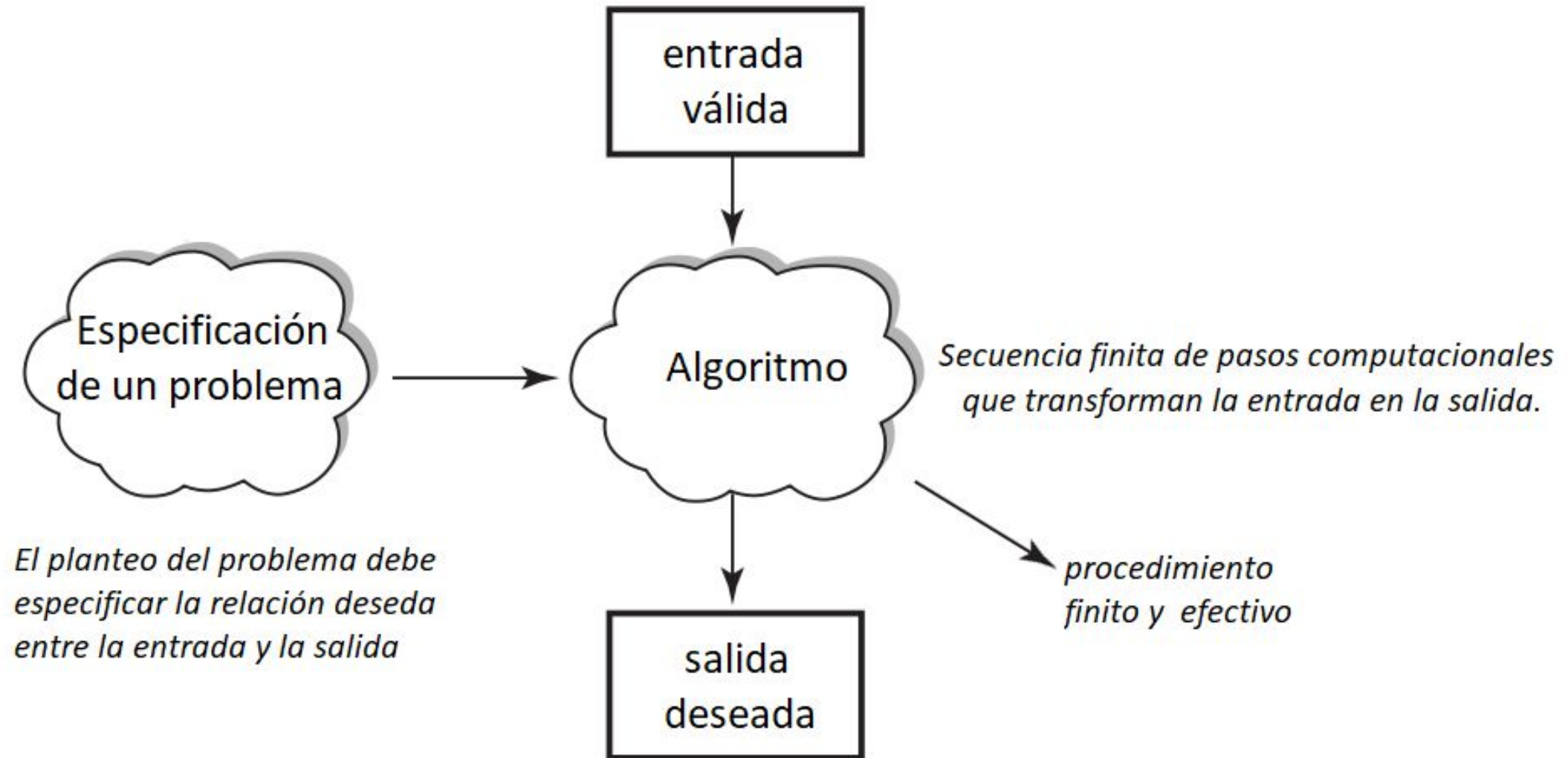
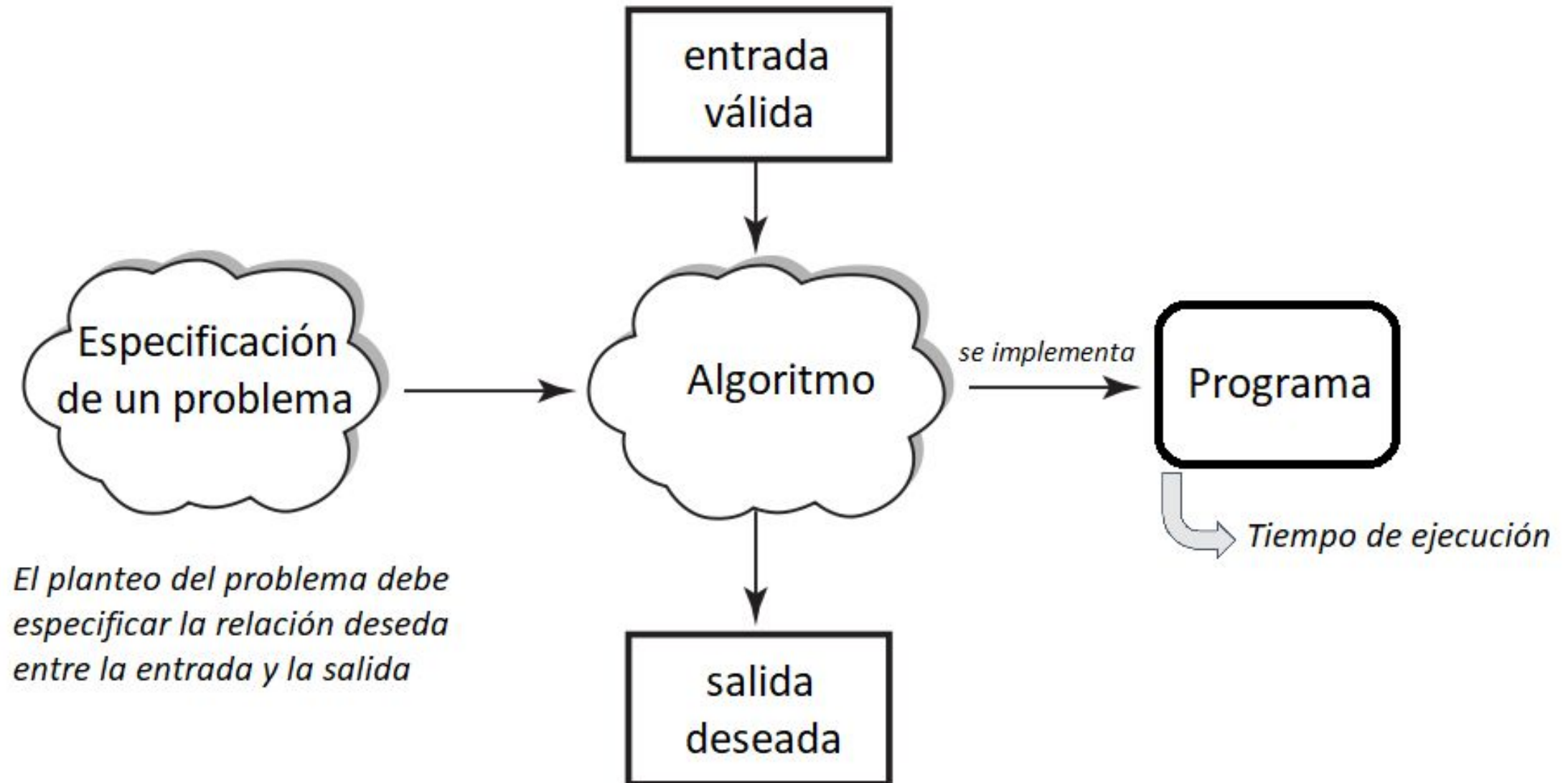


