Máquinas de Turing

Las máquinas de Turing tienen 3 visiones:

## Máquina de Turing calculadora: Una máquina de Turing es un artefacto que resuelve problemas (computacionales), en su visión más general produciendo o calculando una solución.

## Máquina de Turing reconocedora: En esta visión, una máquina sólo puede resolver un problema de decisión, produciendo únicamente la respuesta sí o no ante una instancia. Se denomina así porque la resolución de un problema consiste en reconocer el lenguaje que representa el problema, más precisamente, el lenguaje de las cadenas de símbolos que representan las instancias positivas del problema.

## Máquina de Turing generadora: En está visión se resuelve un problema generando todas sus instancias positivas.

En la gran mayoría de los casos consideraremos la máquina reconocedora.

Una máquina de Turing MT M está compuesta por:

* Una **cinta** infinita en los dos extremos, dividida en celdas. Cada celda puede almacenar un símbolo.
* Una **unidad de control**. En todo momento la unidad de control almacena el estado corriente de M.
* Un **cabezal**. En todo momento el cabezal apunta a una celda. El símbolo apuntado se denomina *símbolo corriente*. El cabezal puede moverse sólo de a una celda por vez, a la izquierda o a la derecha.

Los estados pertenecen a un conjunto **Q**, y los símbolos a un alfabeto **Γ.** Al comienzo,

en la configuración o *instancia inicial* de M, la cinta tiene la entrada limitada a izquierda y derecha por infinitos símbolos blancos, que se denotan con **B**. La unidad de control almacena el estado inicial, denotado en general con **q0.** Y el cabezal apunta al primer símbolo de la entrada, que es su símbolo de más a la izquierda. Si la entrada es la cadena vacía, denotada con **λ**, entonces el cabezal apunta a algún blanco.

A partir de la instancia inicial, M se comporta de acuerdo a lo especificado en su **función de transición δ**. M en cada paso lee un estado y un símbolo, eventualmente los modifica, y se mueve un lugar a la derecha o a la izquierda o no se mueve. Cuando **δ** no está definida para el estado corriente y el símbolo corriente, M se detiene. Si esto nunca sucede, es decir si M no se detiene, se dice que M tiene o entra en un *loop*

Formalmente, una máquina de Turing M es una **6-tupla (Q, Ʃ, Γ, δ, q0, F)**, tal que:

* **Q** es el conjunto de estados de M.
* **Ʃ** es el alfabeto de las entradas de M.
* **Γ** es el alfabeto de las cadenas de la cinta de M. Por convención, B ∈ (Γ– Ʃ).
* **δ** es la función de transición de M. Se define δ: Q x Γ → Q x Γ x {L, R, S}, tal que L representa el movimiento del cabezal a la izquierda, R el movimiento a la derecha, y S indica que el cabezal no se mueve.
* **q0** es el estado inicial de M.
* **F** es el conjunto de estados finales de M.

.

Considerando la visión de *MT reconocedora* de un lenguaje, si a partir de la entrada w

la MT M se detiene en un estado q ∈ F, se dice que **M acepta w**. En cambio, cuando a

partir de w la MT M se detiene en un estado q ∈ (Q – F) o no se detiene, se dice que M

**no acepta (o rechaza) w.** El conjunto de las cadenas aceptadas por la MT M es el

lenguaje aceptado o reconocido por M, y se denota con **L(M)**. Considerando la visión de

*MT calculadora*, sólo cuando M se detiene en un estado q ∈ F debe tenerse en cuenta

el contenido final de la cinta, es decir la salida.

Existen distintos *modelos de MT equivalentes* al modelo descripto previamente. Dos modelos de MT son equivalentes cuando para toda **MT M1** de un modelo existe una **MT M2** equivalente del otro, es decir que **L(M1) = L(M2).**

## Máquina de Turing con estados finales qA y qR: Las MT M de este modelo tienen dos estados especiales, el *estado qA de aceptación* y el *estado qR de rechazo*. Por definición se los considera fuera del conjunto Q. Cuando M se detiene lo hace sólo en qA o en qR.

M *acepta* una entrada w cuando a partir de w se detiene en el estado **qA** .Si en cambio se

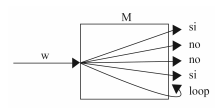
detiene en el estado **qR** o no se detiene, M *rechaza* w.

## Máquina de Turing con varias cintas: Las MT de este modelo tienen una cantidad finita K de cintas. La cinta de entrada es siempre la primera. Cuando es necesario se especifica también una cinta de salida. Por cada cinta existe un cabezal, y sigue habiendo una sola unidad de control.

Para toda MT M1 con K cintas, existe una MT M2 equivalente con una sola cinta.

## Máquina de Turing con un solo estado: Para toda MT M1 con varios estados existe una MT M2 equivalente con un solo estado.

## Máquina de Turing no determinística: La elección de por cuál terna (q’, x’, d) a partir de (q, x) la MT no determinística (o MTN) continúa, es como el nombre del modelo lo indica *no determinística*. Ahora se considera una relación de transición ∆, en lugar de una función de transición δ. Una manera de interpretar cómo trabaja una MTN M es suponer que todas sus computaciones o secuencias de pasos se ejecutan en paralelo. Asumiendo que tiene los estados qA y qR, M acepta una entrada w si a partir de w *al menos una computación se detiene en qA*. En caso contrario, es decir si todas las computaciones de M terminan en el estado qR o son infinitas, M rechaza w.



