de examen

12.1. Teoría

1. Contestad Verdadero o Falso a cada una de las siguientes preguntas. ATENCIÓN: Las respuestas erróneas tienen puntuación negativa.

- Si A es un lenguaje decidible entonces A es semidecidible.
- Si A es un lenguaje semidecidible y B es un lenguaje indecidible entonces $A \leq_{\mathrm{m}} B$.
- Todo problema NP-completo está en NP.
- Si $A \in \mathrm{P}$ y B es un lenguaje decidible entonces $A \leq_{\mathrm{m}} B$.
- Si $A \leq_m B$ entonces $B \leq_m A$
- La intersección de dos lenguajes semidecidibles es un lenguaje semidecidible.
- Si $A \leq_m K$ y $A \leq_m \bar{K}$ entonces A es decidible.
- Si A es el dominio de una función calculable entonces A es semidecidible.
- Si $K \leq_{\mathrm{m}} A$ entonces A es semidecidible.
- La intersección de dos lenguajes decidibles es un lenguaje decidible.
- Si A es el conjunto imagen de una función calculable entonces A es semidecidible.

- La unión de dos lenguajes decidibles es un lenguaje decidible.
- NP $\subseteq$ EXP.
- Si $A \in$ P y $B \in$ NP entonces $A \leq_{\mathrm{m}}^{\mathrm{p}} B$.
- Si A es un lenguaje semidecidible entonces A es el conjunto imagen de una función calculable.
- Si A es un lenguaje NO decidible y B es un lenguaje semidecidible entonces $A \leq_{\mathrm{m}} B$.
- La intersección de un lenguaje decidible y un lenguaje semidecidible es un lenguaje decidible.
- Si $A \leq_{\mathrm{m}} K$ entonces A es semidecidible.
- Si $A \in$ NP y B no es NP-difícil entonces $A \leq_{\mathrm{m}}^{\mathrm{p}} B$.
- Si A es NP-completo y B no es NP-difícil entonces $A \leq_{\mathrm{m}}^{\mathrm{p}} B$.
- Si A es el dominio de una función calculable entonces A es decidible.
- La unión de un lenguaje decidible y un lenguaje semidecidible es un lenguaje decidible.
- Si A es un lenguaje semidecidible entonces A es decidible.
- Si $A \in$NP y B es NP-difícil entonces $A \leq_{\mathrm{m}}^{\mathrm{p}} B$.
- Si A es un lenguaje decidible entonces A es el dominio de una función calculable.
- Si $L_1 \cup L_2$ es decidible entonces L_1 es decidible ó L_2 es decidible.
- P $\subseteq$ NP.
- Si A es un lenguaje decidible y $B \in$ P entonces $A \leq_{\mathrm{m}} B$.
- La unión de dos lenguajes semidecidibles es un lenguaje semidecidible.
- SAT $\in$ P y existe $B \in$ NP $-$ P.
- SAT está en EXP.

2. Sean A y B dos lenguajes o problemas decisionales. Demostrar la veracidad o falsedad de cada una de las siguientes afirmaciones. Si alguna de ellas es falsa, decir si es falsa en todos los casos o bien existen A y B para los cuales es cierta.

a. Si A es un lenguaje indecidible y B es un lenguaje semidecidible entonces $A \leq_{\mathrm{m}} B$.

b. Si A es un lenguaje semidecidible y B es un lenguaje indecidible entonces $A \leq_{\mathrm{m}} B$.

c. Si A es un lenguaje decidible y $B \in \mathrm{P}$ entonces $A \leq_{\mathrm{m}} B$.

d. Si $A \in \mathrm{P}$ y B es un lenguaje decidible entonces $A \leq_{\mathrm{m}} B$.

e. Si $A \in \mathrm{P}$ y $B \in \mathrm{NP}$ entonces $A \leq_{\mathrm{m}}^{\mathrm{p}} B$.

f. Si $A \in \mathrm{NP}$ y B no es NP-difícil entonces $A \leq_{\mathrm{m}}^{\mathrm{p}} B$.

g. Si A es NP-completo y B no es NP-difícil entonces $A \leq_{\mathrm{m}}^{\mathrm{p}} B$.

3. Demostrar la veracidad o falsedad de cada una de las siguientes afirmaciones:

a. Si A es un lenguaje decidible entonces A es el dominio de una función calculable.

b. Si A es el dominio de una función calculable entonces A es decidible.

c. Si A es un lenguaje semidecidible entonces A es el conjunto imagen de una función calculable.

d. Si A es el conjunto imagen de una función calculable entonces A es semidecidible.

4.

- Demostrar que $\mathrm{P} \subseteq \mathrm{NP}$.
- Demostrar que las siguientes afirmaciones son equivalentes para un lenguaje A:
  a. A es semidecidible.
  b. Existe un lenguaje B decidible tal que para cualquier x

$$x \in A \iff \exists\, t \;\; x, t \in B$$

5. Para los siguientes conjuntos de problemas decisionales:

NP, P, SEMIDEC, EXP, NP-completos, DEC

(SEMIDEC son los problemas semidecidibles, DEC los decidibles)

a. ¿Qué contenidos sabes que se cumplen entre los conjuntos anteriores? Para cada uno de ellos, explicar porqué.

b. ¿Qué contenidos sabes que NO se cumplen entre los conjuntos anteriores? Para cada uno de ellos, explicar porqué.

6. ¿Es correcta la siguiente demostración de que HAM $\notin$ P? Razonar la respuesta.

> Consideramos el siguiente algoritmo para SAT: "Con entrada X, F, probar todas las posibles asignaciones de las variables de X. Si alguna hace cierta F devuelve Sí, en caso contrario devuelve No." Este algoritmo requiere claramente tiempo exponencial. Por tanto SAT no está en P. Como SAT $\leq_{\mathrm{m}}^{\mathrm{p}}$ HAM, debe cumplirse que HAM $\notin$ P.

7. Definir los siguientes conceptos:

a. Máquina de Turing

b. Función calculada por una máquina de Turing.

8. ¿Qué función calcula la máquina de Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, q_F)$ con $Q = \{q_0, q_C, q_A, q_B, q_F\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \triangleright, b\}$ y δ definida como:

$$\begin{aligned}
\delta(q_0, 0) &= (q_0, 0, d) \\
\delta(q_0, 1) &= (q_0, 1, d) \\
\delta(q_0, b) &= (q_C, b, i) \\
\delta(q_C, 0) &= (q_F, 1, n) \\
\delta(q_C, 1) &= (q_C, 0, i) \\
\delta(q_C, \triangleright) &= (q_A, \triangleright, d) \\
\delta(q_A, 0) &= (q_B, 1, d) \\
\delta(q_B, 0) &= (q_B, 0, d) \\
\delta(q_B, b) &= (q_F, 0, n) \quad ?
\end{aligned}$$

9. Para cada una de las siguientes afirmaciones, decir si es cierta o falsa, *razonándolo* en cualquiera de los dos casos. Se puede utilizar cualquier propiedad vista en clase, enunciándola adecuadamente.

