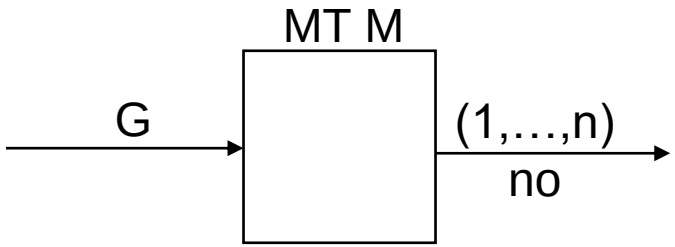
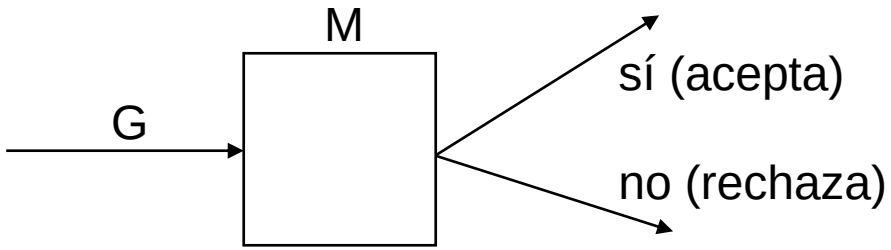
Fundamentos de Teoría de la Computación

Repaso clases 1 a 7

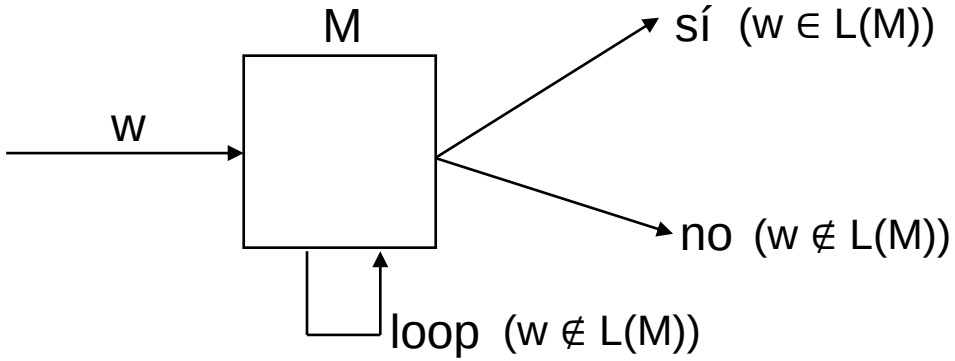
Clase 1. Máquina de Turing (MT)



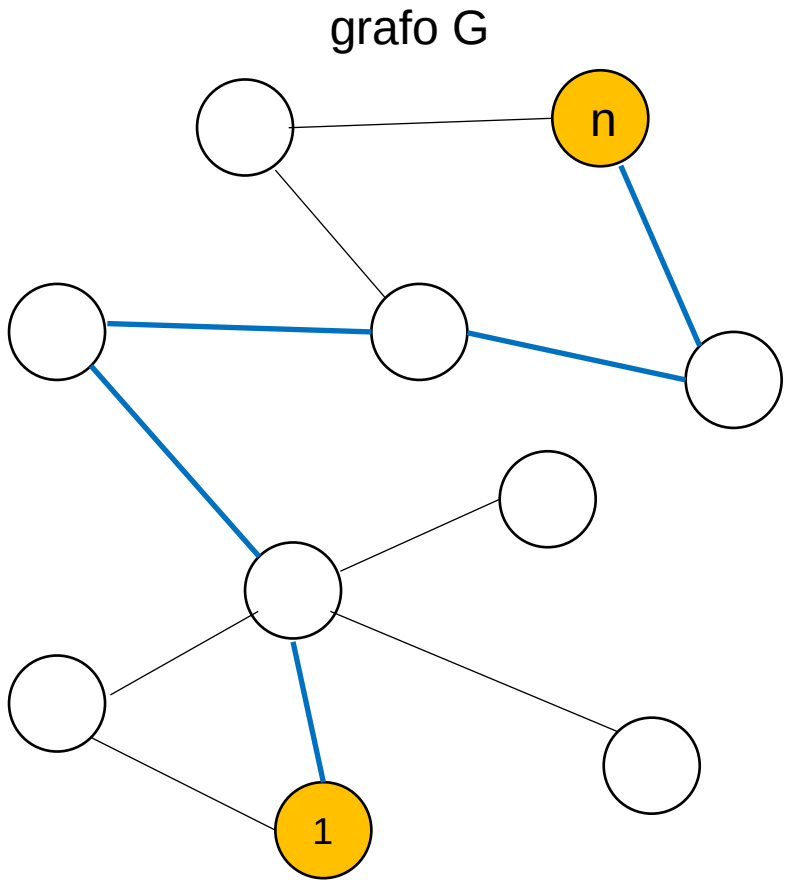
Problema de búsqueda



Problema de decisión (M acepta o reconoce un lenguaje)

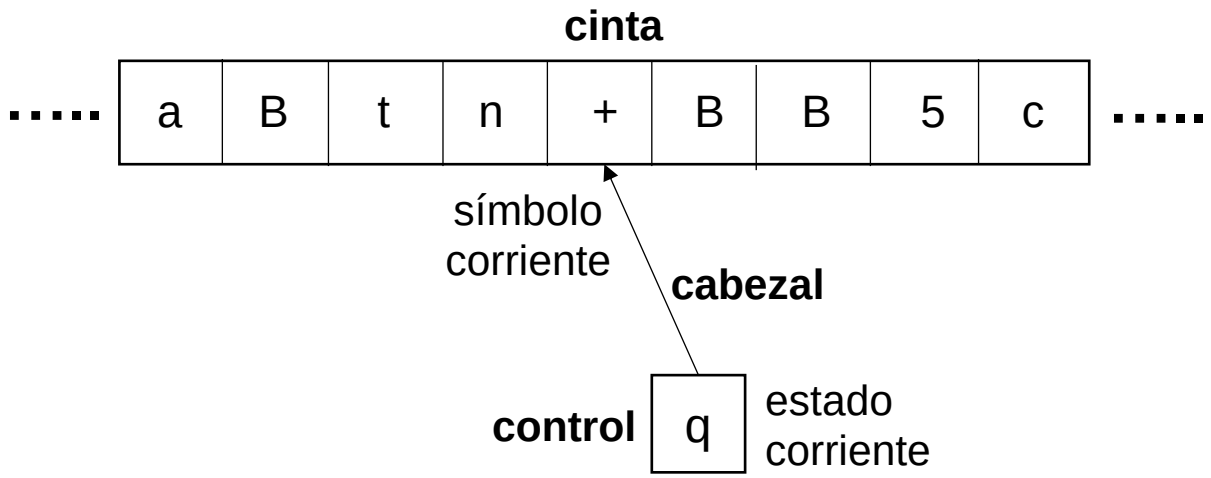


Posibilidad de no detención



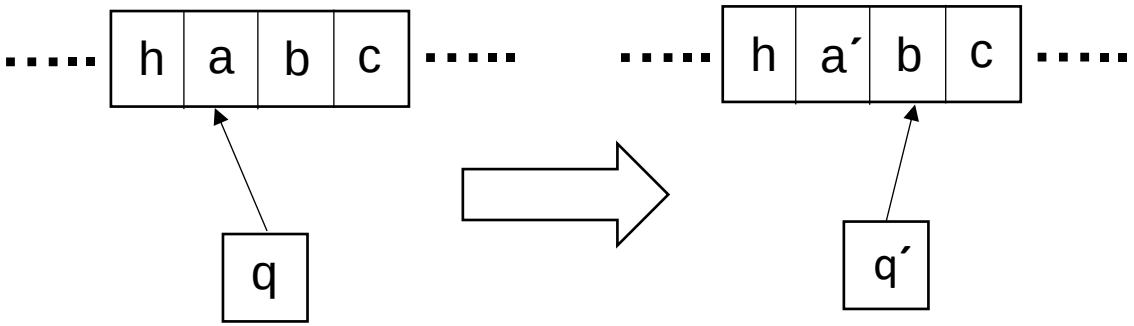
Grafo con un camino del vértice 1 al vértice n

Definición



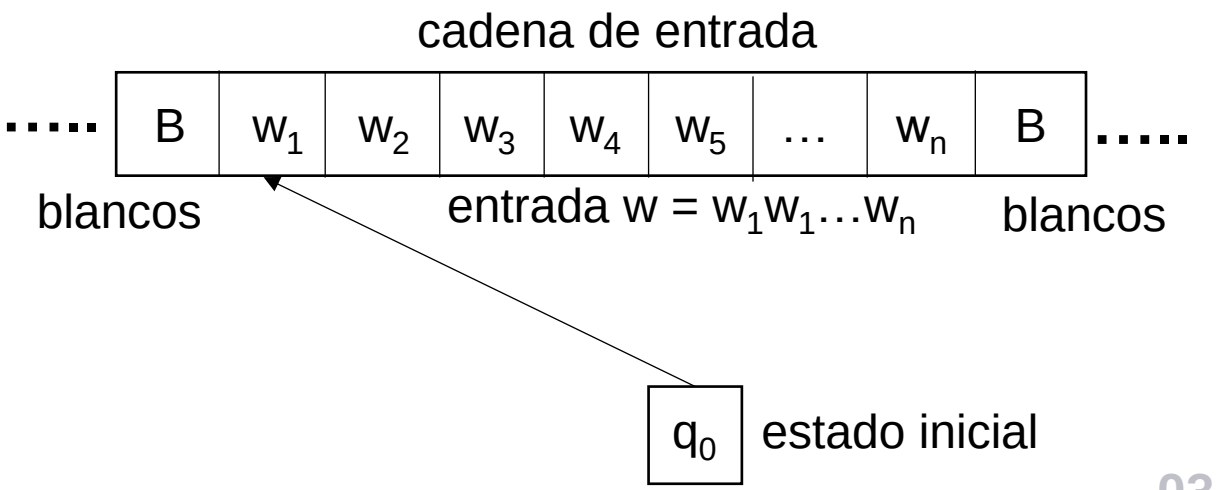
Tesis de Church-Turing
¡Todo lo computable puede ser computado por una Máquina de Turing!

En un paso, una MT puede modificar el símbolo corriente, el estado corriente, y moverse un lugar a derecha o izquierda. Por ejemplo:



un paso de la MT

Al inicio, la entrada se delimita por blancos, y el cabezal apunta al 1er símbolo de la izquierda:



Ejemplo ($L = \{a^n b^n \mid n \geq 1\}$)

Definición de la MT $M = (Q, \Gamma, \delta, q_0, q_A, q_R)$:

Estados $Q = \{q_0, q_a, q_b, q_L, q_H\}$

q_0 : estado inicial

q_a : M busca una *a*

q_b : M busca una b

q_L : M vuelve a buscar otra *a*

q_H : no hay más *a*

Alfabeto $\Gamma = \{a, b, \alpha, \beta, B\}$

Ayuda memoria del algoritmo con la entrada aaabbb:

aaabbb
αaabbb
αaaβbb
ααaβbb
ααaββb
αααββb
αααβββ

Función de transición δ:

1. $\delta(q_0, a) = (q_b, \alpha, R)$
2. $\delta(q_a, a) = (q_b, \alpha, R)$
3. $\delta(q_a, \beta) = (q_H, \beta, R)$
4. $\delta(q_b, a) = (q_b, a, R)$
5. $\delta(q_b, b) = (q_L, \beta, L)$
6. $\delta(q_b, \beta) = (q_b, \beta, R)$
7. $\delta(q_L, a) = (q_L, a, L)$
8. $\delta(q_L, \alpha) = (q_a, \alpha, R)$
9. $\delta(q_L, \beta) = (q_L, \beta, L)$
10. $\delta(q_H, \beta) = (q_H, \beta, R)$
11. $\delta(q_H, B) = (q_A, B, S)$

Todos los pares omitidos corresponden a rechazos, no se especifican para abreviar la descripción. Por ejemplo: $\delta(q_0, B)$, $\delta(q_0, b)$, $\delta(q_b, B)$, $\delta(q_H, b)$, etc.

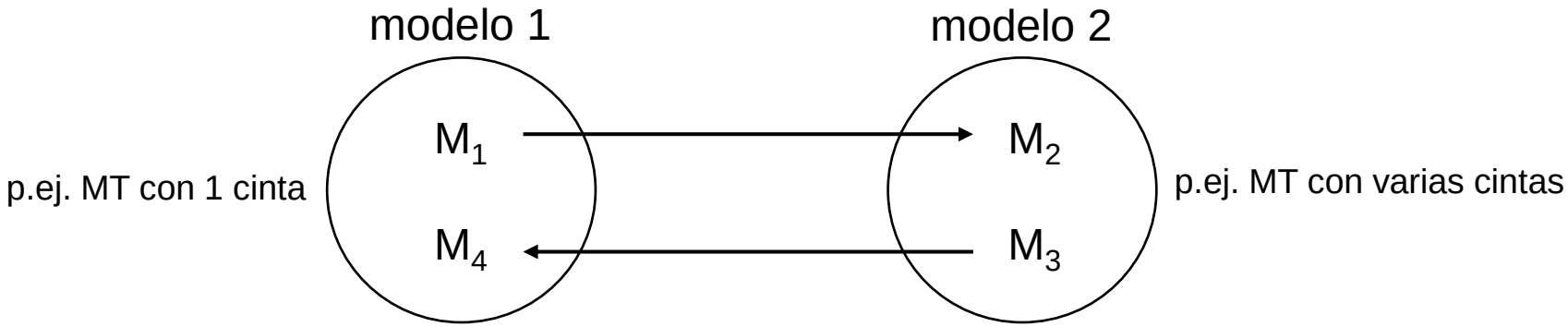
Forma alternativa de describir la función de transición δ

	a	**b**	**α**	**β**	**B**
$\mathbf{q_0}$	q_b, α, R				
$\mathbf{q_a}$	q_b, α, R			q_H, β, R	
$\mathbf{q_b}$	q_b, a, R	q_L, β, L		q_b, β, R	
$\mathbf{q_L}$	q_L, a, L		q_a, α, R	q_L, β, L	
$\mathbf{q_H}$				q_H, β, R	q_A, B, S

Las celdas en blanco representan los casos de rechazo (estado q_R).

Modelos equivalentes de MT

- Dos MT son **equivalentes** si aceptan el mismo lenguaje.
- Dos **modelos de MT** son **equivalentes** si dada una MT de un modelo existe una MT equivalente del otro.

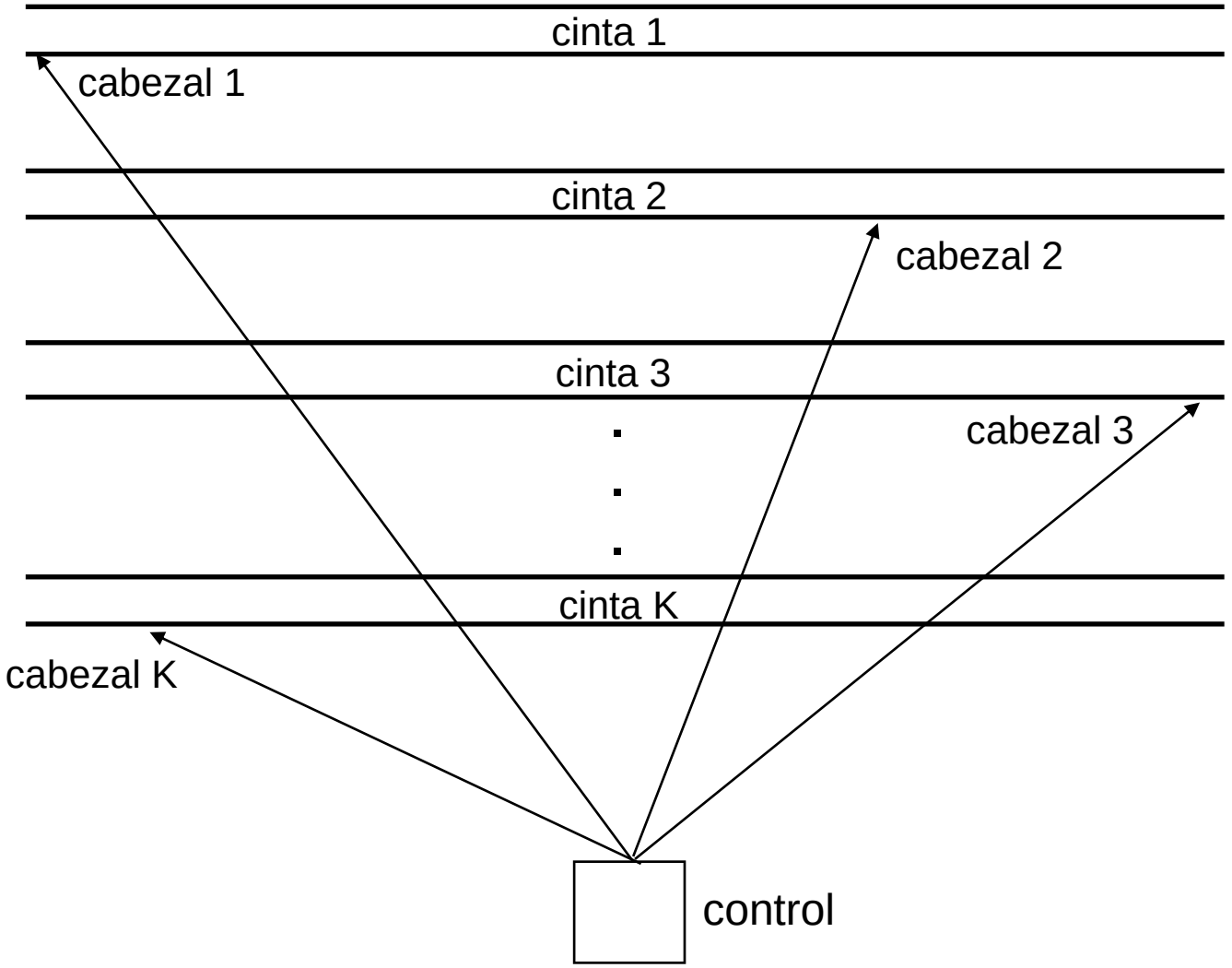


- Ejemplos de modelos de MT equivalentes al modelo de MT con **1 cinta**: MT con **varias cintas**, MT con **cintas semi-infinitas**, MT con **2 cintas y un solo estado**, MT **no determinísticas** (a partir de un estado y un símbolo pueden avanzar de distintas maneras), etc.

- Ejemplos de modelos computacionales equivalentes al modelo de las MT: **máquinas RAM**, **circuitos booleanos**, **lambda cálculo**, **funciones recursivas parciales**, **gramáticas**, **programas Java**, etc.

- Todo esto refuerza la **Tesis de Church-Turing**.

Modelo equivalente de MT con varias cintas



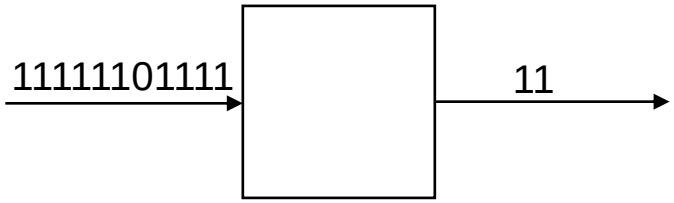
Se puede simular con una MT de una cinta, con un retardo cuadrático.

Otras visiones de MT

<u>Visión de MT calculadora</u> (el caso que vimos al comienzo, el más general, para problemas de búsqueda)

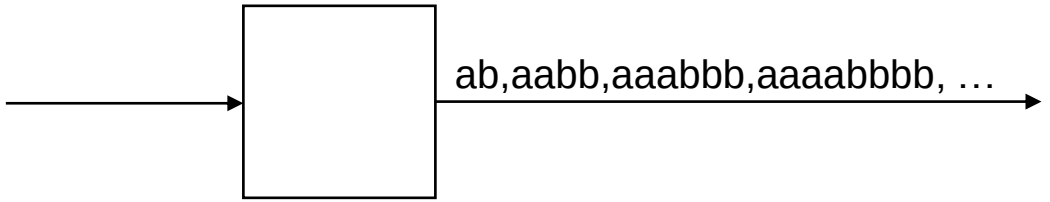
Ejemplo: construir una MT que reste 2 números representados en notación unaria y separados por un cero.
Por ejemplo, dada la entrada 11111101111, obtener la salida 11 (6 – 4 = 2).
<u>Idea general</u>: tachar el primer 1 antes del 0, luego el primer 1 después del 0, luego el segundo 1 antes del 0, y así hasta tachar al final el 0.

<u>Comentario</u>: en este caso, además del estado final, interesa el contenido final de la cinta.



