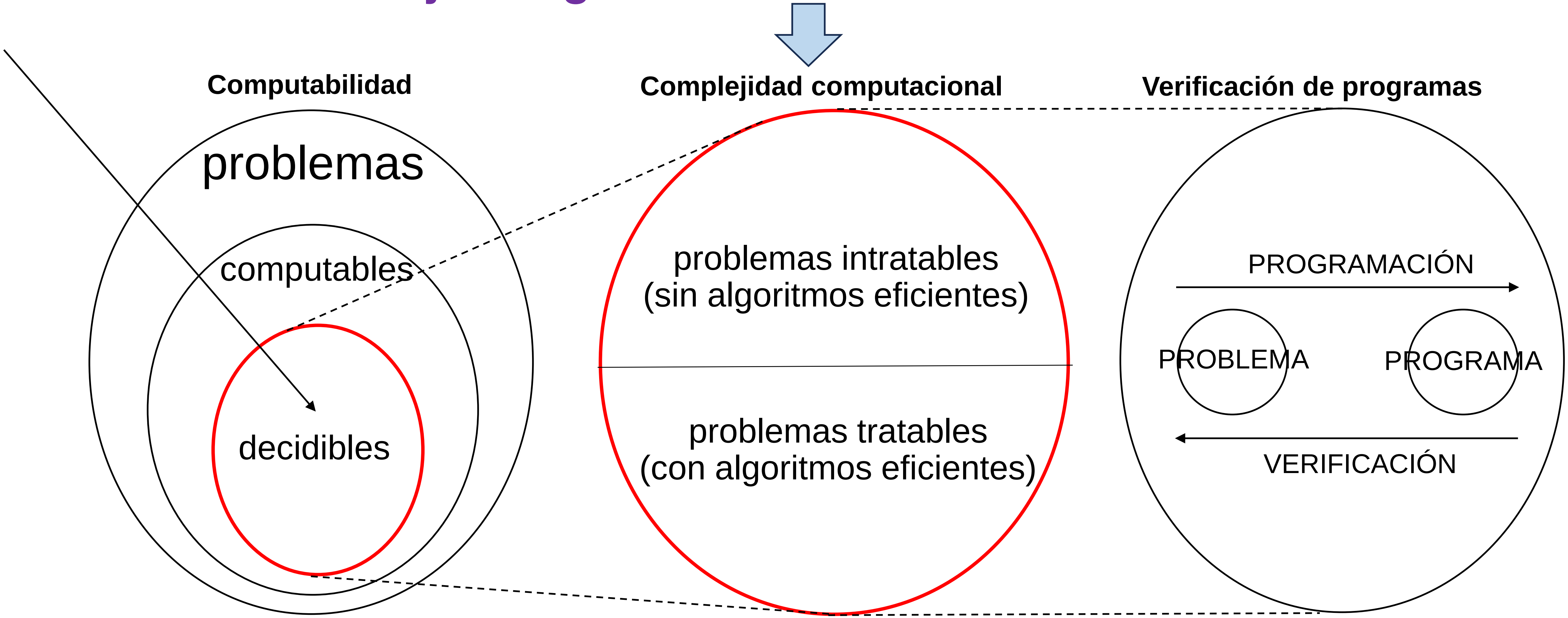
Clase teórica 5

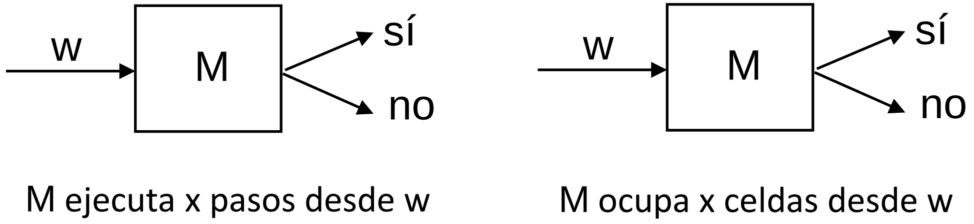
Tiempo polinomial y no polinomial

Continuamos el viaje imaginario

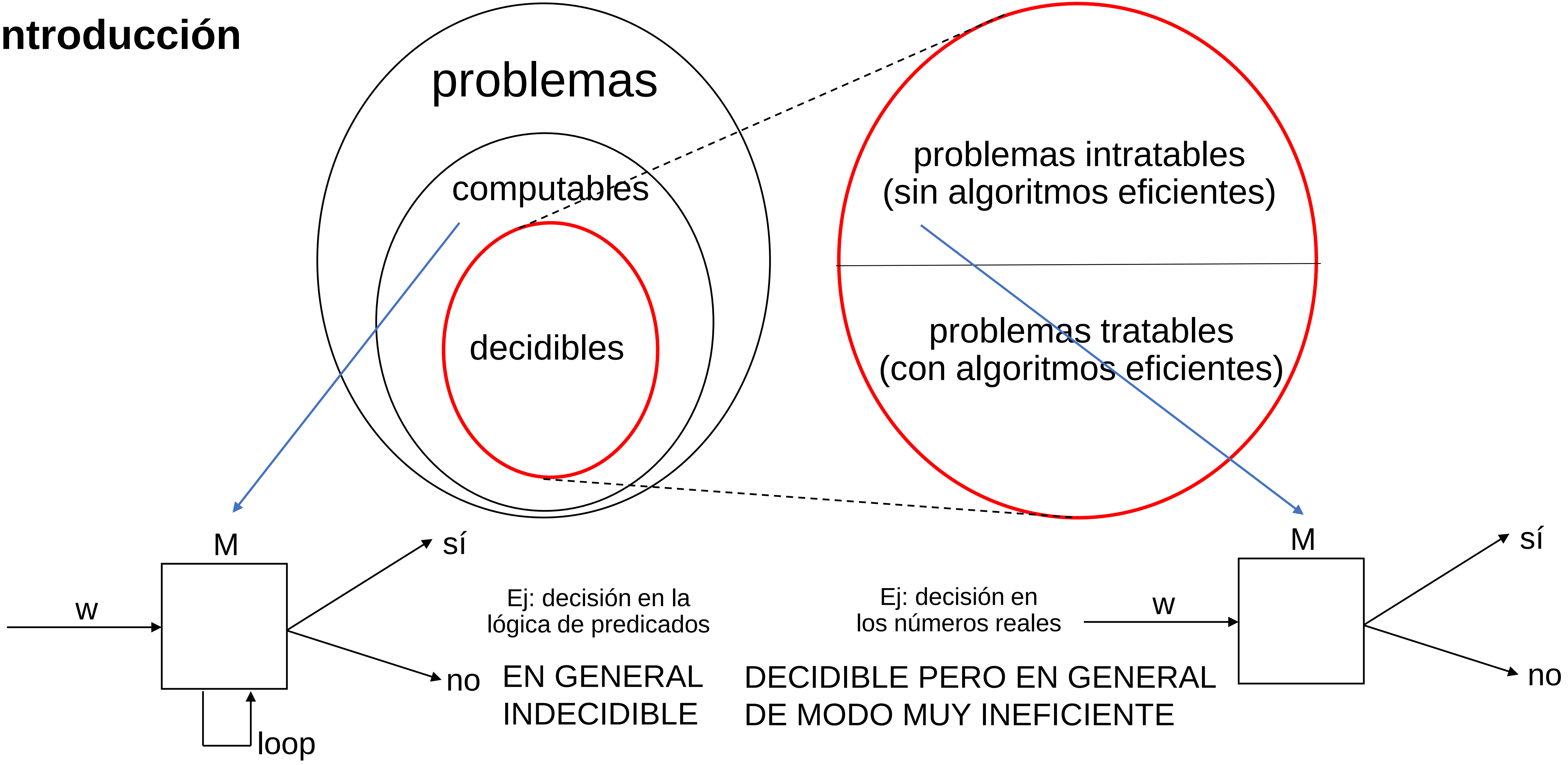


- Métricas de complejidad computacional más utilizadas:

Tiempo = cantidad de pasos ejecutados por una MT.
Espacio = cantidad de celdas ocupadas por una MT.

