Capítulo 8

Indecidibilidad

Índice General

8.1. Concepto de Problema. 123
8.1.1. Introducción. 123
8.1.2. Concepto de Problema. 126
8.2. La Máquina Universal de Turing y Dos Problemas Indecidibles. . . 127
8.2.1. Codificación de Máquinas de Turing. 127
8.2.2. Ejemplo de un Lenguaje que NO es Recursivamente Enumerable. 129
8.2.3. La Máquina Universal de Turing. 131
8.2.4. Dos Problemas Indecidibles. 133
8.3. La Indecidibilidad del Problema de la Correspondencia de Post. . . 138
8.4. Problemas Propuestos. 141

8.1. Concepto de Problema.

8.1.1. Introducción.

Como se ha visto en los capítulos anteriores, una Máquina de Turing se puede estudiar como un calculador de funciones o como un reconocedor de lenguajes. Así, la clase de los lenguajes recursivos se puede identificar con la clase de las funciones recursivas totales, mientras que la clase de los lenguajes recursivamente enumerables se puede identificar con la clase de las funciones recursivas parciales.

Cualquier Máquina de Turing reconocedora de cadenas es una Máquina de Turing calculadora de la *función característica* del lenguaje que reconoce: es una función que asocia el valor 1 a las cadenas que pertenecen al lenguaje y 0 a las cadenas que no pertenecen al lenguaje. Cualquier Máquina de Turing calculadora de funciones es una Máquina de Turing reconocedora de lenguajes, ya que se puede representar cada función por el lenguaje formado por las tuplas que se pueden formar con sus parámetros de entrada y de salida.

Ejemplo:

La función suma *se podría representar como el conjunto*

{(0, 0, 0), (0, 1, 1), (0, 2, 2), .., (1, 0, 1), (1, 1, 2), .., (1, 9, 10), .., (325, 16, 341), ..}.

Este conjunto de cadenas, este lenguaje, estará formado por tripletas de números tales que el tercero representa al suma de los dos primeros.

Lo anterior pretende solamente incidir en lo que se viene repitiendo desde el primer tema: las dos visiones son completamente equivalentes, y una función total puede ser representada mediante un lenguaje recursivo y una función parcial mediante un lenguaje recursivamente enumerable.

Además, la hipótesis de Church permite identificar las funciones computables con las funciones recursivas parciales. Es decir, se pueden diseñar Máquinas de Turing para calcular las funciones recursivas parciales. O, desde el punto de vista del reconocimiento de lenguajes, se pueden diseñar Máquinas de Turing que aceptan una cadena si forma parte de un lenguaje recursivamente enumerable (de ahí que algunos autores denominen indistintamente a las funciones computables, como funciones *Turing-computables* o funciones *Turing-aceptables*).

Si una función es computable, se puede calcular su solución *cuando exista*. Y esa solución se puede calcular mediante una Máquina de Turing. Cuestión aparte es garantizar a priori qué ocurrirá con el proceso para cualquier parámetro. Está garantizado que será finito cuando la función es total, ya que se conoce su dominio de definición (para estos valores finalizará con éxito, para los demás finalizará con un error). Pero en una función parcial, el dominio de definición no está determinado. Por lo tanto, no siempre se podrá garantizar la finitud y corrección del proceso, sólo para ciertos valores de los parámetros. Como un ejemplo, se considera el siguiente algoritmo:

```
boole maravilloso(int n){
/* pre: n=N, entero positivo */

   boole lo_es;

   if (n==1)
      lo_es=cierto;
   else
      if (n%2==0)
         lo_es=maravilloso(n/2);
      else
         lo_es=maravilloso(3*n+1);

   return lo_es;
}
/* post: cierto, si n es maravilloso, si no lo es... :-( */
```

Este algoritmo se ha construido siguiendo la definición de Número Maravilloso: *“Un número n es maravilloso si es el 1, o puede alcanzarse el 1 a través del siguiente proceso: si es par, se considera el valor de n/2; si es impar, el valor de 3*n+1”*. Se *conjetura* que todos los números enteros son maravillosos, pero no ha podido demostrarse. De ahí que el Dominio de Definición de esta algoritmo no esté determinado; y, por lo tanto, se esté hablando de una función parcial, en la que no es posible garantizar que el proceso finalice para *cualquier* valor entero que se considere.

Si una función es total, existe al menos una Máquina de Turing que siempre se detiene, bien dando el resultado del cálculo, bien indicando la existencia de un error. Ese error sólo se dará si *los parámetros no pertenecen al dominio de definición* de la función. La situación equivalente, desde el punto de vista del reconocimiento de un lenguaje, es *la cadena no pertenece al lenguaje*.

La MT calcula funciones:	La MT reconoce lenguajes:
Si la función es total	Si el lenguaje es recursivo
la MT siempre para.	la MT siempre para.
Devuelve el resultado si los parámetros $\in$ Dominio Definición.	Acepta la cadena si la cadena $\in$ L(M).
Devuelve error si los parámetros $\notin$ Dominio Definición.	Rechaza la cadena si la cadena $\notin$ L(M).

Si una función es parcial existirá una Máquina de Turing de la que sólo se asegura que parará devolviendo el resultado del cálculo cuando *los parámetros pertenecen al dominio de definición* de la función. No se puede garantizar qué ocurre en caso contrario. De nuevo, se establece un paralelismo evidente entre este comportamiento y lo que ocurre desde el punto de vista de reconocimiento del lenguaje,

La MT calcula funciones:	La MT reconoce lenguajes:
Si la función es parcial	Si el lenguaje es rec. enumerable
la MT siempre para devolviendo el resultado si los parámetros $\in$ Dominio Definición.	la MT siempre para aceptando la cadena si la cadena $\in$ L(M).
Pero el comportamiento está indefinido si los parámetros $\notin$ Dominio Definición.	Pero el comportamiento está indefinido si la cadena $\notin$ L(M).

Por lo tanto, ¿cuál es el interés en determinar si un lenguaje es o no es recursivo?. En este tema, mediante el *concepto de problema* y la definición de la *Máquina Universal de Turing*, el objetivo será estudiar los mecanismos que permiten establecer cuáles son los problemas en los que se puede garantizar la existencia de una computación finita por medio de un computador. En este tema se demostrará que existen problemas para los cuales esa computación finita no existe. Por lo tanto, se demostrará que la computación tiene límites.

8.1.2. Concepto de Problema.

Definición 8.1 (Problema) *Un Problema es un enunciado cierto o falso dependiendo de los valores de los parámetros que aparecen en su definición.*

Definición 8.2 (Solución) *Una Solución a un Problema es una aplicación entre el conjunto de instancias de los parámetros del problema y el conjunto {cierto, falso}.*

Definición 8.3 (Algoritmo) *Un Algoritmo es un conjunto de pasos cuyo objetivo es resolver un problema.*

Es posible identificar un algoritmo con una función,

$$f : A_1 \times A_2 \times \ldots \times A_n \longrightarrow A,$$

de forma que un algoritmo obtiene un valor de salida a partir de unos valores de entrada si ese valor de salida existe, es decir, si hay solución al problema. Mientras que la solución a un problema se asimilaría al establecimiento de una aplicación

$$P(f) : A_1 \times A_2 \times \ldots \times A_n \times A \longrightarrow \{cierto, falso\}.$$

Sólo si se puede establecer esa aplicación entre parámetros y el conjunto {cierto, falso} hay una solución del problema; y sólo si esa aplicación es una *función total* existe la seguridad de establecer el algoritmo: para cualquier instancia se puede *decidir* si el enunciado es cierto o falso. Y esa decisión se puede obtener aplicando el algoritmo, de forma que se cumple la relación,

$$\mathbf{P(f)(a_1, a_2, \ldots, a_n, a) = cierto \Leftrightarrow f(a_1, a_2, \ldots, a_n) = a}.$$

Por ejemplo,

$$f : \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{N},\ f(x,y) = x + y$$

P(f)(3,5,7)=falso, P(f)(3,5,8)=cierto, ...

En este problema siempre es posible decidir, por lo tanto, se sabe que siempre será posible establecer la solución y se puede estar seguro de que habrá un algoritmo que permita solucionar el problema.

En aquellos problemas para los que exista algún valor en el conjunto de instancias para el que la aplicación no esté definida, NO se puede DECIDIR siempre: no se puede asegurar cuál será el comportamiento ante dichas instancias. Son problemas en los que la aplicación entre el conjunto de parámetros y el conjunto {*cierto, falso*} es una función parcial. Por lo tanto, se establece la relación,

PROBLEMA DECIDIBLE $\equiv$ **LENGUAJE RECURSIVO**.

Así, estudiando si el lenguaje asociado a un determinado problema es o no es recursivo, se puede saber si el problema es o no es decidible y si se puede o no garantizar la existencia de un algoritmo de ejecución finita. Es decir, se asocia la existencia de un *algoritmo* a la existencia de una *Máquina de Turing que siempre pare*, diciendo SI o NO. Y se usarán indistintamente ambos términos:

Existe un algoritmo $\equiv$ Existe una MT que siempre para.

Resulta una práctica habitual que los problemas se enuncien como *problemas de decisión*, de respuesta SI o NO y en los que, por lo tanto, resulta fácil transformar el enunciado en la cuestión de si un lenguaje (el asociado al enunciado) es o no recursivo y, por lo tanto, si existe o no un algoritmo que permita resolverlos. Centrarse en este tipo de problemas no supone restringir el campo de estudio, ya que normalmente cualquier problema se puede plantear como un problema de decisión y viceversa.

El objetivo de este tema será, por lo tanto, establecer resultados que permitan afirmar si un problema es o no *decidible*. O, lo que es lo mismo, si el lenguaje asociado a dicho problema es o no *recursivo*.

8.2. La Máquina Universal de Turing y Dos Problemas Indecidibles.

8.2.1. Codificación de Máquinas de Turing.

Para estudiar la codificación de una Máquina de Turing, se recuerda el siguiente resultado, que ya se comentó en el tema 7:

Teorema 8.1 *Si* $L \in (0+1)^*$ *es aceptado por alguna Máquina de Turing* $\Rightarrow$ *L es aceptado por una Máquina de Turing con alfabeto restringido a* {*0, 1,* B}.

Este teorema sostiene que cualquier lenguaje sobre el alfabeto {0,1} se puede reconocer en una Máquina de Turing cuyo alfabeto de cinta sea {0,1,B}; por lo tanto, sólo son necesarios tres símbolos en dicha Máquina de Turing.

Por otro lado, también es posible afirmar que no hay necesidad de más de un estado final en cualquier Máquina de Turing.

Con estas premisas, se propone la siguiente codificación para cualquier Máquina de Turing con alfabeto restringido: Sea

$$M = \langle \{0,1\},\ Q,\ \{0,1,\mathtt{B}\},\ f,\ q_1,\ \mathtt{B},\ \{q_2\} \rangle$$

una Máquina de Turing con alfabeto de entrada {0,1} y el B como único símbolo adicional en el alfabeto de la cinta. Se asume que Q = {q_1, q_2, q_3, .., q_n} es el conjunto de estados y que sólo hay un estado final q_2 (y, por supuesto, q_1 es el estado inicial).