En la figura, los términos *sí, no* y *loop*, indican que la computación correspondiente se

detiene en qA, se detiene en qR, o no se detiene, respectivamente. En este caso la MTN

*acepta la entrada w porque al menos una de sus computaciones la acepta.*

Una **MT determinística** (o MTD) es un caso particular de MTN, tal que su relación de

transición tiene grado 1. Para toda MTN M1 existe una MTD M2 equivalente.

Jerarquía de la computabilidad

Hay una jerarquía de clases de lenguajes (o problemas de decisión), teniendo en cuenta si son reconocidos y de qué manera por una máquina de Turing.

## Lenguajes recursivamente numerables y recursivos

Un lenguaje es recursivamente numerable si y sólo si existe una *MT que lo reconoce*.

Es decir, si **L** es el conjunto de todos los lenguajes (cada uno integrado por cadenas

finitas de símbolos pertenecientes a un alfabeto universal Ʃ), sólo los lenguajes

recursivamente numerables de L son reconocibles por una MT. A los problemas de decisión asociados se los conoce como **computables**. La clase de los lenguajes recursivamente numerables se denomina **RE**. El nombre se debe a que las cadenas de estos lenguajes se pueden *enumerar*.

Dado L ∈ RE, si M es una MT tal que L(M) = L, se cumple para toda cadena w de Ʃ\* que:

* Si w ∈ L, entonces M a partir de w se detiene en su estado qA
* Si w ∉ L, entonces M a partir de w se detiene en su estado qR o no se detiene.

No todos los lenguajes son recursivamente numerables, y sólo algunos tienen la propiedad de que las MT que los reconocen se detienen siempre. Un lenguaje es recursivo si y sólo

si existe una *MT M que lo reconoce y que se detiene cualquiera sea su entrada*. La clase

de los lenguajes recursivos se denomina **R**. A los problemas de decisión asociados se los

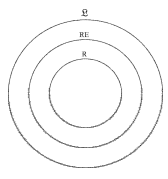
conoce como **decidibles**, porque las MT que los resuelven pueden justamente decidir,

cualquiera sea la instancia, si es positiva o negativa.

Dado L ∈ R, si M es una MT tal que L(M) = L, se cumple para toda cadena w de Ʃ\* que:

* Si w ∈ L, entonces M a partir de w se detiene en su estado qA.
* Si w ∉ L, entonces M a partir de w se detiene en su estado qR.

Se cumple por definición que R ⊆ RE ⊆ L. Es decir que no todos los problemas computables son decidibles, y que no todos los problemas son computables.

.

## Teorema 2.1 - Propiedades de clausura de la clase R

La clase R es cerrada con respecto las operaciones de complemento, intersección, unión y concatenación de lenguajes.

### Ejemplo. Prueba de que R es cerrada respecto al complemento:

Dado un lenguaje L, su lenguaje complemento es LC = {w | w ∈ Ʃ\* ∧ w ∉ L}. Si L ∈ R, entonces también LC ∈ R.

Idea general

Dado un lenguaje recursivo L, sea M una MT que lo acepta y se detiene siempre, es

decir a partir de cualquier entrada. Se va a construir una MT MC que acepta LC y se

detiene siempre, de la siguiente manera: dada una entrada w, si M se detiene en qA,

entonces MC se detiene en qR, y viceversa.

Construcción de la MT MC

Si M = (Q, Ʃ, Γ, δ, q0, qA, qR), entonces MC= (Q, Ʃ, Γ, δ’, q0, qA, qR), con δ y δ’ idénticas salvo en la aceptación y rechazo. Para todo estado q de Q, todo par de símbolos xi y xk de Γ, y todo movimiento d del conjunto {L, R, S}, se define:

* Si δ(q, xi) = (**qA**, xk, d), entonces δ’(q, xi) = (**qR**, xk, d)
* Si δ(q, xi) = (**qR**, xk, d), entonces δ’(q, xi) = (**qA**, xk, d)

Prueba de que MC se detiene siempre

* w ∈ LC → w ∉ L → con entrada w, **M se detiene en qR** → con entrada w, **MC se detiene en qA**.
* w ∉ LC → w ∈ L → con entrada w, **M se detiene en qA** → con entrada w, **MC se detiene en qR.**

Prueba de L(MC) = LC

w ∈ L(MC) ↔ con entrada w, **MC se detiene en qA** ↔ con entrada w, **M se detiene en qR** ↔ w ∉ L ↔ w ∈ LC

Esta es una típica prueba por construcción de que un lenguaje L es recursivo. Se construye una MT M, y se prueba que M se detiene siempre y acepta L.

### Ejemplo. Prueba de que R es cerrada respecto a la intersercción

Idea general

Sean M1 una MT que acepta L1 y se detiene siempre, y M2 una MT que acepta L2 y se

detiene siempre. Se va a construir una MT M que acepta L1 ⋂ L2 y se detiene siempre,

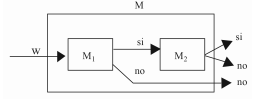
con las siguientes características. M simula primero M1 y luego M2. Dada una entrada

w, si M1 se detiene en su estado qR, entonces directamente M se detiene en su estado qR.