# Complejidad computacional y clases de problemas

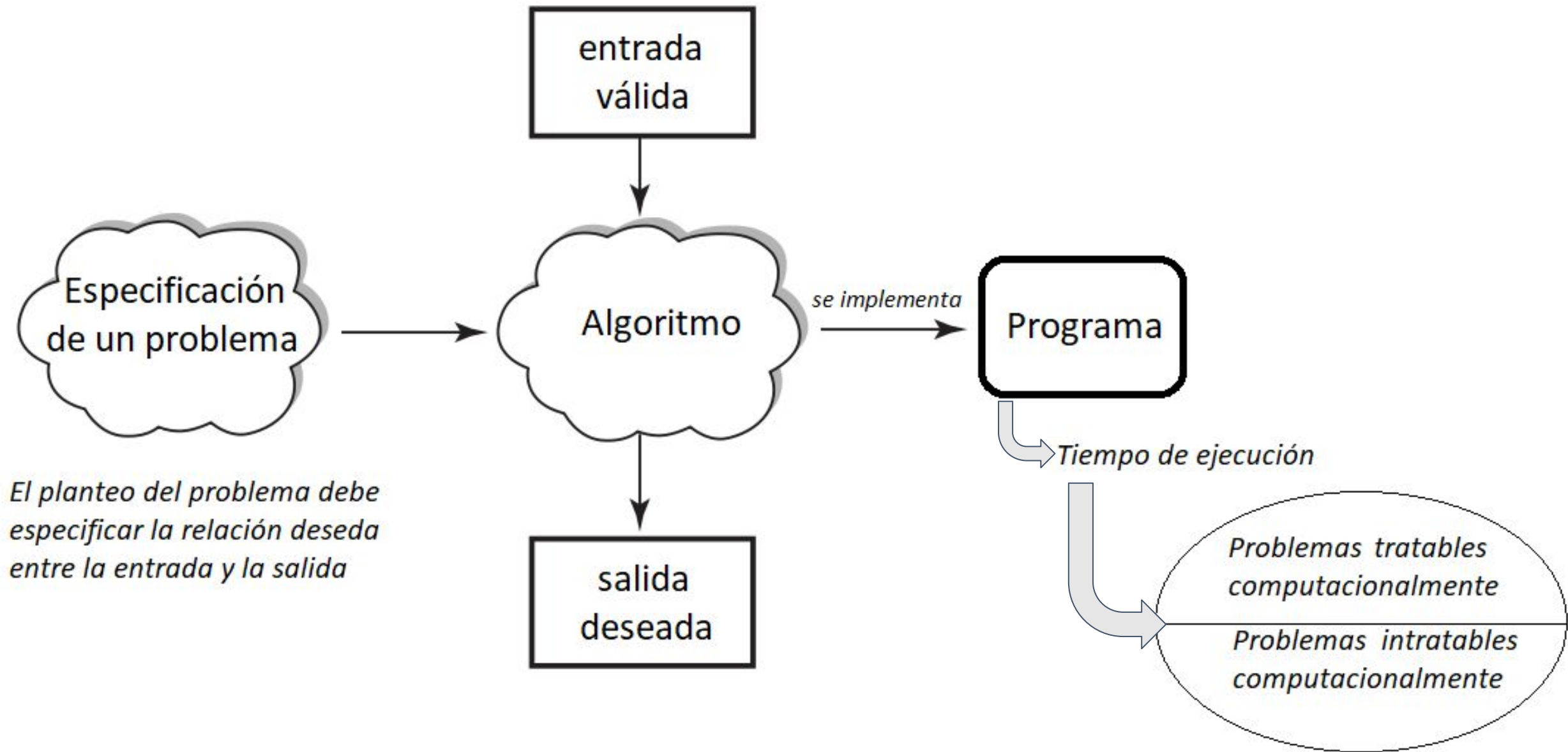
# Problemas y Algoritmos



# Problemas, Algoritmos y Programas



# Problemas, Algoritmos y Programas



# Algoritmos y Complejidad

---

Los algoritmos estudiados hasta el momento -> algoritmos de tiempo polinomial  $O(n^k)$  para algún  $k$  constante

**¿Todos los problemas pueden ser resueltos en tiempo polinomial? NO**

- Algunos **pueden ser resueltos**, pero... requieren un **tiempo exponencial**.
- Algunos **no pueden ser resueltos** por ninguna computadora, sin importar cuánto tiempo esperemos: por ejemplo el problema del Halting de Turing\*.

*\* consiste en determinar para un algoritmo arbitrario determinístico  $A$  y una entrada  $I$ , si el algoritmo  $A$  con la entrada  $I$  terminará alguna vez (o entrará en un loop infinito). Se sabe que este problema es indecidible. Por lo tanto, no existe ningún algoritmo (de cualquier complejidad) para resolver este problema.*

# Algoritmos y Complejidad

---

Generalmente, pensamos los problemas resueltos por algoritmos:

- acotados polinomialmente -> **Tratables o “fáciles”**
- que requieren un tiempo exponencial -> **Intratables o “difíciles”**

# Problemas intratables: un ejemplo

## Un simple rompecabezas

**queremos saber: ¿“existe o no” una solución al rompecabezas?**

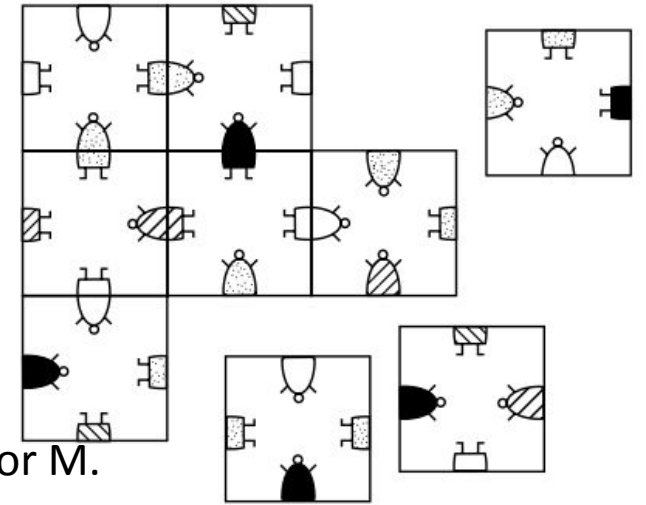
**¿Podemos escribir un algoritmo lo resuelva?**

- Las piezas son finitas y los lugares donde ubicarlas también.
- Hay un número finito de maneras de organizar las piezas en un tablero de  $M$  por  $M$ .
- Se puede probar fácilmente si una disposición dada de las piezas sobre el tablero es legal



Podemos diseñar un algoritmo “ingenuo” que pruebe todas las formas posibles de ubicar las piezas:

- deteniéndose y diciendo “sí” si la disposición actual es legal, o
- deteniéndose y diciendo “no” si se han considerado todas las formas de ubicar las piezas y todas son ilegales.

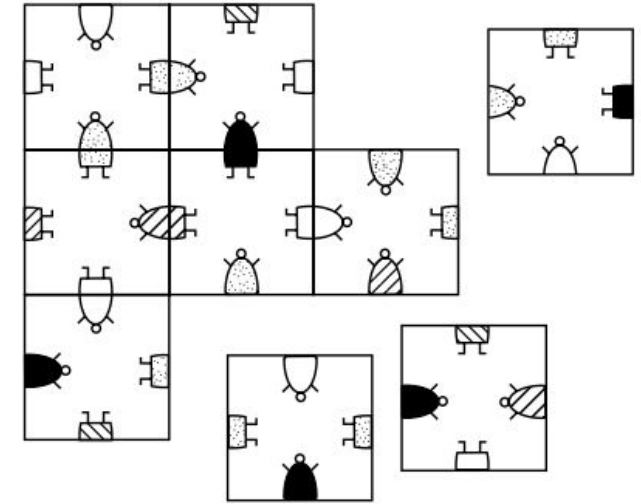


# Problemas intratables: un ejemplo

## Un simple rompecabezas

Supongamos:

- $M = 5$  (tablero de 5 por 5)  $\rightarrow N = 25$  (piezas)
- de cada pieza se conoce su orientación (no se rota)
- una computadora capaz de construir y evaluar mil millones de disposiciones de piezas sobre el tablero cada segundo (es decir, *una disposición cada nanosegundo*), incluyendo todo el cálculo involucrado.



La pregunta es: **¿cuánto tiempo tomará el algoritmo en el peor de los casos?**

Es decir, cuando no haya ninguna disposición legal, de modo que se verifiquen todas las formas posibles de ubicar las piezas sobre el tablero



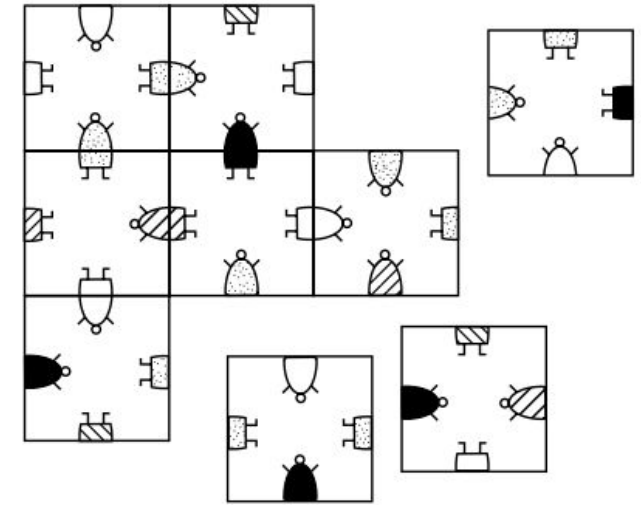
# Problemas intratables: un ejemplo

## Un simple rompecabezas

**N= 25**

Numeramos arbitrariamente las ubicaciones en una cuadrícula de 5 x 5

- 25 posibles piezas para colocar en la primera ubicación.
- 24 posibles piezas para colocar en la segunda ubicación.
- 23 para la tercera, y así sucesivamente.



**El número total de posibles formas de ubicar las piezas puede alcanzar:**

$$25 \times 24 \times 23 \times \dots \times 3 \times 2 \times 1 = \quad \mathbf{25!} \quad \rightarrow \text{contiene 26 dígitos}$$

Lo sorprendente: **nuestra computadora**, capaz de construir y evaluar mil millones de disposiciones de piezas sobre el tablero cada segundo,  
**¡tardará más de 490 millones de años en llegar a las 25! disposiciones!**

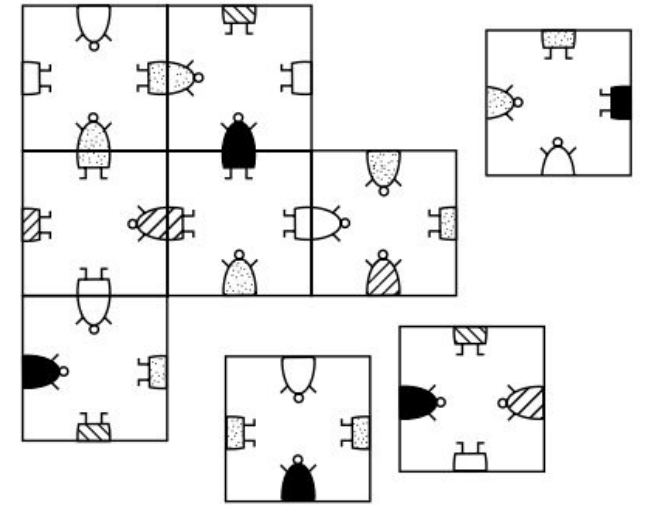
# Problemas intratables: un ejemplo

Un simple rompecabezas

¿Qué pasaría si  $M = 6$ ,  $N = 36$  piezas?

La cantidad de maneras de ubicar las 36 piezas sería 36!

Los valores serían inimaginablemente grandes, lejanos, mucho más lejanos que el tiempo transcurrido desde el Big Bang !!!