Se renombra el alfabeto:

Símbolo					Codificación
0	$\rightarrow$	x_1	(Primer símbolo)	$\rightarrow$	$0^1 = 0$
1	$\rightarrow$	x_2	(Segundo símbolo)	$\rightarrow$	$0^2 = 00$
B	$\rightarrow$	x_3	(Tercer símbolo)	$\rightarrow$	$0^3 = 000$

Se renombran las direcciones:

Dirección					Codificación
L	$\rightarrow$	D_1	(Primera dirección)	$\rightarrow$	$0^1 = 0$
R	$\rightarrow$	D_2	(Segunda dirección)	$\rightarrow$	$0^2 = 00$

Y, si cada identificador de estado q_i se representa como 0^i, entonces se puede codificar cada una de las transiciones:

$$f(q_i, x_j) = (q_k, x_l, D_m) \hookrightarrow 0^i 1 0^j 1 0^k 1 0^l 1 0^m$$

A cada una de estas transiciones así codificadas se le denominará *código* y se le asigna un orden, por lo que se puede codificar una Máquina de Turing M como

$$111 código_1 11 código_2 11 código_3 11 \ldots 11 código_r 111$$

Los códigos siempre estarán entre parejas de 11. El orden de los códigos es irrelevante; una misma Máquina de Turing puede tener distintas codificaciones[1]. Pero cada codificación sólo puede estar asociada a una Máquina de Turing.

Por ejemplo: Sea la Máquina de Turing

$$M = \langle \{0,1\},\ \{q_1, q_2, q_3\},\ \{0,1,B\},\ f,\ q_1,\ \mathtt{B},\ \{q_2\} \rangle$$

[1] Ya que una misma MT puede tener distintas codificaciones según el orden en el que se representen los códigos, cuando se utilice la notación ⟨M⟩ para representar un código de MT, lo que realmente se representará es el conjunto de cadenas que codifican a la MT M.

con

$$\begin{array}{l} \mathtt{f}(\mathtt{q}_1,\mathtt{1}) = (\mathtt{q}_3,\mathtt{0},\mathtt{R}) \\ \mathtt{f}(\mathtt{q}_3,\mathtt{0}) = (\mathtt{q}_1,\mathtt{1},\mathtt{R}) \\ \mathtt{f}(\mathtt{q}_3,\mathtt{1}) = (\mathtt{q}_2,\mathtt{0},\mathtt{R}) \\ \mathtt{f}(\mathtt{q}_3,\mathtt{B}) = (\mathtt{q}_3,\mathtt{1},\mathtt{L}) \end{array}$$

Una posible codificación de M es la cadena:

```
111010010001010011000101010010011000100100101001100010001
00010010111
```

Con esta codificación, *una Máquina de Turing es un número en binario*. Por lo tanto, *cualquier número en binario* podrá ser considerado, inicialmente, un *código de Máquina de Turing*. Evidentemente, habrá cadenas binarias que representen Máquina de Turing y cadenas binarias que no representen ninguna Máquina de Turing: las cadenas que no comiencen por 111, o no acaben en 111 o que no tengan parejas de 1's separando 5 bloques de 0's que, a su vez, están separados por 1, no codifican Máquinas de Turing.

Se denota $\langle$M,$w\rangle$ a la cadena formada al concatenar la codificación de M con la cadena w. Se interpreta que la cadena w sería la cadena de entrada a la Máquina de Turing M.

En el ejemplo anterior, la cadena $\langle$*M,* `1011`$\rangle$ *se codifica como:*

```
111010010001010011000101010010011000100100101001100010001
000100101111011
```

El establecimiento de esta codificación es básico para la definición de la *Máquina Universal de Turing*, como aquella Máquina de Turing cuyas cadenas de entrada son de la forma $\langle$M,$w\rangle$ y cuyo comportamiento consiste en la simulación del comportamiento de la máquina M cuando su entrada es la cadena w.

8.2.2. Ejemplo de un Lenguaje que NO es Recursivamente Enumerable.

A continuación se presenta un lenguaje que no es recursivamente enumerable; su interés consiste en poder utilizarlo como una herramienta para caracterizar el carácter recursivo o recursivamente enumerable de otros lenguajes, utilizando las relaciones vistas en la sección 7.4.

Es un lenguaje construido con el propósito de disponer de un lenguaje no recursivamente enumerable.

Se construye mediante una tabla. La numeración de las filas sirve para representar la cadenas de $(0 + 1)^*$ en orden canónico: la primera fila, representa la primera cadena; la segunda fila, a la segunda cadena en orden canónico, etc. Las columnas se numeran sucesivamente a partir del `1` y cada índice de columna se interpreta en binario, de forma que

dichos números en binario se interpretan como codificaciones de Máquina de Turing (*j* en binario es la Máquina de Turing M_j). Se construye la siguiente tabla infinita[2],

	1	2	3	4	...
1	**0**	1	1	0	...
2	0	**0**	1	1	...
3	1	1	**1**	0	...
4	1	0	0	**1**	...
...	...	...	...	...	...

en la que cada entrada es 1 ó 0 de acuerdo al siguiente convenio: si $w_i \in L(M_j)$, entonces el elemento *(i,j)* es 1, si no es 0,

$$(i,j) = 1 \Leftrightarrow w_i \in L(M_j),$$
$$(i,j) = 0 \Leftrightarrow w_i \notin L(M_j).$$

Se construye el lenguaje $\mathcal{L}_d$, el *lenguaje diagonal*, formado por los elementos diagonales nulos; es decir, en $\mathcal{L}_d$ está la cadena w_i si *(i,i)*=0; por lo tanto, M_i NO acepta la cadena w_i:

$$\mathcal{L}_d = \{w_i \mid M_i \text{ NO acepta } w_i\} = \{w_i \mid w_i \notin L(M_i)\}$$

Lema 8.1 *El lenguaje $\mathcal{L}_d$ NO es recursivamente enumerable.*

Demostración:

Supóngase, como hipótesis de partida, que existe alguna Máquina de Turing, M_j que acepte el lenguaje $\mathcal{L}_d$, es decir, $\mathcal{L}_d = L(M_j)$. Sea w_j la cadena que ocupa la *j-ésima* posición en orden canónico y el valor *j* codificado en binario, el código de M_j.

Si $\mathcal{L}_d = L(M_j)$ se debe cumplir que si $w_j \in \mathcal{L}_d$, entonces $w_j \in L(M_j)$. Y esto es imposible:

$$w_j \in \mathcal{L}_d \Rightarrow (j,j) = 0 \Rightarrow w_j \notin L(M_j),$$
$$w_j \notin \mathcal{L}_d \Rightarrow (j,j) = 1 \Rightarrow w_j \in L(M_j).$$

Por lo tanto, $\mathcal{L}_d \neq L(M_j)$. La hipótesis es falsa y no existe ninguna Máquina de Turing que acepte $\mathcal{L}_d$.

c.q.d

[2]No todas las codificaciones de las columnas tendrán sentido como códigos de MT; de hecho, el ejemplo que muestra cómo se llenaría es ficticio, puesto que todas esas entradas deberían ser ceros, ya que tanto 1, como 10, 11 ó 100 no son códigos válidos de MT.

8.2.3. La Máquina Universal de Turing.

Definición 8.4 (Lenguaje Universal, $\mathcal{L}_U$) *Se denomina Lenguaje Universal al conjunto de cadenas*

$$\mathcal{L}_U = \{\langle M,w \rangle \mid M \text{ acepta } w\}\ .$$

En esta definición se asume que M admite la codificación anteriormente descrita, dado que si su alfabeto no está restringido al alfabeto $\{0,1,\mathtt{B}\}$ siempre se podrá encontrar la máquina equivalente con alfabeto restringido.

Teorema 8.2 $\mathcal{L}_U$ *es un lenguaje recursivamente enumerable.*

Demostración:

La demostración se realiza mediante la construcción de una Máquina de Turing, M1, que acepta $\mathcal{L}_U$. Puesto que para determinar si la cadena $\langle \text{M},w \rangle \in \mathcal{L}_U$ hay que determinar si la máquina M acepta la cadena w, se construirá para permitir la simulación del comportamiento de M con w.

Dicha máquina tendrá tres cintas. La primera cinta es de *entrada*; en ella irá la cadena $\langle \text{M},w \rangle$ y, puesto que en ella estará la codificación de M, su papel en la simulación de su comportamiento con la cadena w sería similar al de la memoria de un ordenador. Todas las transiciones (todos los códigos, entre pares de 11) están en el primer bloque (entre los dos 111).

La segunda cinta de M1, simulará a la cinta de entrada de M y, por lo tanto, es la que se realiza realmente la simulación del comportamiento de M. En ella se copiará la cadena w.

La tercera cinta de M1, servirá para llevar cuenta del estado en que estaría M; para ello, q_i se codificará como 0^i. Su papel en la simulación es similar al del contador de programa de un ordenador.

El comportamiento de M1 es el siguiente:

1. Se comprueba que la cadena que está en la cinta de entrada es correcta, es decir, que el formato es adecuado (bloques 111 y bloques 11) y que no hay dos códigos distintos que comiencen con 0^i10^j para un mismo par *i, j*. También se comprueba que en cada código de la forma $0^i10^j10^k10^l10^m$, $1 \leq j \leq 3$, $1 \leq l \leq 3$ y $1 \leq m \leq 2$ (para ello se pueden usar las otras cintas como auxiliares si hiciera falta).
2. Se inicializa la cinta 2 con la cadena w. Se inicializa la cinta 3 con el valor 0 que codifica a q_1. Los 3 cabezales de lectura/escritura se colocan en el símbolo situado más a la izquierda en las 3 cintas.

3. **Repetir el siguiente proceso**,

   Sea x_j el símbolo que se está leyendo en la cinta 2 y sea 0^i el estado representado en la cinta 3. Se recorre la cinta 1 de izquierda a derecha (parando en el segundo 111) buscando una subcadena (entre 11's) que comience por $0^i10^j\ldots$; si no se encuentra M1 para y rechaza ya que M no tiene transición asociada. Si el código se encuentra, por ejemplo $0^i10^j10^k10^l10^m$, se escribe 0^k en la cinta 3, se escribe x_l en la celda bajo el cabezal en la cinta 2 y en esa cinta el cabezal se desplaza en la dirección D^m.

   hasta que en la cinta 3 se llega al código 00 **o** M1 para y rechaza.

