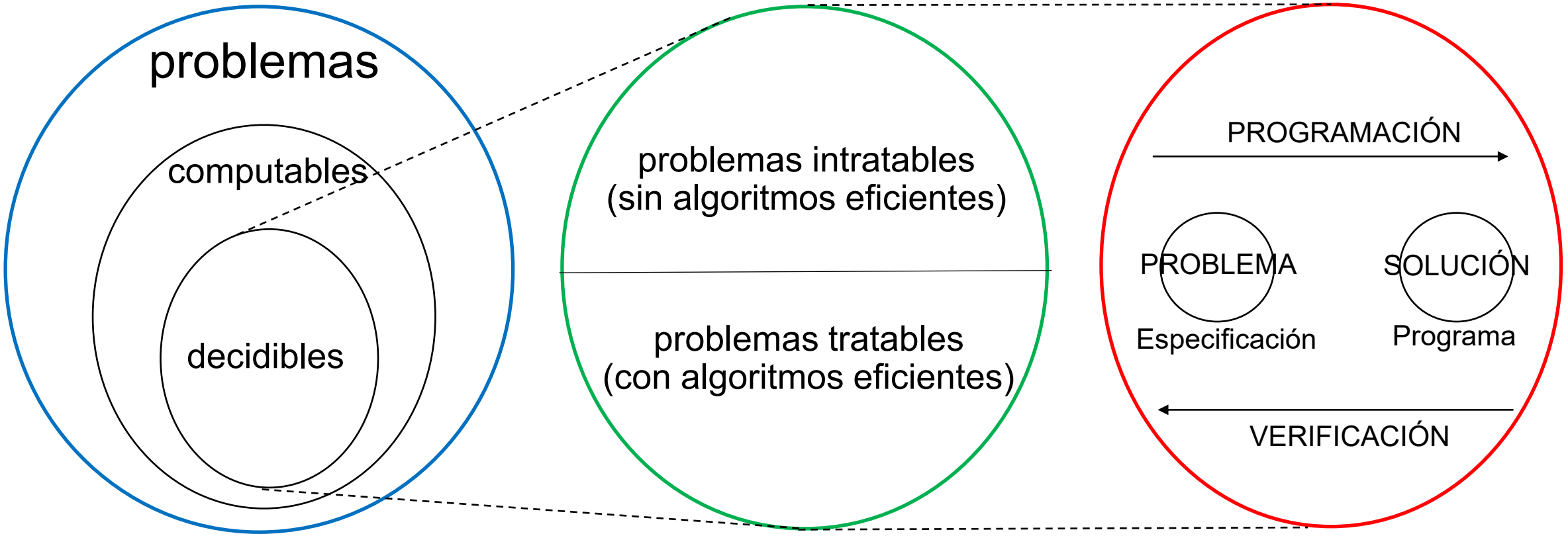
Teoría de la Computación y Verificación de Programas

Repaso

Las tres partes de la materia



Computabilidad
(años 1930)

Complejidad Computacional
(años 1960)

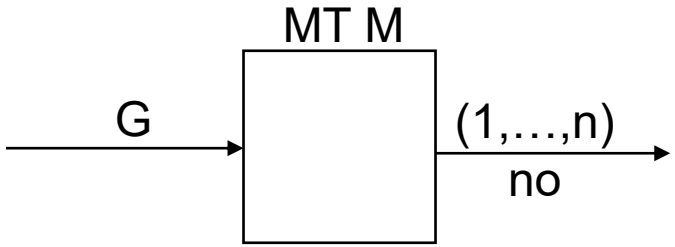
Correctitud de Programas
(años 1970)

Matemática y Lógica

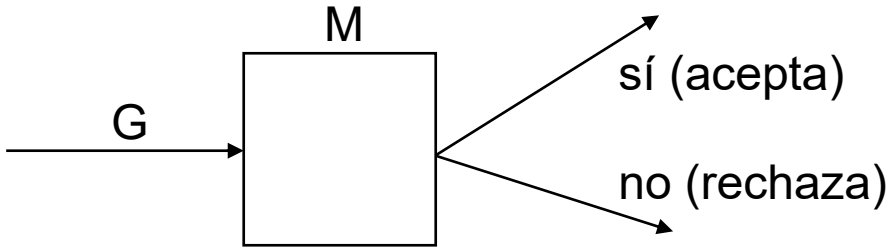
Parte 1
Computabilidad

Clases 1 a 4

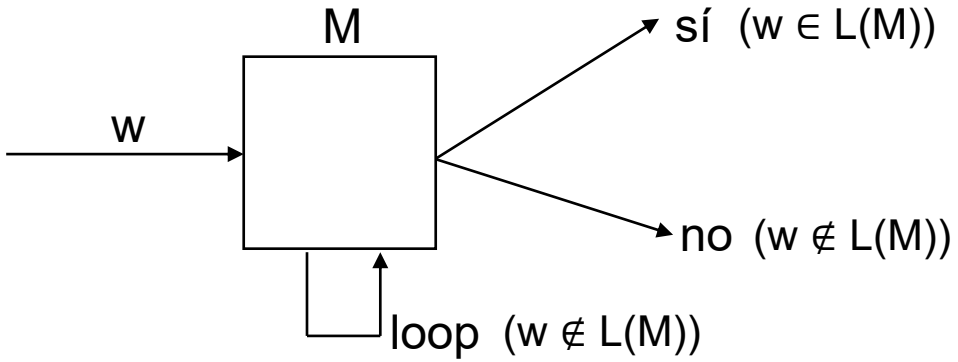
Clase 1. Máquina de Turing (MT)



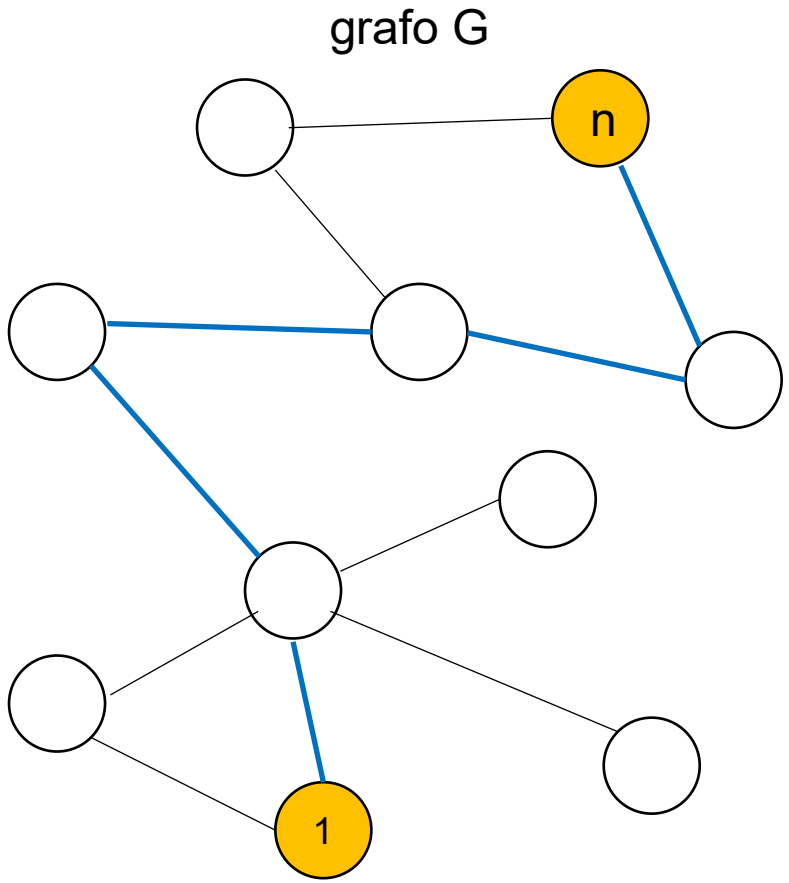
Problema de búsqueda



Problema de decisión (M acepta o reconoce un lenguaje)

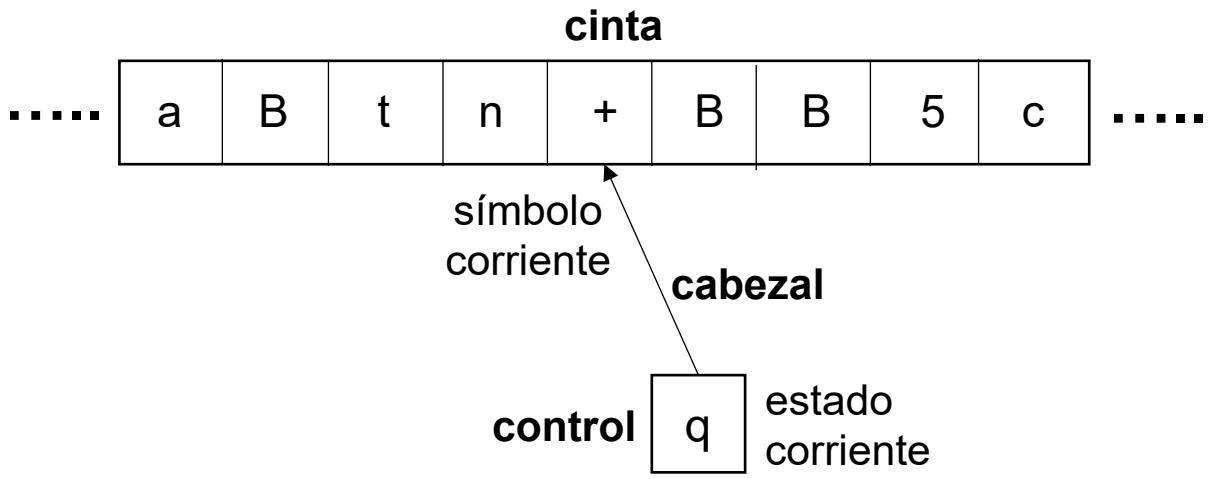


Posibilidad de no detención



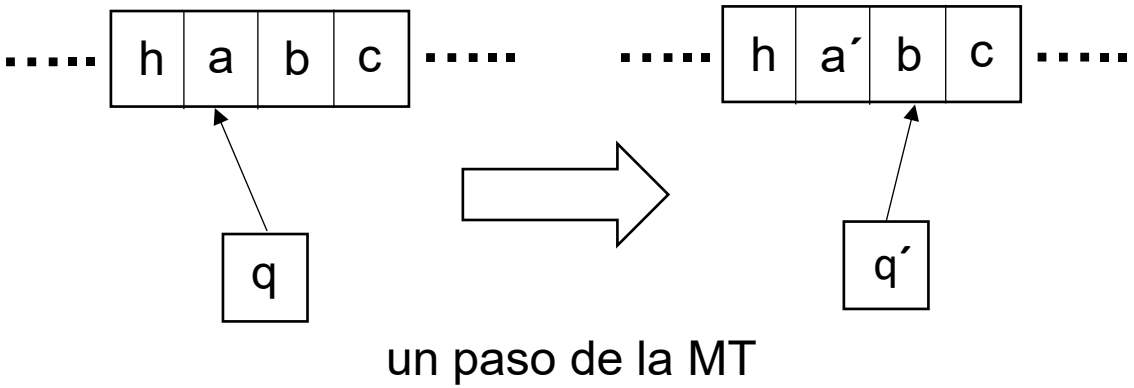
Grafo con un camino del vértice 1 al vértice n

Definición

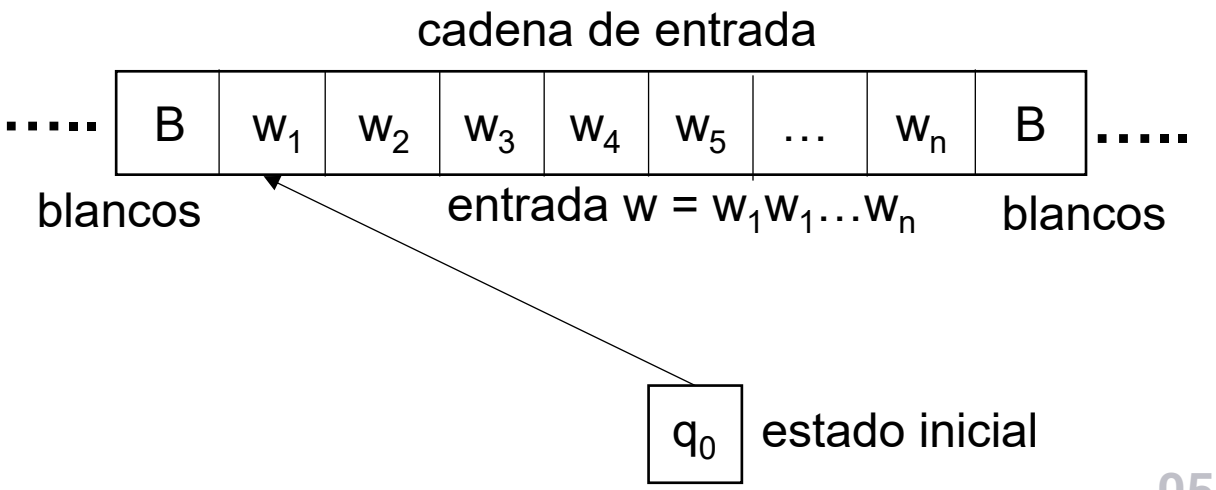


Tesis de Church-Turing
¡Todo lo computable puede ser computado por una Máquina de Turing!

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



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.

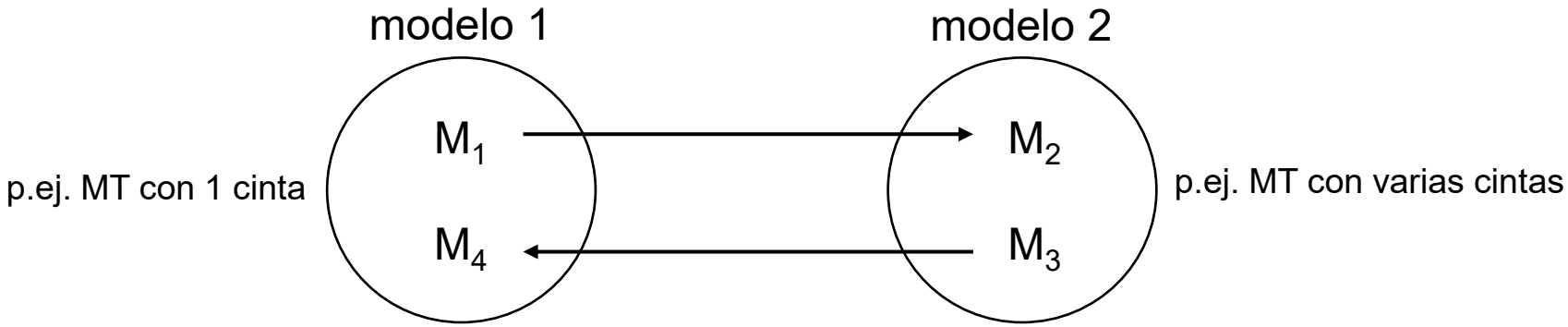
Forma alternativa de describir la función de transición δ

	a	b	α	β	B
q_0	q_b, α, R				
q_a	q_b, α, R			q_H, β, R	
q_b	q_b, a, R	q_L, β, L		q_b, β, R	
q_L	q_L, a, L		q_a, α, R	q_L, β, L	
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.

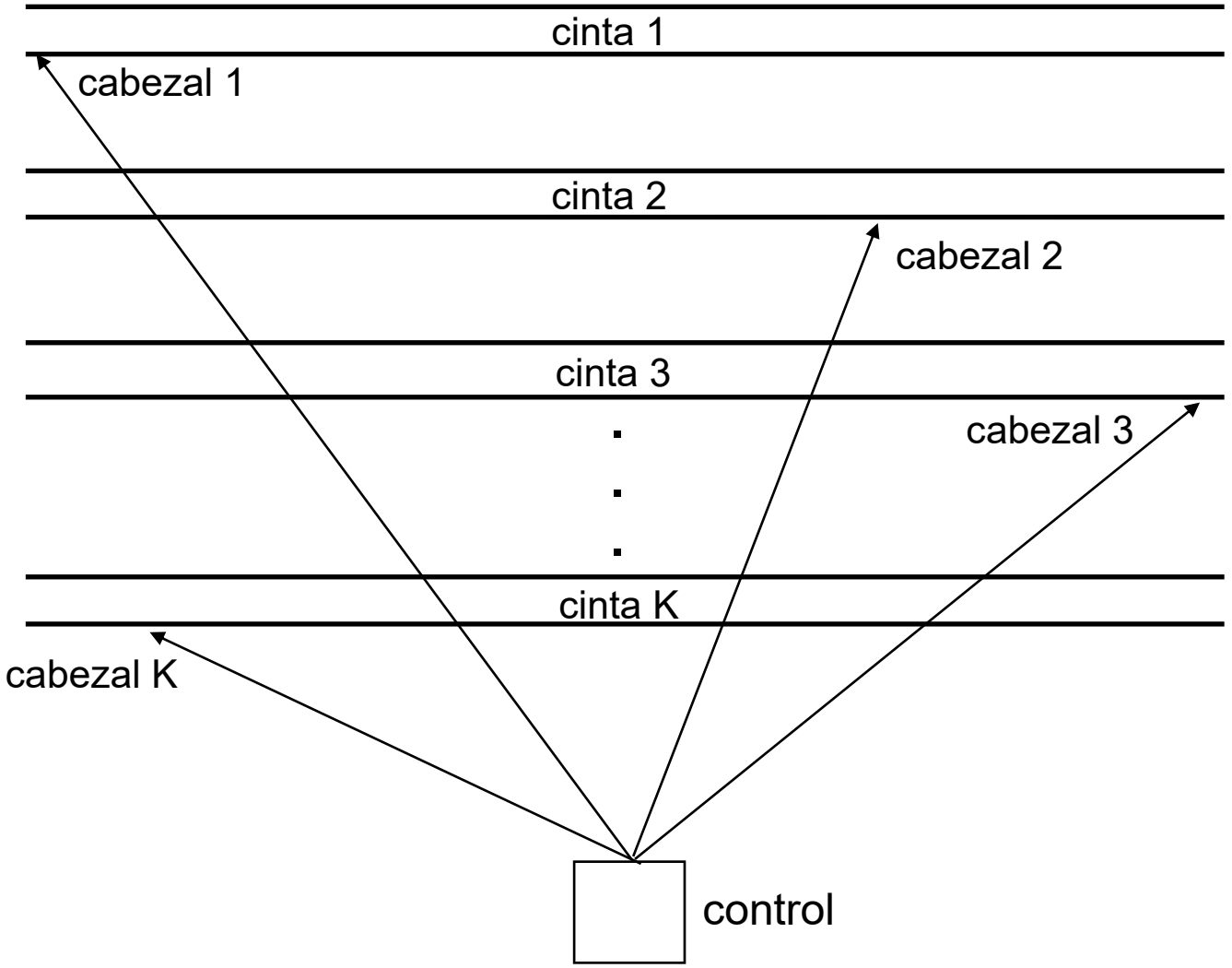


- Ejemplos de modelos de MT equivalentes al modelo de MT con **1 cinta**: MT con **varias cintas**, MT con **cintas semi-infinitas**, MT con **2 cintas y un solo estado**, MT **no determinísticas** (a partir de un estado y un símbolo pueden avanzar de distintas maneras), etc.

- 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



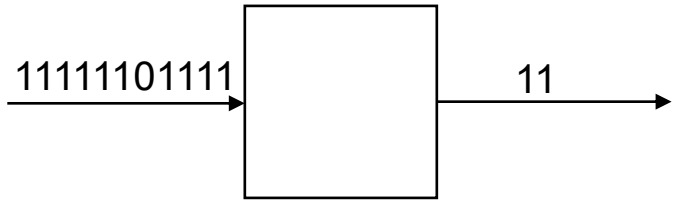
Se puede simular con una MT de una cinta, con un retardo cuadrático.

Otras visiones de MT

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

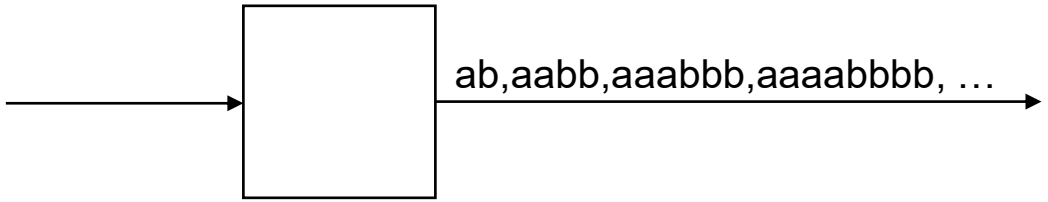
Ejemplo: construir una MT que reste 2 números representados en notación unaria y separados por un cero.
Por ejemplo, dada la entrada 11111101111, obtener la salida 11 (6 – 4 = 2).
<u>Idea general</u>: tachar el primer 1 antes del 0, luego el primer 1 después del 0, luego el segundo 1 antes del 0, y así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

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



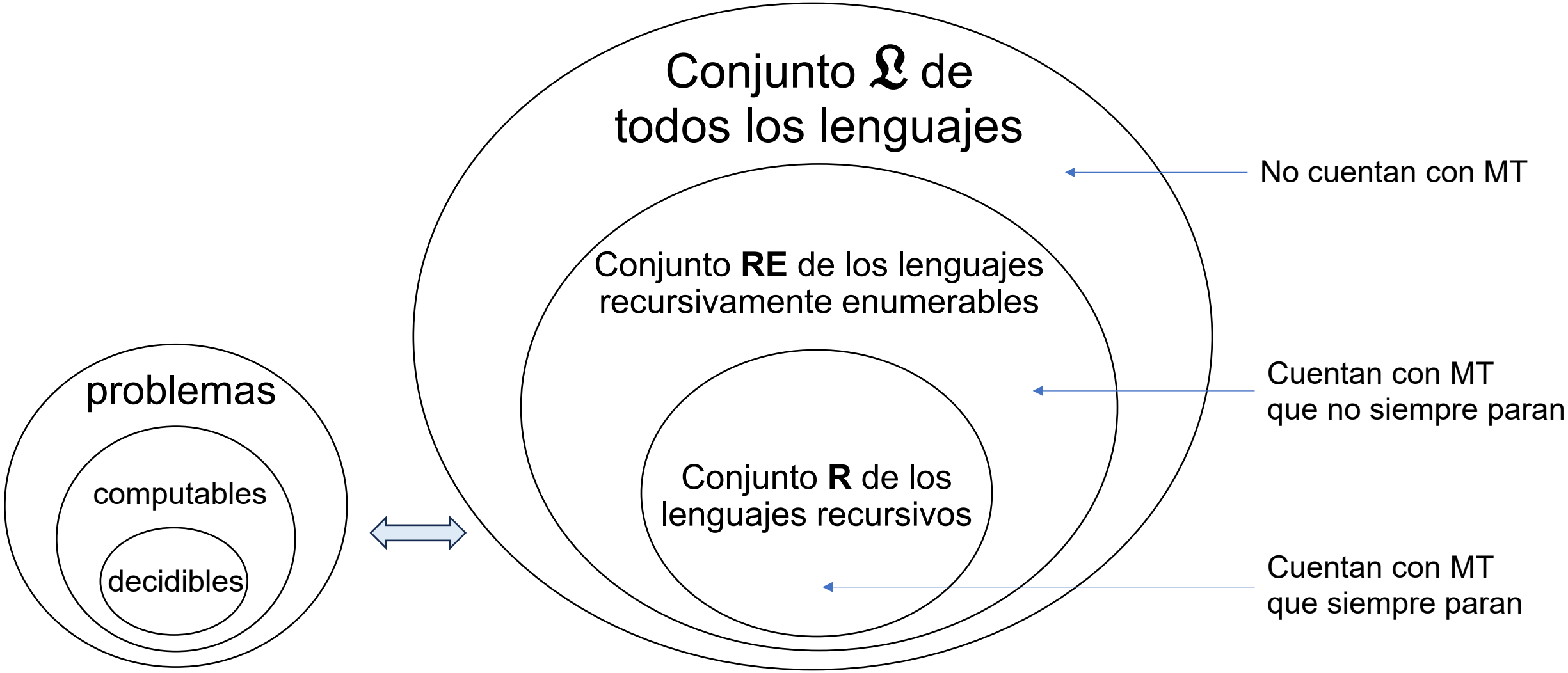
<u>Visión de MT generadora</u>

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



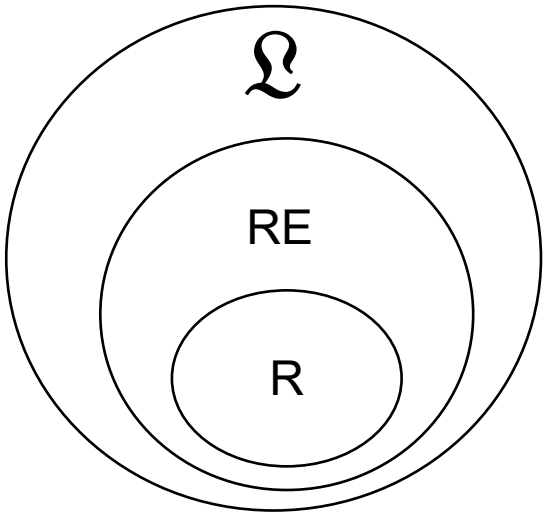
Definiciones

- 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



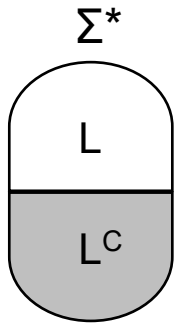
- Un lenguaje L es **recursivamente numerable** (L ∈ RE) sii

  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.

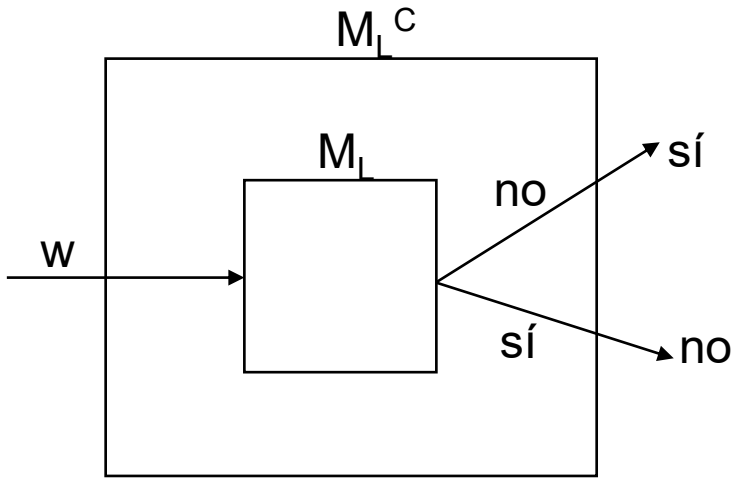
Definición: $L^C = (\Sigma^* - L)$, o en otras palabras, L^C tiene las cadenas que no tiene L.

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:



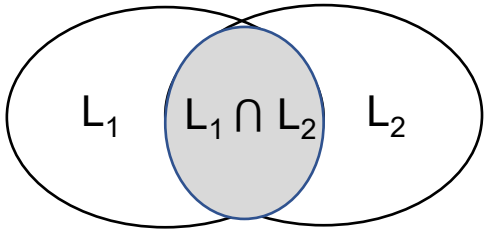
M_L acepta L y para siempre por hipótesis ($L \in R$).

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.

