

Teoría de la Computación

Tiempo: P y NP

Complejidad en tiempo, la clase P, verificadores y máquinas no deterministas

Gerson Garrido

Semana 12

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A

1. De «¿se puede?» a «¿cuánto cuesta?»
2. El tiempo de una máquina. TIME($t(n)$)
3. Tres máquinas para 0^k1^k
4. Varias cintas en una: $O(t(n)^2)$
5. La clase P y la codificación
6. PATH, CYK y el no determinismo

Sesión B

7. Encontrar frente a comprobar
8. Verificadores y la clase NP
9. Cinco problemas y sus certificados
10. Máquinas no deterministas: la misma clase
11. P $\subseteq$ NP $\subseteq$ EXPTIME
12. Lo que NP no es. coNP

Lectura

Apuntes de la semana 12. Apoyo: Sipser, sec. 7.1 a 7.3; Barak, cap. 13 y primera parte del 15.

SECCIÓN 01

De «¿se puede?» a «¿cuánto cuesta?»

01

De dónde venimos

- Semanas 9 a 11: la frontera entre lo decidible y lo indecidible.
- Esa frontera no mira el costo: un segundo o 10^{80} pasos, todo es “decidible”.

De dónde venimos

- Semanas 9 a 11: la frontera entre lo decidible y lo indecidible.
- Esa frontera no mira el costo: un segundo o 10^{80} pasos, todo es “decidible”.
- **Desde hoy, todas las máquinas son decididores.** La pregunta es cuántos pasos dan.

De dónde venimos

- Semanas 9 a 11: la frontera entre lo decidible y lo indecidible.
- Esa frontera no mira el costo: un segundo o 10^{80} pasos, todo es “decidible”.
- **Desde hoy, todas las máquinas son decididores.** La pregunta es cuántos pasos dan.

	En Algoritmos	**Aquí**
Qué se mide	Operaciones del pseudocódigo	Pasos de una máquina de Turing
Qué se clasifica	Algoritmos	Lenguajes
Tamaño de la entrada	Elementos, vértices	Longitud de la cadena
P y NP	Sin modelo fijo	Definidas sobre máquinas

De dónde venimos

- Semanas 9 a 11: la frontera entre lo decidible y lo indecidible.
- Esa frontera no mira el costo: un segundo o 10^{80} pasos, todo es “decidible”.
- **Desde hoy, todas las máquinas son decididores.** La pregunta es cuántos pasos dan.

	En Algoritmos	**Aquí**
Qué se mide	Operaciones del pseudocódigo	Pasos de una máquina de Turing
Qué se clasifica	Algoritmos	Lenguajes
Tamaño de la entrada	Elementos, vértices	Longitud de la cadena
P y NP	Sin modelo fijo	Definidas sobre máquinas

El cambio de punto de vista

Ya no “cuánto tarda este algoritmo”, sino “cuánto tarda el *mejor* algoritmo para este problema”.

El tiempo de una máquina

Complejidad en tiempo *(time complexity)*

Sea M una máquina de Turing determinista que se detiene con todas las entradas. El **tiempo** de M es la función $f : \mathbb{N} \to \mathbb{N}$ con

$$f(n) = \text{el máximo número de pasos de } M \text{ con una entrada de longitud } n.$$

El tiempo de una máquina

Complejidad en tiempo *(time complexity)*

Sea M una máquina de Turing determinista que se detiene con todas las entradas. El **tiempo** de M es la función $f : \mathbb{N} \to \mathbb{N}$ con

$$f(n) = \text{el máximo número de pasos de } M \text{ con una entrada de longitud } n.$$

- Un **paso** es una aplicación de δ. Contar pasos no tiene ambigüedad.

El tiempo de una máquina

Complejidad en tiempo *(time complexity)*

Sea M una máquina de Turing determinista que se detiene con todas las entradas. El **tiempo** de M es la función $f : \mathbb{N} \to \mathbb{N}$ con

$$f(n) = \text{el máximo número de pasos de } M \text{ con una entrada de longitud } n.$$

- Un **paso** es una aplicación de δ. Contar pasos no tiene ambigüedad.
- n es la **longitud de la cadena**: no el valor de un número, ni la cantidad de vértices.