En cambio, si M1 se detiene en su estado qA, entonces con la misma entrada w, la MT M

simula M2, y se detiene en su estado qA (respectivamente qR) si M2 se detiene en su

estado qA (respectivamente qR).



## Teorema 2.2 - Propiedades de clausura de la clase RE

La clase RE es cerrada con respecto las operaciones de intersección, unión y concatenación de lenguajes. A diferencia de la clase R, RE no es cerrada con respecto al complemento.

### Ejemplo. Prueba de que RE es cerrada respecto a la unión: RE es cerrada con respecto a la unión de lenguajes, es decir que si L1 ∈ RE y L2 ∈ RE, entonces L1 ⋃ L2 ∈ RE.

Idea general

Sean M1 una MT que acepta L1 y M2 una MT que acepta L2 (ahora sólo se puede

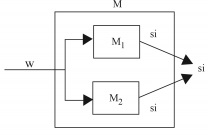
asegurar que estas MT se detienen en los **casos de aceptación**). Se va a construir una

MT M que acepta L1 ⋃ L2. No sirve que la MT M simule primero M1 y luego M2,

porque de esta manera M no acepta las cadenas de M2 a partir de las que M1 no se

detiene. El mismo problema ocurre simulando primero M2 y después M1. La solución es

otra: *se simulan “en paralelo”* las MT M1 y M2, y se acepta si alguna de las dos MT acepta.



## Teorema 2.3 - R = RE ⋂ CO-RE

Se prueba fácilmente que R ⊆ RE ⋂ CO-RE.

La inclusión R ⊆ RE se cumple por definición.

También vale R ⊆ CO-RE porque L ∈ R → LC ∈ R → LC ∈ RE → L ∈ CO-RE.

También se cumple la inversa, es decir, RE ⋂ CO-RE ⊆ R.

---------------------------------------------------------------------------------------------------------------------------

En la jerarquía de la computabilidad se distinguen cuatro categorías de lenguajes. Enumeradas de acuerdo a su dificultad creciente, son:

1. **R**
2. **RE – R**
3. **CO-RE – R**
4. **L – (RE ⋃ CO-RE)**

Dado un par cualquiera de lenguajes L y LC , se cumple alternativamente que:

* Tanto L como LC pertenecen a R.
* L pertenece a RE – R, y LC pertenece a CO-RE – R.
* Tanto L como LC pertenecen a L – (RE ⋃ CO-RE).

Por el Teorema 2.1, si L está en R también lo está LC.

Más adelante vamos a encontrar un primer ejemplo de lenguaje L en RE – R, con lo que probaremos que R ⊂ RE, o en otras palabras, que **hay problemas computables que no son decidibles**.

En este caso LC pertenece a CO-RE – R, de acuerdo también al Teorema 2.1. Notar que con la existencia de LC se prueba además que RE ⊂ L , de acuerdo al Teorema 2.3, o en otras palabras, que **hay problemas que no son computables**.

Indecibilidad

Los *problemas indecidibles* están asociados a los lenguajes no recursivos, los lenguajes de L – R. Presentaremos un primer *lenguaje recursivamente numerable que no es recursivo*, que completa la formalización de la jerarquía de la computabilidad definida en la clase anterior. Para ello nos valemos de la técnica de ***diagonalización***. Luego encontraremos más representantes del conjunto L – R pero con otra técnica, la ***reducción de problemas***.

En diagonalizaciones y reducciones se utilizarán máquinas de Turing universales.

En este caso, identificando con U a la MT universal, U cumple que:

* Sus entradas son pares de la forma (<M>, w), siendo <M> la codificación de una MT M, y w una entrada, también codificada, de M.
* Dada la entrada (<M>, w), su ejecución consiste en simular M a partir de w.

La MT universal U es como toda MT una tupla (Q, Ʃ, Γ, δ, q0, qA, qR), con sus propios

estados y símbolos, y una función de transición δ que establece cómo simular la MT

cuyo código forma parte de su entrada.

Una posible manera de codificar una MT es la siguiente:

* Los estados del conjunto Q = {q1, ..., qk} se representan por los números 1, ..., k, en notación binaria. Además, qA y qR se representan en binario por los números k + 1 y k + 2, respectivamente.
* Los símbolos del alfabeto Γ = {xi1, ..., xin} se representan por los números i1,..., in, en notación binaria.
* Los movimientos L, R y S se representan, respectivamente, por los números 1, 2 y 3, en notación binaria.
* Finalmente, la función de transición δ se representa por una secuencia de 5- tuplas. Las tuplas se separan por el símbolo numeral, es decir #, y sus componentes por medio de una coma.

La codificación de una MT M consiste en la cadena formada por ***|Q|***, en notación binaria, seguida por el separador***#***y luego por la representación de la ***función de transición δ***. Se antepone |Q| para que se puedan identificar los estados qA y qR.

Ejemplo

Si M = ({q1}, {x4, x5}, {x1, x4, x5}, δ, q1, qA, qR) y

la función de transición se define con δ(q1, x4) = (qA, x1, S) y δ(q1, x5) = (qR, x1, R),

El código de la MT M sería → **<M> = 1#1,100,10,1,11#1,101,11,1,10**

## Teorema 3.1. El problema de la detención es computable no decidible

El problema de la detención (de las MT), también conocido como HP (*halting problem*), es un problema que se enuncia de la siguiente manera: dada una MT M y una cadena w, ¿M se detiene a partir de w?.