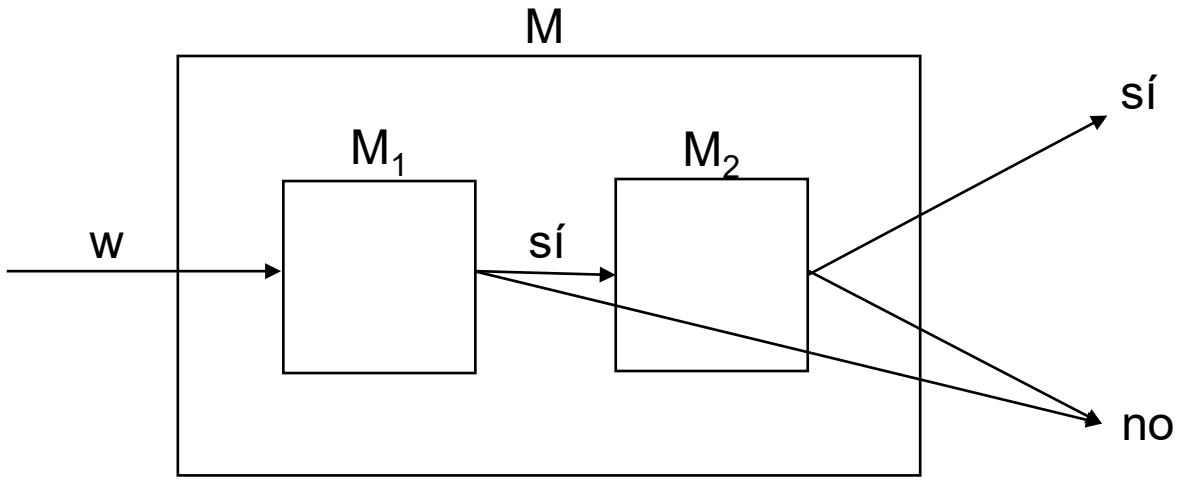


Prueba.

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.

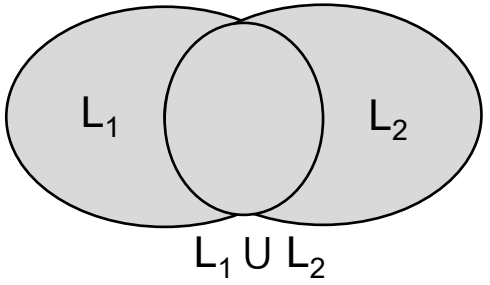
M para siempre porque M_1 y M_2 paran siempre.

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.

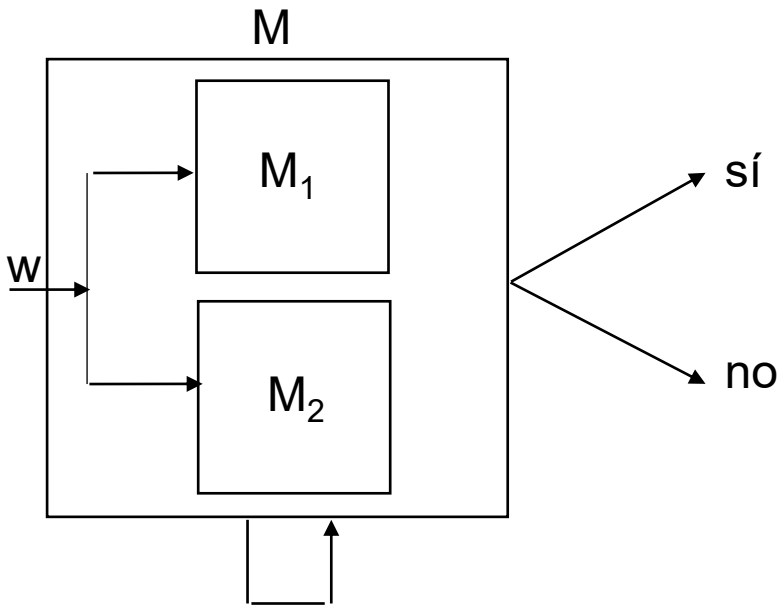


Prueba.

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 $L_1 \cup L_2$.

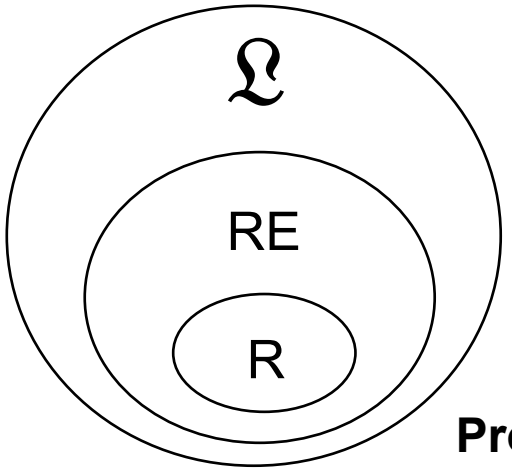
Propuesta de solución: ejecutar las dos MT en paralelo:



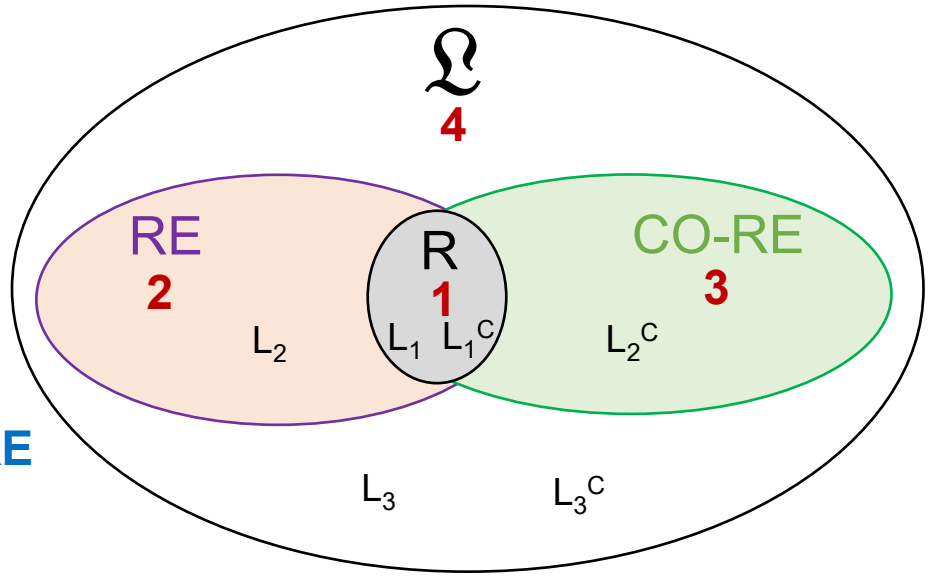
M acepta sii alguna acepta

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.

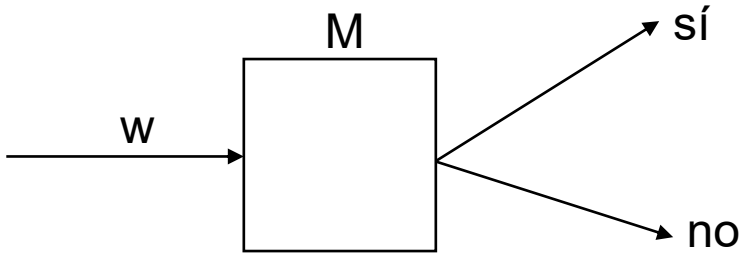
Región 2.
Clase RE – R.
Si L_2 está en RE, entonces L_2^C está en CO-RE.

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

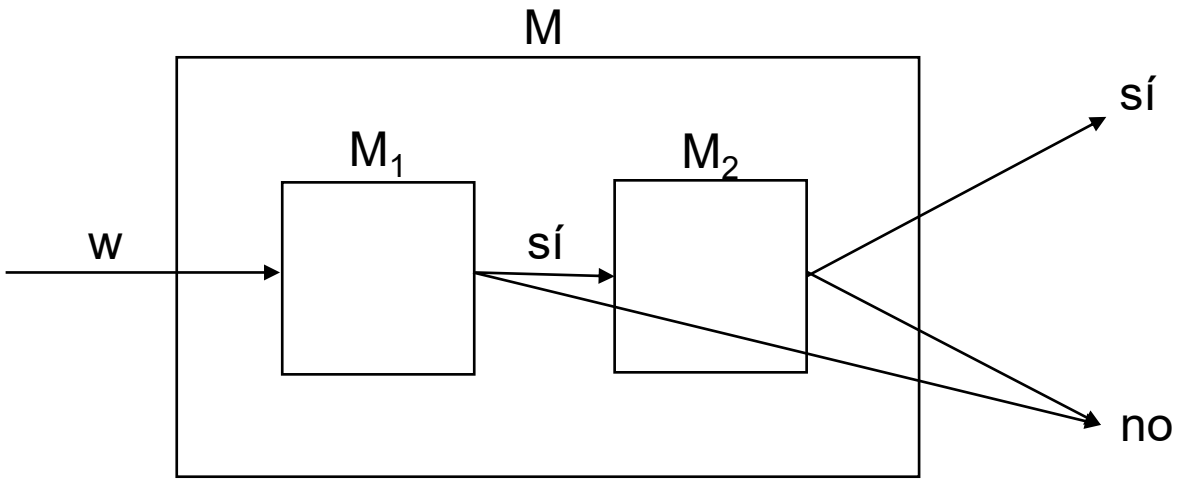
Región 4 (los lenguajes más "difíciles").
Clase Ω – (RE U CO-RE).
Si L_3 está en la clase, también está L_3^C.

Clase 3. Diagonalización

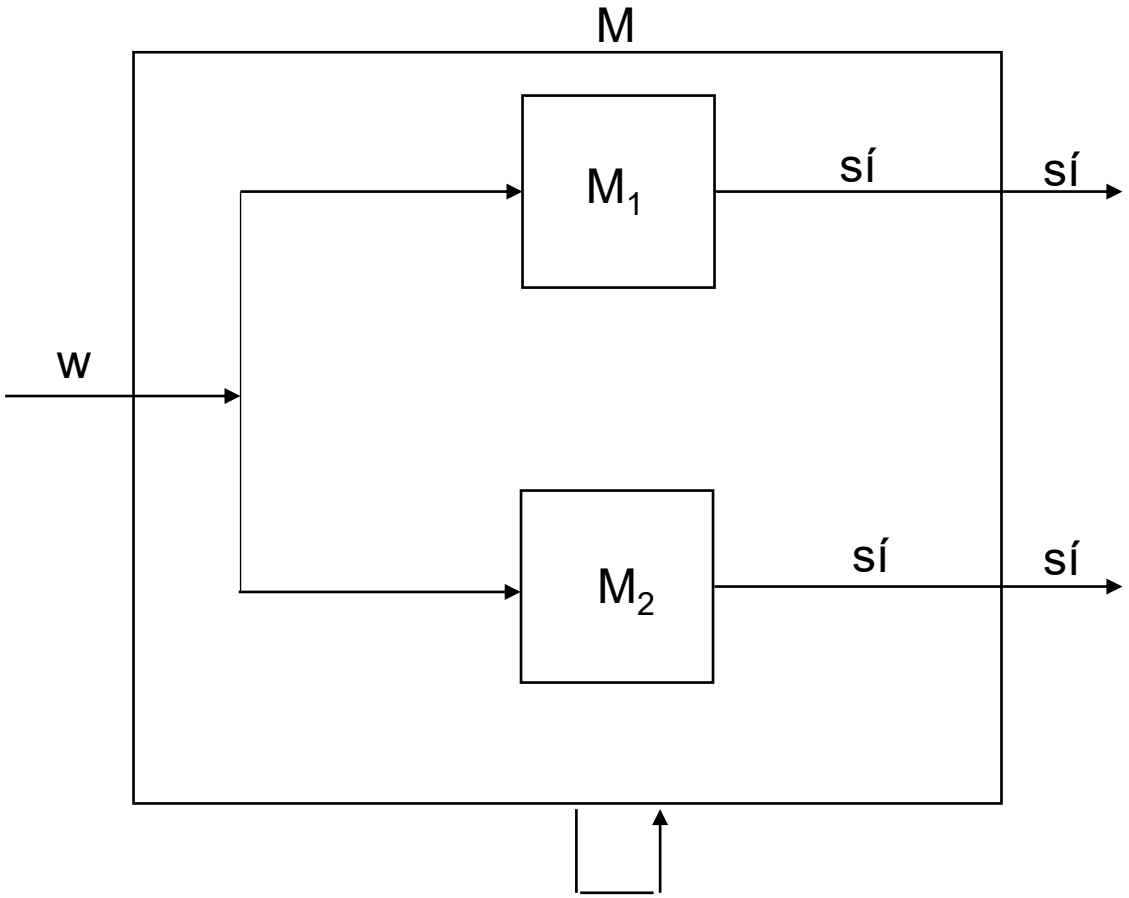
Pruebas constructivas



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

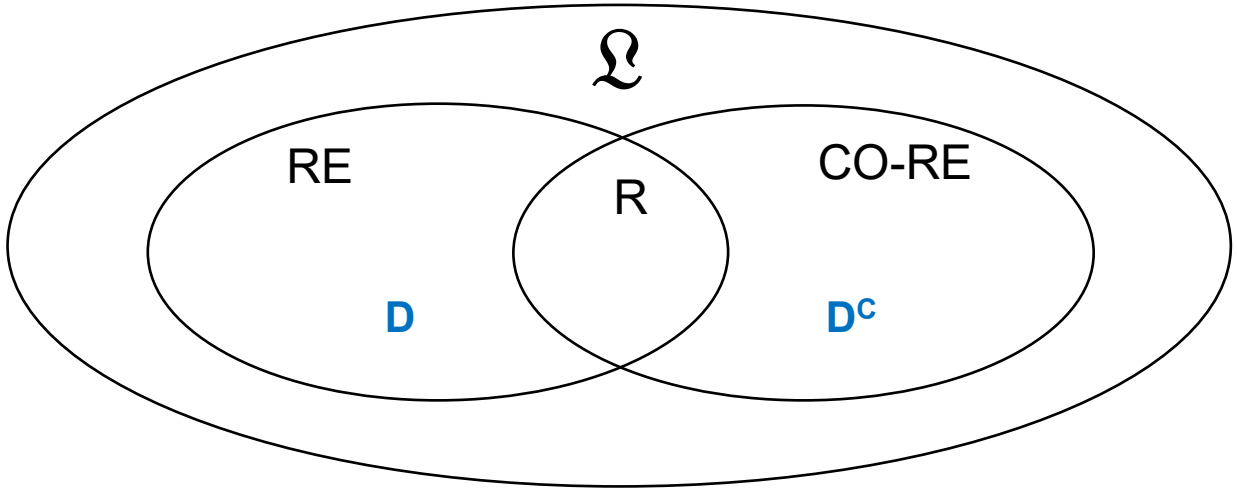


M reconoce la unión de los lenguajes que reconocen M_1 y M_2 (puede no parar)

Prueba no constructiva

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

Primer lenguaje $\mathbf{D^C}$ en CO-RE – R, para probar **RE ⊂ Ω**



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

Luego, prueba **por diagonalización** (no constructiva**)** de que $\mathbf{D^C \notin RE}$.

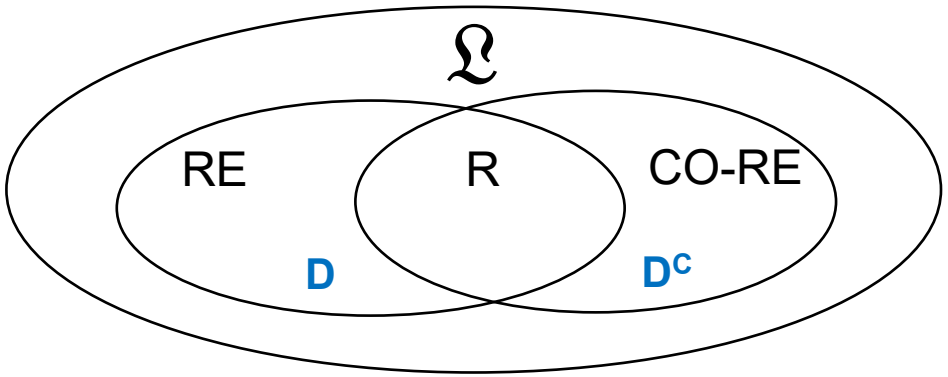
Finalmente, conclusión de que **D ∉ R** (porque si D ∈ R, entonces también $D^C \in R$, absurdo porque $D^C \notin RE$).

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**	.....
.....	.....	.....	.....	.....	.....	.....

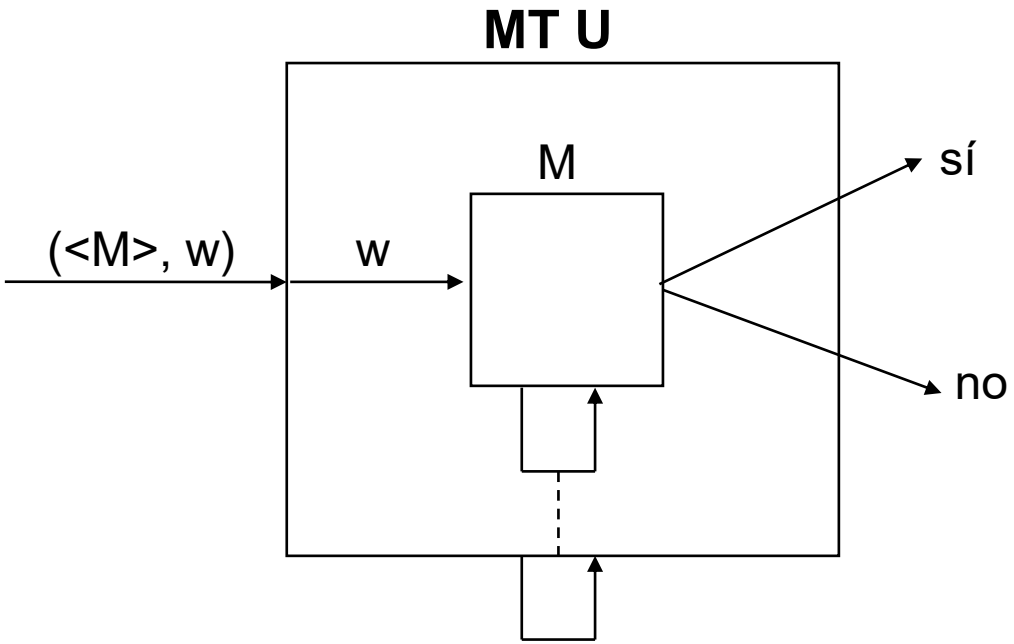
- La diagonal de T representa el lenguaje **D = {w_i | M_i acepta w_i}.**
- La diagonal de T con los 1 y 0 invertidos representa el lenguaje: **D^C = {w_i | M_i rechaza w_i}.**
- Se cumple:

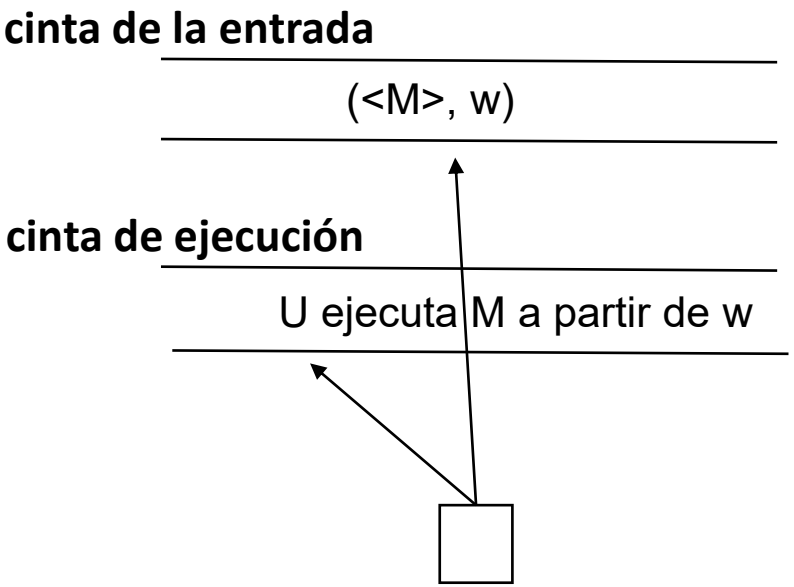
(1) **D está en RE – R**
(2) **D^C está en CO-RE – R**



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).

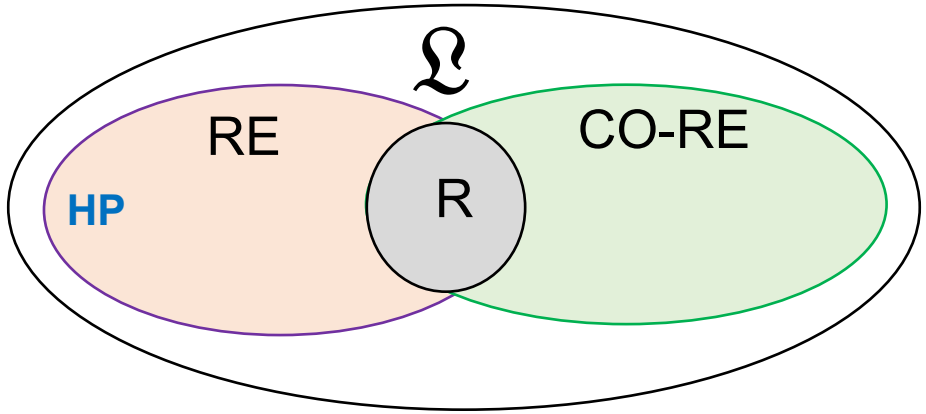
- La siguiente MT M obtiene la MT M_i según el orden canónico. Dada una entrada i, M hace:

  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).

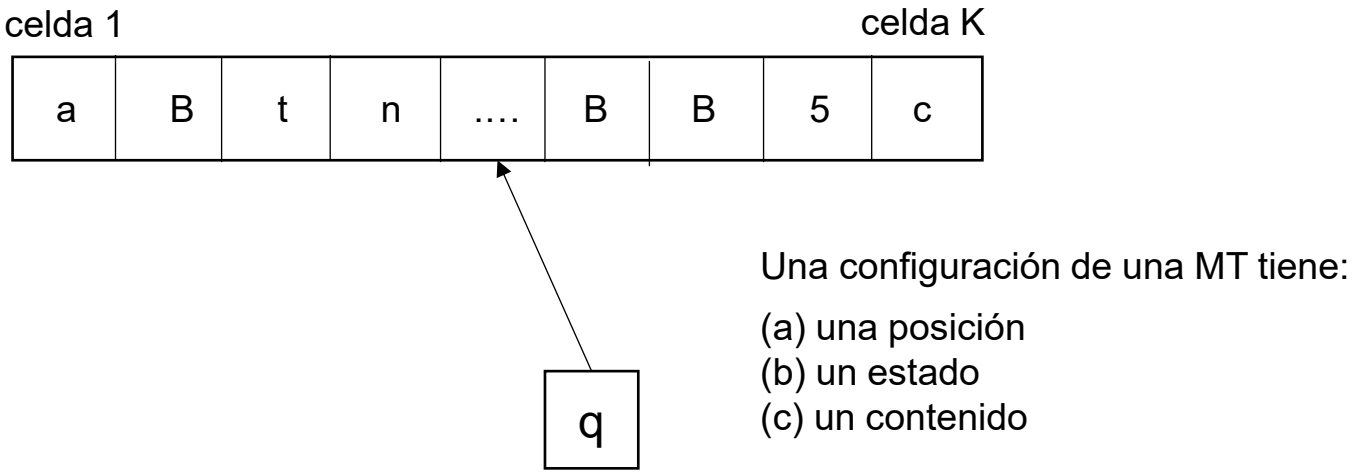


Cómo burlar al *halting problem*

- **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:

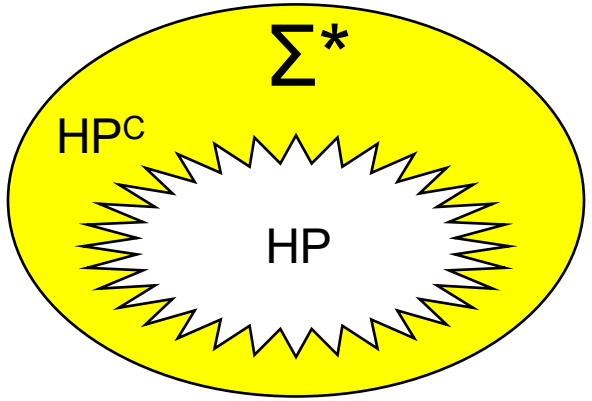
1. Hacer i := 1.
2. Ejecutar i pasos de M sobre todas las cadenas de longitud a lo sumo i.
3. Si M acepta alguna vez, aceptar.
4. Si no, hacer i := i + 1 y volver al paso 2.

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.