El tiempo de una máquina

Complejidad en tiempo *(time complexity)*

Sea M una máquina de Turing determinista que se detiene con todas las entradas. El **tiempo** de M es la función $f : \mathbb{N} \to \mathbb{N}$ con

$$f(n) = \text{el máximo número de pasos de } M \text{ con una entrada de longitud } n.$$

- Un **paso** es una aplicación de δ. Contar pasos no tiene ambigüedad.
- n es la **longitud de la cadena**: no el valor de un número, ni la cantidad de vértices.
- Es el **peor caso** sobre las $|\Sigma|^n$ entradas de esa longitud.

El tiempo de una máquina

Complejidad en tiempo *(time complexity)*

Sea M una máquina de Turing determinista que se detiene con todas las entradas. El **tiempo** de M es la función $f : \mathbb{N} \to \mathbb{N}$ con

$$f(n) = \text{el máximo número de pasos de } M \text{ con una entrada de longitud } n.$$

- Un **paso** es una aplicación de δ. Contar pasos no tiene ambigüedad.
- n es la **longitud de la cadena**: no el valor de un número, ni la cantidad de vértices.
- Es el **peor caso** sobre las $|\Sigma|^n$ entradas de esa longitud.
- Solo tiene sentido para **decididores**.

El tiempo de una máquina

Complejidad en tiempo *(time complexity)*

Sea M una máquina de Turing determinista que se detiene con todas las entradas. El **tiempo** de M es la función $f : \mathbb{N} \to \mathbb{N}$ con

$$f(n) = \text{el máximo número de pasos de } M \text{ con una entrada de longitud } n.$$

- Un **paso** es una aplicación de δ. Contar pasos no tiene ambigüedad.
- n es la **longitud de la cadena**: no el valor de un número, ni la cantidad de vértices.
- Es el **peor caso** sobre las $|\Sigma|^n$ entradas de esa longitud.
- Solo tiene sentido para **decididores**.

Notación de Algoritmos: $f(n) = O(g(n))$ si $f(n) \leq c \cdot g(n)$ para todo $n \geq n_0$. Y $2^{O(g(n))}$ es "a lo sumo $2^{c \cdot g(n)}$".

Clases de tiempo

TIME($t(n)$)

El conjunto de los lenguajes que decide alguna máquina de Turing **determinista**, de **una cinta**, que corre en tiempo $O(t(n))$.

- $\text{TIME}(n) \subseteq \text{TIME}(n^2) \subseteq \text{TIME}(n^3) \subseteq \cdots$

Clases de tiempo

TIME($t(n)$)

El conjunto de los lenguajes que decide alguna máquina de Turing **determinista**, de **una cinta**, que corre en tiempo $O(t(n))$.

- $\text{TIME}(n) \subseteq \text{TIME}(n^2) \subseteq \text{TIME}(n^3) \subseteq \cdots$
- Para mostrar que $L \in \text{TIME}(t(n))$: basta **una** máquina.

Clases de tiempo

TIME($t(n)$)

El conjunto de los lenguajes que decide alguna máquina de Turing **determinista**, de **una cinta**, que corre en tiempo $O(t(n))$.

- $\text{TIME}(n) \subseteq \text{TIME}(n^2) \subseteq \text{TIME}(n^3) \subseteq \cdots$
- Para mostrar que $L \in \text{TIME}(t(n))$: basta **una** máquina.
- Para mostrar que $L \notin \text{TIME}(t(n))$: hay que descartarlas **todas**.

Idea clave

$\text{TIME}(t(n))$ es una clase de **lenguajes**, no de máquinas. Casi todas las preguntas abiertas de esta parte del curso son del segundo tipo.

SECCIÓN 02

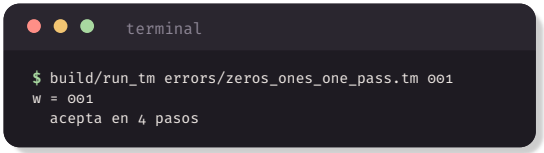
Tres máquinas para 0^k1^k

02

Un lenguaje, tres máquinas

$$A = \{ 0^k 1^k : k \geq 0 \}$$

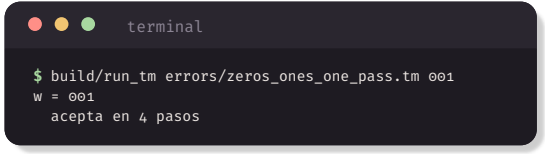
- No es regular. La máquina de la semana 7 lo decide. ¿En cuánto tiempo?