(DEC es el conjunto de los lenguajes decidibles, SEMIDEC es el conjunto de los lenguajes semidecidibles.)

a. SEMIDEC $\subseteq$ DEC
b. NP $\subseteq$ P
c. Si $A \in$ P y $B \in$ NP entonces $A \cup B \in$ NP
d. Si $K \leq_m A$ entonces A es semidecidible

10. En la asignatura de Metodología de la Programación los alumnos deben realizar una práctica 0 que consiste en escribir un programa en ada que calcule la suma de dos números enteros.

a. ¿Puede el profesor corregir automáticamente las prácticas, es decir, hacer un programa que toma como entrada un fuente ada cualquiera y dice si calcula la suma correctamente o no?
b. Si se impone la condición adicional de que el fuente debe tener menos de 100 caracteres, ¿cuál es la respuesta de a)?

En los dos apartados hay que demostrar la respuesta, tanto si esta es afirmativa como si es negativa.

11. Encontrar el fallo en la siguiente demostración errónea de P$\neq$NP:

Consideramos el siguiente algoritmo para SAT: "Con entrada X, F, probar todas las posibles asignaciones de las variables de X. Si alguna hace cierta F devuelve Sí, en caso contrario devuelve No." Este algoritmo requiere claramente tiempo exponencial. Por tanto SAT no está en P. Como SAT está en NP, debe cumplirse que P no es igual a NP.

12. Acabas de empezar a trabajar como analista jefe para la compañía Comunicaciones Universales S.A., que tiene un total de 500.000 centrales telefónicas en esta galaxia. En cada momento algunas conexiones entre centrales pueden fallar, así que tu primer encargo consiste en diseñar un algoritmo que dado un mapa de las conexiones que funcionan en un momento dado, dé el camino más corto que pase por todas las centrales, sin pasar dos veces por ninguna.

Después de varios días de trabajo entusiasta, empiezas a desanimarte al ver que el mejor algoritmo que se te ocurre puede llegar a tardar 10^{100000} años en resolver el problema.

Al día siguiente vas al despacho del jefe con un libro de complejidad en la mano y con la intención de convencerle de que redefina el problema (por ejemplo, ¿no le bastaría un camino cualquiera? o ¿los mapas de conexiones son de algún tipo especial?).

Explica de manera formal esta conversación con el jefe (es importante que le convenzas de que tú no eres un mal programador).

13. a. Definir la clase NP.

b. Demostrar que NP $\subseteq$ EXP.

14. Desarrollar las siguientes cuestiones (con demostraciones):

a. Definir los conceptos de: a) función calculable, b) lenguaje decidible, c) lenguaje semidecidible.

b. Relación entre los conceptos de lenguaje decidible y lenguaje semidecidible.

c. Comportamiento de los lenguajes decidibles y semidecidibles con las operaciones de unión e intersección.

15. Sea 6-SAT el siguiente problema:

Datos: Un conjunto de variables X y una fórmula CNF sobre X, L con exactamente 6 literales por cláusula (y cumpliendo que todas las variables de X aparecen al menos una vez en los literales de L, no hay cláusulas repetidas en L y dentro de cada cláusula no hay literales repetidos).

Salida: ¿Existe una asignación de verdad que satisface L?

Codificación de las entradas: La misma que la de las entradas de SAT.

Demostrar que 6-SAT $\leq_{\mathrm{m}}^{\mathrm{p}}$ 3-SAT, utilizando una versión simplificada de la reducción de SAT a 3-SAT.

16. Sean H y K las dos versiones del problema de parada. Demostrar las siguientes afirmaciones:

a. H no es decidible, y tampoco lo es K.

b. H y K son ambos semidecidibles

c. Ni $\bar{H}$ ni $\bar{K}$ son semidecidibles.

En la realización de este ejercicio no debe utilizarse NINGUN resultado que no se demuestre.

17. Contestar a cada una de las siguientes cuestiones, razonando la respuesta. En dicha respuesta se puede utilizar cualquier propiedad vista en clase, enunciándola adecuadamente.

a. ¿Todo problema NP-completo está en P?

b. ¿SAT está en EXP?

c. ¿La unión de un lenguaje decidible y un lenguaje semidecidible es un lenguaje decidible?

d. ¿Si $A \leq_m B$ entonces $B \leq_m A$?

e. ¿Si $A \leq_m K$ y $A \leq_m \bar{K}$ entonces A es decidible?

18. Enunciar el teorema de Rice. Explicar su significado y utilidad. Demostrar dicho teorema.

19. Para cada una de las siguientes afirmaciones, decir si es cierta o falsa, *razonándolo* en cualquiera de los dos casos.

(DEC es el conjunto de los lenguajes decidibles, SEMIDEC es el conjunto de los lenguajes semidecidibles. L_1 y L_2 son dos lenguajes cualesquiera.)

a. DEC $\subseteq$ SEMIDEC

b. SEMIDEC $\subseteq$ DEC

c. SAT $\in$ P y existe $B \in$ NP $-$ P.

d. Si $L_1 \cup L_2$ es decidible entonces L_1 es decidible ó L_2 es decidible.

e. Sea $A \in$ EXP un problema con entradas codificadas sobre $\Sigma = \{0, 1\}$, definimos el siguiente lenguaje sobre Σ:

$$L_A = \{w \mid w \text{ codifica una entrada con salida SI para } A\}$$

Entonces L_A es un lenguaje decidible.

Nota.- Se puede utilizar sin necesidad de demostrarlo el teorema de Cook (referente al problema SAT), y la clausura de P por $\leq_{\mathrm{m}}^{\mathrm{p}}$.

20. a. Describir un modelo abstracto de cálculo (que no sea la máquina de registros ó RAM).

b. Enunciar la tesis de Turing-Church y explicar su significado.

c. Enunciar la tesis extendida de Turing-Church y explicar su significado.

21. La profesora de Ciencias del Conocimiento ha decidido dar como proyecto el diseño de un algoritmo que, dados un programa p y una entrada x, determine (contestando SI ó NO) si p con entrada x se parará en un tiempo que sea múltiplo de 6 (en 6 pasos, ó en 12, 18, 24, etc.).