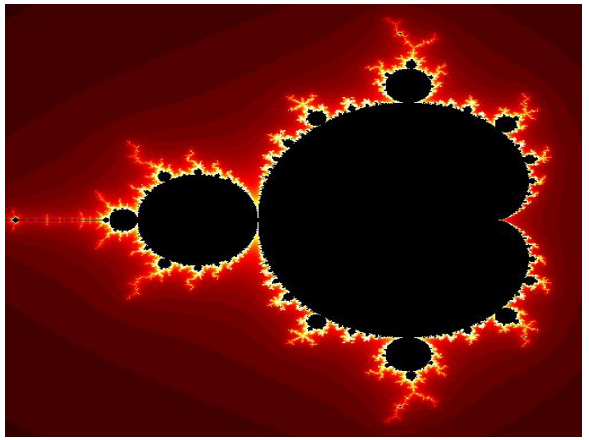
Computabilidad y tamaño de un lenguaje

- 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

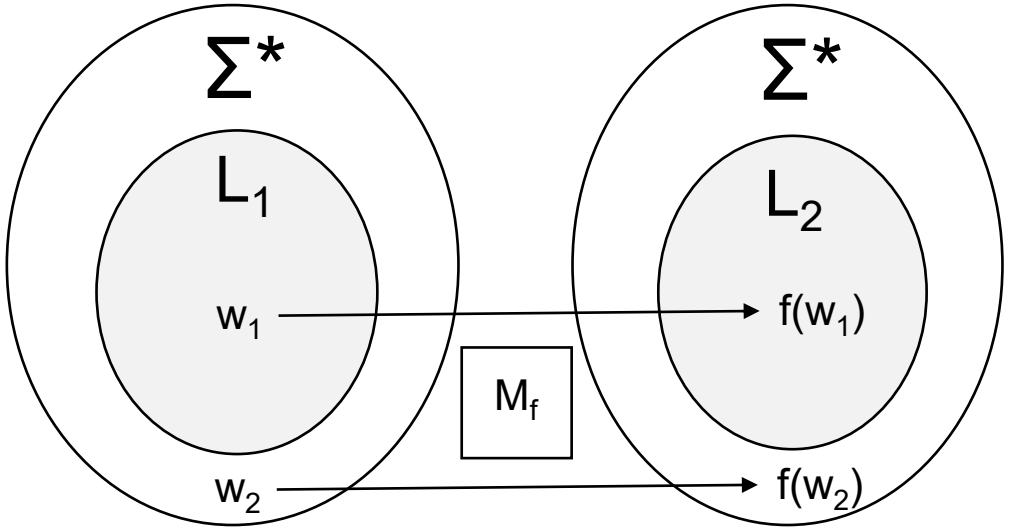


Clase 4. Reducciones

- 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 $f(w) \in L_2$

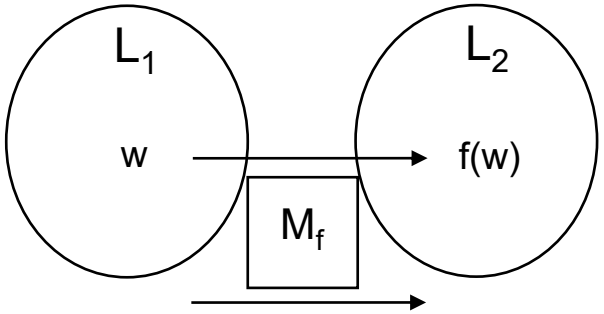
  a partir de todo $w \notin L_1$, la MT M_f genera $f(w) \notin L_2$



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

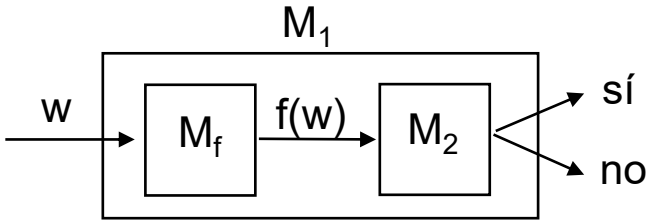


Reducción de L_1 a L_2
Para todo w, $w \in L_1$ sii $f(w) \in L_2$

TEOREMA

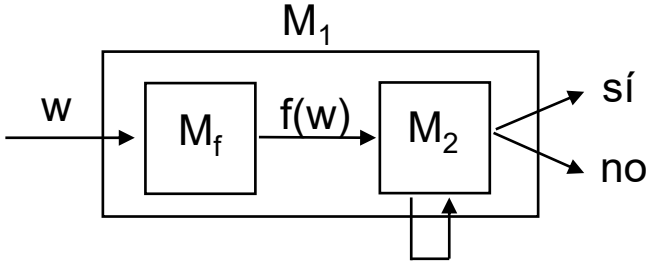
Caso 1

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 ($L_1 \notin RE \longrightarrow L_2 \notin RE$)

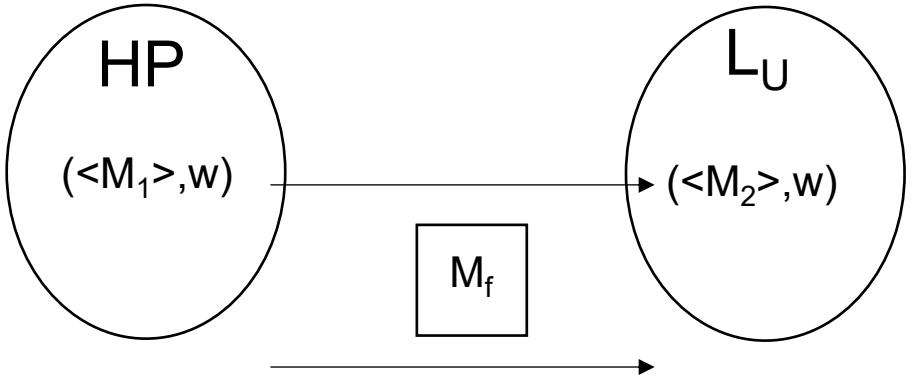


Es decir, si $L_1 \leq L_2$:
Si $L_1 \notin R$, no puede suceder que $L_2 \in R$.
Si $L_1 \notin RE$, no puede suceder que $L_2 \in RE$.
L_2 es **tan o más difícil** que L_1, r**esolviendo L_2 se resuelve L_1.**

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

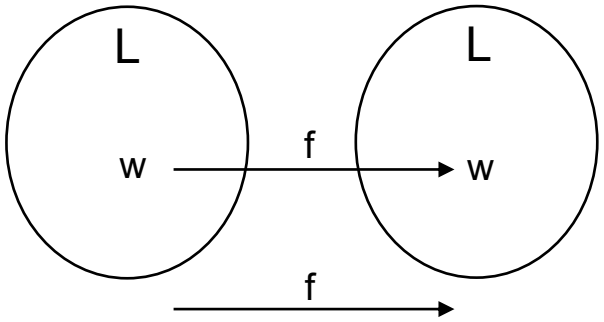
(<M_1>, w) $\in$ HP $\longrightarrow$ M_1 para sobre w $\longrightarrow$ M_2 acepta w $\longrightarrow$ (<M_2>, w) $\in$ L_U

(<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

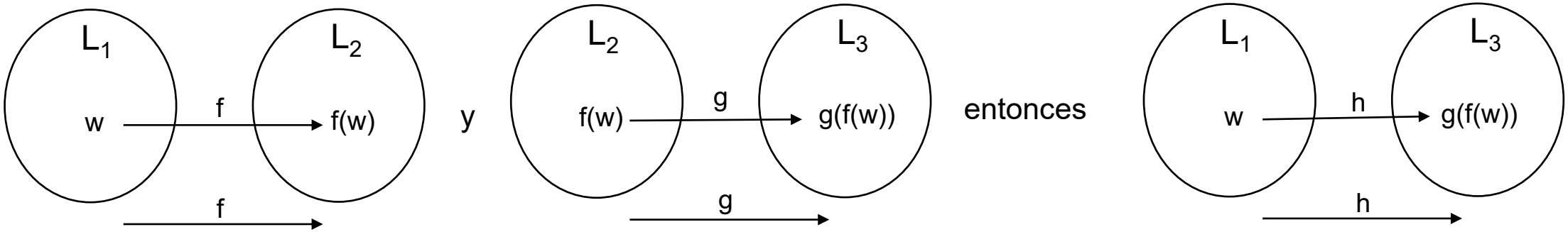
caso de cadena inválida: (<M_2>, w) también es una cadena inválida $\longrightarrow$ (<M_2>, w) $\notin$ L_U

Propiedades

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

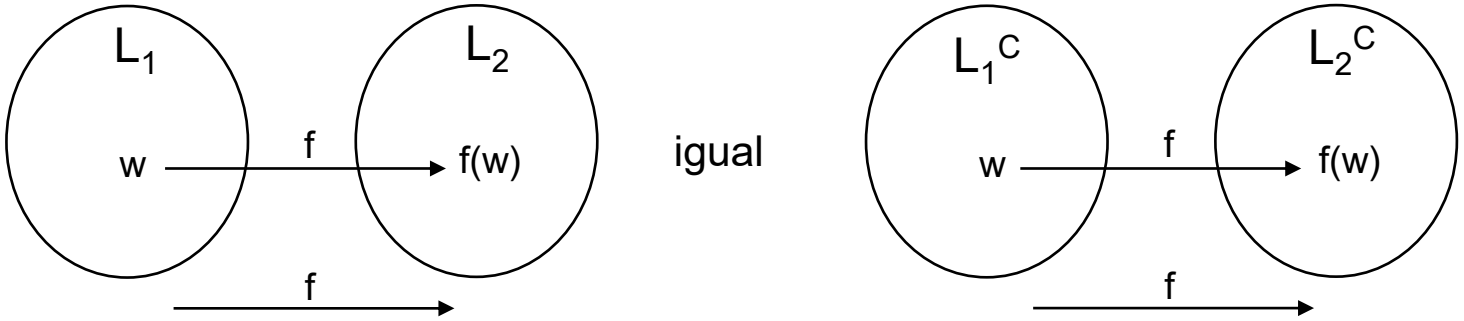


- **Transitividad.** Si $L_1 \leq L_2$ y $L_2 \leq L_3$, entonces $L_1 \leq L_3$.



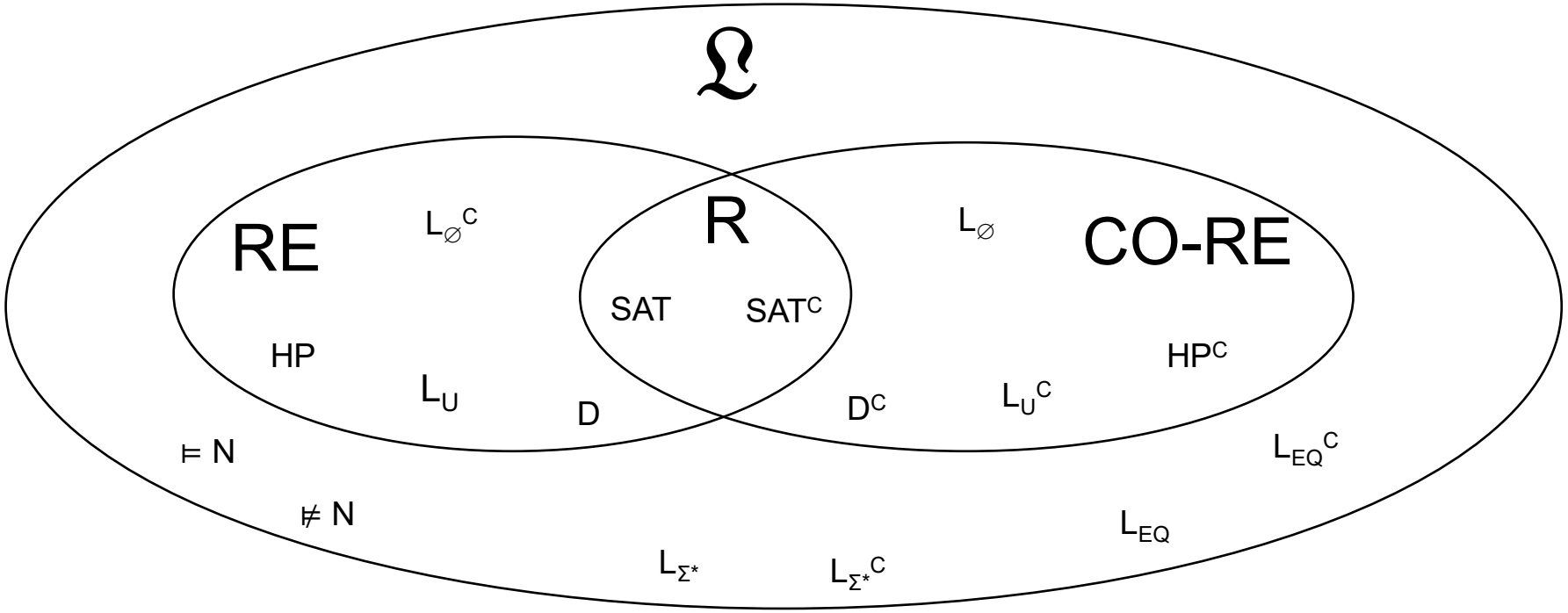
Se componen las reducciones f y g y se obtiene la reducción h.

- **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)

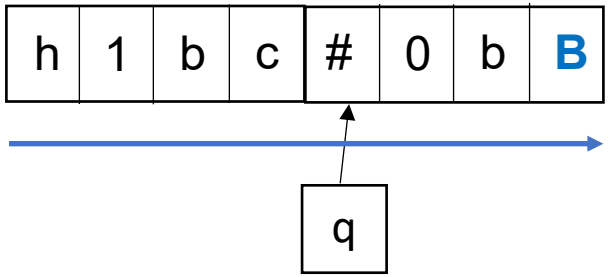
HP: *halting problem* (diagonalización)

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).

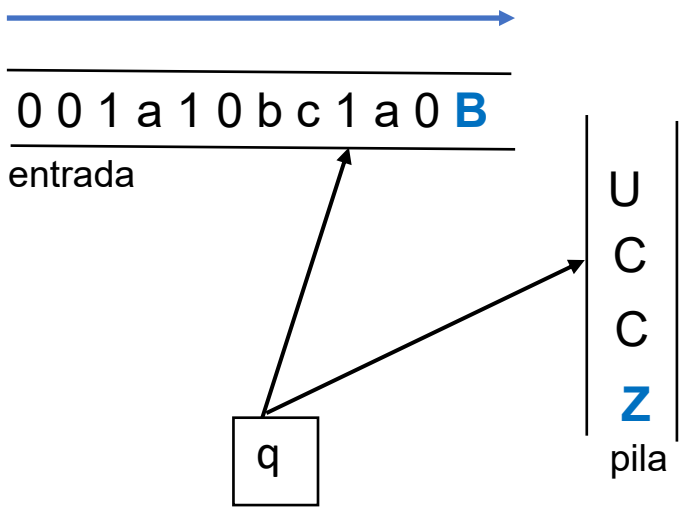


- El AF constituye un tipo de algoritmo muy utilizado. Por ejemplo:
  - Para el **análisis sintáctico a nivel palabra** de los compiladores (if, then, else, while, x, 10, =, +, etc).
  - Para las **inspecciones de código** en el control de calidad del software.

Máquinas de Turing restringidas

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).

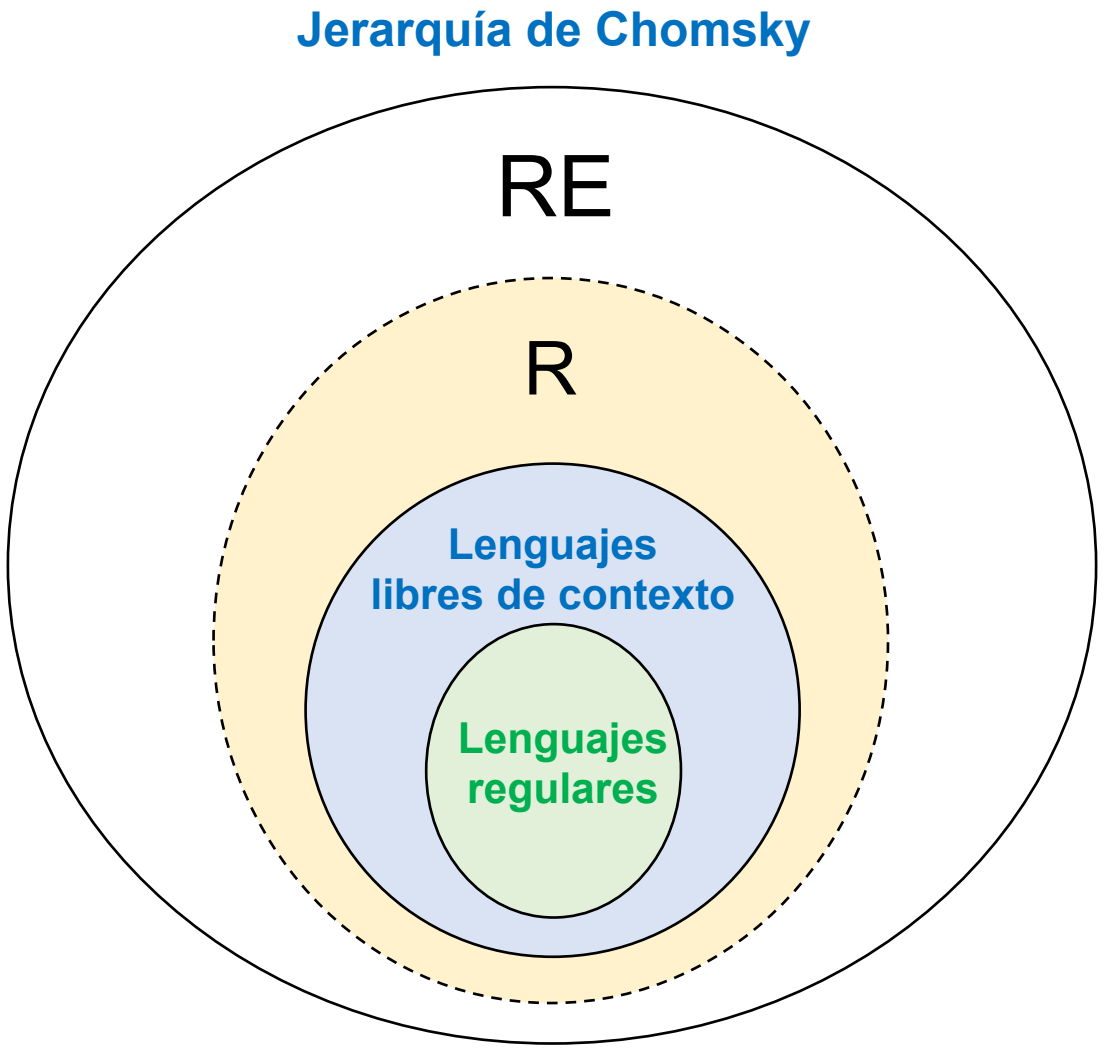


- Problemas típicos que resuelve un AP:
  - **Análisis sintáctico a nivel instrucción** de los compiladores.
  - **Evaluación de expresiones** en la ejecución de programas.

Máquinas de Turing restringidas

- Autómatas finitos (AF)
  Lenguajes regulares

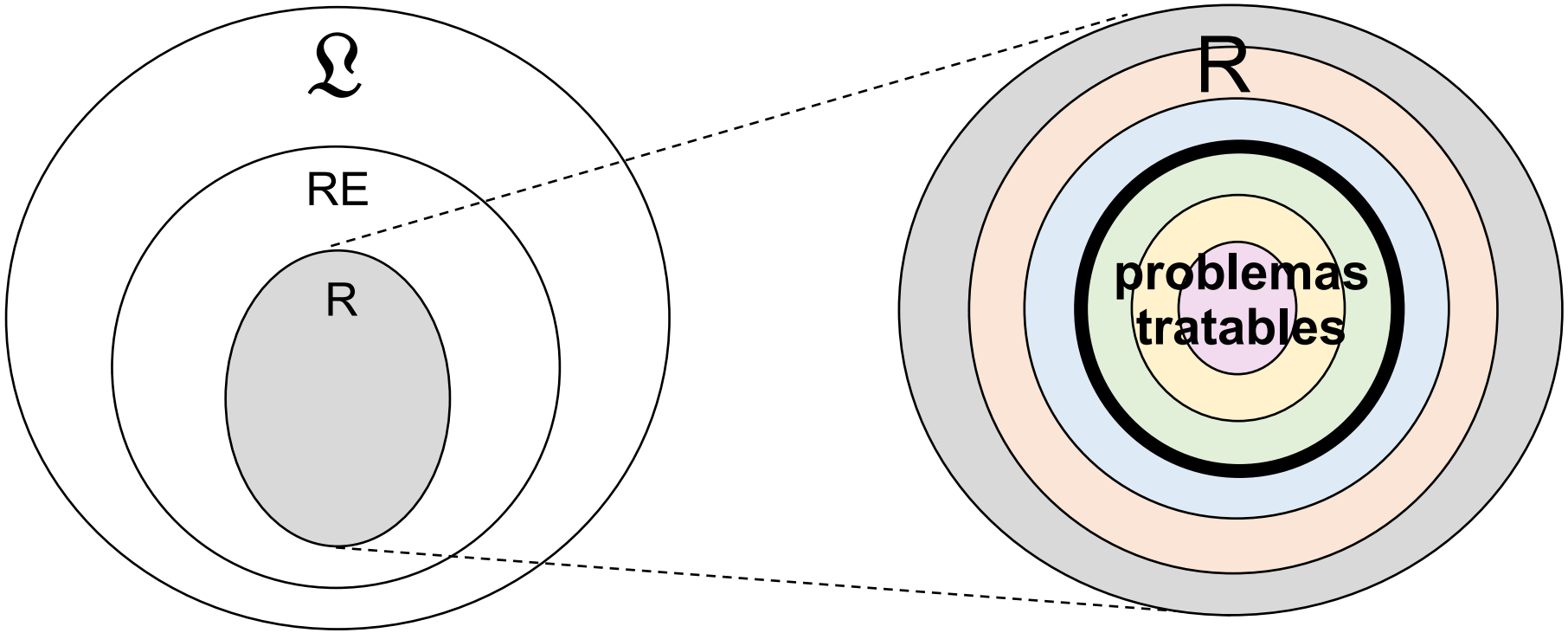
- Autómatas con pila (AP)
  Lenguajes libres de contexto



Parte 2
Complejidad Computacional

Clases 5 a 9

Clase 5. Complejidad temporal

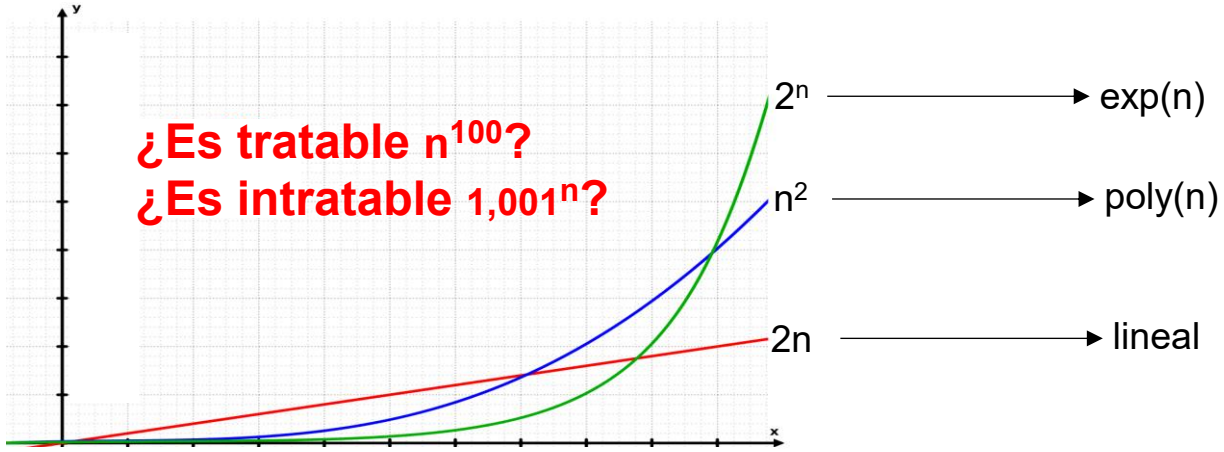


- Se repiten muchos conceptos y técnicas de la computabilidad. Contraste: muchos problemas abiertos.

Definiciones

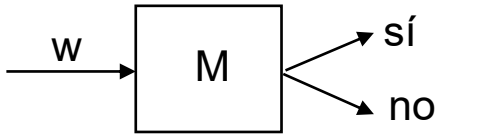
- Una MT tarda más a medida que sus cadenas de entrada w son más grandes.
- Por eso se usan **funciones temporales T(n)** definidas en términos de |w| = n.
- Funciones temporales T(n) típicas: **$5n$, $3n^3$, 2^n, $n^{\log_2 n}$, $7^{\sqrt{n}}$, $n!$, 6^{n^5}**
- Dos grandes grupos de funciones (de acuerdo 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**.

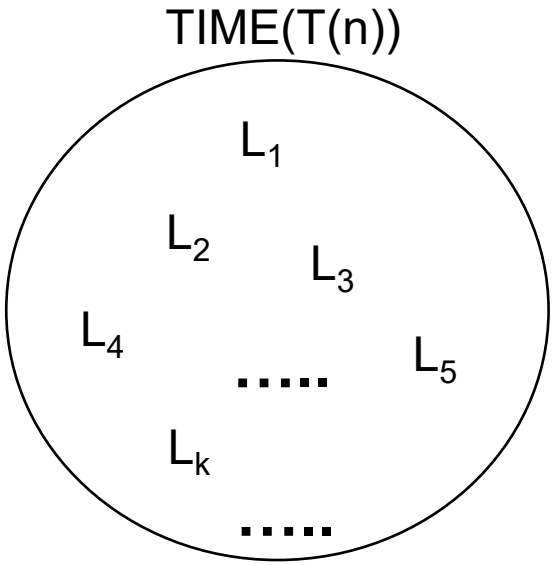


M hace a lo sumo T(|w|) pasos

- 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)).**

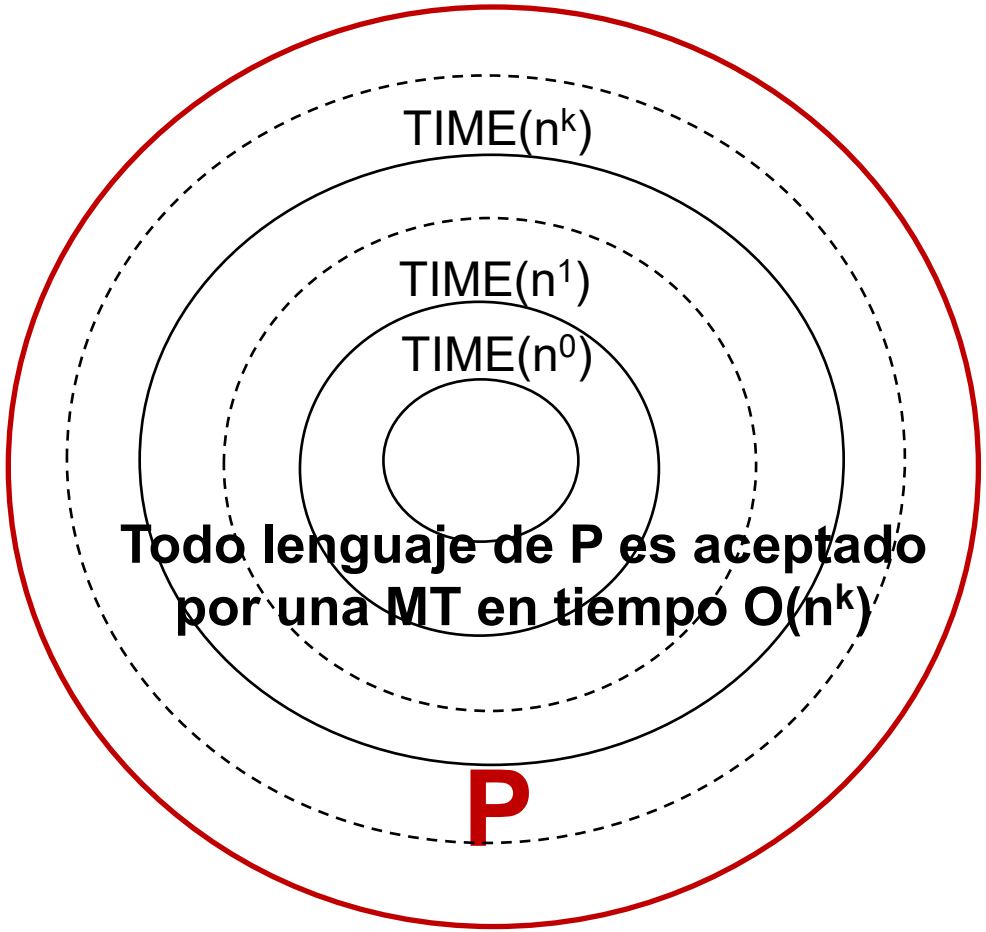


Para todo L_i existe una MT M_i que lo decide en tiempo O(T(n))

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



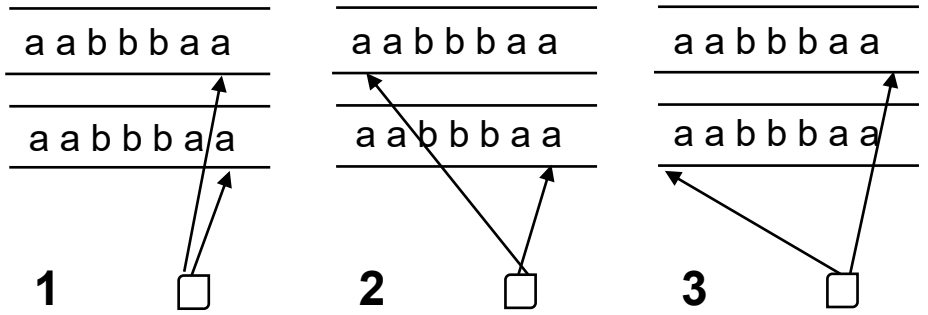
Ejemplo sencillo de lenguaje en P

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)**

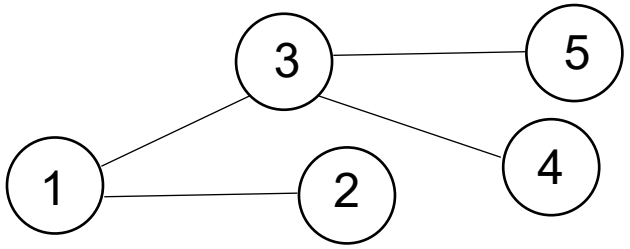


Robustez: si una MT M_1 con K_1 cintas tarda tiempo poly(n), una MT M_2 equivalente con K_2 cintas tarda tiempo poly(n). El retardo es a lo sumo **cuadrático.**