Un lenguaje, tres máquinas

$$A = \{\, 0^k 1^k : k \geq 0 \,\}$$

- No es regular. La máquina de la semana 7 lo decide. ¿En cuánto tiempo?
- Lo más rápido imaginable: leer una sola vez, $n + 1$ pasos.



Un lenguaje, tres máquinas

$$A = \{ 0^k 1^k : k \geq 0 \}$$

- No es regular. La máquina de la semana 7 lo decide. ¿En cuánto tiempo?
- Lo más rápido imaginable: leer una sola vez, $n + 1$ pasos.
- No funciona: una máquina que solo avanza es un autómata finito, y no puede comparar dos cantidades sin tope.

```
terminal

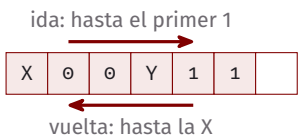
$ build/run_tm errors/zeros_ones_one_pass.tm 001
w = 001
  acepta en 4 pasos
```

Acepta 001

Solo comprueba la forma 0^*1^*. Hay que volver sobre la cinta.

M_1: emparejar de uno en uno

1. Cambiar el primer `0` por `X`.
2. Avanzar hasta el primer `1` y cambiarlo por `Y`.
3. Volver hasta la `X` y repetir.
4. Sin ceros: comprobar que solo quedan `Y`.



Cada ida y vuelta elimina **un** cero y **un** uno.

terminal

```
$ build/run_tm machines/zeros_ones_quadratic.tm \
    --watch '*' 0011
w = 0011
   [q0] 0011
  X [q1] 011
  X0 [q1] 11
  X [q2] 0Y1
   [q2] X0Y1
  X [q0] 0Y1
  XX [q1] Y1
  XXY [q1] 1
  XX [q2] YY
  X [q2] XYY
  XX [q0] YY
  XXY [q3] Y
  XXYY [q3] _
  XXYY_ [q_accept] _
  acepta en 13 pasos
```

M_1 corre en tiempo $O(n^2)$

Con $\texttt{0}^k\texttt{1}^k$, contando a mano:

- Cada ronda: marcar el $\texttt{0}$ (1), avanzar $k-1$ celdas, marcar el $\texttt{1}$ (1), volver $k-1$, leer la $\texttt{X}$ (1). Son $2k+1$ pasos.

M_1 corre en tiempo $O(n^2)$

Con $\texttt{0}^k\texttt{1}^k$, contando a mano:

- Cada ronda: marcar el $\texttt{0}$ (1), avanzar $k-1$ celdas, marcar el $\texttt{1}$ (1), volver $k-1$, leer la $\texttt{X}$ (1). Son $2k+1$ pasos.
- k rondas, y al final $k+1$ pasos sobre las $\texttt{Y}$:

$$k(2k+1)+(k+1) = 2k^2+2k+1 = \tfrac{n^2}{2}+n+1 \qquad (n=2k)$$

M_1 corre en tiempo $O(n^2)$

Con $\texttt{0}^k\texttt{1}^k$, contando a mano:

- Cada ronda: marcar el $\texttt{0}$ (1), avanzar $k-1$ celdas, marcar el $\texttt{1}$ (1), volver $k-1$, leer la $\texttt{X}$ (1). Son $2k+1$ pasos.
- k rondas, y al final $k+1$ pasos sobre las $\texttt{Y}$:

$$k(2k+1) + (k+1) = 2k^2 + 2k + 1 = \tfrac{n^2}{2} + n + 1 \qquad (n = 2k)$$

- Con $k = 2$: $8 + 4 + 1 = 13$. Lo que imprimió el simulador.

M_1 corre en tiempo $O(n^2)$

Con $\texttt{0}^k\texttt{1}^k$, contando a mano:

- Cada ronda: marcar el 0 (1), avanzar $k-1$ celdas, marcar el 1 (1), volver $k-1$, leer la X (1). Son $2k+1$ pasos.
- k rondas, y al final $k+1$ pasos sobre las Y:

$$k(2k+1) + (k+1) = 2k^2 + 2k + 1 = \frac{n^2}{2} + n + 1 \qquad (n = 2k)$$

- Con $k = 2$: $8 + 4 + 1 = 13$. Lo que imprimió el simulador.