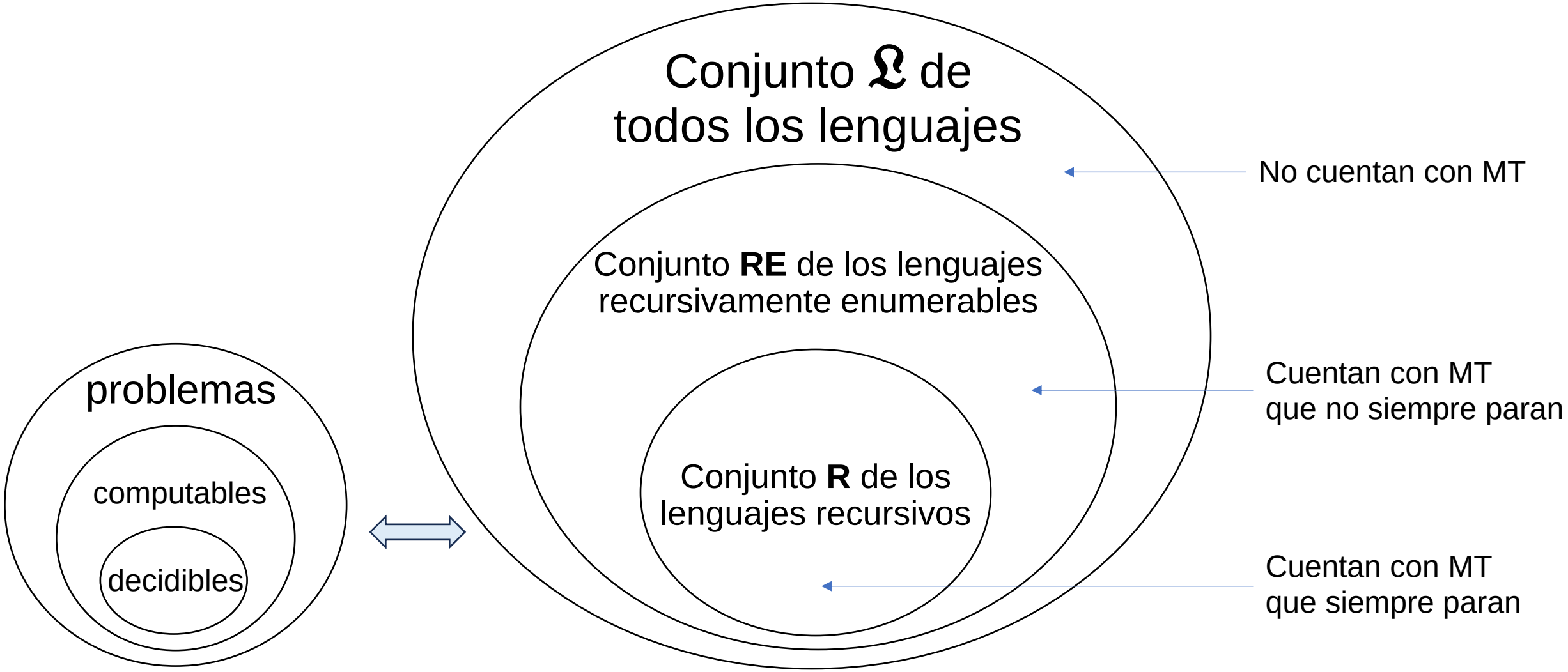
<u>Visión de MT generadora</u>

Ejemplo: construir una MT que genere todas las cadenas de la forma $a^n b^n$, con n ≥ 1. Es decir, que en una cinta especial de salida **genere las cadenas ab, aabb, aaabbb, aaaabbbb, etc.**
<u>Idea general</u>:
(1) i := 1
(2) imprimir i veces *a*, imprimir i veces b, e imprimir una coma
(3) i := i + 1 y volver a (2)



Teorema: Existe una MT M que acepta un lenguaje L sii existe una MT M´ que genera el lenguaje L.

Clase 2. La jerarquía de la computabilidad



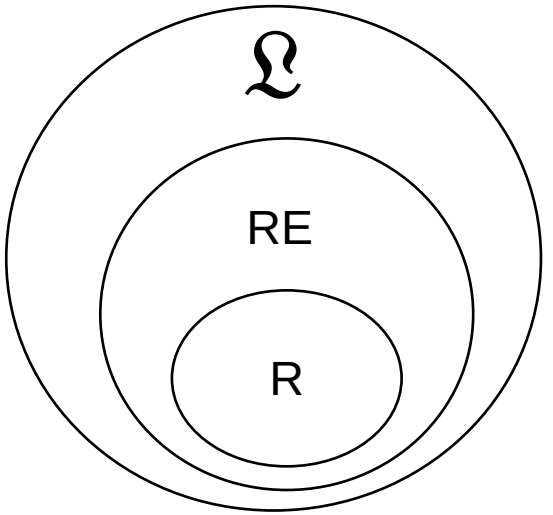
Definiciones

- Un lenguaje L es **recursivo** (L ∈ R) sii

  existe una MT M_L que lo **acepta y para siempre**.

  Para toda cadena w del conjunto universal de cadenas Σ*:

  - Si w ∈ L, entonces M_L a partir de w para en su estado q_A
  - Si w ∉ L, entonces M_L a partir de w para en su estado q_R



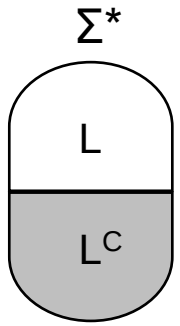
- Un lenguaje L es **recursivamente numerable** (L ∈ RE) sii

  existe una MT M_L que lo **acepta**.

  Para toda cadena w del conjunto universal de cadenas Σ*:

  - Si w ∈ L, entonces M_L a partir de w para en su estado q_A
  - Si w ∉ L, entonces M_L a partir de w para en su estado q_R o no para

Algunas propiedades de la clase R



Propiedad 1. **Si $L \in R$, entonces $L^C \in R$**, tal que L^C es el complemento de L.

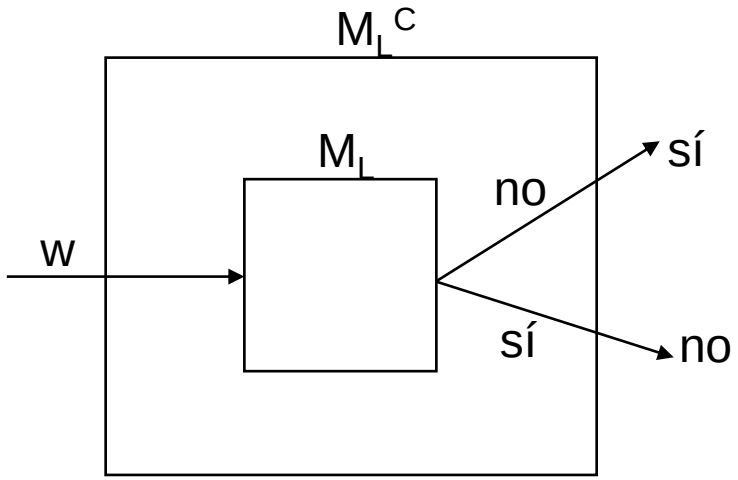
Definición: $L^C = (\Sigma^* - L)$, o en otras palabras, L^C tiene las cadenas que no tiene L.

Prueba.

1. Idea general.

Dada una MT M_L que acepta L y para siempre, la idea es construir una MT $M_L{}^C$ que acepte L^C y pare siempre.

Propuesta de solución: construir $M_L{}^C$ como M_L pero permutando sus estados finales:



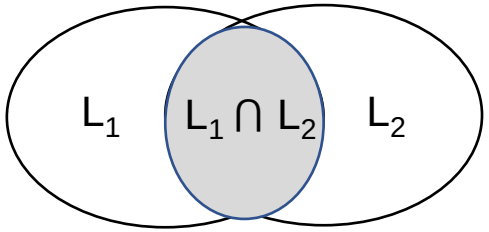
M_L acepta L y para siempre por hipótesis ($L \in R$).

Entonces $M_L{}^C$ acepta L^C y para siempre,

y así también $L^C \in R$.

Propiedad 2. **Si $L_1 \in R$ y $L_2 \in R$, entonces $L_1 \cap L_2 \in R$**, tal que $L_1 \cap L_2$ es la intersección de L_1 y L_2.

Definición: $L_1 \cap L_2$ tiene las cadenas que están en L_1 y L_2.

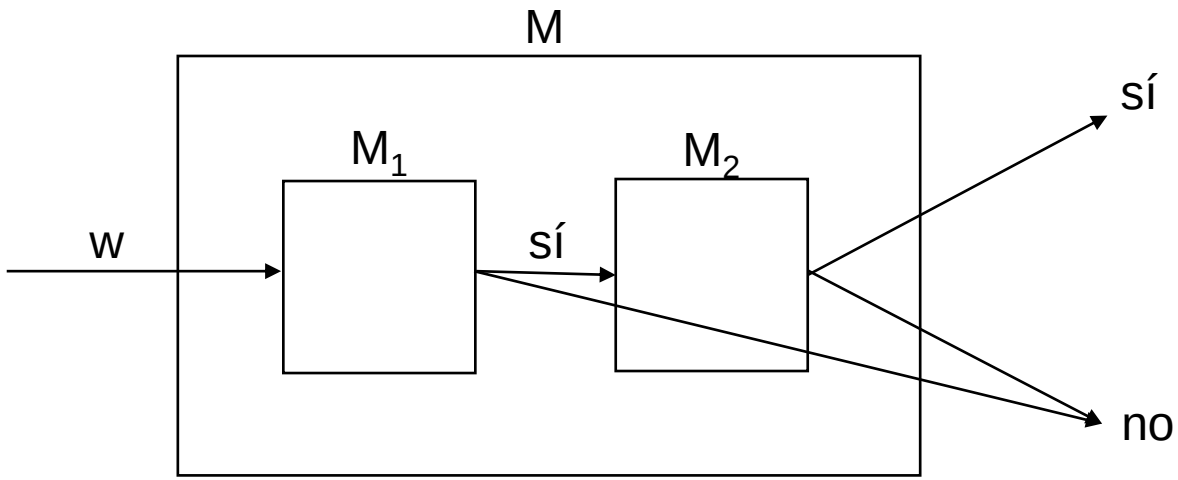


Prueba.

1. Idea general.

Dadas dos MT M_1 y M_2 que respectivamente aceptan L_1 y L_2 y paran siempre, la idea es construir una MT M que acepte $L_1 \cap L_2$ y pare siempre.

Propuesta de solución: ejecutar secuencialmente las dos MT, M_1 y M_2:



M acepta sólo las cadenas que pasan por los "filtros" de M_1 y M_2.

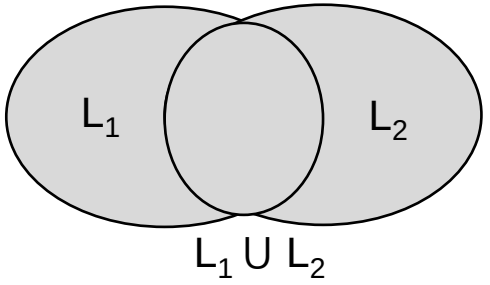
M para siempre porque M_1 y M_2 paran siempre.

Por lo tanto, $L_1 \cap L_2 \in R$.

Algunas propiedades de la clase RE

Propiedad 3. **Si $L_1 \in RE$ y $L_2 \in RE$, entonces $L_1 \cup L_2 \in RE$**, tal que $L_1 \cup L_2$ es la unión de L_1 y L_2.

Definición: $L_1 \cup L_2$ tiene las cadenas que están en L_1 o L_2.

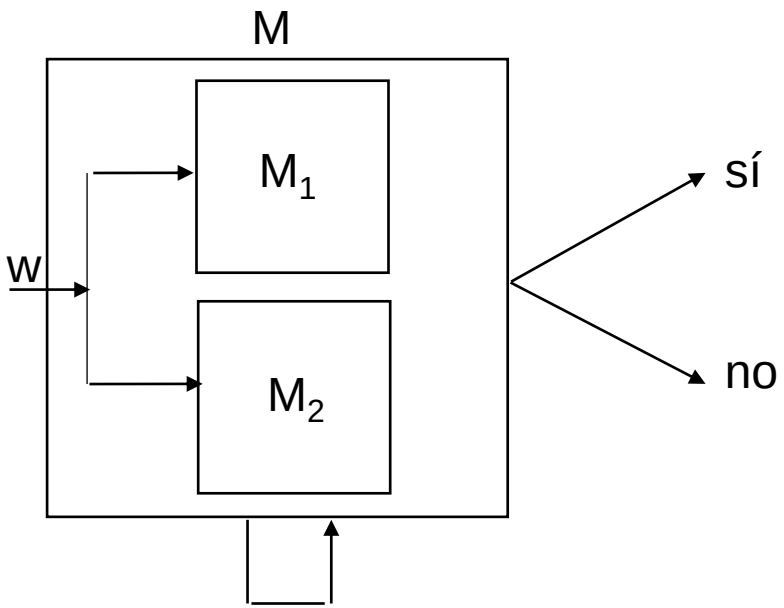


Prueba.

1. Idea general.

Dadas dos MT M_1 y M_2 que respectivamente aceptan L_1 y L_2, la idea es construir una MT M que acepte $L_1 \cup L_2$.

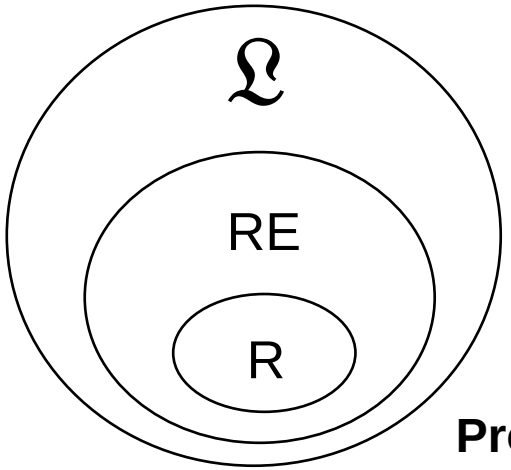
Propuesta de solución: ejecutar las dos MT en paralelo:



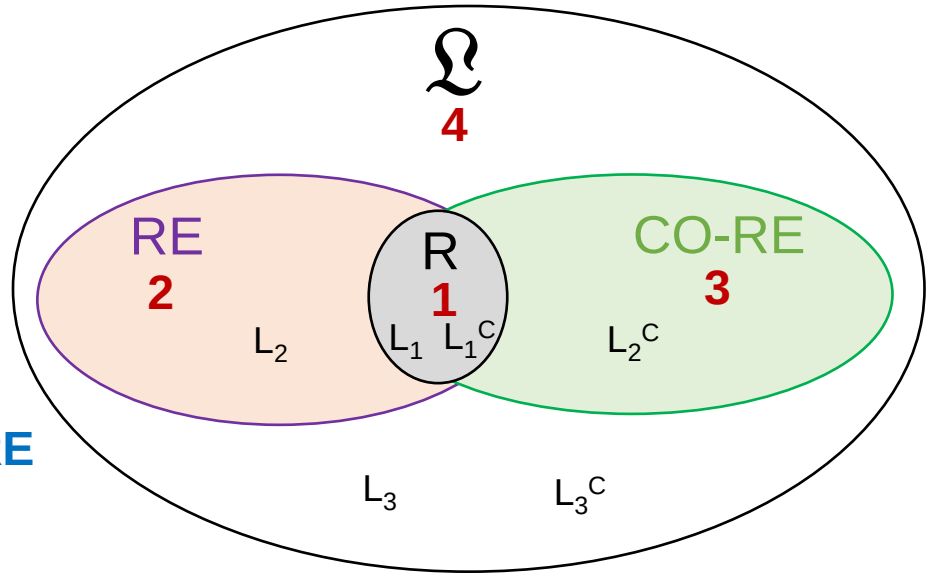
M acepta sii alguna acepta

Versión definitiva de la jerarquía de la computabilidad

Primera versión



Segunda versión



Propiedad 4. **R = RE ∩ CO-RE**

CO-RE tiene los complementos de los lenguajes de RE

Región 1 (los lenguajes más "fáciles").
R es la clase de los lenguajes recursivos.
Si L_1 está en R, entonces también L_1^C está en R.

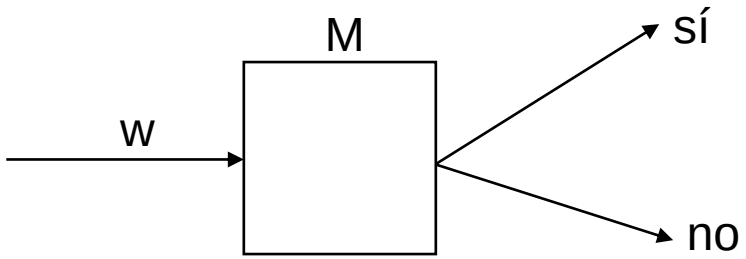
Región 2.
Clase RE – R.
Si L_2 está en RE, entonces L_2^C está en CO-RE.

Región 3.
Clase CO-RE – R.
Si L_2 está en CO-RE, entonces L_2^C está en RE.

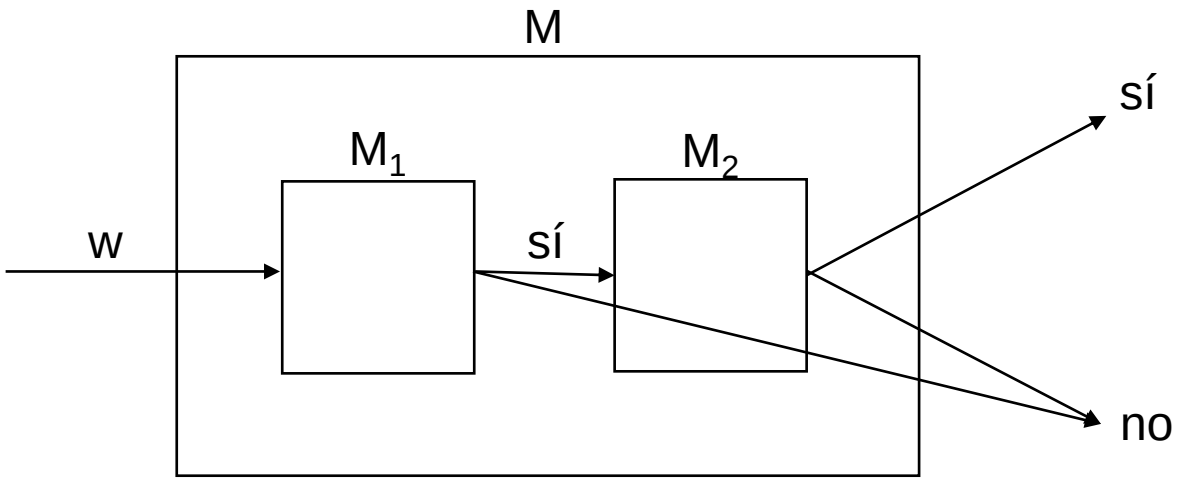
Región 4 (los lenguajes más "difíciles").
Clase Ω – (RE U CO-RE).
Si L_3 está en la clase, también está L_3^C.