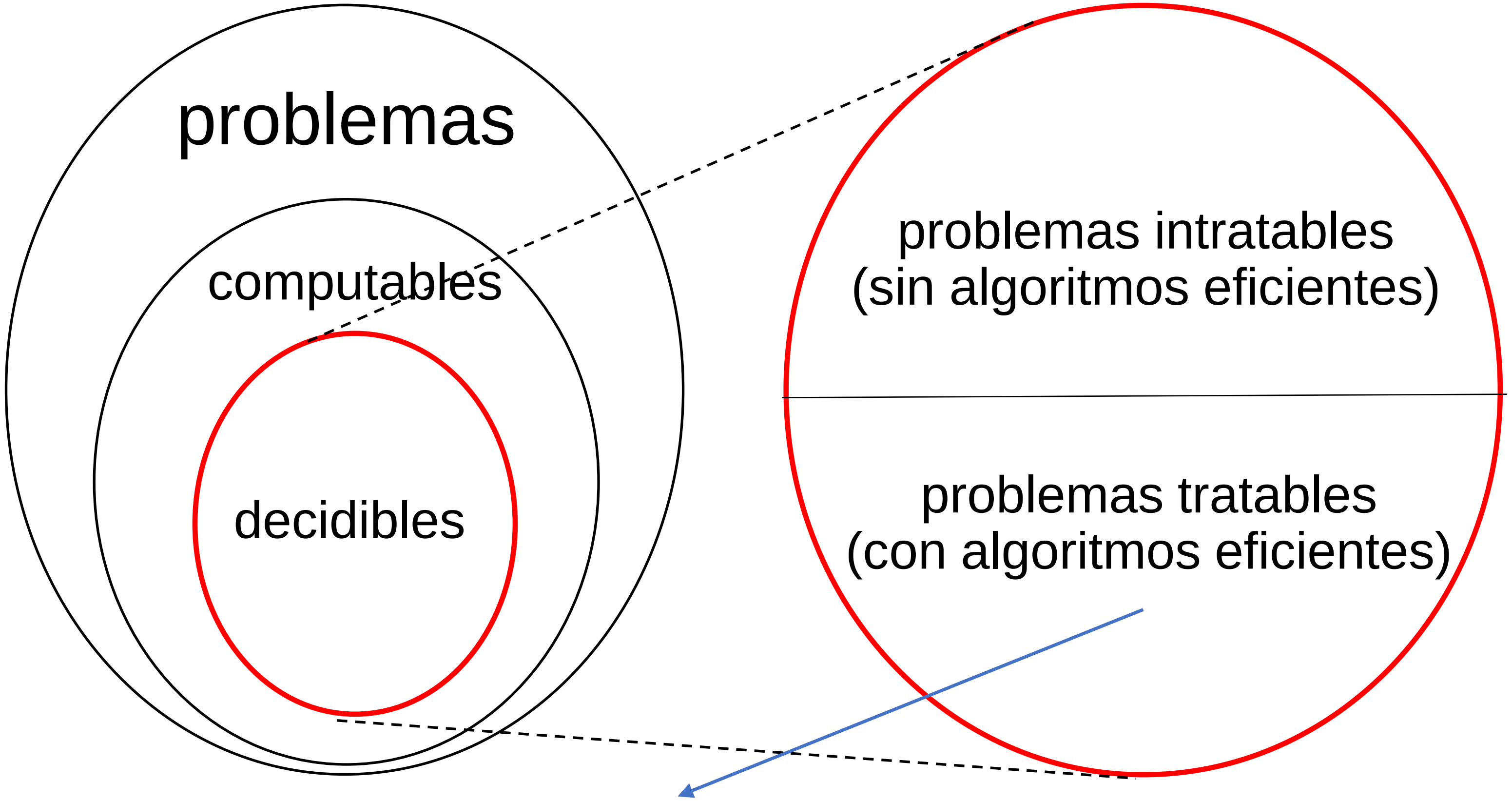


Introducción



en la práctica “se igualan”

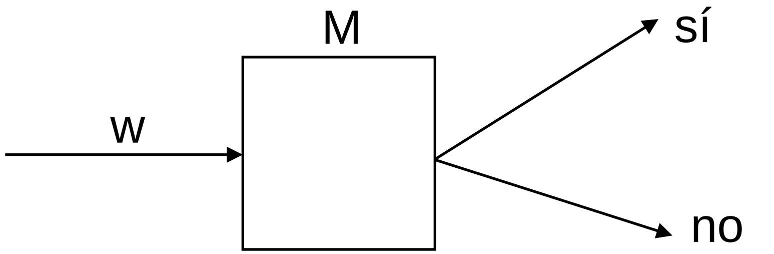
Introducción



operaciones aritméticas

búsqueda de un camino en un grafo

ordenamiento de un arreglo



MT con varias cintas de <u>tiempo polinomial</u>

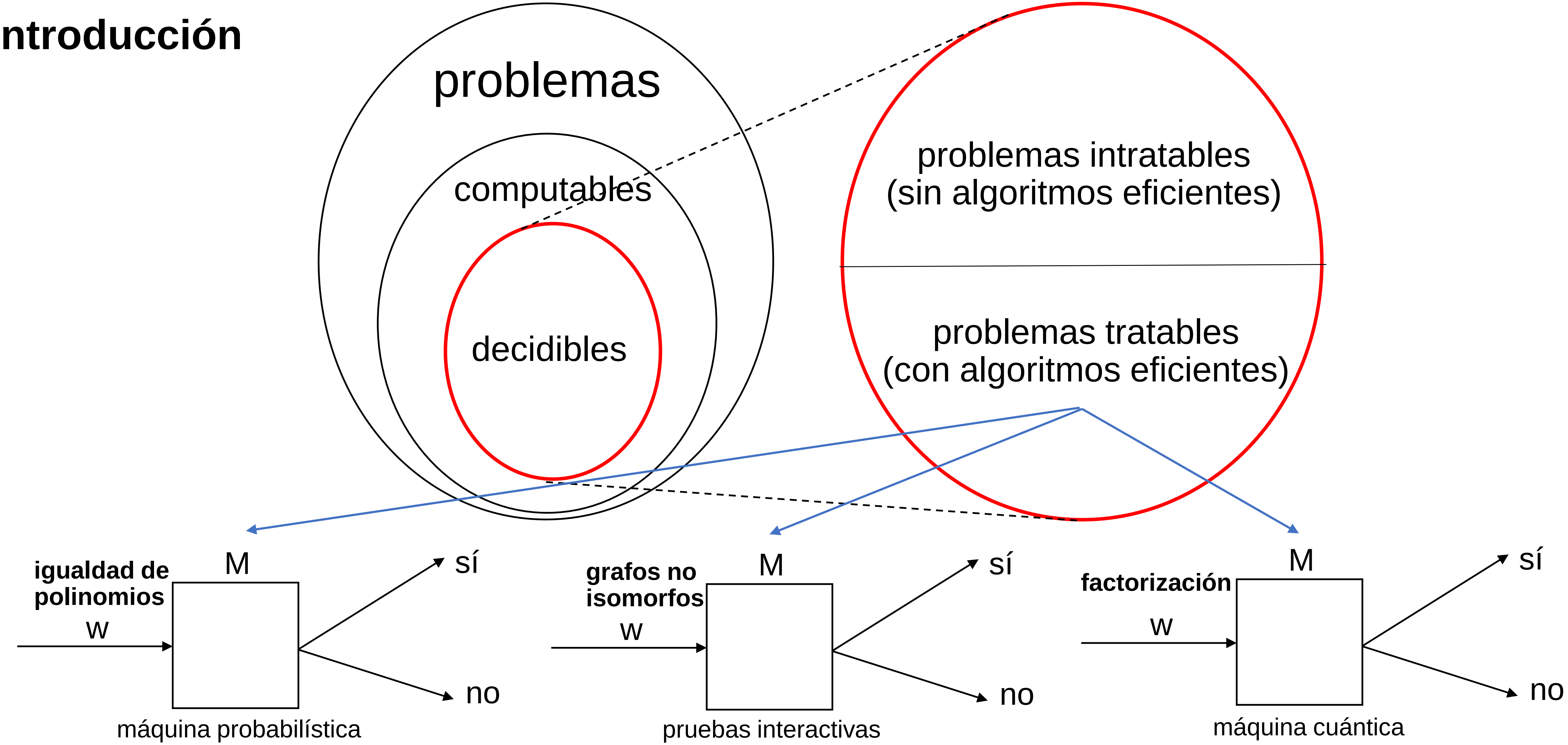
máximo común divisor

búsqueda del máximo flujo en una red

multiplicación de matrices

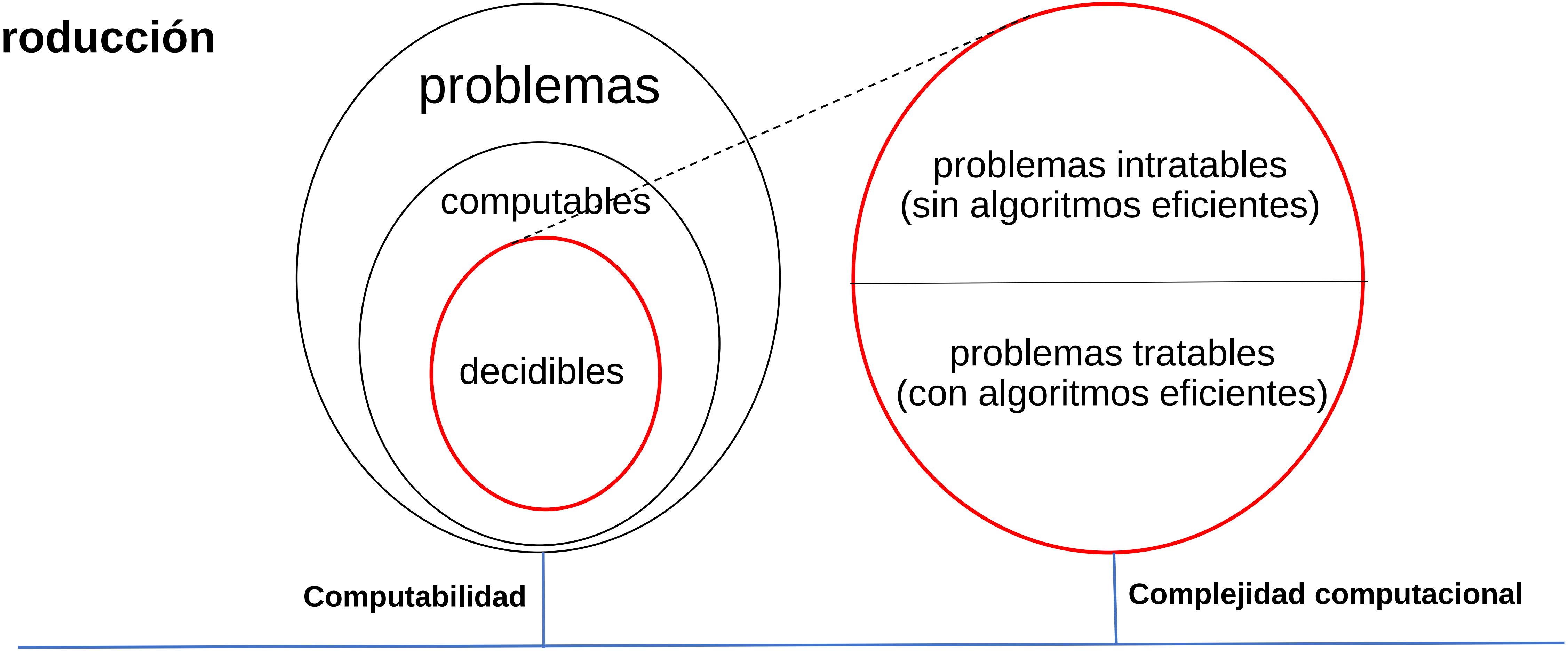
convención aceptada universalmente

Introducción



adicionalmente: análisis en otros modelos computacionales

Introducción



En ambos casos:

- **Análisis estructural**: enfocado no en problemas particulares sino en clases de problemas.
- **Perspectiva:** entender por qué algunos problemas son más difíciles que otros.
- **Escenario de estudio**: MT con varias cintas como modelo computacional, y pruebas por medio de la construcción de MT, la diagonalización y las reducciones.