Si en la cinta 3 se llega al código 00, quiere decir que M trabajando con la cadena w llega a q_2, estado final; en este caso la máquina M1 para y acepta la cadena $\langle$M,$w\rangle$. Por lo tanto, M1 acepta $\langle$M,$w\rangle$ si y sólo si M acepta la cadena w. También se cumple que si M para sin aceptar w, entonces M1 para sin aceptar $\langle$M,$w\rangle$ y que si M no para con w, entonces M1 tampoco para con $\langle$M,$w\rangle$. De la existencia de M1 se sigue que $\mathcal{L}_U$ es un Lenguaje Recursivamente Enumerable.

c.q.d

Aplicando resultados anteriores, se puede construir una máquina equivalente a M1, que tuviera una sola cinta, limitada a la izquierda y el alfabeto restringido a {0, 1, B}. Esta máquina es la definición formal de la Máquina Universal de Turing,

Definición 8.5 (Máquina Universal de Turing, $\mathcal{M}_U$) *Máquina de Turing con una sola cinta, limitada a la izquierda, y con alfabeto {0, 1,* B*} que acepta el lenguaje* $\mathcal{L}_U$.

La culminación del modelo de computación establecido por Alan Turing con esta Máquina Universal ha tenido, y sigue teniendo, mucha más trascendencia del resultado puramente lógico.

Este resultado establece que es posible *enumerar* todas las cadenas que representan computaciones válidas. Desde el punto de vista lógico permitió asegurar que tiene que haber funciones *no computables*, tal y como se estableció en la introducción del tema 7.

Pero, además, cuando Von Neumann estableció la arquitectura que hoy se conoce con su nombre, lo hizo influenciado por el resultado de Turing: hasta que Von Neumann introdujo el concepto de *control por programa almacenado* los computadores se "programaban" (si así se puede denominar) reconfigurando completamente el *hardware* para cada cálculo a realizar. Este modelo inspiró a Von Neumann la idea de que sería más factible disponer de una circuitería simple (un autómata) gobernado por un conjunto de instrucciones, más o menos complejas, y fue, en definitiva, el punto de arranque del desarrollo de la programación.

Se ha comentado que la Máquina de Turing se estudia como modelo formal de algoritmo; se puede establecer la analogía siguiente:

> *Una máquina de Turing procesando una cadena de entrada, es un modelo de un algoritmo procesando sus datos de entrada.*

El concepto de Máquina Universal de Turing permite establecer una analogía que resulta más familiar todavía,

> *Una Máquina Universal de Turing recibe como cadena de entrada el código de una Máquina de Turing y la cadena con la que esta trabajaría; un computador de propósito general recibe como cadena de entrada el código de un programa y los datos con los que el programa trabaja.*

8.2.4. Dos Problemas Indecidibles.

La existencia del lenguaje $\mathcal{L}_d$ permite establecer que existe, al menos, un lenguaje que no pertenece a la clase de los lenguajes recursivamente enumerables.

Es, además, un lenguaje que permitirá establecer resultados sobre la indecidibilidad. No existe una herramienta como el Lema de Bombeo para lenguajes recursivos y recursivamente enumerables. Por lo tanto, para poder estudiar que lenguajes pertenecen o no a dichas clases, sólo se dispone, como herramientas de trabajo, de las propiedades de clausura y de la reducción de problemas.

Si se puede diseñar una Máquina de Turing que acepte las cadenas de un lenguaje, se demuestra que éste es computable. Si, además, la construcción garantiza la parada, el lenguaje es decidible. Y, entonces, existe un algoritmo para resolver el problema asociado.

Ya se ha establecido que $\mathcal{L}_U$ es un lenguaje computable; si, además, fuera decidible, existiría un algoritmo de algoritmos, la posibilidad de determinar automáticamente el éxito o el fracaso de una computación. Algo parecido al teorema de los teoremas que buscaba Hilbert.

Para estudiar si $\mathcal{L}_U$ es o no un lenguaje recursivo, se utiliza como herramienta el lenguaje $\mathcal{L}_d$. En el lema 8.1, quedó establecido que $\mathcal{L}_d$ es un lenguaje no recursivamente enumerable. Por lo tanto, $\mathcal{L}_d$ tampoco es un lenguaje recursivo y, por el teorema 7.5, su complementario, $\bar{\mathcal{L}}_d$,

$$\bar{\mathcal{L}}_d = \{w_i \mid M_i \text{ acepta } w_i\} = \{w_i \mid w_i \in L(M_i)\},$$

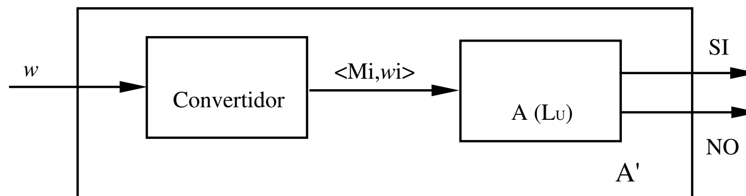
tampoco puede ser un lenguaje recursivo. Este resultado permitirá demostrar que $\mathcal{L}_U$ es un lenguaje recursivamente enumerable no recursivo, puesto que es posible hacer una transformación del lenguaje $\bar{\mathcal{L}}_d$ al lenguaje $\mathcal{L}_U$.

Teorema 8.3 *$\mathcal{L}_U$ es un lenguaje recursivamente enumerable no recursivo.*

Demostración:

La demostración es por contradicción: se parte de la suposición de que existe un algoritmo A (una Máquina de Turing que siempre para) para reconocer $\mathcal{L}_U$. Si esto fuera cierto, entonces se podría establecer el siguiente procedimiento para reconocer $\bar{\mathcal{L}}_d$:

Se transforman las cadenas de $\bar{\mathcal{L}}_d$ en cadenas de $\mathcal{L}_U$ utilizando el *Convertidor de Cadenas*. El convertidor, dada una cadena $w \in (0+1)^*$, determina el valor de *i* tal que $w = w_i$, la *i-ésima* cadena en orden canónico. Este valor *i*, expresado en binario, es el código de alguna Máquina de Turing, M_i. La salida del convertidor es la cadena $\langle \mathrm{M}_i, w_i \rangle$ que se suministra al algoritmo A. Así se consigue una Máquina de Turing que acepta w si y sólo si M_i acepta w_i.



Es decir, se ha construido un algoritmo A$'$ que indica si la cadena w pertenece o no al lenguaje $\bar{\mathcal{L}}_d$. Puesto que esto es imposible, ya que $\bar{\mathcal{L}}_d$ no es recursivo, entonces la suposición de que existe el algoritmo A debe ser falsa.

Por lo tanto, $\mathcal{L}_U$ es un lenguaje recursivamente enumerable no recursivo.

c.q.d

Es decir, el problema

"Dada una Máquina de Turing M y una cadena w, ¿acepta M la cadena w?"

es un problema *indecidible*.

El Problema de la Parada.

Un problema tan importante como el anterior y también indecidible es el *Problema de la Parada*:

"Sea M una Máquina de Turing y una cadena w. ¿Para M con entrada w?".

Es mucha la importancia de este problema: si se pudiera predecir la parada de M, si se pudiera predecir *de forma automática* la parada de cualquier proceso en ejecución, también se podría decidir automáticamente el éxito o el fracaso de esa ejecución.

Supóngase que fuera posible predecir la parada de un proceso; es decir, que se dispone de una función llamada `Halts`, tal que

```
int Halts(char *P, char *I) {
/* Pre: P e I son cadenas de caracteres, siendo P */
/* el código fuente de un programa e I los datos */

   /* (1) se determina si P es un programa correcto */
                    /* sintácticamente (compilación)*/
   /* (2) se determina si P finaliza su ejecución */
               /* cuando lee la cadena de entrada I */

   return halt;
}
/* Post: devuelve 1 si P para con I, 0 en caso contrario */
```

Sabiendo esto, se escribe el programa siguiente:

```
int main() {
   char I[100000000];
   /* hacer I tan grande como se quiera, o usar malloc */

   read_a_C_program_into(I);

   if (Halts(I,I)) {
       while(1){}    /* bucle infinito */
   }
   else
       return 1;
}
```

Este programa se almacena en el fichero `Diagonal.c`. Una vez compilado y montado, se obtiene el código ejecutable, `Diagonal`. A continuación se ejecuta,

`Diagonal<Diagonal.c`

Sólo hay dos casos posibles al ejecutar esta orden, y son mutuamente excluyentes:

Caso 1: `Halts(I,I)` devuelve 1.

Según la definición de `Halts`, esto significa que `Diagonal.c` finaliza su ejecución cuando recibe como entrada `Diagonal.c`. Pero, según la definición de `Diagonal.c`,

que `Halts(I,I)` devuelva `1`, significa que en el condicional se ejecutará la rama "if", que contiene el bucle infinito; es decir, la ejecución de `Diagonal.c` no finaliza NUNCA.

Se llega a una contradicción.

Caso 2: `Halts(I,I)` devuelve 0.

Según la definición de `Halts`, esto significa que `Diagonal.c` NUNCA finaliza su ejecución cuando recibe como entrada `Diagonal.c`. PERO, según la definición de `Diagonal.c`, que `Halts(I,I)` devuelva `0`, significa que en el condicional se ejecutará la rama "else", por lo que finaliza la ejecución de `Diagonal.c`.

Se llega, de nuevo, a una contradicción.

Puesto que no hay más casos posibles la suposición inicial debe ser falsa: no es posible que exista la función `Halts`.

La demostración formal de que el problema de la parada es un problema indecidible, se basa en que el lenguaje asociado,

$$\mathcal{L}_H = \{\langle \text{M}, w\rangle \mid \text{M para con } w\} ,$$

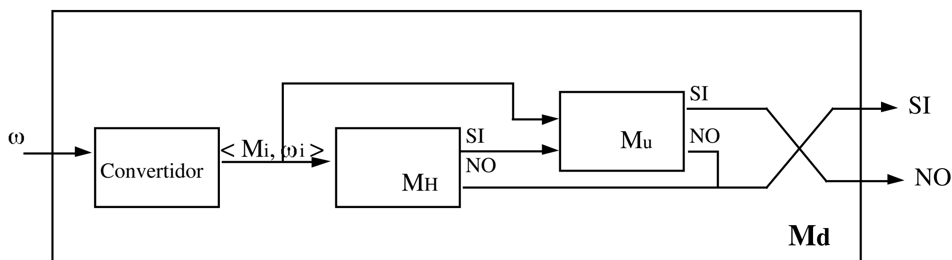
es un lenguaje recursivamente enumerable (modificando el comportamiento de $\mathcal{M}_U$, por ejemplo, se obtiene una Máquina de Turing que indica si M para con la cadena w), pero no es recursivo.

Lema 8.2 *$\mathcal{L}_H$ es un lenguaje recursivamente enumerable no recursivo.*

Demostración:

Supóngase que el problema de parada es decidible; por lo tanto, la Máquina de Turing que reconoce el lenguaje $\mathcal{L}_H$, $\mathcal{M}_H$, ante una entrada $\langle$M,$w\rangle$ respondería SI o NO dependiendo de si M para o no para al trabajar sobre w.