El estudiante Manolito (que tiene sobresaliente en MAC) le dice que es imposible encontrar tal algoritmo. Después de consultarlo con los espíritus, la profesora decide cambiar el problema por el de diseñar un algoritmo que, dados un programa p y una entrada x, determine (contestando SI ó NO) si p con entrada x se parará en un tiempo menor o igual que 3000 y que sea múltiplo de 6. Con esto Manolito ya está satisfecho.

Explicar detalladamente (con las demostraciones formales necesarias) el razonamiento de Manolito y por qué la profesora hace este cambio que convence a Manolito.

Pista: Dado un programa, es fácil construir otro que haga lo mismo en tiempo múltiplo de 6. Si utilizas esto en algún momento de tu demostración, pruébalo.

12.2. Problemas de computabilidad

22. Sea

$$A = \{x, y \,|\, Dom(\varphi_x) = \{2z \,|\, z \in \mathrm{Im}(\varphi_y)\}\ \}$$

¿Es A decidible? ¿Es A semidecidible? ¿Es $\bar{A}$ semidecidible?

23. Sea

$$A = \{x, y \,|\quad \forall n(\text{Si } y(n)\downarrow \text{ entonces } x(n)\downarrow \text{ en } \varphi_y(n) \text{ pasos o menos})\}$$

¿Es A decidible? ¿Es A semidecidible? ¿Es $\bar{A}$ semidecidible?

24. a. ¿Existe un algoritmo que resuelva el siguiente problema?

Datos: p, y, k, donde p es un programa, y es una cadena y k un número natural.

Salida: ¿Existen al menos k entradas distintas de p que dan salida y?

b. Sea

$$A = \{x \mid \exists n > 5 \ (x(n)\uparrow \ \vee (x(n)\downarrow \text{ en más de } n \text{ pasos}))\}$$

¿Es A decidible? ¿Es A semidecidible? ¿Es $\bar{A}$ semidecidible?

c. ¿Existe un algoritmo que resuelva el siguiente problema?

Datos: p, donde p es un programa.

Salida: ¿Existe un programa sin llamadas a procedimientos de menos de 100 caracteres que calcula exactamente lo mismo que p?

25. Sea A el siguiente lenguaje:

$$A = \{x, y \mid \exists n \ (\ x(n)\downarrow \ \wedge \ y(n)\uparrow\ )\}$$

a. ¿Es A decidible? ¿Es A semidecidible?

b. ¿Es $\bar{A}$ semidecidible?

26. Sea A el siguiente lenguaje:

$$A = \{x \mid \forall n \ (\ x(n)\downarrow \ \wedge (\varphi_x(n) < \varphi_x(n+1))\ )\}$$

a. ¿Es A decidible? ¿Es A semidecidible?

b. ¿Es $\bar{A}$ semidecidible?

27. Sean L_1, L_2 los siguientes lenguajes:

$$L_1 = \{x, y \mid \mathrm{Dom}(\varphi_x) \neq \mathrm{Dom}(\varphi_y)\},$$
$$L_2 = \{z \mid \forall x \quad z(x)\downarrow \quad \text{y} \quad \varphi_{\varphi_z(x)} \text{ es total}\}.$$

a. ¿Es L_1 decidible? ¿Es L_1 semidecidible? ¿Es $\bar{L}_1$ semidecidible?

b. ¿Es L_2 decidible? ¿Es L_2 semidecidible? ¿Es $\bar{L}_2$ semidecidible?

28. Sean L_1, L_2 los siguientes lenguajes:

$$L_1 = \{m, n \mid m(n)\downarrow \text{ y tarda tiempo múltiplo de } n\},$$
$$L_2 = \{z \mid \forall x\ z(x) \text{ ejecuta la antepenúltima instrucción de } z\}.$$

a. ¿Es L_1 decidible? ¿Es L_1 semidecidible? ¿Es $\bar{L}_1$ semidecidible?
b. ¿Es L_2 decidible? ¿Es L_2 semidecidible? ¿Es $\bar{L}_2$ semidecidible?

29. Demostrar que el siguiente problema no es resoluble con nigún programa:

Datos: Sean p y q dos programas con su codificación habitual.

Salida: ¿p y q hacen exactamente lo mismo, es decir, para cada entrada se cumple que o bien no da salida ninguno de los dos programas o bien ambos dan la misma salida?

30. Sea L el siguiente lenguaje:

$$L = \{x \mid \mathrm{Im}(\varphi_x) \text{ es infinito}\}.$$

a. ¿Es L semidecidible?
b. ¿Es $\bar{L}$ semidecidible?

31. Sean L_1, L_2 los siguientes lenguajes:

$$L_1 = \{z \mid \exists x, y \quad z(x, y) \neq x + y\},$$
$$L_2 = \{x, y \mid \varphi_x(y) = \varphi_y(y) + 1\}.$$

a. ¿Es L_1 decidible? ¿Es L_1 semidecidible?
b. ¿Es $\bar{L}_1$ decidible? ¿Es $\bar{L}_1$ semidecidible?
c. ¿Es L_2 decidible? ¿Es L_2 semidecidible?

32. Sean A y B los siguientes lenguajes:

$$A = \{x \mid \forall n \quad \varphi_x(n) = n + 1\}$$
$$B = \{x \mid \mathrm{Dom}(\varphi_x) \text{ contiene algún número primo}\}.$$

a. ¿Es A decidible? ¿Es A semidecidible? ¿Es $\bar{A}$ semidecidible?
b. ¿Es B decidible? ¿Es B semidecidible? ¿Es $\bar{B}$ semidecidible?

33. Sean A y B los siguientes lenguajes:

$$\begin{aligned} A = &\quad \{x \mid \exists y \quad y \in \mathrm{Dom}(\varphi_x) \ \wedge \ \mathrm{Dom}(\varphi_y) \neq \emptyset\} \\ B = &\quad \{x \mid \exists y \quad y \in \mathrm{Dom}(\varphi_x) \ \wedge \ x \in \mathrm{Dom}(\varphi_y)\}. \end{aligned}$$

a. ¿Es A decidible? ¿Es A semidecidible?

b. ¿Es B decidible? ¿Es B semidecidible?

c. ¿Es $\bar{A}$ semidecidible?

34. Sea A el siguiente lenguaje:

$$A = \{x \mid \forall n \quad x(2n)\downarrow \text{ si y sólo si } x(2n+1)\downarrow\}.$$

a. ¿Es A decidible?

b. ¿Es A semidecidible?

c. ¿Es $\bar{A}$ semidecidible?

35. Sean L_1, L_2 los siguientes lenguajes:

$$L_1 = \{x \mid \mathrm{Dom}(\varphi_x) \text{ contiene todos los números pares}\},$$
$$L_2 = \{x, y \mid \exists z \quad \varphi_z(x) = y\}.$$

a. ¿Es L_1 decidible? ¿Es L_1 semidecidible? ¿Es $\bar{L}_1$ semidecidible?

b. ¿Es L_2 decidible? ¿Es L_2 semidecidible?