Clase 3. Diagonalización

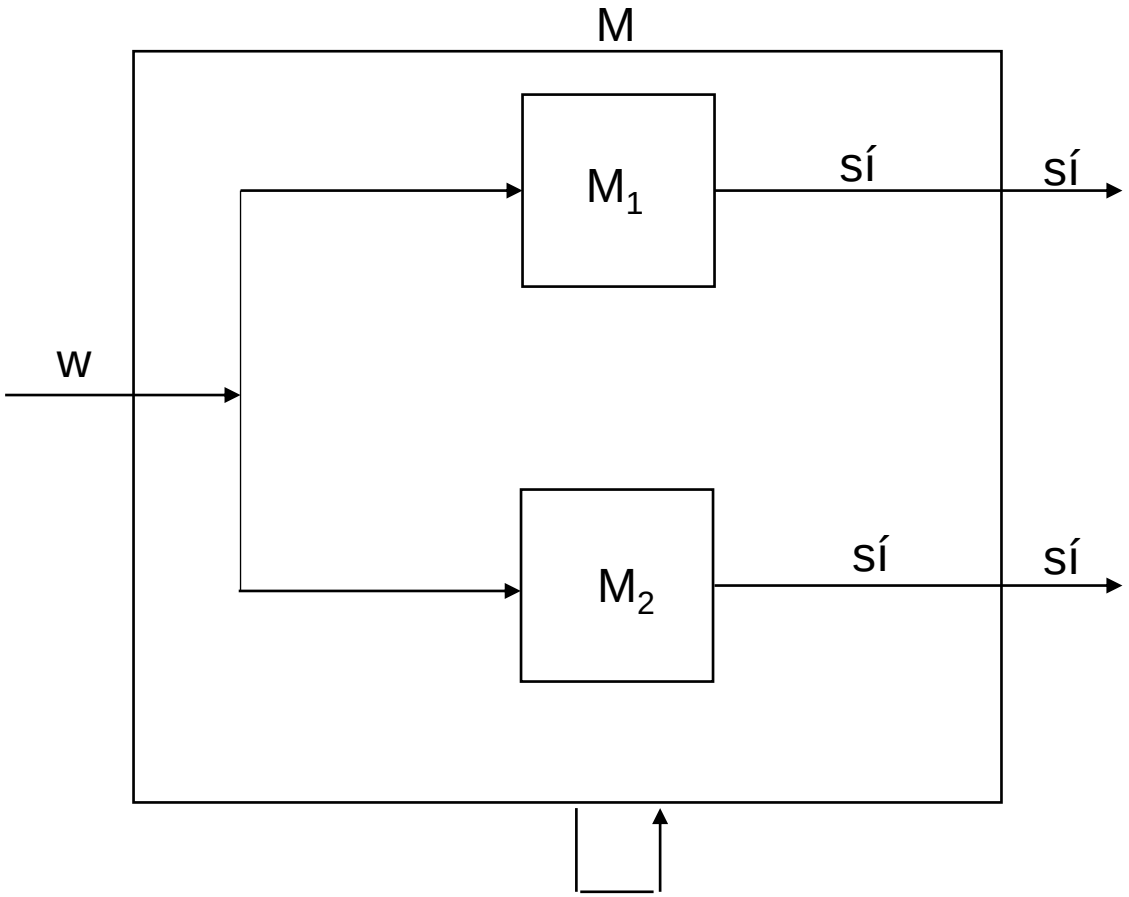
Pruebas constructivas



M decide el lenguaje de las cadenas $a^n b^n$, con $n \geq 1$



M decide la intersección de los lenguajes que deciden M_1 y M_2

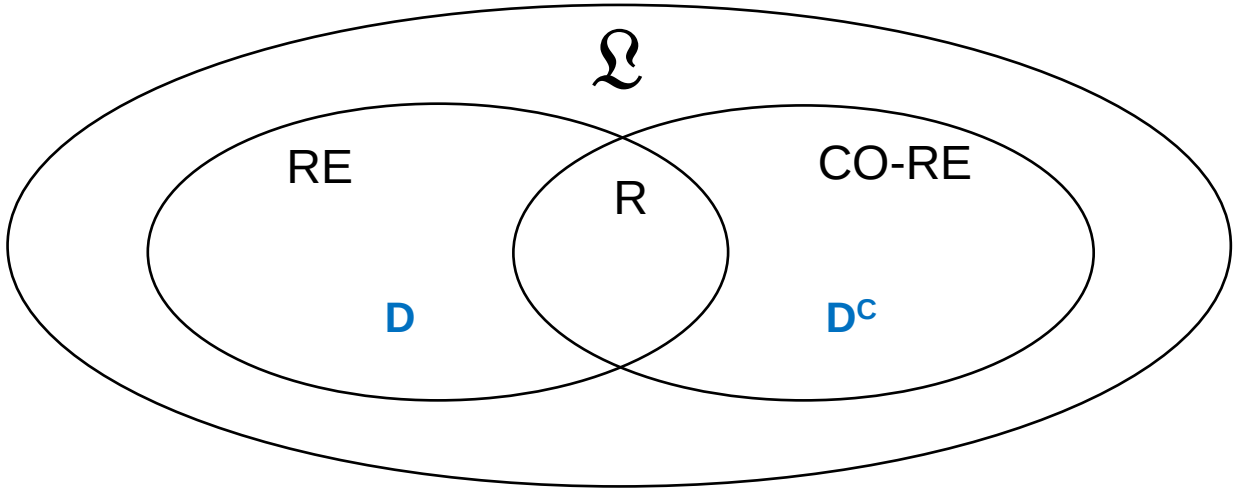


M reconoce la unión de los lenguajes que reconocen M_1 y M_2 (puede no parar)

Prueba no constructiva

Primer lenguaje **D** en RE – R, para probar **R ⊂ RE**

Primer lenguaje **D^C** en CO-RE – R, para probar **RE ⊂ Ω**



Primero, prueba constructiva de que **D ∈ RE**.

Luego, prueba **por diagonalización** (no constructiva**)** de que **D^C ∉ RE**.

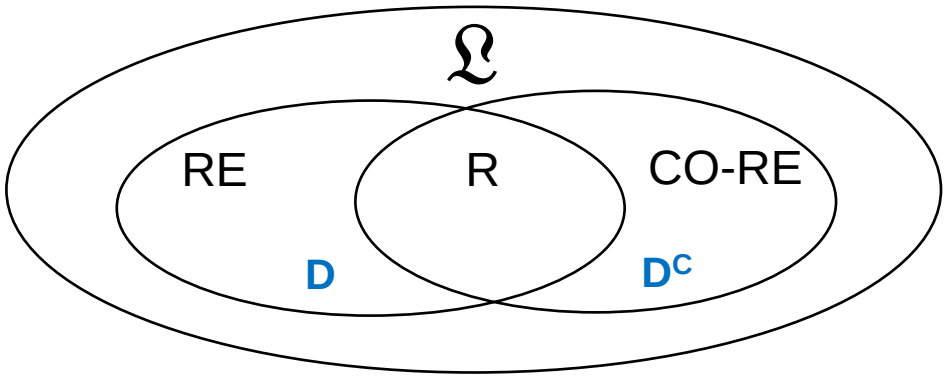
Finalmente, conclusión de que **D ∉ R** (porque si D ∈ R, entonces también D^C ∈ R, absurdo porque D^C ∉ RE).

Detalle de la diagonalización

T	w_0	w_1	w_2	w_3	w_4	.....
M_0	**1**	0	1	1	1	.....
M_1	1	**0**	0	1	0	.....
M_2	0	0	**1**	0	1	.....
M_3	0	1	1	**1**	1	.....
M_4	0	1	1	1	**0**	.....
.....	.....	.....	.....	.....	.....	.....

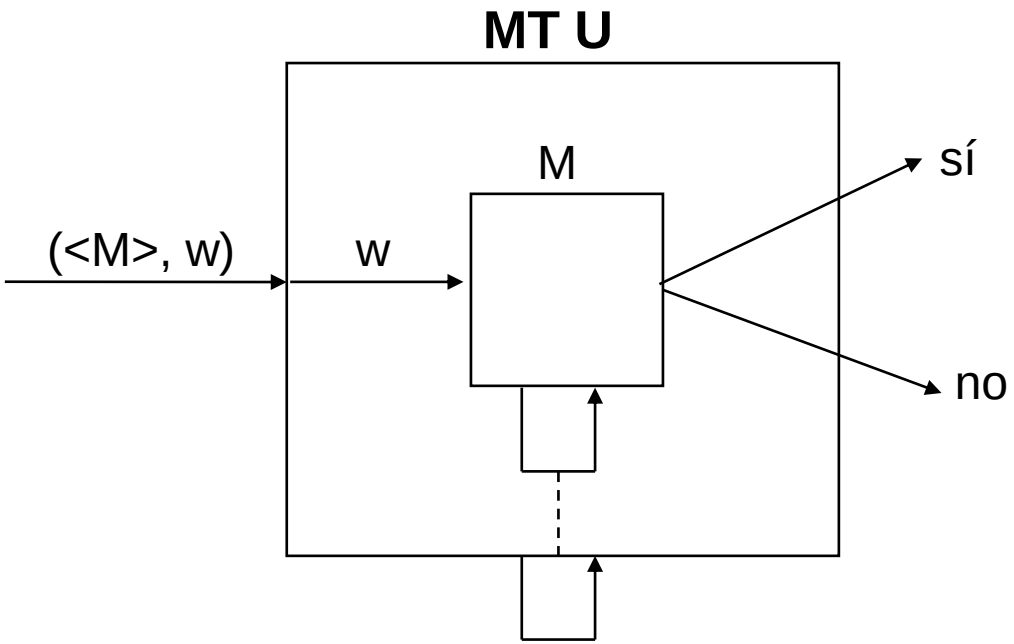
- La diagonal de T representa el lenguaje **D = {w_i | M_i acepta w_i}.**
- La diagonal de T con los 1 y 0 invertidos representa el lenguaje: **D^C = {w_i | M_i rechaza w_i}.**
- Se cumple:

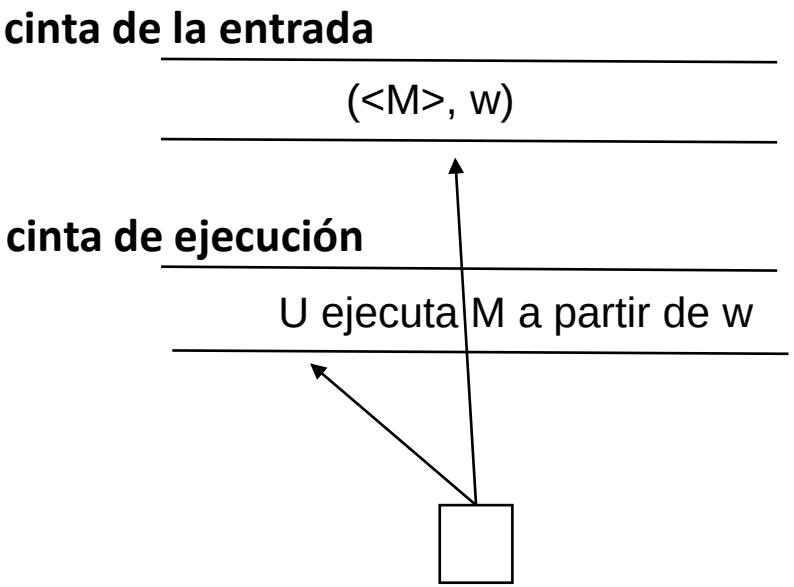
(1) **D está en RE – R**
(2) **D^C está en CO-RE – R**



Máquina de Turing universal

- Una máquina de Turing universal (MT U) es una máquina de Turing capaz de ejecutar otra (noción de **programa almacenado**, Turing 1936). El esquema más general es:





- La MT U recibe como entrada una **MT M** (codificada mediante una cadena <M>) y una **cadena w**, y **ejecuta M a partir de w**. Puede tener una o más cintas de ejecución.

Codificación y enumeración de las máquinas de Turing

- Codificación con números en binario. Para enumerar las cadenas utilizamos **el orden canónico:**

  1) De menor a mayor longitud.

  2) Con longitudes iguales desempata el orden alfanumérico (números y símbolos especiales).

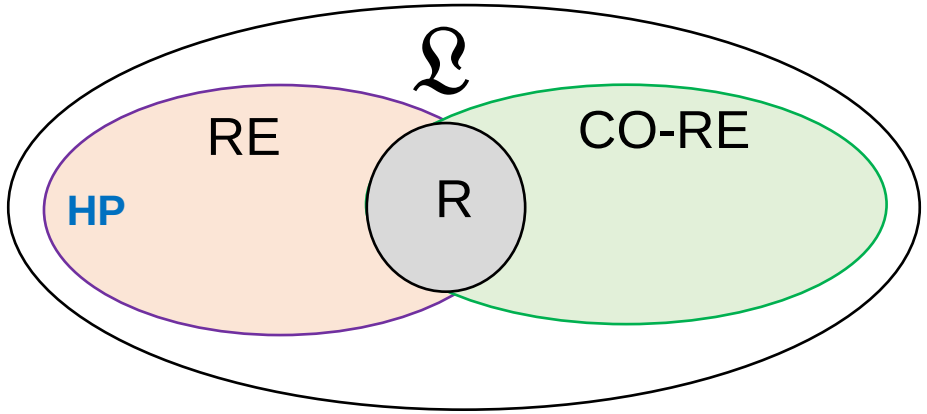
- La siguiente MT M obtiene la MT M_i según el orden canónico. Dada una entrada i, M hace:

  1. Hace n := 0.
  2. Genera la siguiente cadena v según el orden canónico.
  3. Si v no es el código de una MT vuelve al paso 2.
  4. Si n = i, devuelve v **(v es el código de la MT M_i)** y para.
     Si n ≠ i, hace n := n + 1 y vuelve al paso 2.

  Validar que v es el código de una MT implica un chequeo sintáctico.

Problema de la detención de una MT (*halting problem*)

- Dada una MT M y una cadena w, ¿M para a partir de w?
- El lenguaje que representa el problema es **HP = {(<M>, w) | M para a partir de w}.**
- HP pertenece al conjunto **RE – R**. Probado por Turing en 1936 (por diagonalización).

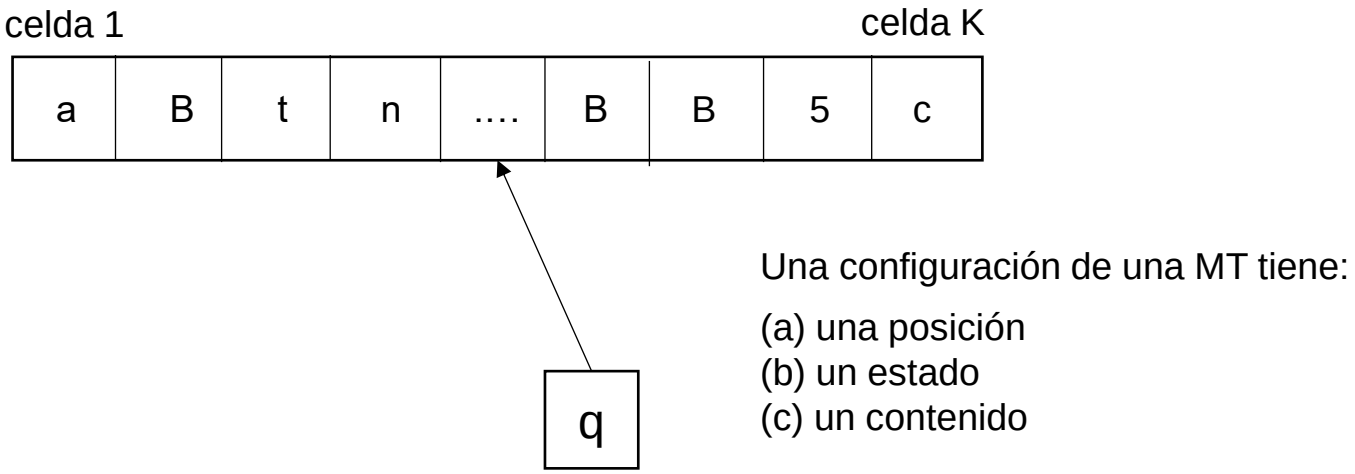


Cómo burlar al *halting problem*

- **Ejemplo 1.** Si una MT se mueve en un espacio limitado, se puede detectar cuándo entra en un loop.

  Por ejemplo, supongamos que una MT M con una cinta se mueve en no más de K celdas.

  ¿Por cuántas configuraciones distintas puede pasar M antes de loopear?



Si M tiene |Q| estados y |Γ| símbolos, antes de repetir una configuración hará a lo sumo:

$K.|Q|.|\Gamma|^K$ pasos (K posiciones, |Q| estados, $|\Gamma|^K$ contenidos).

Se puede detectar si una MT loopea **ejecutándola y llevando la cuenta de sus pasos**.

Cómo burlar al *halting problem* (continuación)

- **Ejemplo 2.** ¿Cómo detectar si una MT M acepta al menos una cadena?

  Tenemos que tener cuidado en cómo construimos una MT M´ que chequee si M acepta una cadena.
  No sirve que M´ ejecute M sobre la 1ra cadena, luego sobre la 2da, luego sobre 3ra, …, porque M puede no detenerse en algunos casos.

Solución:

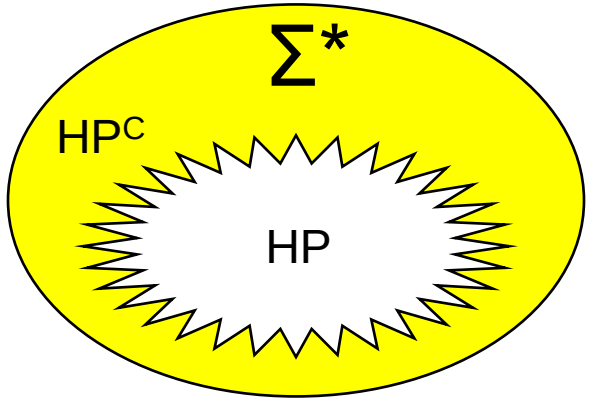
1. Hacer i := 1.
2. Ejecutar i pasos de M sobre todas las cadenas de longitud a lo sumo i.
3. Si M acepta alguna vez, aceptar.
4. Si no, hacer i := i + 1 y volver al paso 2.

pasos	**cadenas**
1	λ w_0 w_1 w_2 w_3 …
2	λ w_0 w_1 w_2 … w_0w_0 w_0w_1 w_0w_2 …
3	λ w_0 w_1 … w_0w_0 w_0w_1 … $w_0w_0w_0$ $w_0w_0w_1$ …
...	……

Por ejemplo, si la primera cadena que M acepta mide 80 símbolos, y M la acepta en 120 pasos, entonces cuando la MT M´ ejecute 120 pasos de M sobre todas las cadenas de a lo sumo 120 símbolos la va a encontrar.

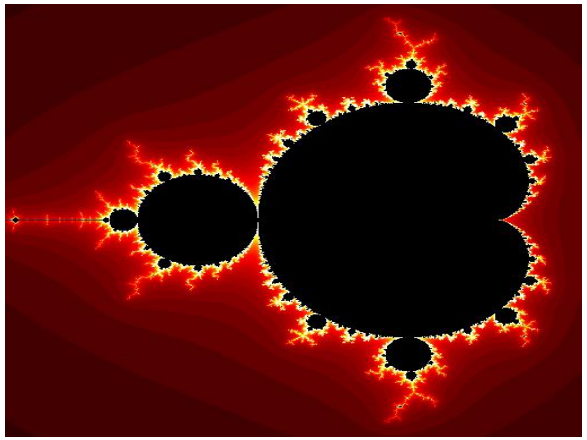
Computabilidad y tamaño de un lenguaje

- La computabilidad de un lenguaje (o problema) tiene más que ver con su definición, su **contorno** (representación gráfica), que con su tamaño.



- El contorno del Conjunto de Mandelbrot es un muy buen ejemplo de un lenguaje no recursivo.

CONJUNTO DE MANDELBROT

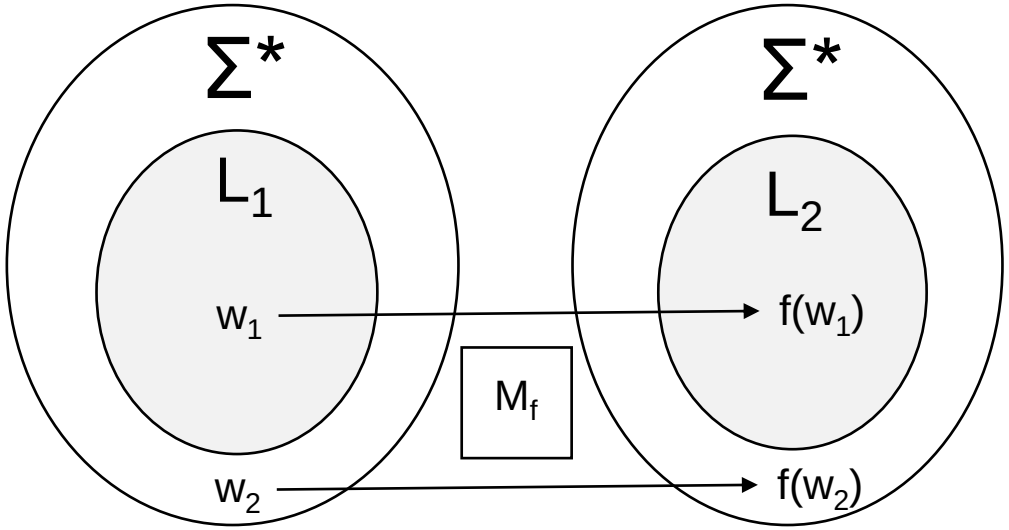


Clase 4. Reducciones

- Dados dos lenguajes L_1 y L_2, supongamos que existe una MT M_f que **computa** una función $f : \Sigma^* \longrightarrow \Sigma^*$ de la siguiente forma:

  a partir de todo $w \in L_1$, la MT M_f genera $f(w) \in L_2$

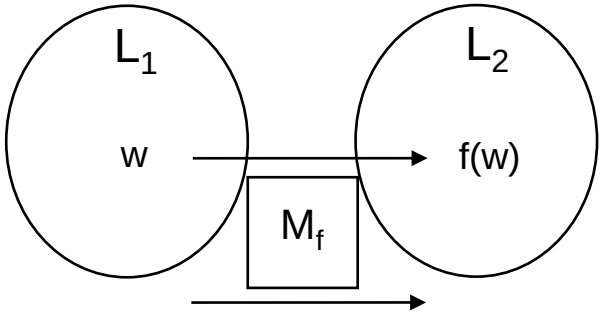
  a partir de todo $w \notin L_1$, la MT M_f genera $f(w) \notin L_2$



Se define que la función f es una **reducción** de L_1 a L_2.