Complejidad temporal

- Una MT M tarda más (hace más pasos) a medida que sus cadenas de entrada w son más grandes.

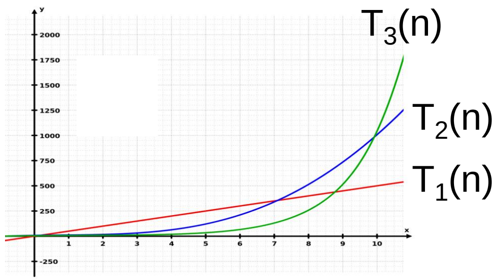
- Por eso el tiempo de M no se mide en términos absolutos sino con **funciones temporales T(n)**, que se definen en términos del tamaño |w| de w (se usa n = |w|).

- <u>Ejemplos de funciones temporales T(n)</u>:

  $5n + 8$, $3n^2$, $3n^3 + n^2 + 25n$, etc. **polinomiales o poly(n) *** exponentes constantes**
  $20^{\log_2 n}$, $12^n + 10n + 5$, 6^{n^5}, etc. **exponenciales o exp(n) *** exponentes polinomiales en n**
  Otras: 2 elevado a la 2^n (**doble exponencial**), 2 elevado a la 2 elevado a la 2^n (**triple exponencial**), etc.

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

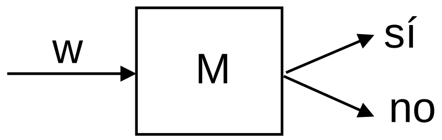
  Por ejemplo (ejercicio):
  $5n^3 + 8n + 25 = O(n^3)$
  $n^2 = O(n^3)$
  $n^3 = O(2^n)$.



$T_i(n) = O(T_j(n))$, con $i < j$
Cuando n tiende a infinito,
$T_i(n)$ se mantiene por debajo de $T_j(n)$

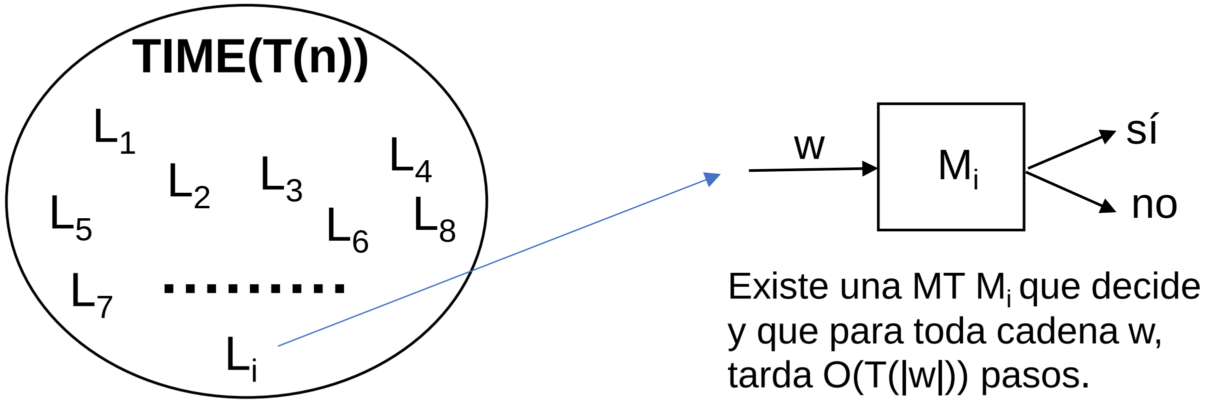
Usar funciones O(T(n)) en lugar de funciones T(n) permite manejar un **nivel de abstracción adecuado** (tiempo lineal, cuadrático, polinomial, exponencial, doble exponencial, etc).

- Una MT M **tarda 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

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



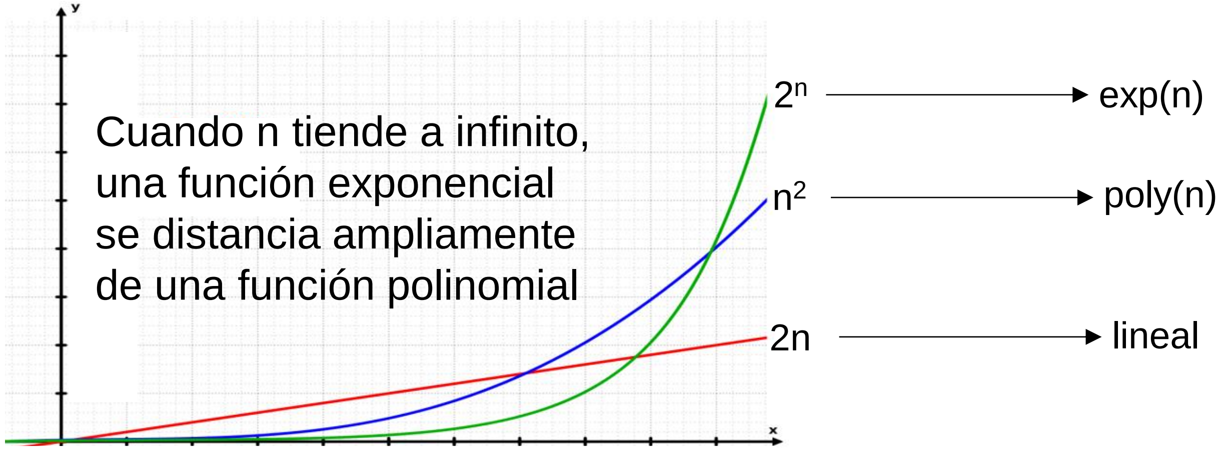
Existe una MT M_i que decide L_i, y que para toda cadena w, tarda O(T(|w|)) pasos.

- Se considera el **tiempo máximo** (procesar cualquier cadena w consume **a lo sumo** T(|w|) pasos).

  Otros criterios son el **tiempo promedio** y el **tiempo mínimo,** en general muy difíciles de calcular (a pesar de buenos avances, al día de hoy se conocen aún pocos valores de este tipo).

- 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



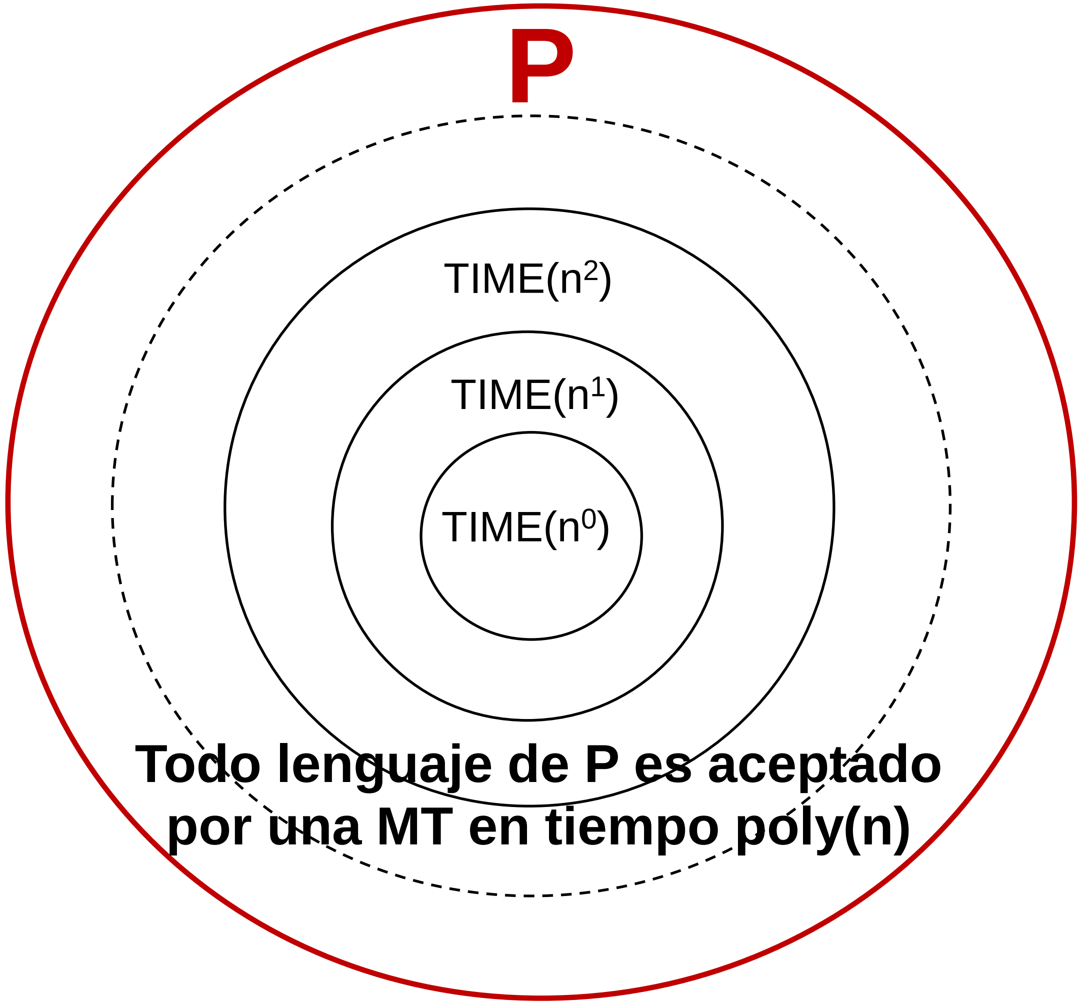
¿Es tratable n^{100}?
¿Es intratable $1{,}001^n$?
No, pero estos valores no se dan en la práctica