Si esto fuera cierto, se podría construir el siguiente algoritmo para reconocer $\mathcal{L}_d$:

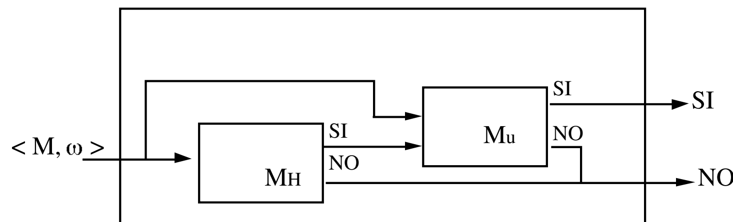


Esta Máquina de Turing, a la que se llamaría $\mathcal{M}_d$, determinaría si una cadena w_i es aceptada por M$_i$, es decir, si la cadena w pertenece o no al lenguaje $\mathcal{L}_d$. Puesto que es

imposible, $\mathcal{M}_H$ no puede existir tal y como se ha descrito y $\mathcal{L}_H$ no puede ser un lenguaje recursivo.

c.q.d

En la construcción anterior, nótese que si se sabe que M_i para al trabajar sobre w_i, entonces seguro que al suministrar esa información a $\mathcal{M}_U$, su respuesta será del tipo SI/NO.



Por lo tanto, si $\mathcal{L}_H$ fuera un lenguaje recursivo, entonces también $\mathcal{L}_U$ sería un lenguaje recursivo. Y existiría la posibilidad de desarrollar herramientas automáticas que permitieran comprobar si un algoritmo es o no es correcto. Ello imposibilita que se pueda disponer de un método automático de "análisis semántico[3]", similar a las herramientas de "análisis léxico" o de "análisis sintáctico".

En el libro de Douglas R. Hofstadter, "Gödel, Escher , Bach. Un eterno y grácil bucle", se hace la siguiente reflexión sobre las dificultades de tal análisis semántico: el autor comienza la reflexión sobre la diferencia entre apreciar los aspectos sintácticos de una forma (reconocer una fórmula bien construida, apreciar las líneas y colores de una pintura o ser capaces de leer una partitura de música) y los aspectos semánticos asociados a dicha forma (reconocer la verdad o falsedad de una fórmula o los sentimientos que puedan inspirar una pintura o una determinada pieza de música):

> *"Subjetivamente, se percibe que los mecanismos de extracción de la significación interior carecen por completo de parentesco con los procedimientos de decisión que verifican la presencia o ausencia de una cualidad específica, como, por ejemplo, el carácter de una fórmula bien formada. Tal vez sea por ello que la significación interior sea algo que descubre más cosas de sí misma a medida que el tiempo pasa. A este respecto jamás se puede estar seguro, de la misma forma en que sí es posible estarlo a propósito de lo bien formado, de que uno ha finiquitado el tema.*
>
> *Esto nos propone la posibilidad de trazar una distinción entre los dos tipos sentidos de "forma" de las pautas que hemos comentado. Primero, tenemos cualidades [. . .] que pueden ser aprehendidas mediantes* verificaciones predictiblemente finalizables. *Propongo llamar a estas las* cualidades sintácticas

[3]No debe confundirse el concepto que aquí se pretende describir con la fase habitualmente denominada de análisis semántico en la compilación; el concepto que se pretende desarrollar aquí es el "entender el significado de".

de la forma. [. . .] los aspectos semánticos *de la forma son aquellos que no pueden ser verificados dentro de un lapso predictible: requieren* verificaciones de finalización imprevisible *[. . .]. Por lo tanto, las propiedades "semánticas" están vinculadas con búsquedas no finalizables ya que, en un sentido importante, la significación de un objeto no está situada en el objeto mismo. Esto no equivale a sostener que no es posible captar la significación de ningún objeto [. . .]. Con todo siempre quedan aspectos de aquella que siguen ocultos durante lapsos no previsibles. [. . .]*

Así, otra manera de caracterizar la diferencia entre propiedades "sintácticas" y "semánticas", es la observación de que las primeras residen en el objeto bajo examen, en tanto que las segundas dependen de sus relaciones con una clase potencialmente infinita de otros objetos."

8.3. La Indecidibilidad del Problema de la Correspondencia de Post.

Los Sistemas de Correspondencia de Post fueron formulados por Emil Post en 1931, la misma época en la que Turing formuló su modelo de computación; la idea básica era la misma en ambos modelos formales y, de hecho, está demostrado que son equivalentes en cuanto a poder computacional.

Al margen de su propio interés como modelo de computación, este problema sirve de conexión entre los resultados sobre Indecidibilidad en el Problema de Aceptación, "*¿M acepta w?*" (y en el Problema de la Parada, "*¿M para con w?*"), con cuestiones indecidibles en el ámbito de las gramáticas y lenguajes de contexto libre.

Definición 8.6 (Sistemas de Correspondencia de Post) *Una instancia del Problema de la Correspondencia de Post (PCP), se denomina un Sistema de Correspondencia de Post (SCP) y consta de tres elementos: un alfabeto Σ y dos conjuntos A y B de cadenas de Σ^+, tales que A y B contienen el mismo número de cadenas. Si*

$$A = \{u_1, u_2, \ldots, u_k\} \text{ y } B = \{v_1, v_2, \ldots, v_k\},$$

una solución para esta instancia del PCP es una secuencia de índices $i_1, i_2, \ldots, i_n$ tal que

$$u_{i_1} u_{i_2} \ldots u_{i_n} = v_{i_1} v_{i_2} \ldots v_{i_n}.$$

Por ejemplo:

A = {a, abaaa, ab},

B = {aaa, ab, b}

Solución: i_1 = *2,* $i_2 = i_3$ = *1,* i_4 = *3,* $abaaa|a|a|ab = ab|aaa|aaa|b$.

Normalmente se obtienen una visión más clara del problema si se ve como una colección de bloques

$$\boxed{\frac{u_1}{v_1}}\ \boxed{\frac{u_2}{v_2}} \dots \boxed{\frac{u_k}{v_k}}$$

y se busca una secuencia de bloques tal que la cadena superior es igual a la inferior.

El ejemplo anterior puede verse, entonces, como:

$$\boxed{\frac{a}{aaa}}\ ,\ \boxed{\frac{abaaa}{ab}}\ ,\ \boxed{\frac{ab}{b}}$$

y la solución la secuencia

$$\boxed{\frac{abaaa}{ab}}\ \boxed{\frac{a}{aaa}}\ \boxed{\frac{a}{aaa}}\ \boxed{\frac{ab}{b}}$$

siendo iguales la cadena superior y la inferior.

El PCP consiste en el problema de determinar si un SCP arbitrario tiene o no una solución. El PCP es un problema INDECIDIBLE. Se puede demostrar a través del PCPM, Problema de la Correspondencia de Post Modificado, en el cual la secuencia de índices debe ser

$$1, i_2, \dots, i_n \text{ tal que } u_1 u_{i_2} \dots u_{i_n} = v_1 v_{i_2} \dots v_{i_n}.$$

El siguiente resultado establece la conexión entre el PCP y el PCPM:

Lema 8.3 *Si el PCP fuese DECIDIBLE, lo sería también el PCPM.*

La idea básica para demostrar este lema sería similar al siguiente razonamiento: si se conoce una solución del PCP, con "cambiar dos bloques de sitio" en la instancia se obtiene una solución del PCPM. Si este lema es cierto, también lo es su contrarrecíproco:

Lema 8.4 *Si el PCPM es INDECIDIBLE, entonces también lo es el PCP.*

Según esto, para establecer la indecidibilidad del PCP, basta con establecer la indecidibilidad del PCPM. Esto se puede hacer por reducción del Problema de la Aceptación, "*¿M acepta w?*", al PCPM. Puesto que este problema es indecidible, entonces el PCPM también lo es.

Teorema 8.4 *El PCPM es INDECIDIBLE.*

La demostración de este teorema consiste en, primero, establecer la reducción entre el PCPM y el Problema de la Aceptación, para poder concluir entonces que el PCPM es indecidible. La reducción se puede hacer mediante la siguiente construcción,

Sea M, con alfabetos Σ y Γ, y sea $w \in \Sigma^*$. Para ver si $w \in$ L(M), se estudia si la siguiente instancia del PCPM

$$\text{A} = \{u_1, u_2, \ldots, u_k\}, \text{B} = \{v_1, v_2, \ldots, v_k\},$$

tiene solución, sabiendo que los conjuntos A y B se forman a partir de la función de transición de la Máquina de Turing M, mediante la construcción de cinco grupos de fichas: Sea \$ $\notin \Gamma$, sea q_1 el estado inicial de M,

Grupo 1: $\boxed{\frac{\$}{\$q_1w\$}}$, siendo q_1 el estado inicial de M.

Grupo 2: $\boxed{\frac{\$}{\$}}$, $\boxed{\frac{\gamma}{\gamma}}, \forall\, \gamma \in \Gamma, \gamma \neq \texttt{B}$

Grupo 3: Formado a partir de f, distinguiendo entre cuatro tipos de transiciones,

1. $f(q,\sigma) = (p,\tau,R) \longrightarrow \boxed{\frac{q\sigma}{\tau p}}$
2. $f(q,\texttt{B}) = (p,\tau,R) \longrightarrow \boxed{\frac{q\$}{\tau p\$}}$
3. $f(q,\sigma) = (p,\tau,L) \longrightarrow \boxed{\frac{\gamma q\sigma}{p\gamma\tau}}, \forall\, \gamma \in \Gamma, \gamma \neq \texttt{B}$
4. $f(q,\texttt{B}) = (p,\tau,L) \longrightarrow \boxed{\frac{\gamma q\$}{p\gamma\tau\$}}, \forall\, \gamma \in \Gamma, \gamma \neq \texttt{B}$

Grupo 4: A partir de los estados finales, $\forall\, q \in F$, $\forall\, \sigma, \gamma \in \Gamma - \{\texttt{B}\}$,

$$\boxed{\frac{\sigma q\tau}{q}}, \boxed{\frac{\sigma q\$}{q\$}}, \boxed{\frac{\$q\tau}{\$q}}$$

Grupo 5: $\forall\, q \in F$,

$$\boxed{\frac{q\$\$}{\$}}$$

Además de establecer la reducción anterior, para completar la demostración del teorema 8.4, se debe probar el siguiente resultado:

Lema 8.5 *M acepta $w \Leftrightarrow$ hay una solución a la instancia derivada del PCPM.*

Del teorema 8.4 y de los lemas 8.3 y 8.4 se concluye que:

Teorema 8.5 *El PCP es INDECIDIBLE.*

8.4. Problemas Propuestos.

1. Dada la cadena

   `1110101001010011010010010010011010001001000100111`

   ¿cuál es el lenguaje que reconoce la MT que codifica?