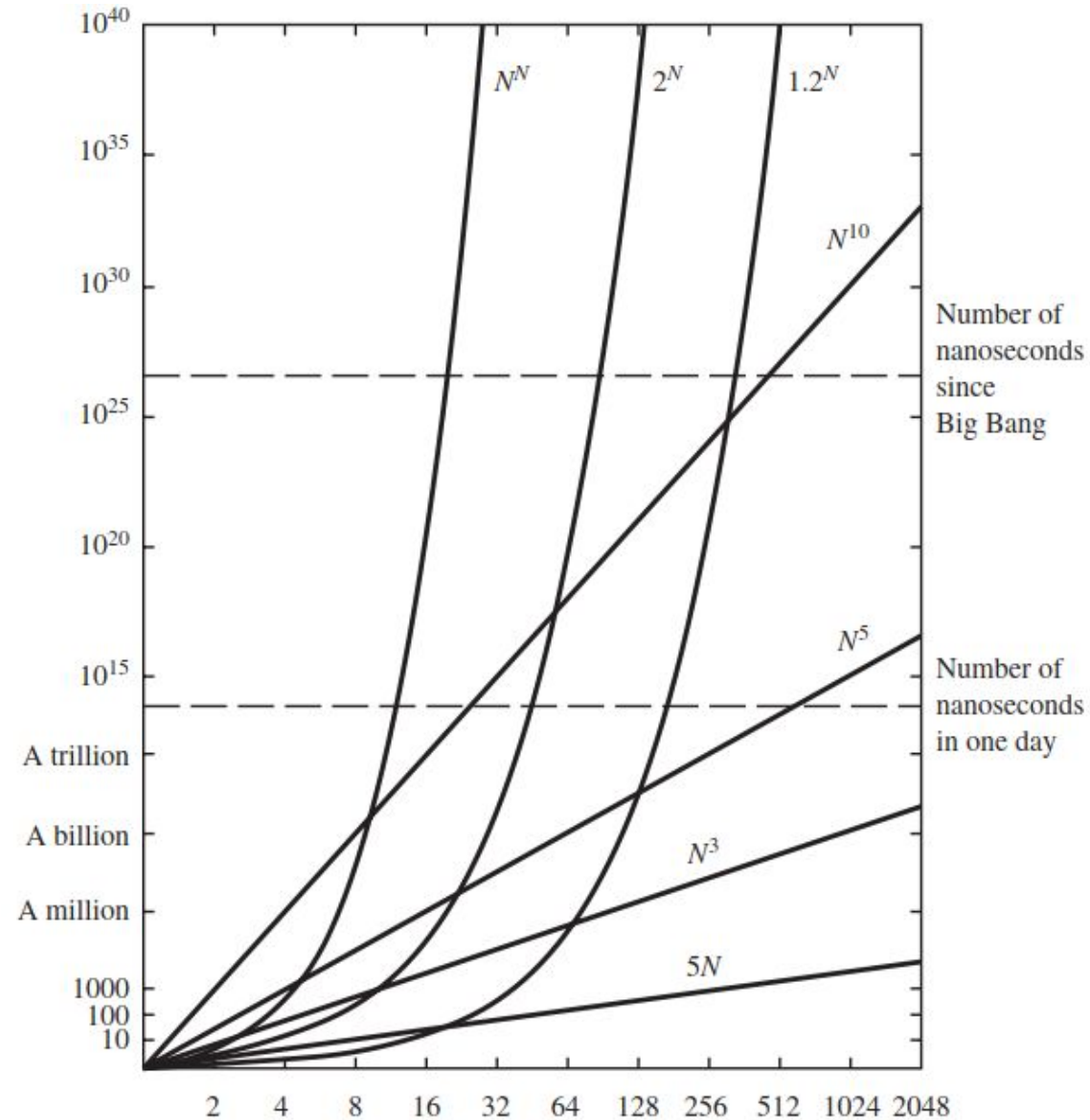
# Algoritmos y Complejidad

$N \backslash$ Function		20	60	100	300	1000
Polynomial	$5N$	100	300	500	1500	5000
	$N \times \log_2 N$	86	354	665	2469	9966
	$N^2$	400	3600	10,000	90,000	1 million (7 digits)
	$N^3$	8000	216,000	1 million (7 digits)	27 million (8 digits)	1 billion (10 digits)
Exponential	$2^N$	1,048,576	a 19-digit number	a 31-digit number	a 91-digit number	a 302-digit number
	$N!$	a 19-digit number	an 82-digit number	a 161-digit number	a 623-digit number	unimaginably large
	$N^N$	a 27-digit number	a 107-digit number	a 201-digit number	a 744-digit number	unimaginably large

El tiempo de ejecución “crece” como una función del tamaño de las entradas

David Harel (2012)

# Ritmo de crecimiento de algunas funciones



# Algoritmos y Complejidad

	$N$ Function	20	40	60	100	300
Polynomial	$N^2$	1/2500 millisecond	1/625 millisecond	1/278 millisecond	1/100 millisecond	1/11 millisecond
	$N^5$	1/300 second	1/10 second	78/100 second	10 seconds	40.5 minutes
Exponential	$2^N$	1/1000 second	18.3 minutes	36.5 years	400 billion centuries	a 72-digit number of centuries
	$N^N$	3.3 billion years	a 46-digit number of centuries	an 89-digit number of centuries	a 182-digit number of centuries	a 725-digit number of centuries

**Tiempos reales de ejecución de varios algoritmos hipotéticos para el problema del rompecabezas**

Referencia: 1 instrucción -> 1 nanosegundo ( $10^{-9}$  seg)

Para comparar: **El BIG BANG fue hace 13-15 billones de años!!**

David Harel (2012)

# Algoritmos y Complejidad

Que pasa si las máquinas aumentan la velocidad 100 veces, 1000 veces ...

Función	Número máximo de piezas manipuladas en una hora:		
	Con la computadora de hoy en día	con una computadora 100 veces más rápida	con una computadora 1000 veces más rápida
$N$	$A$	$100 \times A$	$1000 \times A$
$N^2$	$B$	$10 \times B$	$31.6 \times B$
$2^N$	$C$	$C + 6.64$	$C + 9.97$

$$\begin{aligned} N^2 &= 1000 B^2 \\ N &= 31.6 B \end{aligned}$$

$$\begin{aligned} 2^N &= 1000 2^C \\ \log 2^N &= \log (1000 2^C) \\ N &= C + \log 1000 \\ N &= C + 9.97 \end{aligned}$$

→ **Mejorar la velocidad de la computadora por un factor constante,**  
incluso uno grande, mejorará las cosas, pero ...  
**si el algoritmo es exponencial, lo hará de una manera muy insignificante.**

# Algoritmos y Complejidad

---

Generalmente, pensamos los problemas resueltos por algoritmos:

- acotados polinomialmente -> **Tratables o “fáciles”**

son los que vimos hasta ahora....

Problemas que tienen soluciones algorítmicas y tratables computacionalmente

- que requieren un tiempo exponencial -> **Intratables o “difíciles”**

Lo que se viene...

Problemas “difíciles” que tienen soluciones algorítmicas pero intratables computacionalmente

# Clases de Problemas

---

## NP-completitud y las clases P y NP

### Informalmente:

- **La clase P** problemas resolubles en un tiempo polinomial,  $O(n^k)$   $k$  constante y  $n$  tamaño de la entrada al problema.
- **La clase NP** problemas “verificables” en un tiempo polinomial.  
Ejemplo: para el rompecabezas, dada una configuración de piezas en el tablero, podemos chequear en un tiempo polinomial que esa configuración es válida.  
Cualquier problema en P también está NP. Si el problema está en P entonces podemos resolverlo en un tiempo polinomial ( no hay necesidad de “verificarlo”).
- **La clase NPC.** Un problema está en la clase NPC si está en NP y es tan “difícil” como cualquier problema en NP.  
Si algún problema NP-completo pudiera resolverse en un tiempo polinomial, entonces, cada problema en NP tiene un algoritmo que lo resuelve en un tiempo polinomial.



# Clases de Problemas

---

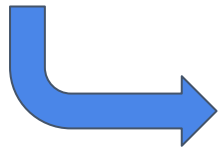
## ¿Porqué son interesantes los problemas NP-completos?

- 1º) Aunque ningún algoritmo eficiente ha sido encontrado para un problema NP-completo, nadie ha probado aún que dicho algoritmo no pueda existir. **Nadie sabe si un algoritmo eficiente existe o no para los problemas NP-completos.**
- 2º) El conjunto de problemas NP-completos tiene la propiedad notable de que **si existe un algoritmo eficiente para cualquiera de ellos, entonces existen algoritmos eficientes para todos ellos.** Esta relación entre los problemas NP-completos hace que la falta de soluciones eficientes sea aún más tentadora.
- 3º) **Varios problemas NP-completos son similares, pero no idénticos, a problemas para los cuales sí conocemos algoritmos eficientes.** Los informáticos están intrigados por cómo un pequeño cambio en la declaración del problema puede causar un gran cambio en la eficiencia del mejor algoritmo conocido.

# Clases de Problemas

---

## ¿Por qué conocer los problemas de NP-completos?



porque surgen muy a menudo en aplicaciones reales.

- ★ Si nos piden producir un algoritmo eficiente para un problema de NP-completo



perderemos tiempo en una búsqueda infructuosa.

- ★ Si podemos demostrar que el problema es NP-completo,



dediquemos tiempo a desarrollar un algoritmo eficiente que ofrezca una solución buena, pero no la mejor posible.

# Clases de Problemas

---

Varios problemas **NP-completos** son aparentemente similares a **problemas que sabemos cómo resolver en tiempo polinomial**.

★ **Camino simple más corto vs. camino simple más largo:**

El problema de hallar el **camino simple más corto** en un grafo puede resolverse en un tiempo acotado polinomialmente.

Sin embargo, encontrar **el camino simple más largo entre dos vértices es difícil**.  
Simplemente, determinar si un grafo contiene un camino simple con al menos un número dado de arcos es **NP-completo**.

# Clases de Problemas

---

Varios problemas **NP-completos** son aparentemente similares a problemas que sabemos cómo resolver en tiempo polinomial.