- Ejemplo, considerando una computadora determinada. Notar el salto abrupto en las funciones exponenciales:

TAMAÑO

	10	20	30	40	50	60
n	0,00001 segundos	0,00002 segundos	0,00003 segundos	0,00004 segundos	0,00005 segundos	0,00006 segundos
n^2	0,0001 segundos	0,0004 segundos	0,0009 segundos	0,0016 segundos	0,0025 segundos	0,0036 segundos
n^3	0,001 segundos	0,008 segundos	0,027 segundos	0,064 segundos	0,125 segundos	0,216 segundos
n^5	0,1 segundos	3,2 segundos	24,3 segundos	1,7 minutos	5,2 minutos	13,0 minutos
2^n	0,001 segundos	1,0 segundos	17,9 minutos	12,7 días	35,7 años	366 siglos
3^n	0,059 segundos	58,0 minutos	6,5 años	3855 siglos	2.10^8 siglos	$1{,}3.10^{13}$ siglos

La clase P



Todo lenguaje de P es aceptado por una MT en tiempo poly(n)

Robustez: si una MT M_1 con K_1 cintas tarda tiempo poly(n), existe una MT M_2 equivalente con K_2 cintas que también tarda tiempo poly(n).

- **Ejemplo de lenguaje en la clase P**

L = {w | w es un palíndromo (capicúa) con símbolos *a* y b}

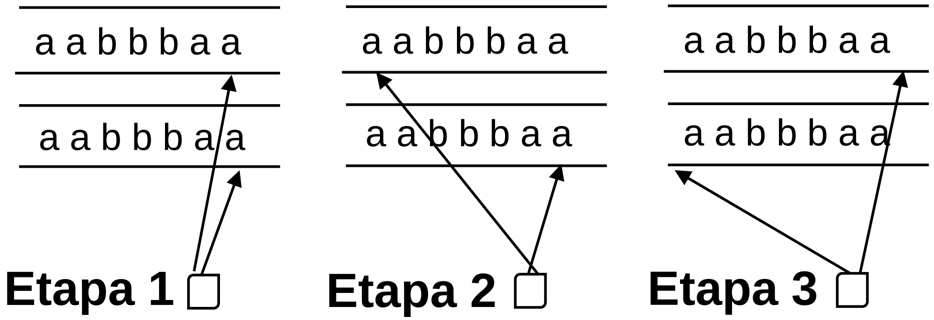
Una MT muy simple que decide L, con 2 cintas, hace:

1. Copia w de la cinta 1 a la cinta 2
   Tiempo O(n) ¿por qué?

2. Posiciona el cabezal de la cinta 1 a la izquierda
   Tiempo O(n) ¿por qué?

3. Compara en direcciones contrarias uno a uno los símbolos de las 2 cintas hasta llegar a un blanco en ámbas
   Tiempo O(n) ¿por qué?

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

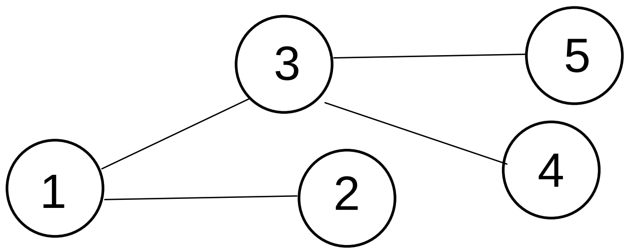


- **Otro ejemplo de lenguaje en la clase P**

ACC = {G | G es un grafo con m vértices y tiene un camino del vértice 1 al vértice m}

Nota: salvo mención explícita, trabajaremos con grafos ***no dirigidos***

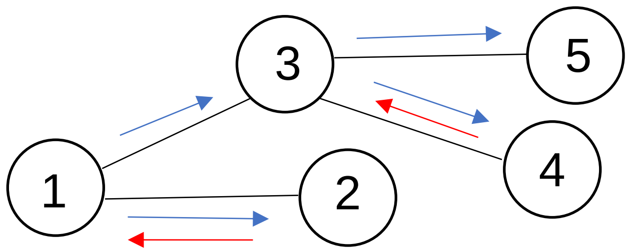
Por ejemplo, sea el siguiente grafo:



G = (V, E), siendo 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)})

Idea general de una MT M que decide el lenguaje ACCES en tiempo poly(n):

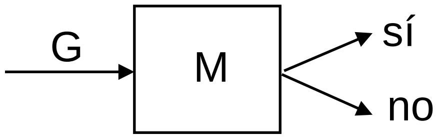


Método DFS (búsqueda en profundidad):

En el peor caso, M recorre cada arco 2 veces

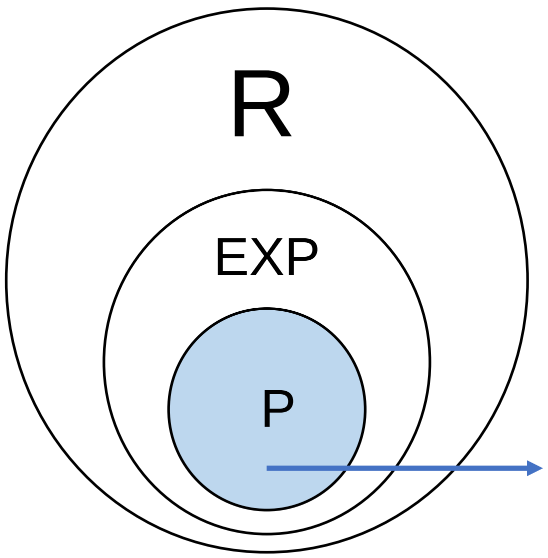
T(n) = O(|E|) = O(|G|) = O(n) pasos

Por lo tanto, ACCES ∈ P



M tarda tiempo O(|G|)

Primera versión de la jerarquía temporal



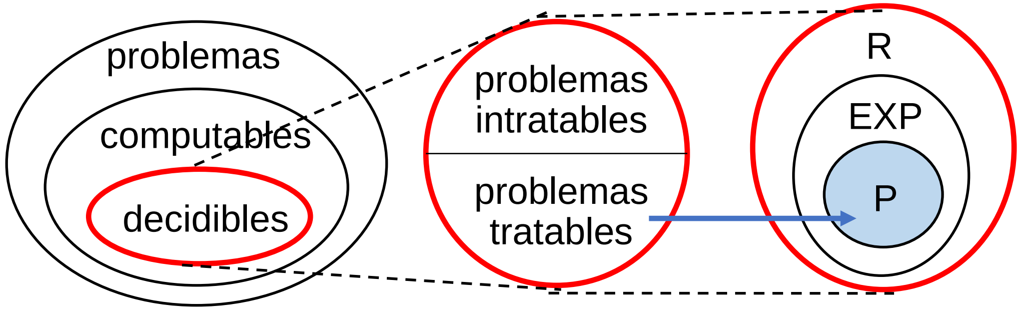
R es la clase de los lenguajes recursivos.

P es la clase de los lenguajes decidibles en tiempo poly(n), es decir $O(n^k)$.

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

Se prueba que **P** $\subset$ **EXP** $\subset$ **R.**

LENGUAJES TRATABLES



Tesis fuerte de Church-Turing

Si L es decidible en tiempo poly(n) por un modelo computacional **físicamente realizable,** también es decidible en tiempo poly(n) por una MT **(al menos hasta que las máquinas cuánticas sean una realidad).**

Codificación de las cadenas

Todo símbolo se codifica con un número. Se utiliza cualquier codificación **distinta de la unaria** **(codificar números grandes en unario no es físicamente realizable, y además genera inconsistencias).** Para uniformar usaremos siempre la **codificación binaria**: 1, 10, 11, 100, 101, 110, 111, 1000, 1001,1010, 1011, etc.

El tamaño n de un número N en binario es **$O(\log_2 N)$.**
Por ej., N = 29 en binario es 11101 (n = 5 y $\log_2 N$ = 4,…).