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}

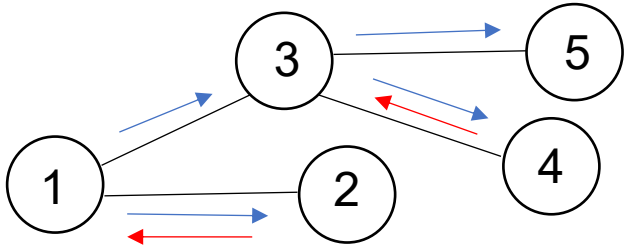
Representación de un grafo G



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)

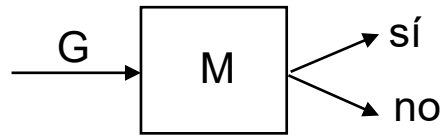


Recorrido DFS (en profundidad):

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)

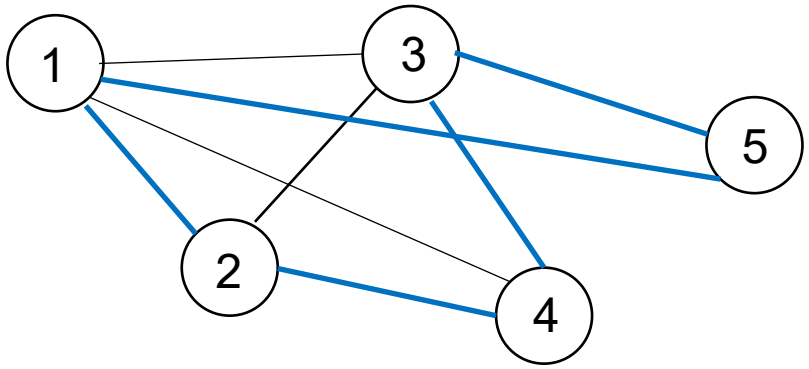
Cada evaluación se puede hacer en tiempo **$O(|\varphi|^2)$,** con el uso de una pila

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

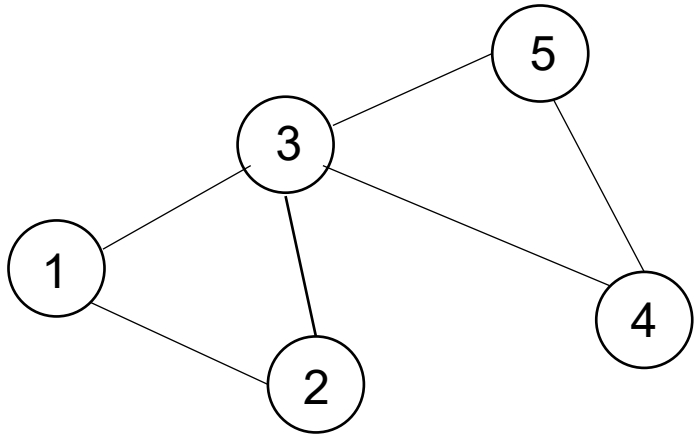
Otro ejemplo de lenguaje que no estaría en P

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)



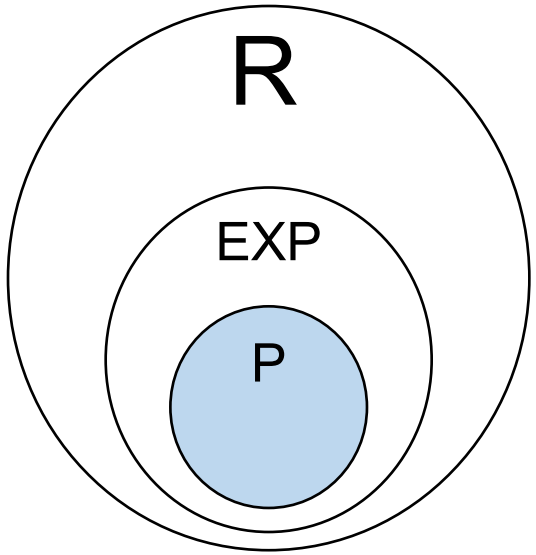
Grafo sin CdeH

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



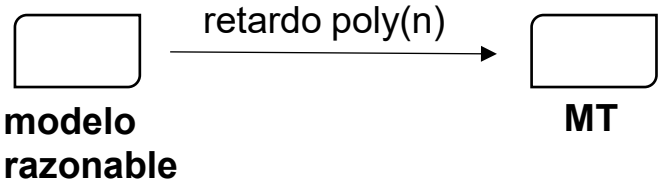
P es la clase de los lenguajes decidibles en tiempo poly(n) (**lenguajes tratables**)

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.

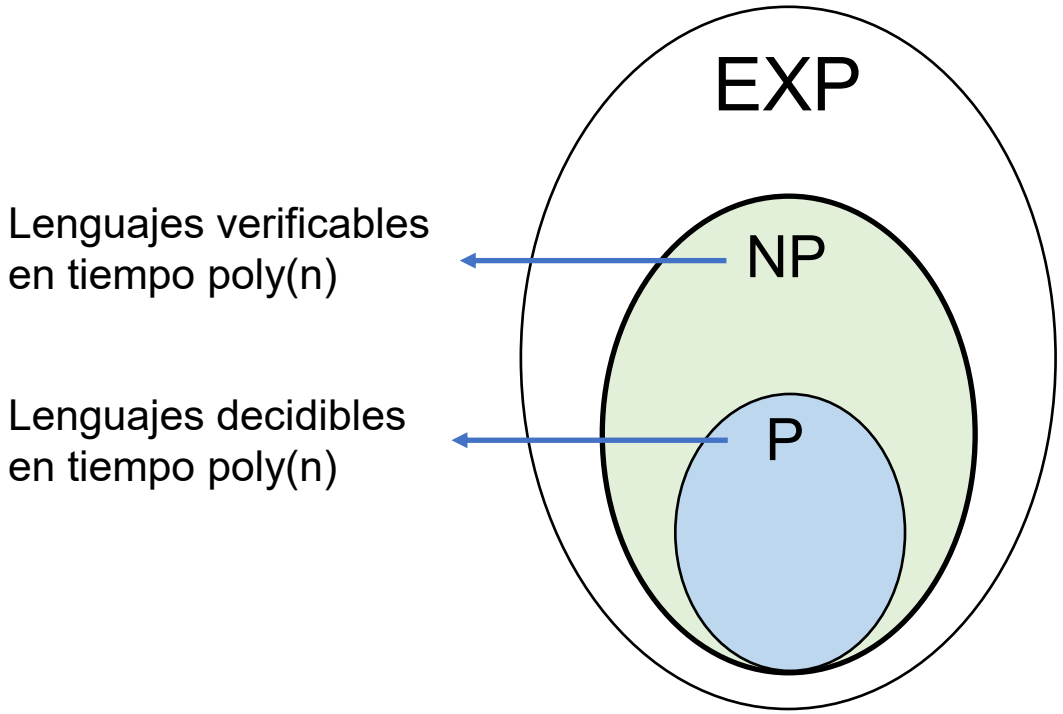
Tesis fuerte de Church-Turing (robustez de la clase P)
Si un lenguaje es decidible en tiempo poly(n) por un modelo computacional **razonable (realizable),** también es decidible en tiempo poly(n) por una MT **(¿hasta que las máquinas cuánticas sean una realidad?).**

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…).**

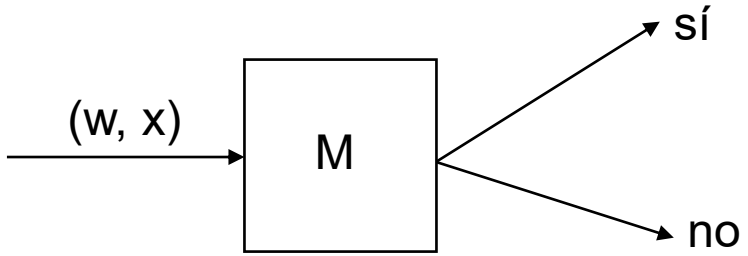
Segunda versión de la jerarquía temporal. La clase NP.



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.

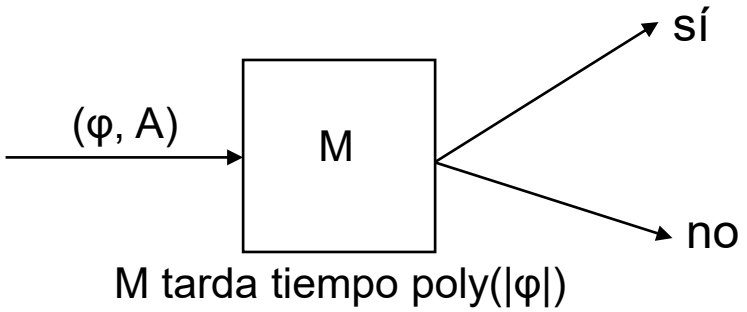
Aunque intuitivo, hasta hoy día no se ha encontrado una prueba de P ≠ NP.

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.

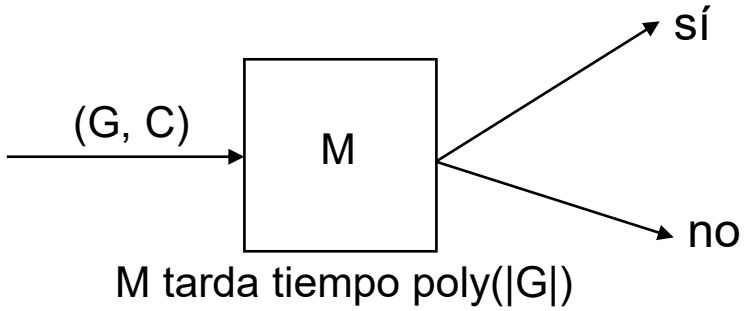


SAT^c no estaría en NP: para verificar si φ no es satisfactible hay que chequear 2^m asignaciones.

- **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.

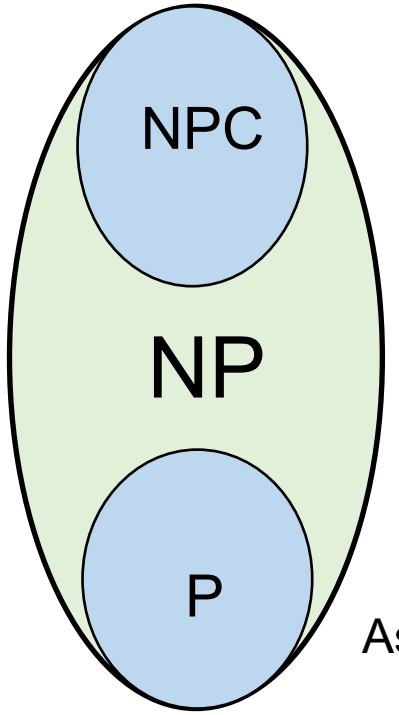


CH^c no estaría en NP: para verificar si G no tiene un CH hay que chequear m! secuencias.

CO-NP sería distinto a NP

Clase 6. Lenguajes NP-completos

- Una visión más detallada de NP:

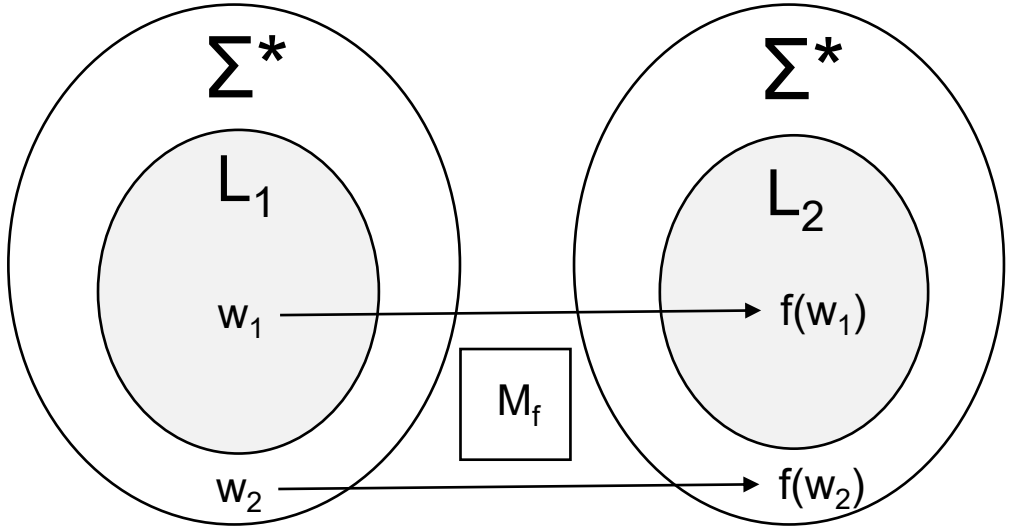


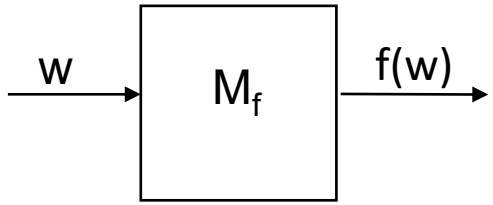
Asumiendo P ≠ NP

- 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):





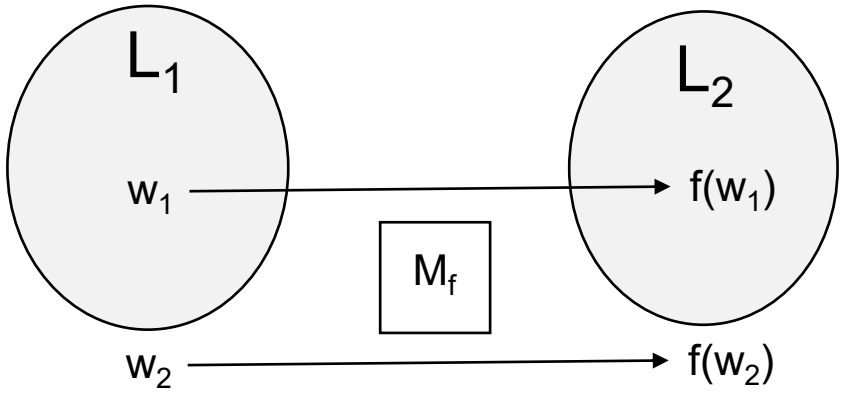
tiempo poly(|w|)

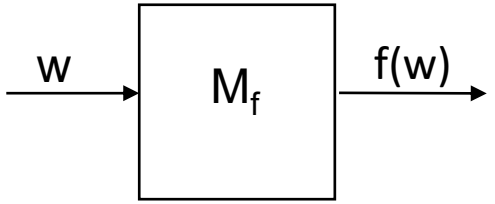
- La expresión $\mathbf{L_1 \leq_p L_2}$ establece que existe una reducción polinomial de L_1 a L_2.
- Se cumple, como en el caso general, que las reducciones polinomiales son **reflexivas, transitivas** y **no simétricas**.
- También como en el caso general, las reducciones polinomiales permiten **relacionar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

Reducciones polinomiales (continuación)

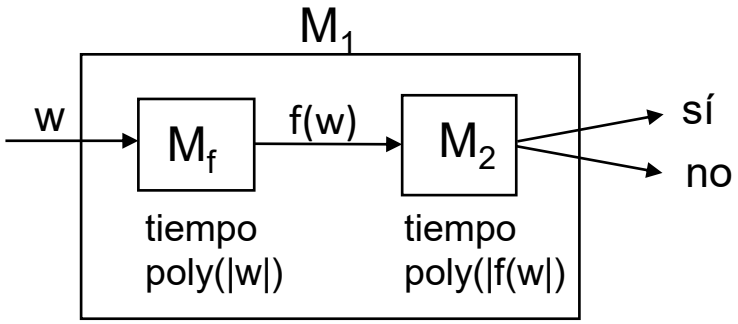
Teorema

(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 $L_1 \notin NP$: $L_2 \notin NP$

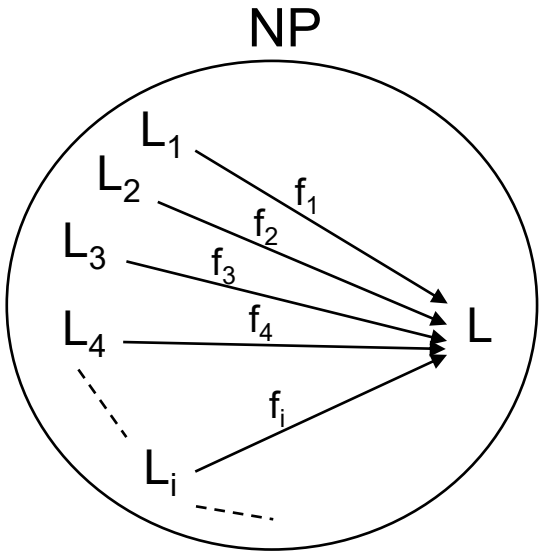
Si $L_1 \leq_p L_2$, L_2 es tan o más difícil que L_1

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**)



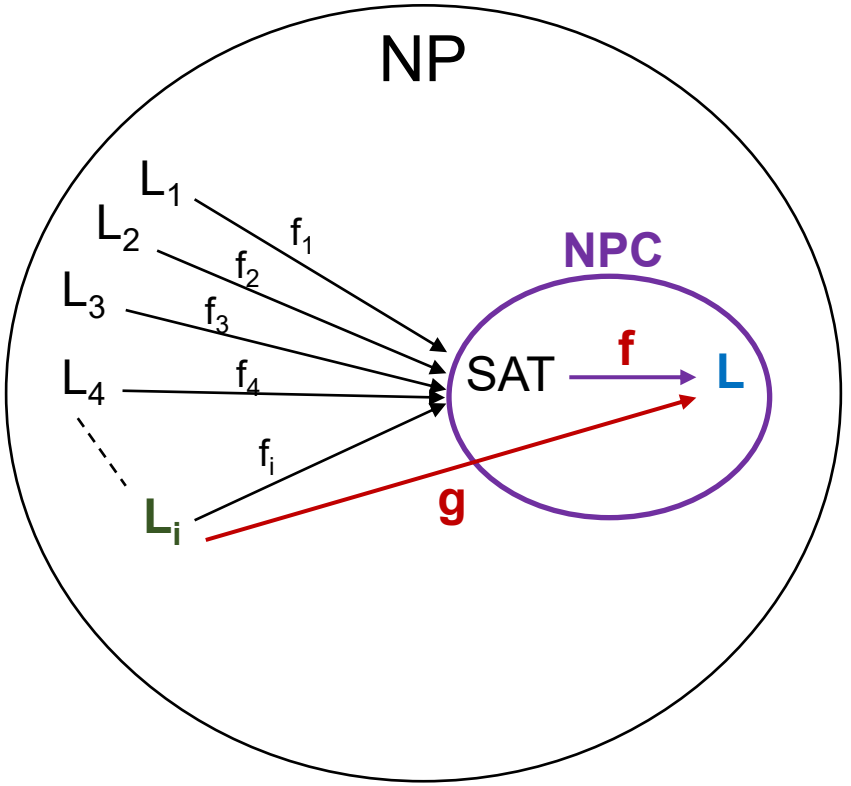
Todos los lenguajes de NP
se reducen polinomialmente a L

- 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**}

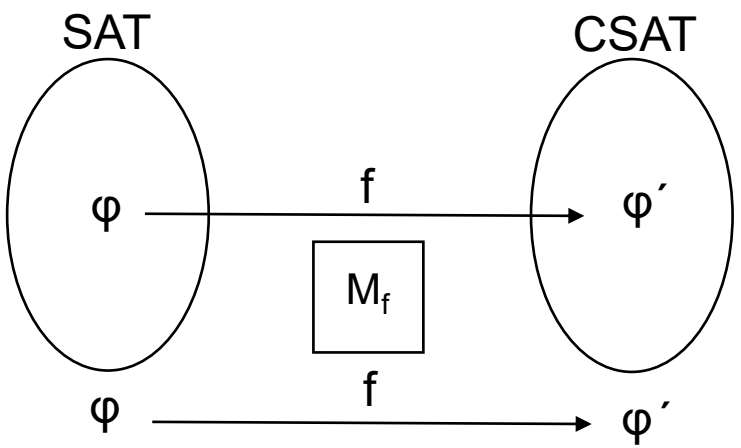
$$x_1 \vee (x_2 \wedge \neg x_3 \vee x_1) \wedge (x_3 \vee \neg x_1 \wedge x_3) \vee (x_5 \vee \neg x_1) \wedge \neg x_4$$

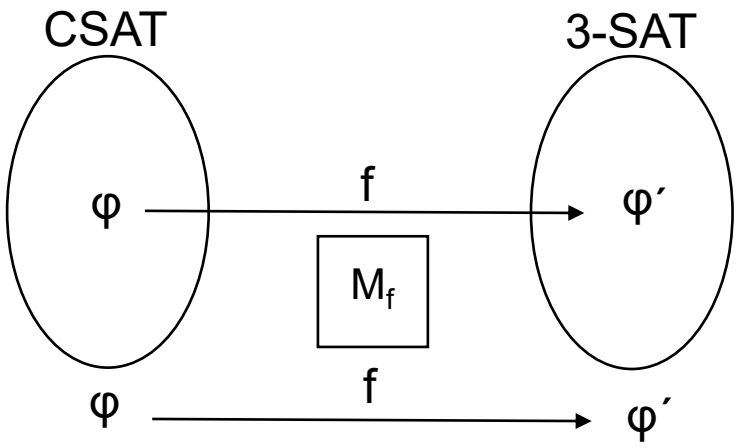
CSAT = {φ | φ es una fórmula booleana sin cuantificadores en la **forma normal conjuntiva (FNC), satisfactible**}

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

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

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



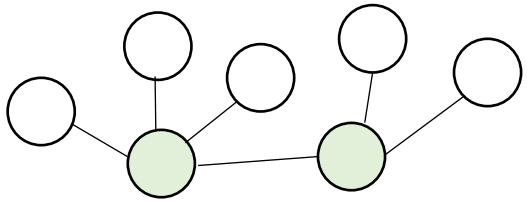


Ejemplos clásicos (continuación)

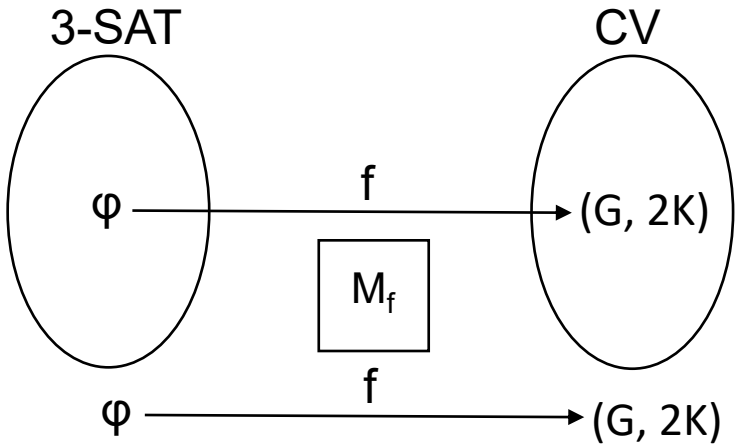
3-SAT = {φ | φ es una fórmula booleana sin cuantificadores en FNC **con tres literales por cláusula, satisfactible**}

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

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

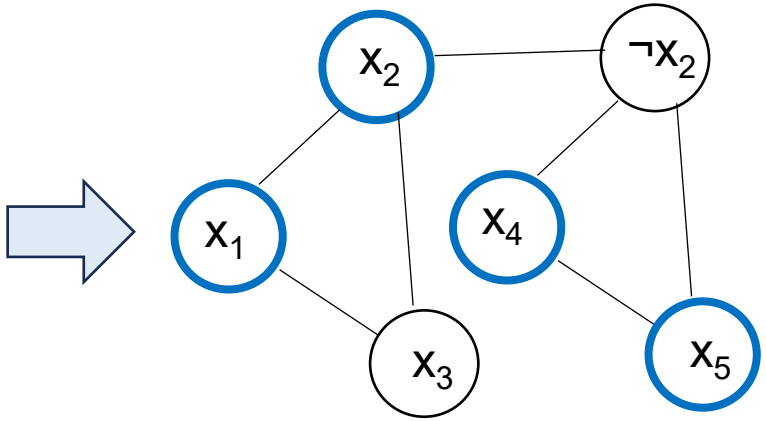


Ejemplo de cubrimiento de vértices de tamaño 2 (con 2 vértices toca todos los arcos)



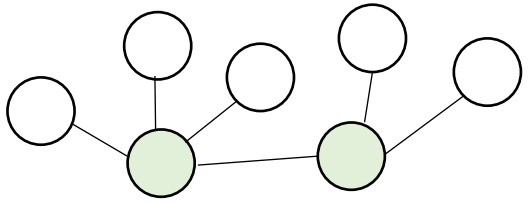
K = nro de clásulas

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

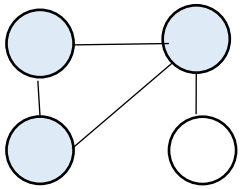


Ejemplos clásicos (continuación)

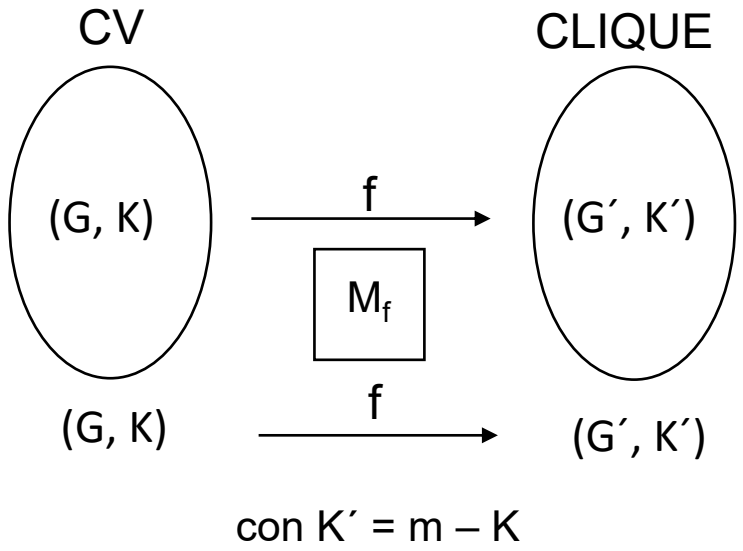
CV = {(G, K) | G es un grafo y tiene un **cubrimiento de vértices de tamaño K**}

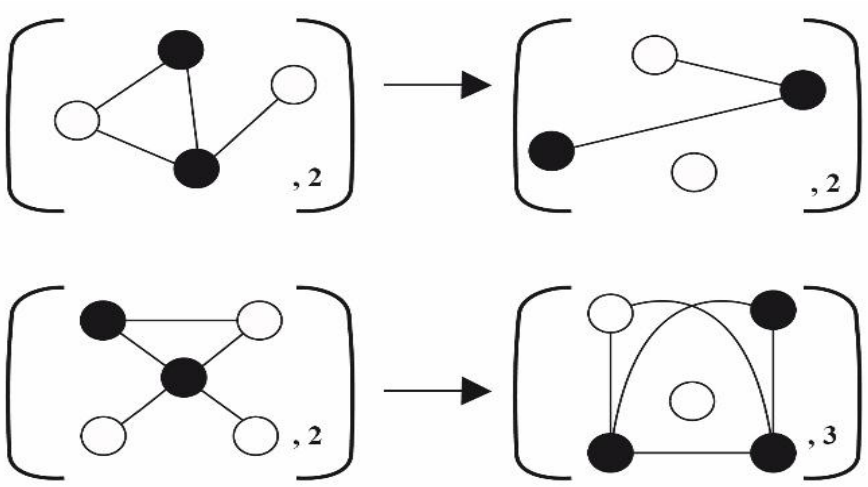


CLIQUE = {(G, K) | G es un grafo y tiene un **clique de tamaño K**}



Ejemplo de clique de tamaño 3



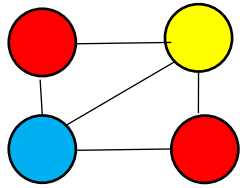


Ejemplos clásicos (continuación)

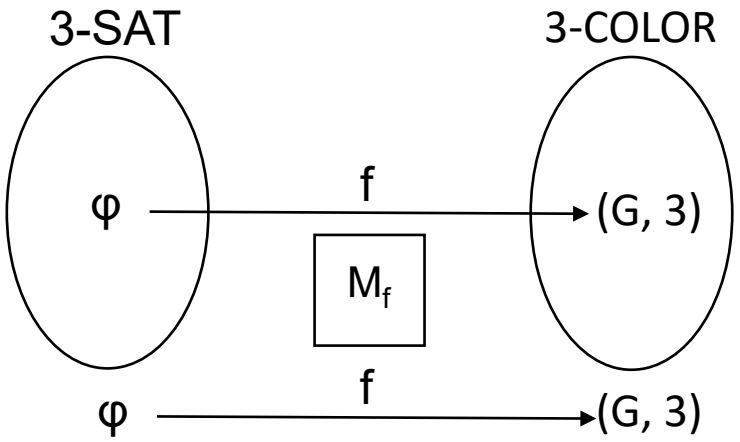
3-SAT = {φ | φ es una fórmula booleana sin cuantificadores en FNC **con tres literales por cláusula satisfactible**}

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

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

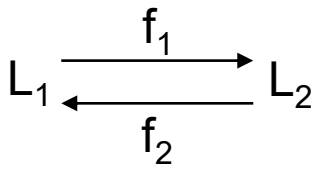


Teorema de los Cuatro Colores

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

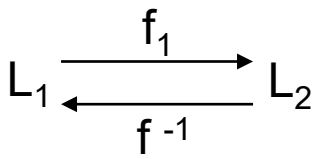
Dos características de los problemas NP-completos

- 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.**

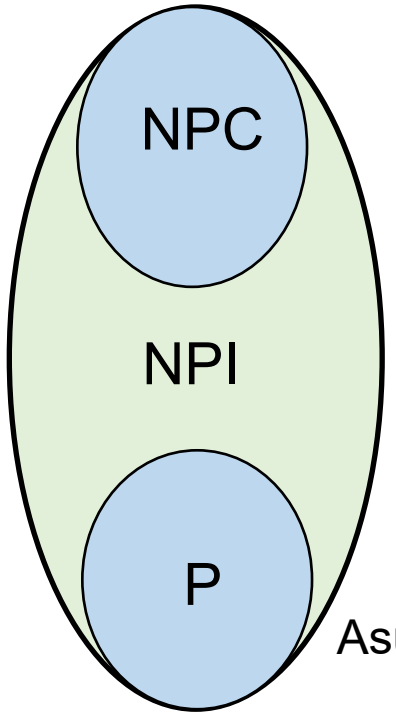
- **Todos los lenguajes NP-completos conocidos son densos**.