## ★ Ciclo de Euler vs. Ciclo Hamiltoniano:

Un **ciclo euleriano** de un grafo conectado, es un ciclo que atraviesa cada arista del grafo exactamente una vez, pero puede visitar cada vértice más de una vez. Determinar si un grafo tiene un ciclo euleriano toma un tiempo  $O(E)$ , de hecho podemos encontrar los arcos del ciclo en  $O(E)$ .

Determinar si un grafo dirigido tiene un **ciclo hamiltoniano es NP- COMPLETO**.

# Problemas $\mathcal{NP}$ -completos

---

## ¿Cómo demostramos que un problema está en $\mathcal{NP}$ -Completo?

- Cuando demostramos que un problema es NP-COMPLETO, estamos haciendo una declaración acerca de lo “difícil” que es.
- Estamos tratando de demostrar que probablemente no exista ningún algoritmo eficiente.

La demostración se basa en **tres conceptos claves**:

1. *problemas de decisión vs problemas de optimización*
2. *reducciones polinómicas*
3. *el primer problema NP-completo*

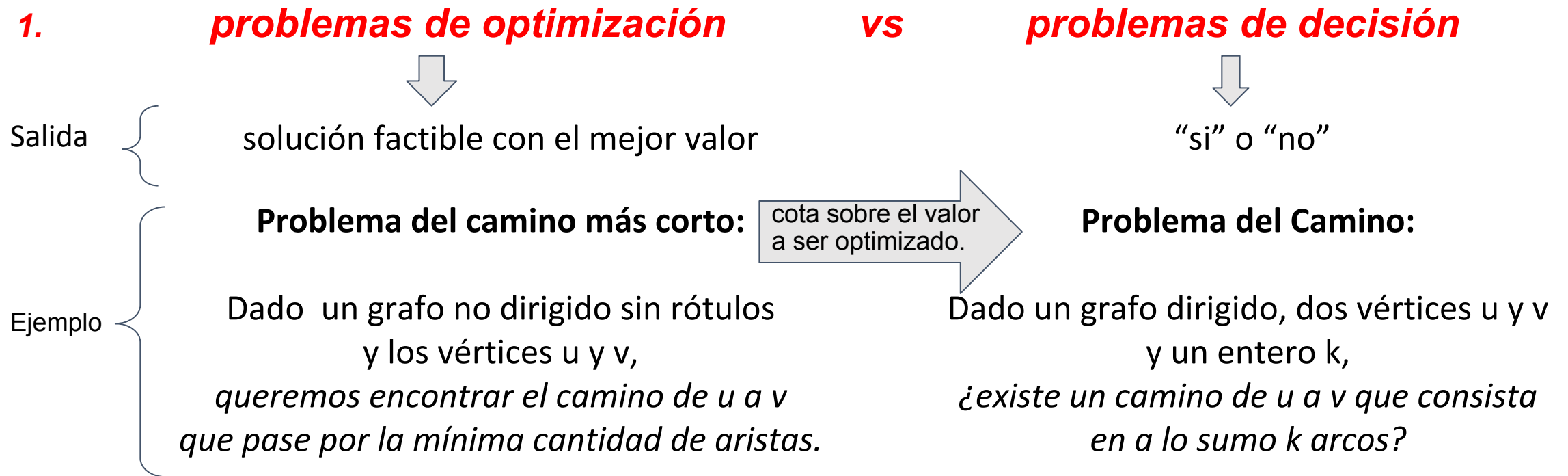
# Problemas $\mathcal{NP}$ -completos

¿Cómo demostramos que un problema está en NP-Completo?



# Problemas $\mathcal{NP}$ -completos

¿Cómo demostramos que un problema está en NP-Completo?



Podemos resolver **camino** resolviendo el **problema del camino más corto** y luego comparando el número de aristas con el valor del parámetro  $k$  del problema de decisión.

Luego, si un problema de optimización es fácil, el problema de decisión relacionado es fácil también.

# Problemas $\mathcal{NP}$ -completos

---

¿Cómo demostramos que un problema está en NP-Completo?

**NP-completitud  $\rightarrow$  problemas de decisión**

Los problemas de optimización son **por lo menos tan difíciles de resolver** como los problemas de decisión correspondientes



Las afirmaciones acerca de la dificultad de los problemas de decisión NP-completos son válidos para los problemas de optimización asociados



# Problemas $\mathcal{NP}$ -completos

---

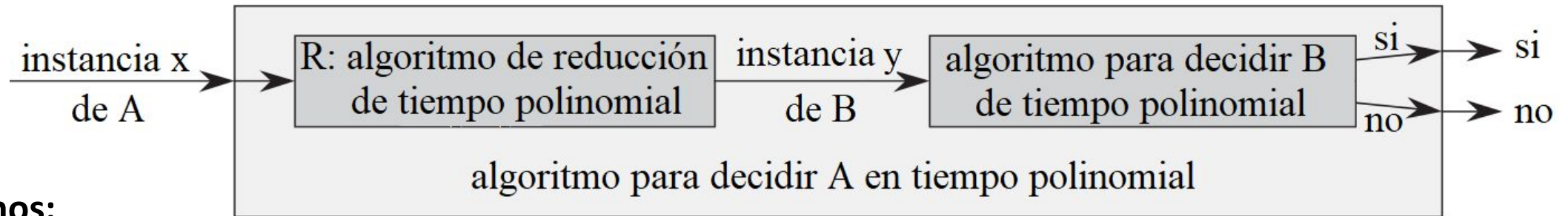
La demostración se basa en **tres conceptos claves**:

1. *problemas de decisión vs problemas de optimización*
2. ***reducciones polinómicas***
3. *el primer problema  $\mathcal{NP}$ -completo*

# Problemas $\mathcal{NP}$ -completos

¿Cómo demostramos que un problema está en NP-Completo?

## 2. Reducciones polinómicas



Supongamos:

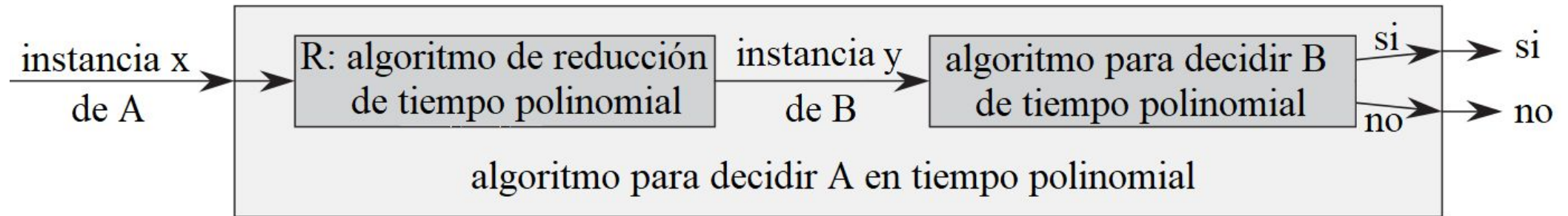
- nos interesa resolver un problema A y ya tenemos un algoritmo para otro problema B
- Sea  $R$  un algoritmo de reducción, que *en un tiempo polinomial* toma una entrada  $x$  para A y produce  $R(x)$ , una entrada para B tal que la respuesta correcta para A con  $x$  es **sí**, si y sólo si la respuesta correcta para B con  $R(x)$  es **sí**.

Entonces, componiendo  $R$  y el algoritmo para resolver B, tendremos un algoritmo para resolver A.

# Problemas $\mathcal{NP}$ -completos

¿Cómo demostramos que un problema está en NP-Completo?

## 2. Reducciones polinómicas



**Ejemplo:**

**Problema A:** Dada una sucesión de valores booleanos, ¿al menos uno de ellos tiene el valor verdadero? (En otras palabras, se trata del cálculo del or booleano de los  $n$  valores)

**Problema B:** Dada una sucesión de enteros, ¿el máximo de los enteros es positivo?

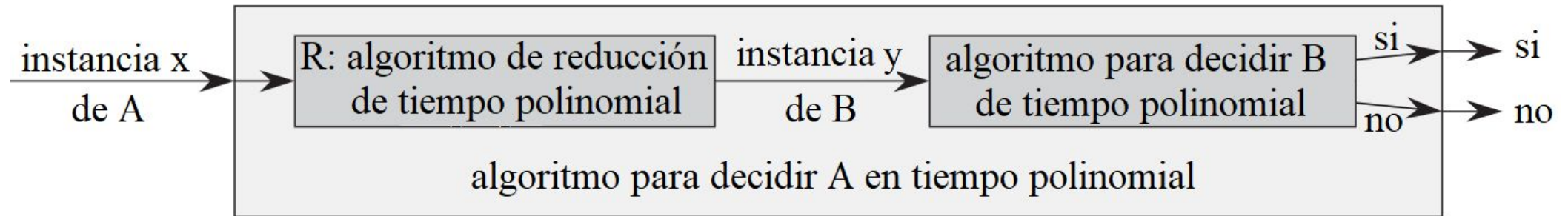
**Reducción R:**  $R(x_1, \dots, x_n) = (y_1, \dots, y_n)$  donde  $y_i = 1$  si  $x_i = \text{verdadero}$ , y  $y_i = 0$  si  $x_i = \text{falso}$ .

Es evidente que, si aplicamos a  $y_1, \dots, y_n$  un algoritmo para resolver B, resolveremos A para  $x_1, \dots, x_n$ .

# Problemas $\mathcal{NP}$ -completos

¿Cómo demostramos que un problema está en NP-Completo?