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

SAT = {φ | φ es una fórmula booleana satisfactible con m variables}

*Nota: salvo mención explícita, trabajaremos con fórmulas booleanas **sin cuantificadores***

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 $\mathcal{A} = (V, V, V)$

$\varphi_2 = (x_1 \wedge x_2 \wedge x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$ 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 posibles evaluaciones (en este caso 8)
(por cada variable se prueba con los valores V y F)

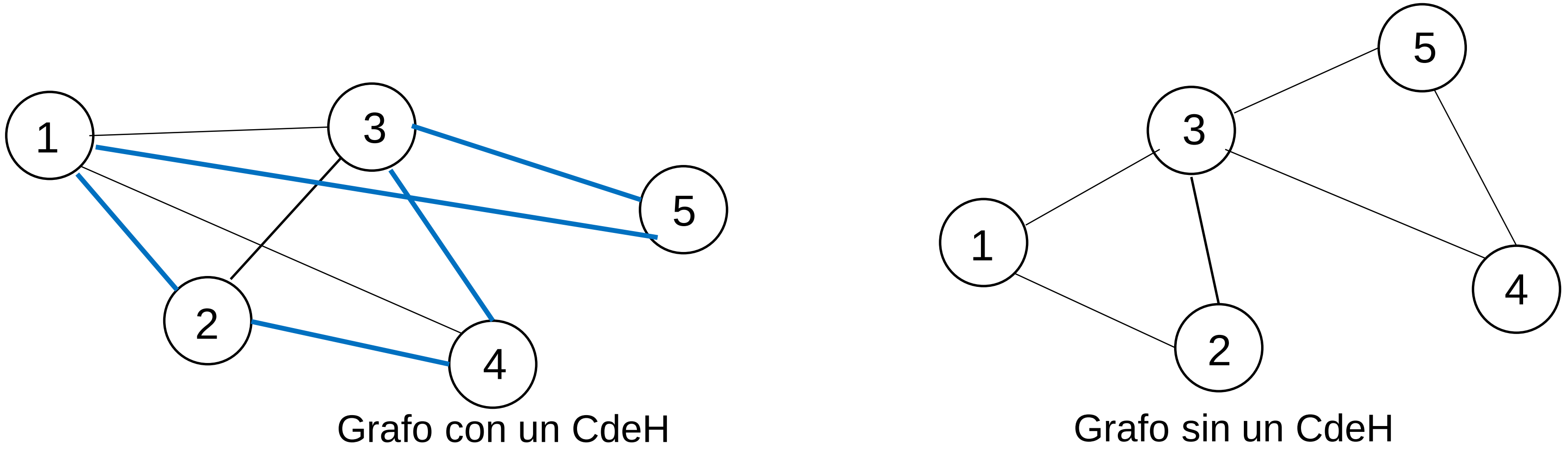
Cada evaluación se puede hacer en tiempo **$O(|\varphi|^2)$**

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 y tiene un circuito de Hamilton}

Circuito de Hamilton (CdeH): recorrido de todos los vértices de un grafo sin repetirlos, salvo el primero al final.



Una MT M que decide CH prueba con todas las **permutaciones** de vértices (**no se conoce otro algoritmo**). Por ej., en grafos con 5 vértices, hay que probar con 12345, 12354, 12435, 12453, 12534, 12543, etc.

Las permutaciones de m vértices suman $m! = m.(m – 1).(m – 2).(m – 3)…2.1 =$ $\mathbf{O(m^m)}$.

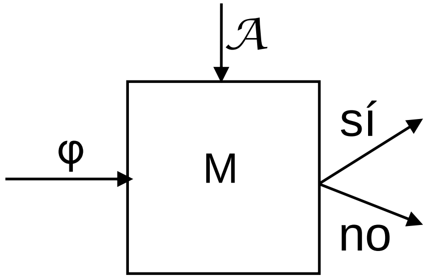
Por otro lado, chequear si una permutación C es un CdeH se puede hacer en tiempo $\mathbf{O(|G|^2)}$ (ejercicio)

En definitiva, M puede llegar a ejecutar $\mathbf{O(m^m.|G|^2) = O(n^n.n^2) = exp(n)}$ **pasos.**

¿Qué tienen en común los lenguajes SAT y CH?

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

Dada una fórmula booleana φ, una MT M puede llegar a decidir si φ ∈ SAT en tiempo exponencial, pero contando con una asignación $\mathcal{A}$, M puede **verificar** si φ ∈ SAT **en tiempo polinomial.**



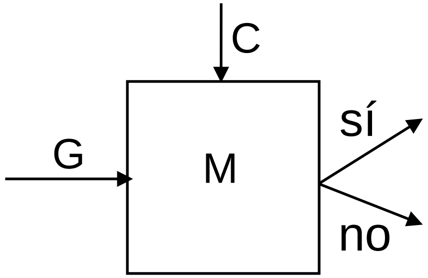
$\mathcal{A}$ es un **certificado** o **prueba** de φ.

M es un **verificador eficiente** del lenguaje SAT.

En tiempo poly(|φ|), M puede verificar si φ ∈ SAT con la ayuda de $\mathcal{A}$.

- **CH = {G : G es un grafo y tiene un circuito de Hamilton}**

Dado un grafo G = (V, E), una MT M puede llegar a decidir si G ∈ CH en tiempo exponencial, pero contando con una permutación C de V, M puede **verificar** si G ∈ CH **en tiempo polinomial.**

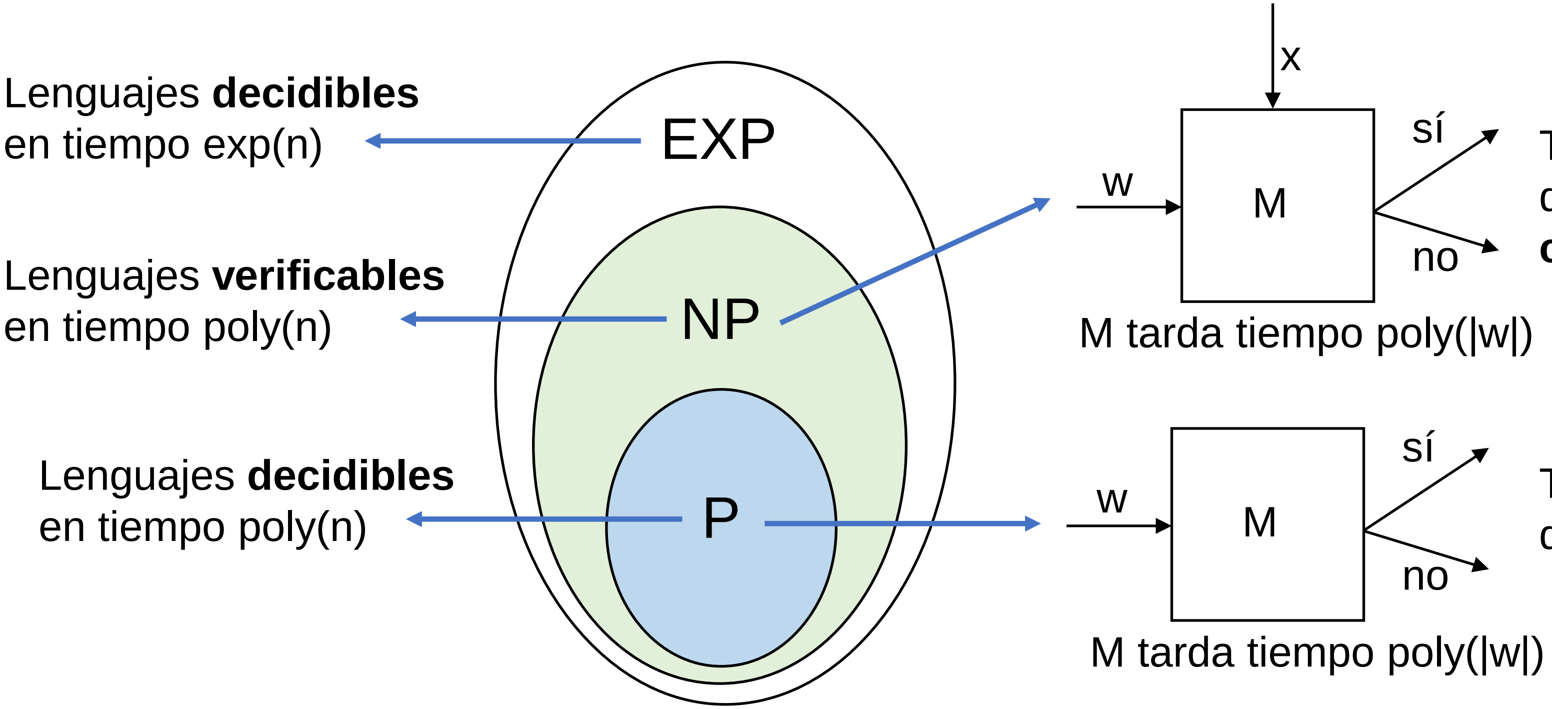


C es un **certificado** o **prueba** de G.

M es un **verificador eficiente** del lenguaje CH.

En tiempo poly(|G|), M puede verificar si G ∈ CH con la ayuda de C.

Segunda versión de la jerarquía temporal (acotada a la clase EXP)



Todo lenguaje L de NP cuenta con una MT M que en poly(|w|) pasos puede **verificar** si w ∊ L, **con la ayuda de otra cadena (certificado x).**

$|x| \leq$ poly(|w|) ¿por qué?
Se dice que x es un <u>certificado sucinto</u>.

Todo lenguaje L de P cuenta con una MT M que en poly(|w|) pasos puede **decidir** si w ∊ L.

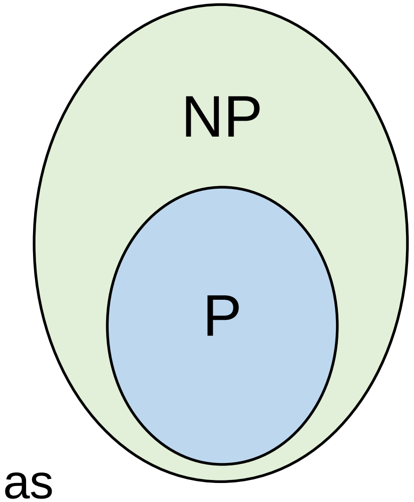
Se <u>cumpliría</u> $P \subset NP \subset EXP$

Ejercicio: ¿por qué $P \subseteq NP$?