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:



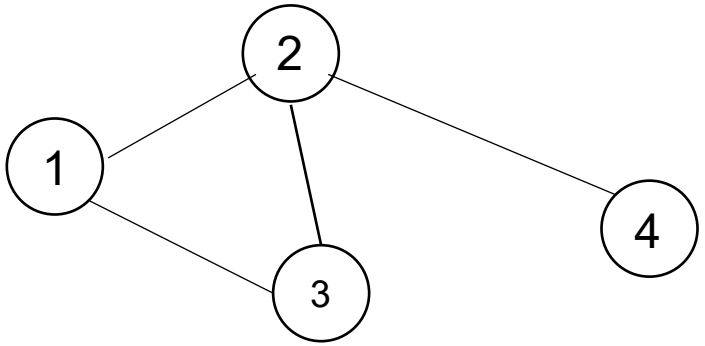
NP

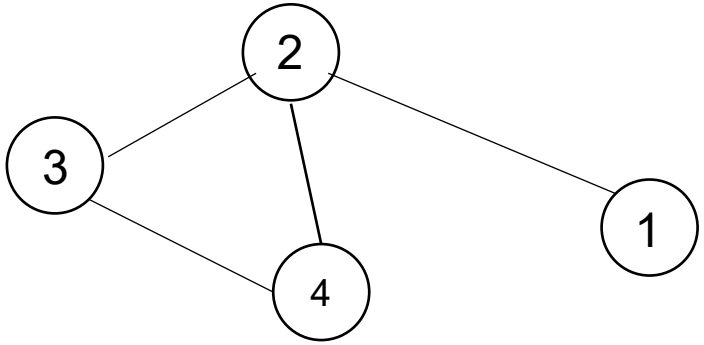
Asumiendo P ≠ NP

- 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}



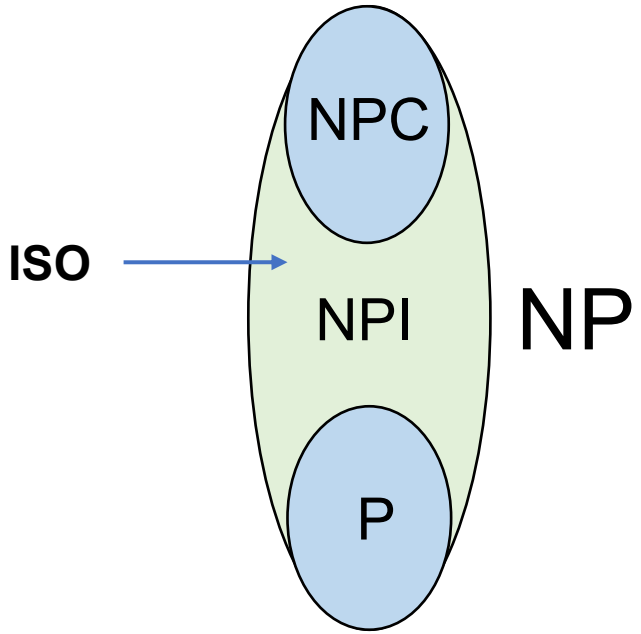


ISO está en NP
los certificados suscintos son permutaciones de V

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

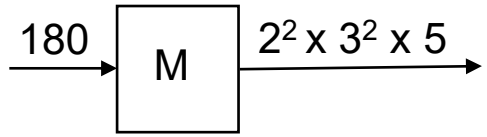
ISO^C no estaría en NP
no se han encontrado sucintos para ISO^C



Asumiendo P ≠ NP

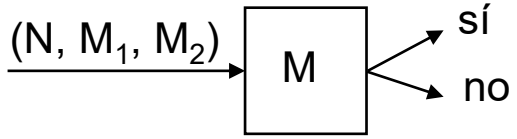
El problema de la factorización (otro candidato a estar en NPI)

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.

mensaje encriptado (sistema RSA)

10011110101000011101011100011111100101010

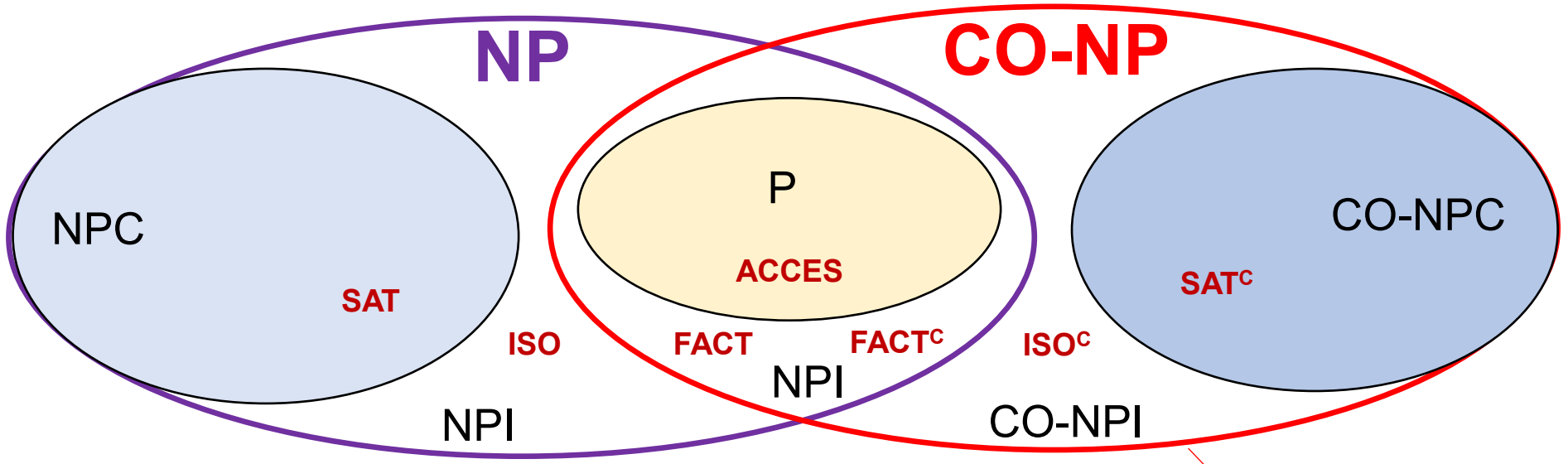
se encripta con N — clave pública
se desencripta con N_1 y N_2 — clave privada

- 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)

- 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

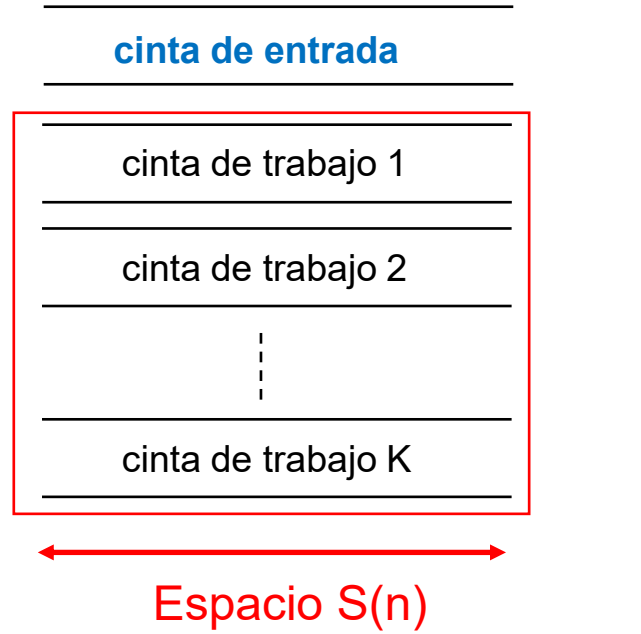
Jerarquía de lenguajes (de menor a mayor dificultad)

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).**

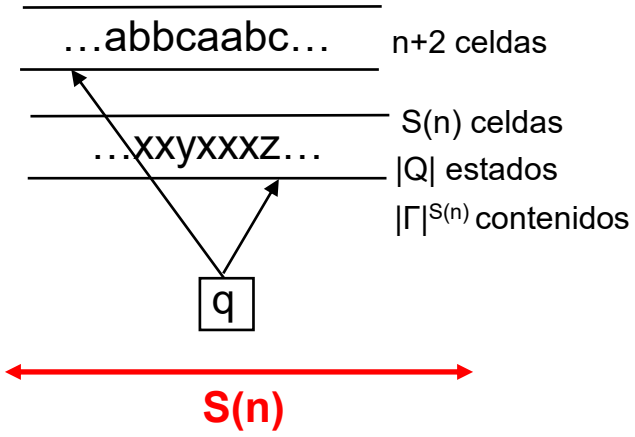


- Una MT M que ocupa espacio S(n) **puede no parar**, pero dada M, existe una MT M´ equivalente que ocupa espacio S(n) y **para siempre.**

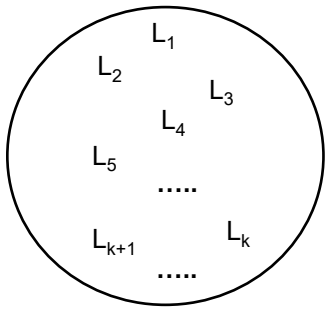
  P.ej., una MT con 1 cinta de entrada de sólo lectura y 1 cinta de trabajo entra en loop si hace más de **$n.S(n).|Q|.|\Gamma|^{S(n)} \leq c^{S(n)}$) pasos (c depende de M).** Tener en cuenta entonces que:

  tiempo T(n) implica espacio S(n)

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



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



SPACE(S(n))

Los Li se deciden 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 $c^{S(n)}$, con c constante**

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

  Y como **$c^{\log_2 n} = n^{\log_2 c}$** = 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 $c^{S(n)}$, para alguna constante c.

Ejemplo. El lenguaje QSAT.

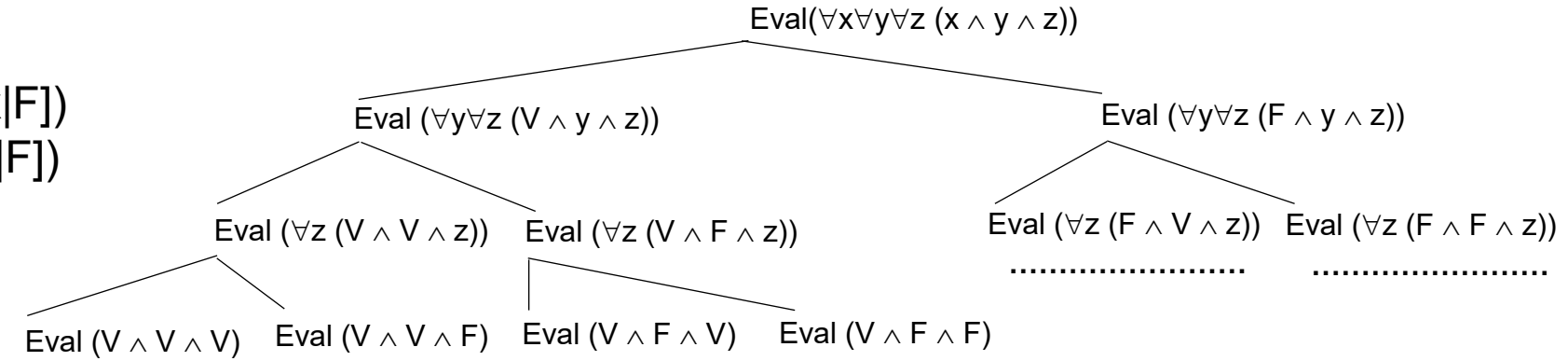
QSAT = {φ | φ es una fórmula booleana con cuantificadores, no tiene variables libres, y es verdadera}.

- 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):

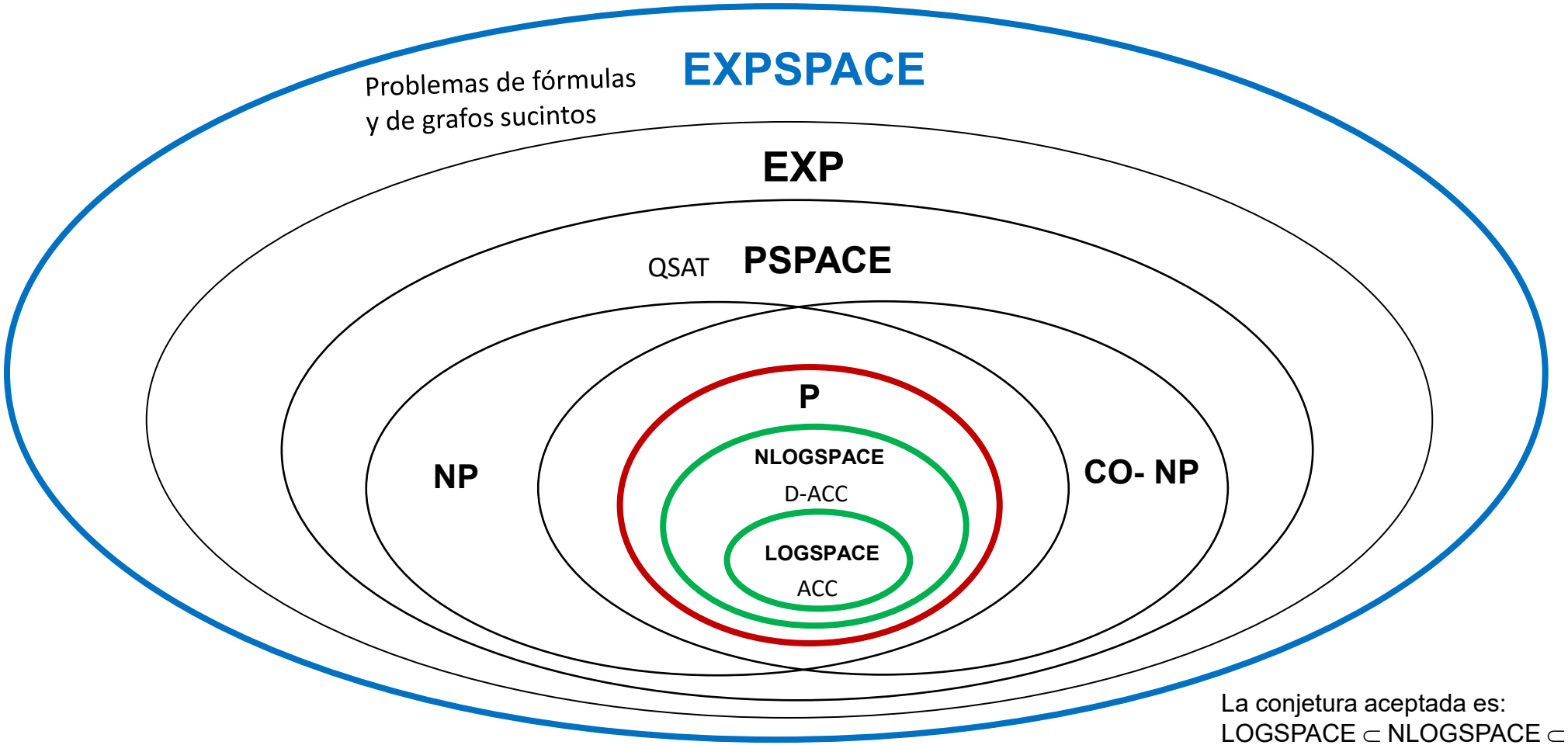
$Eval(\varphi,\forall x) = Eval(\varphi[x|V]) \wedge Eval(\varphi[x|F])$
$Eval(\varphi,\exists x) = Eval(\varphi[x|V]) \vee Eval(\varphi[x|F])$
$Eval(\varphi_1,\varphi_2,\wedge) = Eval(\varphi_1) \wedge Eval(\varphi_2)$
$Eval(\varphi_1,\varphi_2,\vee) = Eval(\varphi_1) \vee Eval(\varphi_2)$
$Eval(\varphi,\neg)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



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



La conjetura aceptada es:
LOGSPACE ⊂ NLOGSPACE ⊂ P ⊂ PSPACE

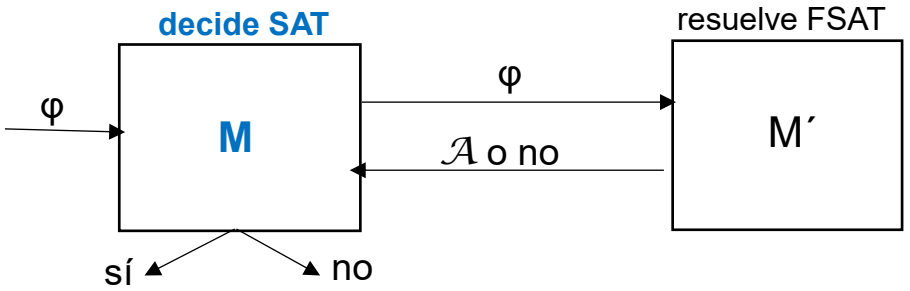
- D-ACC es **NLOGSPACE-completo** con respecto a las reducciones logarítmicas.
- QSAT es **PSPACE-completo** con respecto a las reducciones polinomiales.
- QSAT es una instancia de un problema más general: la búsqueda de una **estrategia ganadora en una competencia entre dos jugadores J_1 y J_2** (ajedrez, damas, go, hexágono, geografía, etc).

Clases 7 (parte 3), 8 y 9. Misceláneas de complejidad computacional

Problemas de búsqueda

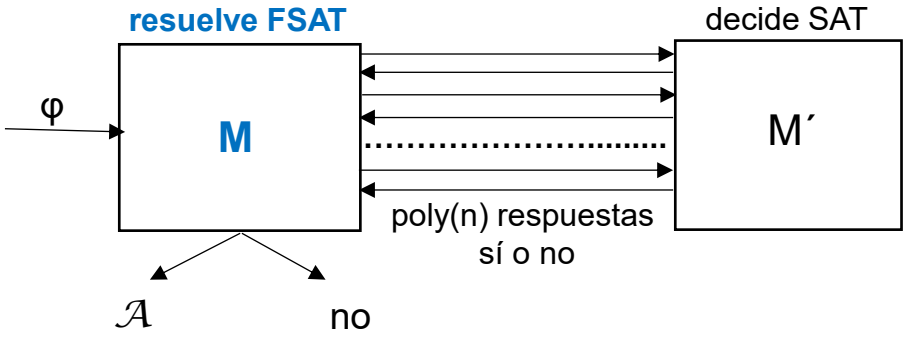
- Las clases P y NP correspondientes a los **problemas de búsqueda** se denominan **FP** y **FNP**.
- **Ejemplo. Problemas FSAT (búsqueda) y SAT (decisión).**

Claramente, FSAT es tan o más difícil que SAT:



A partir de M´ se puede construir M, tal que: si M´ resuelve FSAT en tiempo poly(n), entonces M decide SAT en tiempo poly(n).

Menos intuitivo, también se cumple que SAT es tan o más difícil que FSAT:



A partir de M' se puede construir M, tal que: si M´ decide SAT en tiempo poly(n), entonces M resuelve FSAT en tiempo poly(n).

- Esto se cumple **para todos los lenguajes NP-completos**.

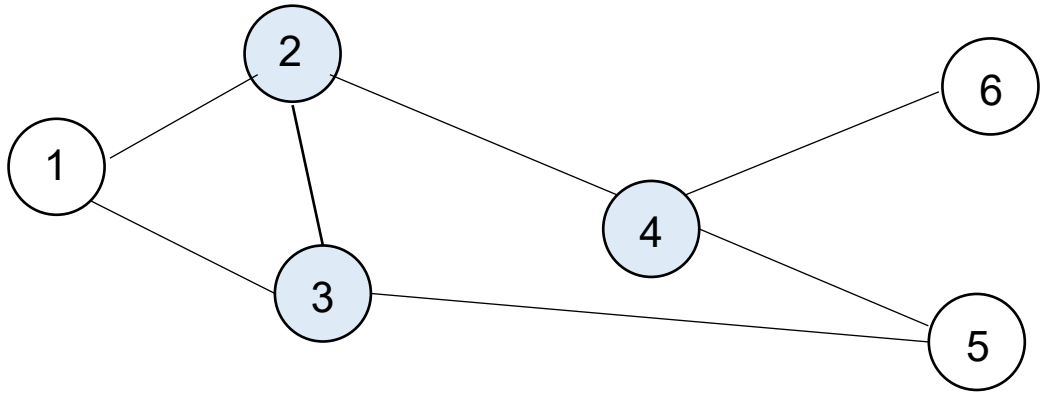
Aproximaciones polinomiales

- Cuando a un problema de búsqueda de **óptimo (máximo o mínimo)** no se le conoce resolución polinomial, se plantea resolverlo con una **aproximación polinomial** (MT de tiempo polinomial con una solución "buena").

- **Ejemplo. Mínimo cubrimiento de vértices de un grafo (no tendría resolución polinomial).**
  La siguiente MT M, en el peor caso, encuentra un cubrimiento con el **doble** de vértices del mínimo:

Dado G = (V, E), la MT M hace:

1. V' := ∅ y E' := E
2. Si E' = ∅, acepta
3. E' := E' – {(v_1, v_2)}, siendo (v_1, v_2) algún arco de E'
4. Si $v_1 \notin V'$ y $v_2 \notin V'$, entonces V' := V' U {v_1, v_2}
5. Vuelve al paso 2



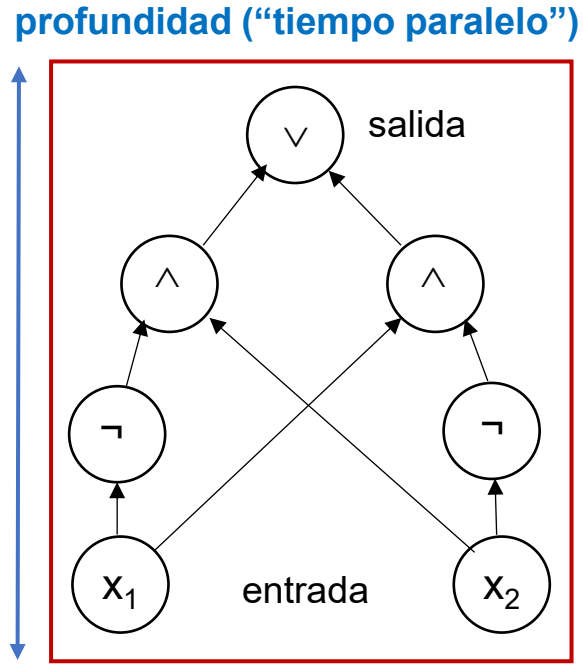
Es decir, se toman arcos no adyacentes de E y se construye un cubrimiento de vértices con sus dos extremos.

