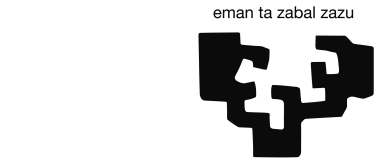


# Lenguajes, Computación y Sistemas Inteligentes

Grado en Ingeniería Informática de Gestión y Sistemas de Información

Escuela de Ingeniería de Bilbao (UPV/EHU)

2º curso

Curso académico 2023-2024

# Tema 1: Introducción

JOSÉ GAINTZARAIN IBARMIA

Departamento de Lenguajes y Sistemas Informáticos

Última actualización: 04 - 09 - 2023

# Índice general

# Índice de figuras

# Índice de tablas

# Tema 1

# Introducción

# 1.1.

# Presentación

En la asignatura “Lenguajes, Computación y Sistemas Inteligentes” se abordará el estudio formal de la computabilidad y se presentarán los fundamentos básicos para llevar a cabo dicho estudio. Para ello, se mostrarán algunos resultados bien establecidos y se mencionarán problemas abiertos y nuevos enfoques que pretenden ampliar los límites actuales de la computabilidad. El conocer los resultados bien establecidos y los límites actuales sirve para saber cómo se ha enfocado el estudio de la computabilidad, qué mecanismos y técnicas se han utilizado y hasta dónde se ha llegado. Dicho conocimiento facilita, además, la comprensión de los nuevos enfoques que están siendo investigados dentro de la ciencia de la computación.

# 1.2.

# Motivación

La informática (o ciencia de la computación) es la rama de la ciencia que estudia el tratamiento automático de la información.

El objetivo último de la informática sería desarrollar sistemas informáticos que pudieran realizar de forma más eficiente todos los cálculos (o tareas de procesamiento de información) que pueden llevar a cabo los humanos y, también, realizar de forma eficiente todos los cálculos (o tareas de procesamiento de información) que pueden plantear los humanos pero que son difíciles de llevar a cabo para los humanos.

Aunque en el campo de la informática se ha avanzado mucho, de momento no se ha podido llegar a tener sistemas informatizados para todos los cálculos o tareas de procesamiento de información planteados por los humanos. Además, los sistemas informatizados desarrollados para resolver algunas tareas, son muy ineficientes, hasta el punto de que no resultan útiles en la práctica y no se conoce si es posible desarrollar sistemas más eficientes para esas tareas. Por tanto, se han encontrado dos tipos de dificultades:

(1) Dificultades relacionadas con la **computabilidad**: Tareas para las que no se encuentra la manera de desarrollar un sistema informático que las realice. Por ejemplo, se ha demostrado formalmente que no es posible diseñar un programa $P$ que reciba, como datos de entrada, otro programa $Q$ y un dato en entrada $d$ para $Q$ y decida (respondiendo con un "Sí" o con un "No") si $Q$ va a terminar el cálculo correspondiente a $d$. En la figura 1.2.1 de la página 15 se muestra la especificación correspondiente a dicho problema.

Que $Q$ no termine el cálculo correspondiente a $d$ significa que, tras recibir $d$ como dato de entrada, $Q$ se introduce en un proceso infinito y nunca para (nunca se detiene) y nunca devuelve una respuesta o resultado para $d$.

Ese programa $P$ sería muy interesante, porque si lo tuviésemos, dados un programa $Q$ y un dato $d$ para $Q$, si vemos que $Q$ tarda en responder, entonces podríamos preguntar a $P$ si $Q$ va a terminar o no. Pero como ya se ha dicho antes, no es posible diseñar un

programa $P$ que sea capaz de realizar dicha tarea.

Nótese que lo de pasar un programa $Q$ a otro programa $P$ no es un problema en sí. Por ejemplo, un compilador de Haskell —o cualquier otro lenguaje de programación— es un programa $K$ al cual se le pasa un programa $Q$ escrito en Haskell y $K$ decide si $Q$ está correctamente escrito en Haskell. En la figura 1.2.2 de la página 15 se muestra la especificación correspondiente a dicho problema.

Por tanto, sí es posible decidir si $Q$ es un programa de Haskell sintácticamente correcto, pero no es posible decidir si $Q$ terminará o no terminará el cálculo correspondiente a cualquier dato $d$.

(2) Dificultades relacionadas con la **complejidad computacional**: Tareas para las que, habiendo encontrado la manera de desarrollar un sistema informático que las realice, dicho sistema es demasiado ineficiente, y no se ha encontrado la manera de desarrollar un sistema informático eficiente. Por ejemplo, es posible diseñar un programa $S$ para decidir si una fórmula de la lógica proposicional es satisfactible o no. Es decir, dada una fórmula $\varphi$ de la lógica proposicional y el número de variables proposicionales $n$ que hay en esa fórmula, $S$ decide (respondiendo con un "Sí" o con un "No") si para alguna valoración de las $n$ variables proposicionales, la fórmula $\varphi$ es cierta. En la figura 1.2.3 de la página 15 se muestra la especificación correspondiente a dicho problema.

El método que se conoce para realizar este cálculo, consiste en ir probando con todas las valoraciones posibles para las $n$ variables proposicionales involucradas, hasta encontrar una valoración que haga cierta a la fórmula proposicional $\varphi$ o hasta agotar todas las posibilidades, en cuyo caso se sabrá que $\varphi$ no es satisfactible. En el peor caso, este método requiere generar $2^n$ valoraciones.

Si consideramos la fórmula proposicional $\gamma$, definida como $(x_1 \vee \neg x_3) \wedge (x_2 \vee x_3)$, donde el número de variables proposicionales es 3, la respuesta de $S(\gamma, 3)$ será "Sí" porque para la valoración $x_1 = False$, $x_2 = True$ y $x_3 = False$, la fórmula $\gamma$ es cierta. Hay también otras valoraciones que hacen cierta a $\gamma$. En cambio, si consideramos la fórmula proposicional $\psi$, definida como $(\neg x_1) \wedge (x_1 \vee x_3) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3)$, donde el número de variables proposicionales es también 3, la respuesta de $S(\psi, 3)$ será "No" porque ninguna valoración de $x_1$, $x_2$ y $x_3$ hace cierta a $\psi$. En la figura 1.2.4 de la página 17 se muestran las ocho valoraciones posibles para $\gamma$ y $\psi$.

Si el programa $S$ sigue el orden mostrado en la figura 1.2.4 de la página 17, entonces en el caso de $\gamma$, tendría que generar las primeras tres valoraciones, puesto que $v_3$ es la primera valoración que hace que $\gamma$ sea cierta. Por su parte, en el caso de $\psi$, tendría que generar las ocho valoraciones, para terminar constatando que ninguna valoración hace que $\psi$ sea cierta. Si en vez de 3 variables proposicionales, se tienen 150, generar las $2^{150}$

$$P(Q,d) = \begin{cases} \text{Sí} & \text{si } Q \text{ termina el cálculo para } d \\ \text{No} & \text{si } Q \text{ no termina el cálculo para } d \end{cases}$$

**Figura 1.2.1.** Especificación del problema que consiste en decidir si el programa Q termina el cálculo correspondiente al dato $d$. Este problema no es computable, es decir, no es posible implementar el programa $P$.

$$K(Q) = \begin{cases} \text{Sí} & \text{si } Q \text{ es un programa correcto en Haskell} \\ \text{No} & \text{si } Q \text{ no es un programa correcto en Haskell} \end{cases}$$

**Figura 1.2.2.** Especificación del problema que consiste en decidir si el programa Q es sintácticamente correcto en Haskell. Este problema es computable, es decir, el programa $K$ es implementable.

$$S(\varphi,n) = \begin{cases} \text{Sí} & \text{si } \varphi \text{ es cierta para alguna valoración de las } n \\ & \text{variables proposicionales que aparecen en } \varphi. \\ \text{No} & \text{si } \varphi \text{ no es cierta para ninguna valoración de las } n \\ & \text{variables proposicionales que aparecen en } \varphi. \end{cases}$$

**Figura 1.2.3.** Especificación del problema que consiste en decidir si la fórmula proposicional $\varphi$, que utiliza $n$ variables, es satisfactible. Este problema es computable pero intratable, es decir, el programa $S$ es implementable pero solo se han conseguido implementaciones muy ineficientes.

valoraciones posibles es muy ineficiente, puesto que requeriría mucho tiempo. Por tanto, se considera que el programa $S$ no es práctico, es decir, no es útil. Al no disponer de un método eficiente para el problema de la satisfactibilidad de fórmulas proposicionales, este problema se clasifica como **intratable**.

Ante las dificultades encontradas, se ha querido estudiar desde un punto de vista teórico tanto la existencia de problemas computables e incomputables como la existencia de problemas que, siendo computables, parecen ser —o son— intratables. Se pretende averiguar dónde están los límites de la computación: ¿todavía no hemos desarrollado la metodología adecuada o existen límites teóricos insalvables?

Fórmulas proposicionales $\gamma$ y $\psi$:

$$\begin{aligned} \gamma &\equiv (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3) \\ \psi &\equiv (\neg x_1) \wedge (x_1 \vee x_3) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3) \end{aligned}$$

Valoraciones para las variables proposicionales $x_1$, $x_2$ y $x_3$ y los correspondientes valores de $\gamma$ y $\psi$:

| | $x_1$ | $x_2$ | $x_3$ |
|---|---|---|---|
| $v_1$ | $F$ | $F$ | $F$ |
| $v_2$ | $F$ | $F$ | $T$ |
| $v_3$ | $F$ | $T$ | $F$ |
| $v_4$ | $F$ | $T$ | $T$ |
| $v_5$ | $T$ | $F$ | $F$ |
| $v_6$ | $T$ | $F$ | $T$ |
| $v_7$ | $T$ | $T$ | $F$ |
| $v_8$ | $T$ | $T$ | $T$ |

| $\gamma$ | $\psi$ |
|---|---|
| $F$ | $F$ |
| $F$ | $F$ |
| $T$ | $F$ |
| $F$ | $F$ |
| $F$ | $F$ |
| $T$ | $F$ |
| $T$ | $F$ |
| $T$ | $F$ |

**Figura 1.2.4.** Las 8 valoraciones posibles para $\gamma$ y $\psi$.

# 1.3.

# Fases en el desarrollo de programas

A la hora de desarrollar un sistema informático que resuelva un problema, las etapas o fases básicas son las siguientes:

- **Análisis** del problema: quien se encargue de realizar el análisis, estudiará el problema para establecer las características de los datos de entrada y las características del resultado que se desea obtener. Al finalizar esta fase, se tendrá la **especificación** del problema. Dicha especificación expresará, de manera concisa, **qué** se desea hacer.
- **Diseño** de la solución: quien se encargue de realizar el diseño, estudiará la especificación del problema —preparada en la fase de análisis— y elaborará una estrategia para resolver el problema. Dicha estrategia ha de ser formalizada mediante un **algoritmo**. Mediante el algoritmo se ha de precisar **cómo** resolver el problema que se ha expresado mediante la especificación.
- **Implementación** del algoritmo: quien se encargue de implementar el algoritmo, trasladará a un **programa ejecutable**, la estrategia recogida en el algoritmo elaborado en la fase de diseño. Por tanto, se ha de trasladar a un lenguaje de programación concreto, el **cómo** expresado mediante el algoritmo.
- **Mantenimiento**: siempre que surja el requerimiento de modificar alguna funcionalidad del sistema o de añadir alguna nueva funcionalidad, se volverá a alguna de las tres fases anteriores, dependiendo del cambio a realizar.

En la figura 1.3.1 de la página 21, se muestra el esquema de las fases del desarrollo de programas.

## 1.3.1 Análisis y especificación

La especificación ha de ser una descripción precisa de los datos de entrada de un programa y del resultado que se ha de obtener. La especificación ha de indicar **qué** se desea calcular pero no ha de indicar **cómo** se ha de realizar el cálculo.

#### 1.3.1.1 Especificación funcional

Habitualmente, al formular la especificación, el resultado se podrá describir en función de los datos de entrada con los que se cuenta al principio. Es decir, el resultado puede ser expresado utilizando los datos de entrada con los que se cuenta desde el principio. Por ejemplo, si se tiene el problema de calcular el factorial de un número natural, la especificación podría ser la que se muestra en la tabla 1.3.1 de la página 21. Ahí se puede observar que el resultado $r$ depende únicamente del dato de entrada $x$ que se tiene de partida. Una vez conocido el valor de $x$, se puede calcular el resultado sin recibir posteriormente (durante el proceso de cálculo) más datos desde el exterior (desde el usuario o alguna otra fuente de información o alguna otra fuente de datos).

Cuando una especificación cumple ese criterio, se dice que se tiene una **especificación funcional**, porque el resultado se puede expresar en función de los datos de entrada.

En la figura 1.3.2 de la página 22, se muestra el esquema general de un problema que admite especificación funcional.

#### 1.3.1.2 Especificación no funcional

En algunos problemas, al formular la especificación, la descripción del resultado no se puede expresar utilizando solo los datos de entrada que se tienen al principio. Por ejemplo, consideremos el problema de "aprender" o "adivinar", a base de hacer preguntas al usuario, una fórmula proposicional $g$ que sea monótona (sin negaciones) y esté escrita en forma normal disyuntiva (FND). Se supone que el usuario conoce la fórmula $g$ o al menos tiene información adicional sobre $g$. Si intentamos dar la especificación del problema, el dato de entrada será un número entero y positivo $n$ que indique cuántas variables proposicionales pueden aparecer ($x_1$, $x_2$, ..., $x_n$) y el resultado será una fórmula $h$, en formato FND, construida a base de las $n$ variables permitidas. Esa fórmula $h$ ha de ser equivalente (o igual) a la fórmula $g$ que tiene el usuario en mente. El sistema a diseñar, irá recibiendo información adicional sobre el objetivo $g$ durante el proceso de cálculo de $h$. Puesto que ni $g$ ni el resto de la información adicional sobre $g$ forman parte de los datos iniciales de entrada, el resultado no puede ser descrito en función de los datos iniciales de entrada. El único dato de entrada inicial es $n$ pero luego se requiere que el usuario vaya introduciendo más datos acerca de $g$ durante el proceso de cálculo. La especificación correspondiente al problema de adivinar —a base de hacer preguntas al usuario— la fórmula que el usuario tiene en mente, se puede formular tal como se muestra en la tabla 1.3.2 de la página 25. Ahí se puede observar que el resultado $h$ no depende únicamente del dato de entrada $n$ que se tiene al principio. No se puede calcular el resultado utilizando solo $n$, sin recibir posteriormente (durante el proceso de cálculo) más datos desde el exterior (desde el usuario o alguna otra fuente de información o alguna otra fuente de datos).

Cuando una especificación cumple ese criterio, se dice que se tiene una **especificación no funcional**, porque el resultado no se puede expresar en función de los datos de entrada iniciales. Hay información adicional que llega más tarde, durante el proceso de cálculo. Esa información

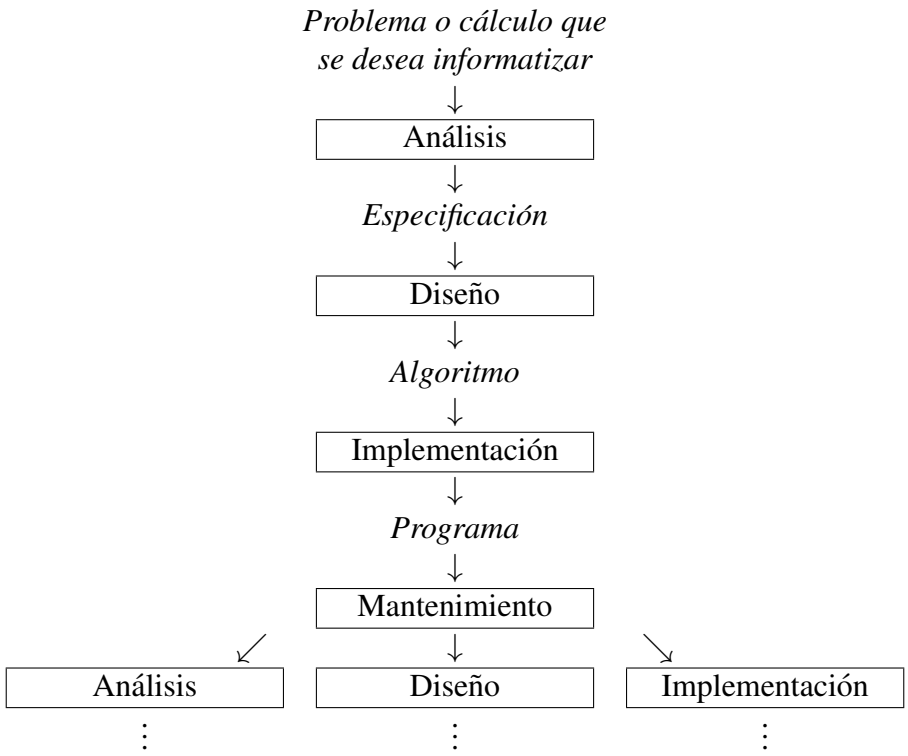


**Figura 1.3.1.** Fases del desarrollo de programas.

| Especificación funcional |
|---|
| Descripción del dato de entrada $x$: |
| $x \in \mathbb{N}$ |
| Descripción del resultado $r$: |
| $r = \prod_{k=1}^{x} k$ |

**Tabla 1.3.1.** Especificación funcional del problema consistente en calcular en la variable $r$ el factorial de un número natural $x$.