Con cualquier entrada de longitud n:

- Cada ronda cambia un 0 por X: a lo sumo n rondas.
- Cada ronda solo avanza y después solo retrocede: a lo sumo $2n + 3$ pasos.

M_1 corre en tiempo $O(n^2)$

Con $\texttt{0}^k\texttt{1}^k$, contando a mano:

- Cada ronda: marcar el 0 (1), avanzar $k-1$ celdas, marcar el 1 (1), volver $k-1$, leer la X (1). Son $2k+1$ pasos.
- k rondas, y al final $k+1$ pasos sobre las Y:

$$k(2k+1) + (k+1) = 2k^2 + 2k + 1 = \tfrac{n^2}{2} + n + 1 \qquad (n = 2k)$$

- Con $k = 2$: $8 + 4 + 1 = 13$. Lo que imprimió el simulador.

Con cualquier entrada de longitud n:

- Cada ronda cambia un 0 por X: a lo sumo n rondas.
- Cada ronda solo avanza y después solo retrocede: a lo sumo $2n + 3$ pasos.

Conclusión

$A \in \mathrm{TIME}(n^2)$. Y esta máquina no baja de $n^2/2$: es cuadrática, ni más ni menos.

M_2: tachar uno de cada dos

Quedan z ceros y u unos. Comparar sin emparejar:

$$z = u \iff z \text{ y } u \text{ tienen la misma paridad, y } \lfloor z/2 \rfloor = \lfloor u/2 \rfloor$$

M_2: tachar uno de cada dos

Quedan z ceros y u unos. Comparar sin emparejar:

$$z = u \iff z \text{ y } u \text{ tienen la misma paridad, y } \lfloor z/2 \rfloor = \lfloor u/2 \rfloor$$

- La paridad se calcula en una pasada, con el estado.

M_2: tachar uno de cada dos

Quedan z ceros y u unos. Comparar sin emparejar:

$$z = u \iff z \text{ y } u \text{ tienen la misma paridad, y } \lfloor z/2 \rfloor = \lfloor u/2 \rfloor$$

- La paridad se calcula en una pasada, con el estado.
- Pasar de z a $\lfloor z/2 \rfloor$: tachar un cero sí y uno no, empezando por el primero.

M_2: tachar uno de cada dos

Quedan z ceros y u unos. Comparar sin emparejar:

$$z = u \iff z \text{ y } u \text{ tienen la misma paridad, y } \lfloor z/2 \rfloor = \lfloor u/2 \rfloor$$

- La paridad se calcula en una pasada, con el estado.
- Pasar de z a $\lfloor z/2 \rfloor$: tachar un cero sí y uno no, empezando por el primero.
- Repetir con las mitades. Si no queda nada, aceptar.

Con 5 ceros y 5 unos quedan 2 y 2, después 1 y 1, después nada.

M_2: tachar uno de cada dos

Quedan z ceros y u unos. Comparar sin emparejar:

$$z = u \iff z \text{ y } u \text{ tienen la misma paridad, y } \lfloor z/2 \rfloor = \lfloor u/2 \rfloor$$

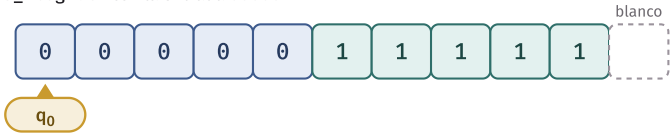
- La paridad se calcula en una pasada, con el estado.
- Pasar de z a $\lfloor z/2 \rfloor$: tachar un cero sí y uno no, empezando por el primero.
- Repetir con las mitades. Si no queda nada, aceptar.

Con 5 ceros y 5 unos quedan 2 y 2, después 1 y 1, después nada.

Lo que hace sin saberlo. Paridades vistas: impar, par, impar. Al revés, `101`: el 5 en binario. Compara las dos cantidades bit por bit.