36. Sean L_1, L_2 los siguientes lenguajes:

$$L_1 = \{z \mid \forall x \ \ \varphi_z(x) \text{ es un número par}\},$$
$$L_2 = \{z, y \mid \exists x > y \quad \varphi_z(x) \text{ es un número par}\}.$$

a. ¿Es L_1 decidible? ¿Es L_1 semidecidible?

b. ¿Es $\bar{L}_1$ decidible? ¿Es $\bar{L}_1$ semidecidible?

c. ¿Es L_2 decidible? ¿Es L_2 semidecidible?

37. Sean L_1 y L_2 los siguientes lenguajes:

$$L_1 = \{z \mid \exists x \ \ \varphi_z(x) \text{ es un número primo}\},$$
$$L_2 = \{x \mid x(x)\Downarrow \text{ en tiempo menor o igual que } x^2\}.$$

a. ¿Es L_1 decidible? ¿Es L_1 semidecidible? ¿Es $\bar{L}_1$ semidecidible?
b. ¿Es L_2 decidible? ¿Es L_2 semidecidible? ¿Es $\bar{L}_2$ semidecidible?

38. Sean L_1 y L_2 los siguientes lenguajes:

$$L_1 = \{x \mid \text{Im}(\varphi_x) \text{ no contiene ningún número par}\},$$
$$L_2 = \{z, x \mid \exists y \;\; \varphi_z(y) > x\}.$$

a. ¿Es L_1 decidible? ¿Es L_1 semidecidible? ¿Es $\bar{L}_1$ semidecidible?
b. ¿Es L_2 decidible? ¿Es L_2 semidecidible? ¿Es $\bar{L}_2$ semidecidible?

39. Sean L_1 y L_2 los siguientes lenguajes:

$$L_1 = \{x \mid x \text{ calcula la función identidad}\},$$
$$L_2 = \{x, y \mid \varphi_x \equiv \varphi_y\}.$$

a. ¿Es L_1 decidible? ¿Es L_1 semidecidible? ¿Es $\bar{L}_1$ semidecidible?
b. ¿Es L_2 decidible? ¿Es L_2 semidecidible? ¿Es $\bar{L}_2$ semidecidible?

40. Sean L_1 y L_2 los siguientes lenguajes:

$$L_1 = \{z \mid \exists x \;\; \varphi_z(x) \text{ es múltiplo de 7 }\},$$

$$L_2 = \{\, x, y \mid \quad \text{al ejecutar } x \text{ con entrada } y, \text{ en algún IF-THEN-ELSE se toma la opción else}\,\}.$$

a. ¿Es L_1 decidible? ¿Es L_1 semidecidible?
b. ¿Es L_2 decidible? ¿Es L_2 semidecidible? ¿Es $\bar{L}_2$ semidecidible?

41. Sean L_1 y L_2 los siguientes lenguajes:

$$L_1 = \{x \mid K \subseteq \text{Dom}(\varphi_x)\},$$
$$L_2 = \{u, v, w \mid u(v)\downarrow \;\wedge\; v(w)\downarrow \;\wedge \varphi_u(v) = (\varphi_v(w))^2\}.$$

a. ¿Es L_1 decidible? ¿Es L_1 semidecidible?
b. ¿Es L_2 decidible? ¿Es L_2 semidecidible?

42. Sean L_1 y L_2 los siguientes lenguajes:

$$L_1 = \{x \mid \text{Dom}(\varphi_x) \text{ es finito }\},$$
$$L_2 = \{z, y \mid \exists x < y \;\; z(x)\downarrow\}.$$

a. ¿Es L_1 decidible? ¿Es L_1 semidecidible?

b. ¿Es L_2 decidible? ¿Es L_2 semidecidible?

43. Sea f una función TOTAL que cumple:

$$f(x) = \varphi_x(x) \quad \text{si } x \in K$$

¿Es f calculable?

Pista: Estudiar el lenguaje

$$A = \{x \mid \quad \varphi_x(x) = \text{ número de pasos en la ejecución de } x \text{ con entrada } x\}.$$

44. Sea calc el conjunto de todas las funciones calculables. Sean g, h funciones calculables tales que $\text{Dom}(g) = \emptyset$. Sea $H = \text{calc} - \{h\}$.

a. Sea L un lenguaje tal que

$$\text{IND}_g \subseteq L \subseteq \text{IND}_H$$

¿Es L decidible? ¿Es L semidecidible?

b. Sea A el lenguaje:

$$A = \{\, x \mid x(5x+1) \uparrow \}.$$

¿Es A decidible? ¿Es A semidecidible?

45. Dado Π el problema que se enuncia a continuación, ¿existe un programa que resuelve Π?

Datos: p, x, k

Salida: ¿En la ejecución del programa codificado por p con la entrada codificada por x, se repite la penúltima instrucción del fuente al menos k veces?

Codificación de las entradas: sobre el alfabeto $\{0, 1\}$, con la codificación usual de programas, entradas y números naturales.

46. Sean L_1 y L_2 los siguientes lenguajes:

$$L_1 = \{x \mid \exists\, y \text{ tal que } \varphi_x \equiv \varphi_y \text{ y } x \neq y\},$$
$$L_2 = \{z, n \mid \exists\, x \text{ tal que } \varphi_z(x) = 1 \text{ y } |x| = n\}.$$

a. ¿Es L_1 decidible? ¿Es L_2 decidible?

b. ¿Es L_2 semidecidible?

c. Sea f la función definida como:

$$f(x) = x + 5 \text{ si } x \notin L_2$$

¿Es f calculable?

47. Sean L_1 y L_2 los siguientes lenguajes:

$$L_1 = \{x \,|\, \varphi_x(5) \neq 30\},$$

$$L_2 = \{z, x, t \,|\quad z \text{ codifica un programa que, con entrada } x, \text{ para en tiempo menor o igual que } t\}.$$

a. ¿Es L_1 decidible? ¿Es L_2 decidible?

b. ¿Es L_1 semidecidible?

c. Sea f la función definida como:

$$f(x) = x + 5 \text{ si } x \notin L_1$$

¿Existe una máquina de Turing que calcule f?

48. Sean L_1 y L_2 los siguientes lenguajes:

$$L_1 = \{\, x \,|\, \mathrm{Im}(\varphi_x) \,=\, \mathrm{Dom}(\varphi_x) \,\},$$

$$L_2 = \{\, x, y \,|\, \mathrm{Im}(\varphi_x) \,=\, \mathrm{Dom}(\varphi_y) \,\}.$$

a. ¿Es L_1 decidible?

b. ¿Es L_2 semidecidible?

c. Sea χ_{L_1} la función característica de L_1. ¿Existe una máquina de Turing que calcule χ_{L_1}? ¿Existe una máquina de Turing que calcule χ_{L_2}?