El lenguaje que lo representa es HP = {(<M>, w) | la MT M se detiene a partir de w}.

Se cumple que **HP ∈ RE – R**. Es decir, **HP ∈ RE**, y **HP ∉ R**.

### Prueba de que HP ∈ RE.

Sea la siguiente MT MHP , que a partir de una entrada v, hace:

* Si v no es un código válido (<M>, w), rechaza.
* Simula M a partir de w. Si M se detiene, acepta.

Se cumple que L(MHP) = HP: v ∈ L(MHP) ↔ MHP acepta v ↔ v = (<M>, w) y M se detiene a partir de w ↔ v ∈ HP.

Es decir, el lenguaje reconocido por MHP es **HP**. Si V pertenece a dicho lenguaje, la máquina MHP acepta v.

### Prueba de que HP ∈ R → D ∈ R.

Si el lenguaje HP fuera recursivo, también lo sería el lenguaje D, dado que D es un caso

particular de HP, que en lugar de considerar todos los pares (<Mi>, wk) sólo considera los pares **(<Mi>, wi).** Más precisamente, si HP ∈ R, entonces existe una MT MHP que se

detiene siempre y reconoce HP. Para probar que D ∈ R, se debería construir una MT MD

que se detiene siempre y reconoce D.

------------------------------------------------------------------------------------------------------------------------

La técnica de **diagonalización** es muy útil para probar que dos conjuntos difieren en al menos un elemento, al que se lo suele denominar *separador*.

El separador que existe entre las clases RE y R es el lenguaje **HP** (en realidad también el lenguaje D). La idea de la diagonalización es muy sencilla. En una de sus variantes, se toma como base una *tabla de unos y ceros* y se utiliza el hecho de que “el complemento a dos” de su diagonal difiere de todas las filas: difiere de la primera fila en el primer elemento, de la segunda fila en el segundo elemento, y así sucesivamente.

Con ésta técnica se puede probar, por ejemplo:

* Que los números reales y el conjunto de partes de los números naturales no son numerables.
* Que el conjunto de partes de los números naturales no es numerable.
* Que RE ∈ L. (Se puede encontrar el lenguaje EC como separador)

Hasta el momento, contamos con distintos caminos para formalizar los límites de la computabilidad, es decir para probar la inclusión estricta RE ⊂ L . Por ejemplo, HPC ∉ RE, porque de lo contrario HP sería recursivo.

Un lenguaje de RE - R es Lu = {(<Mi>, wk) | Mi acepta wk}.

El lenguaje Lu es recursivamente numerable, y no es recursivo porque de lo contrario el lenguaje E sería recursivo.

Lu representa el problema de la pertenencia de una cadena a un lenguaje, o directamente el **problema de la pertenencia**. Se lo conoce como lenguaje universal.

Reducciones de problemas

Ahora pondremos nuestro análisis en los problemas indecidibles, haciendo uso de la técnica de **reducción de problemas**, que nos permitirá encontrar lenguajes no recursivos y no recursivamente numerables de una manera en general mucho más sencilla que por medio de la diagonalización.

La noción de reducción de problemas es muy simple: para resolver un problema se lo relaciona con otro, que se sabe cómo resolverlo; a partir de este conocimiento se resuelve el problema original.

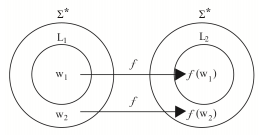
## Definición

Sean L1 y L2 dos lenguajes incluidos en Ʃ\*. Existe una reducción del lenguaje L1 al lenguaje L2, si y sólo si existe una función total computable

**f: Ʃ\* → Ʃ\*** tal que **∀w ∈ Ʃ\*: w ∈ L1 ↔ f(w) ∈ L2.**

La función f se denomina **función de reducción**. Que f sea **total computable** significa que existe una MT que a partir de cualquier cadena w computa f(w) en su cinta de salida y se detiene. En general, identificaremos con Mf a la MT que computa f.

Que haya una reducción de L1 a L2 significa, entonces, que existe una MT que transforma toda cadena de L1 en una cadena de L2, y toda cadena no perteneciente a L1 en una cadena no perteneciente a L2.



Se va a utilizar la notación **L1 α L2** para expresar que existe una reducción del lenguaje (o

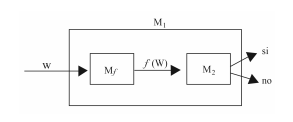
problema) L1 al lenguaje (o problema) L2.

## Teorema 4.1. Si L2 está en R (o RE) y L1 α L2, entonces L1 está en R (o RE)

Probaremos que si L2 ∈ R y L1 α L2, entonces L1 ∈ R.

Idea general

Componiendo la MT Mf que computa la función de reducción f del lenguaje L1 al lenguaje L2, con la MT M2 que reconoce el lenguaje L2 y se detiene siempre, se obtiene una MT M1 que reconoce el lenguaje L1 y se detiene siempre.



Construcción de la MT M1

Sea Mf una MT que computa la función de reducción f, con w ∈ L1 ↔ f(w) ∈ L2, y sea M2 una MT que reconoce L2 y se detiene siempre. La MT M1 trabaja de la siguiente manera:

* Simula Mf a partir de la entrada w y obtiene f(w).
* Simula M2 a partir de f(w) y acepta si y sólo si M2 acepta.