M_2, pasada por pasada

`zeros_ones_nlogn.tm` **con la entrada** `0000011111` paso 0 de 71



El estado recuerda: nada todavía; en el primer paso marca el borde con S

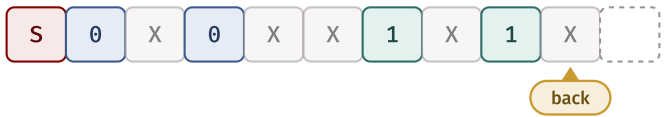
pasada	ceros sin tachar	unos sin tachar	paridades	después de tachar
1	5	5		

Cada pasada recorre la cinta y tacha un símbolo sí y uno no, empezando por el primero de cada grupo.

El estado lleva la paridad de los ceros y la de los unos que estaban sin tachar.

M_2, pasada por pasada

`zeros_ones_nlogn.tm` **con la entrada** `0000011111` paso 11 de 71



El estado recuerda: vuelve al borde izquierdo

pasada	ceros sin tachar	unos sin tachar	paridades	después de tachar
1	5	5	impar e impar	**quedan 2 y 2**

Pasada 1: llegó al blanco con las dos paridades iguales (impar), así que sigue.

De 5 y 5 quedan 2 y 2: la mitad, redondeada hacia abajo. Vuelve al borde izquierdo.

M_2, pasada por pasada

`zeros_ones_nlogn.tm` **con la entrada** `0000011111` paso 31 de 71



El estado recuerda: vuelve al borde izquierdo

pasada	ceros sin tachar	unos sin tachar	paridades	después de tachar
1	5	5	impar e impar	**quedan 2 y 2**
2	2	2	par y par	**quedan 1 y 1**

Pasada 2: llegó al blanco con las dos paridades iguales (par), así que sigue.

De 2 y 2 quedan 1 y 1: la mitad, redondeada hacia abajo. Vuelve al borde izquierdo.

M_2, pasada por pasada

`zeros_ones_nlogn.tm` **con la entrada** `0000011111` paso 51 de 71



El estado recuerda: vuelve al borde izquierdo

pasada	ceros sin tachar	unos sin tachar	paridades	después de tachar
1	5	5	impar e impar	**quedan 2 y 2**
2	2	2	par y par	**quedan 1 y 1**
3	1	1	impar e impar	**quedan 0 y 0**

Pasada 3: llegó al blanco con las dos paridades iguales (impar), así que sigue.

De 1 y 1 quedan 0 y 0: la mitad, redondeada hacia abajo. Vuelve al borde izquierdo.

M_2, pasada por pasada

`zeros_ones_nlogn.tm` **con la entrada** `0000011111` paso 71 de 71



La máquina se detuvo en el estado de aceptación.

pasada	ceros sin tachar	unos sin tachar	paridades	después de tachar
1	5	5	impar e impar	**quedan 2 y 2**
2	2	2	par y par	**quedan 1 y 1**
3	1	1	impar e impar	**quedan 0 y 0**
4	0	0	no hay nada	**acepta**

Pasada 4: no queda ningún símbolo sin tachar. La máquina acepta en 71 pasos.

Paridades de las tres pasadas: impar, par, impar. Leídas al revés, `101`: el 5 en binario.

M_2 en el simulador

terminal

```
$ build/run_tm machines/zeros_ones_nlogn.tm \
    --watch n 0000011111
w = 0000011111
  S [n] 0X0XX1X1X
  S [n] XX0XXXX1X
  S [n] XXXXXXXXX
  acepta en 71 pasos
$ build/run_tm machines/zeros_ones_nlogn.tm \
    --watch n 00000111
w = 00000111
  S [n] 0X0XX1X
  rechaza en 25 pasos
```

- 9 estados más los dos finales: `machines/zeros_ones_nlogn.tm`.
- La primera celda se marca con S: cuenta como tachada y señala el borde.

M_2 en el simulador

terminal

```
$ build/run_tm machines/zeros_ones_nlogn.tm \
    --watch n 0000011111
w = 0000011111
  S [n] 0X0XX1X1X
  S [n] XX0XXXX1X
  S [n] XXXXXXXXX
  acepta en 71 pasos
$ build/run_tm machines/zeros_ones_nlogn.tm \
    --watch n 00000111
w = 00000111
  S [n] 0X0XX1X
  rechaza en 25 pasos
```

- 9 estados más los dos finales: `machines/zeros_ones_nlogn.tm`.
- La primera celda se marca con S: cuenta como tachada y señala el borde.
- Con 5 y 3: la primera pasada ve impar e impar, y sigue. La segunda ve 2 y 1: rechaza.