El cubrimiento mínimo del grafo de arriba tiene **tamaño 3**. La MT M puede generar, entre otros cubrimientos: V´ = {1, 2, 3, 5, 4, 6}, de **tamaño 6**, y V´ = {1, 2, 4, 5}, de **tamaño 4.**

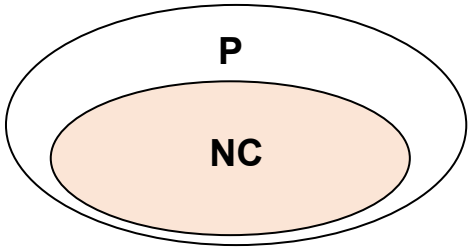
- Hay problemas con aproximaciones polinomiales muy buenas (**la diferencia con el óptimo tiende a 0**), y otros problemas que no cuentan con aproximaciones polinomiales **(cualquier diferencia fijada se supera).**

Lenguajes con algoritmos paralelos eficientes

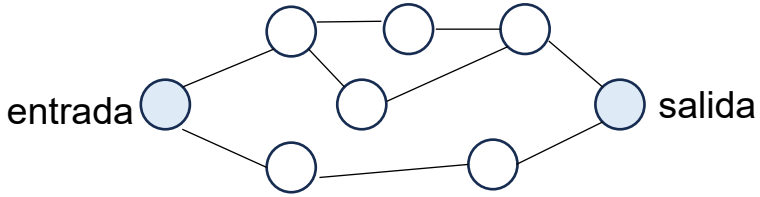
- Modelo más usual de ejecución paralela: **circuito booleano**.
- Un circuito booleano acepta una cadena sii **su salida es 1** (ver ejemplo).
- Para manejar todos los tamaños de cadenas, se usan **familias de circuitos.**
- Se define que un lenguaje cuenta con un **algoritmo paralelo eficiente,** sii lo acepta una familia de circuitos de tamaño **poly(n)** y profundidad **$\log^k(n)$.**
- La clase de estos lenguajes se llama **NC** (*Nick class*, por Nicholas Pippenger).
- Problemas clásicos de NC: **operaciones aritméticas, producto de matrices, búsqueda del camino mínimo en un grafo, ordenamiento de un vector, etc.**



Se cumple **NC ⊆ P.**
La conjetura más aceptada es que **NC ≠ P.**



Problema clásico que estaría en P – NC:
máximo flujo en una red



La jerarquía polinomial

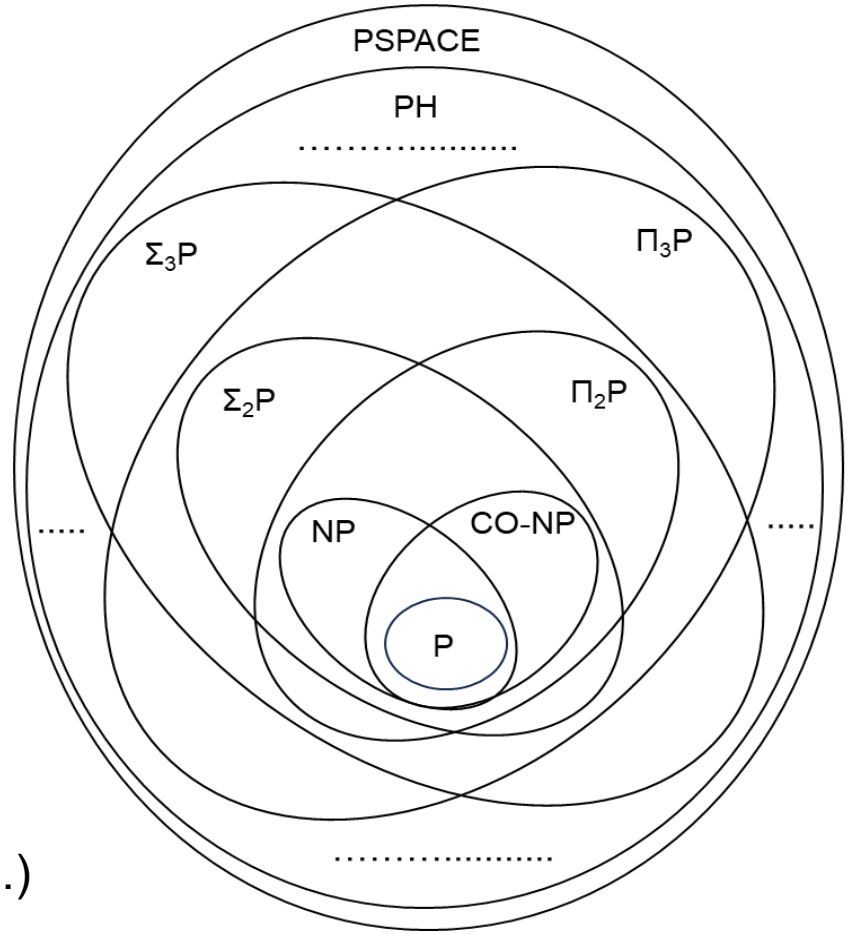
- L ∈ NP sii existe una MT M polinomial y un polinomio p tal que para toda cadena w: **w ∈ L sii (∃x: |x| ≤ p(|w| tal que M acepta (w, x))**

- Lenguajes comno los siguientes no responden a dicha definición:

  MAX-IND = {(G, K) | el grafo G tiene un conjunto independiente de K vértices y no tiene conjuntos independientes de vértices más grandes}

  MIN-FORM = {(φ, K) | la fórmula booleana φ tiene K símbolos y no existe ninguna fórmula booleana equivalente más chica}

  No siguen el esquema de los lenguajes de NP.



Conjetura aceptada:
PH no colapsa y PH ⊂ PSPACE

- L ∈ Σ_iP, con i ≥ 0, si existen una MT M polinomial y un polinomio p tal que, para toda cadena w, w ∈ L sii
  $\exists x_1$: $|x_1| \leq p(|w|, \forall x_2$: $|x_2| \leq p(|w|), \exists x_3$: $|x_3| \leq p(|w|, .., M$ acepta $(w, x_1, x_2, x_3, ..)$

  L ∈ Π_iP, con i ≥ 0, si existen una MT M polinomial y un polinomio p tal que, para toda cadena w: w ∈ L sii
  $\forall x_1$: $|x_1| \leq p(|w|, \exists x_2$: $|x_2| \leq p(|w|), \forall x_3$: $|x_3| \leq p(|w|, .., M$ acepta $(w, x_1, x_2, x_3, ..)$

  $\Sigma_0P = \Pi_0P = P$, $\Sigma_1P = NP$, $\Pi_1P =$ CO-NP

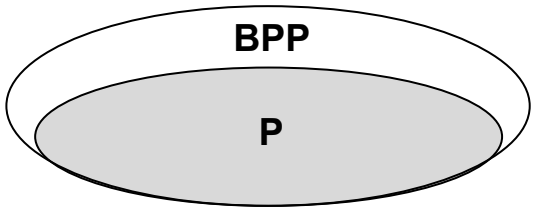
- Se prueba que ISO no es NP-completo a menos que la jerarquía polinomial colapse.

MT probabilísticas o MTP (algoritmos aleatorios)

- Una **MT probabilística** (MTP), en cada paso elige **aleatoriamente** una entre dos continuaciones, cada una con **probabilidad 1/2** ("tiro de moneda").

- Un lenguaje L pertenece a la clase **BPP** (*bounded probabilistic polynomial*) sii existe una MTP M con computaciones de tiempo poly(n) tal que:
  - Si w ∈ L, M acepta w en **al menos 2/3** de sus computaciones.
  - Si w ∉ L, M rechaza w en **al menos 2/3** de sus computaciones.

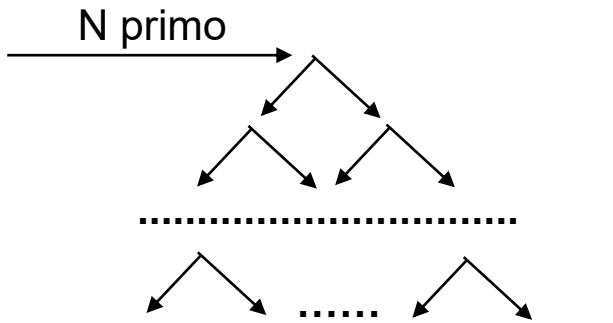
  Por lo tanto, las MTP asociadas a BPP tienen una **probabilidad de error Ɛ ≤ 1/3.**



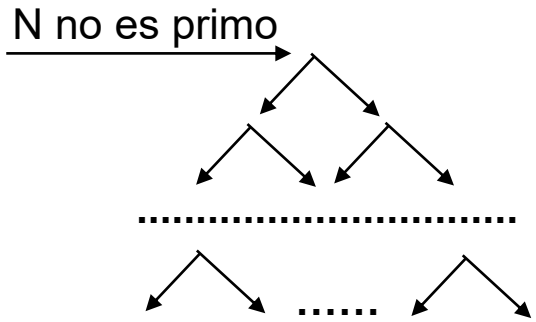
Conjetura aceptada: P = BPP
(generadores pseudoaleatorios)

Ejemplo. PRIMOS = {N | N es un número primo} pertenece a BPP (y en 2002 se demostró que pertenece a P):

Existe una MTP M que, dada una entrada N:
- Si N ∈ PRIMOS, M acepta en **al menos la mitad de sus computaciones.**
- Si N ∉ PRIMOS, M rechaza en **todas sus computaciones.**



M acepta con probabilidad ≥ 1/2



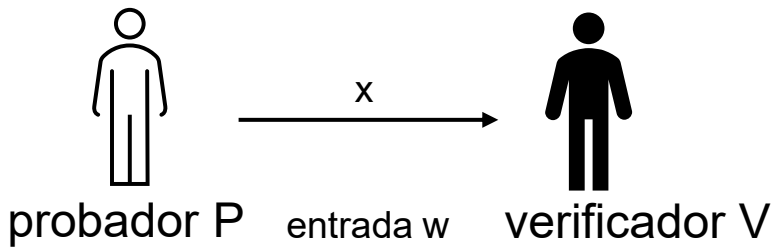
M rechaza con probabilidad = 1

Si N no es primo, M **nunca** se equivoca.

Si N es primo, M se puede equivocar pero **con probabilidad a lo sumo 1/2**.
E iterando varias veces, la probabilidad de error **tiende a cero**.

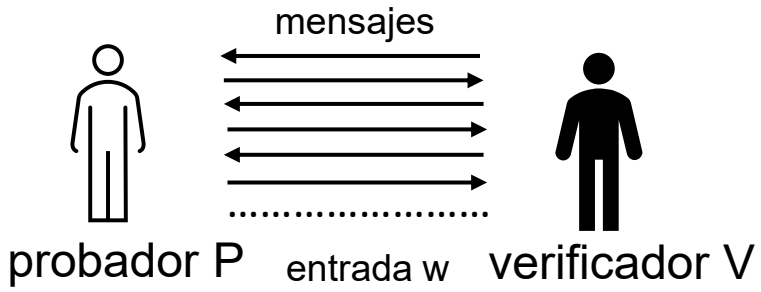
Sistemas interactivos y pruebas chequeables probabilísticamente

- **Todo lenguaje L de NP cuenta con un verificador eficiente (MT de tiempo polinomial):**



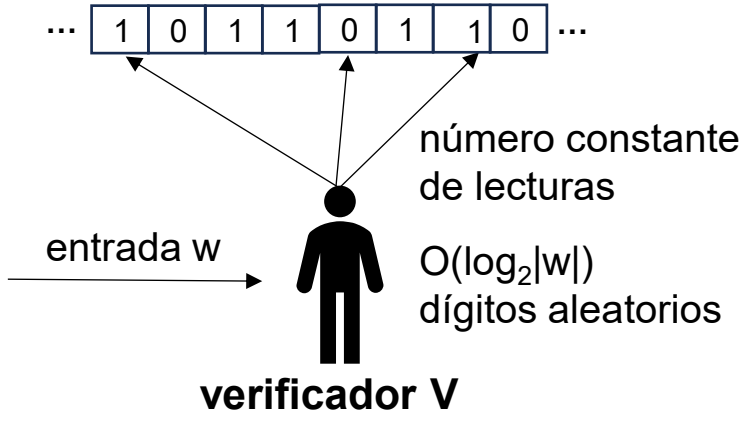
- Si w ∈ L, existe un probador P que **puede convencer** al verificador V de que w ∈ L, por medio de un certificado sucinto x.
- Si w ∉ L, **no existe ningún probador P que pueda convencer** al verificador V de lo contrario, utilice el certificado x que utilice.

- **El mismo esquema para NP se puede generalizar a una serie de interacciones entre P y V:**



- V y P se intercambian poly(|w|) mensajes de tamaño poly(|w|).
- Con ciertas asunciones, todo L de NP cuenta con un sistema interactivo en el que V no aprende nada de P (**prueba de conocimiento cero**).
- Cuando V es probabilístico, se pasa de la clase NP a la clase **PSPACE**.

- **Pruebas chequeables probabilísticamente**



- Prueba de una cadena w, a la que V accede un número **constante** de veces, usando **$\log_2|w|$** dígitos aleatorios.
- Así, es tan seguro revisar una prueba línea por línea que leer apenas unos pocos bits de la misma. Esta propiedad es útil en el área de las aproximaciones polinomiales (**Teorema PCP**).

Máquinas cuánticas o MC (algoritmos cuánticos)

- Las **máquinas cuánticas** (MC) tienen **cubits** en lugar de bits.
- Un cúbit puede estar en los estados 0 o 1, o también en un **estado de superposición** de 0 y 1.

Estados posibles de un cúbit		
|0>	|1>	$\alpha_0\|0> + \alpha_1\|1>$

α_0 y α_1 son números complejos que cumplen $|\alpha_0|^2 + |\alpha_1|^2 = 1$.

Si un cúbit está en una superposición, al medirlo se obtiene |0> con probabilidad $|\alpha_0|^2$ o |1> con probabilidad $|\alpha_1|^2$, y luego de la medición el cúbit **colapsa** al estado obtenido.

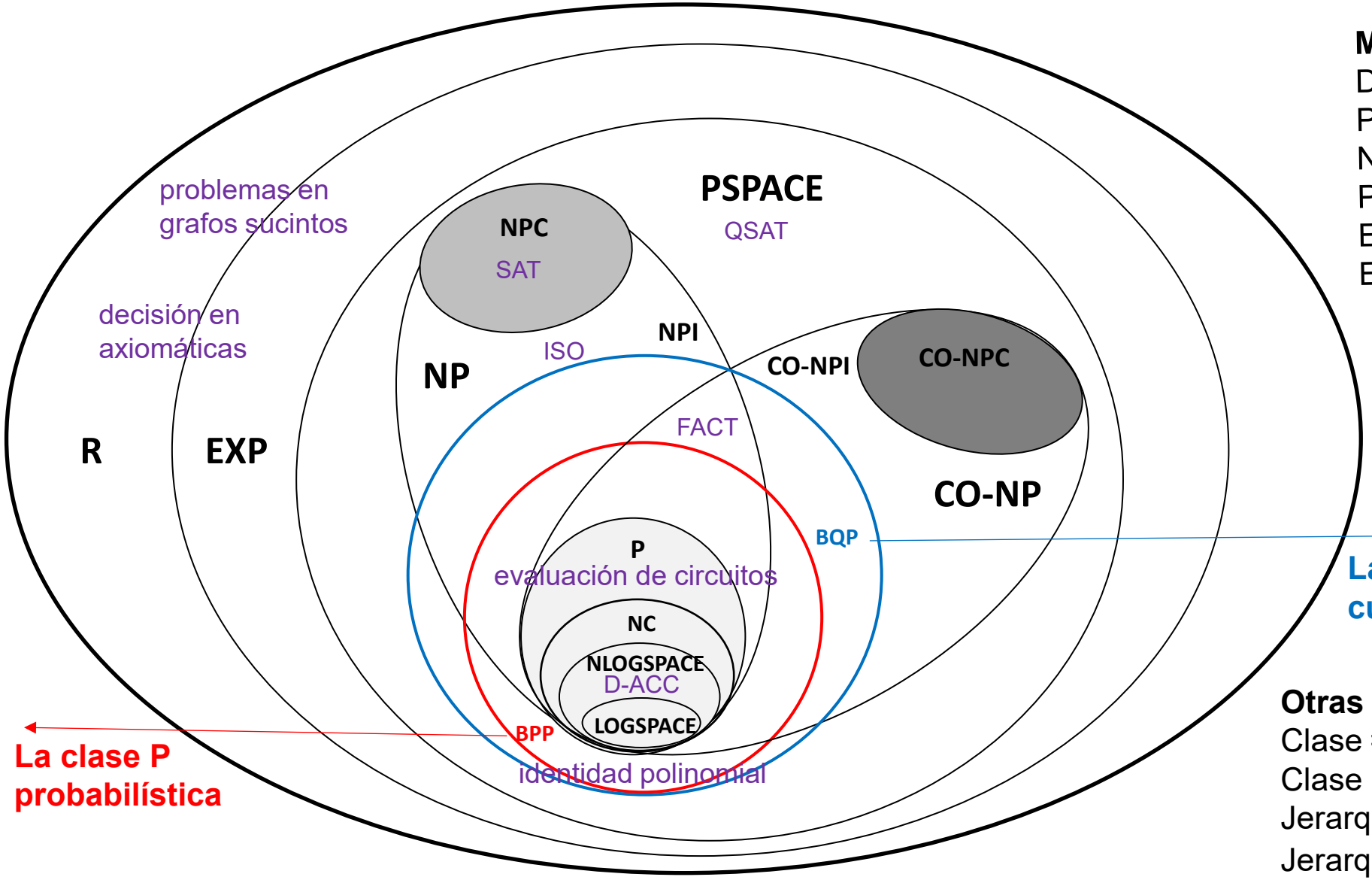
- Utilizando varios cubits, la capacidad computacional de las MC se torna muy grande:
- Los cambios en los cubits se hacen con operaciones o **puertas cuánticas**. La cantidad de puertas ejecutadas establece el **tiempo** de la MC.
- La clase **BQP** contiene los lenguajes aceptados por MC de tiempo **poly(n)**, con probabilidad de error **≤ 1/3,** que se puede reducir con muchas iteraciones.

MC con 3 cubits

α_0	000
α_1	001
α_2	010
α_3	011
α_4	100
α_5	101
α_6	110
α_7	111

- Se cumple **P ⊆ BQP** y también **BPP ⊆ BQP**. La conjetura aceptada es que las inclusiones son estrictas. Otra conjetura aceptada es que **BQP y NP son incomparables.** También se cumple **BQP ⊆ PSPACE.**
- El algoritmo cuántico de **P. Shor** resuelve la factorización en tiempo **poly(n).** El algoritmo de **Grover** acelera **cuadráticamente** las búsquedas.

Las jerarquías clásica y cuántica



Métodos de prueba
Diagonalización
P-completitud
NP-completitud
PSPACE-completitud
EXP-completitud
Etc.

La clase P cuántica

La clase P probabilística

Otras clases y jerarquías
Clase #P
Clase IP
Jerarquía polinomial PH
Jerarquía PAS $\subset$ APX $\subset$ ONP

Parte 3
Verificación de Programas

Clases 10 a 13

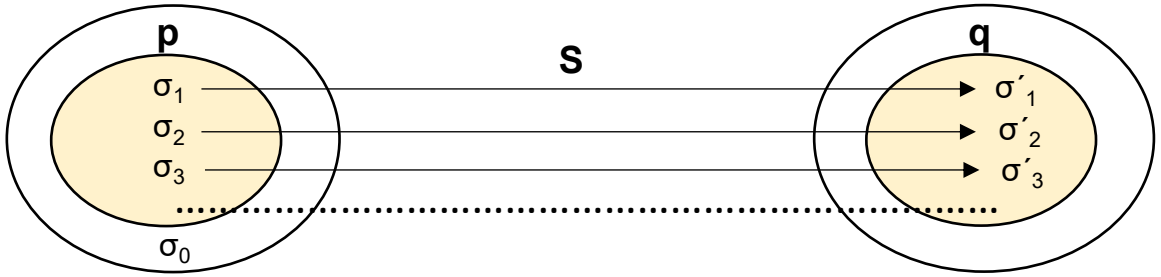
Clases 10 y 11. Definiciones y conceptos fundamentales

- Tomando como ejemplo un programa de intercambio de variables:
  - $<x = X \wedge y = Y> S_{swap} <y = X \wedge x = Y>$ es una **terna de Hoare** o **fórmula de correctitud**.
  - El predicado $x = X \wedge y = Y$ es la **precondición** de S_{swap}.
  - El predicado $y = X \wedge x = Y$ es la **postcondición** de S_{swap}.
  - El par $(x = X \wedge y = Y, y = X \wedge x = Y)$ es la **especificación** de S_{swap}.
  - Un **estado** σ es una función que asigna a toda variable un valor. Por ejemplo: $\sigma(x) = 1$, $\sigma(y) = 2$, etc.
  - Un estado σ **satisface** un predicado p, si p evaluado con σ es verdadero. Se expresa así: **$\sigma \models p$**.

    Por ejemplo: si $\sigma(x) = 1$ y $\sigma(y) = 2$, entonces $\sigma \models x < y$.

- Un programa S es **correcto con respecto a una especificación** (p, q) sii:

Para todo estado σ, **si $\sigma \models p$ entonces S ejecutado a partir de σ termina en un estado σ' tal que $\sigma' \models q$**



P. ej., si $p = (x = X > 0)$ y $q = (y = 2.X)$, S debe hacer:
Si $\sigma_1 \models x = 1$, entonces $\sigma'_1 \models y = 2$.
Si $\sigma_2 \models x = 2$, entonces $\sigma'_2 \models y = 4$.
Si $\sigma_3 \models x = 3$, entonces $\sigma'_3 \models y = 6$.
Etc.

Correctitud parcial y total de un programa

- **π(S, σ)** denota la computación de un programa S a partir de un estado inicial σ.

- **val(π(S, σ)) = σ´** denota el estado final de π(S, σ).
  En particular, **val(π(S, σ)) =** ⊥ denota que π(S, σ) no termina.

- Un programa S es **correcto parcialmente** con respecto a una especificación (p, q) sii:

  Para todo estado σ: $[\sigma \models p \ \wedge \ val(\pi(S, \sigma)) = \sigma' \neq \perp] \ \rightarrow \ \sigma' \models q$

  es decir, a partir de un estado σ |= p, si S termina (o no diverge) lo hace en un estado σ´ |= q.

- Un programa S es correcto totalmente con respecto a una especificación (p, q) sii:

  Para todo estado σ: $\sigma \models p \ \rightarrow \ [val(\pi(S, \sigma)) = \sigma' \neq \perp \ \wedge \ \sigma' \models q]$

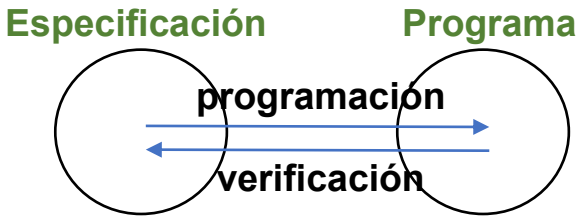
  es decir, a partir de un estado σ |= p, S termina (o no diverge) lo hace en un estado σ´ |= q.

- **{p} S {q}** denota la correctitud parcial y **⟨p⟩ S ⟨q⟩** denota la correctitud total.

Verificación sintáctica vs verificación semántica

Las dos propiedades se distinguen porque se prueban **axiomáticamente** con **métodos distintos**

Componentes de la Lógica de Hoare



1. Lenguaje de programación

- Instrucciones:

  S :: skip | x := e | S_1 ; S_2 | if B then S_1 else S_2 fi | while B do S_1 od

- Expresiones de tipo entero:

  e :: n | x | e_1 + e_2 | e_1 – e_2 | e_1 . e_2 | …

  n es una constante entera. x es una variable entera.

- Expresiones de tipo booleano:

  B :: true | false | e_1 = e_2 | e_1 < e_2 | … | ¬B_1 | B_1 ∨ B_2 | B_1 ∧ B_2 | …

Ejemplo de programa

```
a := 1 ; y := 1 ;
while a < x do
    a := a + 1 ; y := y . a
od
```

2. Lenguaje de especificación (lógica de predicados):

p :: true | false | e_1 = e_2 | e_1 < e_2 | … | ¬p | p_1 ∨ p_2 | … | ∃x: p | ∀x: p

Ejemplos de predicados

true
x + 1 = y
¬(a < x)
∀x: (x > y ∨ x ≤ y)

3.1. Axiomática para la correctitud parcial (Método H)

1. **Axioma del skip (SKIP)** $\{p\}\ \text{skip}\ \{p\}$

2. **Axioma de la asignación (ASI)** $\{p[x|e]\}\ x := e\ \{p\}$

   p[x|e] denota la sustitución en el predicado p de toda ocurrencia libre de la variable entera x libre por la expresión entera e

3. **Regla de la secuencia (SEC)**

$$\frac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$$

4. **Regla del condicional (COND)**

$$\frac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \text{if B then } S_1 \text{ else } S_2 \text{ fi } \{q\}}$$

5. **Regla de la repetición (REP)**

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \text{while B do S od } \{p \wedge \neg B\}}$$

El predicado p es un predicado invariante del while

6. **Regla de consecuencia (CONS)**

$$\frac{p \rightarrow p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$$

La regla formaliza la posibilidad de reemplazar un predicado p_1 por otro que lo implique, y/o un predicado q_1 por otro al cual implique

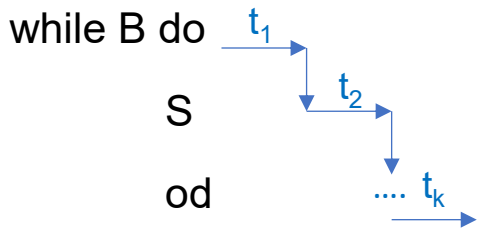
3.2. Axiomática para la correctitud total (Método H*)

- El método H* es el método H pero con la regla REP del while ampliada.
- La regla REP sólo permite probar la **invariancia de un predicado p**.
- En el método H* se usa **la regla REP***, que permite probar además la **terminación del while**.
- La regla REP* se basa en un **predicado invariante p** y una **función variante t.**

  t es una función entera definida, como el invariante, en términos de las variables de programa.

Regla REP*:

$$\frac{\langle p \wedge B\rangle\ S\ \langle p\rangle, \quad \langle p \wedge B \wedge (t = Z)\rangle\ S\ \langle t < Z\rangle, \quad p \rightarrow t \geq 0}{\langle p\rangle \text{ while } B \text{ do } S \text{ od } \langle p \wedge \neg B\rangle}$$



La función t representa el máximo número de iteraciones del while

Se agregan dos premisas a REP, y se reemplazan los símbolos { } por ⟨ ⟩.

Por la segunda premisa, **t se decrementa en cada iteración.**
Z es una variable lógica, no aparece en p, B, t ni S.
El objetivo de Z es fijar el valor de t antes de la ejecución de S.

Por la tercera premisa, **t se mantiene ≥ 0 a lo largo de todo el while.**

($\mathcal{N}$, <) es un ejemplo de relación de orden bien fundada (todo subconjunto tiene minimal).

Así, el while debe terminar, porque en $\mathcal{N}$ (números naturales) **no hay cadenas descedentes infinitas**

¿Por qué separar la prueba del while en dos, correctitud parcial y terminación?

- Porque son dos pruebas distintas, utilizan técnicas distintas:
  - Correctitud parcial: **inducción** (invariante que vale al comienzo y luego de cada iteración).
  - Terminación: **relación de orden bien fundada** (variante entero ≥ 0 que se decrementa tras cada iteración).

- En verdad, la regla REP* permite probar todo junto, pero la recomendación es probar ⟨p⟩ S ⟨q⟩ en dos pasos:

(a) {p} S {q}
(b) ⟨p⟩ S ⟨true⟩

(a) **Si S termina** a partir de p, lo hace en q.
(b) **S termina** a partir de p.
(a, b) **S termina a partir de p en q.**

$$\frac{\langle p \wedge B\rangle\ S\ \langle p\rangle,\quad \langle p \wedge B \wedge (t = Z)\rangle\ S\ \langle t < Z\rangle,\quad p \rightarrow t \geq 0}{\langle p\rangle\ \textbf{while B do S od}\ \langle q\rangle,\ \text{tal que } (p \wedge \neg B) \rightarrow q}$$

- Por ejemplo:

$$\langle x > 0\rangle\ S_{fac} :: a := 1\ ;\ y := 1\ ;\ \text{while } a < x \text{ do } a := a + 1\ ;\ y := y\ .\ a \text{ od } \langle y = x!\rangle$$

Para probar $\{x > 0\}\ S_{fac}\ \{y = x!\}$ se utiliza el invariante $\mathbf{y = a!} \wedge \mathbf{a \leq x}$,

Para probar $\langle x > 0\rangle\ S_{fac}\ \langle true\rangle$, se utiliza un invariante más simple: $\mathbf{a \leq x}$.
evitando arrastrar información innecesaria.