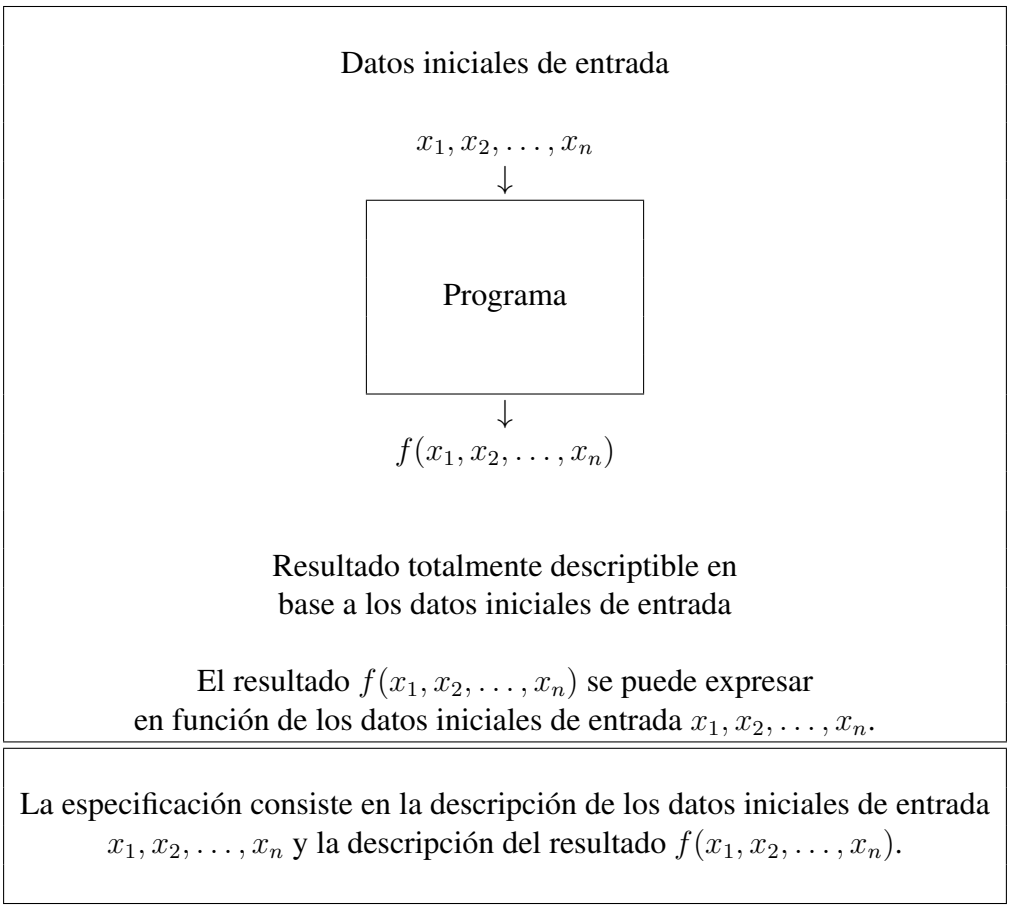


**Figura 1.3.2.** Esquema de especificación funcional.

adicional no está disponible al principio del programa.

Los sistemas que no admiten especificación funcional, deben llevar a cabo una "búsqueda" de información y deben "obtener conclusiones". Por ello, reciben el nombre de **sistemas inteligentes**. El adjetivo "inteligente" ha de ser entendido como "vivo", "espabilado" o "hábil". En inglés sería "smart": **smart systems**. La búsqueda de información y la elaboración de conclusiones pueden ser tareas más o menos complicadas dependiendo del sistema y del cometido que ha de realizar el sistema. La información adicional requerida por el sistema, pueda provenir del usuario o de alguna otra fuente (base de datos, etc.). De ahí que, en general, se diga que estos sistemas inteligentes obtienen información adicional planteando preguntas a un **oráculo**. Por tanto, cuando se habla de contar con un oráculo, ese oráculo puede ser el usuario o algún otro sistema informático que pueda proveer de datos al sistema inteligente en cuestión. También puede ocurrir que haya un usuario y un oráculo y que estos sean diferentes. Generalizando la situación, puede haber varios oráculos.

Un ejemplo comercial de sistema inteligente sería una aplicación de venta de artículos por internet. El sistema nos pide que nos identifiquemos. Luego nos presenta productos teniendo en cuenta nuestras compras anteriores tras consultar dicha información en una base de datos (la base de datos puede ser entendida como un oráculo). Nos pregunta qué tipo de artículo buscamos. Nos muestra una variedad de artículos de ese tipo, tras buscarlos en una base de datos (esa base de datos puede ser entendida como otro oráculo). Si realizamos una búsqueda, o una compra, nos propone artículos que nos pueden interesar por estar relacionados (otra vez contacta con un oráculo que le indica qué productos mostrar). A la hora de pagar, nos pide un número de tarjeta y antes de aceptar la venta, consulta con otro oráculo —el banco asociado a la tarjeta— para comprobar que el número existe y que tenemos crédito. Por tanto, el banco es un oráculo con el que contacta el sistema. Si consideramos que el dato de entrada es nuestra identidad y que el resultado es la factura final, esa factura final no depende solo de nuestra identidad, depende también de los artículos comprados, de las decisiones que hemos tomado (puede que hayamos decidido comprar algo que al principio no pensábamos comprar), de posibles rebajas por rebasar cierta cantidad, de la respuesta del banco (puede que un banco rechace la compra porque ya hemos superado el crédito mensual y que tengamos que dar otro número de tarjeta de otro banco).

En la figura 1.3.3 de la página 26, se muestra el esquema general de un problema que no admite especificación funcional.

El esquema correspondiente a una especificación funcional es un caso particular del esquema correspondiente a la especificación no funcional: Un sistema que admite una especificación funcional realiza cero preguntas al oráculo.

En el caso de los sistemas que requieren especificación no funcional, estudiar la computabilidad no tiene sentido, puesto que esos sistemas tienen acceso a un oráculo de capacidad desconocida que podría resolver los cálculos que el sistema por sí mismo no puede.

Debido a ello, el estudio de la computabilidad se centra en los sistemas que admiten especificación funcional. Por tanto, a partir del tema 3, nos centraremos en problemas que admiten especificación funcional y estudiaremos aspectos relacionados con la computabilidad y la complejidad computacional. Pero ha de quedar claro que los sistemas que requieren especificación no funcional son muy importantes en informática (aplicaciones de ventas, etc.). Consecuentemente, en el tema 2, estudiaremos algunos sistemas que requieren una especificación no funcional, es decir, se presentarán algunos sistemas inteligentes.

Para finalizar este apartado, es importante recalcar que **sistema inteligente e inteligencia artificial son cosas distintas**. En inglés no hay confusión porque se les llama **Smart Systems** y **Artificial Intelligence**. En un sistema informático, algunos componentes pueden tener características de los sistemas inteligentes y otros componentes pueden estar basados en técnicas de inteligencia artificial.

Por ejemplo, consideremos un programa que juega al ajedrez contra una persona. El único dato de entrada que se le dará al programa será el color con el que va a jugar. Es decir, se le indica quién empieza a jugar. Supongamos que el resultado de ese programa es la tabla de ajedrez que queda al terminar el juego. El juego puede terminar por varias razones: la tabla indica que el programa o la persona ha ganado; se detecta una situación de empate (tablas); el usuario abandona el juego. No es posible describir el resultado (la tabla que se obtendrá al final) solo con el dato inicial dado al programa: quién empieza. El juego se puede ver como que el programa pide a la persona que le indique un movimiento. Ahí, la persona hace de oráculo, puesto que provee un dato al programa. El programa es, por tanto, un sistema inteligente (smart system). Por otra parte, una vez obtenido el movimiento de la persona, el programa utilizará técnicas de inteligencia artificial (artificial intelligence) para decidir qué movimiento realizar. El proceso de pedir un movimiento a la persona y calcular el movimiento que realizará el programa, se repetirá. En conclusión, ese programa tiene características de sistema inteligente (obtiene datos externos durante el proceso de ejecución que no estaban disponibles al principio) y, además, utiliza técnicas de inteligencia artificial para realizar algunos cálculos.

### 1.3.2 Diseño y algoritmo

El algoritmo ha de ser una descripción precisa de la estrategia ideada para obtener el resultado a partir de los datos de entrada. Por ejemplo, si se tiene el problema de calcular el factorial de un número natural, el algoritmo podría ser el que se muestra en la tabla 1.3.3 de la página 25. Ese algoritmo expresa que para calcular el factorial de $x$ en la variable $r$, se utilizará una variable auxiliar $i$, que irá recorriendo los números naturales comprendidos entre 1 y $x+1$. En cada momento se tendrá en $r$ el producto de los números naturales comprendidos entre 1 e $i-1$.

Puesto que habitualmente es posible idear distintas estrategias para resolver un problema, también es posible tener distintos algoritmos. En la tabla 1.3.4 de la página 28, se muestra un

| Especificación no funcional |
|---|
| Descripción del dato de entrada $n$: |
| $n \in \mathbb{N} \wedge n \geq 1$<br>donde $n$ indica que se pueden utilizar las variables proposicionales $x_1$, $x_2$, ..., $x_n$. |
| Descripción del resultado $h$: |
| $h$ es una fórmula FND monótona y $h \leftrightarrow g$ (es decir, $h$ ha de ser equivalente a $g$)<br>donde $g$ es la fórmula FND monótona que el usuario tiene en mente. |

**Tabla 1.3.2.** Especificación no funcional del problema consistente en aprender o adivinar una fórmula monótona $g$, en formato FND, que el usuario tiene en mente.

| Algoritmo 1 para el cálculo del factorial de $x$: |
|---|
| $x \in \mathbb{N} \wedge i \in \mathbb{N} \wedge (1 \leq i \leq x+1) \wedge r = \prod_{k=1}^{i-1} k$ |

**Tabla 1.3.3.** Un algoritmo para el cálculo del factorial de $x$, obtenido a partir de la especificación de la tabla 1.3.1 de la página 21.

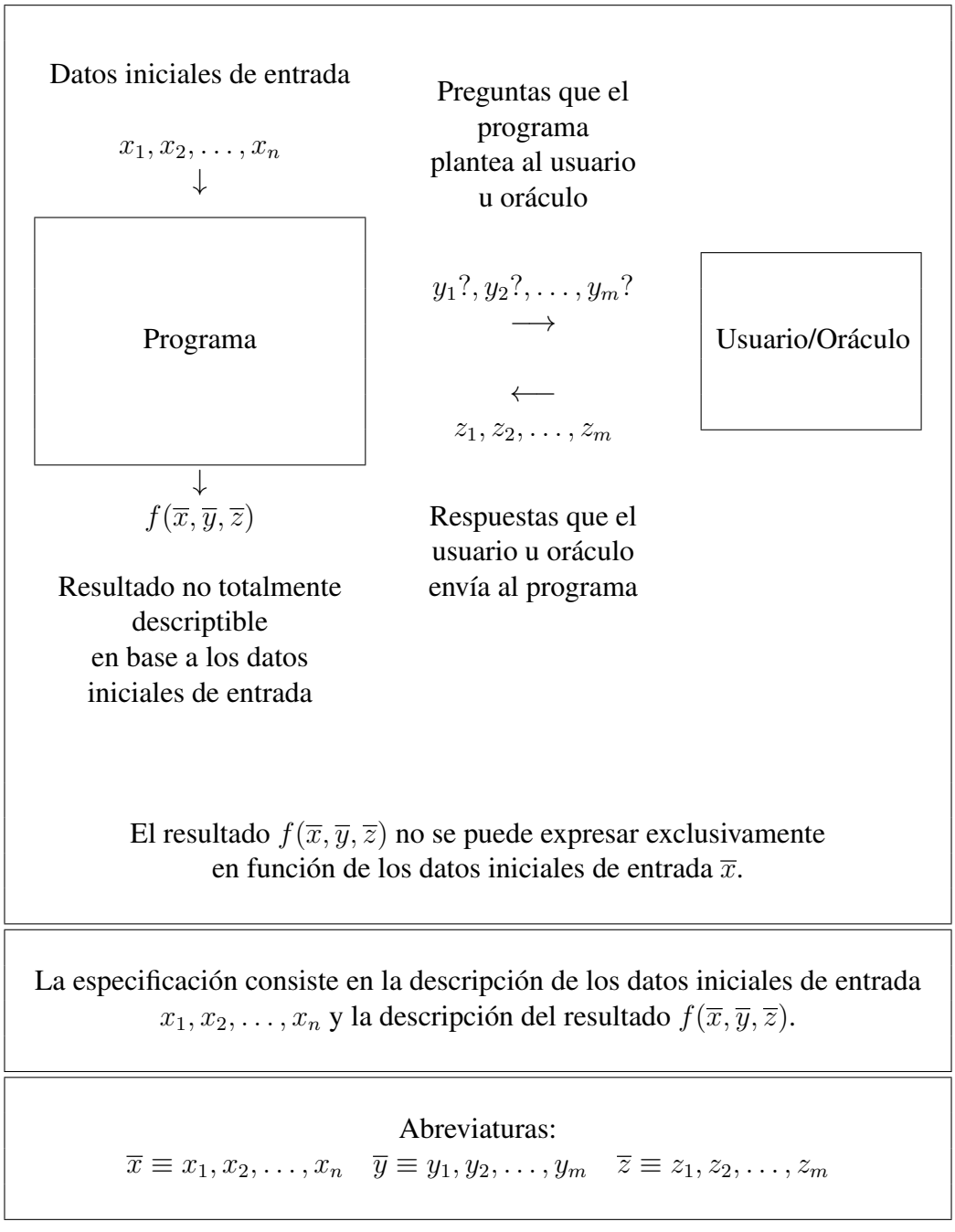


**Figura 1.3.3.** Esquema de especificación no funcional.

segundo algoritmo para el cálculo del factorial. Ese segundo algoritmo expresa que para calcular el factorial de $x$ en la variable $r$, se utilizará una variable auxiliar $i$, que irá recorriendo los números naturales comprendidos entre 0 y $x$. En cada momento se tendrá en $r$ el producto de los números naturales comprendidos entre 1 e $i$.

### 1.3.3 Implementación y programa

Una vez que se tiene un algoritmo, se ha de implementar un programa escrito en un lenguaje de programación concreto. En general, un algoritmo puede dar lugar a distintas implementaciones. Por ejemplo, si se tiene el problema de calcular el factorial de un número natural, el programa correspondiente al algoritmo que se muestra en la tabla 1.3.3 de la página 25 podría ser el que se muestra en la tabla 1.3.5 de la página 28. Si tenemos en cuenta la especificación y el algoritmo, podemos incluso eliminar los casos de error. En ese caso, la especificación y el algoritmo actuarán de precondiciones que se han de cumplir para que el programa calcule el factorial de manera correcta. En la tabla 1.3.6 de la página 29, se muestra el programa resultante. La especificación del dato de entrada y el algoritmo actúan de precondiciones. Nótese, además, que cuando se realiza la llamada *fact1_aux x 1 1* desde *fact1*, el segundo y tercer parámetros (1 y 1) cumplen la precondición establecida en la definición de *fact1_aux* (sustituyendo $i$ por 1 y $r$ por 1). De la misma forma, cuando se realiza la llamada *fact1_aux x (i+1) (r*i)* desde *fact1_aux*, el segundo y tercer parámetros (i+1 y r*i) cumplen la precondición establecida en la definición de *fact1_aux* (sustituyendo $i$ por $i + 1$ y $r$ por $r * i$).

El programa correspondiente al algoritmo que se muestra en la tabla 1.3.4 de la página 28 podría ser el que se muestra en la tabla 1.3.7 de la página 29. En la tabla 1.3.8 de la página 31, se muestra el mismo programa sin casos de error y con precondiciones que hacen posible omitir los casos de error.

### 1.3.4 Mantenimiento de un programa

El mantenimiento puede ser provocado por diversos motivos. Dependiendo del motivo y de los cambios a realizar, se volverá a la fase de implementación, a la fase de diseño o a la fase de análisis.

Supongamos que se ha desarrollado el programa mostrado en la tabla 1.3.5 de la página 28 a partir del algoritmo mostrado en la tabla 1.3.3 de la página 25, el cual ha sido diseñado a partir de la especificación mostrada en la tabla 1.3.1 de la página 21.