M_2 en el simulador

terminal

```
$ build/run_tm machines/zeros_ones_nlogn.tm \
    --watch n 0000011111
w = 0000011111
  S [n] 0X0XX1X1X
  S [n] XX0XXXX1X
  S [n] XXXXXXXXX
  acepta en 71 pasos
$ build/run_tm machines/zeros_ones_nlogn.tm \
    --watch n 00000111
w = 00000111
  S [n] 0X0XX1X
  rechaza en 25 pasos
```

- 9 estados más los dos finales: `machines/zeros_ones_nlogn.tm`.
- La primera celda se marca con `S`: cuenta como tachada y señala el borde.
- Con 5 y 3: la primera pasada ve impar e impar, y sigue. La segunda ve 2 y 1: rechaza.
- Un `0` después de un `1` se detecta en la primera pasada.

M_2 corre en tiempo $O(n \log n)$

- Una pasada completa: ir hasta el blanco y volver. A lo sumo $2n + 1$ pasos.

M_2 **corre en tiempo** $O(n \log n)$

- Una pasada completa: ir hasta el blanco y volver. A lo sumo $2n + 1$ pasos.
- Después de j pasadas quedan $\lfloor r/2^j \rfloor$ símbolos de la cantidad mayor $r \leq n$. Eso es 0 cuando $j = \lfloor \log_2 r \rfloor + 1$.

M_2 corre en tiempo $O(n \log n)$

- Una pasada completa: ir hasta el blanco y volver. A lo sumo $2n + 1$ pasos.
- Después de j pasadas quedan $\lfloor r/2^j \rfloor$ símbolos de la cantidad mayor $r \leq n$. Eso es 0 cuando $j = \lfloor \log_2 r \rfloor + 1$.
- Una pasada más, la que no encuentra nada: a lo sumo $\lfloor \log_2 n \rfloor + 2$ pasadas.

$$(2n + 1)(\lfloor \log_2 n \rfloor + 2) = O(n \log n)$$

M_2 corre en tiempo $O(n \log n)$

- Una pasada completa: ir hasta el blanco y volver. A lo sumo $2n + 1$ pasos.
- Después de j pasadas quedan $\lfloor r/2^j \rfloor$ símbolos de la cantidad mayor $r \leq n$. Eso es 0 cuando $j = \lfloor \log_2 r \rfloor + 1$.
- Una pasada más, la que no encuentra nada: a lo sumo $\lfloor \log_2 n \rfloor + 2$ pasadas.

$$(2n + 1)(\lfloor \log_2 n \rfloor + 2) = O(n \log n)$$

Por qué es correcta: inducción sobre los símbolos sin tachar. Si las paridades difieren, $z \neq u$. Si coinciden, $z = u$ equivale a $\lfloor z/2 \rfloor = \lfloor u/2 \rfloor$, que es lo que decide el resto de la ejecución.

M_2 corre en tiempo $O(n \log n)$

- Una pasada completa: ir hasta el blanco y volver. A lo sumo $2n + 1$ pasos.
- Después de j pasadas quedan $\lfloor r/2^j \rfloor$ símbolos de la cantidad mayor $r \leq n$. Eso es 0 cuando $j = \lfloor \log_2 r \rfloor + 1$.
- Una pasada más, la que no encuentra nada: a lo sumo $\lfloor \log_2 n \rfloor + 2$ pasadas.

$$(2n + 1)(\lfloor \log_2 n \rfloor + 2) = O(n \log n)$$

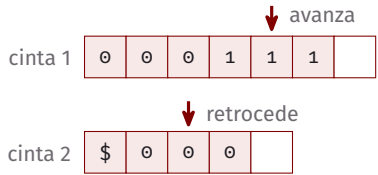
Por qué es correcta: inducción sobre los símbolos sin tachar. Si las paridades difieren, $z \neq u$. Si coinciden, $z = u$ equivale a $\lfloor z/2 \rfloor = \lfloor u/2 \rfloor$, que es lo que decide el resto de la ejecución.

No se puede mejorar con una cinta

Todo lenguaje que una máquina de una cinta decide en tiempo $o(n \log n)$ es regular. A no lo es.