12.3. Problemas de complejidad

49. Sea X el siguiente problema. Demostrar que es NP-completo.

Datos: $G = (V, A)$ un grafo NO DIRIGIDO de n vértices, $k \in \mathbb{N}$ con $k \leq n$.

Salida: ¿Existe un conjunto de k vértices $U \subseteq V$ tal que para todo $u, v \in U$ se cumple que $\{u, v\} \notin A$?

Codificación de las entradas: Sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $r \in \{0,1\}^*$ es el número de vértices escrito en binario, $s \in \{0,1\}^*$ es la matriz de adyacencia de G escrita por filas, y x es k en binario, entonces la codificación de la entrada G, k es la palabra r, s, x.

50. Sea X el siguiente problema. Demostrar que es NP-completo.

Datos: $n, m, p_1, \ldots, p_n, v_1, \ldots, v_n, B, K \in \mathbb{N}$, $U_1, \ldots, U_m$ subconjuntos disjuntos de $\{1, \ldots, n\}$ tales que $U_1 \cup \ldots \cup U_m = \{1, \ldots, n\}$.

Salida: ¿Existe un conjunto $A \subseteq \{1, \ldots, n\}$ *que contenga como mucho un elemento de cada conjunto* U_i ($\|A \cap U_i\| \leq 1 \quad \forall i$) y que cumpla las siguientes condiciones:

$$\sum_{j \in A} p_j \leq B$$

$$\sum_{j \in A} v_j \geq K$$

Codificación de las entradas: Sobre el alfabeto $\Sigma = \{0, 1, ,\}$, los $2n+4$ números naturales se escriben en binario y se separan por comas, seguidos de los m subconjuntos codificados cada uno con n bits y separados por comas.

51. a. Sea X el siguiente problema. Demostrar que es NP-completo.

Datos: G, donde G es un grafo dirigido de n vértices.

Salida: ¿Existe un camino de G con longitud n que pase por todos los vértices de G (es decir, un camino que pase por uno de los vértices 2 veces y por todos los demás una vez)?

Codificación de las entradas: Utilizando la matriz de adyacencia del grafo.

b. Sea Y el siguiente problema. Demostrar que es NP-completo.

Datos: G, k, donde G es un grafo dirigido de n vértices y k es un número natural con $k \leq n$.

Salida: ¿Existe un camino de G que pase por uno de los vértices exactamente k veces y por todos los demás una sola vez?

Codificación de las entradas: Utilizando la matriz de adyacencia del grafo.

52. Sea NUMSAT el siguiente problema. Demostrar que es NP-completo.

Datos: X, F, k, donde X es un conjunto de n variables, F una fórmula booleana sobre X en forma normal conjuntiva (es decir, escrita como conjunción de cláusulas) y k es un número natural $k \leq n$.

Salida: ¿Existen al menos k asignaciones distintas de X que hacen cierta F?

Codificación de las entradas: La misma que la de las entradas de SAT.

53. Sea 2dHAM el siguiente problema. Demostrar que es NP-completo.

Datos: $G = (V, A)$ un grafo *DIRIGIDO*.

Salida: ¿¿G tiene más de un camino hamiltoniano?

Codificación de las entradas: Utilizando listas de adyacencia.

54. Sea CUBRECICLOS el siguiente problema. Demostrar que es NP-completo.

Datos: $G = (V, A)$ un grafo *dirigido*, $k \in \mathbb{N}$ con $k \leq n$ (donde n es el número de vértices de G).

Salida: ¿Existe un conjunto de k vértices $U \subseteq V$ que cubre todos los circuitos de G? (U cubre todos los circuitos de G si para todo $c_1, \ldots, c_a$ circuito de G existe algún i entre 1 y a con $c_i \in U$).

Codificación de las entradas: Utilizando la matriz de adyacencia del grafo.

Pista: utilizar dVERTEX_COVER.

55. Sea INECUACIONES el siguiente problema. Demostrar que es NP-completo.

Datos:

$n \in \mathbb{N}$ el número de inecuaciones, $m \in \mathbb{N}$ el número de incógnitas,

$a_{i,j} \in Q$ para cada $1 \leq i \leq n$, $0 \leq j \leq m$ los coeficientes racionales,

$\sigma_i \in \{\leq, \geq\}$ para cada $1 \leq i \leq n$, las desigualdades de las inecuaciones. (Por ejemplo para $-7/3 + 3x_1 \geq 0$, es $\geq$)

Salida: ¿Existe una solución al sistema de inecuaciones *formada sólo por ceros y unos*? Es decir, $v_1, \ldots, v_m \in \{0, 1\}$ que cumplan

$$\begin{array}{llll} a_{1,0} + a_{1,1}v_1 + & a_{1,2}v_2 + & \ldots + a_{1,m}v_m & \sigma_1\ 0 \\ a_{2,0} + a_{2,1}v_1 + & a_{2,2}v_2 + & \ldots + a_{2,m}v_m & \sigma_2\ 0 \\ \ldots & & & \\ a_{n,0} + a_{n,1}v_1 + & a_{n,2}v_2 + & \ldots + a_{n,m}v_m & \sigma_n\ 0 \end{array}$$

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,, (,), \leq, \geq\}$, se escriben los números racionales como (a, b, s), a y b numerador y denominador en binario y s un bit extra para el signo, los naturales en binario, las desigualdades con el símbolo correspondiente y todos ellos separados por comas.

Pista: utilizar MOCHILA o PARTICION.

56. Sea GTSP (problema del viajante generalizado) el siguiente problema:

Datos: $n \in \mathbb{N}$ el número de ciudades,

$d(i, j) \in \mathbb{N}$ para cada $1 \leq i, j \leq n$ las distancias entre cada dos ciudades (tales que $d(i, j) = d(j, i)$ y $d(i, i) = 0$ para todo i, j),

$k \in \mathbb{N}$.

Salida: ¿Existe un camino cualquiera (se admiten repeticiones) que pasa por todas las ciudades y que tiene una longitud total menor o igual que k?

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, se escriben los números naturales en binario y separados por comas.

Demostrar que GTSP es *NP-difícil* (es decir, que $A \leq_{\mathrm{m}}^{\mathrm{p}} \mathrm{GTSP}$ para algún NP-completo A).

Pista: TSP no es una buena elección para esta reducción.