A continuación se van a plantear tres escenarios posibles:

1. Una vez que el cliente ejecuta el programa mostrado en la tabla 1.3.5 de la página 28, puede que no le guste el mensaje “Fuera de rango” y prefiera el mensaje “Fuera de intervalo”. Esa modificación no afecta ni a la fase de análisis (especificación) ni a la fase

| Algoritmo 2 para el cálculo del factorial de $x$: |
|---|
| $x \in \mathbb{N} \wedge i \in \mathbb{N} \wedge (0 \leq i \leq x) \wedge r = \prod_{k=1}^{i} k$ |

**Tabla 1.3.4.** Segundo algoritmo para el cálculo del factorial de $x$, obtenido a partir de la especificación de la tabla 1.3.1 de la página 21.

Programa en Haskell. Primera versión para el algoritmo de la tabla 1.3.3 de la página 25:

```
fact1 :: Integer -> Integer
fact1 x = fact1_aux x 1 1

fact1_aux :: Integer -> Integer -> Integer -> Integer
fact1_aux x i r
   | x <= (-1)          = error "Negativo"
   | i <= 0 || i> x+1   = error "Fuera de rango"
   | i == x+1           = r
   | otherwise          = fact1_aux x (i+1) (r*i)
```

**Tabla 1.3.5.** Programa, en Haskell, para el cálculo del factorial a partir de la especificación de la tabla 1.3.1 de la página 21 y del algoritmo de la tabla 1.3.3 de la página 25 (primera versión).

Programa en Haskell. Segunda versión para el algoritmo de la tabla 1.3.3 de la página 25:
Se da por hecho que se cumple lo indicado por la especificación
y el algoritmo. Por tanto, se han eliminado los casos de error.

```
fact1 :: Integer -> Integer
- - Precondición: x ∈ ℕ
fact1 x = fact1_aux x 1 1

fact1_aux :: Integer -> Integer -> Integer -> Integer
- - Precondición:  x ∈ ℕ ∧ i ∈ ℕ ∧ (1 ≤ i ≤ x + 1) ∧ r = ∏_{k=1}^{i-1} k
fact1_aux x i r
   | i == x+1     = r
   | otherwise    = fact1_aux x (i+1) (r*i)
```

**Tabla 1.3.6.** Programa, en Haskell, para el cálculo del factorial a partir de la especificación de la tabla 1.3.1 de la página 21 y del algoritmo de la tabla 1.3.3 de la página 25 (segunda versión).

Programa en Haskell. Primera versión para el algoritmo de la tabla 1.3.4 de la página 28:

```
fact2 :: Integer -> Integer
fact2 x = fact2_aux x 0 1

fact2_aux :: Integer -> Integer -> Integer -> Integer
fact2_aux x i r
   | x <= (-1)            = error "Negativo"
   | i <= (-1) || i> x    = error "Fuera de rango"
   | i == x               = r
   | otherwise            = fact2_aux x (i+1) (r*i)
```

**Tabla 1.3.7.** Programa para el cálculo del factorial a partir de la especificación de la tabla 1.3.1 de la página 21 y del algoritmo de la tabla 1.3.4 de la página 28 (primera versión).

de diseño (algoritmo). Afecta solo a la fase de implementación (programa). Por tanto, el programa vuelve a la persona encargada de la implementación y esta persona ha de modificar la implementación, es decir, el programa, para que se muestre el mensaje deseado por el cliente. Cambiará el programa pero la especificación y el algoritmo no cambian.

2. Una vez que el cliente tiene el programa mostrado en la tabla 1.3.5 de la página 28, puede que haya querido ver el código escrito en Haskell y no le haya gustado que el parámetro $i$ se mueva entre 1 y $x + 1$. Prefiere que $i$ siempre esté entre 1 y $x$. Esa modificación no afecta a la fase de análisis (especificación) pero sí a la fase de diseño (algoritmo) y luego a la fase de implementación (programa). Por tanto, el programa vuelve a la persona encargada del diseño y esta persona ha de modificar el algoritmo para que $i$ siempre esté entre 1 y $x$. Por ejemplo, una opción sería el algoritmo de la tabla 1.3.9 de la página 31 y, a continuación, se implementaría ese nuevo algoritmo y se obtendría el programa de la tabla 1.3.10 de la página 32. Por tanto, cambiarán el algoritmo y el programa pero la especificación no cambia.

   Nótese que el cambio de algoritmo ha conllevado la necesidad de hacer ajustes en la función *fact3*. Para detectar esa necesidad, hacen falta tanto la especificación como el algoritmo. El asunto surge del hecho de que en la especificación se admite que $x$ sea 0 pero en el algoritmo no, porque ha de cumplirse $1 \leq i \leq x$, es decir, $x$ ha de ser al menos 1. Debido a ello, el caso de $x$ igual a 0 se ha de tratar en la función *fact3* y no en *fact3_aux*.

3. Una vez que el cliente tiene el programa, puede que, además del factorial, quiera también la suma de los números del intervalo $[1..x]$. Esa modificación afecta a la fase de análisis (especificación) y, por consiguiente, también a la fase de diseño (algoritmo) y luego a la fase de implementación (programa). Por tanto, el programa vuelve a la persona encargada de realizar el análisis, que ha de modificar la especificación para que se calcule también la suma de los números del intervalo $[1..x]$. Por ejemplo, se obtendría la especificación de la tabla 1.3.11 de la página 32 y, a continuación, la persona encargada de realizar el diseño, produciría el algoritmo de la tabla 1.3.12 de la página 32. Finalmente, la persona encargada de realizar la implementación implementaría ese nuevo algoritmo obteniendo el programa de la tabla 1.3.13 de la página 33. Por tanto, cambiarán la especificación, el algoritmo y el programa.

Programa en Haskell. Segunda versión para el algoritmo de la tabla 1.3.4 de la página 28:

Se da por hecho que se cumple lo indicado por la especificación
y el algoritmo. Por tanto, se han eliminado los casos de error.

```
fact2 :: Integer -> Integer
-- Precondición: x ∈ ℕ
fact2 x = fact2_aux x 0 1

fact2_aux :: Integer -> Integer -> Integer -> Integer
-- Precondición:  x ∈ ℕ ∧ i ∈ ℕ ∧ (0 ≤ i ≤ x) ∧ r = ∏_{k=1}^{i} k
fact2_aux x i r
  | i == x       = r
  | otherwise    = fact2_aux x (i+1) (r*i)
```

**Tabla 1.3.8.** Programa para el cálculo del factorial a partir de la especificación de la tabla 1.3.1 de la página 21 y del algoritmo de la tabla 1.3.4 de la página 28 (segunda versión).

Algoritmo 3 para el cálculo del factorial de $x$:

$$x \in \mathbb{N} \land i \in \mathbb{N} \land (1 \leq i \leq x) \land r = \prod_{k=1}^{i-1} k$$

**Tabla 1.3.9.** Algoritmo para el cálculo del factorial tras modificar el algoritmo de la tabla 1.3.3 de la página 25.

| Programa: |
|---|
| fact3 :: Integer -> Integer<br>\| x == 0 = 1<br>\| otherwise = fact3_aux x 1 1<br><br>fact3_aux :: Integer -> Integer -> Integer -> Integer<br>fact3_aux x i r<br>\| x <= (-1) = error "Negativo"<br>\| i <= 0 \|\| i> x = error "Fuera de rango"<br>\| i == x = r * x<br>\| otherwise = fact3_aux x (i+1) (r*i) |

**Tabla 1.3.10.** Programa para el cálculo del factorial a partir de la especificación de la tabla 1.3.1 de la página 21 y del algoritmo de la tabla 1.3.9 de la página 31.

| Descripción del dato de entrada $x$: |
|---|
| $x \in \mathbb{N}$ |
| **Descripción de los resultados $r_1$ y $r_2$:** |
| $(r_1 = \prod_{k=1}^{x} k) \wedge (r_2 = \sum_{k=1}^{x} k)$ |

**Tabla 1.3.11.** Especificación del problema consistente en calcular el factorial de $x$ y la suma de los elementos del intervalo $[1..x]$.

| Algoritmo: |
|---|
| $i \in \mathbb{N} \wedge (1 \leq i \leq x+1) \wedge (r_1 = \prod_{k=1}^{i-1} k) \wedge (r_2 = \sum_{k=1}^{i-1} k)$ |

**Tabla 1.3.12.** Algoritmo para el problema consistente en calcular el factorial de $x$ y la suma de los elementos del intervalo $[1..x]$ (especificado en la tabla 1.3.11 de la página 32).

Programa en Haskell para el algoritmo de la tabla 1.3.12 de la página 32:

```
fact_sum :: Integer -> (Integer,Integer)
fact_sum x = fact_sum_aux x 1 1 0

fact_sum_aux :: Integer -> Integer -> Integer -> Integer -> (Ineger,Integer)
fact_sum_aux x i r1 r2
   | x <= (-1)            = error "Negativo"
   | i <= 0 || i> x+1     = error "Fuera de rango"
   | i == x+1             = (r1,r2)
   | otherwise            = fact_sum_aux x (i+1) (r1* i) (r2 + i)
```

**Tabla 1.3.13.** Programa para el problema consistente en calcular el factorial de $x$ y la suma de los elementos del intervalo $[1..x]$ (obtenido a partir de la especificación de la tabla 1.3.11 y del algoritmo de la tabla 1.3.12 de la página 32).

# 1.4.

# Computabilidad

En este apartado, se considerarán solo cálculos que admiten especificación funcional.

## 1.4.1 Clasificación de problemas

Los problemas que son candidatos a ser informatizados pueden ser clasificados de distintas maneras. Una de esas clasificaciones es la siguiente:

### 1.4.1.1 Problemas de decisión

Un problema de decisión consiste en, dado un dato, decidir si ese dato cumple una determinada propiedad. La respuesta ha de ser "Sí" o "No".

Ejemplo: Dada una fórmula de la lógica proposicional, averiguar si existe alguna valoración que haga que la fórmula sea $True$. Para la fórmula $\gamma$ de la figura 1.2.4 de la página 17, sí existe al menos una valoración que la hace $True$. En cambio, para la fórmula $\psi$ de la figura 1.2.4 de la página 17, no existe ninguna valoración que la haga $True$.

Ejemplo: Dada una fórmula de la lógica proposicional, averiguar si todas las valoraciones hacen que la fórmula sea $True$. Para las fórmulas $\gamma$ y $\psi$ de la figura 1.2.4 de la página 17, la respuesta sería negativa. En cambio, para la fórmula $(x_1 \vee \neg x_1) \wedge (x_2 \vee \neg x_2) \wedge (x_3 \vee \neg x_3)$ la respuesta es afirmativa.

Ejemplo: Dada una fórmula de la lógica proposicional y una valoración, averiguar si la valoración hace que la fórmula sea $True$. Para la fórmula $\gamma$ de la figura 1.2.4 de la página 17 y la valoración $(F, T, T)$, la respuesta sería negativa.

Ejemplo: Dados dos números enteros no negativos $x$ e $y$, averiguar si $y$ es el factorial de $x$. En el caso concreto de $x = 4$ e $y = 24$, la respuesta sería afirmativa pero en el caso $x = 4$ e

$y = 15$ la respuesta sería negativa.

Ejemplo: Dado un número entero positivo $x$, averiguar si $x$ tiene algún divisor en el intervalo $[2..x-1]$. En el caso concreto de $x = 12$, la respuesta sería afirmativa pero en el caso de $x = 17$ la respuesta sería negativa.

#### 1.4.1.2 Problemas de búsqueda

Un problema de búsqueda consiste en intentar encontrar uno o más elementos que cumplen una determinada propiedad. La respuesta ha de ser el o los elementos encontrados.

Ejemplo: Dada una fórmula de la lógica proposicional, buscar una valoración que haga que la fórmula sea $True$. Para la fórmula $\gamma$ de la figura 1.2.4 de la página 17, una valoración que la haga $True$ sería $(T, T, F)$. En cambio, para la fórmula $\psi$ de la figura 1.2.4 de la página 17, no existe ninguna valoración que la haga $True$. Por tanto, hay procesos de búsqueda en los que no se encontrará ningún elemento que cumpla la propiedad deseada.

Ejemplo: Dada una fórmula de la lógica proposicional, buscar todas las valoraciones que hacen que la fórmula sea $True$. Para la fórmulas $\gamma$ de la figura 1.2.4 de la página 17, la respuesta sería el conjunto formado por las siguientes valoraciones:

$$\{(F, T, F), (T, F, T), (T, T, F), (T, T, T)\}$$

En cambio, para la fórmula $\psi$ de la figura 1.2.4 de la página 17, la respuesta sería el conjunto vacío.

Ejemplo: Dado un número entero no negativo $x$, calcular el factorial de $x$. En el caso concreto de $x = 4$, la respuesta sería 24. Puesto que para cada número entero no negativo el factorial es único, la búsqueda consistirá en calcular ese valor único.

Ejemplo: Dado un número entero positivo $x$, devolver un divisor de $x$ que pertenezca al intervalo $[2..x-1]$. En el caso concreto de $x = 12$, una respuesta posible sería 4, pero en el caso de $x = 17$ no existe ningún divisor. De nuevo, observamos que hay procesos de búsqueda en los que no se encontrará ningún elemento que cumpla la propiedad deseada.

#### 1.4.1.3 Problemas de optimización

Un problema de optimización consiste en intentar encontrar un elemento que, bajo algún criterio preestablecido, sea el resultado óptimo de entre los que cumplen una determinada propiedad. La respuesta ha de ser el o los elementos óptimos encontrados.

Ejemplo: Dada una fórmula de la lógica proposicional, buscar una valoración que contenga el menor número posible de apariciones del valor $True$ y que haga que la fórmula sea $True$. Para la fórmula $\gamma$ de la figura 1.2.4 de la página 17, la valoración que contiene el menor número posible de apariciones del valor $True$ y que hace que la fórmula sea $True$ es $(F, T, F)$. En cambio, para la fórmula $\psi$ de la figura 1.2.4 de la página 17, no existe ninguna valoración que la haga $True$ y, consecuentemente, no existe ninguna valoración óptima.

Ejemplo: Dado un número entero no negativo $x$, calcular el mayor factorial de $x$. Puesto que para cada número entero no negativo el factorial es único, ese único valor será el óptimo. En el caso concreto de $x = 4$, la respuesta sería 24.

Ejemplo: Dado un número entero positivo $x$, devolver el mayor divisor de $x$ que pertenezca al intervalo $[2..x - 1]$. En el caso concreto de $x = 55$, la respuesta sería 11, pero en el caso de $x = 17$ no existe ningún divisor y, por tanto, no hay solución óptima.

### 1.4.2 Computabilidad e incomputabilidad

Dado un problema, un cálculo o una tarea que se desea informatizar y que admite especificación funcional, se dice que el problema o el cálculo o la tarea es **computable** si existe un algoritmo (una estrategia) para llevarlo a cabo y que para cualquier dato de entrada devuelve un resultado.

Por ejemplo, el problema de calcular el factorial de un número natural es computable. El problema de calcular la lista de divisores enteros de un número entero positivo es computable. El problema de calcular el máximo común divisor de dos números enteros positivos es computable. También es computable el problema de decidir si para una fórmula de la lógica proposicional existe alguna valoración que la haga cierta.

Dado un problema, un cálculo o una tarea que se desea informatizar y que admite especificación funcional, si no existe ningún algoritmo que resuelva el problema para todos los datos de entrada posibles, entonces no es computable, es decir, es **incomputable**.

El conjunto de problemas incomputables se puede partir en dos subconjuntos disjuntos y no vacios: el subconjunto de los **problemas semicomputables** y el subconjunto de los **problemas que no son ni semicomputables**.

### 1.4.3 Semicomputabilidad

Dado un problema, un cálculo o una tarea que se desea informatizar y que admite especificación funcional, si existe algún algoritmo capaz de resolver el problema para un subconjunto no trivial bien caracterizado de todos los posibles datos de entrada, entonces el problema es semicomputable. El resto de posibles datos de entrada no recibirá ningúna respuesta porque en

esos casos el proceso de cálculo se sumergirá en un bucle infinito. Pero el usuario, dado un dato, en general no tiene manera alternativa de saber si el dato pertenece al subconjunto de los datos para los cuales sí habrá solución. La única manera de saberlo es ejecutando el algoritmo, lo cual conlleva el riesgo de introducirse en un proceso infinito.