Se anota $\mathbf{L_1 \leq L_2}$, y se dice que la función f es **total computable** (se computa sobre todas las cadenas).

Utilidad de las reducciones

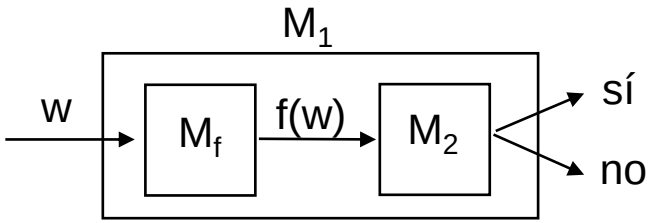


Reducción de L_1 a L_2
Para todo w, $w \in L_1$ sii $f(w) \in L_2$

TEOREMA

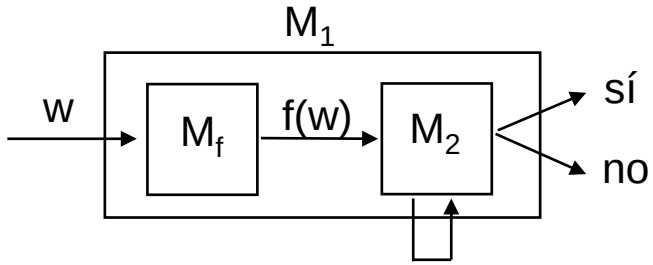
Caso 1

Si $L_1 \leq L_2$ entonces ($L_2 \in R \longrightarrow L_1 \in R$)
O bien:
Si $L_1 \leq L_2$ entonces ($L_1 \notin R \longrightarrow L_2 \notin R$)



Caso 2

Si $L_1 \leq L_2$ entonces ($L_2 \in RE \longrightarrow L_1 \in RE$)
O bien:
Si $L_1 \leq L_2$ entonces ($L_1 \notin RE \longrightarrow L_2 \notin RE$)

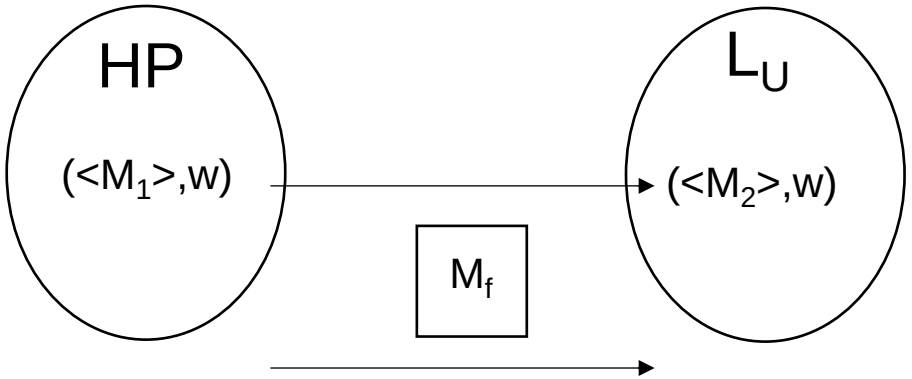


Es decir, si $L_1 \leq L_2$:
Si $L_1 \notin R$, no puede suceder que $L_2 \in R$.
Si $L_1 \notin RE$, no puede suceder que $L_2 \in RE$.
L_2 es **tan o más difícil** que L_1, r**esolviendo L_2 se resuelve L_1.**

De esta manera, las reducciones permiten encontrar lenguajes **dentro y fuera de R y RE.**

Ejemplo

HP = {(<M>, w) | M para sobre w} y L_U = {(<M>, w) | M acepta w}.



De acuerdo al teorema.

si $L_1 \leq L_2$, entonces $L_1 \notin R \longrightarrow L_2 \notin R$.

Por lo tanto, como HP $\notin$ R,

también probamos que $L_U \notin R$.

Definición de la reducción

f((<M_1>, w)) = (<M_2>, w), con M_2 como M_1, salvo que **los estados q_R de M_1 se cambian en M_2 por estados q_A**.

Computabilidad

Existe una MT M_f que computa f: copia (<M_1>, w) pero cambiando los estados q_R de M_1 por estados q_A en M_2.

Correctitud

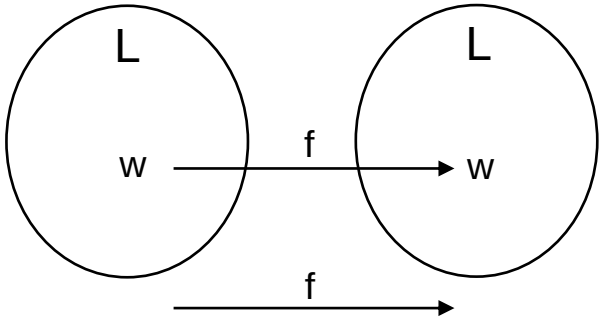
(<M_1>, w) $\in$ HP $\longrightarrow$ M_1 para sobre w $\longrightarrow$ M_2 acepta w $\longrightarrow$ (<M_2>, w) $\in$ L_U

(<M_1>, w) $\notin$ HP $\longrightarrow$ caso de cadena válida: M_1 no para sobre w $\longrightarrow$ M_2 no para sobre w $\longrightarrow$ (<M_2>, w) $\notin$ L_U

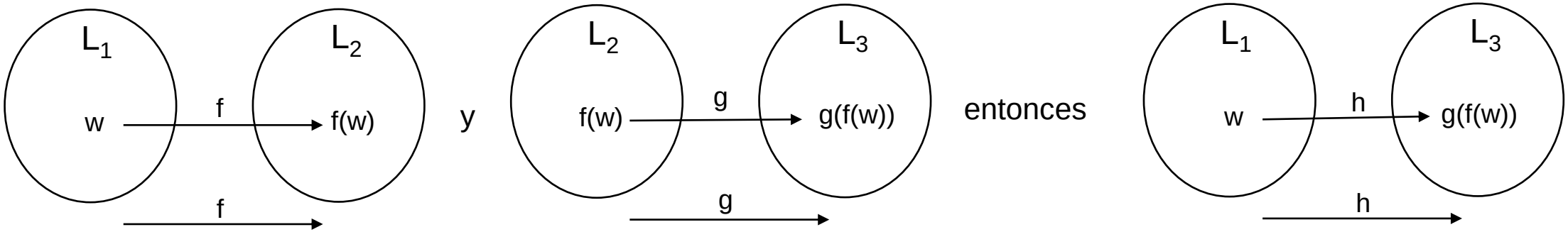
caso de cadena inválida: (<M_2>, w) también es una cadena inválida $\longrightarrow$ (<M_2>, w) $\notin$ L_U

Propiedades

- **Reflexividad.** Para todo lenguaje L se cumple $L \leq L$.
  La función de reducción es la función identidad.

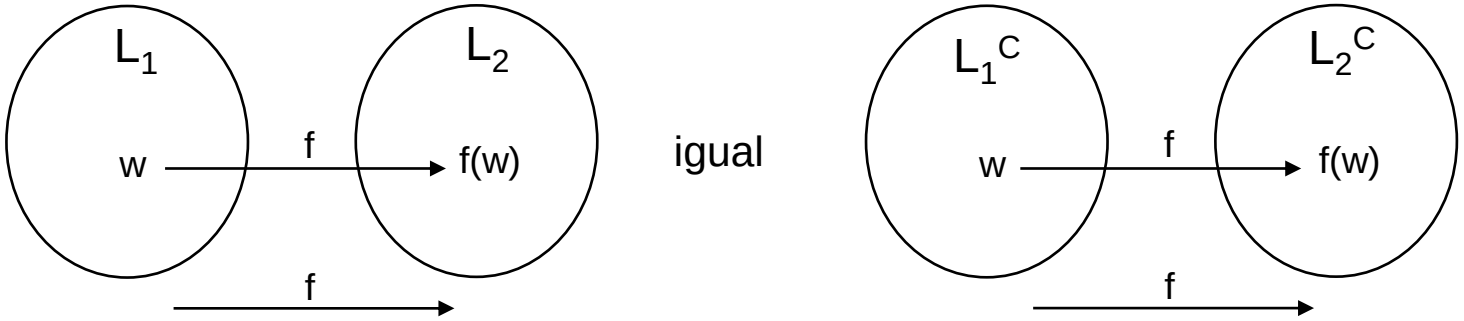


- **Transitividad.** Si $L_1 \leq L_2$ y $L_2 \leq L_3$, entonces $L_1 \leq L_3$.



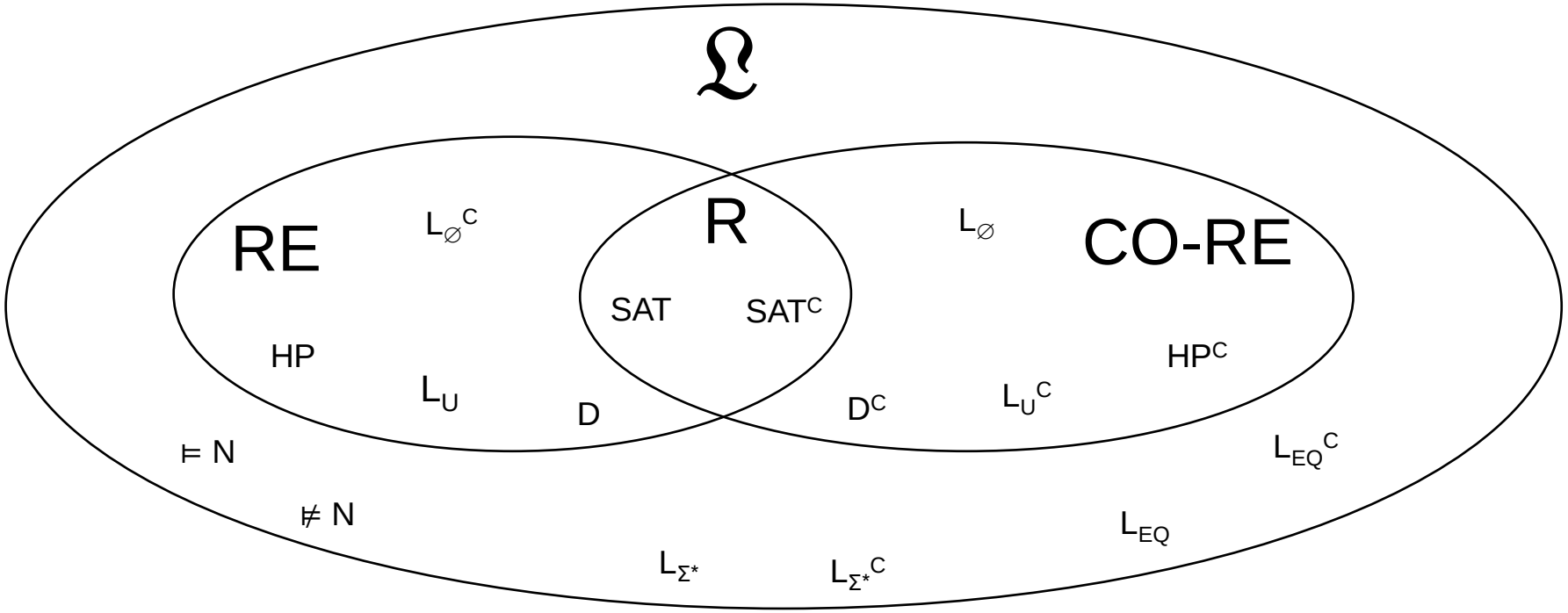
Se componen las reducciones f y g y se obtiene la reducción h.

- **Otra propiedad:** $L_1 \leq L_2$ sii $L_1^C \leq L_2^C$. Es la misma función de reducción.



No se cumple la simetría.
$L_1 \leq L_2$ no implica $L_2 \leq L_1$.

Poblando la jerarquía de la computabilidad



SAT: problema de satisfactibilidad (construcción de una MT)

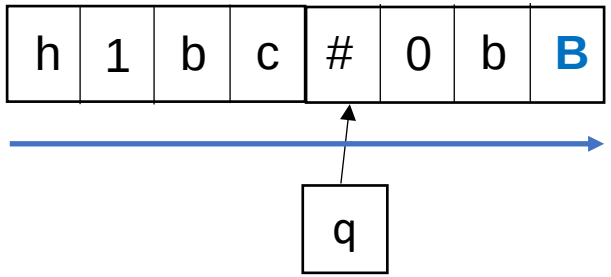
HP: *halting problem* (diagonalización)

L_{EQ}: problema de equivalencia de máquinas de Turing (reducción)

Máquinas de Turing restringidas

AUTÓMATAS FINITOS (AF)

- Una cinta de sólo lectura.
- Sólo movimiento a la derecha.
- Conjunto F de estados finales.



- Cuando se alcanza el símbolo B (blanco) el AF para (acepta sii el estado alcanzado es final).

- El AF constituye un tipo de algoritmo muy utilizado. Por ejemplo:
  - Para el **análisis sintáctico a nivel palabra** de los compiladores (if, then, else, while, x, 10, =, +, etc).
  - Para las **inspecciones de código** en el control de calidad del software.

Ejemplo

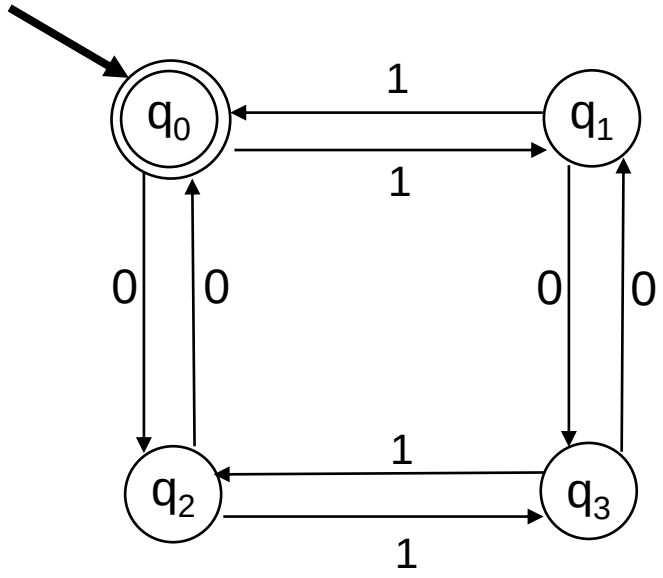
$Q = \{q_0, q_1, q_2, q_3\}$ $\Gamma = \{0, 1\}$ Estado inicial q_0 $F = \{q_0\}$

1. $\delta(q_0, 1) = q_1$
2. $\delta(q_1, 1) = q_0$
3. $\delta(q_1, 0) = q_3$
4. $\delta(q_3, 0) = q_1$
5. $\delta(q_3, 1) = q_2$
6. $\delta(q_2, 1) = q_3$
7. $\delta(q_2, 0) = q_0$
8. $\delta(q_0, 0) = q_2$

Diagrama de transición de estados

La ejecución arranca desde la flecha.

Los estados con doble contorno son los estados finales.



El AF descripto acepta todas las cadenas de 1 y 0 con **una cantidad par de 1 y una cantidad par de 0.**

Algunas características de los AF

- Los AF **siempre paran.**

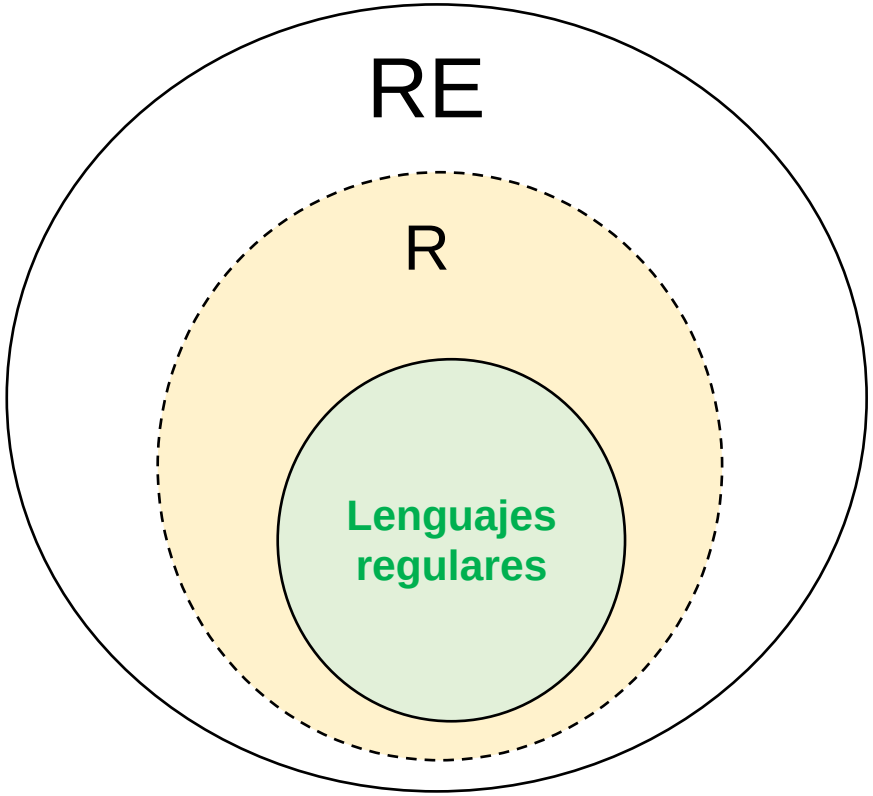
- Aceptan un tipo limitado de cadenas **(no tienen memoria).**

  No pueden aceptar cadenas con igual cantidad de *a* y b.
  No pueden chequear si una cadena es un palíndromo.
  Etc. (en general, no pueden calcular).

- Los lenguajes que aceptan se llaman **regulares** o de **tipo 3**.

- A diferencia de las MT generales, **pueden decidir**:

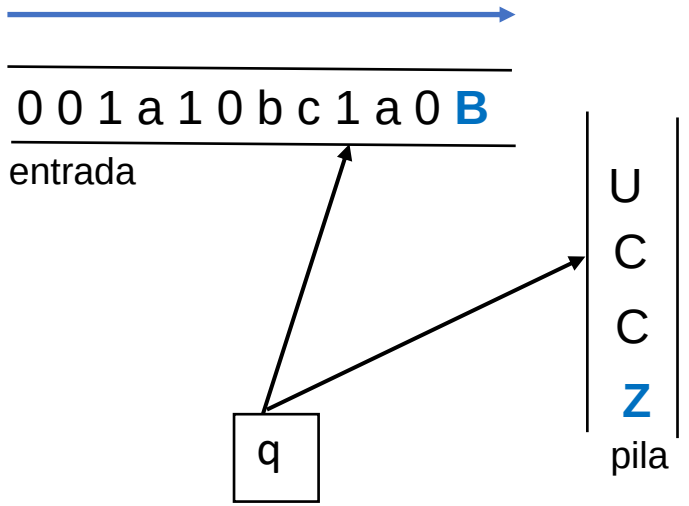
  ¿$w \in L(M)$?
  ¿$L(M) = \varnothing$?
  ¿$L(M) = \Sigma^*$?
  ¿$L(M_1) = L(M_2)$?