Prueba de que M1 se detiene siempre

La MT M1 se detiene siempre porque Mf y M2 se detienen siempre.

Prueba de L1 = L(M1)

* w ∈ L1 → Mf a partir de w computa f(w) ∈ L2 → M2 a partir de f(w) se detiene en su estado **qA** → M1 a partir de w se detiene en su estado **qA** → w ∈ L(M1).
* w ∉ L1 → Mf a partir de w computa f(w) ∉ L2 → M2 a partir de f(w) se detiene en su estado **qR** → M1 a partir de w se detiene en su estado **qR** → w ∉ L(M1).

### Conclusión: si L1 no es recursivo y existe una reducción de L1 a L2, entonces L2 tampoco es recursivo (de lo contrario L1 sería recursivo). Lo mismo se puede decir para el caso de los lenguajes recursivamente numerables.

Por lo tanto, las reducciones se pueden emplear también para probar que un lenguaje no

es recursivo o no es recursivamente numerable

## Ejemplo. Reducción de HP a LU

Se probó por diagonalización que el lenguaje recursivamente numerable HP = {(<M>, w) | M se detiene a partir de w}, que representa el *problema de la detención,* no es recursivo.

También a partir de una diagonalización se estableció la no recursividad de otro

lenguaje clásico, recursivamente numerable, de la computabilidad, el lenguaje LU o

lenguaje universal, definido por LU = {(<M>, w) | M acepta w}, que representa el

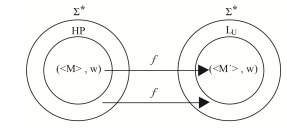
*problema de la pertenencia.*

Una manera alternativa de **probar que LU no es recursivo**, aplicando el corolario del

teorema anterior, es construir directamente una reducción de HP a LU.

Definición de la función de reducción

Si la entrada es sintácticamente válida se define f((<M>, w)) = (<M’>, w) tal que M’ se comporta como M, salvo que cuando M se detiene, ya sea en el estado qA o en el estado qR, **M’ acepta**

****

La función f es total computable

Si la entrada es una cadena **inválida** (<M>, w), la MT Mf genera la *cadena 1*. En caso

contrario, para generar <M’>, Mf modifica las 5-tuplas de <M> **reemplazando todo**

**estado qR por el estado qA.**

Se cumple (<M>, w) ∈ HP ↔ f((<M>, w)) ∈ LU

(<M>, w) ∈ HP ↔ M se detiene a partir de w ↔f(w) = (<M’>,w)↔ M’ acepta w ↔ (<M’>, w) ∈ LU.

### Conclusión: para probar que LU no es recursivo, se construyó una reducción desde un lenguaje no recursivo, HP, a LU. LU no puede ser recursivo, porque si lo fuera también lo sería HP (absurdo). Combinando Mf con una supuesta MT que acepta LU y se detiene siempre, se obtendría una MT que acepta HP y se detiene siempre.

Probando que existe una reducción de L1 a L2, se demuestra que *el lenguaje L2 es tan o más difícil que el lenguaje L1, desde el punto de vista de la computabilidad*.

Si no se hubiera tenido indicios acerca de la no recursividad del lenguaje LU, una primera aproximación para determinar la ubicación de LU en la jerarquía de la computabilidad podría haber sido la siguiente:

* Construir una MT que a partir de (<M>, w) simula M a partir de w.
* Como M puede no detenerse a partir de w, entonces establecer que LU no es recursivo.

Obviamente esta prueba no es correcta, pero de todos modos un primer intento de esta

naturaleza puede servir para orientar la demostración hacia una prueba “*positiva*” de un

determinado lenguaje L (la construcción de una MT que pruebe que **L ∈ R**), o

“*negativa*” (una reducción que pruebe que **L ∉ R**).

## Ejemplo. Reducción de LU a HP

## 

Definición de la función de reducción

Para pares válidos (<M>, w), se define f((<M>, w)) = (<M’>, w) tal que M’ se comporta como M, salvo que cuando M **se detiene en qR**, M’ **no se detiene**.

La función f es total computable

Si la entrada es una cadena **inválida** (<M>, w), la MT Mf genera la cadena 1. En caso

contrario, para generar <M’> la MT Mf modifica <M> de modo tal que M’ entre en un

**loop** cuando M se detiene en **qR**: por ejemplo, como ya se vio en el Teorema 3.1, se

puede reemplazar qR por un estado nuevo q y definir una 5-tupla (q, x, q, x, S) por cada

símbolo x del alfabeto de M.

Se cumple (<M>, w) ∈ LU ↔ f((<M>, w)) ∈ HP

(<M>, w) ∈ LU ↔ M acepta w ↔ M’ se detiene a partir de w ↔ (<M’>, w) ∈ HP.

---------------------------------------------------------------------------------------------------------------------------

Anteriormente, en la prueba de que el lenguaje HP no es recursivo (Teorema 3.1), se utilizó un lenguaje similar D = {wi | Mi se detiene a partir de wi}. Se demostró primero que **HP ∈**

**R → D ∈ R**. Notar que esta prueba no es sino una **reducción de D a HP**. Lo mismo sucedió con **LU** en relación al lenguaje similar **E**.

HP y LU están entre los lenguajes más difíciles de la clase RE, en el sentido de la

computabilidad. Más precisamente, se prueba que todos los lenguajes recursivamente

numerables se reducen a ellos. HP y LU son lenguajes **RE-completos**.