Tal como se ha indicado, en el caso de los problemas semicomputables, se sabe qué propiedad cumplen los datos que recibirán respuesta y qué propiedad cumplen los datos que no recibirán respuesta:

- En los problemas de decisión, se recibirá la respuesta afirmativa correctamente para aquellos datos que han de recibir una respuesta afirmativa. Pero, en el caso de los datos que no han de recibir una respuesta afirmativa, se quedarán sin llegar a recibir una respuesta negativa porque el programa quedará inmerso en un proceso infinito de decisión. Es decir, el usuario queda a la espera de una respuesta que no llegará nunca.
- En los problemas de búsqueda, se encontrará una solución para aquellos datos de entrada para los cuales existe alguna solución. Pero, en el caso de datos de entrada para los cuales no existe ninguna solución, no se conseguirá llegar a la conclusión de que no existe ninguna solución y el programa quedará inmerso en un proceso infinito de búsqueda y no responderá nunca. Por tanto, el usuario queda a la espera de una respuesta que no llegará nunca.
- En los problemas de optimización, se encontrará una solución óptima para aquellos datos de entrada para los cuales existe alguna solución óptima. Pero, en el caso de datos de entrada para los cuales no existe ninguna solución óptima, no se conseguirá llegar a la conclusión de que no existe ninguna solución óptima y el programa quedará inmerso en un proceso infinito de búsqueda de una solución óptima y no responderá nunca. Esto significa que el usuario queda a la espera de una respuesta que no llegará nunca.

Por ejemplo, el problema de decidir si un polinomio general —definido sobre más de una variable— se hace 0 para alguna asignación de valores enteros a sus variables, es incomputable pero semicomputable.

Un ejemplo de polinomio definido sobre tres variables $x$, $y$ y $z$ sería el siguiente:

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

La única manera sistemática de averiguar si existe una combinación de valores enteros para $x$, $y$ y $z$ que hagan que el polinomio valga 0, es ir probando —de una manera ordenada— con todas las combinaciones posibles:

$$(0,0,0), (0,0,1), (0,0,-1), (0,1,0), (0,-1,0), (1,0,0), (-1,0,0), (0,0,2), (0,0,-2), \ldots$$

Si existe una combinación que haga que el polinomio valga 0, esa combinación se encontrará en un número finito de pasos. Pero si no existe, la búsqueda será infinita, no hay manera de

saber que no existe.

La propiedad que cumplen los polinomios que reciben una respuesta es que existe una combinación de valores de las variables que hace que el polinomio valga 0. Por otra parte, la propiedad que cumplen los polinomios que no reciben una respuesta es que no existe una combinación de valores de las variables que haga que el polinomio valga 0. Pero dado un polinomio, en general el usuario no tiene una manera alternativa —sin aplicar el proceso sistemático indicado— para averiguar si existe una combinación que haga que el valor del polinomio sea 0. La única manera es aplicar dicho proceso sistemático, con el riesgo de entrar en un proceso infinito.

Otro ejemplo de problema incomputable pero semicomputable es el del programa $P$ que tras recibir, como datos de entrada, otro programa $Q$ y un dato en entrada $d$ para $Q$, ha de decidir (respondiendo con un "Sí" o con un "No") si $Q$ va a terminar el cálculo correspondiente a $d$. En la figura 1.2.1 de la página 15 se muestra la especificación correspondiente a dicho problema. Se puede diseñar un programa $P$ que es capaz de responder afirmativamente siempre que $Q$ termine el cálculo correspondiente a $d$. Dicho programa $P$, dados $Q$ y $d$, simulará la ejecución de $Q$ con dato $d$. Pero cuando $Q$ no termine para $d$, $P$ no podrá devolver ninguna respuesta porque se quedará esperando infinitamente a que $Q$ termine el cálculo correspondiente a $d$. Aquí también sucede que el usuario no tiene una manera alternativa —sin simular $Q$ con dato de entrada $d$— para averiguar si $Q$ va a terminar el cálculo correspondiente a $d$. La única manera es simular $Q$ con dato de entrada $d$, con el riesgo de entrar en un proceso infinito.

### 1.4.4 Ni siquiera semicomputabilidad

Por último, dado un problema, un cálculo o una tarea que se desea informatizar y que admite especificación funcional, se dice que ni siquiera es semicomputable si no existe ningún algoritmo capaz de resolver el problema ni siquiera para un subconjunto no trivial bien caracterizado de todos los posibles datos de entrada.

Por ejemplo, el problema de, dado un programa $Q$ encontrar un dato $d$ para el cual $Q$ no responda (es decir, cicle infinitamente), no es semicomputable. La idea es ir probando con distintos datos hasta encontrar un dato para el cual $Q$ no termine el cálculo correspondiente. Pero para averiguar si el cálculo de $Q$ para un dato $d$ va a terminar o no va a terminar, hay que ejecutar el programa $Q$ con dato de entrada $d$. Pero una vez puesto en marcha esa ejecución de $Q$ con dato $d$, si el cálculo entra en un proceso infinito, entonces nos quedaremos en espera sin fin. Y, aunque $d$ sea un dato para el cual $Q$ no termine, no podremos saberlo nunca y nunca podremos decir que se ha encontrado un dato $d$ para el cual $Q$ no responde.

### 1.4.5 Computabilidad irrealizable o inalcanzable

Ante un problema que es computable, es decir, existe un algoritmo que lo resuelve, puede ocurrir que no sea posible encontrar el algoritmo. Existe pero no es posible determinar cuál es ese algoritmo. En estos casos se dice que el problema es computable pero la computabilidad es inalcanzable o no realizable o que no se puede efectuar. En los subapartados que vienen a continuación se muestran algunos ejemplos.

#### 1.4.5.1 Ejemplo: Seres inteligentes en la Vía Láctea

Por ejemplo, el problema que devuelve True si existen más de tres planetas distintos habitados por seres inteligentes en la Vía Láctea y devuelve False en caso contrario, es computable. Existe un programa/algoritmo que lo resuelve. Es alguno de los dos programas mostrados en la tabla 1.4.1 de la página 41 y la tabla 1.4.2 de la página 41. No sabemos cuál.

Un problema similar sería el de que, dado un número entero $x$, se decida si hay más de $x$ planetas distintos habitados por seres inteligentes en la Vía Láctea, devolviendo $True$ en caso afirmativo y $False$ en caso contrario. Hay un programa que responde de manera adecuada pero no sabemos cuál es. La especificación se puede formular como en la figura 1.4.1 de la página 41. El tipo de ese programa es $\mathbb{N} \to \{True, False\}$. Por tanto, el dominio es infinito: $\mathbb{N}$. Hay infinitas funciones/programas que tienen ese tipo y una de ellas es la adecuada para nuestro problema, pero no sabemos cuál de ellas es la adecuada. Existe pero no podemos saber cuál es.

#### 1.4.5.2 Ejemplo: Lluvia en Bilbao

Un caso similar, pero aparentemente más manejable, sería el de que, dado un año comprendido entre 1000 y 1999, nos diga cuántos días del mes de noviembre de ese año llovió más de 1 litro por metro cuadrado en Bilbao. Existe un programa que da esa respuesta, pero no podemos saber cuál es. En la figura 1.4.2 de la página 41, se muestra la especificación correspondiente a este problema. Existe un programa que calcula ese resultado pero no sabemos cuál es. Hay $31^{1000}$ programas candidatos, puesto que para cada año, hay 31 respuestas posibles: 0 días, 1 día, 2 días, 3 días, y así hasta 30 días.

Del mismo estilo sería también el siguiente caso: programa que, dados un año $x$ comprendido entre 1000 y 1999 y una cantidad de litros $\ell$, comprendida entre 0 y 49, nos diga si hubo algún día del mes de noviembre de ese año en el que llovió más de $\ell$ litros por metro cuadrado en Bilbao. En la figura 1.4.3 de la página 43, se muestra la especificación correspondiente a esta tarea. Existe un programa que lo resuelve correctamente, pero no sabemos cuál es. Hay $2^{1000\times 50}$ programas candidatos.

| Programa: |
| --- |
| vida1 :: Bool<br>vida1 = True |

**Tabla 1.4.1.** Programa que siempre devuelve True (Ejemplo para el apartado 1.4.5 de la página 40).

| Programa: |
| --- |
| vida2 :: Bool<br>vida2 = False |

**Tabla 1.4.2.** Programa que siempre devuelve False (Ejemplo para el apartado 1.4.5 de la página 40).

Especificación:

$$vida\_mas(x) = \begin{cases} True & \text{si hay más de } x \text{ planetas distintos habitados por seres inteligentes en la Vía Láctea} \\ False & \text{si no hay más de } x \text{ planetas distintos habitados por seres inteligentes en la Vía Láctea} \end{cases}$$

donde $x \in \mathbb{N}$

**Figura 1.4.1.** Especificación del problema que consiste en decidir si hay más de $x$ planetas distintos habitados por seres inteligentes en la Vía Láctea (Ejemplo para el apartado 1.4.5 de la página 40).

Especificación:

$$cuantos\_dias\_mas\_uno(x) = \begin{cases} \text{número de días del mes de noviembre del año } x \text{ en los que llovió más de 1 litro por metro cuadrado en Bilbao} \end{cases}$$

donde $x \in \mathbb{N}$ y $1000 \leq x \leq 1999$

**Figura 1.4.2.** Especificación del problema que consiste en indicar cuántos días del mes de noviembre de ese año llovió más de 1 litro por metro cuadrado en Bilbao (Ejemplo para el apartado 1.4.5 de la página 40).

### 1.4.6 Computabilidad intratable

Un problema **computable** es **intratable** si todos los algoritmos conocidos para resolver ese problema son muy ineficientes (porque requieren demasido tiempo o demasiada memoria para realizar los cálculos correspondientes).

Hay muchos problemas intratables. Un ejemplo concreto es el mencionado en el apartado 1.2 de la página 13:

Es posible diseñar un programa $S$ para decidir si una fórmula de la lógica proposicional es satisfactible o no. Es decir, dada una fórmula $\varphi$ de la lógica proposicional y el número de variables proposicionales $n$ que hay en esa fórmula, $S$ decide (respondiendo con un "Sí" o con un "No") si para alguna valoración de las $n$ variables proposicionales, la fórmula $\varphi$ es cierta. En la figura 1.2.3 de la página 15 se muestra la especificación correspondiente a dicho problema.

El método que se conoce para realizar este cálculo, consiste en ir probando con todas las valoraciones posibles para las $n$ variables proposicionales involucradas, hasta encontrar una valoración que haga cierta a la fórmula proposicional $\varphi$ o hasta agotar todas las posibilidades, en cuyo caso se sabrá que $\varphi$ no es satisfactible. En el peor caso, este método requiere generar $2^n$ valoraciones.

Si consideramos la fórmula proposicional $\gamma$, definida como $(x_1 \vee \neg x_3) \wedge (x_2 \vee x_3)$, donde el número de variables proposicionales es 3, la respuesta de $S(\gamma, 3)$ será "Sí" porque para la valoración $x_1 = False$, $x_2 = True$ y $x_3 = False$, la fórmula $\gamma$ es cierta. Hay también otras valoraciones que hacen cierta a $\gamma$. En cambio, si consideramos la fórmula proposicional $\psi$, definida como $(\neg x_1) \wedge (x_1 \vee x_3) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3)$, donde el número de variables proposicionales es también 3, la respuesta de $S(\psi, 3)$ será "No" porque ninguna valoración de $x_1$, $x_2$ y $x_3$ hace cierta a $\psi$. En la figura 1.2.4 de la página 17 se muestran las ocho valoraciones posibles para $\gamma$ y $\psi$.

Si el programa $S$ sigue el orden mostrado en la figura 1.2.4 de la página 17, entonces en el caso de $\gamma$, tendría que generar las primeras tres valoraciones, puesto que $v_3$ es la primera valoración que hace que $\gamma$ sea cierta. Por su parte, en el caso de $\psi$, tendría que generar las ocho valoraciones, para terminar constatando que ninguna valoración hace que $\psi$ sea cierta. Si en vez de 3 variables proposicionales, se tienen 150, generar las $2^{150}$ valoraciones posibles es muy ineficiente, puesto que requeriría mucho tiempo. Por tanto, se considera que el programa $S$ no es práctico, es decir, no es útil. Al no disponer de un método eficiente para el problema de la satisfactibilidad de fórmulas proposicionales, este problema se clasifica como **intratable**.

Especificación:

$$cuantos\_dias\_mas\_ele(x, \ell) = \begin{cases} True & \text{si hubo algún día del mes de noviembre del año } x \text{ en el que llovió más de } \ell \text{ litros por metro cuadrado en Bilbao} \\ False & \text{si no hubo ningún día del mes de noviembre del año } x \text{ en el que llovió más de } \ell \text{ litros por metro cuadrado en Bilbao} \end{cases}$$

donde $x \in \mathbb{N}$, $1000 \leq x \leq 1999$, $\ell \in \mathbb{N}$ y $0 \leq \ell \leq 49$

**Figura 1.4.3.** Especificación del problema que consiste en indicar si hubo algún día del mes de noviembre del año $x$ en el que llovió más de $\ell$ litros por metro cuadrado en Bilbao (Ejemplo para el apartado 1.4.5 de la página 40).

# 1.5.

# Complejidad computacional

Ante un problema computable o semicomputable, el primer objetivo es diseñar un algoritmo que sirva para realizar el cálculo correspondiente. Pero la eficiencia del algoritmo que se diseñe es muy importante. Si un algoritmo que no es eficiente —porque necesita demasiado tiempo para realizar los cálculos o porque necesita demasiada memoria para realizar los cálculos— no es útil en general.

Al hablar sobre la eficiencia con respecto a los algoritmos que resuelven un problema, hay que diferenciar dos casos:

(1) Puede ocurrir que el algoritmo diseñado por nosotros no sea eficiente pero que exista algún algoritmo más eficiente. En este caso, el problema no es intrínsecamente difícil —no es un problema complejo— pero nuestro algoritmo es malo. Es posible diseñar algoritmos más eficientes.

(2) Puede ocurrir que cualquier algoritmo que sirva para resolver ese problema sea ineficiente. En este caso, el problema es intrínsecamente difícil —es un problema complejo— y no es posible resolverlo mediante un algoritmo sencillo, es decir, mediante un algoritmo eficiente.

Consecuentemente, por una parte, conviene analizar la eficiencia de algoritmos concretos que resuelven un problema. Y, por otra parte, conviene determinar la dificultad intrínseca del problema, es decir, la complejidad computacional del problema, para saber si es posible diseñar algoritmos eficientes.

## 1.5.1 Análisis de algoritmos y complejidad de los problemas

En relación con la eficiencia de los algoritmos, hay que distinguir dos áreas:

(1) **Análisis de los algoritmos**: En este área se determina la eficiencia de algoritmos concretos, pero no se determina la dificultad intrínseca —es decir, la complejidad computacional— de los problemas. Por tanto, tras obtener un algoritmo concreto que sirva para resolver

un problema, se analiza la eficiencia de ese algoritmo concreto. Pueden existir otros algoritmos mejores y otros algoritmos peores para resolver ese problema. En general, el analizar la eficiencia de un algoritmo concreto diseñado para un problema, no sirve para decidir cuál es la dificultad intrínseca —la complejidad computacional— del problema.

(2) **Complejidad computacional**: En este área se determina la dificultad intrínseca de los problemas, es decir, la complejidad computacional de los problemas. Dado un problema concreto, el objetivo es averiguar el nivel de eficiencia del mejor algoritmo posible para resolver ese problema.

## 1.5.2 Complejidad temporal: clases

En las décadas previas, se han identificado diferentes niveles de complejidad al estudiar el tiempo mínimo requerido para resolver problemas computacionales. Los problemas computacionales que tienen el mismo nivel de complejidad conforman una clase de complejidad computacional. A continuación se presenta una clasificación básica con el objetivo de dar una perspectiva general de esa clasificación de la complejidad computacional.

### 1.5.2.1 Clase de complejidad P

Un problema de decisión pertenece a la clase de complejidad P, si se ha conseguido probar que existe un algoritmo eficiente para resolver ese problema.

Se considera que un algoritmo es eficiente, si realiza los cálculos en tiempo polinómico. Esto significa que si el tamaño de un dato de entrada es $n$, el tiempo necesario para realizar el cálculo es descriptible mediante un polinomio que depende de $n$. Lo cual se representa como $poli(n)$.

Muchos de los problemas de decisión más conocidos y habituales pertenecen a la clase P: decidir si un vector está ordenado, decidir si un número entero positivo es primo, decidir si un número $x$ es el máximo común divisor de dos números enteros positivos $v$ y $w$, decidir si $x$ es el valor máximo de un vector numérico, etc.

También pertenecen a P problemas de decisión como el de determinar si existe alguna valoración que haga que una conjunción de cláusulas de Horn proposicionales sea cierta. Las cláusulas de Horn son cláusulas de la forma $(\neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_n \vee z)$ o de la forma $(\neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_n)$ donde $n \geq 0$. Este problema se conoce como HORN.

### 1.5.2.2 Clase de complejidad NP

Un problema de decisión pertenece a la clase de complejidad NP si en el caso de los datos de entrada que reciben la respuesta “Sí”, existe una prueba que en tiempo polinómico demuestra

que la respuesta es "Sí".

Todos los problemas pertenecientes a P pertenecen también a NP. Pero no se sabe si P y NP son iguales o no.