Composicionalidad

- El método de prueba presentado es **composicional**:
  Dado un programa S, compuesto por subprogramas S_1, …, S_n,
  que valga la fórmula {p} S {q} depende **sólo** de que valgan fórmulas $\{p_1\}\ S_1\ \{q_1\}$, …, $\{p_n\}\ S_n\ \{q_n\}$,
  sin importar el contenido de los S_i (noción de **caja negra**).

- Por ejemplo, dado el programa S :: S_1 ; S_2, si se cumplen las fórmulas: **$\{p\}\ S_1\ \{r\}$ y $\{r\}\ S_2\ \{q\}$,**
  también se cumple la fórmula: **$\{p\}\ S_1\ ;\ S_2\ \{q\}$,**
  independientemente del contenido de S_1 y S_2.

- Más aún, si en lugar de **S_2** utilizamos un subprograma **S_3** que también satisface la fórmula: **$\{r\}\ S_3\ \{q\}$,**
  entonces también se cumple la fórmula: **$\{p\}\ S_1\ ;\ S_3\ \{q\}$,**
  lo que significa que S_2 y S_3 son **intercambiables** (son funcionalmente equivalentes respecto de (r, q)).

- **En los programas concurrentes la composicionalidad se pierde.**

Especificaciones

- Hemos especificado un programa para calcular el factorial de la siguiente forma:

$$(x > 0, y = x!)$$

Pero la especificación no es correcta:

El programa **S :: x := 1 ; y := 1** satisface (x > 0, y = x!) pero no es el programa pedido:
Por ejemplo, si al inicio x = 5, entonces al final debe ser y = 5! = 120.

- Lo que sucede es que **las variables de la precondición pueden modificarse a lo largo del programa**.

- Lo que se hace es utilizar **variables lógicas**, para **congelar valores**. En ej ejemplo considerado haríamos:

$$(x = X \wedge X > 0, y = X!\}$$

Por otro lado, no se puede agregar a la especificación que la variable x no se modifique nunca:

La lógica de predicados no lo permite. Una lógica que sí lo permite es la **lógica temporal**.

Proof outlines (esquemas de prueba)

- Presentación alternativa de una prueba: se intercalan los pasos de la prueba entre las instrucciones del programa.
- Se obtiene una prueba más estructurada, que documenta adecuadamente el programa.
- En la verificación de los programas concurrentes las *proof outlines* son **imprescindibles**.
- Por ejemplo, la siguiente es una *proof outline* de correctitud parcial de un programa que calcula el factorial:

$\{x > 0\}$
S_{fac} :: a := 1 ; y := 1 ;
while a < x do
a := a + 1 ; y := y . a
od
$\{y = x!\}$

$\{x > 0\}$
a := 1 ;
y := 1 ;
$\{inv: y = a! \wedge a \leq x\}$
while a < x do
$\{y = a! \wedge a < x\}$
a := a + 1 ;
y := y . a
od
$\{(y = a! \wedge a \leq x) \wedge \neg(a < x)\}$
$\{y = x!\}$

En una proof outline de correctitud total se agrega el variante.

- Mínimamente se suelen documentar la **precondición**, la **postcondición,** los **invariantes** y los **variantes**.

Invariantes, variantes, y propiedades *safety* (seguridad) y *liveness* (progreso)

- Todo predicado utilizado en una prueba es en realidad un **invariante**. Volviendo a las *proof outline*s: cualquiera sea el estado inicial, todo predicado siempre se cumple en el lugar donde se establece:

{x > 0}
a := 1 ; y := 1 ;
{y = a! ∧ a ≤ x}
while a < x do
{y = a! ∧ a < x}
a := a + 1 ; y := y . a
od
{y = x!}

- El uso de un invariante para la prueba de un *while* determina una prueba por **inducción**: es un predicado que vale al comienzo del *while* (base inductiva) y que toda iteración preserva (paso inductivo). De esta manera se prueba que el invariante se cumple **a lo largo de toda la computación** del *while*.

- Ligado a lo anterior, la correctitud parcial pertenece a la familia de las propiedades ***safety***. Son propiedades que se prueban por **inducción**. Se asocian al enunciado: “Algo malo no puede suceder”. Otros ejemplos son la *ausencia de deadlock* y la *exclusión mutua* o *ausencia de interferencia*, en los programas concurrentes.

- La terminación o no divergencia, en cambio, pertenece a la familia de las propiedades **liveness**, basadas en **funciones variantes** definidas en **relaciones de orden bien fundadas.** Se asocian al enunciado: “Algo bueno va a suceder”. Otro ejemplo es la *ausencia de inanición (non starvation),* en los programas concurrentes.

Axiomas y reglas adicionales

- Por la **completitud** del método de prueba estudiado (que se prueba en la clase que viene), agregarle axiomas y reglas resulta **redundante**. De todos modos, esta práctica es usual en los sistemas deductivos, facilita y acorta las pruebas. Algunos ejemplos clásicos de axiomas y reglas adicionales son:

- Axioma de invariancia (INV) $\{p\}\ S\ \{p\}$

  Cuando las variables libres de p y las variables que modifica S **son disjuntas**.

- Regla de la disyunción (OR)

$$\frac{\{p\}\ S\ \{q\},\ \{r\}\ S\ \{q\}}{\{p \vee r\}\ S\ \{q\}}$$

  Util para una **verificación por casos**.

- Regla de la conjunción (AND)

$$\frac{\{p_1\}\ S\ \{q_1\},\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$$

  También útil para una **verificación por casos**.

El axioma INV suele emplearse en combinación con la regla AND, para producir la **Regla de invariancia (RINV):**

$$\frac{\{p\}\ S\ \{q\}}{\{r \wedge p\}\ S\ \{r \wedge q\}}$$

tal que ninguna variable libre de r es modificable por S (se cumple {r} S {r}).

Clase 12. Sensatez y completitud de la Lógica de Hoare

- Los métodos H y H* son **sensatos** y **completos**.

- Formalmente, si:

  $\vdash_H \{p\}\ S\ \{q\}$ expresa que H permite probar **sintácticamente** $\{p\}\ S\ \{q\}$

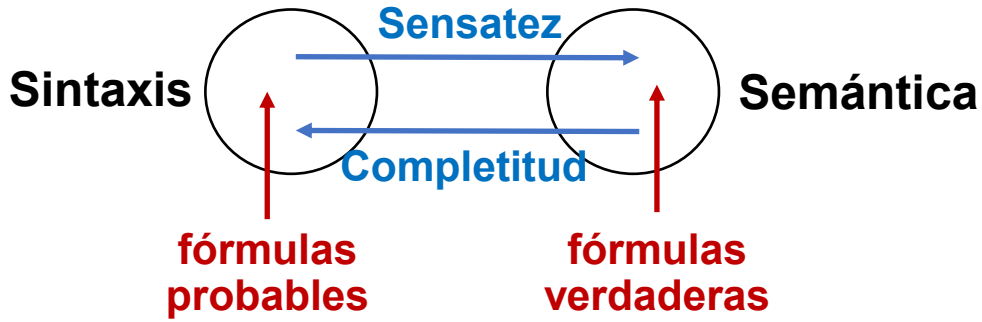
  $\vdash_{H^*} \langle p\rangle\ S\ \langle q\rangle$ expresa que H* permite probar **sintácticamente** $\langle p\rangle\ S\ \langle q\rangle$

  $\models \{p\}\ S\ \{q\}$ expresa que se cumple **semánticamente** $\{p\}\ S\ \{q\}$ (correctitud parcial de S con respecto a (p, q))

  $\models \langle p\rangle\ S\ \langle q\rangle$ expresa que se cumple **semánticamente** $\langle p\rangle\ S\ \langle q\rangle$ (correctitud total de S con respecto a (p, q))

  Se cumple que para todo programa S y para toda especificación (p, q) valen las implicaciones:

  1. $\mathbf{\vdash_H \{p\}\ S\ \{q\} \rightarrow \models \{p\}\ S\ \{q\}}$ (sensatez de H)
  2. $\mathbf{\models \{p\}\ S\ \{q\} \rightarrow \vdash_H \{p\}\ S\ \{q\}}$ (completitud de H)
  3. $\mathbf{\vdash_{H^*} \langle p\rangle\ S\ \langle q\rangle \rightarrow \models \langle p\rangle\ S\ \langle q\rangle}$ (sensatez de H*)
  4. $\mathbf{\models \langle p\rangle\ S\ \langle q\rangle \rightarrow \vdash_{H^*} \langle p\rangle\ S\ \langle q\rangle}$ (completitud de H)



- En palabras:
  - Las fórmulas que prueban H y H* son **verdaderas** (sensatez).
  - Todas las fórmulas verdaderas **se pueden proba**r en H y H* (completitud).

Inducción matemática

- Sea P una propiedad a probar en el dominio $\mathcal{N}$ de los números naturales.

- Si

  1) **P(0)**

  2) **Para todo k de $\mathcal{N}$: P(k) ⟶ P(k + 1)**

  entonces

  3) **Para todo n de $\mathcal{N}$: P(n)**

- (1) es la **base inductiva**

  (2) es el **paso inductivo**

  P(k) es la **hipótesis inductiva**

- <u>Variante fuerte de la inducción matemática</u>

  Paso inductivo: $(P(i) \wedge P(i + 1) \wedge \ldots \wedge P(k - 1) \wedge P(k)) \longrightarrow P(k + 1)$, para algún i

Inducción estructural

- Generalización de la inducción matemática.
- Util también para definir conjuntos.
- Se aplica a cualquier dominio, no sólo $\mathcal{N}$, considerando una determinada relación menor <.
- Puede haber varios minimales, varias bases inductivas y varios pasos inductivos.

- **Ejemplo**

  Definición por inducción estructural de las expresiones aritméticas con símbolos de {0 , 1 , x , + , . , (,)}:

  (a) bases inductivas: 0, 1, x, son expresiones aritméticas.

  (b) pasos inductivos: si e_1 y e_2 son expresiones aritméticas, también lo son $(e_1 + e_2)$ y $(e_1 . e_2)$.

Semántica operacional del lenguaje de programación

- **Idea general.** Dados un programa S y un estado inicial σ, a dicha **configuración inicial** (S, σ) se le asocia una **computación** π(S, σ), que es la secuencia de configuraciones de la ejecución de S a partir de σ.
  P.ej.: (x := 0 ; y := 1 ; z := 2, σ) → (y := 1 ; z := 2, σ[x|0]) → (z := 2, σ[x|0][y|1]) → (E, σ[x|0][y|1][z|2]).

- **Notación.** val(π(S, σ)) es el estado final de π(S, σ). Si π(S, σ) es infinita, val(π(S, σ)) = ⊥.
  P.ej.: val(π(while true do skip od, σ)) = ⊥, y val(π(x := 10 ; while x > 0 do x := x – 1 od, σ)) = σ[x|0].

- **Semántica de las instrucciones.** Se define por inducción estructural, mediante una relación → de **transición de configuraciones** (pares (S, σ) con **continuación sintáctica** S y estado σ):

  1. (skip, σ) → (E, σ)
     El skip se consume en un paso (es atómico) y no modifica el estado inicial. E es la continuación sintáctica vacía.
  2. (x := e, σ) → (E, σ[x|σ(e)])
     La asignación también es atómica y el estado final es como el inicial salvo que el valor de la variable x es el de la expresión e según σ.
  3. Para toda instrucción T, si (S, σ) → (S', σ'), entonces (S ; T, σ) → (S' ; T, σ')
     La secuencia S ; T se ejecuta de izquierda a derecha. Una vez consumido S, si no diverge, se ejecuta T. Se define E ; S = S ; E = S.
  4. (if B then S_1 else S_2 fi, σ) → (S_1, σ), si se cumple σ(B) = V.
     (if B then S_1 else S_2 fi, σ) → (S_2, σ), si se cumple σ(B) = F.
     La evaluación de la expresión B es atómica y no modifica el estado inicial. Su evaluación determina si se ejecuta S_1 o S_2.
  5. (while B do S od, σ) → (S ; while B do S od, σ), si se cumple σ(B) = V.
     (while B do S od, σ) → (E, σ), si se cumple σ(B) = F.
     La evaluación de B es atómica y no modifica el estado inicial. Su evaluación determina si se ejecuta S o se termina la repetición.

Sensatez del método H

- Se cumple, para todo programa S y toda especificación (p, q), que: **$\vdash_H$ {p} S {q} $\rightarrow$ |= {p} S {q}.**

- Lo probaremos por inducción matemática fuerte, considerando la **longitud de la prueba** (1 o más pasos):
  (a) Base inductiva: los **axiomas** son verdaderos (pruebas de tamaño 1).
  (b) Paso inductivo: las **reglas** son sensatas, preservan la verdad de las premisas (pruebas de tamaño ≥ 2).

- Ejemplo de (a). Prueba de que el axioma **SKIP** es verdadero.

  - Dado |– {p} skip {p}, hay que probar **|= {p} skip {p}.**
  - Por la semántica del lenguaje: (skip, σ) $\rightarrow$ (E, σ).
  - Sea **σ |= p.** Luego del skip se cumple **σ |= p.**

- Ejemplo de (b). Prueba de que la regla **SEC** es sensata.

  - Dado |– {p} S_1 ; S_2 {q}, hay que probar **|= {p} S_1 ; S_2 {q}.**
  - |– {p} S_1 ; S_2 {q} se obtiene de **|– {p} S_1 {r} y |– {r} S_2 {q}** (pruebas más cortas que |– {p} S_1 ; S_2 {q}).
  - Entonces, por hipótesis inductiva: **|= {p} S_1 {r}** y **|= {r} S_2 {q}.** Veamos que se cumple **|= {p} S_1 ; S_2 {q}:**
  - Sea **σ_1 |= p,** y asumamos que S_1 ; S_2 termina desde σ_1 (si no temina, la terna se cumple trivialmente).
  - Por la semántica del lenguaje y la hipótesis inductiva: (S_1 ; S_2, σ_1) $\rightarrow$ … (S_2, σ_2) $\rightarrow$ … (E, σ_3), tal que se cumple σ_2 |= r y **σ_3 |= q,** que es lo que queríamos demostrar.

Completitud del método H

- Se cumple, para todo programa S y toda especificación (p, q), que **$\models \{p\}\ S\ \{q\} \rightarrow \vdash_H \{p\}\ S\ \{q\}$.**

- Lo probaremos por inducción estructural, considerando las **5 formas que pueden tener los programas.**
  (a) Base inductiva: considera los **programas atómicos:** skip y x := e.
  (b) Paso inductivo: considera los **programas compuestos:** secuencia, if then else y while.

- Ejemplo de (a). Prueba de que: **$\models \{p\}\ skip\ \{q\} \rightarrow \vdash \{p\}\ skip\ \{q\}$.**

  - Se tiene **$\models \{p\}\ skip\ \{q\}$.**
  - Por la semántica del skip vale $\models \{p\}\ skip\ \{p\}$, por lo que debe cumplirse **$p \rightarrow q$**.
  - De esta manera se logra: **$\vdash \{p\}\ skip\ \{q\}$:**

    1. $\{p\}\ skip\ \{p\}$ (SKIP)
    2. $p \rightarrow q$ (MAT)
    3. $\{p\}\ skip\ \{q\}$ (CONS, 1, 2)

- Ejemplo de (b). Prueba de que: **$\models \{p\}$ if B then S_1 else S_2 fi $\{q\} \rightarrow \vdash \{p\}$ if B then S_1 else S_2 fi $\{q\}$.**

  - Se tiene **$\models \{p\}$ if B then S_1 else S_2 fi $\{q\}$.**
  - Por la semántica del if vale $\models \{p \wedge B\}\ S_1\ \{q\}$ y $\models \{p \wedge \neg B\}\ S_2\ \{q\}$ (S_1 y S_2 son más simples que todo el if).
  - Entonces, por hipótesis inductiva: **$\vdash \{p \wedge B\}\ S_1\ \{q\}$** y **$\vdash \{p \wedge \neg B\}\ S_2\ \{q\}$.**
  - De esta manera, aplicando la regla COND, se logra: **$\vdash \{p\}$ if B then S_1 else S_2 fi $\{q\}$.**

- Otro ejemplo de (b). Prueba de que: **|= {p} S_1 ; S_2 {q}** $\rightarrow$ **|– {p} S_1 ; S_2 {q}:**
  - Se tiene **|= {p} S_1 ; S_2 {q}.**
  - Por la semántica de la secuencia vale |= {p} S_1 {r} y |= {r} S_2 {q} (S_1 y S_2 son más simples que S_1 ; S_2).
  - Entonces, por hipótesis inductiva: **|– {p} S_1 {r}** y **|– {r} S_2 {q}.**
  - De esta manera, aplicando la regla SEC, se logra: **|– {p} S_1 ; S_2 {q}**.

- Pero por qué se puede asegurar que existe un predicado r expresable en la lógica de predicados:

  Porque la lógica de predicados es **expresable** con respecto al lenguaje de programación definido y el dominio de los números enteros (no tiene por qué cumplirse en otros casos):

  Para todo predicado p y todo programa S, siempre se puede expresar la postcondición q de {p} S {q}

- Lo mismo ocurre en la prueba de |= {p} while B do S od {q} $\rightarrow$ |– {p} while B do S od {q}:

  Siempre se pueden expresar los invariantes.

- Pero H no es completo en términos **absolutos**:
  - La axiomática de los números enteros es incompleta (Teorema de Gödel).
  - Por lo tanto, hay que asumir que H tiene todos los axiomas de los enteros.
  - P.ej., si vale |= {true} skip {p}, para probar |– {true} skip {p} hay que asumir que p es un axioma de H.
  - De todos modos, de lo que se trata es de probar programas, no enunciados de los números enteros.

- Lo anterior impacta en la automatización.

Sensatez del método H*

$$\frac{\langle p \wedge B\rangle\ S\ \langle p\rangle\ ,\ \langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle\ ,\ p \rightarrow t \geq 0}{\langle p\rangle \text{ while B do S od } \langle p \wedge \neg B\rangle}$$

regla REP*

- Sólo falta probar que la regla REP* es sensata:

  $\vdash_{H^*} \langle p\rangle$ while B do S od $\langle p \wedge \neg B\rangle \rightarrow \models \langle p\rangle$ while B do S od $\langle p \wedge \neg B\rangle$

Inducción matemática fuerte

Sea: **$\vdash \langle p\rangle$ while B do S od $\langle p \wedge \neg B\rangle$**

Entonces: $\vdash \langle p \wedge B\rangle\ S\ \langle p\rangle$, $\vdash \langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle$, $p \rightarrow t \geq 0$

Por hip. ind.: **$\models \langle p \wedge B\rangle\ S\ \langle p\rangle$, $\models \langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle$, $p \rightarrow t \geq 0$** (S es más simple que while B do S od)

Veamos que: **$\models \langle p\rangle$ while B do S od $\langle p \wedge \neg B\rangle$**

Sea: **$\sigma_0 \models p$**

Hay que probar que el while termina desde σ_0 en un estado **$\sigma_k \models p \wedge \neg B$**

Supondremos que no, y llegaremos a una contradicción:

Como: **(while B do S od, σ_0) $\rightarrow$.. (while B do S od, σ_1) $\rightarrow$.. (while B do S od, σ_2) $\rightarrow$.., con $\sigma_i \models ((p \wedge B) \wedge t \geq 0)$**

Entonces: **$\sigma_0(t)$ > $\sigma_1(t)$ > $\sigma_2(t)$ > …**

lo que es absurdo porque esta cadena es una cadena descendente infinita de números naturales.

Por lo tanto, el while termina y lo hace en un estado $\sigma_k \models p \wedge \neg B$

Completitud del método H*

$$\frac{\langle p \wedge B\rangle\ S\ \langle p\rangle\ ,\ \langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle\ ,\ p \rightarrow t \geq 0}{\langle p\rangle \textbf{ while B do S od } \langle p \wedge \neg B\rangle}$$

regla REP*

- Sólo falta probar que la regla REP* es completa:

  $\models \langle p\rangle$ while B do S od$S\ \langle q\rangle \rightarrow \vdash_{H^*} \langle p\rangle$ while B do S od $\langle q\rangle$

<u>Inducción estructural</u>

La idea central es probar la **expresividad** del **invariante** y el **variante**.

La expresividad del invariante ya la comentamos en la prueba sobre H (ver más detalle en el Anexo).

Veamos el caso del **variante**:

Hay que definir una **función entera t** que exprese **la cantidad de iteraciones del while:**

Sea **S :: while B do T od** y un estado inicial **σ.**

Sea **iter(S, σ)** una función parcial que especifica la cantidad de iteraciones de S a partir de σ.

La función iter(S, σ) es **computable**:

S_x :: x := 0 ; while B do x := x + 1 ; T od

Si S termina en un estado σ´, **iter(S, σ) = σ´(x)** contiene la cantidad de iteraciones del while.

En definitiva, el lenguaje de especificación debe poder expresar **todas las funciones computables.**

Clase 13. Misceláneas. Verificación de programas concurrentes.

Modelo 1. Lenguaje de variables compartidas

- Programas de la forma:

$$\mathbf{S :: S_0 ; [S_0 \,||\, \ldots \,||\, S_n]}$$

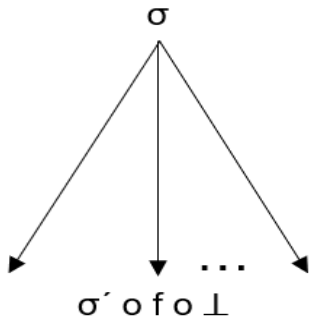
  tal que S_0 tiene inicializaciones y $[S_0 \,||\, \ldots \,||\, S_n]$ es una composición paralela de procesos $S_0, \ldots, S_n$.

- Los procesos pueden **compartir variables** y pueden **sincronizarse** con una instrucción *await*:

$$\text{await B} \rightarrow \text{S end}$$

  tal que si se cumple B, S se ejecuta atómicamente, y si no, el proceso se **bloquea** hasta que B se cumpla.

- Un programa puede tener **varias computaciones** (semántica de *interleaving*), y así, a partir de un estado inicial, puede producir **varios estados finales**:



Computaciones con 3 posibilidades

1) Finitas sin falla (estado final σ´)
2) Finitas con *deadlock* (estado de falla f)
3) Infinitas (estado indefinido ⊥)

¿En qué consiste ahora entonces la correctitud total, es decir, cuándo se cumple la fórmula **⟨p⟩ S ⟨q⟩**?

Cuando para todo estado inicial σ |= p, **toda computación** de S termina en un estado final σ´ |= q.

Pérdida de la composicionalidad, si se imita la Lógica de Hoare del paradigma secuencial

- Se cumple: $\{x = 0\}\ S_1 :: x := x + 2\ \{x = 2\}$ y $\{x = 0\}\ S_2 :: z := x\ \{z = 0\}$ **pero no se cumple:** $\{x = 0 \wedge x = 0\}\ [S_1 :: x := x + 2 \parallel S_2 :: z := x]\ \{x = 2 \wedge z = 0\}$

  porque si en el programa $[S_1 \parallel S_2]$ se ejecuta S_2 después de S_1, al final se cumple z = 2. En efecto vale:

$$\{x = 0 \wedge x = 0\}$$
$$[S_1 :: x := x + 2 \parallel S_2 :: z := x]$$
$$\{x = 2 \wedge (z = 0 \vee z = 2)\}$$

- Y además, cambiando $S_1 :: x := x + 2$ por el proceso equivalente $S_3 :: x := x + 1 ; x := x + 1$, no vale:

$$\{x = 0 \wedge x = 0\}$$
$$[S_3 :: x := x + 1 ; x := x + 1 \parallel S_2 :: z := x]$$
$$\{x = 2 \wedge (z = 0 \vee z = 2)\}$$

  porque si S_2 se ejecuta entre las dos asignaciones de S_3, al final se cumple z = 1. En efecto, vale:

$$\{x = 0 \wedge x = 0\}$$
$$[S_3 :: x := x + 1 ; x := x + 1 \parallel S_2 :: z := x]$$
$$\{x = 2 \wedge (z = 0 \vee z = 1 \vee z = 2)\}$$

  con lo que en la concurrencia dos procesos funcionalmente equivalentes **no son intercambiables.**

- **Se pierde la noción de caja negra**. Se plantean distintas técnicas de remediación.

Un par de soluciones al problema de la pérdida de la composicionalidad

1. **Cambio en la manera de especificar,** para obtener composicionalidad, fortaleciendo la pre y postcondición con condiciones **rely** (qué espera un proceso del resto), y **guarantee** (qué le asegura cada proceso al resto. **Método de Jones**.

2. **Prueba en dos etapas** (lo más habitual, y es lo que describiremos). **Método de Owicki y Gries**.

   Etapa 1. Pruebas locales de cada proceso (*proof outlines*).

   Etapa 2. Chequeo global de consistencia de las pruebas de la primera etapa. Se chequea que todos los predicados sean verdaderos, cualquiera sea el *interleaving* ejecutado.

   Por ejemplo, considerando un programa genérico, la **prueba de correctitud parcial** se plantea así:

S_1		S_2		S_3
$\{p_1\}$		$\{p_2\}$		$\{p_3\}$
$[S_{11}$		S_{21}		S_{31}
$\{p_{12}\}$	$\|\|$	$\{p_{22}\}$	$\|\|$	$\{p_{32}\}$
S_{12}		S_{22}		S_{32}
…..		…..		…..
$\{p_{1k1}\}$		$\{p_{2k2}\}$		$\{p_{3k3}\}$
S_{1k1}		S_{2k2}		$S_{3k3}]$
$\{q_1\}$		$\{q_2\}$		$\{q_3\}$

pruebas locales → prueba global

S_1 vs S_2

$\{p_1 \wedge p_2\}\ S_{21}\ \{p_1\}$
$\{p_1 \wedge p_{22}\}\ S_{22}\ \{p_1\}$
……
$\{q_1 \wedge p_{2k2}\}\ S_{2k2}\ \{q_1\}$

S_1 vs S_3

$\{p_1 \wedge p_3\}\ S_{31}\ \{p_1\}$
$\{p_1 \wedge p_{32}\}\ S_{32}\ \{p_1\}$
……
$\{q_1 \wedge p_{3k3}\}\ S_{3k3}\ \{q_1\}$

Y lo mismo con el resto de los pares

Los predicados deben ser **invariantes** (las *proof outlines deben ser* **libres de interferencia).**

Idea general de la prueba de terminación

- Hay que probar que todas las computaciones sean **finitas**.

- La prueba parte también de *proof outlines* **libres de interferencia**.
  Pero debe contemplar un aspecto adicional. Dadas *proof outlines*:

$$\begin{array}{cccc} \langle p_1 \rangle & \langle p_2 \rangle & & \langle p_n \rangle \\ S_1^* & S_2^* & \ldots & S^*_n \\ \langle q_1 \rangle & \langle q_2 \rangle & & \langle q_n \rangle \end{array}$$

  se debe chequear en **cada *while* de cada proceso S_i**
  que el variante t asociado **no sea incrementado** por una instrucción T de otro proceso S_j
  (sí puede ser decrementado, en cuyo caso el proceso adelanta su terminación).