Veámoslo para el caso de LU. Si **L ∈ RE** y M es una MT que reconoce L, entonces la función f definida de la siguiente manera: f(w) = (<M>, w) para toda cadena w, es claramente una **reducción de L a LU**.

En cambio, en la clase **R** se cumple que, sin considerar los lenguajes Ʃ\* y ∅, *cualquier*

*lenguaje recursivo L1 se puede reducir a cualquier lenguaje recursivo L2*, es decir que,

siguiendo con la terminología anterior, todo lenguaje recursivo es **R-completo**.

Si L1 y L2 son dos lenguajes recursivos distintos de Ʃ\* y ∅, **a ∈ L2,** **b ∉ L2**, y M1 y M2 se

detienen siempre y reconocen L1 y L2, respectivamente, entonces la función f definida

de la siguiente manera:

f(w) = a, si w ∈ L1

f(w) = b, si w ∉ L1

para toda cadena w, es claramente una reducción de L1 a L2.

Las reducciones poseen las propiedades de *reflexividad* y *transitividad*, y que en cambio no son *simétricas*, porque por lo visto recién, **todo lenguaje recursivo L se reduce a LU**, y en cambio **no puede existir una reducción de LU a L**, porque de lo contrario *LU sería recursivo.* El hecho de que haya una reducción de L1 a L2 no implica que haya una reducción en el sentido contrario. Para el caso de HP y LU vimos que se cumple, pero no es lo que sucede en general. general. Otra propiedad útil para considerar es que existe una reducción de L1 a L2 si y sólo si existe una reducción de L1C a L2C (es la misma función de reducción).

## Teorema 4.2. Teorema de Rice

Vamos a demostrar que **si P es un subconjunto de RE**, con ∅ ⊂ P ⊂ RE, entonces el

lenguaje L = {<M> | L(M) ∈ P } no es recursivo. P representa una *propiedad no trivial*

*de RE*. Intuitivamente, es razonable que L no sea recursivo: probar que los lenguajes

L(M) cumplen una determinada propiedad requiere chequear, en cada caso, el

comportamiento de M sobre las infinitas cadenas de Ʃ\*.

Supongamos primero que ∅ ∉ P . Sea L1 ≠ ∅ un lenguaje de P, y M1 una MT que

reconoce L1. Se va a construir una reducción de LU α LP, y de esta manera se probará

que LP ∉ R.

Definición de la función de reducción

Para pares válidos (<M>, w) se define f((<M>, w)) = <Mw> donde Mw es una MT que a partir de una entrada v:

* Simula M a partir de w, y si M no acepta, entonces no acepta.
* Simula M1 a partir de v, y acepta si y sólo si M1 acepta.

Se comprueba fácilmente que

* Si M acepta w, entonces L(Mw) = L1 (que pertenece a P).
* Si M no acepta w, entonces L(Mw) = ∅ (que no pertenece a P).

La función f es total computable.

Si la entrada no es una cadena válida (<M>, w), la MT Mf genera la cadena 1 (por

convención en este caso L(Mw) = ∅). En caso contrario, Mf genera <Mw> básicamente

agregándole a <M> el código <M1> más un fragmento relacionado con la simulación de

M a partir de w y de M1 a partir de v.

Se cumple (<M>, w) ∈ LU ↔ f((<M>, w)) ∈ LP

(<M>, w) ∈ LU ↔ M acepta w ↔ L(Mw) = L1 ∈ P ↔ <Mw> ∈ LP.

---------------------------------------------------------------------------------------------------------------------------

El problema de determinar si una MT reconoce el lenguaje vacío es otro problema clásico de la computabilidad. Para completar su ubicación en la jerarquía vamos a probar que L∅ no es ni siquiera recursivamente numerable.

Jerarquía de la complejidad temporal

Caracterizados los problemas decidibles en el marco de la jerarquía de la computabilidad, ahora se los va a estudiar desde el punto de vista de los recursos que requieren las máquinas de Turing para resolverlos. Nos referiremos fundamentalmente al **tiempo**, es decir a la cantidad de pasos efectuados por las MT.

A las medidas de complejidad de este tipo, relacionadas con las computaciones de las MT, se las conoce como **dinámicas**. Otra medida dinámica es la cantidad de veces que en la ejecución de una MT con una sola cinta, su cabezal cambia de dirección.

El tópico de las medidas dinámicas de complejidad, de las que el tiempo y el espacio son las más populares e intuitivas, es lo que trata la **complejidad computacional.**

Una aproximación diferente para el estudio de la complejidad se basa en las medidas **estáticas**, en cuyo caso se hace referencia a la **complejidad estructural** de los problemas.

Se presentarán las definiciones básicas de la complejidad computacional temporal (o directamente complejidad temporal), y la jerarquía de clases de problemas asociada, conocida como jerarquía de la complejidad temporal (o directamente jerarquía temporal).

## Conceptos básicos de la complejidad temporal

Como las instancias de un problema pueden ser de cualquier tamaño, y en general una

MT que lo resuelve *tarda más* (efectúa más pasos) a medida que las entradas son más

grandes, resulta natural definir el tiempo de ejecución en términos del *tamaño de las*

*entradas*. Es decir, la idea es medir el tiempo de ejecución de una MT a partir de una entrada w, con |w| = n, por medio de una función T(n), y analizar la razón de crecimiento de la función temporal, categorizando la complejidad del problema asociado según si la MT que lo resuelve trabaja en tiempo *lineal, polinomial, exponencial, doble exponencial,* etc.