2. Demostrar que $\mathcal{L}_H$ es un lenguaje recursivamente enumerable.

3. Demostrar que el complementario de $\mathcal{L}_d$, $\bar{\mathcal{L}}_d$, es un lenguaje recursivamente enumerable no recursivo.

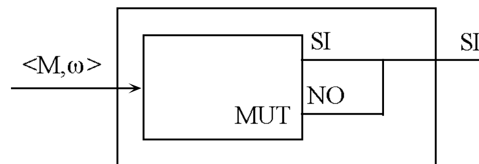
4. Indicar cómo se podría construir un generador para $\bar{\mathcal{L}}_d$.

5. Imagínese que no se sabe que el lenguaje $\mathcal{L}_d$ no es recursivamente enumerable y que hay que construir un generador de dicho lenguaje, ¿qué problema se pondría de manifiesto?

6. ¿Cuál es la intersección de $\mathcal{L}_U$ y de $\mathcal{L}_H$? ¿Por qué?

7. Construir el convertidor de cadenas, a partir de $G_c(\Sigma^*)$.

8. Dada la construcción de la figura:



   a) ¿Qué lenguaje acepta? ¿Por qué?.

   b) Si $\mathcal{L}_U$ fuera recursivo, ¿sería recursivo el lenguaje al que se refiere el apartado (a)? ¿Por qué?.

9. Determinar si los siguientes lenguajes son recursivos, recursivamente enumerables no recursivos o no recursivamente enumerables:

   a) $L = \{w \mid w = 0x,\ x \in \mathcal{L}_U\}$,

b) $L = \{w \mid w = 0x,\ x \in \mathcal{L}_U \cup w = 1y,\ y \in \bar{\mathcal{L}}_U\}$,

c) El lenguaje complementario del lenguaje del apartado (b).

10. *a*) Justificar la siguiente afirmación (que es cierta): *Los códigos de MT forman un lenguaje recursivo*.

    b) Dado que el enunciado a comentar en el apartado (a) es cierto, ¿cómo es posible que $\mathcal{L}_U$ sea un lenguaje recursivamente enumerable no recursivo?

11. Sea un lenguaje $\mathcal{L}$, tal que

$$\mathcal{L} = \{(i, j) \mid \ M_i \ acepta \ w_j \wedge M_j \ acepta \ w_i\}.$$

¿$\mathcal{L}$ es Recursivo o Recursivamente Enumerable No Recursivo? Justificar. Construir un generador para dicho lenguaje.

12. Indicar para cada uno de los siguientes lenguajes si son recursivos, recursivamente enumerables no recursivos o no recursivamente enumerables, justificando el porqué:

    a) $L = \{(i, j, k) \in N \times N \times N \mid M_i \ acepta \ w_j \ en \ k \ pasos\}$

    b) $L = \{(i, j, k) \in N \times N \times N \mid M_i \ acepta \ w_j \ en \ más \ de \ k \ pasos\}$

13. Indicar si las siguientes instancias del PCP tienen o no tienen solución:

    a) $\boxed{\frac{aaa}{aa}}, \boxed{\frac{baa}{abaaa}}$

    b) $\boxed{\frac{ab}{aba}}, \boxed{\frac{bba}{aa}}, \boxed{\frac{aba}{bab}}$

    c) $\boxed{\frac{a}{aa}}, \boxed{\frac{bb}{b}}, \boxed{\frac{a}{bb}}$

    d) $\boxed{\frac{a}{aaa}}, \boxed{\frac{aab}{b}}, \boxed{\frac{abaa}{ab}}$

    e) $\boxed{\frac{ab}{a}}, \boxed{\frac{ba}{bab}}, \boxed{\frac{b}{aa}}, \boxed{\frac{ba}{ab}}$

    f) $\boxed{\frac{ab}{aba}}, \boxed{\frac{baa}{aa}}, \boxed{\frac{aba}{aaa}}$

    g) $\boxed{\frac{aa}{aab}}, \boxed{\frac{bb}{ba}}, \boxed{\frac{abb}{b}}$

14. Por reducción al PCP, determinar si la MT con función de transición

	a	b	B
q_1	(q_2, b, R)	(q_2, a, L)	(q_2, b, L)
q_2	(q_3, a, L)	(q_1, a, R)	(q_2, a, R)

acepta o no a la cadena *w* = *ab*.

15. Contestar de forma razonada a la siguiente cuestión: ¿se podría escribir un programa P que, para cualquier función F que se le pasara como dato de entrada, devolviera *cierto* si la ejecución de F provoca que se escriba en pantalla

    ```
    hola, mundo!
    ```

    y *falso* en caso contrario?

    > Antes de contestar a la pregunta anterior, creemos que os puede ayudar la siguiente consideración:
    >
    > La conjetura de los números perfectos dice que *todo número perfecto es par*. Considérese lo que ocurriría si la siguiente función fuese la función F del enunciado:

```
boole perfecto(int n){
  int i, acum;
  boole loEs;

  acum=0;
  i=1;
  loEs=falso;
  while (i<=n/2){
     if (n%i==0)
        acum=acum+i;
     i=i+1;
  }
  if (acum==n) {
     loEs=cierto;
     if (n%2!=0)
        printf("hola,mundo!");
  }
  return loEs;
}
```

Capítulo 9

Introducción a la Complejidad Computacional

Índice General

9.1. Introducción. 145
9.1.1. El Dilema del Contrabandista. 146
9.2. Definiciones Básicas. 148
9.2.1. Clases de Complejidad. 149
9.3. Relaciones entre las Clases de Complejidad Computacional. 150
9.3.1. Relaciones entre Clases Deterministas y No Deterministas. . . . 151
9.3.2. Las Clases de Complejidad Polinómica. 153
9.4. Introducción a la Teoría de Complejidad Computacional. 156
9.5. Problemas Propuestos. 159

9.1. Introducción.

Una vez establecidos los resultados que permiten saber qué problemas se pueden resolver por medio de un algoritmo, el objetivo es fijar el "precio" a pagar; es decir, cuál va a ser el coste asociado al algoritmo y si dicha solución algorítmica será factible, esto es, con un tiempo de ejecución "razonablemente" breve.

La *Teoría de la Complejidad* permite estimar la dificultad o, mejor dicho, la *tratabilidad* o *intratabilidad* de un problema y establecer si se dispone o no de una solución algorítmica *factible*.

Esta teoría dota a la informática de las herramientas necesarias para, por ejemplo, poder determinar a priori cuál es el comportamiento asintótico de un algoritmo a medida que crece el tamaño del problema (y si merece o no la pena intentar abordarlo) o cuando un determinado problema requiere de una solución tan compleja, que es preferible no intentar

resolver directamente su caso general, sino ir tratando subproblemas que representan casos particulares.

El objetivo de la teoría sería establecer una taxonomía, una clasificación de los problemas, atendiendo a cuál es la complejidad del mejor algoritmo conocido[1] para solucionar dicho problema. Desafortunadamente, esta teoría no suele ofrecer resultados absolutos: no se suelen realizar afirmaciones del tipo,

"... el problema X tiene un nivel de dificultad D ..."

sino como la siguiente:

"... el problema X es tan difícil de resolver como el problema Y; por lo tanto, encontrar una solución eficiente para X es tan difícil como encontrarla para Y..."

No es habitual que se enuncien categóricamente resultados sobre un problema dado (salvo la indicación de que todavía no se dispone de un algoritmo eficiente para ese problema), sino que lo habitual es expresar la dificultad de resolver un problema en términos de comparación con otro. Tal y como se verá, esto conduce a que la teoría clasifique los problemas en *clases* que representan el mismo nivel de dificultad[2]. Entre esas clases, destacan las que comprenden problemas tan difíciles (tan complejos) que aún no se han encontrado algoritmos eficientes para ellos y que llevan a buena parte de la comunidad científica a creer que esos problemas son realmente intratables. De hecho, el reto que se plantea la Teoría de la Complejidad puede ser tan simple como determinar si existen problemas intratables *"per se"* o si dicha comunidad científica es tan lerda como para no haber encontrado todavía la solución eficiente (¡y no resulta especialmente atrayente elegir ninguna de esas dos opciones!).

Los contenidos teóricos de este tema, no son más que la formalización en el marco de la Teoría de Complejidad, de resultados que se pueden haber visto, con mayor o menor detalle, en asignaturas de programación. Como una introducción informal de las principales ideas que se van a desarrollar y formalizar en el tema, se presenta el problema denominado *Dilema del contrabandista*.

9.1.1. El Dilema del Contrabandista.

Un aprendiz de contrabandista se está financiando el Máster en "Contrabando de Calidad–Superior" con los beneficios obtenidos al pasar monedas antiguas desde Asia. Para ello utiliza un compartimento secreto en su riñonera, que le permite almacenar, si no quiere que en la frontera descubran su "negocio", sólo 500 gramos de monedas.

[1] Un algoritmo resuelve un problema, pero dado un problema hay más de un algoritmo que lo resuelve.

[2] A lo largo de este tema, se podría pensar en traducir la palabra "dificultad" por "complejidad"; pero, se pretende diferenciar entre la *dificultad* de resolver un *problema* y la *complejidad* del *algoritmo* que lo resuelva: es decir, cuando se dice que los problemas X e Y son igual de difíciles de resolver, es porque sus mejores algoritmos conocidos tienen una complejidad similar. Por purismo, o por mera deformación profesional (tanto leer en Inglés tiene consecuencias perniciosas sobre el vocabulario propio ;-), se mantiene "dificultad".

Para poder realizar comparaciones, se va a completar el enunciado con dos escenarios distintos:

Primer escenario:

> *Sus suministradores asiáticos le ofrecen 20 monedas distintas; cada una de ellas pesa 50 gramos y sus precios de venta en el mercado negro, sus valores, van desde los 100 euros hasta los 2000 euros.*

Segundo escenario:

> *Sus suministradores asiáticos le ofrecen 20 monedas distintas; cada moneda tiene distinto peso, entre 30 y 200 gramos, y sus valores van desde 100 euros hasta 2000 euros.*

Evidentemente, el dilema del contrabandista consiste en cómo realizar una elección óptima de monedas, de forma que le permita obtener el máximo beneficio en un viaje. Y cada escenario va a suponer una estrategia distinta:

Primer escenario:

> *Si todas las monedas pesan lo mismo, 50 gramos, puede llevar 10 monedas en la riñonera. Obviamente, le interesa elegir las 10 más valiosas, por lo que si ordena las 20 monedas por valor decreciente, le basta con elegir las 10 primeras.*

Segundo escenario:

> *Como cada moneda tiene distinto peso y valor, no es suficiente con realizar una ordenación por valor, ya que no garantiza el máximo beneficio. El contrabandista debería estudiar todas las combinaciones de monedas que pesan menos de 500 gramos y quedarse con la más ventajosa.*

El coste del algoritmo del primer escenario es el coste asociado a ordenar 20 monedas por valor decreciente: aún utilizando un algoritmo tan simple como el de ordenación por selección, la complejidad resulta ser de $O(\frac{N^2}{2})$, siendo N el número de monedas a ordenar. En este escenario, ello viene a suponer unas 200 comparaciones, para obtener un beneficio máximo.