57. Sea IPL (programación lineal entera) el siguiente problema.

Datos: $n \in \mathbb{N}$ el número de ecuaciones, $m \in \mathbb{N}$ el número de incógnitas,

$a_{i,j} \in Z$ para cada $1 \leq i \leq n$, $0 \leq j \leq m$ los coeficientes enteros,

$c \in \mathbb{N}$.

Salida: ¿Existe una solución entera $v_1, \ldots, v_m \in Z$ al sistema de inecuaciones:

$$\begin{array}{llll} a_{1,0} + a_{1,1}x_1 + & a_{1,2}x_2 + & \ldots + a_{1,m}x_m & \geq 0 \\ a_{2,0} + a_{2,1}x_1 + & a_{2,2}x_2 + & \ldots + a_{2,m}x_m & \geq 0 \\ \ldots & & & \\ a_{n,0} + a_{n,1}x_1 + & a_{n,2}x_2 + & \ldots + a_{n,m}x_m & \geq 0 \end{array}$$

que cumpla valor_absoluto$(v_j) \leq c$ para $1 \leq j \leq m$?

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, se escriben los números en binario (con un bit extra para el signo) y separados por comas.

Demostrar que IPL $\in$ NP.

58. Sea CAMINO_GUIADO el siguiente problema. Demostrar que es NP-completo.

CAMINO_GUIADO

Datos: G un grafo NO DIRIGIDO de n vértices

$k \in \mathbb{N}$ con $2 \leq k \leq n$.

$v_1, \ldots, v_k$, k vértices distintos de G.

Salida: ¿Existe $C = (c_1, \ldots, c_n)$ un camino hamiltoniano de G y existen $1 = i_1 < i_2 < \ldots < i_k = n$ de forma que $v_1 = c_{i_1}$, $v_2 = c_{i_2}$, $\ldots$, $v_k = c_{i_k}$ (es decir, C empieza en v_1, pasa siempre por v_j antes que por v_{j+1} y acaba en v_k)?

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, se escribe el número de vértices en binario, seguido de coma, seguido de la matriz de adyacencia de G escrita por filas, seguido de coma y por último k, $v_1, \ldots, v_k$ en binario y separados por comas.

59. Sean SATGRAL y SATSIMPL los siguientes problemas. Demostrar que ambos son NP-completos.

SATGRAL

Datos: X, F, donde $X = \{x_1, \ldots, x_n\}$ es un conjunto de variables y F una fórmula booleana cualquiera escrita con las variables de X, conectivas $\wedge$, $\vee$, $\neg$ y los paréntesis necesarios.

Salida: ¿Existe una asignación de X que hace cierta F?

Codificación de las entradas: Sobre el alfabeto $\{\wedge, \vee, \neg, (,), 0, 1\}$ codificando cada variable por su número en binario y las conectivas y paréntesis con los símbolos correspondientes.

Por ejemplo $(x_5 \vee (x_1 \wedge (\neg x_6 \vee x_4)))$ se codifica como $(101 \vee (1 \wedge (\neg 110 \vee 100)))$

SATSIMPL

Datos: X, F, donde X es un conjunto de variables, y F una fórmula booleana en forma normal conjuntiva (es decir, escrita como conjunción de cláusulas), y tal que *en ninguna cláusula aparece la misma variable más de una vez.*

Salida: ¿Existe una asignación de X que hace cierta F?

Codificación de las entradas: La misma que la de las entradas de SAT.

60. Sea 3HAM el siguiente problema. Demostrar que es NP-completo.

3HAM

Datos: G un grafo NO DIRIGIDO de n vértices

a, b, c, tres vértices de G.

Salida: ¿Existe un camino hamiltoniano de G (es decir, un camino simple que pasa por todos los vértices) que empiece en el vértice a, acabe en el vértice c, y tenga el vértice b en la posición central (es decir, en el lugar $\lfloor n/2 \rfloor$)?

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $u \in \{0,1\}^*$ es el número de vértices escrito en binario, $v \in \{0,1\}^*$ es la matriz de adyacencia de G escrita por filas, y x, y, $z \in \{0,1\}^*$ son a, b y c en binario, entonces la codificación de la entrada G, a, b, c es la palabra u, v, x, y, z.

61. Sean MPATH y SPATH los siguientes problemas:

MPATH

Datos: G un grafo NO DIRIGIDO,

r, s, dos vértices de G,

$k \in \mathbb{N}$ con $k \leq n$.

Salida: ¿Existe un camino simple de G (es decir, sin vértices repetidos) de longitud $\geq k$ que empiece en el vértice r y acabe en el vértice s?

SPATH

Datos: G un grafo NO DIRIGIDO,

r, s, dos vértices de G,

$k \in \mathbb{N}$ con $k \leq n$.

Salida: ¿Existe un camino simple de G (es decir, sin vértices repetidos) de longitud $\leq k$ que empiece en el vértice r y acabe en el vértice s?

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $u \in \{0,1\}^*$ es el número de vértices escrito en binario, $v \in \{0,1\}^*$ es la matriz de adyacencia de G escrita por filas, y x, y, $z \in \{0,1\}^*$ son r y s y k en binario, entonces la codificación de la entrada G, r, s, k es la palabra u, v, x, y, z.

a. Demostrar que MPATH es NP-completo. *Pista para a):* comparar con HAM.

b. Demostrar que SPATH $\in$ P. *Pista para b):* la existencia de un camino cualquiera de longitud $\leq k$ de r a s implica la existencia de un camino SIMPLE de longitud $\leq k$ de r a s.

62. Sea MULTI-MOCHILA el siguiente problema. Demostrar que es NP-completo.

Datos: $n, M \in \mathbb{N}$;

$U_{1,1}, \ldots, U_{1,M}, U_{2,1}, \ldots, U_{n,M}$ un total de $n \cdot M$ números naturales;

$C_1, \ldots, C_M \in \mathbb{N}$.

Salida: ¿Existe un conjunto de objetos $A \subseteq \{1, \ldots, n\}$ que cumpla para todo j desde 1 hasta M

$$\begin{array}{rl} \text{Si } j \text{ es impar:} & \sum_{i \in A} U_{i,j} \leq C_j \\ \text{Si } j \text{ es par:} & \sum_{i \in A} U_{i,j} \geq C_j \quad ? \end{array}$$

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, los $n \cdot M + M + 2$ números naturales que componen una entrada se escriben en binario y se separan por comas.

63. Sea 7-PARTICION el siguiente problema. Demostrar que es NP-completo.

Datos: $n \in \mathbb{N}$ el número de objetos;

$p_1, p_2, \ldots, p_n \in \mathbb{N}$, los pesos de los objetos.

Salida: ¿Existen siete conjuntos disjuntos de objetos $A_1, \ldots, A_7$ que cumplan para todo k desde 1 hasta 7:

$$\sum_{i \in A_k} p_i = \frac{\sum_{1 \leq i \leq n} p_i}{7} \ ?$$

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, todos los números naturales escritos en binario y separados por comas.