En lugar de funciones T se utilizan funciones O(T), que se leen *“orden de T”*.

La expresión **O(T)** denota el conjunto de todas las **funciones f que cumplen f(n) ≤ c.T(n),** para toda constante c > 0 y todo número natural n ≥ 0.

Dada la función T: N → N, se define que *una MT M trabaja en tiempo T(n) si y sólo si para toda entrada w, con |w| = n, M hace a lo sumo T(n) pasos*, en su única computación si es determinística, o en cada una de sus computaciones si es no determinística. De modo similar se define *una MT que trabaja en tiempo O(T(n))*. Se asume que una MT hace siempre al menos n + 1 pasos, para leer toda su entrada.

Los problemas que pueden ser resueltos por MT que *trabajan en tiempo O(T(n)*) se

agrupan en una *misma clase*: **un problema** (o lenguaje) **pertenece a la clase DTIME(T(n)**) (respectivamente NTIME(T(n))) **si y sólo si existe una MTD** (respectivamente MTN), **con una o más cintas, que lo resuelve** (o reconoce) **en tiempo O(T(n)).**

------------------------------------------------------------------------------------------------------------------------

Si un problema pertenece a la clase DTIME(T(n)) (respectivamente NTIME(T(n))), cualquiera sea la instancia w considerada la respuesta de la MTD (respectivamente MTN) que lo resuelve no tarda más de O(T(|w|)) pasos. Esto significa que el criterio de medición es por el **peor caso**. Otro criterio es por el **caso promedio**, pero para ello se necesita conocer cómo se distribuyen las entradas. Por otro lado, que un problema pertenezca a una clase temporal no implica que no tenga un tiempo de resolución menor. El ideal por supuesto es determinar la cota temporal mínima, establecer la complejidad intrínseca de un problema, independientemente de los algoritmos que se conozcan, pero este objetivo es difícil en general. Lo habitual es identificar la complejidad temporal de un problema con la del algoritmo encontrado más eficiente para resolverlo. Trataremos esta cuestión a lo largo de las próximas clases, con foco en clases específicas de problemas.

Cabe recordar que en la primera parte del libro se mostró que pasar de una MT con varias cintas a una MT equivalente con una cinta puede aumentar el tiempo de trabajo en el orden cuadrático, es decir **O(n2)**. Se puede demostrar también que reduciendo la cantidad de cintas a dos, el retardo disminuye a **O(n.log2 n).**

En el conjunto R de los lenguajes recursivos o problemas de decisión decidibles se

distinguen dos clases temporales, P y NP, que se definen de la siguiente manera:

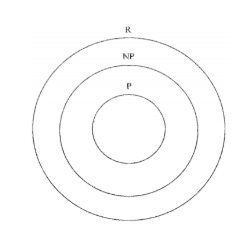
P = ⋃i ≥ 0 DTIME(ni)

NP = ⋃i ≥ 0 NTIME(ni)

La clase **P** agrupa a los problemas que se resuelven en tiempo determinístico polinomial, y **NP** es la clase de los problemas que se resuelven en tiempo no determinístico polinomial.

Se cumple por definición que P ⊆ NP, y además que las dos clases están incluidas estrictamente en R.

Sólo podemos afirmar que “se sospecha” que P ≠ NP.



Se considera a P la clase de los problemas **tratables**, en el sentido de que si bien todos

los problemas de R son decidibles, las resoluciones que consumen más que tiempo determinístico polinomial no se consideran aceptables.

R – P queda como la clase de los problemas **intratables**, siendo entonces la frontera de la clase P la que separa los problemas con *resolución temporal aceptable* o eficiente de los de *resolución temporal inaceptable* o ineficiente.

Asumiremos que lo que no es polinomial es exponencial.

Identificaremos de ahora en más con **EXP** (por tiempo exponencial) a la clase de todos los **problemas decidibles**, es decir **R**.

Un problema que se resuelve en tiempo *polinomial* con una MTD con K1 cintas, también se resuelve en tiempo polinomial con una MTD con K2 cintas, cualesquiera sean K1 y K2. En este sentido, otro modelo razonable lo constituyen las máquinas *RAM*. En cambio, el modelo de las MTN no es razonable: **podría darse la inconsistencia de que un problema con resolución exponencial mediante una MTD, tuviera resolución polinomial con una MTN.**

------------------------------------------------------------------------------------------------------------------------

**Características de la jerarquía temporal.**

Si bien T1(n) = O(T2(n)) implica que DTIME(T1(n)) ⊆ DTIME(T2(n)), esta inclusión no es necesariamente estricta.

El **Teorema de Aceleración Lineal** (Linear Speed Up Theorem), en una de sus variantes, establece que si existe una MT M1 con K cintas que trabaja en tiempo T(n), entonces existe una MT M2 con K + 1 cintas equivalente que trabaja en tiempo c.T(n) + (n + 1), para c = 1/2, 1/4, 1/8, ..., es decir 1/2d , siendo d cualquier número natural mayor que cero.

Una clase DTIME(T2(n)) no incluye más problemas que una clase DTIME (T1(n)), si es que T2(n) difiere de T1(n) en un factor constante.