Por ejemplo, dada una fórmula $\delta$ de la lógica proposicional, para decidir si $\delta$ es satisfactible, es decir, para decidir si existe alguna valoración que la haga cierta, es posible que haya que generar todas las valoraciones posibles. Si la fórmula contiene $n$ variables diferentes, habrá que generar $2^n$ valoraciones. Eso requiere tiempo exponencial descriptible mediante una expresión de la forma $2^{poli(n)}$. En cambio, si nos dan una valoración $v$ que haga cierta a $\delta$, comprobar que $v$ hace que realmente $\delta$ sea cierta requiere solo tiempo polinómico. Este problema de las fórmulas proposicionales se conoce como SAT. No se ha conseguido determinar si existe algún algoritmo polinómico para SAT. Pero tampoco se ha conseguido determinar que no existe ningún algoritmo polinómico para SAT. El problema SAT es una generalización del problema HORN del apartado anterior. Pero mientras HORN pertenece a P, no se sabe si SAT pertenece a P. Ambos pertenecen a NP.

### 1.5.2.3 Clase de complejidad co-NP

Un problema de decisión pertenece a la clase de complejidad co-NP si el problema de decisión complementario pertenece a la clase de complejidad NP.

Por ejemplo, el problema SAT, consistente en decidir si para una fórmula $\delta$ de la lógica proposicional existe alguna valoración que la haga cierta, pertenece a la clase NP y para dar una respuesta afirmativa es suficiente con encontrar una valoración que haga que $\delta$ sea cierta. Además, para comprobar que una valoración concreta $v$ hace cierta a $\delta$ se requiere solo tiempo polinómico. El complementario de SAT recibe el nombre de NONSAT y consiste en decidir si para una fórmula $\delta$ de la lógica proposicional todas las valoraciones la hacen falsa. En el caso de NONSAT, para dar una respuesta negativa es suficiente con encontrar una valoración que haga que $\delta$ sea cierta. Además, para comprobar que una valoración concreta $v$ hace cierta a $\delta$ se requiere solo tiempo polinómico.

SAT pertenece a NP mientras que NONSAT pertenece a co-NP.

Las clases TAUT y NONTAUT constituyen otro ejemplo similar. TAUT consiste en decidir si para una fórmula $\delta$ de la lógica proposicional todas las valoraciones la hacen cierta. En el caso de TAUT, para dar una respuesta negativa es suficiente con encontrar una valoración que haga que $\delta$ sea falsa. Además, para comprobar que una valoración concreta $v$ hace falsa a $\delta$ se requiere solo tiempo polinómico. El complementario de TAUT recibe el nombre de NONTAUT y consiste en decidir si para una fórmula $\delta$ de la lógica proposicional, alguna valoración la hace falsa. En el caso de NONTAUT, para dar una respuesta afirmativa es suficiente con encontrar una valoración que haga que $\delta$ sea falsa. Además, para comprobar que una valoración concreta

$v$ hace falsa a $\delta$ se requiere solo tiempo polinómico.

TAUT pertenece a co-NP y NONTAUT pertenece a NP.

#### 1.5.2.4 Clase de complejidad EXPTIME

Un problema de decisión pertenece a la clase de complejidad EXPTIME, si se ha conseguido probar que existe un algoritmo exponencial para resolver ese problema. Con un algoritmo exponencial, dado un dato de entrada de tamaño $n$, el tiempo necesario para obtener el resultado estará descrito mediante una espresión exponencial de la forma $2^{poli(n)}$.

Por ejemplo, en el juego de ajedrez, dada una situación concreta, el tiempo para decidir cuál es el mejor movimiento es exponencial. Debido a ello, los programas que juegan al ajedrez no pueden calcular el mejor movimiento en cada situación porque se requiere demasiado tiempo.

#### 1.5.2.5 Clase de complejidad 2-EXPTIME

Un problema de decisión pertenece a la clase de complejidad 2-EXPTIME, si se ha conseguido probar que existe un algoritmo que, dado un dato de entrada de tamaño $n$, calcula el resultado en un tiempo que puede ser descrito mediante una espresión exponencial de la forma $2^{2^{poli(n)}}$.

Ejemplos: calcular el complementario de una expresión regular; decidir si una fórmula de la lógica $CTL^+$ es satisfactible.

#### 1.5.2.6 Clases de complejidad k-EXPTIME

Existe un número infinito de clases formadas por problemas de decisión que requieren un tiempo de cálculo descriptible mediante torres de exponenciales, es decir, expresiones de la forma $2^{2^{2^{poli(n)}}}$, $2^{2^{2^{2^{poli(n)}}}}$, etc.

#### 1.5.2.7 Clases de complejidad ELEMENTARY y NONELEMENTARY

La clase ELEMENTARY es la clase constituida por la unión de todas las clases de la forma k-EXPTIME. La clase NONELEMENTARY es la clase de los problemas de decisión que no están en ELEMENTARY. El tiempo requerido para resolver los problemas que pertenecen a NONELEMENTARY es indescriptible: necesitan un tiempo superior a cualquier torre de exponenciales.

Ejemplos de problemas en NONELEMENTARY: decidir si dos expresiones regulares en las que aparece la operación del complementario son equivalentes; el problema de decisión de la lógica de segundo orden monádica sobre árboles; decidir si dos términos cerrados del $\lambda$-cálculo con tipos son $\beta$-convertibles; etc.

### 1.5.3 ¿P = NP? ¿NP = co-NP? Cuestiones sin aclarar

La pregunta ¿P = NP? fue planteada hace décadas pero todavía no se ha conseguido dar con la respuesta. Tampoco se ha conseguido responder a la pregunta ¿NP = co-NP?. Hay muchas más cuestiones de este estilo no resueltas aún.

### 1.5.4 Complejidad espacial: clases

A la hora de determinar la eficiencia de un algoritmo, además de considerar el tiempo requerido para realizar los cálculos, también es importante considerar el espacio de memoria requerido para realizar los cálculos. Si se necesita mucho espacio de memoria para realizar los cálculos, el algoritmo no será eficiente: habrá que guardar muchos datos, habrá que recuperar los datos guardados, habrá que recorrer y gestionar el espacio de memoria ocupado. De la misma forma que se definen clases de complejidad teniendo en cuenta el tiempo, también se definen clases de complejidad teniendo en cuenta el espacio de memoria requerido. A continuación se presenta una clasificación muy básica.

#### 1.5.4.1 Clase de complejidad PSPACE

Un problema de decisión pertenece a la clase de complejidad espacial PSPACE si existe un algoritmo que realiza los cálculos correspondientes utilizando un espacio de memoria descriptible mediante un polinomio con respecto al tamaño del dato de entrada. Es decir, si el tamaño del dato de entrada es $n$, el espacio de memoria requerido para realizar el cálculo será descriptible mediante un polinomio que depende de $n$: $poli(n)$.

A modo de ejemplo, podemos mencionar el problema de decidir si una fórmula de la lógica proposicional cuantificada es cierta. También podemos mencionar el caso de la lógica proposicional temporal lineal.

#### 1.5.4.2 Clase de complejidad NPSPACE

A diferencia de lo que ocurre con la complejidad temporal, en el caso de la complejidad espacial se ha conseguido probar que la clase PSPACE y la clase NPSPACE son iguales.

#### 1.5.4.3 Clase de complejidad EXPSPACE

Un problema de decisión pertenece a la clase de complejidad espacial EXPSPACE si existe un algoritmo que realiza los cálculos correspondientes utilizando un espacio de memoria descriptible mediante una expresión exponencial con respecto al tamaño del dato de entrada. Es decir, si el tamaño del dato de entrada es $n$, el espacio de memoria requerido para realizar el cálculo será descriptible mediante una expresión exponencial que depende de $n$: $2^{poli(n)}$.

Por ejemplo, decidir si dos expresiones regulares que cumplen ciertos requerimientos definen el mismo lenguaje.

### 1.5.5 Relación entre las diferentes clases de complejidad

A continuación se indican las relaciones entre las distintas clases de complejidad presentadas en los apartados anteriores:

- $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME} \subseteq \mathsf{EXPSPACE} \subseteq \mathsf{2\text{-}EXPTIME} \subseteq \mathsf{ELEMENTARY}$
- $\mathsf{P} \subseteq \mathsf{co\text{-}NP} \subseteq \mathsf{PSPACE}$
- $\mathsf{PSPACE} = \mathsf{NPSPACE} \subsetneq \mathsf{EXPSPACE}$
- $\mathsf{P} \subsetneq \mathsf{EXPTIME}$

# 1.6.

# Modelos de computación

Tal como se ha dicho en el apartado anterior, estudiar la computabilidad de una tarea consiste en averiguar si existe un algoritmo (o, equivalentemente, un programa) que realice esa tarea.

El estudio de los límites de la computación se ha llevado a cabo desde distintos enfoques. Cada enfoque ha dado lugar a un modelo de computación. Un modelo de computación es un sistema matemático con una sintáxis formal [1] y una semántica formal [2]. **Cada modelo computacional da una definición de la computación y ofrece una manera de formular los algoritmos**. [3] Los modelos computacionales más relevantes son los siguientes:

- El modelo de las máquinas de Turing o el modelo de los autómatas. El formalismo que se utiliza en este modelo computacional está basado en estados y transiciones entre estados. Una máquina de Turing —o un autómata— es un algoritmo expresado mediante el formalismo de este modelo computacional. En este modelo, la idea básica de computación consiste en manipular estructuras lineales formadas por distintos tipos de elementos. Dichas estructuras lineales son, por ejemplo, de la forma $(a(b(a(a(c(a()))))))$, donde $a$, $b$ y $c$ son tres elementos distintos. Las estructuras habitualmente son abreviadas como $abaaca$ o como $abaaca\varepsilon$, donde $\varepsilon$ representa la estructura vacía $()$. Las estructuras como $abaaca$ —o $abaaca\varepsilon$—, reciben el nombre de palabras. Los cálculos que se realizan en este modelo computacional consisten en analizar ese tipo de estructuras para obtener otra estructura o para decidir si la estructura cumple o no cumple alguna determinada propiedad. Cada algoritmo diseñado para realizar una tarea de ese estilo recibe el nombre de máquina de Turing o, alternativamente, el nombre de autómata. Las máquinas de Turing son los autómatas más generales que se pueden diseñar. Puesto que las máquinas de Turing pueden ser, en general, muy complicadas, se suelen presentar primero los autómatas finitos y los autómatas de pila, que son casos particulares de las máquinas de Turing, pero bastante más sencillos.

[1] Normas precisas para escribir expresiones correctas.

[2] Normas precisas para interpretar el significado de las expresiones correctas.

[3] Dada la especificación $S$ de un problema $P$, un algoritmo $A$ para $P$ es un procedimiento que siempre encuentra la solución teniendo en cuenta la especificación $S$.

- El modelo del $\lambda$-cálculo. El formalismo que se utiliza en este modelo computacional está basado en funciones matemáticas. En este modelo computacional, los elementos básicos son las funciones. Realizar un cálculo consiste en transformar expresiones que han sido construidas utilizando funciones. Un cálculo termina cuando la expresión que se tiene no admite más transformaciones. Esa expresión que no puede ser transformada será el resultado del cálculo.

- El modelo de las funciones recursivas. El formalismo que se utiliza en este modelo computacional está basado en funciones iniciales, funciones recursivas primitivas y funciones recursivas parciales.

- El modelo de la lógica. El formalismo que se utiliza en este modelo computacional está basado fundamentalmente en la resolución de la Lógica de Primer Orden.

- El modelo de las máquinas de Post. El formalismo que se utiliza en este modelo computacional está basado en diagramas de flujo.

Cada modelo de computación da lugar a un estilo (o paradigma) de programación:

- Máquinas de Turing: Programación imperativa (ADA, C, Java, etc.)

- $\lambda$-cálculo: Programación funcional (Haskell, Scheme, ML, etc.)

- Funciones recursivas: Programación algebraica (Opal, Magma).

- Lógica: Programación lógica (Prolog).

- Máquinas de Post: Programación imperativa (ADA, C, Java, etc.).

En esta asignatura, por una parte, se utilizará el modelo del $\lambda$-cálculo para estudiar la computabilidad —en concreto, para probar la existencia de funciones no computables. Por otra parte, se utilizará el modelo basado en estados y transiciones de estados (Máquinas de Turing o autómatas generales) para estudiar los aspectos teóricos de la complejidad computacional. De esa manera, se trabajarán dos de los modelos de computación mencionados.

# 1.7.

# Máquinas basadas en estados y transiciones

Una máquina basada en estados o una máquina de Turing o un autómata es una manera de representar un algoritmo, es decir, es un formalismo para representar una estrategia que sirve para realizar un cálculo. Se utiliza el nombre "máquina" o "máquina de Turing" o "autómata" porque cuando surgió este formalismo, todavía los nombres de "programa" y "algoritmo" no estaban extendidos. Puesto que se quería expresar qué secuencia de acciones se debía realizar para llevar a cabo un cálculo, se asoció esa idea a la de una máquina que paso a paso iba calculando el resultado. A las máquinas basadas en estados se les llama, habitualmente, máquinas de Turing porque fue el británico Turing [1] el que las inventó. Por tanto, hay que tener claro que una máquina de Turing es otro nombre más para designar "algoritmo". De la misma forma que en la asignatura Metodología de la Programación al algoritmo se le llamaba "invariante" en vez de "algoritmo".

Los autómatas finitos y autómatas de pila son algoritmos expresados mediante un formalismo que se obtiene al simplificar el formalismo de las máquinas de Turing.

Una máquina basada en estados (máquina de Turing, autómata finito, autómata de pila, etc.) constará de los siguientes elementos:

- Un dispositivo de **entrada**, desde donde recibirá datos.
- Un dispositivo de **salida**, donde se escribirán resultados.
- Un **sistema de control** formado por estados y una función de transición entre estados. Tras leer cada dato, la función de transición indica cuál es el nuevo estado. En general, al leer un dato se cambia de estado. Si se ha de generar alguna salida, esta puede ser generada durante el proceso de cálculo o al final.
- Un dispositivo de **memoria** para poder almacenar información.

[1] Alan Mathison Turing (Reino Unido, 23-06-1912—07-06-1954).

## 1.7.1 Ejemplos de máquinas basadas en estados y transiciones

### 1.7.1.1 Máquina que controla un LED

En este ejemplo mostramos una máquina (ver figura 1.7.1 de la página 55) que recibe, como dato de entrada, una secuencia de repeticiones de los símbolos 0 y 1. Esas repeticiones aparecerán mezcladas de cualquier manera. La salida es un diodo emisor de luz (LED)[2]. Cada vez que se recibe el símbolo 1, se ha de encender el LED y cuando se recibe el símbolo 0, se ha de apagar el LED. Si se reciben varios unos seguidos, el LED se ha de mantener encendido. Cuando se reciben varios ceros seguidos, el LED se ha de mantener apagado. El control de la máquina se realiza mediante dos estados denominados "Apagado" y "Encendido". Mediante la flecha que llega al estado "Apagado" desde la izquierda se indica que al poner la máquina en marcha estamos en ese estado y, por tanto, con el LED apagado. La flecha que sale desde el estado "Apagado" y vuelve a él mismo, indica que mientras se lean ceros se seguirá en ese estado. Si se lee un uno se pasará al estado "Encendido" y el LED se encenderá. Una vez que la máquina esté en el estado "Encendido", si se lee un uno se seguirá en ese estado, mientras que si se lee un cero se volverá al estado "Apagado" y el LED se apagará. El proceso continúa mientras haya datos de entrada.

### 1.7.1.2 Máquina que decide si una palabra es no vacía y consta solo de $a$'s

En la figura 1.7.2 de la página 55, se muestra una máquina que recibe como entrada una cadena de símbolos que puede contener los símbolos $a$, $b$ y $c$ cualquier número de veces y en cualqier orden. La máquina decide si la palabra es una secuencia no vacía de repeticiones de $a$. En caso afirmativo responde con "Sí" y en caso negativo con "No". Por tanto, la salida es un "Sí" o un "No". El control de la máquina está formado por tres estados. El estado $q_0$ es el estado inicial. Al empezar a leer una cadena de símbolos estamos en el estado $q_0$. Esto se indica en la figura 1.7.2 de la página 55 mediante la flecha que llega al estado $q_0$ desde la izquierda. Si el primer símbolo es $a$, la máquina pasa al estado $q_1$. Si el primer símbolo es $b$ o $c$, la máquina pasa al estado $q_2$. Si la máquina está en el estado $q_1$, siempre que venga $a$, la máquina se mantiene en el mismo estado, en el $q_1$. En cambio si viene $b$ o $c$, se pasa al estado $q_2$. Si la máquina está en el estado $q_2$, venga lo que venga, se mantiene en ese mismo estado. Si cuando se termina de leer toda la cadena de símbolos la máquina está en el estado $q_1$, eso quiere decir que la cadena de símbolos no contenía ninguna aparición de $b$ ni ninguna aparición de $c$ y que contenía al menos una aparición del símbolo $a$ (es decir, no era una secuencia vacía). Por el contrario, si cuando se termina de leer toda la cadena de símbolos la máquina está en el estado $q_2$, eso quiere decir que la cadena de símbolos no estaba formada exclusivamente por repeticiones de $a$. Conviene darse cuenta de que al finalizar de leer la cadena, la máquina estará en el estado $q_0$ solo si la cadena de entrada es vacía. Por tanto, si se termina en el estado $q_1$, la salida será "Sí", mientras que si se termina en un estado que no sea el $q_1$, la salida será "No". El estado $q_1$ está señalado con doble círculo para indicar que es el estado en el que se ha de terminar para que la respuesta sea afirmativa.