64. Sea BOTTLENECK TRAVELING SALESMAN el siguiente problema. Demostrar que es NP-completo.

Datos: n, $d(1,1), d(1,2), \ldots, d(n,n)$, B números naturales.

Representan n ciudades y $d(i,j)$ es la distancia de la ciudad i a la ciudad j (tales que $d(i,j) = d(j,i)$ y $d(i,i) = 0$ para todo i,j).

Salida: ¿Existe un camino simple $C = (c_1, \ldots, c_n)$ (es decir, que pase por todas las ciudades una sola vez) y que cumpla que para todos los i desde 1 hasta $n-1$ $d(c_i, c_{i+1}) \leq B$?

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, todos los números naturales escritos en binario y separados por comas.

Pista: Relacionarlo con el problema HAM (caminos hamiltonianos sobre grafos no dirigidos). Sólo algunas de las conexiones entre ciudades son "útiles".

65. Sea HAMILTONIAN PATH BETWEEN TWO VERTICES el siguiente problema. Demostrar que es NP-completo.

Datos: G un grafo NO DIRIGIDO de vértices $V = \{1, \ldots, n\}$ y aristas $A = \{a_1, \ldots, a_k\}$;

$r, s \in V$, dos vértices de G.

Salida: ¿Existe un camino hamiltoniano de G (es decir, sin vértices repetidos y que pase por todos los vértices) que empiece en el vértice r y acabe en el vértice s?

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $u \in \{0,1\}^*$ es n escrito en binario, $v \in \{0,1\}^*$ es la matriz de adyacencia de G escrita por filas, y x, $y \in \{0,1\}^*$ son r y s en binario, entonces la codificación de la entrada G, r, s es la palabra u, v, x, y.

Pista: Relacionarlo con el problema HAM (caminos hamiltonianos sobre grafos no dirigidos). La reducción utilizada añade 2 vértices nuevos a cada grafo que son principio y final de cualquier camino hamiltoniano.

66. Sea X el problema que se enuncia a continuación. Demostrar que X es NP-completo.

Datos: $n, p_1, p_2, \ldots, p_n, P, k \in \mathbb{N}$

Salida: ¿Existen $A_1, \ldots, A_k$ subconjuntos que cumplan:

$$A_1 \cup \ldots \cup A_k = \{1, \ldots, n\}$$

$$\sum_{s=1}^{k} \left(\sum_{r \in A_s} p_r \right)^3 \leq P \ ?$$

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, cada número natural escrito en binario y separados por comas.

Pista: Relacionarlo con el problema PARTICION.

67. Sea MAYOR-SUBGRAFO el siguiente problema. Demostrar que es NP-completo.

Datos: G grafo NO DIRIGIDO de vértices $V = \{1, \ldots, n\}$ y aristas A,

G_2 grafo no dirigido de vértices $V_2 = \{1, \ldots, r\}$ $(r \leq n)$ y aristas A_2,

$k \in \mathbb{N}$ $(k \leq n^2)$.

Salida: ¿Existe un subconjunto X de k aristas de G_2 (es decir, $X \subseteq A_2$, con $\#X = k$) tal que $H = (V_2, X)$ es un subgrafo de G?

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $u \in \{0,1\}^*$ es la matriz de adyacencia de G escrita por filas, $v \in \{0,1\}^*$ es la matriz de adyacencia de G_2 escrita por filas, y $w \in \{0,1\}^*$ es k en binario entonces la codificación de la entrada G, G_2, k es la palabra u, v, w.

Pista: Relacionarlo con el problema CLIQUE.

Nota: $H = (V_2, X)$ es un subgrafo de $G = (V, A)$ si existe $V' \subseteq V$ y $f : V_2 \to V'$ biyectiva tal que para cada $u, v \in V_2$, $\{u, v\} \in X$ si y sólo si $\{f(u), f(v)\} \in A$.

68. Sea LONGEST-PATH el siguiente problema. Demostrar que es NP-completo.

Datos: G un grafo DIRIGIDO de vértices $V = \{1, \ldots, n\}$ y aristas $A \subseteq V \times V$

$k \in \mathbb{N}$, tal que $k \leq n$.

Salida: ¿Existe un camino simple (es decir, sin vértices repetidos) de G que empiece en el vértice 1 y tenga longitud k?

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, \ , \ , \ ; \ \}$. Si $v \in \{0, 1, \ , \ , \ ; \ \}^*$ es la codificación mediante LISTAS DE ADYACENCIA de G, y $w \in \{0, 1\}^*$ es k en binario entonces la codificación de la entrada G, k es la palabra v, w

69. a. Sea FSAT el mismo problema que SAT pero restringido a fórmulas en las que cada variable aparece como máximo 2 veces. Demostrar que FSAT está en P.

Pista: Podemos transformas las entradas en fórmulas equivalentes en que cada variable aparezca exactamente dos veces, una afirmada y otra negada, y en cláusulas distintas, y además todas las cláusulas tengan al menos dos literales (hay que justificarlo). Una vez hecho eso, hay un método para ir asignando valores a las variables (¿cuál?).

b. Sea 3-PARTICION el siguiente problema. Demostrar que es NP-difícil.

Datos: $n \in \mathbb{N}$ el número de objetos;

$p_1, p_2, \ldots, p_n \in \mathbb{N}$, los pesos de los objetos.

Salida: ¿Existe un conjunto de objetos $A \subseteq \{1, \ldots, n\}$ que cumpla

$$\sum_{i \in A} p_i = \frac{\sum_{1 \leq i \leq n} p_i}{3} \quad ?$$

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, los $n+1$ numeros naturales que componen una entrada $n, p_1, \ldots, p_n$ se escriben en binario y se separan por comas.

70. Sea X el problema que se enuncia a continuación. Demostrar que X es NP-completo.

Datos: $n, p_1, p_2, \ldots, p_n, P, k \in \mathbb{N}$

Salida: ¿Existen $A_1, \ldots, A_k$ subconjuntos de $\{1, \ldots, n\}$ que cumplan:

$$A_1 \cup \ldots \cup A_k = \{1, \ldots, n\}$$

$$\sum_{s=1}^{k} \left(\sum_{r \in A_s} p_r \right)^2 \leq P \ ?$$

Codificación de las entradas: sobre el alfabeto $\{0, 1, ,\}$, cada número natural escrito en binario y separados por comas.

Ayuda Relacionarlo con el problema PARTICION.