**Teorema 6.1. Teorema de la jerarquía temporal**

Si T2(n) > T1(n).logT1(n) para infinitos n, entonces DTIME(T1(n)) ≠ DTIME(T2(n)) siendo T2(n) una función tiempo-construible.

Como corolario de otro resultado en el marco de la jerarquía temporal, conocido como

el **Teorema de la Unión**, existe una función total computable T(n) tal que DTIME(T(n)) = P. De esta manera, por el segundo resultado del teorema anterior se deduce que la clase P está incluida estrictamente en la clase EXP. De igual modo se prueba la inclusión estricta de NP en EXP.

Las clases P y NP

Se presentan a continuación tres problemas clásicos de resolución determinística polinomial, de la teoría de grafos, la aritmética y la lógica, no tan simples como los mencionados antes. Para distinguirla de P, llamaremos **FP** a la clase de las funciones que se calculan en tiempo determinístico polinomial.

## 

## El problema del camino mínimo en un grafo está en P (TSP)

El problema (de decisión) del camino mínimo en un grafo consiste en determinar si en un grafo existe un camino entre dos vértices v1 y v2, de longitud a lo sumo K.

Un grafo se representará por un par (V, E) como se describió previamente, utilizando **números en binario** para la identificación de los **vértices**, y el símbolo **# como separador**.

La idea general del algoritmo propuesto se basa en lo siguiente. Si *Ah(i, j)* es la *longitud* del camino mínimo entre los vértices i y j que no pasa por ningún vértice mayor que h, entonces se cumple



Naturalmente, el camino mínimo entre i y j que no pasa por ningún vértice mayor que h+1, pasa o no pasa por el vértice h+1. La siguiente MTD M, basada en la igualdad anterior, trabaja en tiempo polinomial y reconoce el lenguaje SP (por shortest path o camino mínimo) que representa el problema, siendo SP = {(G, v1, v2, K) | G es un grafo y tiene un camino entre sus vértices v1 y v2 de longitud a lo sumo K}. Dada una entrada w = (G, v1, v2, K), M obtiene Am (v1, v2), el camino mínimo en G entre v1 y v2, y acepta si y sólo si Am (v1, v2) ≤ K.

---------------------------------------------------------------------------------------------------------------------------

La clase P es cerrada con respecto a la unión, intersección y complemento, entre otras

operaciones entre lenguajes

## El problema del máximo común divisor está en FP

## El problema 2-SAT está en P.

## 

## El problema del circuito de Hamilton está en NP

Sea HC (por Hamiltonian circuit o circuito hamiltoniano) el lenguaje que representa el

problema, con HC = {G | G es un grafo que tiene un circuito de Hamilton}. El algoritmo

determinístico natural para reconocer HC chequea en el peor caso todas las permutaciones de V, para detectar si una de ellas es un circuito de Hamilton. Tarda tiempo exponencial: hay m! permutaciones de V, por lo que el tiempo de ejecución es al menos O(m!) = O(mn)

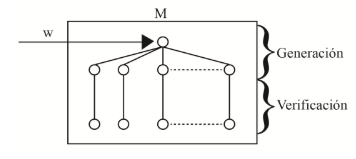
Para probar que HC está en NP, se debería construir una MTN M que reconoce HC en tiempo polinomial.}

---------------------------------------------------------------------------------------------------------------------------

La forma de una MTN M que trabaja en tiempo polinomial, construida para demostrar que L(M) pertenece a la clase NP, siendo L(M) la representación de un problema (de decisión), es siempre la misma. Dada una entrada w, M hace:

1. Genera no determinísticamente, en tiempo polinomial, una posible solución del problema (solución en el sentido más amplio: por ejemplo, en el caso del problema de primalidad, la cadena generada puede simplemente establecer que w es un número primo).

2. Chequea determinísticamente en cada computación, en tiempo polinomial, si lo generado en el paso 1 es efectivamente una solución del problema. El chequeo incluye la validación sintáctica de la entrada w, que se puede ejecutar al principio para optimizar el algoritmo.



## El problema del clique está en NP

Este problema consiste en determinar si un grafo tiene un clique de tamaño (al menos) K. Un clique de tamaño K en un grafo G es un subgrafo completo de G con K vértices.

---------------------------------------------------------------------------------------------------------------------------

Una definición alternativa de la clase NP, esta vez sin utilizar MTN, contribuye sobremanera a entender el tipo de problemas que la integran. Según dicha definición, un lenguaje L que pertenece a NP goza de la propiedad de que cada una de sus cadenas w tiene al menos un certificado suscinto z que atestigua su pertenencia a L. Se define que z es un certificado suscinto de w si:

• Es una cadena de tamaño polinomial con respecto al tamaño de w.

• El predicado R(w, z), que expresa que z es un certificado suscinto de w, se decide en tiempo determinístico polinomial.

Dicho de otra manera: para toda instancia positiva de un problema de NP existe al menos un certificado suscinto de que es positiva. Podemos no saber cómo encontrar el certificado eficientemente, pero seguro que existe. Por ejemplo, en el caso del problema del circuito de Hamilton, los grafos con circuitos de Hamilton tienen certificados suscintos: son los mismos circuitos de Hamilton. Otro ejemplo lo constituye el problema de primalidad, recién mencionado: todo número primo tiene un certificado suscinto de su primalidad.