[2]LED: Light-Emitting Diode

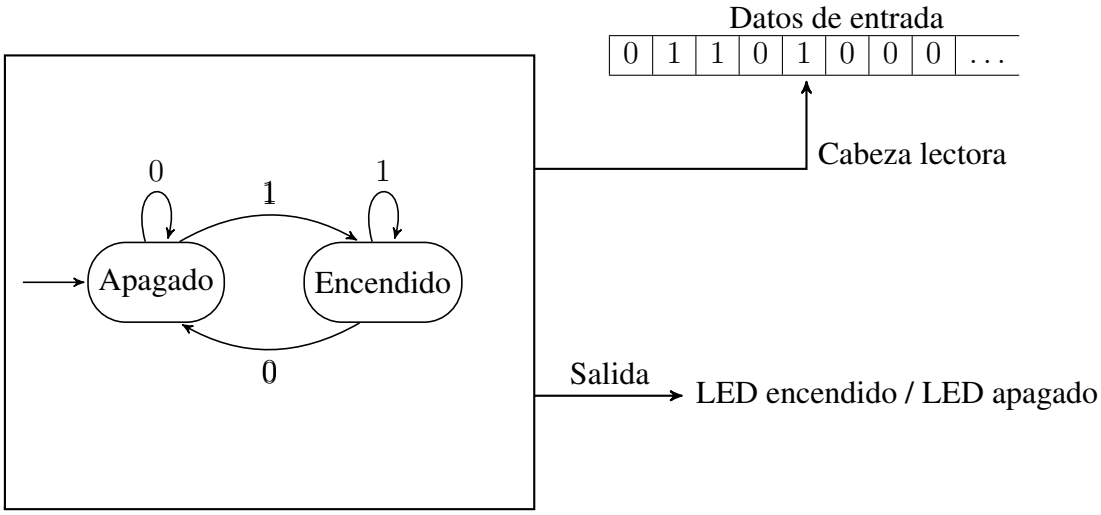


**Figura 1.7.1.** Máquina de estados para controlar el encendido y el apagado de un LED. Ejemplo del apartado 1.7.1.1 de la página 54.

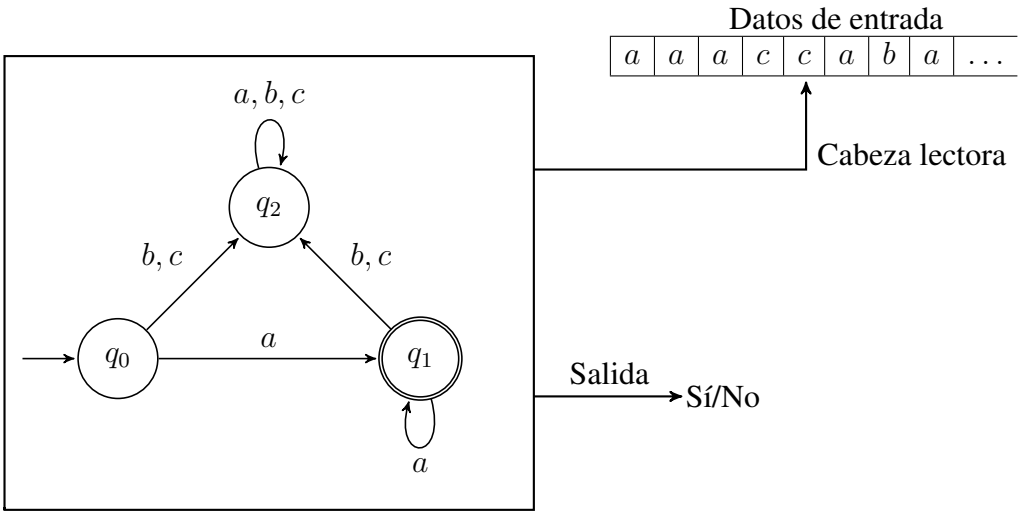


**Figura 1.7.2.** Máquina que decide si la palabra es no vacía y está formada solo por repeticiones de $a$. Ejemplo del apartado 1.7.1.2 de la página 54.

#### 1.7.1.3 Máquina que junta dos cadenas de unos separadas por # y con un blanco al final

En la figura 1.7.3 de la página 57, se muestra una máquina que recibe como entrada dos secuencias de unos. Las dos secuencias terminan con el símbolo # y después del segundo # hay por lo menos un blanco, representado mediante el símbolo ⊔. El control de la máquina consta de cuatro [3] estados, $q_0$, $q_1$, $q_2$ y $q_3$. En las transiciones de las máquinas mostradas en los ejemplos 1.7.1.1 (página 54) y 1.7.1.2 (página 54), solo aparece un símbolo. Dicho símbolo indica que si se lee ese símbolo desde el dispositivo de entrada, se ha de ir al estado señalado por el arco correspondiente. Además, en las máquinas mostradas en los ejemplos 1.7.1.1 (página 54) y 1.7.1.2 (página 54), la cabeza lectora siempre se mueve hacia la derecha. En cambio, en la máquina de este ejemplo, en las transiciones tenemos tripletas ordenadas [4] de la forma $\alpha/\beta/\gamma$ [5]. Los componentes $\alpha$ y $\beta$ son siempre del conjunto de los símbolos que pueden aparecer en la Entrada/Salida, es decir, del conjunto $\{1, \#, \sqcup\}$. El componente $\gamma$ es un símbolo del conjunto $\{D, I, P\}$, donde $D$ indica "mover la cabeza de lectura y escritura una posición hacia la derecha", $I$ indica "mover la cabeza de lectura y escritura una posición hacia la izquierda" y $P$ indica "permanecer en la misma posición". Por tanto, una tripleta ordenada de la forma $\alpha/\beta/\gamma$ expresa que si en la posición en la que se encuentra la cabeza de lectura y escritura se tiene el símbolo $\alpha$, entonces en esa misma posición hay que escribir el símbolo $\beta$ y la cabeza de lectura y escritura ha de realizar el movimiento representado por $\gamma$ (derecha, izquierda o permanecer en el mismo sitio). Además, habrá que ir al estado señalado por la flecha correspondiente. Mediante este ejemplo se muestra que en las máquinas de Turing es posible escribir resultados y controlar el movimiento de la cabeza de lectura y escritura.

La máquina parte desde el estado $q_0$ y con la cabeza de lectura y escritura posicionada al principio del dispositivo de Entrada/Salida. En este ejemplo, suponemos que la entrada es siempre correcta, es decir, se nos darán dos secuencias de unos con un símbolo # al final de cada secuencia y luego habrá por lo menos un blanco (representado mediante ⊔). Cuando estamos en el estado $q_0$, al leer un uno se escribe un uno en la misma posición y la cabeza de lectura y escritura se mueve una posición hacia la derecha, pero la máquina se mantiene en el estado $q_0$. Al leer la primera aparición del símbolo #, es decir, cuando ha terminado la primera secuencia de unos, se escribe un uno en esa posición (y se elimina #) y, a continuación, la cabeza de lectura y escritura se mueve una posición hacia la derecha. Además, la máquina cambia de estado, y pasa al estado $q_1$. Una vez en el estado $q_1$, se procede a tratar la segunda secuencia de unos. Al leer un uno, se escribe un uno en la misma posición y la cabeza de lectura y escritura se mueve una posición hacia la derecha. La máquina se mantiene en el estado $q_1$. Cuando aparece el símbolo # por segunda vez, es sustituido por un blanco (representado mediante ⊔) y a continuación la cabeza de lectura y escritura se mueve hacia la izquierda, y queda posicionada en el último uno de la segunda secuencia. Además, la máquina cambia al estado $q_2$. En el estado

[3] Es posible hacer lo mismo con una máquina de tres estados, tal como se muestra en la figura 1.7.4 de la página 57.

[4] tripleta: conjunto de tres personas, animales o cosas; sinónimos: trío, terna. Una tripleta ordenada es una tripleta en la que se ha establecido un orden entre los tres componentes que conforman la tripleta.

[5] $\alpha/\beta/\gamma$: alfa/beta/gamma

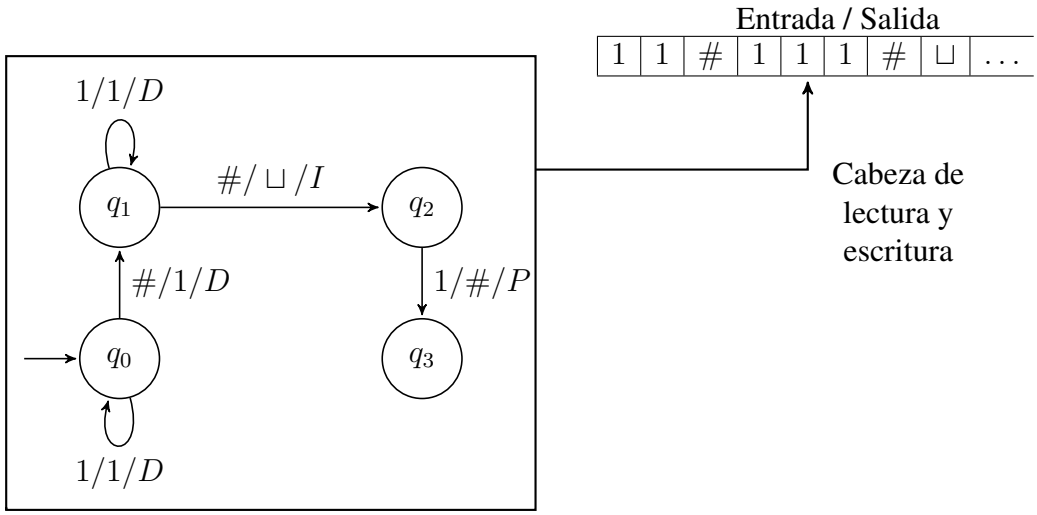


**Figura 1.7.3.** Máquina que junta dos cadenas de unos terminadas en # y un blanco al final. Ejemplo del apartado 1.7.1.3 de la página 56.

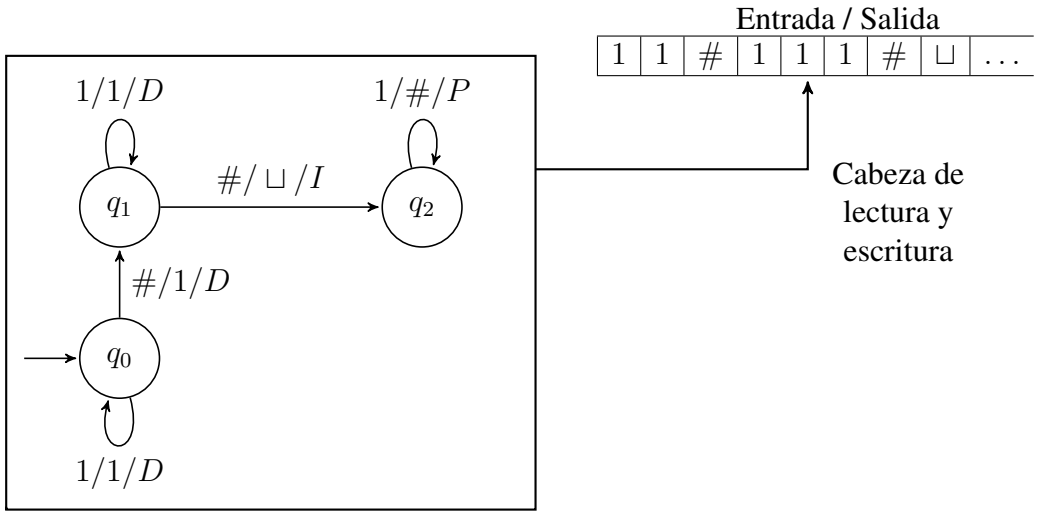


**Figura 1.7.4.** Máquina de tres estados que junta dos cadenas de unos terminadas en # y un blanco al final. Ejemplo del apartado 1.7.1.3 de la página 56.

$q_2$, se lee otra vez el último uno (que es donde estamos posicionados), ese uno es sustituido por $\#$ y la máquina pasa al estado $q_3$ y termina el cálculo.

Si consideramos la entrada que aparece en la figura 1.7.3 de la página 57,

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | # | 1 | 1 | 1 | # | ⊔ | ··· |

el resultado final será el siguiente:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | # | ⊔ | ⊔ | ··· |

La máquina de la figura 1.7.4 de la página 57 realiza la misma tarea pero solo tiene tres estados.

## 1.7.2 Memoria de las máquinas basadas en estados

En el ejemplo mostrado en el apartado 1.7.1.1 (página 54), el LED permanece encendido siempre que la máquina esté en el estado "Encendido" y el LED permanece apagado siempre que la máquina esté en el estado "Apagado". Por tanto, en ese ejemplo, los estados son la memoria de esa máquina ya que le sirven para recordar qué es lo último que se ha leído y saber si el LED ha de permanecer apagado o encendido hasta que llegue otro dato de entrada (otro 0 u otro 1).

También en el ejemplo mostrado en el apartado 1.7.1.2 (página 54), los estados son la memoria de la máquina en cuestión, ya que le sirven para recordar si hasta el momento solo se han leído $a$'s o no. Así, siempre que se está en el estado $q_1$, se sabe que se ha leído al menos una $a$ y que de momento todos los caracteres leídos han sido $a$'s, mientras que si estamos en el estado $q_2$, sabemos que ha aparecido alguna $b$ o alguna $c$. Además, al finalizar de leer la cadena de entrada, si estamos en el estado $q_1$, hay que responder que "Sí" mientras que si no estamos en el estado $q_1$, hay que responder que "No". Por tanto, los estados desempeñan el papel de dispositivo de memoria.

En el ejemplo mostrado en el apartado 1.7.1.3 (página 56), por una parte los estados desempeñan el papel de dispositivo de memoria de esa máquina ya que le sirven para recordar si estamos en la primera o en la segunda secuencia de unos y también para recordar si estamos en el primer o segundo $\#$ cuando se lee $\#$. Así, si estamos en el estado $q_0$, sabemos que estamos en la primera secuencia de unos mientras que si estamos en el estado $q_1$, sabemos que estamos en la segunda secuencia de unos. De la misma forma, si estamos en el estado $q_0$ y se lee $\#$, sabemos que estamos ante la primera aparición de $\#$, mientras que si estamos en el estado $q_1$ y se lee $\#$, sabemos que estamos ante la segunda aparición de $\#$. Por otra parte, el propio dispositivo de Entrada/Salida, funciona en este caso como memoria, ya que en él se va almacenando el resultado.

En resumen, se considera que el dispositivo de memoria, necesario para realizar cálculos, está repartido entre los estados y el dispositivo de Entrada/Salida. La memoria da capacidad de

cálculo, es decir —al menos hasta cierto punto— a más memoria más capacidad de cálculo. Por tanto, dependiendo de la memoria disponible, se podrán diseñar máquinas de mayor o menor capacidad de cálculo.

### 1.7.3 Representación de los datos de entrada y salida en las máquinas basadas en estados: Lenguajes

Los datos de entrada y salida serán cadenas de símbolos. En cada caso se indicará qué símbolos se pueden utilizar al generar las cadenas de símbolos. Por tanto, realizar cálculos significa procesar las cadenas de entrada para obtener el resultado, tal como se ha visto en los ejemplos de los apartados 1.7.1.1 (página 54), 1.7.1.2 (página 54) y 1.7.1.3 (página 56). En el modelo de computación basado en máquinas de estados, la computación es planteada como procesamiento de cadenas de símbolos. Este planteamiento es muy natural, ya que las tareas realizadas por un ordenador pueden ser vistas como procesamiento de cadenas de símbolos. Por ejemplo, cuando se quiere ejecutar un programa, le damos al ordenador el programa (que viene a ser una cadena de símbolos o caracteres) y los datos de entrada (también son cadenas de símbolos o caracteres) y el ordenador realiza las tareas representadas por el programa que le hemos dado. En este modelo computacional, los lenguajes son conjuntos de cadenas de símbolos (o caracteres) y cada máquina tendrá asociado un lenguaje, tal como se verá más adelante.

De todas formas, conviene recordar que, tal como se ha dicho en el apartado 1.6 (página 51), las cadenas de símbolos o caracteres representan, en realidad, estructuras lineales donde cada estructura se obtiene añadiendo un componente a otra subestructura.