- En otras palabras:
  Si un problema está en P, podemos asegurar que sus soluciones **se encuentran eficientemente**.
  Si un problema está en NP, sólo podemos asegurar que sus soluciones **se verifican eficientemente**.

- Intuitivamente, **P ≠ NP** (encontrar una solución parece ser más difícil que verificarla). Sin embargo, al día de hoy esta relación **no ha podido ser probada**.

El problema P vs NP (¿P = NP o P ≠ NP?)

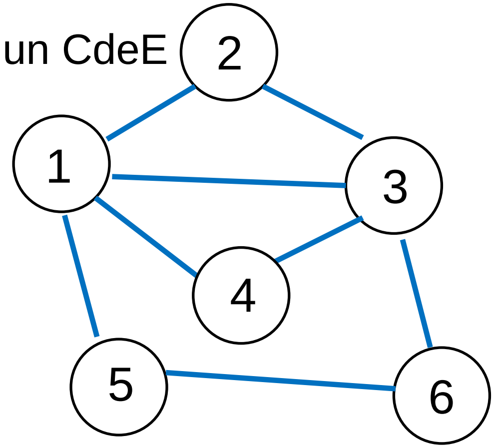


- El **problema P vs NP** fue planteado hace más de 50 años.
- Es uno de los **siete problemas del milenio** (definidos en el año 2000).
  Quien lo resuelva recibirá un premio de USD 1 MM del Instituto Clay (EEUU).
- No parece que pueda resolverse en el corto plazo, con las **matemáticas existentes**.
- **Miles de lenguajes de interés están en NP – P** (lenguajes relacionados con problemas de la lógica, grafos, aritmética, teoría de conjuntos, álgebra, combinatoria, redes, etc).
- Pareciera que la única manera de decidir los lenguajes de NP – P es por medio de la **fuerza bruta** (**búsqueda exhaustiva** en el espacio de todas las soluciones posibles).
- Por el contrario, los algoritmos que deciden los lenguajes de P son **analíticos** (parten del conocimiento profundo del problema, y se han elaborado con el empleo de ingenio, creatividad, experiencia, etc).
- La mayoría de los investigadores opina que **P ≠ NP.** Una minoría opina lo contrario, con el argumento de que **queda mucho por aprender de la algorítmica** (tiene asidero, ya que a lo largo de los años han habido importantes descubrimientos - primalidad, producto de matrices, camino en grafos no dirigidos, etc. -).
- **En la práctica se asume P ≠ NP**, y por eso a los lenguajes de interés de NP – P se los procesa con **remediaciones específicas** (cadenas de cierta longitud y/o forma, aproximaciones polinomiales, etc).

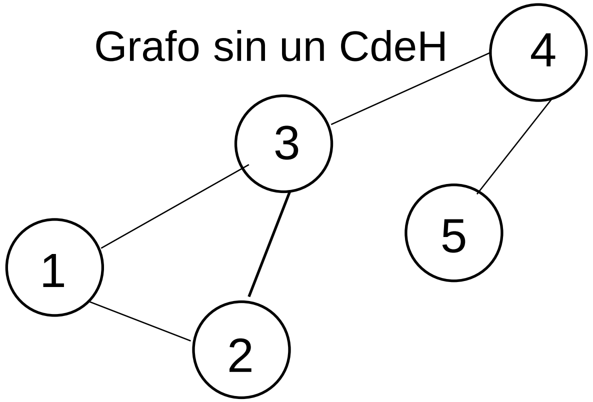
Otros ejemplos clásicos de lenguajes en P y NP

- **CE = {G | G es un grafo y tiene un circuito de Euler}**
  Circuito de Euler (CdeE): recorrido de todos los arcos del grafo sin repetirlos, desde y hasta un mismo vértice.

Grafo con un CdeE



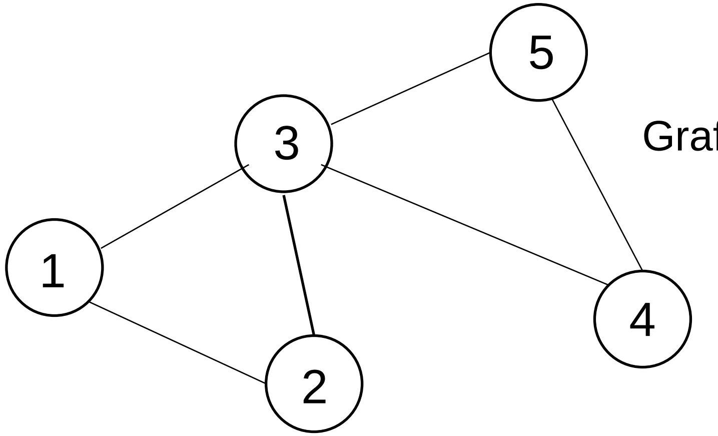
Grafo sin un CdeH



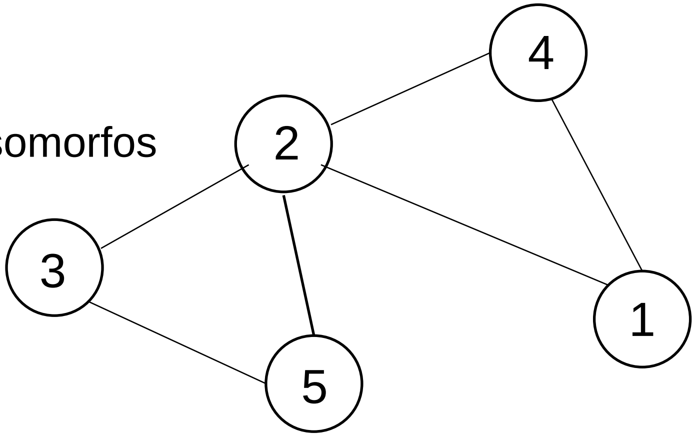
El lenguaje CE está en P

Vimos que en cambio, **el lenguaje CH** de los grafos con un circuito de Hamilton **no estaría en P**

- **ISO = {(G_1, G_2) | G_1 y G_2 son grafos isomorfos}**
  Grafos isomorfos: son iguales salvo por la denominación de sus arcos.



Grafos isomorfos



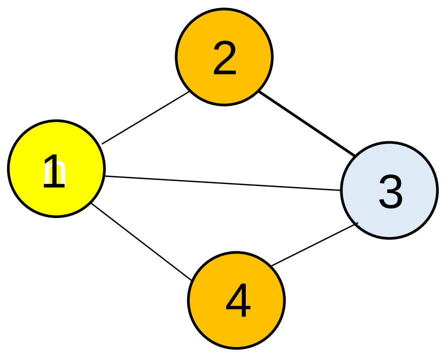
El lenguaje ISO no estaría en P

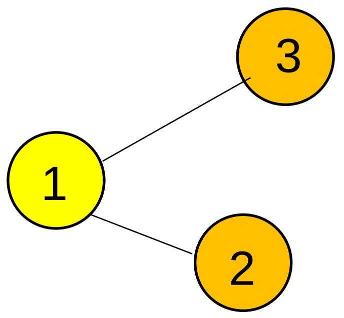
El lenguaje ISO está en NP

Recientemente se probó que se lo puede decidir en tiempo **cuasi polinomial** (del tipo $n^{\log n}$)

Otros ejemplos clásicos de lenguajes en P y NP (continuación)

- **K-COLOR = {G | G es un grafo y sus vértices se pueden colorear con K colores sin producir vértices vecinos con el mismo color}**





El lenguaje K-COLOR está en NP

Si K = 2, **K-COLOR está en P**

Si K ≥ 4 y el grafo es planar (se puede dibujar en el plano sin que los arcos se crucen), **siempre existe solución (**Teorema de los Cuatro Colores)

- **K-SAT = {φ | φ es una fórmula booleana satisfactible y es una conjunción de cláusulas de K variables}**

P. ej., $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \in$ 3-SAT

$(x_1 \vee \neg x_1) \wedge (x_2 \vee \neg x_2) \wedge (x_3 \vee \neg x_3) \in$ 2-SAT

El lenguaje K-SAT está en NP

Si K = 2, **K-SAT está en P**

Si K ≥ 3, K-SAT **no estaría en P**

- **PRIMOS = {N | N es un número primo}**
  Número primo: número natural mayor que 1 que es sólo divisible por 1 y por sí mismo

El lenguaje PRIMOS está en P
En cambio, obtener los factores primos de un número (problema de **factorización**) **no estaría en P**

- ¿Por qué el problema de los circuitos de Euler **sería más fácil** que el problema de los circuitos de Hamilton?

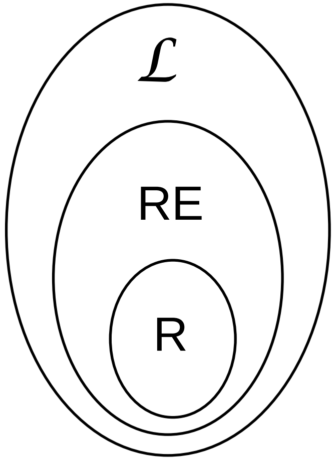
  ¿Por qué 2-COLOR y 2-SAT **están en P**, mientras que 3-COLOR y 3-SAT **no estarían en P**?

  ¿Por qué K-COLOR desde K = 4 **tiene solución**?