## 2. Reducciones polinómicas



Decimos que el **problema A es reducible polinomialmente al problema B** si existe una transformación polinómica de A a B

Notación:  $A \leq_p B$

Usamos la “facilidad” de B para probar la facilidad de A

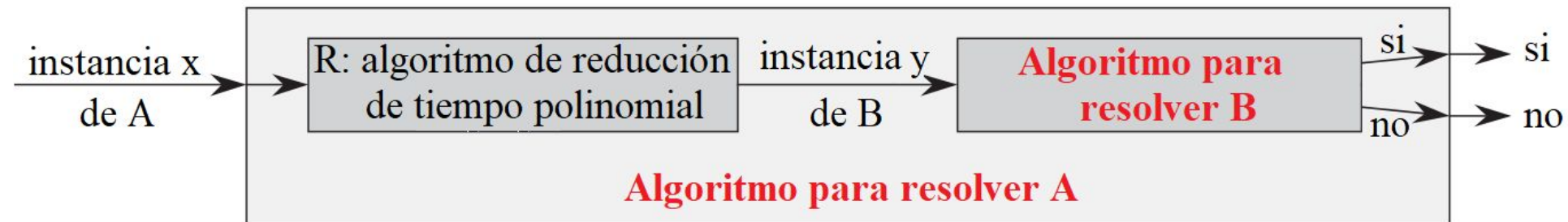
**NP-completitud trata de mostrar cuán difícil es un problema → usamos reducciones de tiempo polinomial en la forma opuesta para mostrar que un problema es NP-completo**

# Problemas $\mathcal{NP}$ -completos

¿Cómo demostramos que un problema está en NP-Completo?

## 2. Reducciones polinómicas:

\**supongamos* un problema de decisión A del que ya sabemos que ningún algoritmo de tiempo polinomial (ATP) existe.



Usaremos una prueba por contradicción para mostrar que no existe un ATP para B.

Supongamos existe un ATP que resuelve B  $\Rightarrow$  existe un ATP que resuelve A  
contradice la suposición \*

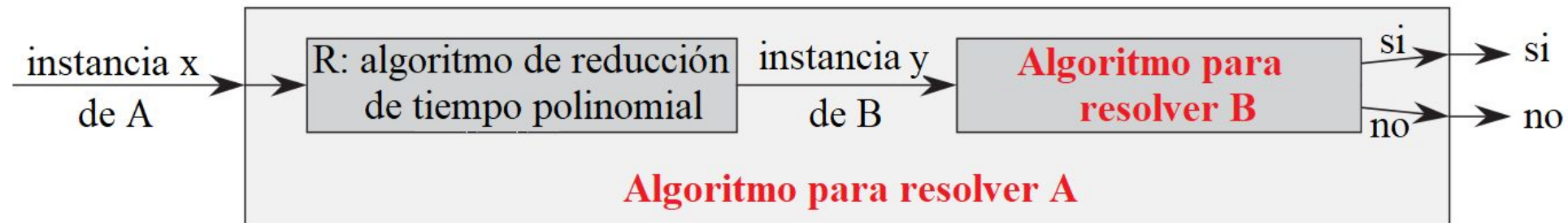


# Problemas $\mathcal{NP}$ -completos

¿Cómo demostramos que un problema está en NP-Completo?

## 2. Reducciones polinómicas:

\**supongamos* un problema de decisión A del que ya sabemos que ningún algoritmo de tiempo polinomial (ATP) existe.



*Usamos* una prueba por contradicción para mostrar que **no existe un ATP para B**.

**Para NP-completitud: no podemos asumir que no existe en absoluto ningún ATP para A.**  
**sin embargo, podemos probar que “B es tan difícil como A”**

# Problemas $\mathcal{NP}$ -completos

---

La demostración se basa en **tres conceptos claves**:

1. *problemas de decisión vs problemas de optimización*
2. *reducciones polinómicas*
3. ***el primer problema NP-completo***

# Problemas $\mathcal{NP}$ -completos

---

¿Cómo demostramos que un problema está en NP-Completo?

## 3. *El primer problema NP-completo*\*

“El problema de la satisfactibilidad” (CNF-SAT o SAT.)

Problema de decisión: **Dada una fórmula CNF\*\*, ¿existe alguna asignación de verdad que la satisfaga?**

\* La noción de NP-completitud fue propuesta en 1971 por Steven Cook quien dio la primera prueba de NP-completitud para la satisfactibilidad de la fórmula.

\*\* Una fórmula proposicional está en forma normal conjuntiva (CNF) si consiste en una sucesión de cláusulas (sucesión de literales separadas por el operador or booleano ( $\vee$ )) separadas por el operador booleano and ( $\wedge$ ). Un ejemplo de fórmula proposicional en CNF es

$(p \vee q \vee s) \wedge (q \vee r) \wedge (p \vee r) \wedge (r \vee s) \wedge (p \vee s \vee q)$ , donde  $p, q, r$  y  $s$  son variables proposicionales.



# NP completitud: Clases de problemas

---

★  $\mathcal{P}$

★  $\mathcal{NP}$

★  $\mathcal{NP}$ -Completo

# Clase $\mathcal{P}$

---

$\mathcal{P}$  es la clase de problemas de decisión que están polinomialmente acotados.

# Clase $\mathcal{NP}$

---

$\mathcal{NP}$  es la clase de problemas de decisión que pueden ser verificados por un algoritmo en tiempo polinomial.

El nombre  $\mathcal{NP}$  proviene de “**No determinista Polinomialmente acotado**”.

La clase NP fue estudiada originalmente en el contexto del no-determinismo, aquí usaremos la noción de verificación (noción equivalente pero más simple)\*

\* *“Introduction to Algorithms” . Cormen y otros*

# Clase $\mathcal{NP}$

---

## Ejemplo: Problema de decisión del *Camino*

Dado un grafo dirigido, dos vértices  $u$  y  $v$  y un entero  $k$ ,  
*¿existe un camino de  $u$  a  $v$  que consista en a lo sumo  $k$  arcos?*

Dados:

- una **instancia** dada  $\langle G, u, v, k \rangle$  del problema de decisión
- una **ruta  $p$**  de  $u$  a  $v$ .

**Podemos verificar fácilmente si  $p$  es una ruta en  $G$  y si la longitud de  $p$  es como máximo  $k$** , y si es así, podemos ver a  $p$  como un "**certificado**" de que la instancia dada pertenece al problema.

Para el problema de decisión del *Camino*, este certificado no parece proveernos mucho. Después de todo, *Camino* pertenece a  $\mathcal{P}$ , de hecho, podemos resolver *Camino* en tiempo lineal, y así verificar la pertenencia de un certificado dado lleva tanto tiempo como resolver el problema desde cero.

# Clase $\mathcal{NP}$

---

## Problema del ciclo hamiltoniano

Un ciclo hamiltoniano en un grafo no dirigido es un ciclo simple que pasa exactamente una vez por cada uno de los vértices.

**Versión de decisión:** ¿un grafo no dirigido contiene un ciclo hamiltoniano?

## ¿Cómo un algoritmo podría decidir si el grafo contiene un ciclo hamiltoniano ?

Dada una instancia del problema <Grafo  $G$ >, un algoritmo de decisión ingenuo

- lista todas las permutaciones de los vértices de  $G$  y
- chequea cada permutación para ver si es un camino hamiltoniano.

## ¿Cuál es el tiempo de corrida del algoritmo?

Hay  $n!$  permutaciones de vértices por lo tanto el tiempo de ejecución de este algoritmo ingenuo no es polinomial.

# Clase $\mathcal{NP}$

---

## Problema del ciclo hamiltoniano

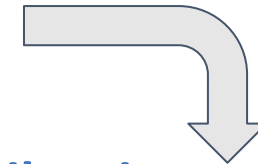
Consideremos un problema un poco más fácil:

Supongamos que un amigo nos dice que un grafo  $G$  dado es hamiltoniano, y ofrece *probarlo* dándonos los vértices en orden a lo largo del ciclo hamiltoniano.

**Es fácil verificar la prueba:**

- verificar que el ciclo es una permutación de los vértices de  $V$  y
- si cada una de las aristas consecutivas a lo largo del ciclo realmente existe en el grafo.

Este algoritmo de verificación puede ejecutarse en tiempo polinomial



**una prueba de que existe un ciclo hamiltoniano en un grafo  
se puede “verificar en tiempo polinomial”**

# Clase $\mathcal{NP}$

---

## Algoritmo de verificación $A$ :

tiene dos argumentos:

1. una instancia del problema
2. un ***certificado***.

**$A$**  verifica que el *certificado* es una solución a la *instancia* del problema.

## Ejemplo:

En el problema del ciclo hamiltoniano, el certificado es la lista de vértices del ciclo hamiltoniano.

Si el grafo es hamiltoniano, el ciclo en sí ofrece suficiente información para verificar este hecho.

# Clase $\mathcal{NP}$

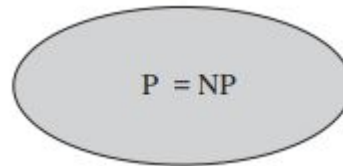
Entonces, si

$\mathcal{P}$  es la clase de problemas de decisión que están polinomialmente acotados.

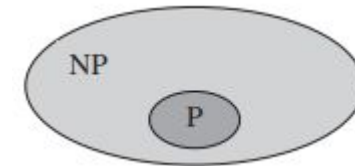
$\mathcal{NP}$  es la clase de problemas de decisión que pueden ser verificados por un algoritmo en tiempo polinomial

Si un problema está en  $\mathcal{P}$  también está en  $\mathcal{NP}$  ya que puede ser resuelto en tiempo polinomial aún sin dar un certificado  $\Rightarrow \mathcal{P} \subseteq \mathcal{NP}$

Pero, lo que no sabemos aún **¿Es  $\mathcal{P} = \mathcal{NP}$  o es  $\mathcal{P}$  un subconjunto propio de  $\mathcal{NP}$ ?**



$\mathcal{P} = \mathcal{NP}$



$\mathcal{P} \neq \mathcal{NP}$



# Clase $\mathcal{NP}$

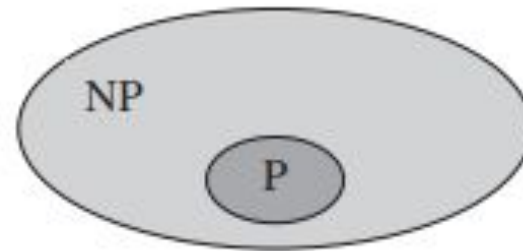
Problema aún sin resolver en ciencias de la computación:

**¿Es  $P = NP$  o es  $P$  un subconjunto propio de  $NP$ ?**

Ningún algoritmo de tiempo polinómico para ningún problema NP-completo se ha descubierto aún **¿Es posible que existan?**

Parece poco probable por los esfuerzos que los investigadores han realizado sobre estos problemas.