Pero en el segundo escenario, el coste no es tan bajo: hay 2^N formas de ordenar un conjunto de N monedas; en el ejemplo concreto, $N = 20$, se obtienen unas 2^{20} posibilidades a considerar (aproximadamente, un millón), ver cuáles son factibles y determinar cuál maximiza el beneficio.

De lo anterior, se puede sacar la impresión (correcta) de que el segundo escenario supone un problema "más difícil" de resolver que el problema asociado al primer escenario.

Pero aún falta introducir un nuevo aspecto: en su próximo viaje, el contrabandista se encuentra con una agradable sorpresa. Sus suministradores le informan de que se acaba de descubrir una tumba pre-babilónica con 100 monedas distintas. ¿Cuál es la influencia en los dos escenarios? En el primero, el contrabandista tendrá que realizar una ordenación que le supondrá unas 5000 comparaciones, para maximizar beneficios. En el segundo, sin embargo, ahora tendrá que considerar 2^{100} posibilidades distintas, ¿cuánto tiempo le llevará la elección?

En el primer escenario, el comportamiento es *polinómico*: a medida que crece el tamaño del problema (de 20 a 100, de 100 a 1000, ...) hay que realizar más operaciones, pero el crecimiento viene dado por un factor cuadrático: de 20 a 100, será 5^2, de 100 a 1000, será 10^2...

En el segundo escenario, el comportamiento es *exponencial*: con añadir un único elemento, con incrementar en una unidad el tamaño del problema, el número de operaciones se multiplica por 2; cuando se pasa de 20 a 100 monedas, se pasa de $1{,}000{,}000$ de posibilidades a $1{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000$.

Estos dos escenarios vienen a mostrar cuál es la importancia de la escalabilidad de un problema y del comportamiento asintótico de los algoritmos resultantes. Usualmente, en Teoría de Complejidad se estima el comportamiento polinómico como deseable, y los problemas que pueden resolverse con un algoritmo que presenta una función de orden polinómica se consideran *tratables*. Cuando no se puede encontrar un algoritmo polinómico para resolver el problema, tal y como ocurre en el segundo escenario, el problema se considera *intratable*.

Y esto lleva al "dilema del informático": no hay resultados que permitan estimar cuándo un problema puede resolverse por medio de un algoritmo polinómico, a no ser, claro está, que tal algoritmo se haya encontrado. Por lo tanto, no hay ningún resultado teórico que permita establecer que un determinado problema es, por naturaleza, intratable: sólo puede afirmarse que el mejor algoritmo conocido hasta el momento tiene una complejidad superior a la polinómica. Una forma de atacar este "dilema" consiste en establecer *clases* de problemas, de forma que problemas con la misma dificultad estén en la misma clase. Esta clasificación, además, dota al informático de una herramienta interesante en su objetivo de encontrar algoritmos eficientes: si todos los problemas de una determinada clase son de la misma dificultad que un problema que es *representante* de esa clase[3], cualquier resultado o mejoría que se obtenga para dicho problema, podrá aplicarse a todos los demás de su misma clase.

9.2. Definiciones Básicas.

El análisis de algoritmos supone la obtención de su complejidad espacial y/o temporal como una función de la talla del problema. En este contexto, se entiende como *talla* el dato o conjunto de datos que cuando varía, hace que varíe el valor de la complejidad. Por su

[3]Es decir, encontrar una solución eficiente para cualquier problema de esa clase es igual de difícil que encontrar una solución eficiente para ese problema "especial".

parte, la *complejidad espacial* da una medida de la cantidad de objetos manejados en el algoritmo (entendida como una medida de la memoria que consumirá una computación) y la *complejidad temporal* da una medida del número de operaciones realizadas (entendido como una medida del tiempo necesario para realizar una computación).

De esta forma, para realizar el análisis primero hay que determinar cuál es la talla del problema. A continuación, se elige una unidad para poder determinar la complejidad espacial y una unidad para poder determinar cuál es la complejidad temporal. Sin embargo, en ocasiones no hay un criterio claro para escoger dichas unidades y poder establecer las correspondientes medidas. El modelo de Máquina de Turing permite definir fácilmente todos los conceptos anteriores:

- la talla del problema se identifica con la longitud de la cadena de entrada,
- la complejidad espacial se asocia al número de celdas de la Máquina de Turing visitadas, y
- la complejidad temporal viene dada por el número de movimientos del cabezal, asumiendo que cada movimiento se realizará en un tiempo fijo.

Las definiciones formales son:

Definición 9.1 (Complejidad Espacial) *Sea M una Máquina de Turing; si, para cualquier cadena de longitud* n*, en M sólo es necesario consultar $\mathcal{S}(n)$ celdas, se dice que M es una Máquina de Turing acotada espacialmente por la función $\mathcal{S}(n)$. También se dice que L(M) es un lenguaje con complejidad espacial $\mathcal{S}(n)$.*

Definición 9.2 (Complejidad Temporal) *Sea M una Máquina de Turing; si, para cualquier entrada de longitud* n*, M realiza como máximo $\mathcal{T}(n)$ movimientos, de cabezal, se dice que M es una Máquina de Turing acotada temporalmente por la función $\mathcal{T}(n)$. También se dice que L(M) es un lenguaje con complejidad temporal $\mathcal{T}(n)$.*

9.2.1. Clases de Complejidad.

Cuando se realiza el análisis de los algoritmos en programación, nunca se busca una expresión exacta de la función de coste de un determinado algoritmo, sino que interesa más establecer cuál es su comportamiento asintótico. Así se obtiene una medida significativa, pero más simple, de cómo evolucionará dicho algoritmo a medida que la talla del problema crece. Y se introduce el concepto de *orden de coste* que permite, además, clasificar los algoritmos en "familias", según su comportamiento asintótico estimado.

Los teoremas siguientes permiten formalizar este concepto en el ámbito de la Teoría de Complejidad:

Teorema 9.1 (de Compresión) *Sea L un lenguaje aceptado por una Máquina de Turing M con k cintas de trabajo con cota espacial $\mathcal{S}(n)$. Para todo $c > 0$ hay una Máquina de Turing M′ acotada espacialmente por $c\mathcal{S}(n)$ que acepta L.*

Teorema 9.2 (de Aceleración Lineal) *Supuesto que $\text{ínf}_{n\to\infty} \frac{\mathcal{T}(n)}{n} = \infty$, sea L un lenguaje aceptado por una Máquina de Turing M de k cintas, con cota temporal $\mathcal{T}(n)$. Para todo $c > 0$ hay una Máquina de Turing M′ de k cintas, acotada temporalmente por $c\mathcal{T}(n)$ que acepta L.*

Estos teoremas permiten realizar el análisis asintótico de la complejidad espacial y temporal, a fin de comparar el comportamiento de dos Máquinas de Turing: por ejemplo, dadas dos Máquina de Turing con complejidades espaciales $\mathcal{S}_1(n) = 3n^3 + 1$ y $\mathcal{S}_2(n) = n^3 + 5n^2 + 1$, se dice que ambas están acotadas espacialmente por $\mathcal{S}(n) = n^3$. Esto permite definir las *clases de complejidad*.

Definición 9.3 (Clases de Complejidad Espacial)
La familia de lenguajes aceptados por Máquinas de Turing deterministas con complejidad espacial $\mathcal{S}(n)$ es DSPACE($\mathcal{S}(n)$).
La familia de lenguajes aceptados por Máquinas de Turing no deterministas con complejidad espacial $\mathcal{S}(n)$ es NSPACE($\mathcal{S}(n)$).
Son conocidas como ***clases de complejidad espacial****.*

Definición 9.4 (Clases de Complejidad Temporal)
La familia de lenguajes aceptados por Máquinas de Turing deterministas con complejidad temporal $\mathcal{T}(n)$ es DTIME($\mathcal{T}(n)$).
La familia de lenguajes aceptados por Máquinas de Turing no deterministas con complejidad temporal $\mathcal{T}(n)$ es NTIME($\mathcal{T}(n)$).
Son conocidas como ***clases de complejidad temporal****.*

9.3. Relaciones entre las Clases de Complejidad Computacional.

Los siguientes teoremas establecen las relaciones básicas entre las clases de complejidad espacial:

Teorema 9.3 *Sean $\mathcal{S}$, $\mathcal{S}_1$ y $\mathcal{S}_2$ funciones de $\mathcal{N}$ en $\mathcal{N}$. Se asume que $\mathcal{S}_1(n) \leq \mathcal{S}_2(n) \; \forall \, n$ y que $c > 0$. Entonces*

1. $DSPACE(\mathcal{S}_1(n)) \subseteq DSPACE(\mathcal{S}_2(n))$,
2. $NSPACE(\mathcal{S}_1(n)) \subseteq NSPACE(\mathcal{S}_2(n))$,
3. $DSPACE(\mathcal{S}(n)) = DSPACE(c\mathcal{S}(n))$,

y entre las clases de complejidad temporal:

Teorema 9.4 *Sean $\mathcal{T}$, $\mathcal{T}_1$ y $\mathcal{T}_2$ funciones de $\mathcal{N}$ en $\mathcal{N}$. Se asume que $\mathcal{T}_1(n) \leq \mathcal{T}_2(n) \; \forall \, n$ y que $c > 0$. Entonces*

1. $DTIME(\mathcal{T}_1(n)) \subseteq DTIME(\mathcal{T}_2(n))$,
2. $NTIME(\mathcal{T}_1(n)) \subseteq NTIME(\mathcal{T}_2(n))$,
3. *Si* $\inf_{n\to\infty} \frac{\mathcal{T}(n)}{n} = \infty$, $DTIME(\mathcal{T}(n)) = DTIME(c\mathcal{T}(n))$.

En ambos teoremas, las relaciones *(1)* y *(2)* dan una medida que no es absoluta: se identifica un problema con la clase más restrictiva a la que pertenece (según el mejor algoritmo conocido para resolverlo), pero la relación entre clases se establece con un "$\subseteq$"; es decir, no se cierra la "frontera" entre clases, de forma que un problema puede pasar a una clase inferior (en cuanto se encuentre un algoritmo más eficiente, por ejemplo). Por su parte, la relación *(3)* supone una extensión de los teoremas de compresión y de aceleración lineal.

Además, puede establecerse una relación entre las clases de complejidad temporal y las clases de complejidad espacial:

Teorema 9.5 (Relación espacio/tiempo) *Si $L \in DTIME(f(n))$ entonces $L \in DSPACE(f(n))$.*

9.3.1. Relaciones entre Clases Deterministas y No Deterministas.

Un algoritmo determinista emplea procedimientos de decisión que siguen un curso único y predeterminado de forma que, con el mismo juego de entradas, siempre seguirá el mismo comportamiento. Pero es posible diseñar también algoritmos no deterministas, que ante una misma entrada pueden obtener salidas distintas, ya que al tener la posibilidad de elegir una de entre varias opciones, elegirá la más beneficiosa en cada situación.