AUTÓMATAS CON PILA (AP)

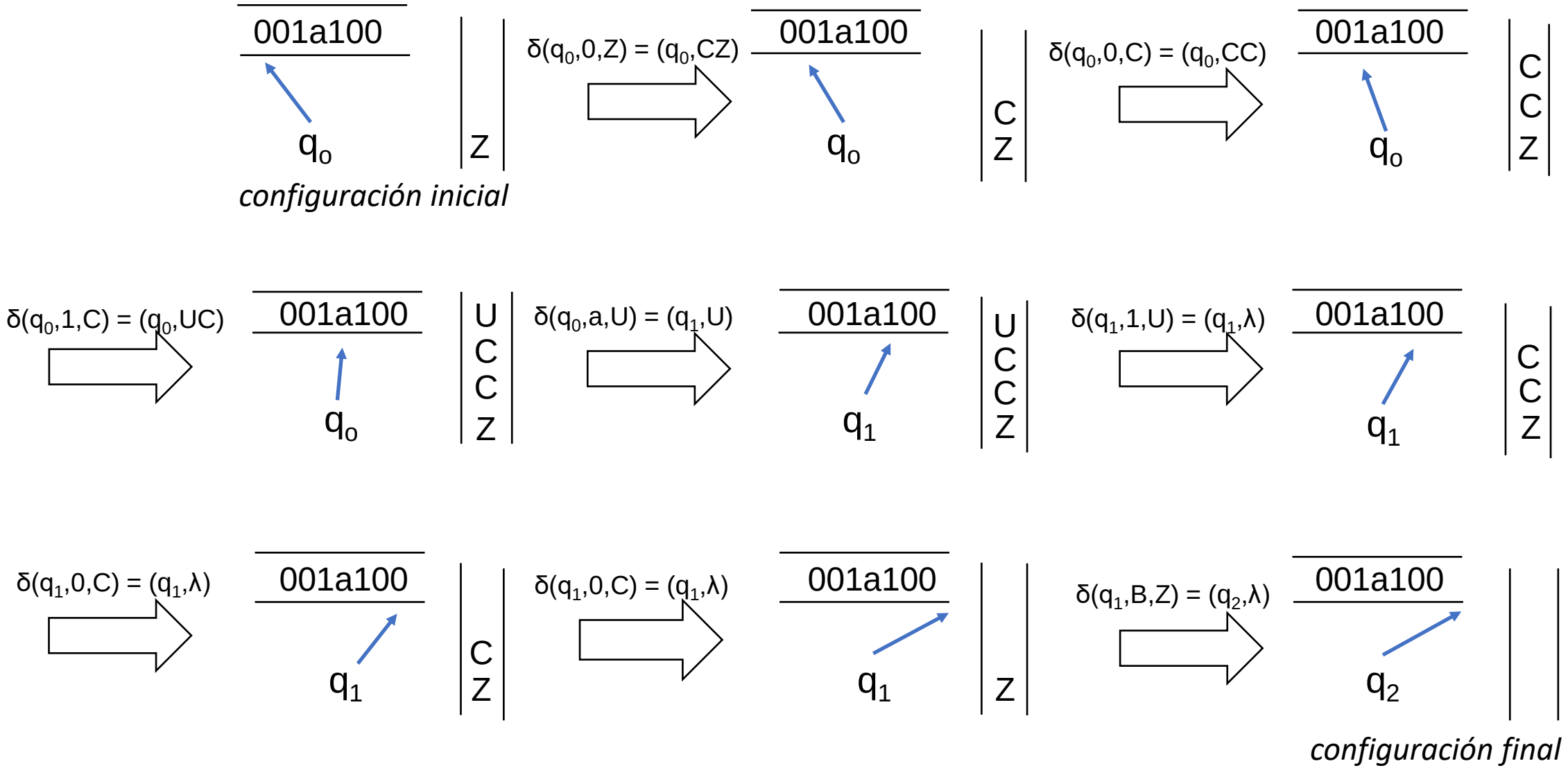
- Una cinta de input de sólo lectura.
- Una cinta de lectura/escritura que se comporta como una pila.
- En un paso se pueden procesar las dos cintas.
- En la cinta de entrada siempre se va a la derecha.
- Cuando se alcanza el símbolo B (blanco) en la cinta de entrada, el AP para (acepta sii la pila está vacía).

- Problemas típicos que resuelve un AP:
  - **Análisis sintáctico a nivel instrucción** de los compiladores.
  - **Evaluación de expresiones** en la ejecución de programas.



Ejemplo

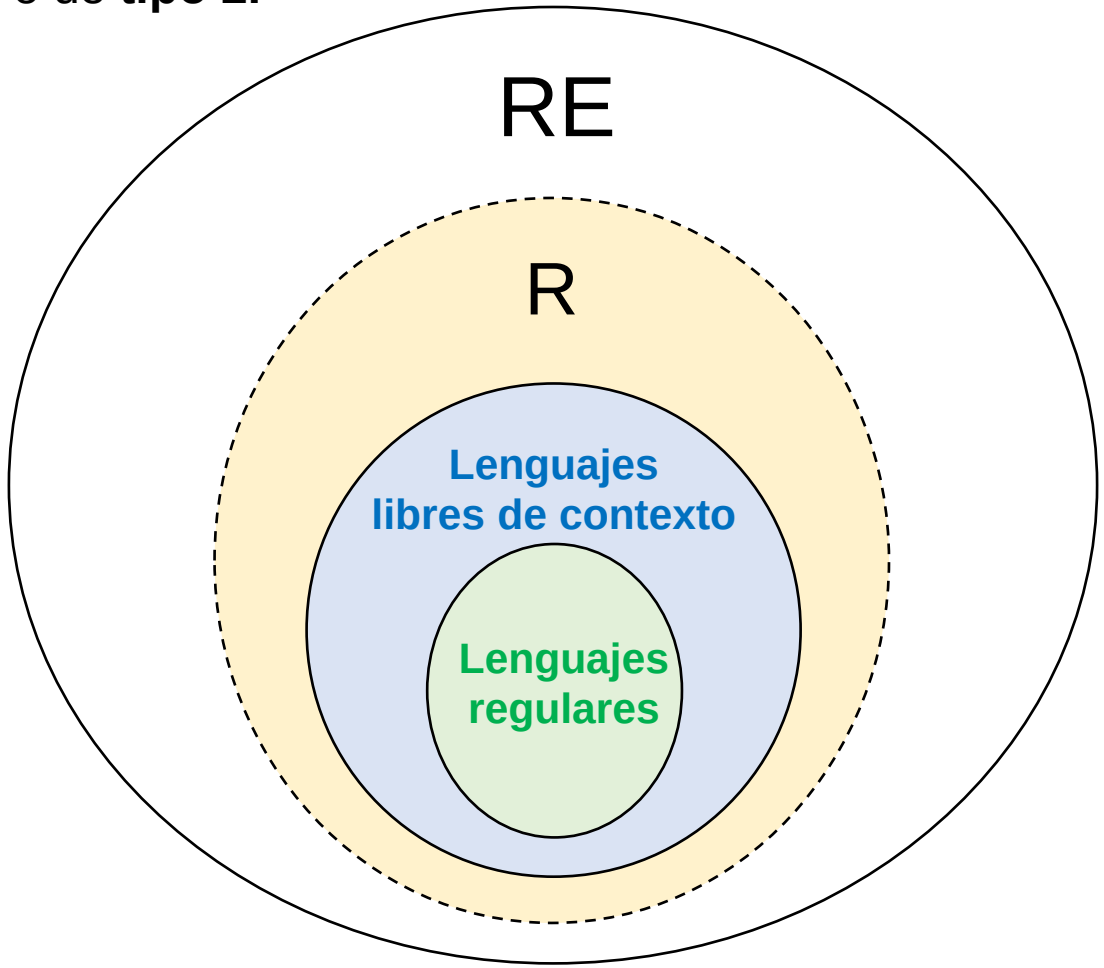
Reconocimiento de cadenas **waw^C,** tales que w tiene 0 y 1 y w^C es la inversa de w.



Algunas características de los AP

- Los AP **siempre paran.**
- Los lenguajes que aceptan se llaman **libres de contexto** o de **tipo 2.**
- A diferencia de las MT generales, **pueden decidir**:

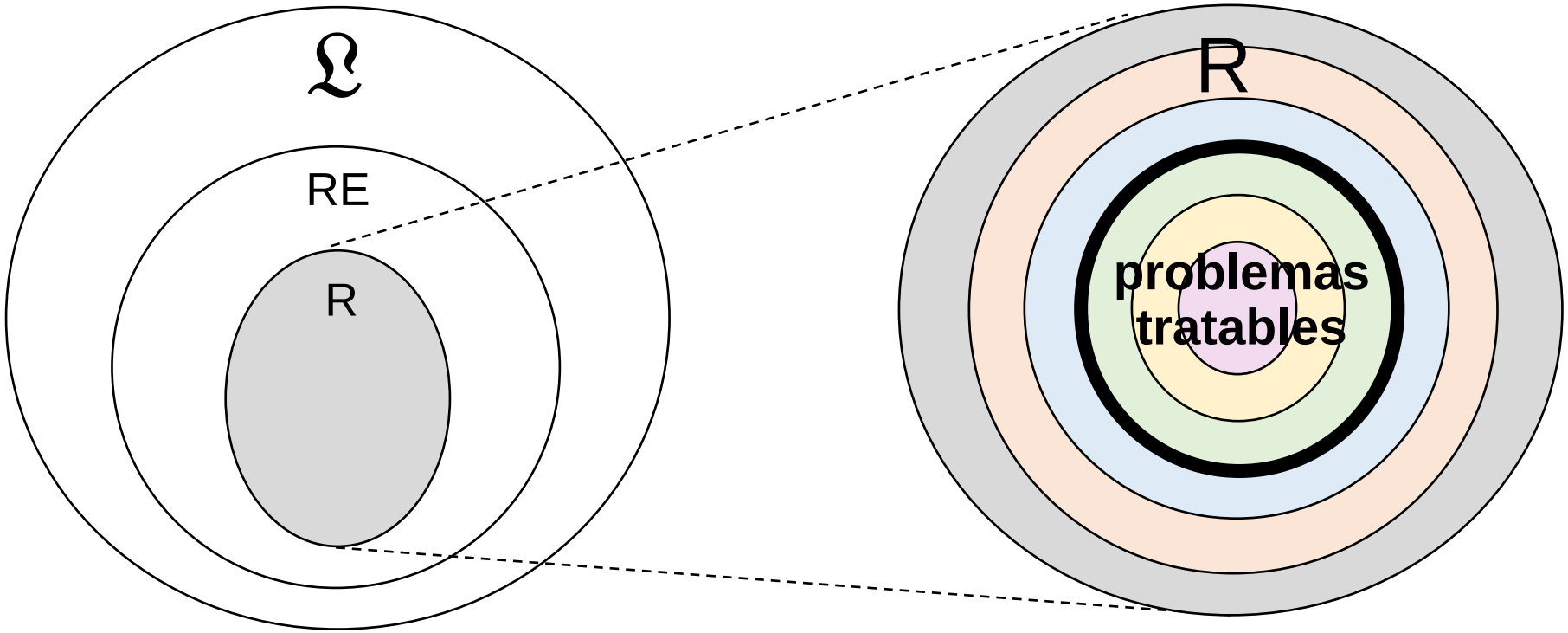
  ¿w ∈ L(M)?
  ¿L(M) = ∅?



- **Jerarquía de Chomsky:**

  Lenguajes de tipo 3 (**regulares**)
  Lenguajes de tipo 2 (**libres de contexto**)
  Lenguajes de tipo 1 (**sensibles al contexto**)
  Lenguajes de tipo 0 (**recursivamente numerables**)

Clase 5. Complejidad temporal

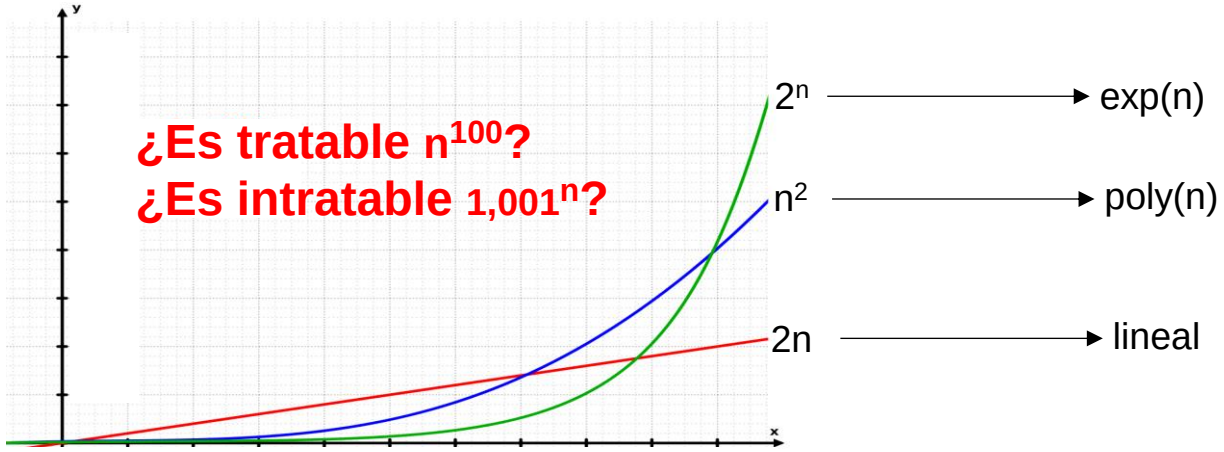


- Se repiten muchos conceptos y técnicas de la computabilidad. Contraste: muchos problemas abiertos.

Definiciones

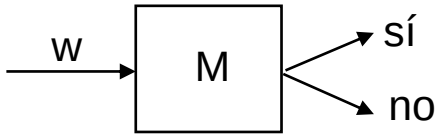
- Una MT tarda más a medida que sus cadenas de entrada w son más grandes.
- Por eso se usan **funciones temporales T(n)** definidas en términos de |w| = n.
- Funciones temporales T(n) típicas: **$5n$, $3n^3$, 2^n, $n^{\log_2 n}$, $7^{\sqrt{n}}$, $n!$, 6^{n^5}**
- Dos grandes grupos de funciones (de acuerdo al nivel de abstracción pretendido):
  Polinomiales o poly(n): $T(n) = c.n^k$
  Exponenciales o exp(n): el resto (cuasipolinomiales, subexponenciales, exponenciales, etc).
- Convención, respaldada por las matemáticas y la experiencia: **tiempo tratable = tiempo poly(n)**

n	2n	n^2	2^n
0	0	0	1
1	2	1	2
2	4	4	4
3	6	9	8
4	8	16	16
5	10	25	32
6	12	36	64
7	14	49	128
8	16	64	256
9	18	81	512
10	20	100	1024



Definiciones (continuación)

- Una MT M **tarda o se ejecuta en tiempo T(n),** sii a partir de toda entrada w, con |w| = n, M hace **a lo sumo T(n) pasos**.

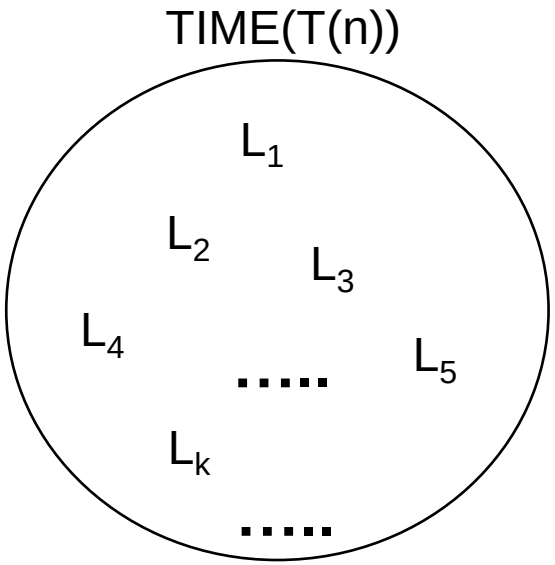


M hace a lo sumo T(|w|) pasos

- Una función $T_1(n)$ es del **orden** de una función $T_2(n)$, es decir $\mathbf{T_1(n) = O(T_2(n))}$, sii para todo $n \geq n_0$ se cumple $\mathbf{T_1(n) \leq c.T_2(n)}$, con $c > 0$.

  Por ejemplo:
  $5n^3 + 8n + 25 = O(n^3)$
  $n^2 = O(n^3)$
  $n^3 = O(2^n)$

- Un lenguaje **L ∈ TIME(T(n))** sii existe una MT M que lo decide en tiempo **O(T(n)).**

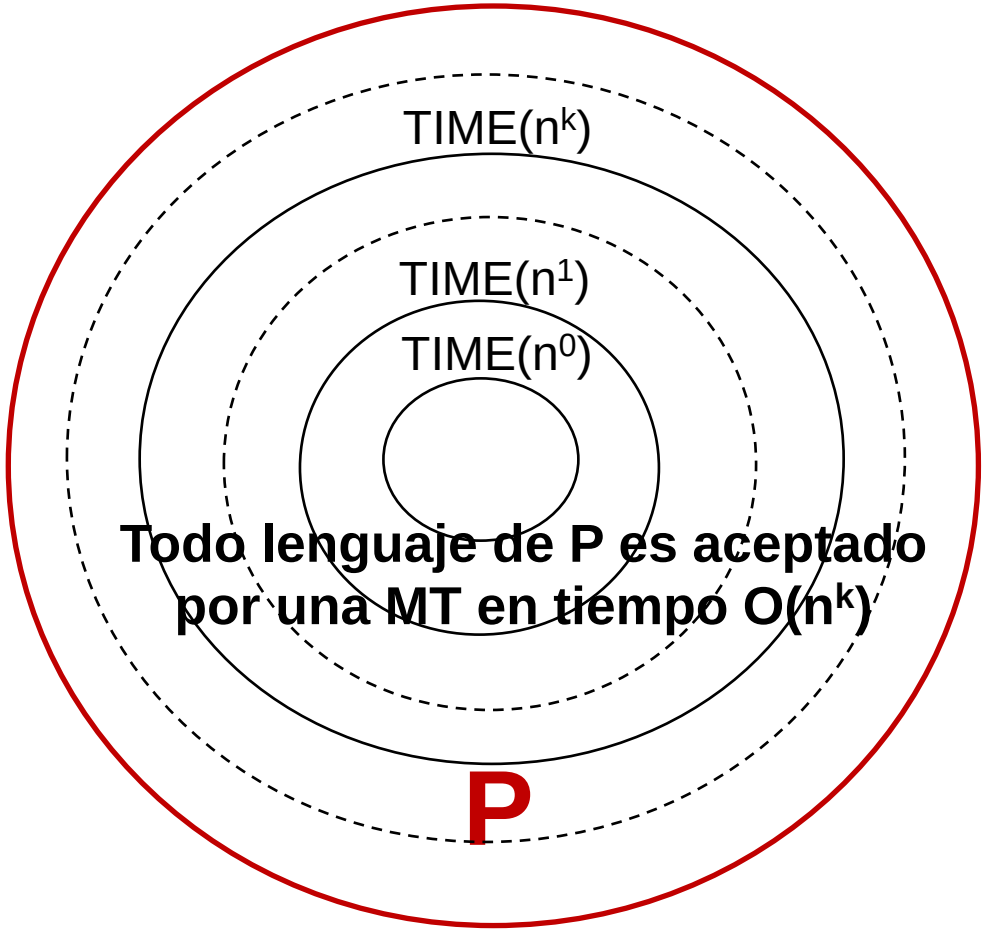


Para todo L_i existe una MT M_i que lo decide en tiempo O(T(n))

Se considera el **peor caso (cota superior).** Por ejemplo, si a partir de la mayoría de las cadenas de un lenguaje el tiempo de ejecución es O(n), y sólo en algunos casos es $O(n^3)$, el tiempo de ejecución queda *castigado* en $O(n^3)$.

Naturalmente. sería mejor considerar el **tiempo promedio** o **mínimo (cota inferior)**, pero son mucho más difíciles de calcular. Hoy día hay pocos valores conocidos de este tipo.