**Se conjetura que:**



**$P \neq NP$**

**La prueba de que un problema está en NP-completo evidencia que es intratable.**

# Clases $\mathcal{NP}$ -completo y $\mathcal{NP}$ -Hard

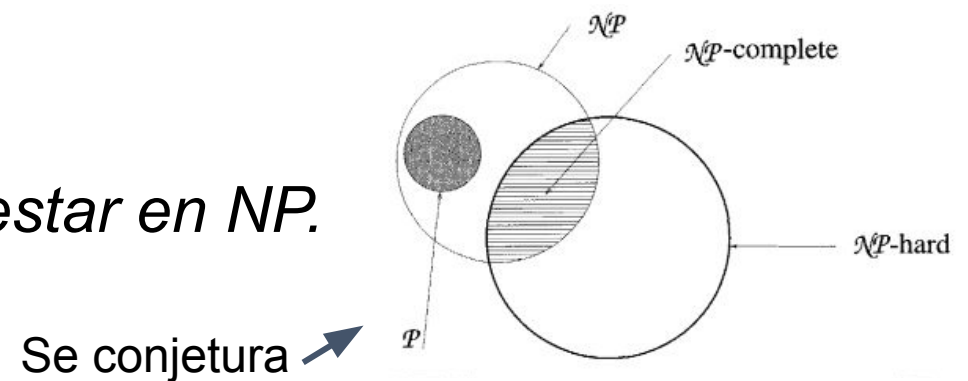
- **Definición de NP-Hard:** un problema  $Q$  está en NP-Hard si todo problema  $P$  en NP se puede reducir a  $Q$ , es decir,

$$\forall P, P \in \mathcal{NP}, P \leq_p Q.$$

significa  $\rightarrow Q$  es “al menos tan difícil como cualquier problema en NP”

- **Definición de NP-completo:** un problema  $Q$  es NP-completo si
  - está en NP-Hard
  - y está en NP

*Así, un problema puede ser NP-Hard y no estar en NP.*



# Clases $\mathcal{NP}$ -completo y $\mathcal{NP}$ -Hard

---

- Teorema:

Si  $P \leq_p Q$  y  $Q$  está en  $\mathcal{P}$ , entonces  $P$  está en  $\mathcal{P}$ .

A partir de este teorema y por las definiciones anteriores, surge el siguiente teorema:

- Teorema:

Si cualquier problema  $\mathcal{NP}$ -completo está en  $\mathcal{P}$  entonces  $\mathcal{P} = \mathcal{NP}$

Este teorema indica lo valioso que sería hallar un algoritmo polinomialmente acotado para cualquier problema en  $\mathcal{NP}$ -completo

# Clases $\mathcal{NP}$ -completo y $\mathcal{NP}$ -Hard

---

¿Cómo demostramos que un problema  $Q$  está en  $\mathcal{NP}$ -Completo?

Tenemos que demostrar que:

- $Q$  está en  $\mathcal{NP}$  y
- $\forall P, P \in \mathcal{NP}, P \leq_p Q.$   $\rightarrow$  ¡ tarea ardua !



# Clases $\mathcal{NP}$ -completo y $\mathcal{NP}$ -Hard

---

¿Cómo demostramos que un problema  $Q$  está en  $\mathcal{NP}$ -Completo?

haciendo uso de:

- la demostración del primer problema  $\mathcal{NP}$ -completo (teorema de Cook)

Satisfactibilidad es  $\mathcal{NP}$ -Completo

- de la transitividad de las reducciones polinómicas:



# Clases $\mathcal{NP}$ -completo y $\mathcal{NP}$ -Hard

---

¿Cómo demostramos que un problema  $Q$  está en  $\mathcal{NP}$ -Completo?

- Seleccionar un problema  $\mathcal{NP}$ -Completo conocido  $L$

sabemos que  $(\forall P, P \in \mathcal{NP}, P \leq_p L) \quad \text{y} \quad L \in \mathcal{NP}$

- Probar que  $L \leq_p Q$

si  $(\forall P, P \in \mathcal{NP}, P \leq_p L) \text{ y } L \leq_p Q \Rightarrow (\forall P, P \in \mathcal{NP}, P \leq_p Q)$

Reduciendo un problema conocido  $L$  a  $Q$ , implícitamente estamos reduciendo cada problema en  $\mathcal{NP}$  a  $Q$ , lo que prueba que  $Q$  está en  $\mathcal{NP}$ -hard.

- Probar que  $Q$  está en  $\mathcal{NP}$

# Clases $\mathcal{NP}$ -completos y $\mathcal{NP}$ -Hard

---

¿Cómo demostramos que el problema del **viajante de comercio** está en  $\mathcal{NP}$ -Completo?

## Problema del viajante

**Versión de decisión:** Dadas las distancias entre un conjunto de  $n$  ciudades, determinar si existe un circuito cuya distancia total sea  $\leq K$  y puede ser recorrido por un viajante que parte de una estas ciudades, visita a todas las demás exactamente una vez y vuelve al punto de partida.

# Clases $\mathcal{NP}$ -completos y $\mathcal{NP}$ -Hard

---

¿Cómo demostramos que el problema del **viajante de comercio** está en  $\mathcal{NP}$ -Completo?

- **Seleccionar un problema  $\mathcal{NP}$ -Completo conocido L**

Seleccionamos el **Problema del ciclo hamiltoniano** que sabemos que está en  $\mathcal{NP}$ -Completo

**Un ciclo hamiltoniano** en un grafo no dirigido es un ciclo simple que contiene cada vértice del grafo.

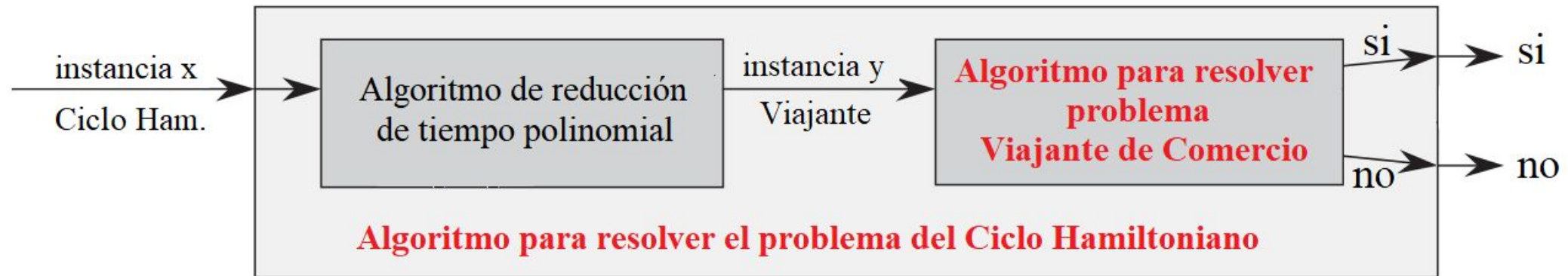
**Versión de decisión:** ¿un grafo no dirigido dado tiene un ciclo hamiltoniano?



# Clases $\mathcal{NP}$ -completos y $\mathcal{NP}$ -Hard

¿Cómo demostramos que el problema del **viajante de comercio** está en  $\mathcal{NP}$ -Completo?

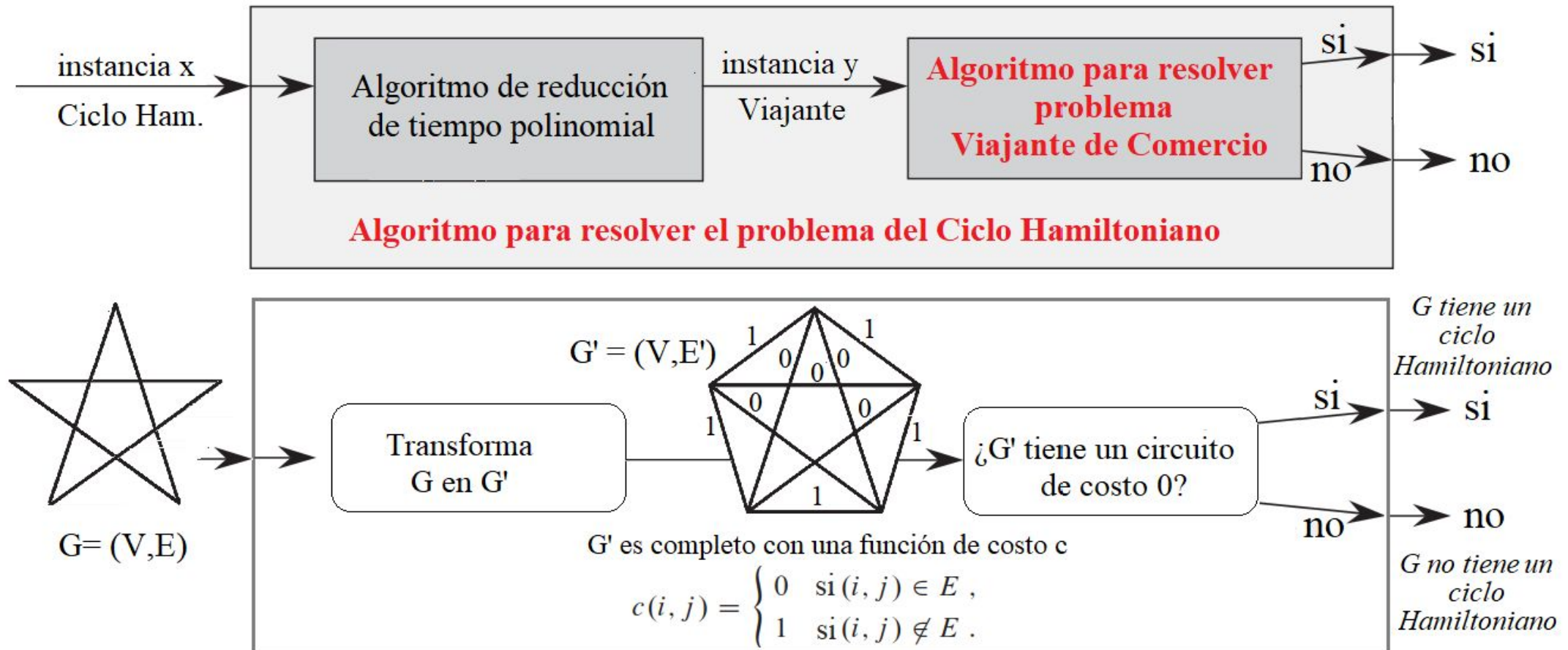
- Probar que **Prob. del ciclo Hamiltoniano**  $\leq_p$  **Prob. del Viajante**



# Clases $\mathcal{NP}$ -completos y $\mathcal{NP}$ -Hard

¿Cómo demostramos que el problema del **viajante de comercio** está en  $\mathcal{NP}$ -Completo?