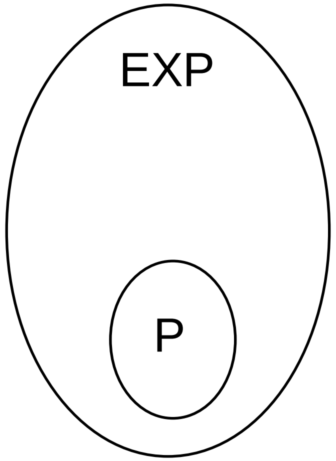
  ¿Por qué decidir si un número N es primo **sería más fácil** que encontrar los factores primos de N?

  Etc., etc., etc.

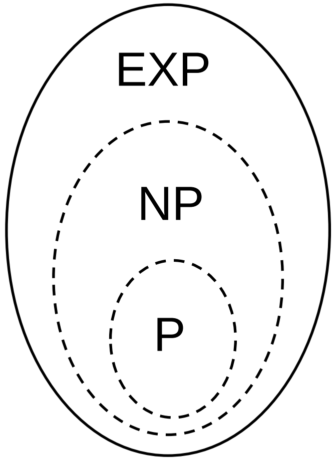
- A diferencia de la computabilidad, la complejidad computacional tiene hoy día **muchas preguntas abiertas.**



Se <u>cumple</u> $R \subset RE \subset \mathcal{L}$



Se <u>cumple</u> $P \subset EXP$



Se <u>cumpliría</u> $P \subset NP \subset EXP$

- En las próximas dos clases veremos cómo se encaran dichas preguntas (**NP-completitud**, **modelos computacionales alternativos**, **jerarquías alternativas**, etc).

Anexo

Otras métricas de complejidad computacional

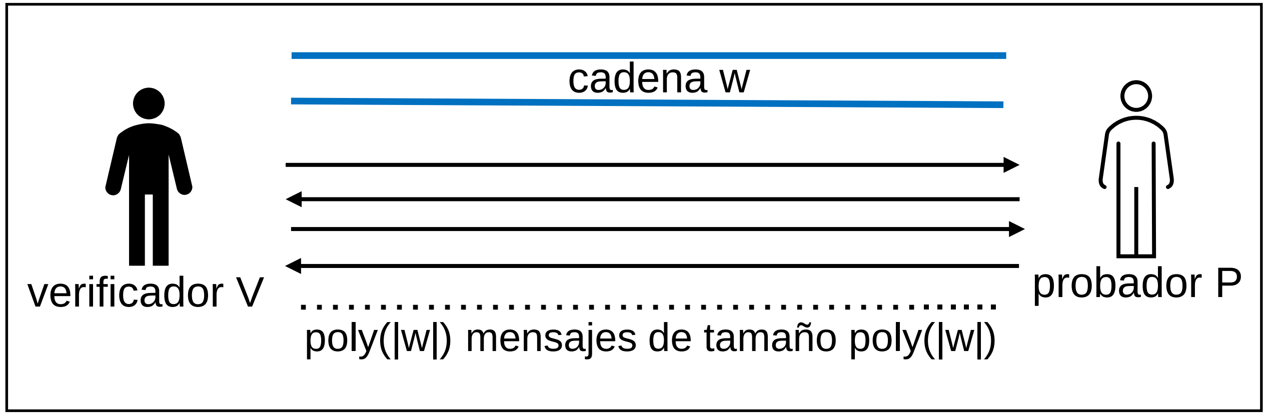
- **Dinámicas**
  - ✓ **Cantidad de cambios de dirección** del cabezal de una MT (Hennie).
  - ✓ **Consumo de recursos abstractos** (Blum).
  - ✓ Etc.

- **Estáticas**
  - ✓ **Mínimo valor |Q|.|Γ|** de una MT (Shannon).
  - ✓ **Complejidad estructural** de una MT (Chomsky): según el tipo de cintas, el tipo de movimientos, el espacio recorrido, etc. (autómatas finitos, autómatas con pila, autómatas linealmente acotados, MT generales).
  - ✓ Etc.

Definición equivalente de la clase NP (menos intuitiva)

- Un **probador P** (MT de poder ilimitado) tiene que convencer a un **verificador V** (MT de tiempo poly(n)) que una cadena w pertenece a un lenguaje L.

- Para ello, V y P intercambian preguntas y respuestas. Las preguntas y respuestas miden y suman poly(|w|).



En la definición anterior de NP: P le envía a V UN certificado.

En la definición nueva de NP (equivalente): P intercambia con V VARIOS mensajes.

La equivalencia de las dos definiciones **no es intuitiva**. Por ejemplo, un maestro puede explicar un tema a sus alumnos de dos formas:

(1) respondiendo preguntas recién cuando termina la explicación, o bien:

(2) respondiendo preguntas durante la explicación,

y es lógico suponer que de la forma (2) los alumnos comprenderán mejor la explicación.

- Si L ∈ NP, entonces existe V tal que:

  Si w ∈ L, existe un P que puede convencer a V.

  Si w ∉ L, no existe ningún P que pueda convencer a V.

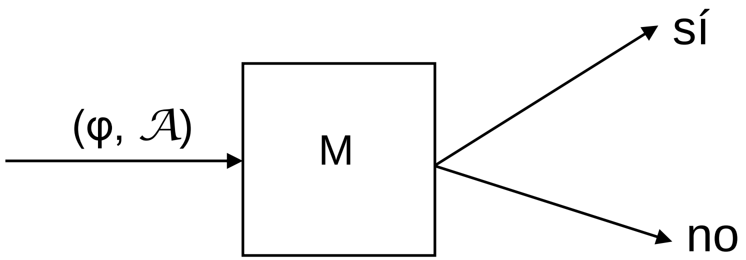
Acerca de los complementos de los lenguajes de NP

- SAT = {φ | φ es una fórmula booleana satisfactible con m variables}

  SAT no estaría en P: hay 2^m asignaciones $\mathcal{A}$ para chequear.

  SAT está en NP: dadas una fórmula booleana φ y una asignación $\mathcal{A}$, se puede verificar en tiempo poly(n) si $\mathcal{A}$ satisface φ.

  ¿SAT^C está en NP? ¿Cuánto mide un certificado en este caso?



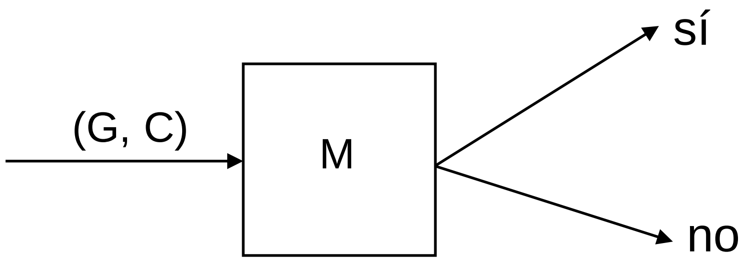
M tarda tiempo poly(|φ|)

- CH = {G : G es un grafo con m vértices y tiene un circuito de Hamilton}

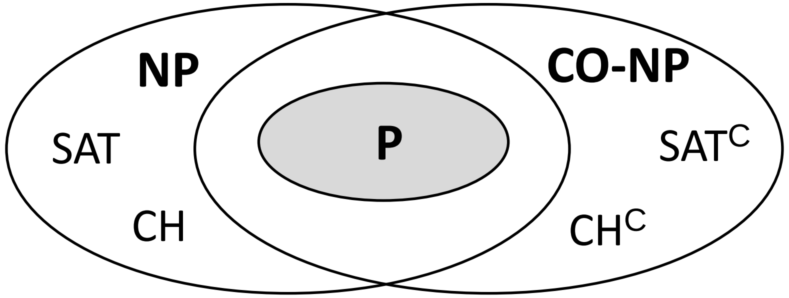
  CH no estaría en P: hay m! permutaciones C de V para chequear.

  CH está en NP: dados un grafo G y una permutación C de V, se puede verificar en tiempo poly(n) si C es un CdeH de G.

  ¿CH^C está en NP? ¿Cuánto mide un certificado en este caso?



M tarda tiempo poly(|G|)



CO-NP tiene los complementos de los lenguajes de NP.

La conjetura aceptada es que **NP ≠ CO-NP**. Es decir que **NP no sería cerrada con respecto al complemento**.

En cambio, **P sí es cerrada con respecto al complemento** (ejercicio), lo que refuerza la conjetura P ≠ NP.

Otra conjetura aceptada es que **P ⊂ NP ⋂ CO-NP**.

Sobre CO-NP profundizamos en la próxima clase.

La complejidad temporal y su relación con la codificación de las cadenas

Ejemplo. Sea DIV-3 = {N | N es un número natural que tiene un divisor que termina en 3}

- Una MT M que decide DIV-3 divide N por 3, 13, 23, etc., hasta encontrar eventualmente un divisor de N (**no se conoce otro algoritmo**).

  P. ej., si N = 100, M divide N por 3, 13, 23, 33, 43, 53, 63, 73, 83, 93 (10 divisiones).

  M ejecuta unas N/10 iteraciones, es decir O(N). Cada división se puede hacer en tiempo polinomial**.**

- Por lo tanto, **M ejecuta O(N) iteraciones de tiempo polinomial.** Resta expresar N en términos de n:

  1) Si N se codifica en **unario**, n = N. Así, M hace O(n) = **poly(n) iteraciones.**

  2) Si N se codifica en **binario**, n = $O(\log_2 N)$, y por lo tanto N = $O(2^n)$. Así, M hace **exp(n) iteraciones.**

  3) Si N se codifica en **cualquier otra base**, M hace **exp(n) iteraciones.**

- **La única codificación de los números no utilizable es la unaria.**

Clase práctica 5

Un par de ejemplos de prueba de pertenencia a la clase P

Ejemplo 1. El lenguaje 2-SAT pertenece a P.