TIME(n^k) = la clase P



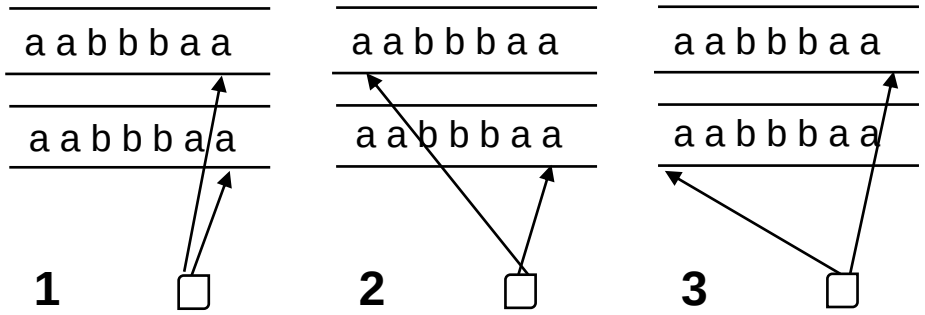
Ejemplo sencillo de lenguaje en P

PALÍNDROMOS = {w | w es un palíndromo con símbolos *a* y b}

Una MT muy simple que decide el lenguaje, con 2 cintas, hace:

1. Copia w de la cinta 1 a la cinta 2
   Tiempo O(n)
2. Posiciona el cabezal de la cinta 1 a la izquierda
   Tiempo O(n)
3. Compara en direcciones contrarias uno a uno los símbolos de las 2 cintas hasta llegar a un blanco en ámbas
   Tiempo O(n)

   Tiempo total: T(n) = O(n) + O(n) + O(n) = **O(n)**

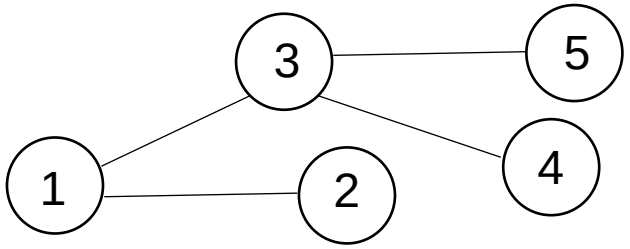


Robustez: si una MT M_1 con K_1 cintas tarda tiempo poly(n), una MT M_2 equivalente con K_2 cintas tarda tiempo poly(n). El retardo es a lo sumo **cuadrático.**

Otro ejemplo de lenguaje en la clase P

ACCES = {G | G es un grafo no dirigido con m vértices y tiene un camino del vértice 1 al vértice m}

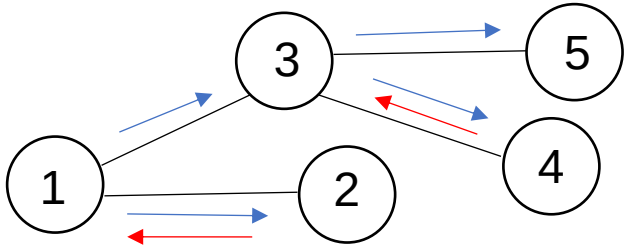
Representación de un grafo G



G = (V, E), con V el conjunto de vértices y E el conjunto de arcos

G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 3), (3, 4), (3, 5)})

MT M que decide ACCES en tiempo poly(n)

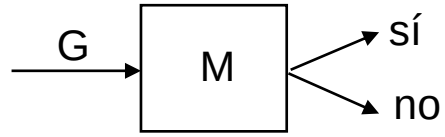


Recorrido DFS (en profundidad):

En el peor caso M recorre los arcos 2 veces

T(n) = O(|E|) = O(|G|) = O(n)

Por lo tanto, ACCES ∈ P



M tarda tiempo O(|G|)

Ejemplo de lenguaje que <u>no estaría</u> en P

SAT = {φ | φ es una fórmula booleana sin cuantificadores, con m variables, y es satisfactible}

P.ej.: $\varphi_1 = (x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3)$ es satisfactible con la asignación A = (V, V, V)

$\varphi_2 = (x_1 \wedge x_2 \wedge x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$ no es satisfactible con ninguna asignación

Una MT M que decide SAT emplea una tabla de verdad (**no se conoce otro algoritmo**). P.ej., para φ_2:

$(x_1 \wedge x_2 \wedge x_3)$			$\wedge$	$(\neg x_1 \vee \neg x_2 \vee \neg x_3)$		
V	V	V	**F**	V	V	V
V	V	F	**F**	V	V	F
V	F	V	**F**	V	F	V
V	F	F	**F**	V	F	F
F	V	V	**F**	F	V	V
F	V	F	**F**	F	V	F
F	F	V	**F**	F	F	V
F	F	F	**F**	F	F	F

2^m posibilidades
(por cada variable, los valores V y F)

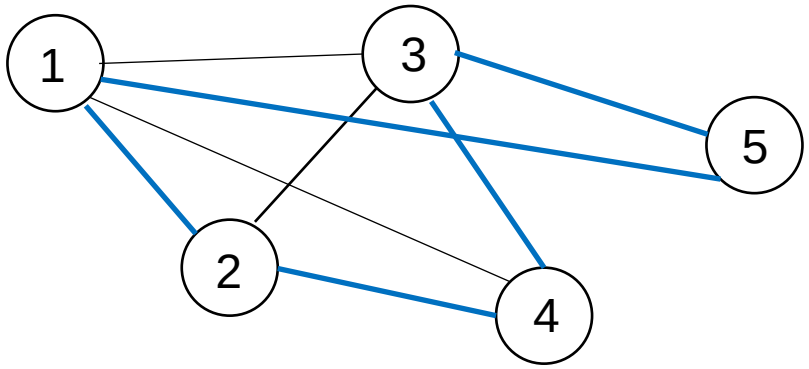
Cada evaluación se puede hacer en tiempo **$O(|\varphi|^2)$,** con el uso de una pila

Por lo tanto, M puede llegar a ejecutar $O(2^m.|\varphi|^2) = O(2^n.n^2)$ = **exp(n) pasos.**

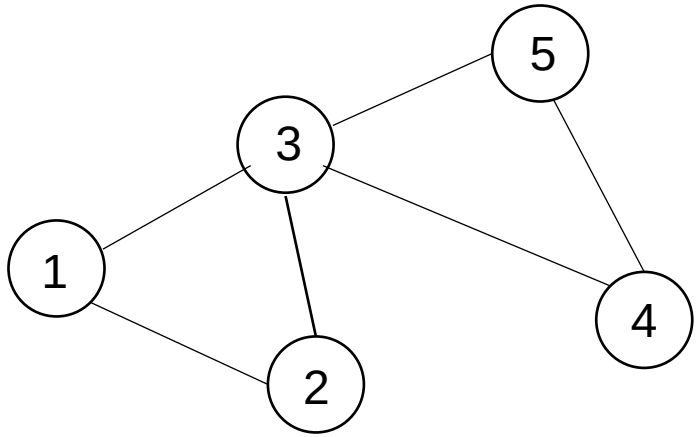
Otro ejemplo de lenguaje que no estaría en P

CH = {G | G es un grafo no dirigido con m vértices y tiene un Circuito de Hamilton}

Circuito de Hamilton (CdeH): recorre todos los vértices del grafo sin repetirlos salvo el primero al final.



Grafo con un CdeH, p.ej. (1, 2, 4, 3, 5)



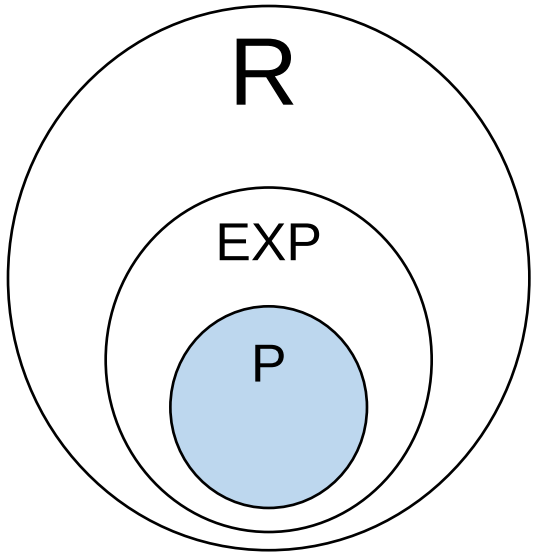
Grafo sin CdeH

Una MT M que decide CH prueba con todas las permutaciones de vértices (**no se conoce otro algoritmo**).

Dados m vértices existen **m!** permutaciones. Por ej., si m = 5: (1,2,3,4,5), (1,2,3,5,4), (1,2,4,3,5), etc.
Por otro lado, chequear si una permutación es un CdeH lleva tiempo $O(|V|.|E|) = O(|G|^2)$ = **$O(n^2)$**

Como $m! = m.(m-1).(m-2).(m-3)\ldots2.1 = O(m^m)$ = **$O(n^n)$,** M puede alcanzar los $O(n^n.n^2)$ = **exp(n) pasos.**

Primera versión de la jerarquía temporal



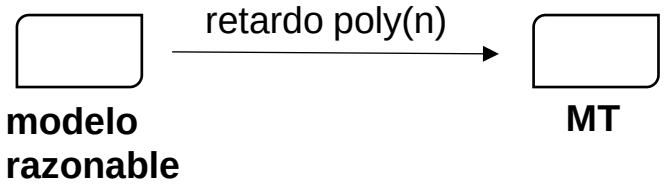
P es la clase de los lenguajes decidibles en tiempo poly(n) (**lenguajes tratables**)

EXP es la clase de los lenguajes decidibles en tiempo $O(c^{poly(n)})$

Se prueba que existen lenguajes fuera de P y fuera de EXP.

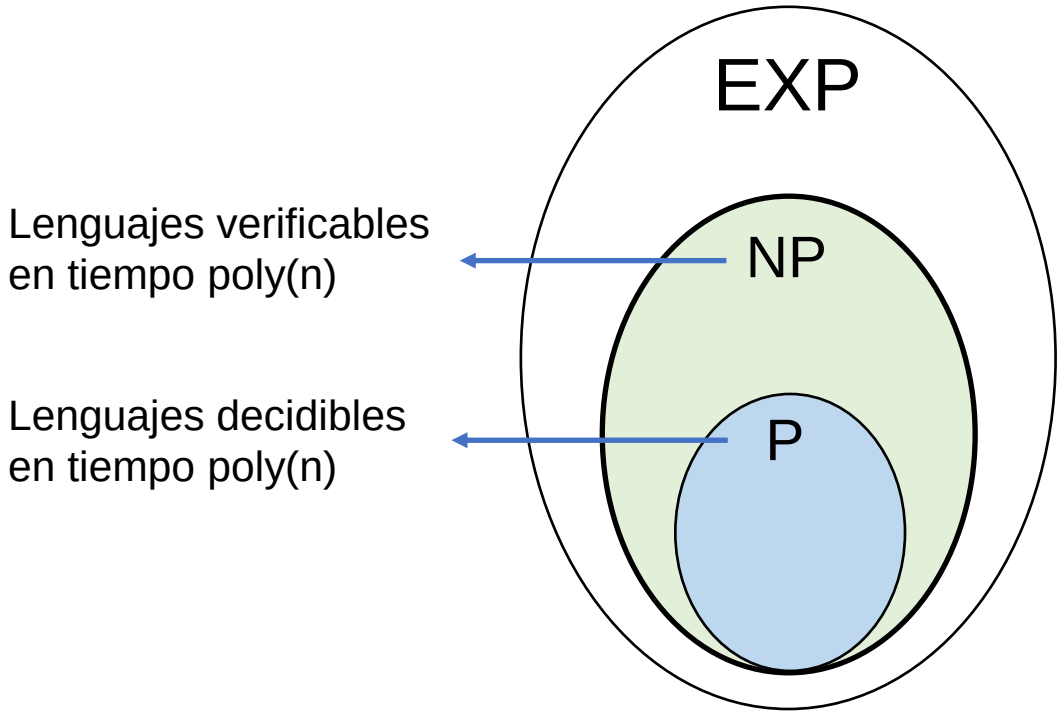
Tesis fuerte de Church-Turing (robustez de la clase P)
Si un lenguaje es decidible en tiempo poly(n) por un modelo computacional **razonable (realizable),** también es decidible en tiempo poly(n) por una MT **(¿hasta que las máquinas cuánticas sean una realidad?).**

Modelos computacionales razonables:
Máquinas RAM, programas Java, circuitos booleanos, etc.



Codificación razonable (realizable) de números
Cualquiera menos la de base unaria **(pensar p.ej. en 1.000.000.000 codificado como 1111111111111111…).**

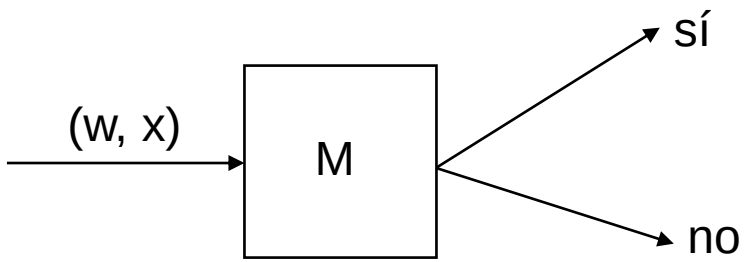
Segunda versión de la jerarquía temporal. La clase NP.



Asumiendo P ≠ NP
Asumiendo NP ≠ EXP

Un lenguaje L está en **NP** si existe una MT M tal que:
w ∈ L sii existe x tal que M acepta (w, x) en tiempo poly(n).

En otras palabras: toda cadena w de L cuenta con un **certificado** o **prueba** x que permite verificar **eficientemente** que pertenece a L.



M tarda tiempo poly(|w|)
|x| ≤ poly(|w|)

La MT M es un **verificador eficiente** de L.
x es un **certificado sucinto (o prueba sucinta)** de w.

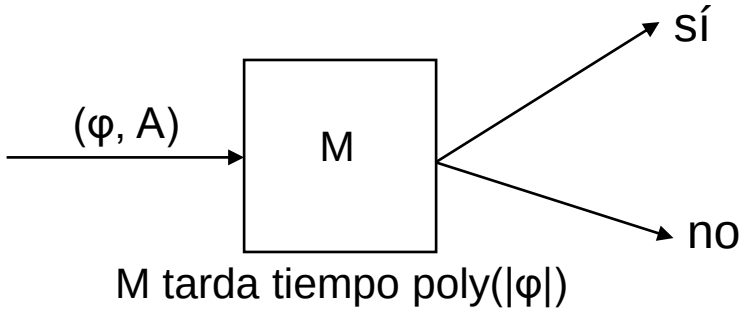
Aunque intuitivo, hasta hoy día no se ha encontrado una prueba de P ≠ NP.

Ejemplos de lenguajes en NP

- **SAT = {φ | φ es una fórmula booleana sin cuantificadores, con m variables, y es satisfactible}**

SAT no estaría en P:
Si φ tiene m variables, hay 2^m asignaciones A para chequear.

SAT está en NP:
Dada una fórmula booleana φ y una asignación A,
se puede **verificar en tiempo poly(n)** si A satisface φ.

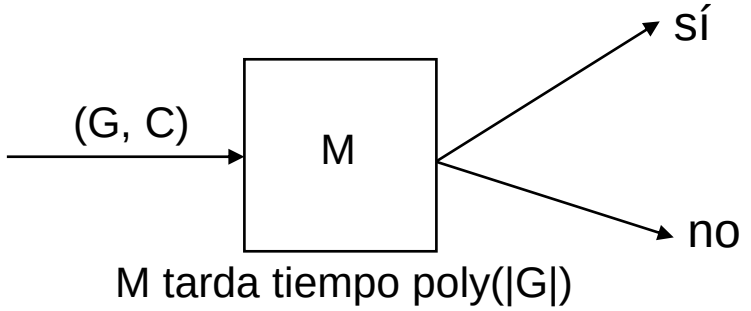


SAT^c no estaría en NP: para verificar si φ no es satisfactible hay que chequear 2^m asignaciones.

- **CH = {G : G es un grafo no dirigido con m vértices y tiene un circuito de Hamilton}**

CH no estaría en P:
Si G tiene m vértices, hay m! secuencias C de m vértices para chequear.

CH está en NP:
Dado un grafo G y una secuencia C de m vértices,
se puede **verificar en tiempo poly(n)** si C es un CdeH de G.

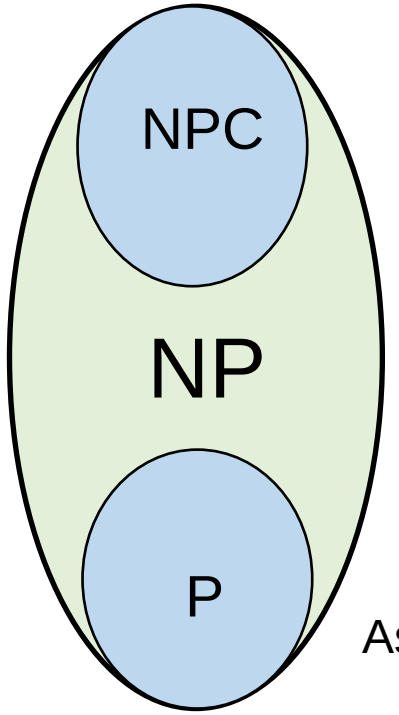


CH^c no estaría en NP: para verificar si G no tiene un CH hay que chequear m! secuencias.

CO-NP sería distinto a NP

Clase 6. Lenguajes NP-completos

- Una visión más detallada de NP:

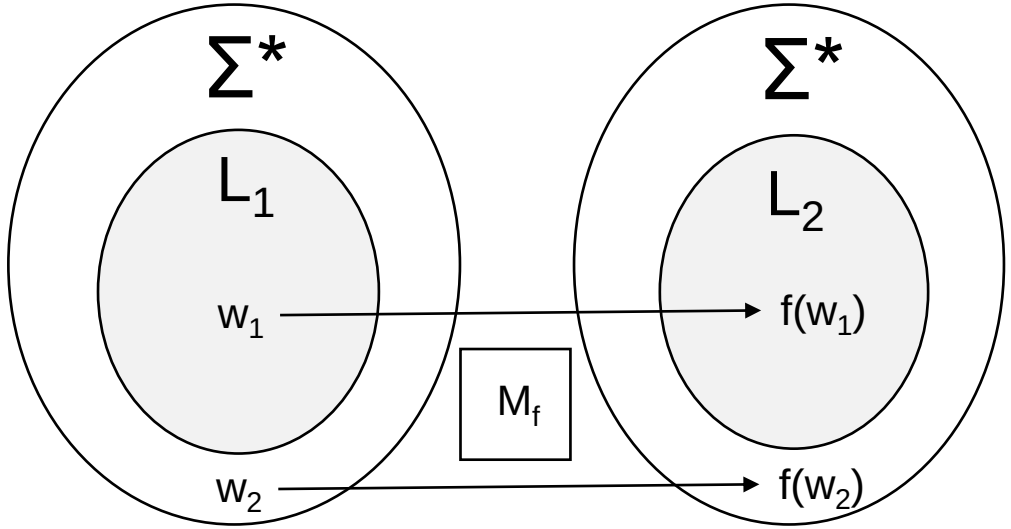


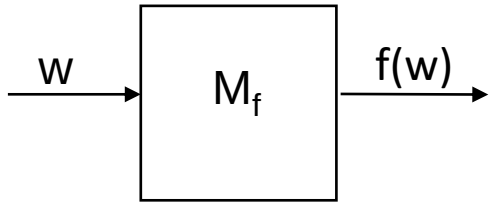
Asumiendo P ≠ NP

- Asumiendo la conjetura P ≠ NP, hay una manera de establecer que un lenguaje L de NP no está en P.
- Esto ocurre cuando se prueba que L es **NP-completo,** o que está en la clase **NPC.**
- Los lenguajes NP-completos son **los más difíciles** de la clase NP.

Reducciones polinomiales

- Para definir a los problemas NP-completos, tenemos que volver a utilizar **reducciones**, ahora **polinomiales**, es decir computables en tiempo poly(n):





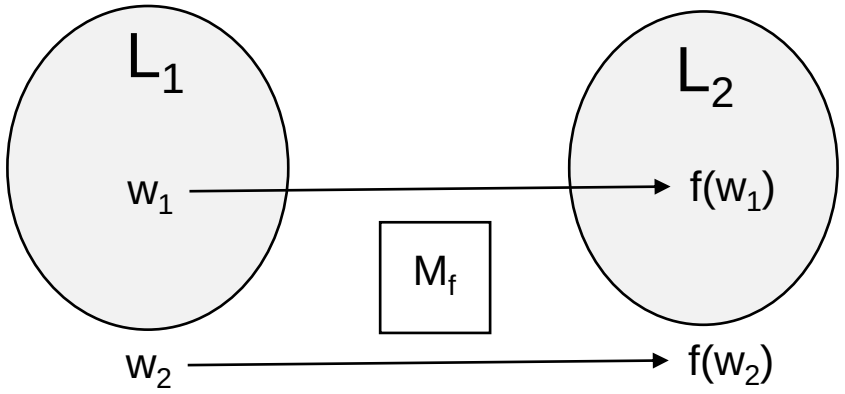
tiempo poly(|w|)

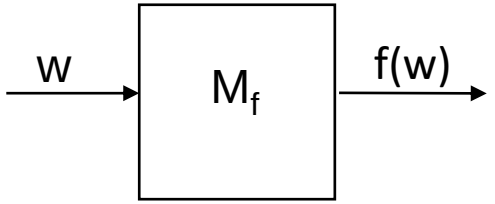
- La expresión $\mathbf{L_1 \leq_p L_2}$ establece que existe una reducción polinomial de L_1 a L_2.
- Se cumple, como en el caso general, que las reducciones polinomiales son **reflexivas, transitivas** y **no simétricas**.
- También como en el caso general, las reducciones polinomiales permiten **relacionar lenguajes.**

Reducciones polinomiales (continuación)

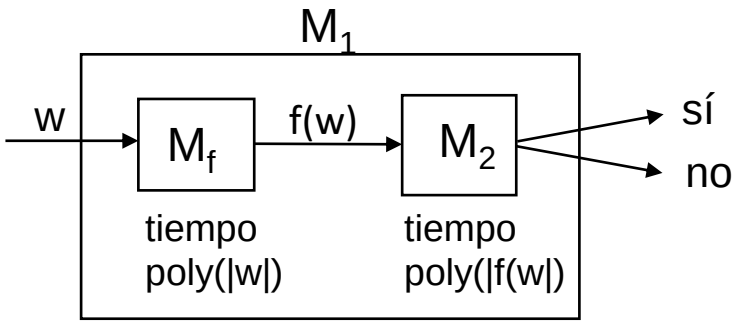
Teorema

(a) $L_1 \leq_p L_2$ y $L_2 \in P$: $L_1 \in P$
(b) $L_1 \leq_p L_2$ y $L_2 \in NP$: $L_1 \in NP$





Idea general de la prueba



(a) M_2 **decide** si $f(w) \in L_2$ y así M_1 **decide** si $w \in L_1$
(b) M_2 **verifica** si $f(w) \in L_2$ y así M_1 **verifica** si $w \in L_1$

Tiempo de M_1 = poly(|w|) + poly(|f(w)|)
Como |f(w)| = poly(|w|)
entonces **tiempo de M_1 = poly(|w|) + poly(poly(|w|)) = poly(|w|)**