Por ejemplo, ante el problema de salir de una habitación con cinco puertas, pero con una única puerta que conduzca a la salida, un algoritmo determinista puede consistir en ir

probando una a una, hasta dar con la correcta; un algoritmo no determinista escogerá directamente cuál es la puerta correcta. Ambos algoritmos solucionan el mismo problema (se consigue encontrar la salida) e, intuitivamente, parece que el no determinista será más rápido en general que el determinista.

El modelo de Máquina de Turing contempla el determinismo y el no determinismo; y, de hecho, al presentar el modelo no determinista también se introducía, intutitamente, la idea de el no determinismo brinda un modelo de computación más eficiente que el determinista. Esa idea intuitiva se formaliza con el siguiente teorema:

Teorema 9.6 *Sean $\mathcal{S}$ y $\mathcal{T}$ funciones de $\mathcal{N}$ en $\mathcal{N}$. Entonces*

1. $DSPACE(\mathcal{S}(n)) \subseteq NSPACE(\mathcal{S}(n))$,
2. $DTIME(\mathcal{T}(n)) \subseteq NTIME(\mathcal{T}(n))$.

Pero, en el mundo real sólo se dispone de máquinas deterministas[4] y, en consecuencia, un algoritmo no determinista se debe transformar en determinista, si se pretende codificarlo en una máquina real. Por eso, interesa saber cuál es el precio que se debe pagar en esa transformación:

Teorema 9.7 (de Savitch) *Sea $\mathcal{S}$ una función de $\mathcal{N}$ en $\mathcal{N}$. Entonces $NSPACE(\mathcal{S}(n)) \subseteq DSPACE((\mathcal{S}(n))^2)$.*

Es decir, en el caso de la complejidad espacial, el paso de un algoritmo no determinista a determinista supone, como mucho, elevar la función de complejidad al cuadrado.

Teorema 9.8 *Si L es aceptado por una Máquina de Turing no determinista con complejidad temporal $\mathcal{T}(n)$, entonces L es aceptado por una Máquina de Turing determinista con complejidad temporal $d^{\mathcal{T}(n)}$, para alguna constante* d.
Es decir,

$$\text{si } L \in NTIME(\mathcal{T}(n)) \text{ entonces } L \in DTIME(d^{\mathcal{T}(n)}),$$

para alguna constante d.

Nótese que una relación que para la complejidad espacial puede, como mucho, acabar resultando cuadrática, en el caso de la complejidad temporal puede llegar a suponer la transformación en una función exponencial. Este resultado tiene importantes consecuencias, tal y como se verá a continuación.

[4] ¿Por ahora? ;-).

9.3.2. Las Clases de Complejidad Polinómica.

Como se comentó en la introducción, se consideran poblemas tratables aquellos problemas que se pueden resolver mediante algoritmos con comportamiento descrito por funciones polinómicas. Por lo tanto, puede resultar interesante agrupar en una misma "superclase" a todos los problemas tratables. Lo que sigue son la definición de esas clases, *PSPACE*, *NPSPACE*, *P* y *NP*, según que se trate de complejidad espacial o temporal, sobre Máquinas de Turing deterministas o no deterministas.

Definición 9.5 *Si L es aceptado por una Máquina de Turing determinista con complejidad espacial polinómica,* $\mathcal{S}(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$*, se dice que L está en la clase de lenguajes PSPACE.*
Si L es aceptado por una Máquina de Turing no determinista con cota espacial polinómica, se dice que L está en la clase NPSPACE.

Definición 9.6 *Si L es aceptado por una Máquina de Turing determinista con complejidad temporal polinómica,* $\mathcal{T}(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0$*, se dice que L está en la clase de lenguajes P.*
Si L es aceptado por una Máquina de Turing no determinista con cota temporal polinómica, se dice que L está en la clase NP.

Es posible caracterizar estas clases, así como establecer relaciones entre ellas. En el caso de la complejidad espacial, se puede caracterizar la clase de lenguajes *PSPACE* como

$$PSPACE = \bigcup_{k=0}^{\infty} DSPACE(n^k)$$

es decir, la unión de todas las clases de complejidad espacial polinómicas sobre Máquinas de Turing deterministas y, de forma análoga, para las no deterministas se tiene que

$$NPSPACE = \bigcup_{k=0}^{\infty} NSPACE(n^k).$$

Además, se puede demostrar lo siguiente:

PSPACE = NPSPACE.

Si un problema se resuelve mediante un algoritmo no determinista de complejidad espacial polinómica, también se puede resolver mediante un algoritmo determinista de complejidad espacial polinómica.

¿Qué ocurre si se intenta llegar al mismo resultado en el ámbito de la complejidad temporal? Se puede caracterizar la clase de lenguajes *P* como

$$P = \bigcup_{k=1}^{\infty} DTIME(n^k),$$

la unión de todos los problemas que se resuelven con cota temporal polinómica usando Máquinas de Turing deterministas y, de forma análoga,

$$NP = \bigcup_{k=1}^{\infty} NTIME(n^k),$$

como la unión de todos los problemas con cota temporal polinómica sobre máquinas no deterministas.

Pero, si se intenta establecer una relación entre ambas clases, similar a la relación entre *PSPACE* y *NPSPACE*, sólo se puede llegar a demostrar la relación

$$P \subseteq NP.$$

No es posible establecer la inclusión inversa para intentar establecer la igualdad: el único resultado del que se dispone es el teorema 9.8, que establece que, en general, el paso de trabajar con una máquina no determinista a una máquina determinista puede llegar a suponer un aumento exponencial de la complejidad temporal[5].

Es decir: hay una serie de problemas en la clase *NP* que podrían solucionarse con algoritmos de cota temporal polinómica si se dispusiera de la máquinas no deterministas, gobernadas por algoritmos no deterministas. Mientras tanto, se deben transformar en sus equivalentes deterministas. Y en esta transformación su coste computacional puede llegar a transformarse en exponencial. Y, de hecho, los denominados en programación *Problemas NP*, son problemas que plantean este reto a la comunidad científica: sus algoritmos no deterministas son polinómicos, sus algoritmos deterministas son exponenciales... ¿son problemas *"intratables"* ... o son problemas que no se sabe cómo tratar? ¿existe un algoritmo determinista polinómico para ellos y no se conoce todavía?

Por lo tanto, queda abierta la cuestión

¿P = NP?

seguramente, el problema más importante en Teoría de Computación, hoy en día. La comunidad científica se divide entre los que pretenden demostrar la igualdad – si esto se demostrara, los problemas que están en la clase NP también podrían resolverse en tiempo polinómico sobre máquinas deterministas – y los que pretenden demostrar la desigualdad, que P $\subset$ NP – en cuyo caso se sabría que hay problemas que nunca se podrán resolver sobre máquinas deterministas con cota polinómica. Es decir, se trabaja sobre las cuestiones:

[5] Un polinomio al cuadrado sigue siendo un polinomio, pero si interviene una función exponencial en la transformación...

¿P = NP? para lo que habría que encontrar un método con cota temporal polinómica (sean o no los polinomios del mismo grado) que permita transformar cualquier Máquina de Turing no determinista en una Máquina de Turing determinista.

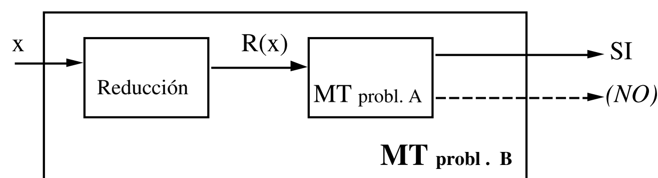
¿P $\neq$NP? para lo que habría que encontrar un lenguaje que esté en NP y no esté en P. Es decir, demostrar para un determinado lenguaje reconocido con cota polinómica por una Máquina de Turing no determinista que es imposible que lo reconozca una Máquina de Turing determinista con cota polinómica.

Hay una tercera línea de trabajo cuyo objetivo sería probar que estas cuestiones son indemostrables.

9.4. Introducción a la Teoría de Complejidad Computacional.

Buena parte de los esfuerzos realizados en la Teoría de la Complejidad Computacional se centra en las clases P y NP y en el estudio de las relaciones entre ambas clases, para intentar establecer resultados que permitan resolver la cuestión *¿P=NP?*. Para ello, además del conocimiento sobre qué son las clases de complejidad y cuáles son las relaciones entre ellas, se utiliza la reducibilidad de lenguajes como herramienta básica.

En general, el mecanismo de reducción del lenguaje A (o problema A) al lenguaje B (o problema B), se define como



Es decir, se puede obtener una "traducción" de las cadenas de B en cadenas de A. Si lo que se pretende es estudiar si A y B pertenecen a la misma clase de complejidad además es importante asegurar que esa transformación no afecta a la complejidad temporal: debe tenerse en cuenta que para resolver B, se debe resolver A y además realizar la reducción (transformar una cadena de B en una cadena de A). El tiempo de cómputo de la función R, afecta al tiempo de cómputo total. Por eso, si se puede calcular en tiempo polinómico, se dice que B se puede reducir en tiempo polinómico a A, B $<_p$ A.

Definición 9.7 (Reducibilidad Polinómica) *Un lenguaje L_1 es reducible en tiempo polinómico a otro lenguaje L_2, si hay una Máquina de Turing que calcule en tiempo polinómico una función, f, cumpliéndose que $f(u) \in L_2 \Leftrightarrow u \in L_1$.*
Se denota como $L_1 <_p L_2$.

Es decir, es posible calcular $f(u)$ en tiempo polinómico en una Máquina de Turing. La importancia de este concepto se debe al siguiente teorema, que garantiza que la reducibilidad polinómica mantiene al lenguaje reducido en la misma clase que el lenguaje al que se reduce:

Teorema 9.9 *Si L_1 es reducible en tiempo polinómico a L_2, entonces:*

1. *si $L_2 \in P \Rightarrow L_1 \in P$,*
2. *si $L_2 \in NP \Rightarrow L_1 \in NP$.*

El concepto de reducibilidad polinómica es una herramienta básica para determinar a cuál de las clases, *P* o *NP*, pertenece un lenguaje. Puede servir como punto de partida para el siguiente razonamiento: si cualquier lenguaje de una determinada clase de complejidad se pudiera reducir a un determinado lenguaje, basta estudiar cómo reconocer ese lenguaje y cuál es la cota temporal de la Máquina de Turing que lo reconoce. Es decir, basta con estudiar cuál es el algoritmo con mejor cota temporal para resolver el problema que representa ese lenguaje y *como todos los problemas de su clase* no superan la cota polinómica en su reducción a él, serán igual de *difíciles* de resolver.