  Formalmente: $\langle t = Z \wedge pre(T) \rangle\ T\ \langle t \leq Z \rangle$, siendo pre(T) la precondición de T

S_i	S_j
{t = Z}	
while	
do	{Pre(T)}
S ←	T
od	
{t ≤ Z}	

T es una asignación
o es un await

- La prueba debe contemplar también las **hipótesis de *fairness*** existentes:
  débil: "todo proceso siempre habilitado para ejecutarse se ejecuta alguna vez", o bien:
  fuerte: "todo proceso infinitamernte habilitado para ejecutarse se ejecuta alguna vez".

  P.ej., con *fairness* débil, el programa S desde x ≠ 0 termina,
  a pesar de que el *while* considerado aisladamente no:

  S :: [while x ≠ 0 do skip od || x := 0]

Idea general de la prueba de ausencia de *deadlock*

- Hay que probar que todas las computaciones terminen **sin *deadlock***.
- La prueba nuevamente parte de *proof outlines* **libres de interferencia**.
- Hay que plantear todas las posibles situaciones de *deadlock* y probar que **no ocurren.**
- Dadas *proof outlines*:

$$\begin{array}{llll} \langle p_1 \rangle & \langle p_2 \rangle & & \langle p_n \rangle \\ S_1^* & S_2^* & \ldots & S^*_n \\ \langle q_1 \rangle & \langle q_2 \rangle & & \langle q_n \rangle \end{array}$$

Paso 1. Se especifican todas las tuplas de predicados que representan potenciales casos de *deadlock*:

$$\begin{array}{l} (p_{11}, p_{12}, \ldots, p_{1n}) \\ (p_{21}, p_{22}, \ldots, p_{2n}) \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ (p_{k1}, p_{k2}, \ldots, p_{kn}) \end{array} \longleftarrow$$

Los p_{ij} se refieren a awaits bloqueados o son postcondiciones

Paso 2. Se chequea en **cada tupla $(p_{i1}, p_{i2}, \ldots, p_{in})$** que el predicado **$p_{i1} \wedge p_{i2} \wedge \ldots \wedge p_{in}$ sea falso**.

Por ejemplo, dado el programa:

$$\begin{array}{lll} \{p_1\} & & \{p_2\} \\ [\text{await } B \longrightarrow S \text{ end} & \| & T] \\ \{q_1\} & & \{q_2\} \end{array}$$

la única tupla a considerar es $(p_1 \wedge \neg B, q_2)$ y el paso 2 consiste en chequear que:

$(p_1 \wedge \neg B \wedge q_2)$ sea falso

Modelo 2. Lenguaje de recursos de variables

- También en este caso los procesos **pueden compartir variables**. Los programas son de la forma:

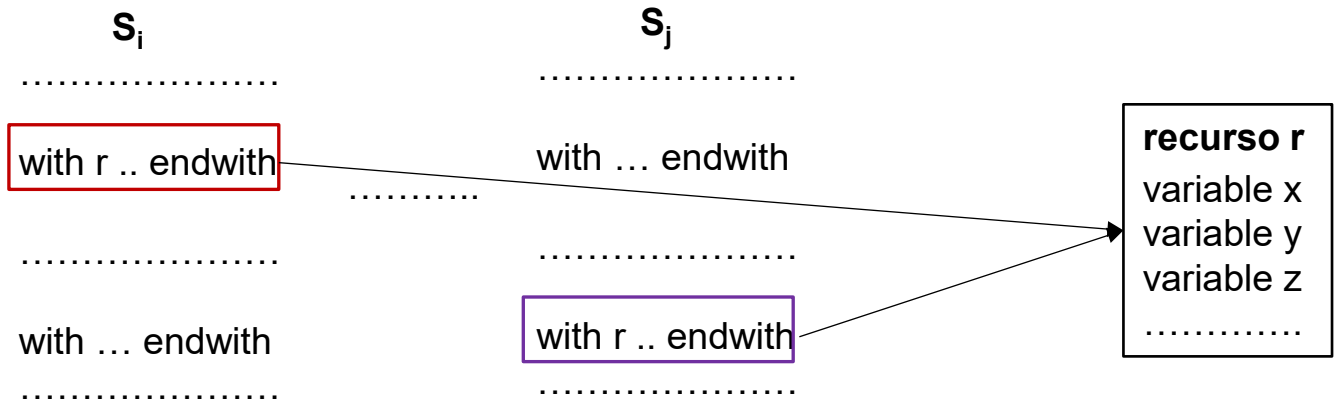
  S :: resource r_1 ($lista_1$) ; … ; resource r_m ($lista_m$) ; S_0 ; [S_1 || … || S_n]

  donde los **r_k** son **recursos de variables**, conjuntos disjuntos de variables definidas en las listas **$lista_k$.**

- Los procesos se **sincronizan** con la instrucción *with*, que provee **acceso exclusivo** a los recursos:

  with r_k when B do S endwith

  1. Cuando un proceso ejecuta una instrucción *with r_k when B do S endwith*, si r_k está libre y B es verdadera, entonces el proceso puede ocupar r_k, utilizar sus variables y progresar en su ejecución.
  2. Cuando el proceso termina de ejecutar el *with*, *libera* r_k.



Sólo un proceso accede por vez a un mismo recurso. Los *with* permiten visualizar las regiones críticas.

- La novedad en la verificación en este caso, en comparación con el modelo anterior, es el uso de **invariantes de recursos** en las *proof outlines*, los cuales se cumplen cuando los recursos **están libres.**

- P. ej., asumiendo un solo recurso r para simplificar, la **regla de correctitud parcial (RPAR)** plantea:
  - Dadas *proof outlines* $\{p_1\}\ S_1^*\ \{q_1\}$, ..., $_1\ \{p_n\}\ S_n^*\ \{q_n\}$ con predicados de las variables **no compartidas**,
  - y un **invariante I_r del recurso r** con las **variables compartidas**,
  - se cumple:

$$\{I_r \wedge p_1 \wedge \ldots \wedge p_n\}\ [S_1 \,||\, \ldots \,||\, S_n]\ \{I_r \wedge q_1 \wedge \ldots \wedge q_n\}$$

- Es decir:

$$\{I_r \wedge p_1 \wedge \ldots \wedge p_n\}$$

S_1	S_2		S_n
$\{p_1\}$	$\{p_2\}$		$\{p_n\}$
.....	.....		.....
with … endwith	with … endwith		with … endwith
$\{p_{1i1}\}$	$\{p_{2j1}\}$		$\{p_{nk1}\}$
.....	.....		.....
with … endwith	with … endwith	.....	with … endwith
$\{p_{1i2}\}$	$\{p_{2j2}\}$		$\{p_{nk2}\}$
.....	.....		.....
$\{q_1\}$	$\{q_2\}$		$\{q_n\}$

$$\{I_r \wedge q_1 \wedge \ldots \wedge q_n\}$$

De esta manera, no hay que chequear que las *proof outlines* sean libres de interferencia.

Los predicados de las proof outlines arrastran la información de las **variables disjuntas**, y el invariante del recurso arrastra la información de las **variables compartidas**.

Modelo 3. Lenguaje de pasajes de mensajes

- Las composiciones concurrentes tienen **procesos etiquetados**:

$$[P_1 :: S_1 \;||\; ... \;||\; P_n :: S_n]$$

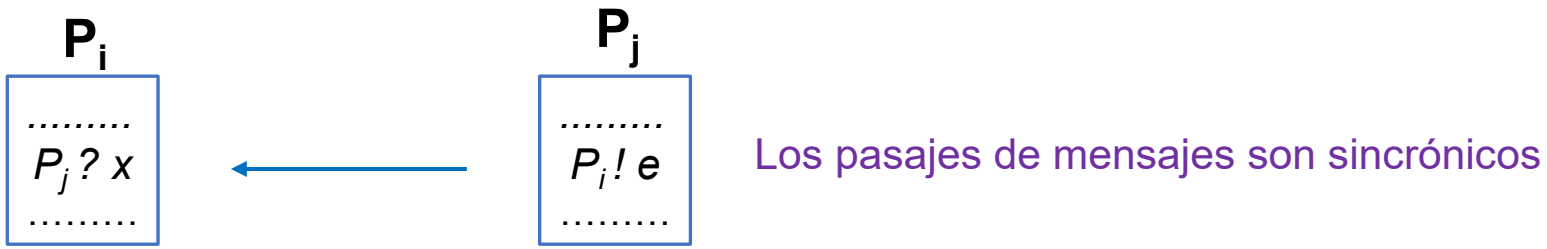
- Los procesos **no comparten variables** y se envían **mensajes** con **instrucciones de comunicación**:

- ***Instrucción de entrada P_j ? x***

  En un proceso P_i con i ≠ j, que tiene una variable local x, la instrucción efectúa un pedido al proceso P_j para que asigne un valor a x. P_i queda **bloqueado** mientras P_j no responda.

- ***Instrucción de salida P_i ! e***

  En un proceso P_j con j ≠ i, la instrucción efectúa un pedido al proceso P_i para que reciba el valor de la expresión *e*, definida con variables locales de P_j. P_j queda **bloqueado** mientras P_i no responda.



- Las instrucciones de comunicación pueden usarse aisladamente o en **condicionales** y **repeticiones**. P.ej.:

P_i :: **if** x > 0 ; P_j ? y → y := y + 1 **or** x < 0 ; P_j ! 0 → skip **fi**

P_j :: **do** x ≥ 0 ; P_i ! x → x := x – 1 **od**

Idea general de las pruebas en este modelo

Método de Apt, Francez y de Roever

- La novedad en este modelo es que en las pruebas locales hay que utilizar **asunciones**:

Axioma de la instrucción de entrada **(IN)**

{p} Pi ? x {q}

Axioma de la instrucción de salida **(OUT)**

{p} Pj ! e {p}

- La segunda etapa consiste en un **test de cooperación** en el que se validan **todas** las asunciones de las pruebas locales. Para ello se utiliza el siguiente axioma:

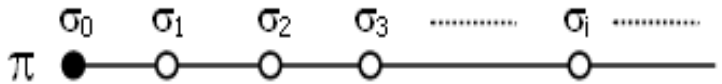
Axioma de comunicación **(COM)**

{p[x|e]} Pi ? x || Pj ! e {p}

- Otra técnica habitual para evitar pruebas de comunicaciones inexistentes, es el uso de **invariantes globales** con variables locales, que son contadores de la cantidad de comunicaciones efectuadas.

Verificación de programas concurrentes utilizando lógica temporal

- La lógica temporal permite especificar propiedades a lo largo de **computaciones.**



- En esencia, extiende la lógica de predicados con operadores temporales. Ejemplo de fórmulas:

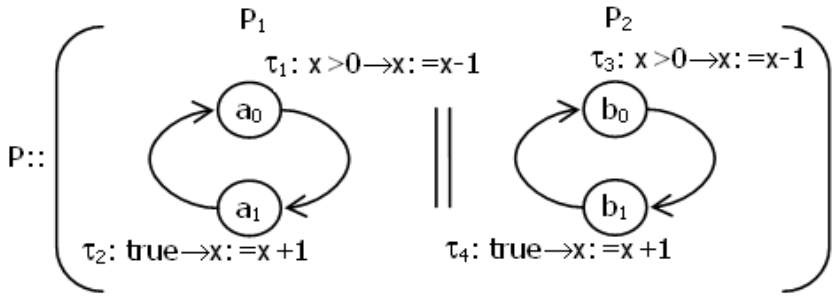
1. $\sigma_0 \models Xp$ significa que $\sigma_1 \models p$
2. $\sigma_0 \models Gp$ significa que para todo $i \geq 0$ se cumple $\sigma_i \models p$
3. $\sigma_0 \models Fp$ significa que para algún $i \geq 0$ se cumple $\sigma_i \models p$
4. $\sigma_0 \models p\ U\ q$ significa que para algún $j \geq 0$ se cumple $\sigma_j \models q$ y para todo $0 \leq i \leq j - 1$ se cumple $\sigma_i \models p$

- Hay dos familias de lenguajes de lógica temporal, **LTL** (lógica lineal), definidos sobre computaciones (como las fórmulas presentadas más arriba), y **CTL** (lógica computacional o arbórea), definidos sobre **árboles de computaciones**, lo que permite especificar computaciones específicas. Ejemplo de fórmulas CTL:

1. $\sigma_0 \models AGp$ significa que sobre toda computación desde σ_0 se cumple Gp
2. $\sigma_0 \models EFp$ significa que sobre alguna computación desde σ_0 se cumple Fp

Ninguna familia es más expresiva que la otra. Hay controversias sobre la conveniencia del uso de una sobre la otra.

- Cuando un programa se puede modelizar con un **diagrama de transición de estados**, p.ej.:



Modelización de la solución al problema del acceso a una sección crítica por medio de un semáforo.

alcanza con la **lógica temporal proposicional**. Las propiedades se especifican componiendo proposiciones atómicas, y la verificación se puede llevar a cabo automáticamente **(*model checking*).** Casos típicos de programas considerados son los protocolos de comunicación y los circuitos digitales.

- **Model checking con lógica LTL.** Para verificar una fórmula F en un modelo M:

  1. Se construye un autómata A para aceptar todas las valuaciones que satisfacen ¬F.
  2. Se combina A con M, produciendo un diagrama D con caminos de A y de M.
  3. El model checker acepta sii no existe ningún camino desde el estado inicial en el diagrama combinado D (si existiese, habría una computación en M que no satisface la propiedad F).

- **Model checking con lógica CTL.** Para verificar una fórmula F en M, se etiquetan los estados de M que satisfacen F:

  1. Se etiquetan los estados que satisfacen las subfórmulas más chicas de F.
  2. El proceso prosigue iterativamente considerando subfórmulas cada vez más grandes, hasta llegar a F.
  3. Al final se detecta si el estado inicial satisface o no F.

- Se considera que LTL es **más fácil de usar.** Como contrapartida, el model checking basado en CTL es **más eficiente**.

Metodología UNITY

- **UNITY** es una metodología de especificación, desarrollo y verificación de programas concurrentes, utilizando una variante de la lógica temporal (**Método de Chandy y Mizra**).
- Dos estrategias de desarrollo, **unión** (composición) y **superposición** (refinamiento).
- La unión de dos procesos S1 y S2 consiste en su **concatenación** [S1 || S2], y la superposición establece una **programación por capas** (se agregan variables y asignaciones que no alteran los cómputos inferiores).
- Las especificaciones cuentan con **operadores temporales** como:

  1. **p *unless* q**: si se cumple p entonces q no se cumple nunca y p se cumple siempre, o q se cumple a futuro y p se cumple mientras q no se cumpla.
  2. **p *ensures* q**: si se cumple p entonces q se cumple a futuro y p se cumple mientras q no se cumpla.
  3. **p *leads-to* q**: si se cumple p entonces q se cumple a futuro.

- Se demuestra, dada la unión [S1 || S2], que:

  a. Si se cumple **p *unless* q en S1 y S2**, se cumple **p *unless* q en [S1 || S2].**
  b. Si se cumple **p *ensures* q en S1** y **p *unless* q en S2**, se cumple **p *ensures* q en [S1 || S2].**
  c. Si se cumple ***p leads-to* q en S1 y S2**, no tiene por qué cumplirse ***p leads-to* q en [S1 || S2].**

- Por su parte, la superposición facilita la verificación, pero como contrapartida, exige conocer en detalle las capas previamente desarrolladas, al tiempo que su tratamiento algebraico es limitado.

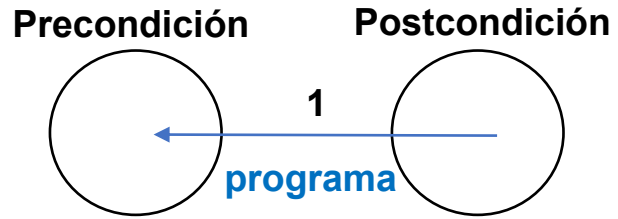
Clase 13. Misceláneas. Extensiones de la lógica de Hoare.

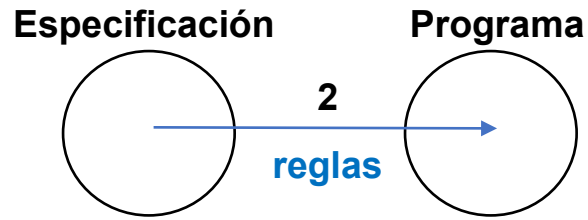
EXTENSIONES

- **Procedimientos.** Distintos tipos de pasajes de parámetros y recursión.
- **Datos.** Estructuras de datos, variables locales, punteros.
- **Programas concurrentes.** Programas paralelos (memoria compartida) y distribuidos (pasajes de mensajes).
- **Programas probabilísticos y cuánticos.**

APLICACIONES (CÁLCULO DE PROGRAMAS)

1. **Precondición más débil** (conjunto de estados iniciales más amplio posible)
2. **Síntesis de programas** (transformaciones a partir de la especificación)
3. Otros métodos de cálculo de programas





Clase 13. Misceláneas. Programar y verificar en simultáneo.

- Lo correcto es **programar al tiempo que verificar**, para obtener un programa **correcto por construcción.**
- El método de prueba definido es un buen soporte para dicha práctica.
- Por ejemplo, supongamos que se quiere construir un programa con la siguiente estructura:

  T ; while B do S od

  que debe ser correcto con respecto a una especificación (r, q), o sea que debe satisfacer la fórmula:

  {r} T ; while B do S od {q}

- Hay que construir el fragmento inicial T y el *while*. De acuerdo al método, se tiene que encontrar un predicado p (**invariante**) y una función t (**variante**) para el *while*, y se deben cumplir **cinco condiciones**:

1. A partir de la precondición r, el fragmento T termina estableciendo el predicado p: **{r} T {p}**
2. El predicado p es un invariante del *while*: **{p ∧ B} S {p}**
3. La función t decrece después de cada iteración del *while*: **{p ∧ B ∧ t = Z} S {t < Z}**
4. El predicado p asegura que la función t siempre es positiva: **p → t ≥ 0**

   (cumplidos (2), (3) y (4), se obtiene **{p} while B do S od {p ∧ ¬B}**)
5. El *while* termina estableciendo la postcondición q: **(p ∧ ¬B) → q**

{r}
T ;
{p}
while B do
{p ∧ B}
S
{p}
od
{p ∧ ¬B}
{q}

Clase 13. Misceláneas. Semántica funcional (o denotacional).

Expresiones enteras

$$\mathbf{e :: n \mid x \mid (e_1 + e_2) \mid (e_1 - e_2) \mid (e_1 . e_2) \mid \ldots}$$

n es una constante entera. x es una variable entera. P.ej: **e = (5 + (x – 1))**

- Se utiliza una **función semántica V**, que asigna a una **expresión entera e** y un **estado σ** un **número entero** n

$$\mathbf{V : Iexp \longrightarrow (\Sigma \longrightarrow Z)}$$

Iexp es el conjunto de las expresiones enteras

- Definición inductiva

1. $V(n)(\sigma) = n$, tal que n es la representación del número entero n (p.ej., el símbolo 1 representa el número 1)
2. $V(x)(\sigma) = \sigma(x)$
3. $V((e_1 + e_2))(\sigma) = V(e_1)(\sigma) + V(e_2)(\sigma)$
4. $V((e_1 - e_2))(\sigma) = V(e_1)(\sigma) - V(e_2)(\sigma)$

Etc.

Expresiones booleanas

$$\mathbf{B :: true \mid false \mid (e_1 = e_2) \mid (e_1 < e_2) \mid \ldots \mid \neg B_1 \mid (B_1 \vee B_2) \mid (B_1 \wedge B_2) \mid \ldots}$$

true y false son las constantes booleanas. Las e_i son expresiones enteras. P.ej: **$B = ((x = 1) \vee (x = 2))$**

- Se utiliza una **función semántica W**, que asigna a una **expresión booleana B** y un **estado σ** un **valor V o F**

$$\mathbf{W : Bexp \longrightarrow (\Sigma \longrightarrow \{V, F\})}$$

Bexp es el conjunto de las expresiones booleanas

- Definición inductiva

1. $W(true)(\sigma) = V$ y $W(false)(\sigma) = F$ (p.ej., la constante true representa el valor de verdad V)
2. $W((e_1 = e_2))(\sigma) = (V(e_1)(\sigma) = V(e_2)(\sigma))$
3. $W((e_1 < e_2))(\sigma) = (V(e_1)(\sigma) < V(e_2)(\sigma))$
4. $W(\neg B_1)(\sigma) = \neg W(B_1)(\sigma)$
5. $W((B_1 \vee B_2))(\sigma) = W(B_1)(\sigma) \vee W(B_2)(\sigma)$

Etc.

Predicados

$$\mathbf{p :: true \mid false \mid (e_1 = e_2) \mid (e_1 < e_2) \mid \ldots \mid \neg p \mid (p_1 \vee p_2) \mid \ldots \mid \exists x{:}\ p \mid \forall x{:}\ p}$$

true y false son las constantes. x es una variable entera. Las e_i son expresiones enteras. P.ej: $\exists x{:}\ \neg(x = 1)$

- Se utiliza una **función semántica T**, que asigna a un **predicado p** y un **estado σ** un **valor V o F**

$$\mathbf{T : Pred \longrightarrow (\Sigma \longrightarrow \{V, F\})}$$

Pred es el conjunto de los predicados

- <u>Definición inductiva</u>

1. $T(true)(\sigma) = V$ y $T(false)(\sigma) = F$ (p.ej., la constante true representa el valor de verdad V)
2. $T((e_1 = e_2))(\sigma) = (T(e_1)(\sigma) = T(e_2)(\sigma))$. Y lo mismo con los otros operadores (<, >, etc.)
3. $T(\neg p_1)(\sigma) = \neg T(p_1)(\sigma)$. Y lo mismo con los otros operadores ($\wedge$, $\vee$, etc.)
4. $T(\exists x{:}\ p_1)(\sigma) = V$ sii existe algún número entero n tal que $T(p_1)(\sigma[x|n]) = V$
5. $T(\forall x{:}\ p_1)(\sigma) = V$ sii para todo número entero n se cumple $T(p_1)(\sigma[x|n]) = V$

Instrucciones

- Se utiliza una función semántica M, que asigna a un programa S y un estado σ otro estado σ.

$$\mathbf{M : L_w \longrightarrow (\Sigma \longrightarrow \Sigma)}$$

L_w es el lenguaje de programación

- Definición inductiva

1. $M(skip)(\sigma) = \sigma$
2. $M(x := e)(\sigma) = \sigma[x|V(e)(\sigma)]$
3. $M(S_1 ; S_2)(\sigma) = M(S_2)(M(S_1)(\sigma))$
4. $M(\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi})(\sigma) = M(S_1)(\sigma)$, si $W(B)(\sigma) = V$
   $= M(S_2)(\sigma)$, si $W(B)(\sigma) = F$
5. $M(\text{while } B \text{ do } S \text{ od})(\sigma) = M(\text{while } B \text{ do } S \text{ od})(M(S)(\sigma))$, si $W(B)(\sigma) = V$
   $= \sigma$, si $W(B)(\sigma) = F$

   con $M(S)(\sigma) = \perp$ si S diverge, y $M(S)(\perp) = \perp$ para todo S

- **La especificación así planteada está incompleta**, falta encontrar la solución de **M(while B do S od)(σ)**

  - Una manera es mediante el *límite* de una secuencia de funciones conocidas como *aproximaciones*, que representan las sucesivas iteraciones del *while*.
  - Para asegurar su *existencia* y *unicidad* se recurre a la **teoría de los *órdenes parciales completos***.

Semántica funcional (o denotacional) del while

- Resolución de la ecuación **M(while B do S od)(σ) = M(while B do S od)(M(S)(σ)), si W(B)(σ) = V**
  = σ, si W(B)(σ) = F

- Se plantea la siguiente secuencia de funciones φ_i:

  1. $\varphi_0(\sigma) = \perp$
  2. $\varphi_{i+1}(\sigma) = \varphi_i(M(S)(\sigma))$, si $W(B)(\sigma) = V$
     σ, si $W(B)(\sigma) = F$

  y se calcula el **límite φ de la cadena $\{\varphi_i\}_{i \geq 0}$**

- Por ejemplo, dado el while: **S :: while x > 0 do x := x – 1 od** y el estado inicial **σ[x|2]**:

  $\varphi_0(\sigma[x|2]) = \perp$

  $\varphi_1(\sigma[x|2]) = \varphi_0(\sigma[x|1]) = \perp$

  $\varphi_2(\sigma[x|2]) = \varphi_1(\sigma[x|1]) = \varphi_0(\sigma[x|0]) = \perp$

  $\varphi_3(\sigma[x|2]) = \varphi_2(\sigma[x|1]) = \varphi_1(\sigma[x|0]) = \sigma[x|0]$

  $\varphi_i(\sigma[x|2]) = \sigma[x|0]$, con i = 4, 5, 6, …, es decir que el límite (estado final) es **σ[x|0]**.

Clase 13. Misceláneas. Verificación con arreglos y punteros.

- El axioma de asignación (ASI) presentado antes no es verdadero cuando se agregan **arreglos** al lenguaje:

  1) {true} a[i] := 1 {a[i] = 1}

     Si **i = "a[2]", a = [2, 2, …],** luego de a[i] := 1 vale a[i] = 2.

     Motivo: en la asignación, a[i] se refiere a a[2], pero en la postcondición, a[i] se refiere a a[1].

  2) {0 + 1 = a[y]} a[x] := 0 {a[x] + 1 = a[y]}

     Si **a = [1, …], x = 1, y = 1,** luego de a[x] := 0 vale a[x] = 0, a[y] = 0.

     Motivo: a[x] y a[y] denotan un mismo elemento, son *alias*.

  Remediación: evitar variables con índices de la forma a[i], y alias, pero esto es muy restrictivo. Otra posibilidad es modificar el mecanismo de sustitución sintáctica, que es la base de la semántica de :=.

- Otro tipo de dato problemático en el marco de la axiomática definida es el **puntero**. En este caso un enfoque muy difundido es la **lógica de separación**.

Aún con variables enteras simples deben tomarse recaudos. La aritmética de las computadoras no es la de la matemática. P.ej., en caso de *overflow*, ¿se cancela el programa?, ¿se devuelve el máximo entero?, ¿se usa aritmética modular? **La axiomática debe contemplar la implementación adoptada**.

Clase 13. Misceláneas. Sensatez total y aritmética.

- La sensatez de H se cumple **independientemente de la interpretación semántica considerada**. Notar que en ningún momento de las pruebas recurrimos al dominio de las variables de programa (en nuestro caso los números enteros), salvo cuando utilizamos sus axiomas (que podrían ser de otro dominio semántico). Por ejemplo, en el programa de división presentado antes, **la prueba también aplica si las variables son de tipo real.**

- En efecto, H tiene **sensatez total**, lo que se formula de la siguiente manera:

$$\mathbf{Tr_I \mid\!-_H \{p\}\ S\ \{q\} \rightarrow \mid\!=_I \{p\}\ S\ \{q\},\ para\ toda\ interpretación\ I}$$

Tr_I tiene todos los axiomas correspondientes a la interpretación I (números enteros, números reales, etc).

Que valga $\mid\!=_I \{p\}\ S\ \{q\}$ significa que $\{p\}\ S\ \{q\}$ es verdadera considerando I.

- En cambio, la sensatez de H* se conoce como **aritmética**. Tal como está planteada la regla REP*:
  - El dominio semántico en el que se interpretan los programas **debe extenderse con el de los números naturales**, en el que está definido el variante t (en la hoja siguiente mostramos un ejemplo con números reales).
  - Correspondientemente, el lenguaje de especificación **debe extenderse con el de la aritmética**.