Corolario

(a´) $L_1 \leq_p L_2$ y $L_1 \notin P$: $L_2 \notin P$
(b´) $L_1 \leq_p L_2$ y $L_1 \notin NP$: $L_2 \notin NP$

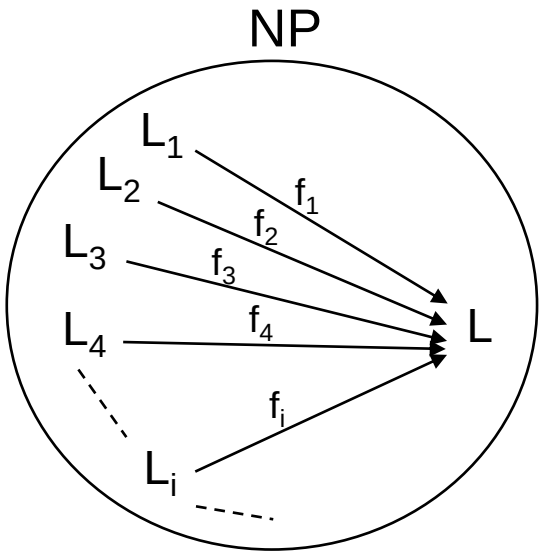
Si $L_1 \leq_p L_2$, L_2 es tan o más difícil que L_1

No puede ser que $L_1 \notin P$ y $L_2 \in P$
No puede ser que $L_1 \notin NP$ y $L_2 \in NP$

Definición de los lenguajes NP-completos

Un lenguaje L es **NP-completo**, o L ∈ NPC, sii:

a) **L ∈ NP**
b) Para todo L´ ∈ NP se cumple $L' \leq_p L$ (se dice que L es **NP-difícil**)



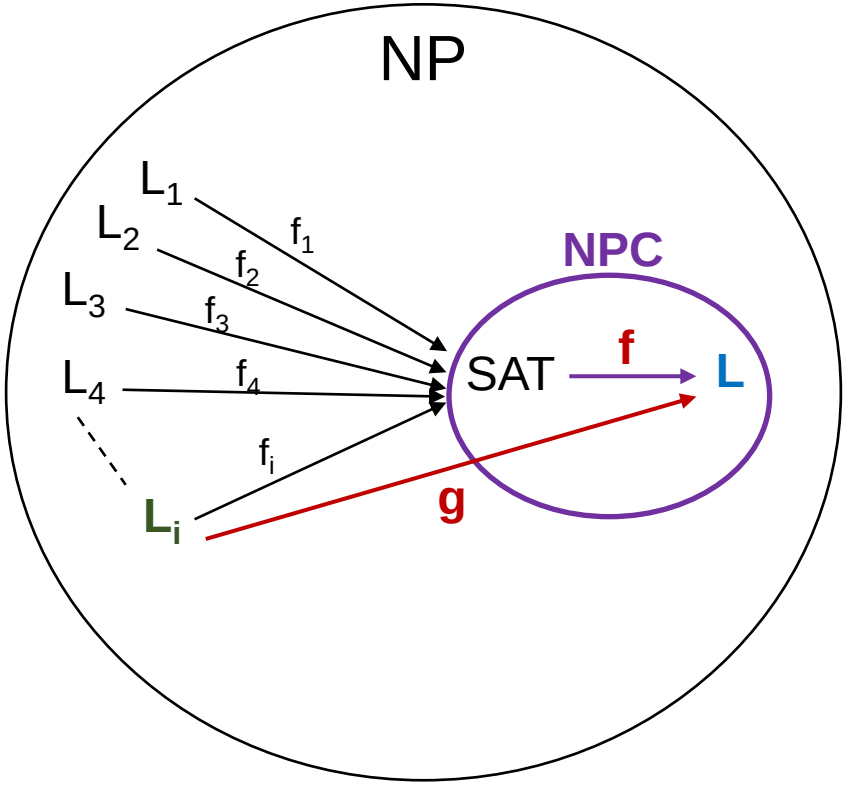
Todos los lenguajes de NP
se reducen polinomialmente a L

- Si L estuviera en P, entonces todos los lenguajes de NP estarían en P, y de esta manera, la relación entre P y NP sería P = NP. Resumiendo: **los lenguajes NP-completos no están en P a menos que P = NP.**

- Históricamente, Cook (EEUU) y Levin (Rusia), en 1971, encontraron casi en simultáneo un primer lenguaje NP-completo, **el lenguaje SAT:** SAT = {φ | φ es una fórmula booleana sin cuantificadores y es satisfactible}.

Cómo poblar la clase NPC

1. Sea **L** un lenguaje de NP
2. Sea **f** una reducción polinomial de SAT al lenguaje L
3. Sea **g** la reducción polinomial obtenida componiendo la reducción polinomial f_i con la reducción **f**
4. Lo anterior vale para todos los lenguajes L_i, lo que significa que **el lenguaje L es NP-completo**



Resumiendo:

$L_1 \in NPC$

$L_2 \in NP$

$L_1 \leq_p L_2$

$L_2 \in NPC$

Ejemplos clásicos

SAT = {φ | φ es una fórmula booleana sin cuantificadores, **satisfactible**}

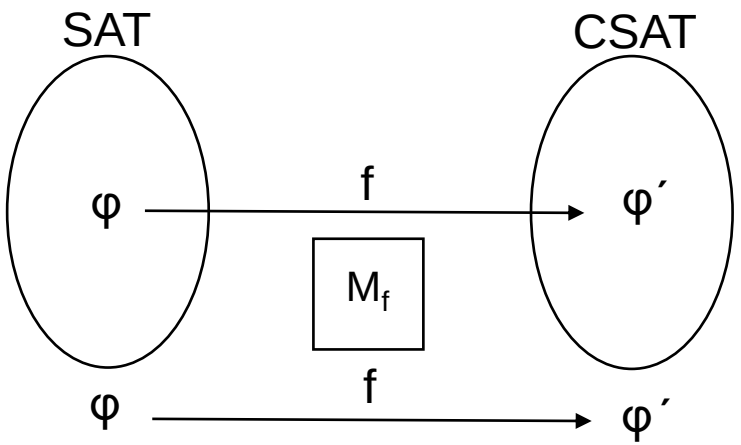
$$x_1 \vee (x_2 \wedge \neg x_3 \vee x_1) \wedge (x_3 \vee \neg x_1 \wedge x_3) \vee (x_5 \vee \neg x_1) \wedge \neg x_4$$

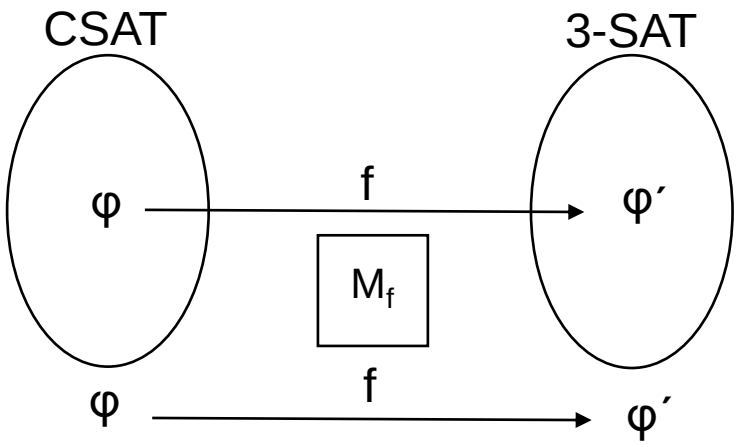
CSAT = {φ | φ es una fórmula booleana sin cuantificadores en la **forma normal conjuntiva (FNC), satisfactible**}

$$x_1 \wedge (x_1 \vee x_2 \vee x_5 \vee \neg x_3) \wedge (\neg x_5 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_4 \vee \neg x_3 \vee x_3)$$

3-SAT = {φ | φ es una fórmula booleana sin cuantificadores en FNC **con tres literales por cláusula, satisfactible**}

$$(x_1 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee x_1)$$



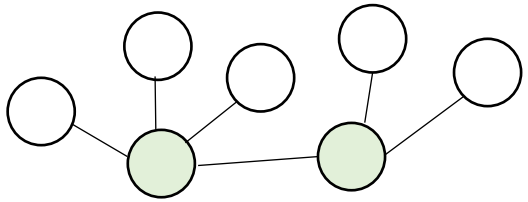


Ejemplos clásicos (continuación)

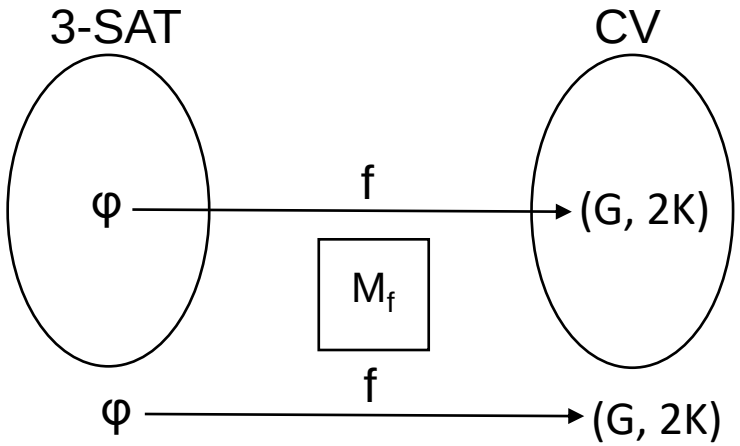
3-SAT = {φ | φ es una fórmula booleana sin cuantificadores en FNC **con tres literales por cláusula, satisfactible**}

$$(x_1 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee x_1)$$

CV = {(G, K) | G es un grafo y tiene un **cubrimiento de vértices de tamaño K**}

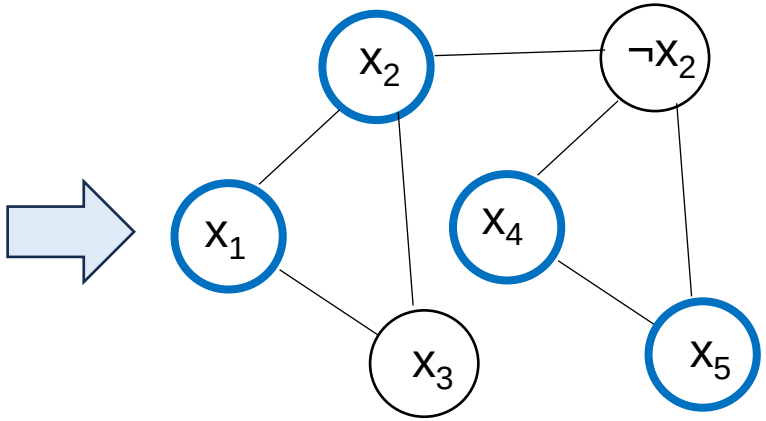


Ejemplo de cubrimiento de vértices de tamaño 2 (con 2 vértices toca todos los arcos)



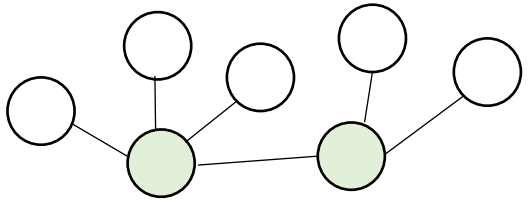
K = nro de cláusulas

$$(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee \neg x_2)$$

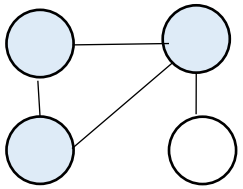


Ejemplos clásicos (continuación)

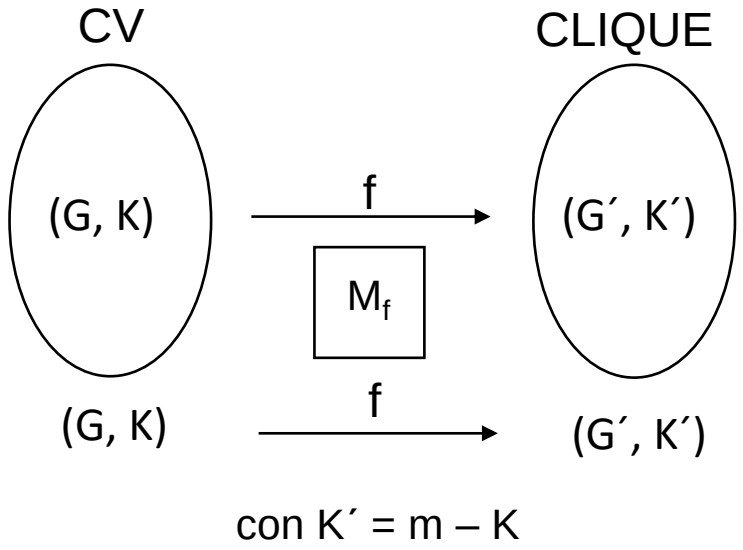
CV = {(G, K) | G es un grafo y tiene un **cubrimiento de vértices de tamaño K**}

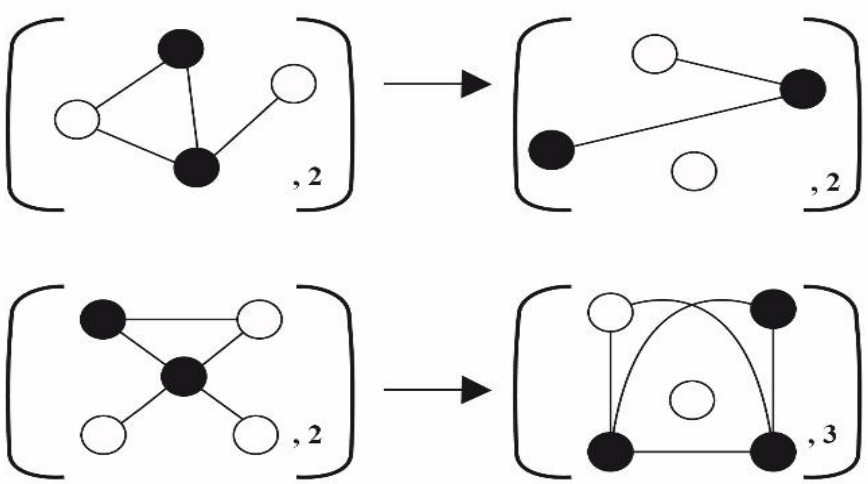


CLIQUE = {(G, K) | G es un grafo y tiene un **clique de tamaño K**}



Ejemplo de clique de tamaño 3



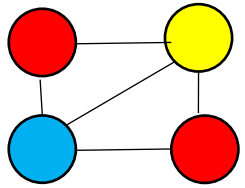


Ejemplos clásicos (continuación)

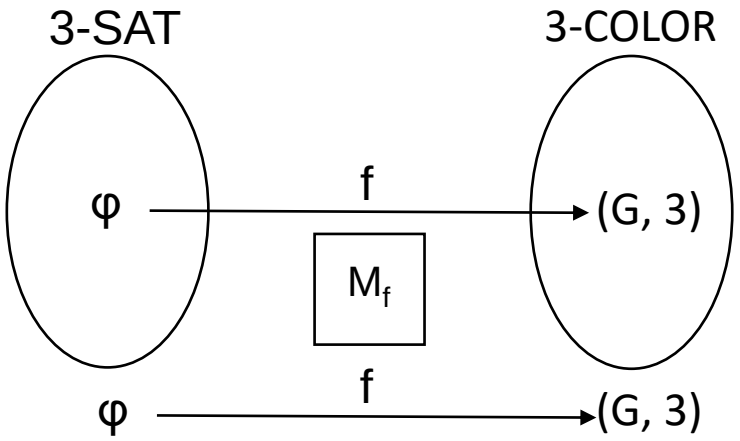
3-SAT = {φ | φ es una fórmula booleana sin cuantificadores en FNC **con tres literales por cláusula satisfactible**}

$$(x_1 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee x_1)$$

K-COLOR = {(G, K) | G es un grafo y es coloreable con K colores sin producir vértices adyacentes con igual color}



Ejemplo de grafo coloreable con 3 colores

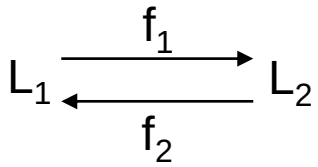


Teorema de los Cuatro Colores

Todo grafo planar se puede colorear con cuatro colores sin producir vértices adyacentes con el mismo color.

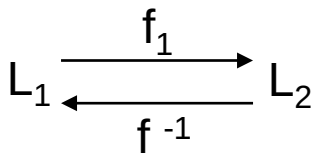
Dos características de los problemas NP-completos

- Por definición, para todo par de lenguajes L_1 y L_2 de NPC se cumple $\mathbf{L_1 \leq_p L_2}$ **y** $\mathbf{L_2 \leq_p L_1}$



No necesariamente f_2 es la función inversa de f_1.

Sin embargo, **todos los lenguajes NP-completos conocidos cumplen dicha propiedad (son p-isomorfos).**



Se conjetura que todos los lenguajes NP-completos cumplen la propiedad de p-isomorfismo.
Se prueba que si se cumple la conjetura, entonces **P ≠ NP.**

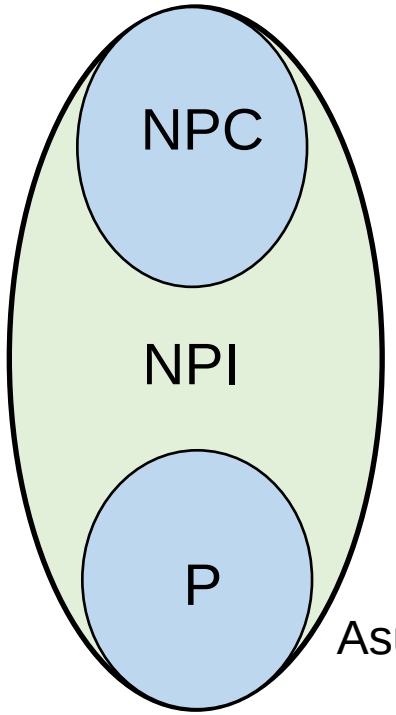
- **Todos los lenguajes NP-completos conocidos son densos**.

Un lenguaje es denso si para todo n tiene exp(n) cadenas de longitud a lo sumo n.
Un lenguaje es disperso en caso contrario (para todo n tiene poly(n) cadenas de longitud a lo sumo n).