- Probar que **Prob. del ciclo Hamiltoniano**  $\leq_p$  **Prob. del Viajante**



# Clases $\mathcal{NP}$ -completos y $\mathcal{NP}$ -Hard

---

¿Cómo demostramos que el problema del **viajante de comercio** está en  $\mathcal{NP}$ -Completo?


Por medio de la reducción hemos probado que el prob. del viajante está en NP-hard

Ahora...

- Probar que **el problema del Viajante de Comercio** está en NP

Dada una instancia del problema, usamos como **certificado** la secuencia de  $n$  vértices en algún circuito hamiltoniano.

El **algoritmo de verificación** chequea:

- si esta secuencia contiene cada vértice exactamente una vez
  - suma los costos de los arcos y chequea si la suma es a lo sumo  $k$ .
- 
- tiempo polinomial**

# Clases $\mathcal{NP}$ -completos y $\mathcal{NP}$ -Hard

---

¿Cómo demostramos que el problema del **viajante de comercio** está en  $\mathcal{NP}$ -Completo?

- Seleccionamos un problema NP-completo conocido (prob. del ciclo Hamiltoniano)
- Por medio de la reducción probamos que el problema del viajante de comercio está en

**NP-hard**

- Probamos que el problema del Viajante de comercio está en **NP**



**Probamos que el problema del viajante de comercio está en NP-completo**

*un algoritmo polinomial para el problema del viajante es poco probable que exista*

# Clases $\mathcal{NP}$ -completos y $\mathcal{NP}$ -Hard

## Ejemplos de problemas

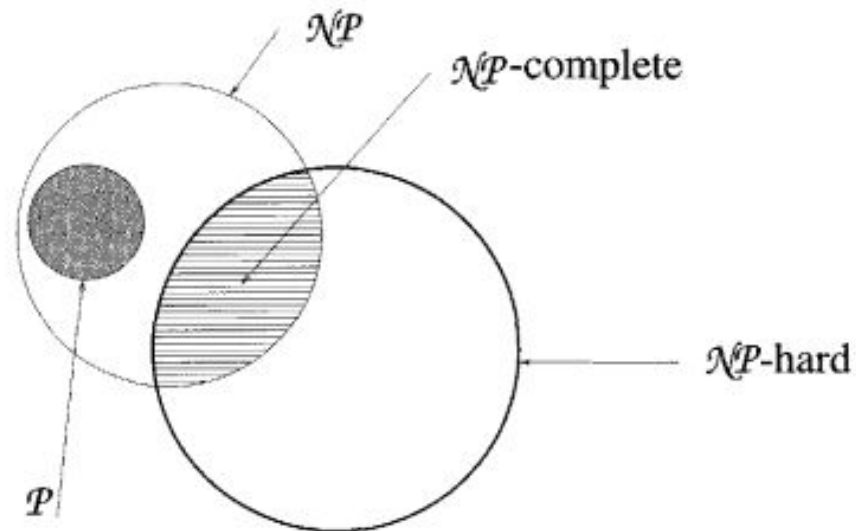
$\mathcal{P}$

búsqueda binaria  
merge sort  
recorrido de grafos  
caminos mínimos  
mochila fraccionada

## $\mathcal{NP}$ -completos

Versión de decisión de:

coloreo de grafo  
ciclo hamiltoniano  
mochila 0/1  
bin packing  
viajante  
suma de subconjuntos



Considerando que  $\mathcal{P} \neq \mathcal{NP}$

## $\mathcal{NP}$ -Hard

Versión de optimización de:

coloreo de grafo  
ciclo hamiltoniano  
mochila 0/1

...

versión de decisión de Ajedrez  
versión de decisión Torres de Hanoi

# Problemas $\mathcal{NP}$ -completos

---

## Problema 3-CNF-SAT

Dada una fórmula CNF en la que ninguna cláusula puede contener más de tres literales, ¿existe alguna asignación de verdad que la satisfaga?

## “Clique” en grafos

Dado un grafo no-orientado  $G = (V, E)$ , un “clique” es un subconjunto  $V' \subseteq V$  de vértices, tal que cada par está conectado por un arco en  $E$ . El tamaño del “clique” es la cantidad de vértices que contiene.

**Versión de decisión:** Dado un grafo no-orientado  $G = (V, E)$ , ¿existe un “clique” de  $k$  vértices?

# Problemas $\mathcal{NP}$ -completos

---

## Problema del ciclo hamiltoniano

Un ciclo hamiltoniano en un grafo no dirigido es un ciclo simple que pasa exactamente una vez por cada uno de los vértices.

**Versión de decisión:** ¿un grafo no dirigido dado tiene un ciclo hamiltoniano?

## Problema del viajante

Dadas las distancias entre un conjunto de  $n$  ciudades, determinar el circuito que debería recorrer un viajante, que parte de una estas ciudades, visita a todas las demás exactamente una vez y vuelve al punto de partida, habiendo recorrido la menor distancia posible.

**Versión de decisión:** Dadas las distancias entre un conjunto de  $n$  ciudades, determinar si existe un circuito cuya distancia total sea  $\leq K$  y puede ser recorrido por un viajante que parte de una estas ciudades, visita a todas las demás exactamente una vez y vuelve al punto de partida.

# Problemas $\mathcal{NP}$ -completos

---

## Cobertura de vértices en grafos (Vertex-Cover)

Dado un grafo no-orientado  $G = (V, E)$ , una cobertura de vértices es un subconjunto de vértices  $V'$  tal que si  $\{u, v\}$  es una arista de  $G$ , luego  $u \in V'$  o  $v \in V'$  (o ambos). El tamaño es la cantidad de vértices en la cobertura.

**Versión de decisión:** ¿existe una cobertura de  $k$  vértices?

## Problema de la suma de subconjuntos

Dados un conjunto finito  $S$  de enteros positivos y un entero  $t > 0$ , se desea determinar un subconjunto  $S' \subseteq S$  cuyos elementos sumen  $t$ .

**Versión de decisión:** ¿existe un subconjunto  $S'$  cuyos elementos sumen  $t$ ?

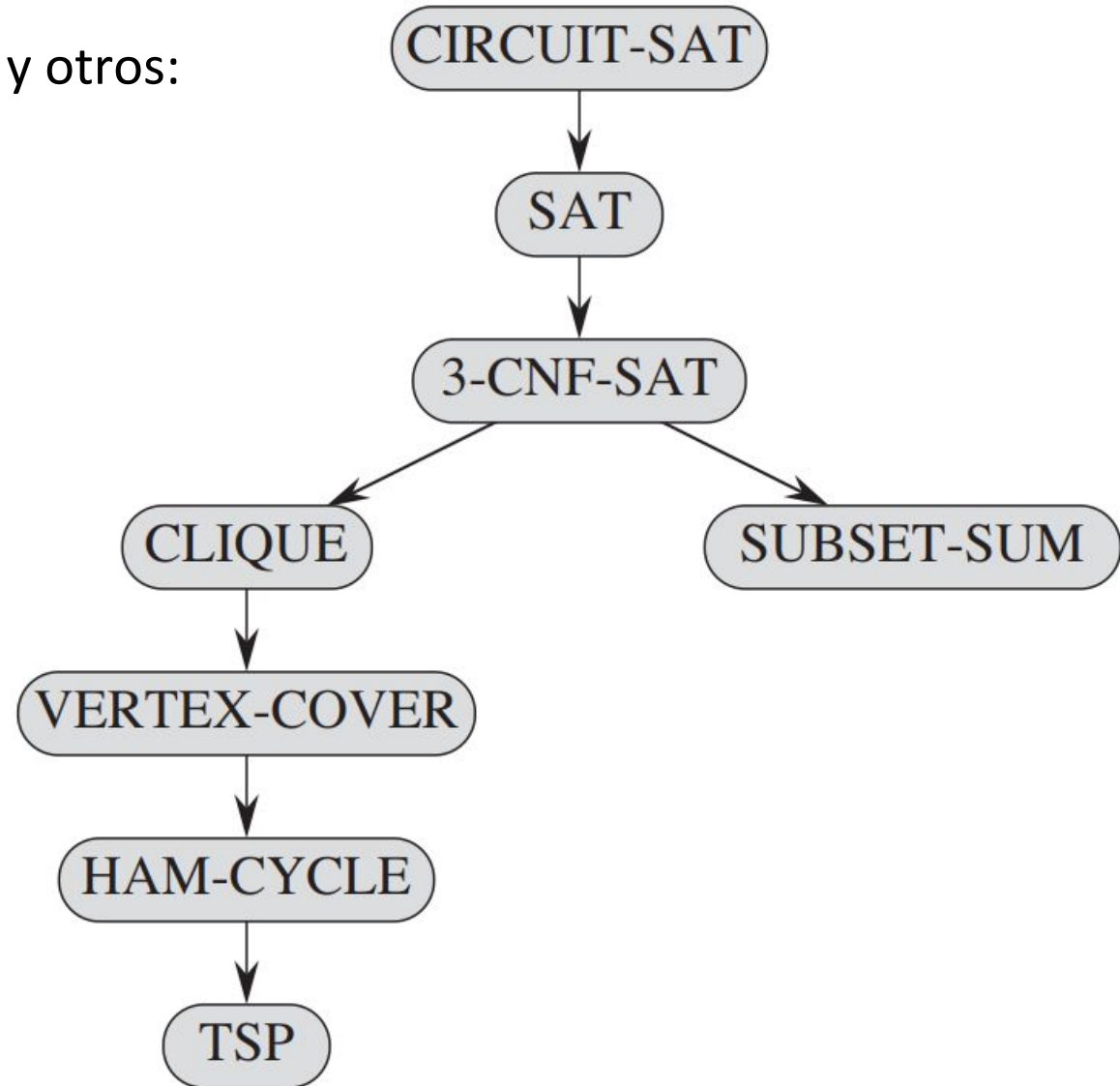


# Problemas $\mathcal{NP}$ -completos

En Introduction to Algorithms de Cormen y otros:

se demuestran estas

reducciones que prueban NP completitud



# Problemas $P$ -space

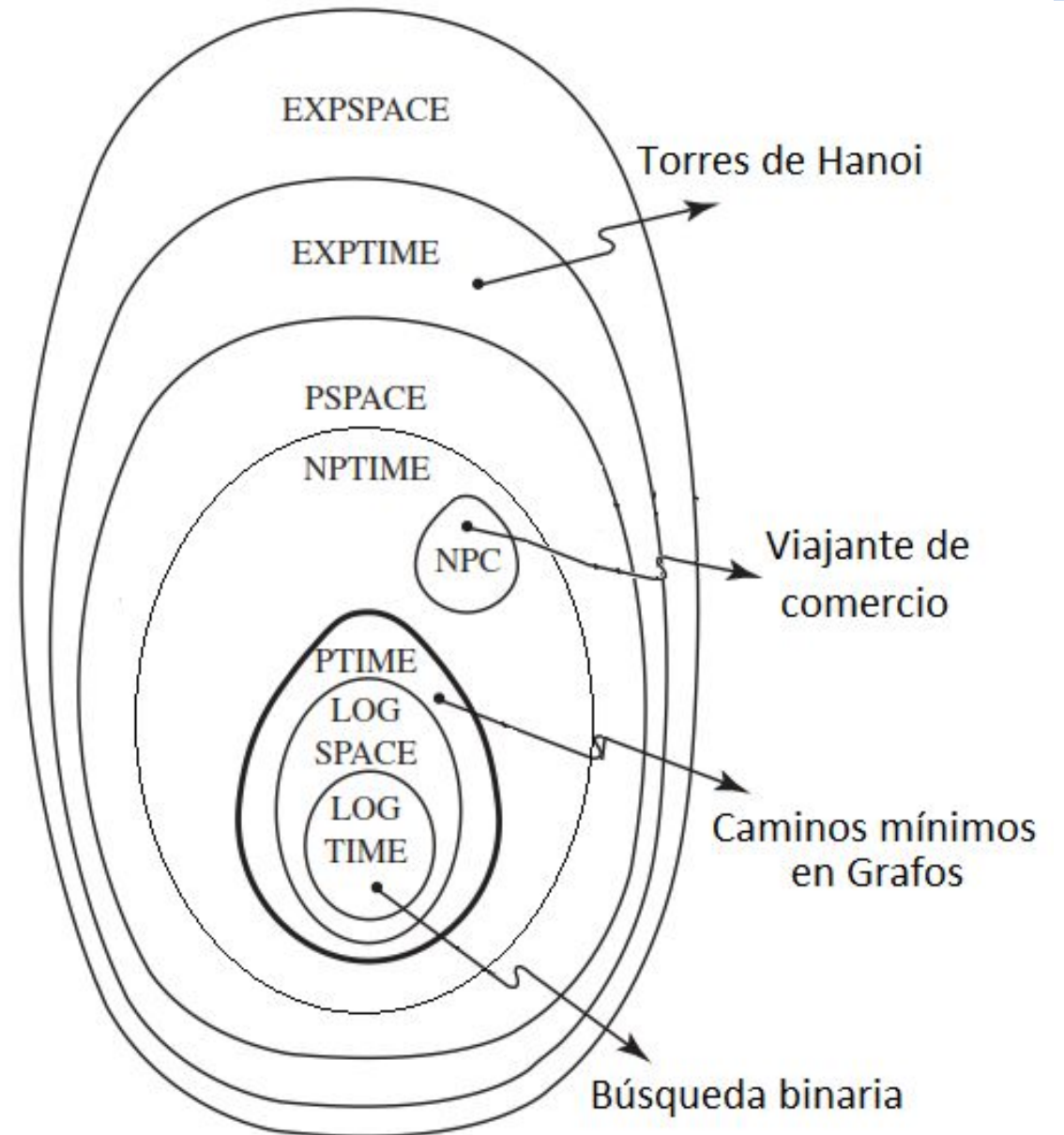
Se conjetura

## P-SPACE:

conjunto de problemas solucionables con una cantidad polinomial de espacio de memoria

## EXP-SPACE:

conjunto de problemas solucionables con una cantidad exponencial de espacio de memoria



# Algoritmos y Complejidad

---

Nuestro Objetivo



Cómo resolver problemas “difíciles”,  
inherentemente intratables  
con la tecnología actual?

# Algoritmos y Complejidad

---

Si bien no se ha podido demostrar aún que no existen algoritmos eficientes para las clases NP-completo, PSPACE completo y NP-Hard, desde el punto de vista práctico hay que resolverlos eficientemente.

- Si las **entradas son “pequeñas”** podríamos construir algoritmos basados en backtracking que realicen una **búsqueda exhaustiva**
- Sino, podríamos construir soluciones aproximadas, **“cercanas”** a la mejor solución, a partir de algoritmos polinomiales.



**Algoritmos heurísticos**

**Algoritmos aproximados**

# Algoritmos y Complejidad

---

## Algoritmo Heurístico (heurística → hallar, inventar):

Algoritmo que encuentra una buena solución no muy alejada de la óptima en un tiempo polinomial.

## Algoritmo aproximado:

Algoritmo que encuentra una solución cercana a la óptima en un tiempo polinomial y se puede medir cuán cercana está la solución dada a la óptima.

- ★ *Los algoritmos heurísticos y aproximados no garantizan encontrar la solución óptima*
- ★ *Los algoritmos aproximados establecen una cota de error.*

# Clases de Problemas y Complejidad Computacional

---

## Bibliografía

- Cormen, Leiserson, Rivest y Stein (2009). *Introduction to Algorithms* (3° Edición)
- Horowitz, Sahni y Rajasekaran (1998). *Computer Algorithms*.
- Harel, David (2004) *Algorithmics: The Spirit of Computing* (3° Edición)
- Baase y Van Gelder (2002) *Algoritmos computacionales: Introducción al análisis y diseño* (3° Edición)

# Anexo

Clase  $\mathcal{NP}$  en el contexto del no determinismo

# Anexo: Clase $\mathcal{NP}$ en el contexto del no determinismo

---

$\mathcal{NP}$  es la clase de problemas de decisión que se pueden resolver por **algoritmos no deterministas** en un tiempo polinomial.

**Algoritmo no determinista**  $\rightarrow$  algoritmo que contiene operaciones cuya salida no está definida de manera única pero está limitada a un conjunto de posibilidades

Se especifica con tres funciones de tiempo constantes  $O(1)$ :

- $\text{Choice}(S) \rightarrow$  elige arbitrariamente uno de los elementos del conjunto  $S$
- $\text{Failure}() \rightarrow$  terminación fallida
- $\text{Success}() \rightarrow$  terminación exitosa

*Un algoritmo no determinista termina sin éxito si y sólo si no existe un conjunto de alternativas que conduce a una terminación exitosa.*



# Anexo: Clase $\mathcal{NP}$ en el contexto del no determinismo

---

## Algoritmo no determinista

El **tiempo** requerido por un algoritmo no determinista está dado por el **mínimo número de pasos necesarios para alcanzar una terminación exitosa** si es que existe una secuencia de elecciones que conducen a tal terminación

# Anexo: Clase $\mathcal{NP}$ en el contexto del no determinismo

## Algoritmo no determinista

**Ejemplo:** Problema de la búsqueda de un elemento  $x$  en un conjunto  $A[1..n]$ ,  $n \geq 1$  elementos. Se requiere determinar el índice  $j$  tal que  $A[j]=x$  or  $j=0$  if  $x$  no está en  $A$ .

```
 $j := \text{Choice}(1, n);$   
if  $A[j] = x$  then {write ( $j$ ); Success();}  
write (0); Failure();
```

**Salida 0**  $\rightarrow$  si y sólo si no hay ningún  $j$  tal que  $A[j]=x$  (el elemento no existe)

## Complejidad temporal:

algoritmo no determinista  $\rightarrow O(1)$

algoritmo determinista  $\rightarrow O(n)$  ( $A$  no está ordenado)

# Anexo: Clase $\mathcal{NP}$ en el contexto del no determinismo

## Algoritmo de decisión no determinista

Ejemplo: Problema de la Satisfactibilidad

```
Algorithm Eval( $E$ ,  $n$ )  
// Determine whether the propositional formula  $E$  is  
// satisfiable. The variables are  $x_1, x_2, \dots, x_n$ .  
{  
    for  $i := 1$  to  $n$  do // Choose a truth value assignment.  
         $x_i := \text{Choice}(\text{false}, \text{true});$   
    if  $E(x_1, \dots, x_n)$  then Success();  
    else Failure();  
}
```

La terminación exitosa: *si y sólo si* la fórmula proposicional dada  $E(x_1, \dots, x_n)$  es satisfacible

**Complejidad temporal:** algoritmo no determinista  $\rightarrow$   **$O(n)$**

algoritmo determinista evaluará  $2^n$  posibilidades!!