71. Sea 2-SAT el problema que se enuncia a continuación. Demostrar que 2-SAT está en P.

Datos: X, F, donde X es un conjunto de variables, y F una fórmula booleana en forma normal conjuntiva (es decir, escrita como conjunción de cláusulas), y *cada cláusula tiene exactamente 2 literales.*

Salida: ¿Existe una asignación de X que hace cierta F?
Codificación de las entradas: como las entradas de SAT.

Ayuda Para hacer cierta una fórmula F:

a. Si una variable sólo aparece afirmada (o sólo negada) en F es conveniente asignarle un valor (¿cuál?).

b. Si una cláusula es de la forma $y \vee \neg y$ es siempre cierta. Si una cláusula es de la forma $y \vee y$ (ó de la forma $\neg y \vee \neg y$), hay que asignar a y un valor (¿cuál?).

c. Si tenemos varias cláusuals de la forma:

$$(l_1 \vee \neg l_2), (l_2 \vee \neg l_3), \ldots, (l_{m-1} \vee \neg l_m), (l_m \vee \neg l_1)$$

con $l_1, \ldots, l_m$ literales distintos (y con variables distintas), entonces para hacerlas todas ciertas hay que dar el mismo valor a todos los $l_1, \ldots, l_m$, luego podemos sustituir $l_2, \ldots, l_m$ por l_1 en toda F.

d. Si tenemos varias cláusuals de la forma:

$$(l_1 \vee \neg l_2), (l_2 \vee \neg l_3), \ldots, (l_{m-1} \vee \neg l_m), (l_m \vee l_1)$$

con $l_1, \ldots, l_m$ literales distintos (y con variables distintas), entonces para hacerlas todas ciertas hay que dar un valor a l_1 (¿cuál?).

72. Para cada uno de los problemas que se enuncian a continuación, responder a las siguientes preguntas: ¿Está en P? ¿Está en NP? ¿Está en EXP?

Las respuestas posibles son tres: *Sí*, *No* y *No se sabe, porque es NP-completo.*

Se puede utilizar que el problema clásico dCLIQUE es NP-completo (así como cualquiera de los otros problemas NP-completos vistos en clase).

a. **Datos:** G un grafo dirigido de vértices $V = \{1, \ldots, n\}$ y aristas $A \subseteq V \times V$;
$k \in \mathbb{N}$, tal que $k \leq n$;
$r \in \mathbb{N}$, tal que $r \leq n$.
Salida: ¿Existen $V_1, \ldots, V_r$ subconjuntos de V que cumplan 1), 2) y 3)?
1) para todo $i \neq j$ $(1 \leq i, j \leq r)$, $V_i \cap V_j = \emptyset$,
2) $V_1 \cup \ldots \cup V_r = V$,
3) para todo i desde 1 hasta r, el subgrafo de G formado por los vértices V_i tiene un subgrafo completo de al menos k vértices.
Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $u \in \{0,1\}^*$ es n escrito en binario, $v \in \{0,1\}^*$ es la matriz de adyacencia de G escrita por filas, $w \in \{0,1\}^*$ es k en binario, y $z \in \{0,1\}^*$ es r en binario entonces la codificación de la entrada G, k, r es la palabra u, v, w, z.

b. **Datos:** G un grafo dirigido de vértices $V = \{1, \ldots, n\}$ y aristas $A \subseteq V \times V$;
$a, b \in V$;
$k \in \mathbb{N}$, tal que $k \leq n$.
Salida: ¿Existe un camino del vértice a al vértice b de longitud menor o igual que k?
Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $u \in \{0,1\}^*$ es n escrito en binario, $v \in \{0,1\}^*$ es la matriz de adyacencia de G escrita por filas, y x, y, $w \in \{0,1\}^*$ son a, b y k en binario, entonces la codificación de la entrada G, a, b, k es la palabra u, v, x, y, w.

73. Para el problema que se enuncia a continuación, responder a las siguientes preguntas: ¿Está en P? ¿Está en NP? ¿Está en EXP?

Las respuestas posibles son tres: *Sí*, *No* y *No se sabe, porque es NP-completo*.

Datos: G un grafo dirigido de vértices $V = \{1, \ldots, n\}$ y aristas $A \subseteq V \times V$;
$k \in \mathbb{N}$, tal que $k \leq n$.
Salida: ¿Existe un camino de longitud mayor o igual que k que no pase dos veces por un mismo vértice?

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $u \in \{0,1\}^*$ es n escrito en binario, $v \in \{0,1\}^*$ es la matriz de adyacencia de G escrita por filas, y $w \in \{0,1\}^*$ es k en binario, entonces la codificación de la entrada G, k es la palabra u, v, w.

Pista para 3.: comparar con el problema clásico de existencia de un circuito hamiltoniano (dHAM).

74. Para cada uno de los siguientes problemas, responder a las tres preguntas siguientes: ¿Está en P? ¿Está en NP? ¿Está en EXP?

Las respuestas posibles son tres: *Sí*, *No* y *No se sabe, porque es NP-completo*.

a. (Este problema es una variación del clásico PARTICION.)
Datos: $n \in \mathbb{N}$ el número de objetos;

$p_1, p_2, \ldots, p_n \in \mathbb{N}$, los pesos de los objetos.

Salida: ¿Existen dos conjuntos de objetos A_1, A_2 que cumplan las tres condiciones siguientes?:

1) $A_1 \cup A_2 = \{1, \ldots, n\}$.

2) Para todo i, j entre 1 y n, si $i \in A_1$ y $j \in A_2$, entonces $i < j$.

3) $\sum_{i \in A_1} p_i = \sum_{i \in A_2} p_i$.

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, los $n+1$ numeros naturales que componen una entrada $n, p_1, \ldots, p_n$ se escriben en binario y se separan por comas.

b. **Datos:** $G = (V, A)$ un grafo dirigido con $V = \{1, \ldots, n\}$ y aristas $A \subseteq V \times V$;

$k \in \mathbb{N}$, tal que $k \leq n$;

$r \in \mathbb{N}$, tal que $r \leq n$.

Salida: ¿Existen $V_1, \ldots, V_r$ subconjuntos de V que cumplan 1) y 2)?

1) $V_1 \cup \ldots \cup V_r = V$,

2) para todo i desde 1 hasta r, el subgrafo de G formado por los vértices V_i tiene un cubrimiento de k vértices como máximo.

Codificación de las entradas: sobre el alfabeto $\Sigma = \{0, 1, ,\}$, si $u \in \{0,1\}^*$ es n escrito en binario, $v \in \{0,1\}^*$ es la matriz de adyacencia de G escrita por filas, $w \in \{0,1\}^*$ es k en binario, y $z \in \{0,1\}^*$ es r en binario entonces la codificación de la entrada G, k, r es la palabra u, v, w, z.

Pista para 3. b): comparar con el problema Vertex Cover.

Índice general