### 1.7.4 Estudio de la computabilidad mediante máquinas basadas en estados

Tal como se ha mostrado en los ejemplos de los apartados 1.7.1.1 (página 54), 1.7.1.2 (página 54)y 1.7.1.3 (página 56), las máquinas de estados pueden tener distintas características:

- Posibilidad de generar distintos tipos de salidas (señales luminosas, respuestas binarias del tipo “Sí”/“No” y cadenas de símbolos o cadenas de caracteres obtenidas mediante un cálculo).
- Posibilidad de leer el dato de entrada una única vez (no permitiendo que la cabeza lectora retroceda) o posibilidad de poder leer más de una vez los datos de entrada (permitiendo que la cabeza lectora retroceda).
- Posibibilidad de hacer uso de distintos tipos de memoria: utilizando únicamente los estados como memoria o permitiendo dispositivos adicionales de memoria (por ejemplo, permitiendo utilizar el dispositivo de Entrada/Salida como memoria para almacenar resultados o para poder releer los datos).

- Posibibilidad de hacer uso de una memoria finita (cuando la memoria la constituyen solo los estados, puesto que el número de estados será siempre finito) o de una memoria potencialmente infinita (cuando, además de los estados, se puedan utilizar dispositivos adicionales como memoria).

Otra característica importante es la de que una máquina sea "Determinista" o "No determinista". En las máquinas deterministas, el cálculo asociado a un dato de entrada, sigue un único camino que está determinado por el propio dato y la máquina en cuestión. Se tiene, por tanto, una manera única para resolver el problema. En cada momento, hay una única opción para avanzar en el cálculo del resultado final. En cambio, en las máquinas no deterministas, para un dato de entrada se pueden plantear, en general, distintos caminos o posibilidades para realizar el cálculo correspondiente. En cada momento se pueden tener, por tanto, varias opciones para avanzar en el cálculo del resultado final, es decir, para abordar la resolución del problema. Algunas de esas opciones pueden ser fallidas (en el sentido de que no sirven para obtener directamente el resultado definitivo) pero con que una de las opciones funcione (en el sentido de que la opción en cuestión sirva para obtener directamente el resultado definitivo) es suficiente para garantizar la corrección del cálculo y la corrección de la máquina.

Los diferentes caminos posibles en la versión no determinista planteada para resolver un problema, se obtendrán, la mayoría de las veces, descomponiendo el único camino posible de la versión determinista planteada para resolver el problema en cuestión. Por ello, los distintos caminos de la versión no determinista se pueden entender como distintas subestrategias que se obtienen al descomponer la estrategia única de la versión determinista. Esto supone que es posible que en las máquinas no deterministas se tengan que ir probando todas las opciones planteadas (puesto que son subestrategias y no estrategias totalmente completas). Si las subestrategias que se van probando van fallando, hay que probar alguna de las que quedan por probar. Aunque en el peor de los casos haya que probar todas las subestrategias, el plantear distintas subestrategias abre la posibilidad de que cada subestrategia sea llevada a cabo por una máquina distinta, posibilitando ejecuciones en paralelo que, en general, reducirán el tiempo de cálculo.

# 1.8.

# Contenido del resto de temas

En el Tema 2, se presentarán algunos algoritmos del área de los **sistemas inteligentes**. En concreto, se estudiarán dos algoritmos para aprender (o adivinar) dos tipos de fórmulas de la lógica proposicional. El usuario tendrá en mente una fórmula de la lógica proposicional y habrá que aplicar uno de los dos algoritmos para adivinar esa fórmula. Pero esos algoritmos no son capaces de adivinar la fórmula utilizando solo los datos que el usuario ha aportado al principio. Necesitan ir obteniendo más información a lo largo del proceso de adivinación: son algoritmos cuya especificación es no funcional.

En el Tema 3, se presentarán las nociones básicas sobre lenguajes, bajo la perspectiva de que un lenguaje es un conjunto (finito o infinito) de palabras o cadenas finitas de símbolos. Se definirán operaciones sobre palabras y sobre lenguajes y se indicará cómo definir lenguajes de manera formal mediante notación matemática. También se tratará la enumerabilidad y la no enumerabilidad de algunos conjuntos infinitos. Los resultados de enumerabilidad sirven para definir codificaciones entre elementos de distintos tipos (números, palabras, pares de números y pares de palabras). Se verá que un problema en el que se ha de **devolver una respuesta afirmativa o negativa para cada dato de entrada**, puede ser planteado como un lenguaje. Cada dato que recibe respuesta afirmativa pertenece al lenguaje y cada dato que no recibe respuesta afirmativa no pertenece al lenguaje. Por tanto, **un lenguaje es la especificación de un problema de decisión** y se quiere saber si es posible diseñar un algoritmo capaz de averiguar si un dato pertenece al lenguaje. El tema terminará con un resultado teórico importante: hay lenguajes no decidibles. Esto último quiere decir que **para algunos lenguajes, no es posible diseñar un algoritmo** que, dada una palabra, sea capaz de responder "Sí" si la palabra pertenece al lenguaje y "No" si la palabra no pertenece al lenguaje. Por tanto, hay problemas en los que se ha de devolver una respuesta afirmativa o negativa para cada dato de entrada pero no es posible diseñar un algoritmo que haga esa tarea adecuadamente.

En el Tema 4, se presentarán unas nociones muy básicas sobre el $\lambda$-cálculo. Se definirá el concepto de **algoritmo universal** utilizando el $\lambda$-cálculo.

En el Tema 5, se tomará el concepto de $\lambda$-término universal como base y se introducirán las nociones de **problema decidible**, problema **no decidible**, problema **reconocible** y problema **no**

**reconocible**. Los problemas decidibles son aquellos para los que se puede diseñar un algoritmo que responde correctamente ("Sí"/"No") para todos los datos de entrada posibles. Los problemas no decidibles pero sí reconocibles son aquellos para los que se puede diseñar un algoritmo que responde correctamente para los datos de entrada que requieren una respuesta afirmativa ("Sí") pero el algoritmo no es capaz de responder negativamente ("No") para todos los datos que requieren una respuesta negativa. La idea es que el algoritmo se sumerge en un cálculo infinito y no termina de generar la respuesta negativa ("No"). Los problemas no reconocibles son aquellos para los que no se puede diseñar ningún algoritmo que responda correctamente ni para los datos de entrada que requieren una respuesta afirmativa ("Sí"). Por tanto, ante datos de entrada que requieren una respuesta afirmativa, cualquier algoritmo que se diseñe se sumergirá en un cálculo infinito y no terminará de generar la respuesta afirmativa ("Sí"). De la misma forma, ante datos de entrada que requieren una respuesta negativa, cualquier algoritmo que se diseñe se sumergirá también en un cálculo infinito y no terminará de generar la respuesta negativa ("No"). Esta clasificación de los problemas en decidibles, no decidibles, reconocibles y no reconocibles, da lugar, a su vez, a la clasificación de **problemas computables**, **semicomputables** y **no computables**. Y, para terminar, se presentarán algunos problemas concretos que son indecidibles y se dará la prueba matemática de la indecidibilidad de esos problemas.

A partir del Tema 6, se presentarán distintas variantes de máquinas de estados que generan solo respuestas binarias del tipo "Sí"/"No". Por una parte, están las máquinas que disponen de memoria finita (puesto que utilizarán únicamente los estados como memoria). Se trata, en concreto, de los autómatas finitos. Mediante autómatas finitos es posible representar estrategias deterministas o no deterministas pero se probará que esta característica (la del no determinismo) no incrementa la capacidad de cálculo en el caso de los autómatas finitos. Es decir, en el ámbito de los autómatas finitos, toda estrategia no determinista puede ser expresada mediante una estrategia determinista. Se terminará el Tema 5 mostrando que este tipo de máquinas (autómatas finitos) no es suficiente (hay cálculos que quedan fuera del alcance de este tipo de máquinas). Consecuentemente, se llegará a la conclusión de que es necesario definir otros tipos de máquinas con más capacidad (memoria adicional, potencialmente infinita) como es el caso de los autómatas con pila que utilizan una memoria externa adicional (en principio infinita o al menos sin restricciones de tamaño) con características de pila (los nuevos datos solo se pueden colocar en la cima y, en cada momento, solo se puede acceder al dato que está en la cima; esto significa que para acceder a un elemento que no está en la cima, hay que eliminar y, por tanto, perder los elementos que están por encima de ese elemento en cuestión). Las ideas de determinismo y no determinismo aparecen también en los autómatas con pila y, además, en este caso algunas estrategias no deterministas no son expresables de manera determinista. En todo momento se ha de tener claro que las nociones de **autómata finito**, **autómata con pila** y **máquina de Turing** son sinónimos de la noción de **algoritmo** (estrategia para realizar un cálculo o resolver un problema). Para diseñar máquinas de Turing se utilizará la versión más general del formalismo definido en el modelo computacional de las máquinas de Turing. En el caso de los autómatas finitos y autómatas de pila, se imponen algunas restricciones sobre dicho formalismo. Las máquinas de Turing pueden generar respuestas binarias del tipo "Sí"/"No" pero también pueden realizar cálculos que terminen generando cadenas de símbolos o caracteres

como respuesta. Estas cadenas de caracteres obtenidas como respuesta, se almacenarán en el dispositivo de Entrada/Salida, el cual hace de memoria adicional sin restricciones de tamaño (es potencialmente infinito) y sin restricciones de acceso, es decir, sin restricciones de una memoria de tipo pila.

Para terminar el temario, se introducirán los conceptos básicos de la Teoría de la **Complejidad Computacional**. En concreto, se presentará el problema **¿P=NP?**. Lo cual quiere decir, ¿son el determinismo y el no determinismo equivalentes en cuanto a complejidad computacional? o dicho de otra forma, ¿son igual de costosos el proceso de buscar una solución y el proceso de comprobar si una propuesta de solución es realmente una solución?

# 1.9.

# Resumen

## 1.9.1 Teoría de la computabilidad: breve historia

El objetivo de la **teoría de la computabilidad** es el estudio matemático de los modelos de computación. La Tería de la computabilidad es una rama de la **lógica matemática**.

Los inicios de la lógica matemática actual, se sitúan en la Grecia Clásica. En el campo de la lógica, podemos mencionar a Aristóteles (384-322 a.C.). Pero también es importante mencionar que en aquella época se formularon algoritmos para realizar cálculos matemáticos. Por ejemplo, el método de Euclides (323-283 a.C.) para calcular el máximo común divisor de dos números enteros positivos. También Pitágoras (572-497 a.C.) dejó resultados relevantes en aritmética y geometría. Tras el ocaso de la Antigua Grecia, durante siglos las técnicas de la lógica se utilizaron sobre todo en filosofía y dialéctica para estructurar y argumentar las discusiones y los razonamientos. Fue en el siglo XIX cuando se reavivó el interés por utilizar las técnicas de la lógica en el campo de la matemática. De esa manera, surgió lo que hoy se conoce como lógica matemática. Inicialmente, el objetivo de la lógica matemática era establecer los fundamentos teóricos de la matemática y, a partir de esos fundamentos, formalizar toda la matemática. Pero en 1931, el matemático Kurt Gödel [1] demostró que ese objetivo de la lógica matemática no es alcanzable: **algunos aspectos de la matemática no se pueden formalizar (teoremas de incompletitud de Gödel)**. De todas formas, se comprobó que muchas áreas de la matemática que son importantes para la ciencia sí son formalizables y, consecuentemente, la lógica matemática siguió adelante y se han obtenido resultados relevantes. El mismo Gödel demostró que la lógica de primer orden (o lógica de predicados) es completa, es decir, dada una fórmula de la lógica de primer orden, si la fórmula es válida (cierta), entonces esa validez se podrá demostrar matemáticamente. En cambio, los teoremas de incompletitud de Gödel establecen que en algunos sitemas más expresivos que la lógica de primer orden, hay fórmulas que son ciertas pero que no pueden ser demostradas matemáticamente.

La teoría de la computabilidad se ha desarrollado como una rama de la lógica matemática y, por consiguiente, está basada en las técnicas y métodos disponibles en la lógica matemática.

[1] Kurt Gödel: Imperio austrohúngaro, 28-04-1906 — Estados Unidos, 14-01-1978

En 1936, el británico Alan Turing[2] demostró que **para algunos cálculos planteables, no es posible diseñar ningún algoritmo que los resuelva**. Pero puesto que sí es posible diseñar programas que son de gran utilidad para las personas, la informática —al igual que la lógica matemática— ha seguido adelante y se han obtenido resultados importantes.

En el área de los cálculos que sí son programables, ha surgido **otra dificultad**: **la eficiencia**. Algunos programas son muy ineficientes porque requieren de mucho tiempo o mucha memoria para obtener el resultado. Para algunos de esos programas ineficientes, se sabe que no es posible diseñar programas eficientes. Pero en el caso de otros programas ineficientes, no se sabe si existen programas eficientes. **La teoría de la complejidad** se encarga del estudio de la eficiencia alcanzable a la hora de realizar cálculos mediante programas.

### 1.9.2 Esquema resumen

En las figuras 1.9.1, 1.9.2, 1.9.3 y 1.9.4 —páginas 67, 68, 69 y 70—, se muestra un esquema en el que se recogen los principales asuntos que se tratan en esta asignatura.

### 1.9.3 Especificación funcional y no funcional: Inteligencia artificial y sistemas inteligentes

En la figura 1.9.5 de la página 71, se muestra la relación entre los conceptos de especificación funcional, especificación no funcional, sistemas inteligentes e inteligencia artificial.

### 1.9.4 Computabilidad e incomputablidad

En la figura 1.9.6 de la página 72, se muestra la relación entre los conceptos de especificación funcional, incomputabilidad, computabilidad, semicomputabilidad, computabilidad no efectiva o inalcanzable y computabilidad intratable o ineficiente.

[2] Alan Mathison Turing (Reino Unido, 23-06-1912—07-06-1954).