Definición 9.8 *Un lenguaje L se dice que es* $\mathcal{NP} - \mathbf{hard}$ *(*$\mathcal{NP}$*-díficil ó* $\mathcal{NP}$*-duro) si,* $\forall\ L' \in \mathcal{NP},\ L' <_p L$*. Es decir, todos los lenguajes de* $\mathcal{NP}$ *se reducen a L en tiempo polinómico.*

Definición 9.9 *Si L es* $\mathcal{NP}$*-hard y* $L \in \mathcal{NP}$*, entonces L es* $\mathcal{NP} - \mathbf{completo}$.

Si $L_1 <_p L_2$ entonces determinar si $w \in L_1$ no es más díficil que determinar si $f(w) \in L_2$, siendo f la función que reduce L_1 a L_2 en tiempo polinómico. Y si se habla de lenguajes $\mathcal{NP} - completos$, la reducción polinómica puede aplicarse a todos los lenguajes de la clase $\mathcal{NP}$. Por eso, el resultado es trascendental: sólo con encontrar un lenguaje *NP-completo* reducible en tiempo polinómico a un lenguaje de la clase *P*, se sabría que cualquier otro lenguaje de *NP* también admitiría una reducción polinómica.

Teorema 9.10 (de Lambsmother)
Si L es un lenguaje NP-completo y $L \in P \Rightarrow$ **P = NP**.

Demostración:

Sea cualquier lenguaje $L_1 \in NP \Rightarrow L_1 <_p L$, que es NP-completo.

Como $L \in P \Rightarrow L_1 \in P$.

c.q.d

Este teorema permitiría demostrar *P=NP*, en el caso de encontrar un lenguaje que cumpliera las condiciones. Por lo tanto, el primer paso en este sentido fue buscar un lenguaje NP completo. El primer lenguaje NP completo encontrado fue $\mathcal{L}_{sat}$.

Este lenguaje representa el *Problema de la Satisfactibilidad*:

> “Dado un conjunto de cláusulas booleanas (expresiones booleanas formadas con negadores y disyunciones de constantes y/o variables booleanas), ¿existe

un conjunto de valores para las variables que intervienen en las cláusulas que las satisfagan todas?".

Este problema de decisión se asocia al lenguaje

$$\mathcal{L}_{sat} = \{w \in \Sigma^* \mid w \text{ representa un conjunto de cláusulas satifactibles}\},$$

siendo $\Sigma = \{$'0', '1', ',', '&', '$\neg$', '$\vee$'$\}$ asumiendo que la variable x_i se codifica como la cadena '&i', en la que *i* se expresa en binario.

Ejemplo:

$$C_1 = \{x_1 \vee \neg x_2, \neg x_1 \vee \neg x_2 \vee x_3, \neg x_1 \vee x_2 \vee x_3\}$$

	$x_1 \vee \neg x_2$	$\neg x_1 \vee \neg x_2 \vee x_3$	$\neg x_1 \vee x_2 \vee x_3$
000	**1**	**1**	**1**
001	**1**	**1**	**1**
010	*0*		
011	*0*		
100	*1*	*1*	*0*
101	**1**	**1**	**1**
110	*1*	*0*	
111	**1**	**1**	**1**

es un conjunto de cláusulas satifactibles, ya que $x_1 = 0$*,* $x_2 = 0$*,* $x_3 = 0$ *ó* $x_1 = 1$*,* $x_2 = 0$*,* $x_3 = 1$*, son valores que satisfacen todas las cláusulas. Sin embargo, C2 no lo es:*

$$C_2 = \{\neg x_1, x_1 \vee x_2, x_1 \vee \neg x_2\}$$

	$\neg x_1$	$x_1 \vee x_2$	$x_1 \vee \neg x_2$
00	*1*	*0*	
01	*1*	*1*	*0*
10	*0*		
11	*0*		

Teorema 9.11 (de Cook) $\mathcal{L}_{sat}$ *es un lenguaje NP-completo.*

La importancia de esta demostración es que, una vez que se ha determinado que un lenguaje es NP-completo, se simplifica la búsqueda de otros ya que, para ello, se pueden aplicar los siguientes resultados:

Lema 9.1 *Si L_1 es NP-completo y $L_1 <_p L_2$, entonces L_2 es NP-hard.*

Corolario 9.1 *Si L_1 es NP-completo y $L_1 <_p L_2$ y $L_2 \in NP$, entonces L_2 es NP-completo.*

La idea es que cuántos más problemas NP-completos se conozcan, más posibilidades existen de que haya alguno que cumpla las condiciones del teorema 9.10. En la actualidad se ha demostrado que más de 400 lenguajes son NP-completos, pero aún no se ha podido encontrar un lenguaje que satisfaga las condiciones de dicho teorema.

9.5. Problemas Propuestos.

1. Teniendo en cuenta los soluciones vistas en clase de problemas para los ejercicios del tema 6 para caracterizar el funcionamiento de esa máquina, ¿cuál es la clase de complejidad temporal más restrictiva en la que situarías a cada uno de los siguientes lenguajes? ¿Por qué?.

   a) $0^n 1^n 2^n,\ n \geq 1$.

   b) $\{ww^{-1} \mid w \in \Sigma^*\}$.

   c) $\{0^{2^n} \mid n \geq 0\}$.

2. ¿Cuál es la relación entre el coste espacial de un lenguaje sobre una MTD y una MTnD? ¿Por qué?.

   ¿Cuál es la relación entre el coste temporal de un lenguaje sobre una MTD y una MTnD? ¿Por qué?.

3. Responder, de forma razonada, si las siguientes afirmaciones son ciertas o falsas:

   a) Todo problema que se resuelva con un algoritmo determinista de coste temporal exponencial, se puede resolver con un algoritmo no determinista de coste temporal polinómico.

   b) No existe relación alguna entre el coste espacial y el coste temporal de un mismo problema.

   c) Todo problema que se pueda reducir en tiempo polinómico al problema de la satisfactibilidad es un problema NP-completo.

   d) Cualquier problema que se resuelva con coste temporal polinómico sobre MT no deterministas, se resolverá con coste temporal exponencial sobre MT deterministas.

   e) Cualquier problema con coste temporal *T(n)* tendrá al menos coste espacial *S(n)=T(n)*.

f) La expresión de la complejidad espacial de un lenguaje no se ve afectada por el número de cintas de la MT que lo reconoce.

g) PSPACE = NPSPACE.

h) P $\neq$ NP.

i) La clase NP agrupa a los problemas cuyo coste computacional es No Polinómico.

j) La clase PSPACE contiene todos los problemas de complejidad temporal polinómica.

k) La inclusión de un problema en una determinada clase de complejidad temporal NTIME(T(n)), garantiza que, como poco, su complejidad temporal es T(n).

l) La reducibilidad polinómica garantiza que la complejidad temporal del problema reducido será exactamente la misma que la del problema al que se reduce.

m) Si un lenguaje A es reducible polinómicamente al lenguaje B, entonces ambos están en la clase P.

n) Si L_2 es NP-completo y $L_1 \in$ P y $L_1 <_p L_2$, entonces $P = NP$.

ñ) Si $L \in DTIME(n^2)$, entonces seguro que $L \notin NP$.

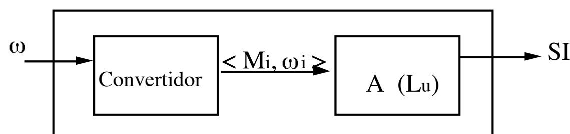
4. Sean L_1 y L_2 , sea f una función de reducibilidad entre L_1 y L_2. ¿Qué se puede afirmar sobre L_1 y L_2 si f es polinómica?

   ¿Y si f es una función exponencial?

5. Demostrar la siguiente afirmación:

   "Si $L_1 \in P$ y $L_2 \in P$, entonces $L_1 \cup L_2 \in P$."

6. El siguiente esquema ilustra la reducción de $\bar{\mathcal{L}}_d$ (el complementario de $\mathcal{L}_d$) a $\mathcal{L}_U$:



   ¿Se podría afirmar $\bar{\mathcal{L}}_d <_p \mathcal{L}_U$? (Pista: determinar la complejidad temporal aproximada del funcionamiento del convertidor de cadenas).

Bibliografía

[Cook71] Stephen Cook. "The Complexity of Theorem Proving Procedures". *Conference Record of Third Annual ACM Symposium on Theory of Computing*, Shaker Heighs, Ohio, pp. 151–158. 1971.

Si alguien está interesado tengo un revisión de este artículo del propio Stephen Cook, titulado *"The P versus NP Problem"*. Creo que es de 2000.

[Dowe01] Gilles Dowek. "El Infinito y el Universo de los Algoritmos". *Investigación y Ciencia*, Temas 23, pp. 74–76. 2001.

[Gare79] Michael R. Garey , David S. Johnson. "Computers and Intractability. A Guide to the Theory of NP-Completeness". *W.H. Freeman and Company, New York*. 1979.

[Hofs79] Douglas Hofstadter. "Gödel, Escher, Bach: Un Eterno y Grácil Bucle". *Colección Metatemas 14, Tusquets Editores*. 1979.

[Hopc79] John E. Hopcroft, Jeffrey D. Ullman. "Introduction to Automata Theory, Languages and Computation". *Addison-Wesley Publishing Company*. 1979.

[Hopc84] John E. Hopcroft. "Máquinas de Turing". *Investigación y Ciencia*, pp. 8–19. Julio 1984.

[Hopc02] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. "Introducción a la Teoría de Autómatas, Lenguajes y Computación". *Addison Wesley*. 2002.

[Johns00] David Johnson. "Challenges for Theorical Computer Science (Draft)". Last Updated, June 2000.

`<http://www.research.att.com/~dsj/nsflist.html>` [Última visita: 10 de Febrero de 2005]

[Kelley95] Dean Kelley. "Teoría de Autómatas y Lenguajes Formales". *Prentice-Hall Inc.* 1995.

[Lewis81] Harry R. Lewis, Christos Papadimitriou. "Elements of the Theory of Computation". *Prentice-Hall, Inc.* 1981.

[Mahon98] Michael S. Mahoney. "The Structures of Computation". *Proc. of the International Conference on the History of Computing, Heinz Nixdorf Forum*. Paderborn, Germany, 14–16. August 1998.

`<http://www.princeton.edu/~mike>` sec: Articles on the History of Computing [Última visita: 10 de Febrero de 2005]

[Shall95] Jeffrey Shallit. “A Very Brief History of Computer Science”. *Handout for CS134 course*. University of Waterloo. 1995.

(Pues lo ha quitado, si alguien está interesado yo lo tengo :-/)

<http://www.cs.uwaterloo.ca/~shallit>

[Última visita: 10 de Febrero de 2005]

[Shall98] Jeffrey Shallit. “The Busy Beaver Problem”. *Handout for CS360 course*. University of Waterloo. 1998.

[Sipser97] Michael Sipser. “Introduction to the Theory of Computation”. *PWS Publishing Company*. 1997.