Prueba (idea general). La siguiente MT M decide 2-SAT en tiempo poly(n). Dada una fórmula booleana, M hace:

1. Asigna arbitrariamente el valor *verdadero* a alguna variable x, y completa consistentemente todas las asignaciones que puede en todas las cláusulas.

2. Si detecta insatisfactibilidad, intenta lo mismo que en (1) pero con el valor *falso* para x.

3. Si vuelve a detectar insatisfactibilidad, **rechaza**.

4. Elimina las cláusulas satisfechas.

5. Si no quedan cláusulas, **acepta** (se encontró una asignación que satisface la fórmula).

1. Si quedan cláusulas, vuelve a (1) para procesar la fórmula booleana reducida.

Notar que luego de cada iteración principal, la fórmula tiene mínimamente una cláusula menos, y que el procesamiento de la fórmula en cada iteración es de tiempo polinomial.

Ejemplo 2. COPRIMOS = {(N_1, N_2) | N_1 y N_2 son coprimos, es decir que tienen como máximo común divisor el número 1} pertenece a P.

Prueba. La idea es construir una MT M basada en el *algoritmo de Euclides* para el cálculo del máximo común divisor. Después de una serie de divisiones y de intercambios entre los números N_1 y N_2, en algún momento N_2 termina quedando con el valor 0 y N_1 con el máximo común divisor de los dos, a partir del cual M acepta o rechaza.

Formalmente, dada una entrada válida w, M hace:

1. Hace $N_1 := N_1 \bmod N_2$ (la función *mod* devuelve el resto de la división entre N_1 y N_2).
2. Intercambia los valores de N_1 y N_2.
   Si $N_2 = 0$, acepta si $N_1 = 1$ y rechaza si $N_1 \neq 1$.
   Si $N_2 \neq 0$, vuelve al bloque 1.

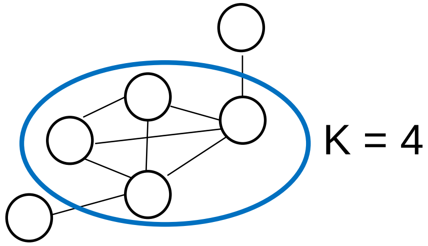
La correctitud del algoritmo se puede comprobar sin dificultad. En cuanto al tiempo de ejecución de M, es el tiempo de ejecución del conjunto de iteraciones de los bloques 1 y 2:

- Bloque 1: una división entre N_1 y N_2 requiere $O(|(N_1, N_2)|^2) = O(n^2)$ pasos.
- Bloque 2: los intercambios y comparaciones de números requiere $O(|(N_1, N_2)|) = O(n)$ pasos.
- Después de ejecutarse el bloque 1, N_1 se reduce a menos de la mitad (salvo la primera vez, si inicialmente es menor que N_2). Así, M ejecuta a lo sumo $O(\log_2 N_1) = O(|N_1|) = O(n)$ iteraciones, y por lo tanto, el tiempo de ejecución de M es $O(n).(O(n^2) + O(n)) = O(n^3)$.

El problema del clique y su relación con las clases P y NP

Caso general. El problema del clique no estaría en P y está en NP. El problema consiste en determinar si un grafo G tiene un clique de tamaño K. Un clique de tamaño K en un grafo G es un subgrafo completo de G con K vértices.

- El lenguaje que representa el problema es:



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

- La mejor MT conocida para **decidir** si un grafo G tiene un clique de tamaño K, consiste en recorrer uno a uno todos los conjuntos C de K vértices de V y chequear en cada caso si C determina un clique.

  Existen $\binom{m}{K}$ conjuntos C de K vértices en V, siendo m la cantidad de vértices de V.

  Así, las iteraciones de la MT suman $O(m.(m – 1).(m – 2)…(m – K + 1) / K!) = O(m^m) = O(n^n) =$ **exp(n).**

  Por lo tanto, CLIQUE no estaría en P.

- Por otro lado, se puede **verificar** en tiempo poly(n) si un conjunto C de K vértices define un clique de tamaño K de un grafo G: se chequea que todos los pares de vértices de C sean arcos de G, lo que tarda $O(|K|^2.|E|) = O(|V|^2.|E|) = O(|G|^3) = O(n^3) =$ **poly(n).**

  Por lo tanto, CLIQUE está en NP. M es un verificador eficiente de CLIQUE, con certificados C de tamaño O(n).

Variante del problema del clique que está en P

El enunciado del problema es el mismo que antes, pero con la diferencia de que ahora el tamaño K del clique no forma parte de las instancias del problema, es una constante.

- El lenguaje que representa el problema es:

$CLIQUE_K$ = {G | G es un grafo que tiene un clique de tamaño K}

- Como K no forma parte de las entradas del algoritmo descripto previamente, el problema ahora tiene resolución polinomial, **está en P**:

a) Hay $O(m.(m – 1).(m – 2)…(m – K + 1) / K!) = O(m^K) = O(n^K)$ conjuntos de K vértices en V para chequear.

b) Cada chequeo se puede hacer en tiempo $O(n^3)$.

Total: $O(n^K.n^3)$ = poly(n) pasos.

Nota: podría discutirse de todos modos si para un K muy grande, por ejemplo 1000, el tiempo de la MT construida puede considerarse aceptable.

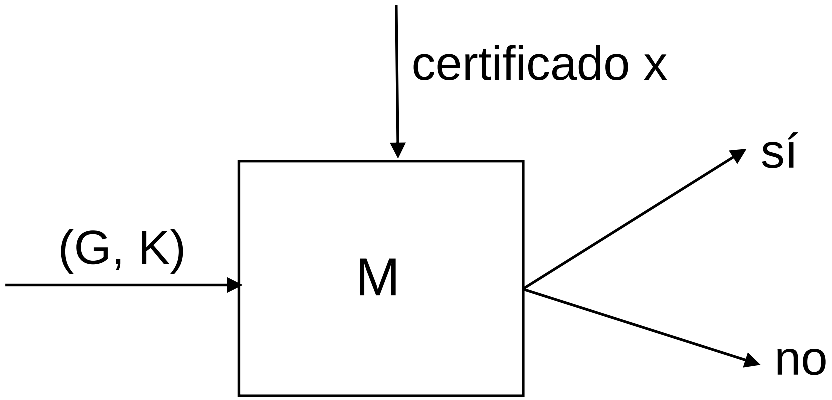
<u>El complemento del problema no estaría en NP</u>

Sea el problema complemento del problema del clique.

- El lenguaje que lo representa es:

CLIQUE^C = {(G, K) | G es un grafo que no tiene un clique de tamaño K}

- En este caso, no alcanza con certificados de K vértices. Un certificado debe incluir a **todos** los conjuntos de K vértices de V, porque se tiene que chequear que **ninguno** determina un clique de tamaño K de G.



El certificado x tiene que incluir los exp(n) conjuntos de K vértices de G. Así, **CLIQUE^C no estaría en NP.**

- La conjetura más aceptada es que si un lenguaje está en NP, su complemento no necesariamente está en NP.

Una tercera manera para definir la clase NP

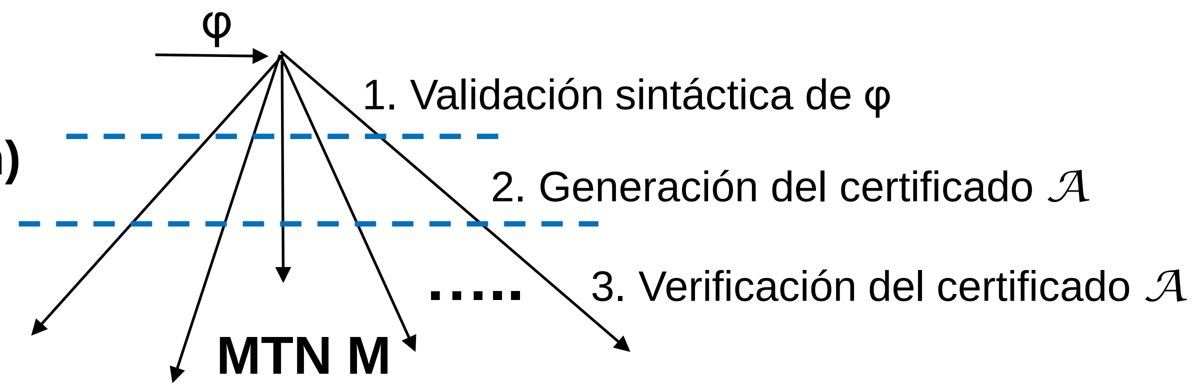
- Uso de **MTN (MT no determinísticas).**

- Definición alternativa: **L ∈ NP sii existe una MTN que decide L en tiempo poly(n),**
  tal que un MTN tarda tiempo poly(n) sii todas sus computaciones tardan tiempo poly(n).

- Por ejemplo, la siguiente MTN M decide SAT en tiempo poly(n). Dada una entrada φ, M hace:

  1. Si φ no es una fórmula correcta, rechaza: **O(n)**
  2. Genera no determinísticamente una asignación $\mathcal{A}$: **O(n)**
  3. Acepta sii $\mathcal{A}$ satisface φ: **O(n²)**

  Por lo tanto, en total M hace **O(n²) pasos**



Todas las computaciones de M tardan tiempo poly(n)

- <u>Idea</u>: en lugar de construir una MTD (MT determinística) que procesa φ con la ayuda de un certificado $\mathcal{A}$, se construye una MTN (MT no determinística) que procesa φ **generando una $\mathcal{A}$ distinta en cada computación**.