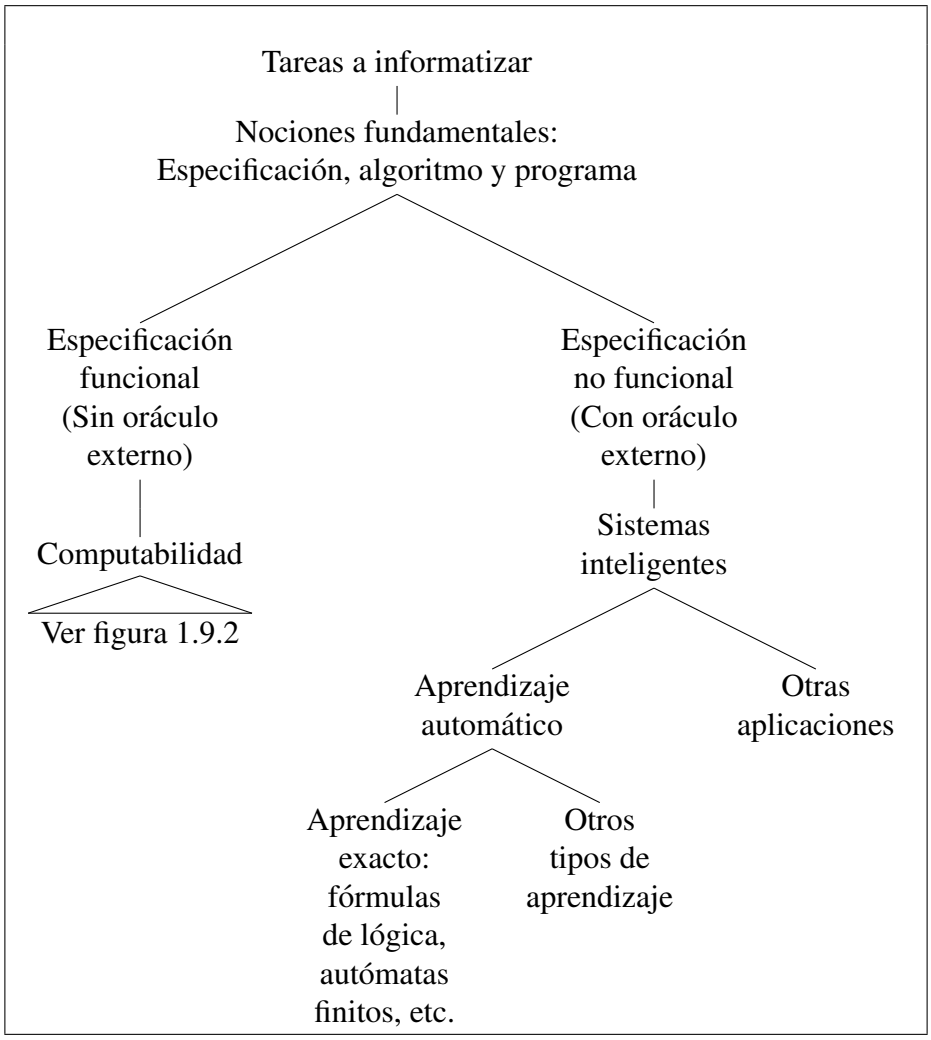


**Figura 1.9.1.** Esquema (parte 1)

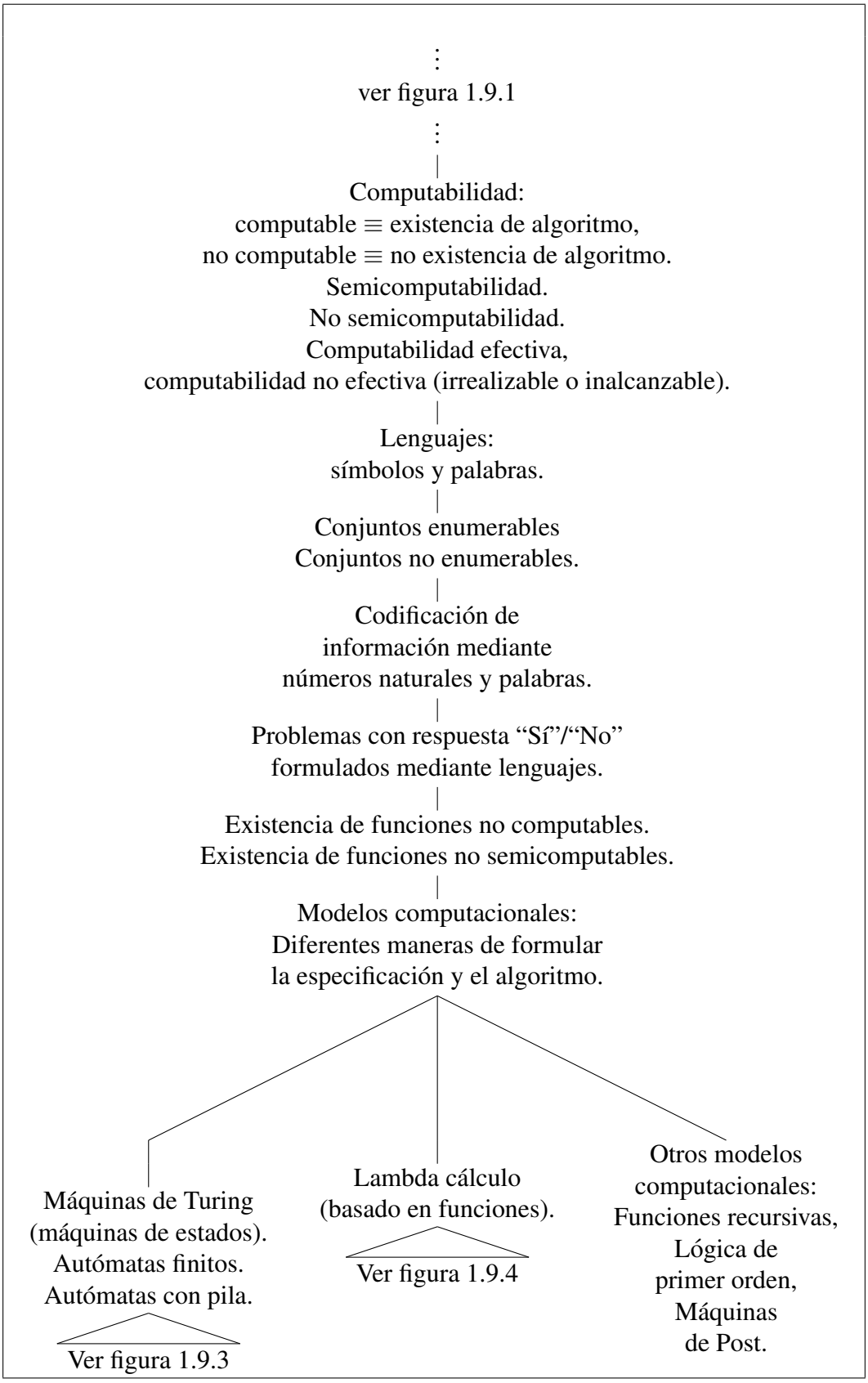


**Figura 1.9.2.** Esquema (parte 2)

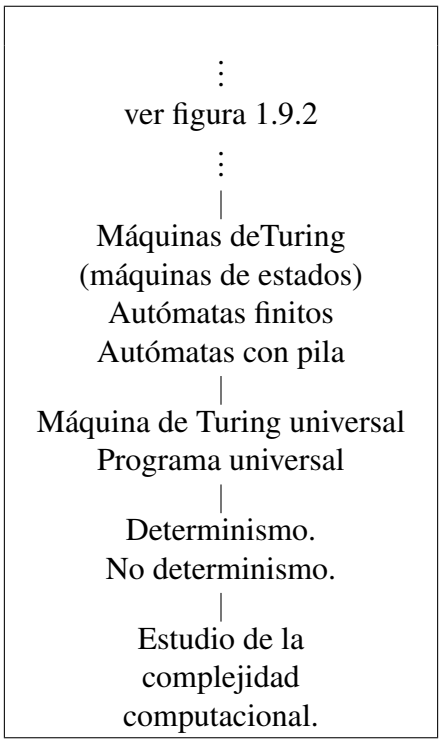


**Figura 1.9.3.** Esquema (parte 3)

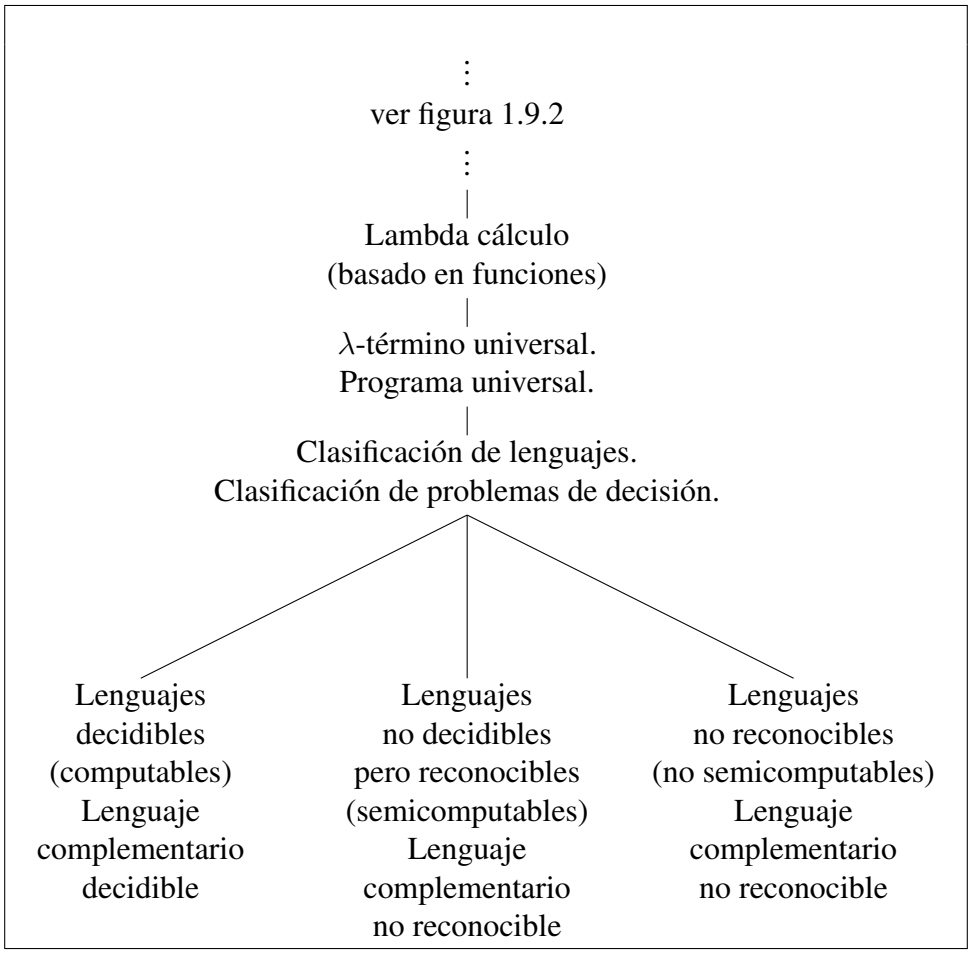


**Figura 1.9.4.** Esquema (parte 4)

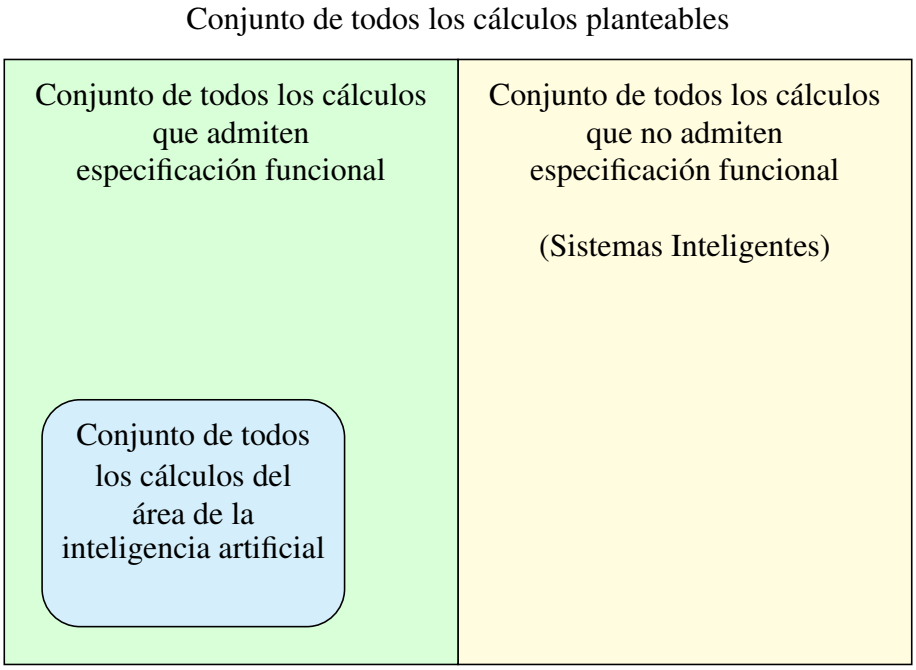


**Figura 1.9.5.** Clasificación de todos los cálculos planteables: por un lado, los cálculos que admiten especificación funcional y, por otro lado, los cálculos que no admiten especificación funcional. A los cálculos que no admiten especificación funcional se les llama Sistemas Inteligentes (Smart Systems), pero no hay que confundirlos con la Inteligencia Artificial (Artificial Intelligence). De hecho, los cálculos del área de la inteligencia artificial admiten especificación funcional.

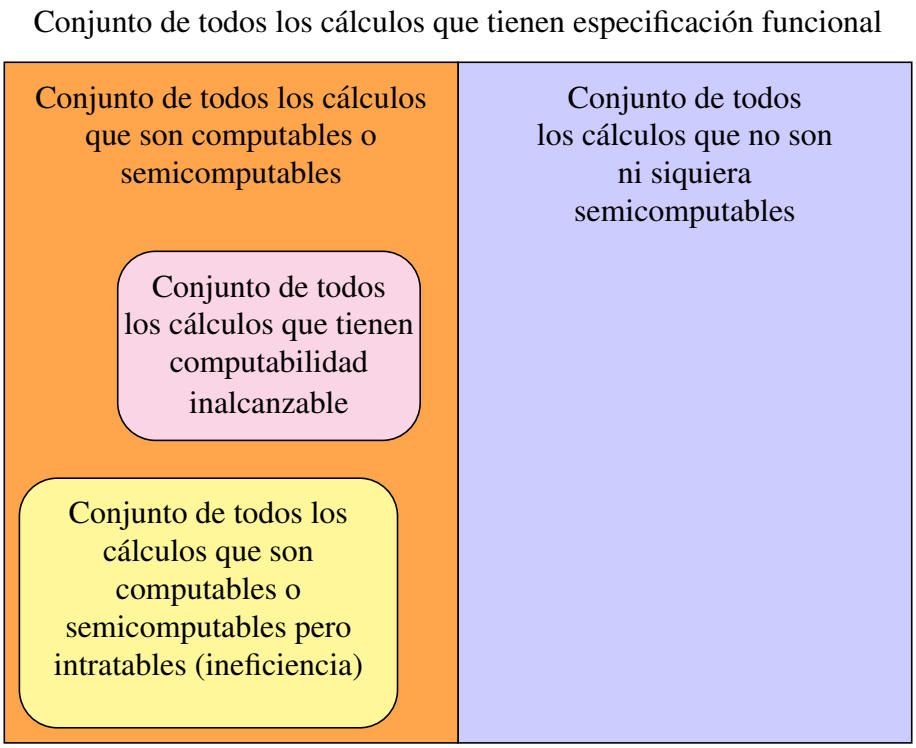


**Figura 1.9.6.** Clasificación de todos los cálculos que admiten especificación funcional: por un lado, están los cálculos que son computables y los cálculos que son semicomputables y, por otro lado, están los cálculos que no son ni siquiera semicomputables. Dentro del conjunto de los cálculos computables y semicomputables, algunos cálculos tienen computabilidad no efectiva o inalcanzable y otros cálculos son intratables porque tienen computablidad efectiva pero sus algoritmos son ineficientes.

# 1.10.

# Símbolos griegos

Es habitual utilizar símbolos del alfabeto griego para representar fórmulas de la lógica matemática y también para denominar funciones. Debido a ello, en la tabla 1.10.1 de la página 74, se muestran los símbolos griegos que se pudieran utilizar.

Alfabeto griego moderno

| | Mayúscula | minúscula | Nombre (en castellano) |
|---|---|---|---|
| 1 | $A, \mathsf{A}$ | $\alpha, \mathsf{\alpha}$ | alfa |
| 2 | $B, \mathsf{B}$ | $\beta, \mathsf{\beta}$ | beta |
| 3 | $\Gamma, \mathsf{\Gamma}$ | $\gamma, \mathsf{\gamma}$ | gamma |
| 4 | $\Delta, \mathsf{\Delta}$ | $\delta, \mathsf{\delta}$ | delta |
| 5 | $E, \mathsf{E}$ | $\epsilon, \mathsf{\epsilon}, \varepsilon, \mathsf{\varepsilon}$ | épsilon |
| 6 | $Z, \mathsf{Z}$ | $\zeta, \mathsf{\zeta}$ | dseta |
| 7 | $H, \mathsf{H}$ | $\eta, \mathsf{\eta}$ | eta |
| 8 | $\Theta, \mathsf{\Theta}$ | $\theta, \mathsf{\theta}, \vartheta$ | ceta |
| 9 | $I, \mathsf{I}$ | $\iota, \mathsf{\iota}$ | iota |
| 10 | $K, \mathsf{K}$ | $\kappa, \mathsf{\kappa}, \varkappa, \mathsf{\varkappa}$ | kappa |
| 11 | $\Lambda, \mathsf{\Lambda}$ | $\lambda, \mathsf{\lambda}$ | lambda |
| 12 | $M, \mathsf{M}$ | $\mu, \mathsf{\mu}$ | mu |
| 13 | $N, \mathsf{N}$ | $\nu, \mathsf{\nu}$ | nu |
| 14 | $\Xi, \mathsf{\Xi}$ | $\xi, \mathsf{\xi}$ | ksi |
| 15 | $O, \mathsf{O}$ | $o, \mathsf{o}$ | ómicron |
| 16 | $\Pi, \mathsf{\Pi}$ | $\pi, \mathsf{\pi}, \varpi$ | pi |
| 17 | $P, \mathsf{P}$ | $\rho, \mathsf{\rho}, \varrho, \mathsf{\varrho}$ | ro |
| 18 | $\Sigma, \mathsf{\Sigma}$ | $\sigma, \mathsf{\sigma}, \varsigma$ | sigma |
| 19 | $T, \mathsf{T}$ | $\tau, \mathsf{\tau}$ | tau |
| 20 | $\Upsilon, \mathsf{Y}$ | $\upsilon, \mathsf{\upsilon}$ | ípsilon |
| 21 | $\Phi, \mathsf{\Phi}$ | $\phi, \mathsf{\phi}, \varphi, \mathsf{\varphi}$ | fi |
| 22 | $X, \mathsf{X}$ | $\chi, \mathsf{\chi}$ | ji |
| 23 | $\Psi, \mathsf{\Psi}$ | $\psi, \mathsf{\psi}$ | psi |
| 24 | $\Omega, \mathsf{\Omega}$ | $\omega, \mathsf{\omega}$ | omega |

Letras griegas no pertenecientes al alfabeto griego moderno

| | Mayúscula | minúscula | Nombre (en castellano) |
|---|---|---|---|
| 25 | $\mathsf{F}$ | $\mathsf{F}$ | digamma |
| 26 | $\mathsf{Q}$ | $\mathsf{\varrho}$ | qoppa |
| 27 | $\mathsf{\Koppa}$ | $\mathsf{\koppa}$ | koppa |
| 28 | $\mathsf{\Sampi}$ | $\mathsf{\sampi}$ | sampi |
| 29 | $\mathsf{\Stigma}$ | $\mathsf{\stigma}$ | estigma |

**Tabla 1.10.1.** Símbolos griegos.