Se conjetura que todos los lenguajes NP-completos son densos.
Se prueba que si existe un lenguaje NP-completo disperso, entonces **P = NP.**

Clase 7 parte 1. Las clases NPI y CO-NP

- Una visión aún más detallada de NP:



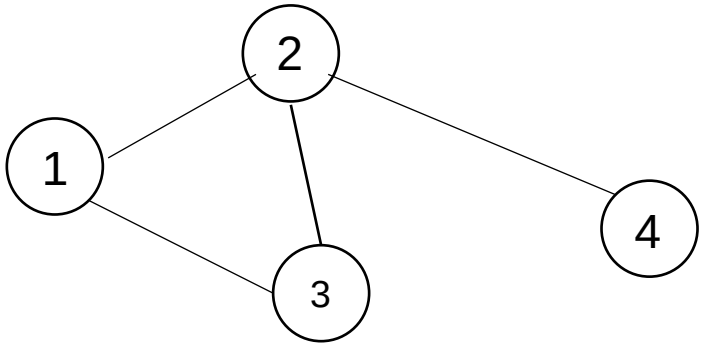
NP

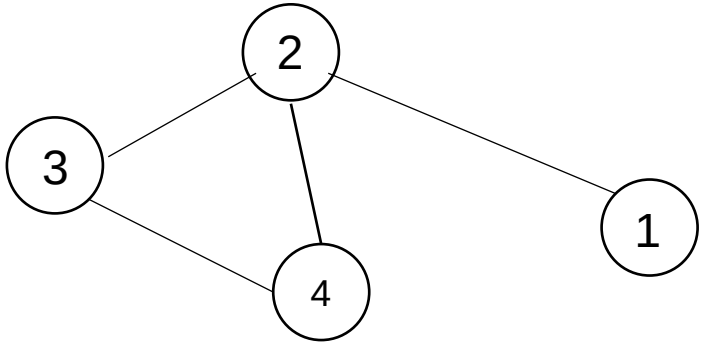
Asumiendo P ≠ NP

- Asumiendo P ≠ NP, se prueba que además de P y NPC, NP incluye una tercera clase de lenguajes: **NPI.**
- Los lenguajes de NPI **no son ni tan fáciles como los de P ni tan difíciles como los de NPC.**

El problema de los grafos isomorfos (candidato a estar en NPI)

ISO = {(G_1, G_2) | G_1 y G_2 son grafos isomorfos, es decir son idénticos salvo por el nombre de sus arcos}



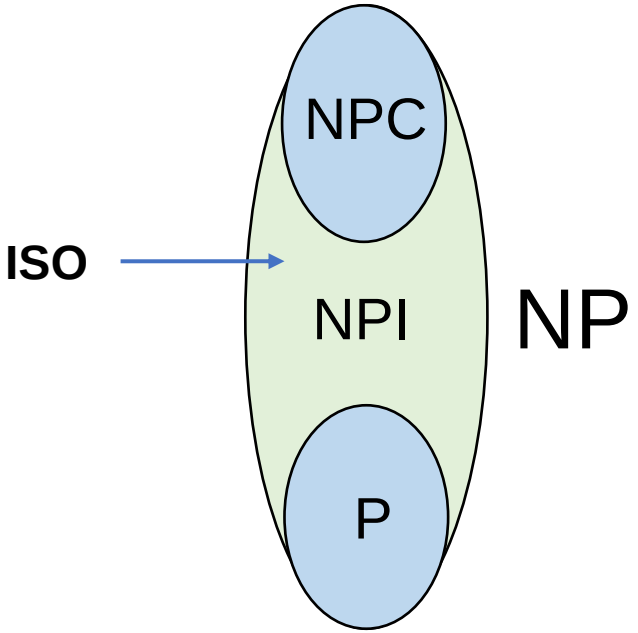


ISO está en NP
los certificados suscintos son permutaciones de V

ISO no estaría en P
en el peor caso hay que probar con todas las permutaciones de V

ISO no estaría en NPC
no se ha encontrado un lenguaje NP-completo L tal que L $\leq_P$ ISO

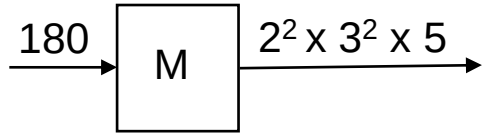
ISO^C no estaría en NP
no se han encontrado sucintos para ISO^C



Asumiendo P ≠ NP

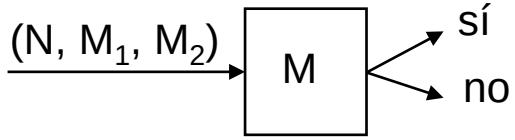
El problema de la factorización (otro candidato a estar en NPI)

Dado N, hay que encontrar todos sus divisores primos. P.ej, **$180 = 2^2 \times 3^2 \times 5$.**



Llevado a un problema de decisión tiene la siguiente forma:

FACT = {(N, M_1, M_2) | N tiene un divisor primo entre M_1 y M_2}



- Como la sospecha es que no está en P, se lo usa para **encriptar mensajes**:
  Dado un número N muy grande,
  si $N = N_1 \times N_2$, y N_1 y N_2 son números primos de tamaño similar,
  resulta muy difícil obtener N_1 y N_2 conociendo solamente N.

- En base a esto, un esquema de seguridad habitual consiste en
  encriptar mensajes con N (que conoce todo el mundo),
  y **desencriptarlos con N_1 y N_2** (qué sólo conoce el receptor)

mensaje encriptado (sistema RSA)

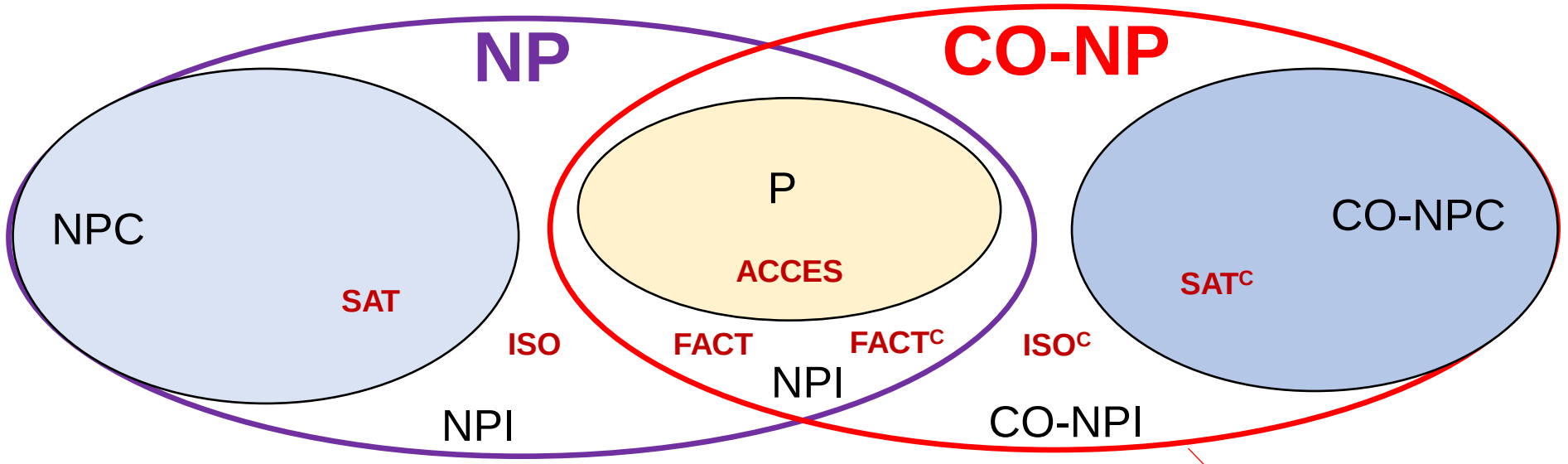
10011110101000011101011100011111100101010

se encripta con N — clave pública
se desencripta con N_1 y N_2 — clave privada

- Si se probase que FACT está en P, habría que reemplazar dicho esquema de seguridad.

- En 1994, P. Shor encontró un **algoritmo cuántico** que factoriza los números en tiempo **poly(n).**
  ¿Será que efectivamente lo cuántico acelera exponencialmente los algoritmos clásicos?
  ¿O Será que la factorización está en P?

Ultima visión de la jerarquía temporal



Asumiendo P ⊂ NP
Asumiendo P ⊂ NP ∩ CO-NP
Asumiendo NP ≠ CO-NP

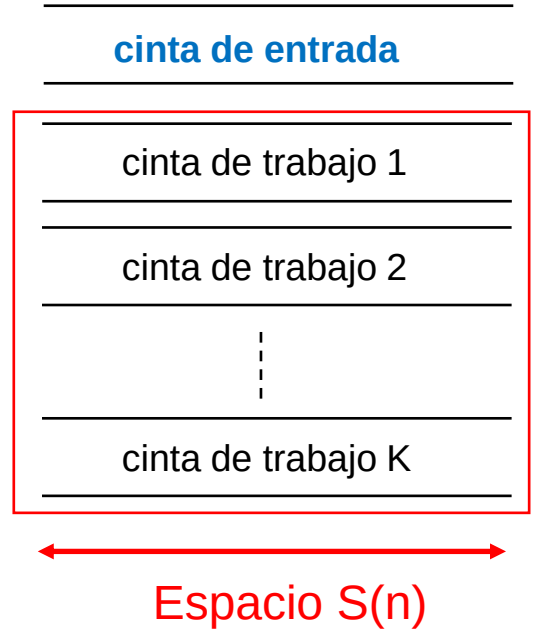
Jerarquía de lenguajes (de menor a mayor dificultad)

1) P
2) (NP ∩ CO-NP) – P
3) NPI - CO-NP
4) NPC
5) CO-NPI – NP
6) CO-NPC

Las cadenas de los lenguajes de CO-NP no tendrían certificados sucintos. Por ejemplo, para verificar si una fórmula booleana no tiene una asignación que la satisface, hay que chequear todas las posibles asignaciones, las cuales suman una cantidad exponencial.

Clase 7 parte 2. Complejidad espacial

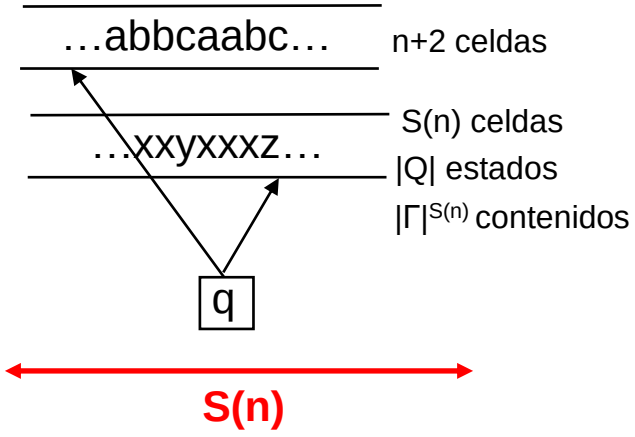
- Se consideran MT tales que la cinta de entrada es de **sólo lectura**. El resto son **cintas de trabajo**.
- Una MT ocupa **espacio S(n)** sii en todas sus cintas de trabajo (no cuenta la cinta de entrada) ocupa **a lo sumo S(n) celdas**, siendo n como siempre el tamaño de las cadenas de entrada.
- La cinta de entrada de sólo lectura permite espacios **menores que O(n).**



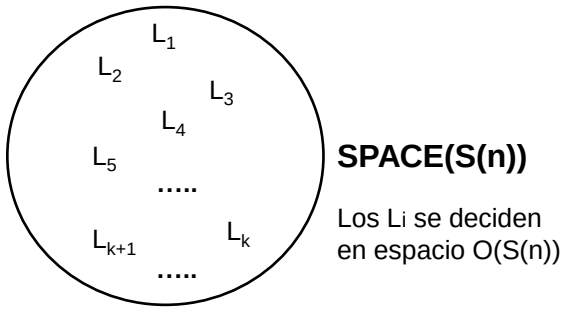
- Una MT M que ocupa espacio S(n) **puede no parar**, pero dada M, existe una MT M´ equivalente que ocupa espacio S(n) y **para siempre.**

  P.ej., una MT con 1 cinta de entrada de sólo lectura y 1 cinta de trabajo entró en loop si hizo más de **$(n+2).S(n).|Q|.|\Gamma|^{S(n)} = O(c^{S(n)})$ pasos.**
  Tener en cuenta entonces que:

  tiempo T(n) implica espacio S(n)
  espacio S(n) implica tiempo $O(c^{S(n)})$



- **Un lenguaje L pertenece a la clase SPACE(S(n))** sii existe una MT que decide L en espacio O(S(n)).



Ejemplo. El lenguaje de los palíndromos o capicúas.

L = {wcw^R | w tiene cero o más símbolos *a* y b, y w^R es la cadena inversa de w}. P.ej., abbbcbbba está en L.

L se puede decidir en espacio **O(n)**. Pero en realidad alcanza con espacio **O(log$_2$n).** Una MT M que decide L en espacio O(log$_2$n), usando codificación binaria, se comporta de la siguiente manera:

1. Hacer i = 1 en la cinta 1.
2. Hacer j = n en la cinta 2, con n = |w|. Si j es par, rechazar.
3. Copiar el símbolo i de w en la cinta 3.
4. Copiar el símbolo j de w en la cinta 4.
5. Si i = j : si los símbolos son **c,** aceptar, si no, rechazar.
   Si i ≠ j : si los símbolos no son igualmente ***a*** o **b**, rechazar.
6. Hacer i = i + 1 en la cinta 1.
7. Hacer j = j – 1 en la cinta 2.
8. Volver al paso 3.

Ejemplo (1ra iteración)

	a b b b c b b b a	
1	i = 1	
2	j = 9	
3	a	
4	a	

Ejemplo (2da iteración)

	a b b b c b b b a	
1	i = 2	
2	j = 8	
3	b	
4	b	

Ejemplo (última iteración)

	a b b b c b b b a	
1	i = 5	
2	j = 5	
3	c	
4	c	

La MT M ocupa el espacio de los contadores i y j, que en binario miden **O(log$_2$n)**, **más 2 celdas** para alojar a los símbolos que se van comparando en cada iteración.

Así, **L ∈ SPACE(log$_2$n).** A esta clase también se la llama **LOGSPACE.**

Jerarquía espacial

- **LOGSPACE** es la clase de los lenguajes aceptados en espacio **$O(\log_2 n)$**
- **PSPACE** es la clase de los lenguajes aceptados en espacio **poly(n)**
- **EXPSPACE** es la clase de los lenguajes aceptados en espacio **exp(n)**

- Por lo dicho antes: **espacio S(n) implica tiempo $O(c^{S(n)})$, con c constante**

  Así, si en particular una MT M ocupa espacio **$\log_2 n$,** entonces M tarda tiempo **$O(c^{\log_2 n})$**

  Y como $\mathbf{c^{\log_2 n}} = \mathbf{n^{\log_2 c}}$ = poly(n), queda: **LOGSPACE** $\subseteq$ **P**

Los problemas tratables en espacio son los que pertenecen a la clase LOGSPACE

Existe una clase NLOGSPACE (homóloga a la clase NP), también incluida en P.

Tiempo T(n) implica espacio T(n).
Espacio S(n) implica tiempo $O(c^{S(n)})$, para alguna constante c.

Ejemplo. El lenguaje QSAT.

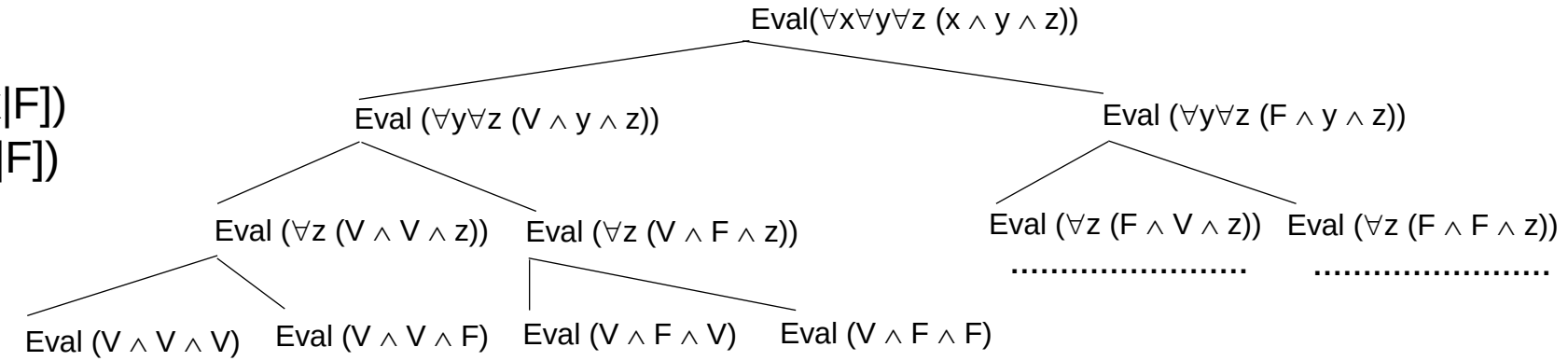
QSAT = {φ | φ es una fórmula booleana con cuantificadores, no tiene variables libres, y es verdadera}.

- Por ejemplo, $\varphi_1 = \exists x \exists y \exists z: x \wedge y \wedge z$, es verdadera

  $\varphi_2 = \forall x \forall y \forall z: x \wedge y \wedge z$, es falsa

- **QSAT pertenece a PSPACE.** La prueba se basa en la construcción de una función recursiva **Eval**. La idea de la recursión es la siguiente (usamos como ejemplo la fórmula φ_2):

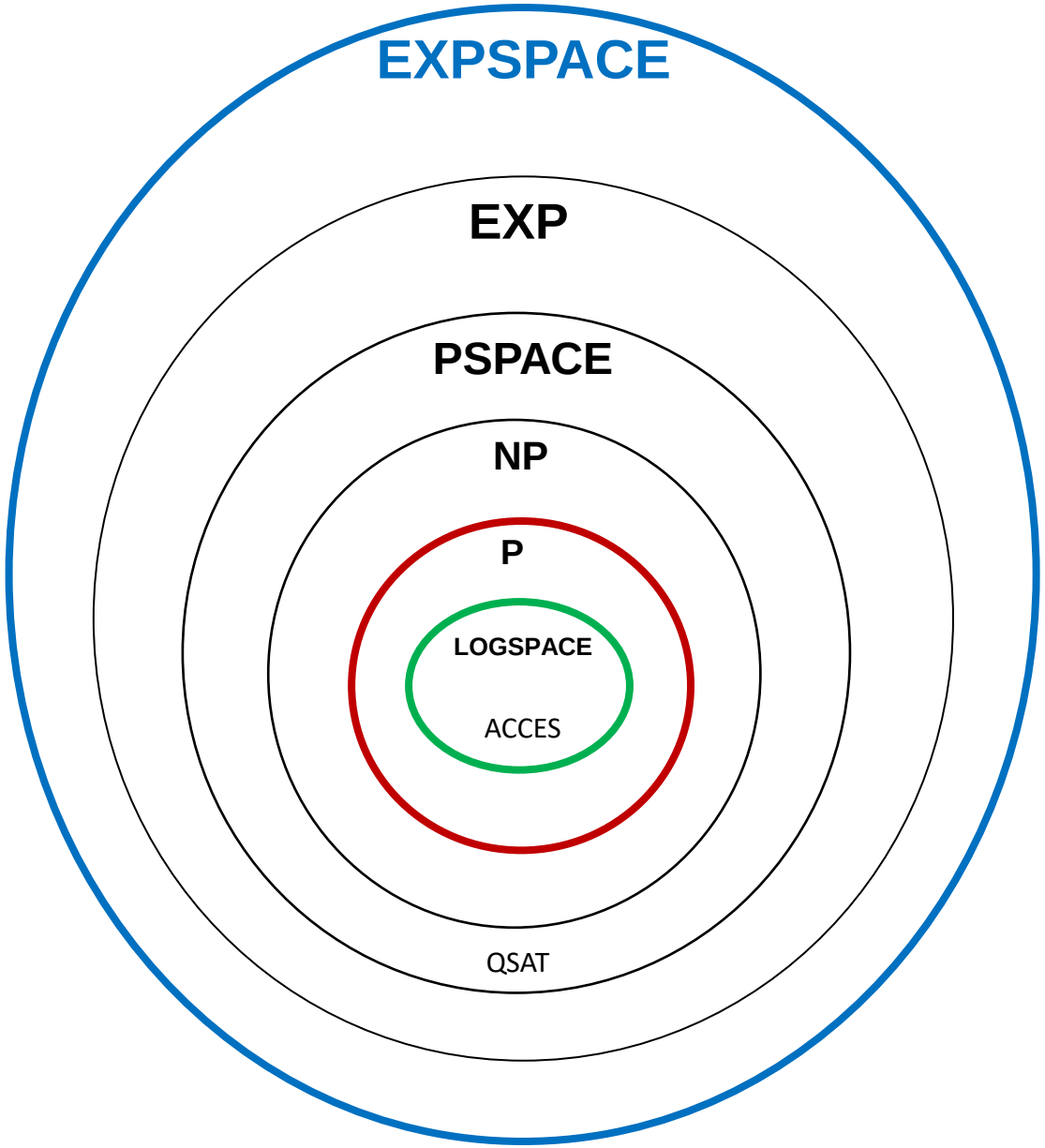
$Eval(\varphi,\forall x) = Eval(\varphi[x|V]) \wedge Eval(\varphi[x|F])$
$Eval(\varphi,\exists x) = Eval(\varphi[x|V]) \vee Eval(\varphi[x|F])$
$Eval(\varphi_1,\varphi_2,\wedge) = Eval(\varphi_1) \wedge Eval(\varphi_2)$
$Eval(\varphi_1,\varphi_2,\vee) = Eval(\varphi_1) \vee Eval(\varphi_2)$
$Eval(\varphi,\neg) = \neg Eval(\varphi)$
$Eval(V) = V$
$Eval(F) = F$



La MT correspondiente <u>reutiliza</u> espacio, característica esencial en la complejidad espacial

La cantidad de cuantificadores más la cantidad de conectivos de φ es a lo sumo $|\varphi| = n$.
Así, tanto la profundidad de la pila como el espacio ocupado en cada instancia miden O(n).
Por lo tanto, **Eval consume espacio $O(n^2)$.**

Jerarquía espacio-temporal



- QSAT está entre los lenguajes más difíciles de PSPACE.
- En efecto, **QSAT es PSPACE-completo**, todos los lenguajes de PSPACE se reducen polinomialmente a QSAT.
- QSAT es una instancia de un problema más general: **la búsqueda de una estrategia ganadora en una competencia entre dos jugadores J_1 y J_2 (ajedrez, damas, go, hexágono, geografía, etc.):**

  ¿Existe una jugada 1 de J_1 tal que para toda jugada 1 de J_2 existe una jugada 2 de J_1 tal que para toda jugada 2 de J_2 existe una jugada 3 de J_1 tal que para toda jugada 3 de J_2 … el jugador J_1 gana?

  En el caso particular de QSAT se puede plantear que ∃ busca que φ sea verdadera y ∀ que φ sea falsa.

  $$\varphi = [\exists x_1 \forall x_2 \exists x_3 \forall x_4 \exists x_5 \forall x_6 \ldots : \Psi(x_1, x_2, x_3, x_4, x_5, x_6, \ldots)]$$