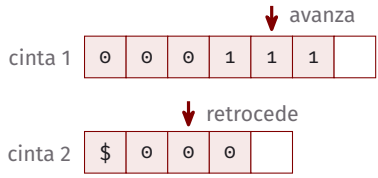
M_3: con dos cintas, tiempo lineal

1. Marcar el inicio de la cinta 2 con `$`.
2. Por cada `0` de la cinta 1, escribir un `0` en la cinta 2.
3. Por cada `1` de la cinta 1, retroceder un `0` en la cinta 2.
4. Aceptar si la cinta 1 llega al blanco justo cuando la cinta 2 vuelve a `$`.



M_3: con dos cintas, tiempo lineal

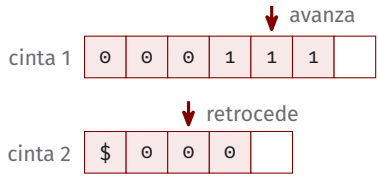
1. Marcar el inicio de la cinta 2 con `$`.
2. Por cada `0` de la cinta 1, escribir un `0` en la cinta 2.
3. Por cada `1` de la cinta 1, retroceder un `0` en la cinta 2.
4. Aceptar si la cinta 1 llega al blanco justo cuando la cinta 2 vuelve a `$`.



- La cinta 2 es la pila del autómata de la semana 6.
- Nadie cruza la entrada de ida y vuelta: a lo sumo $n + 3$ pasos. Tiempo $O(n)$.

M_3: con dos cintas, tiempo lineal

1. Marcar el inicio de la cinta 2 con `$`.
2. Por cada `0` de la cinta 1, escribir un `0` en la cinta 2.
3. Por cada `1` de la cinta 1, retroceder un `0` en la cinta 2.
4. Aceptar si la cinta 1 llega al blanco justo cuando la cinta 2 vuelve a `$`.



- La cinta 2 es la pila del autómata de la semana 6.
- Nadie cruza la entrada de ida y vuelta: a lo sumo $n + 3$ pasos. Tiempo $O(n)$.

El modelo cambia el costo

Con una cinta, $n \log n$ es lo mejor. Con dos, n. En computabilidad esto no pasaba.

El peor caso, por definición

terminal

```
$ build/count_steps
Peor caso sobre las 2^n entradas de longitud n: pasos, y la primera entrada que los alcanza
 n   cuadrática            n log n              dos cintas
 2    5  01                 7  00                5  01
 3    8  001                4  000               6  001
 4   13  0011              21  0000              7  0001
 5   18  00011              6  00000             8  00001
 6   25  000111            31  000001            9  000001
 7   32  0000111            8  0000000          10  0000001
 8   41  00001111          57  00000000         11  00000001
 9   50  000001111         10  000000000        12  000000001
10   61  0000011111        71  0000000001       13  0000000001
```

Filas 2 a 10 de la tabla; el programa imprime de 0 a 12.

El peor caso, por definición

terminal

```
$ build/count_steps
Peor caso sobre las 2^n entradas de longitud n: pasos, y la primera entrada que los alcanza
 n   cuadrática            n log n              dos cintas
 2    5  01                 7  00                5  01
 3    8  001                4  000               6  001
 4   13  0011              21  0000              7  0001
 5   18  00011              6  00000             8  00001
 6   25  000111            31  000001            9  000001
 7   32  0000111            8  0000000          10  0000001
 8   41  00001111          57  00000000         11  00000001
 9   50  000001111         10  000000000        12  000000001
10   61  0000011111        71  0000000001       13  0000000001
```

Filas 2 a 10 de la tabla; el programa imprime de 0 a 12.

- El peor caso no siempre está en el lenguaje: `00011` se rechaza recién en la tercera ronda.

El peor caso, por definición

terminal

```
$ build/count_steps
Peor caso sobre las 2^n entradas de longitud n: pasos, y la primera entrada que los alcanza
 n   cuadrática            n log n              dos cintas
 2    5  01                 7  00                5  01
 3    8  001                4  000               6  001
 4   13  0011              21  0000              7  0001
 5   18  00011              6  00000             8  00001
 6   25  000111            31  000001            9  000001
 7   32  0000111            8  0000000          10  0000001
 8   41  00001111          57  00000000         11  00000001
 9   50  000001111         10  000000000        12  000000001
10   61  0000011111        71  0000000001       13  0000000001
```

Filas 2 a 10 de la tabla; el programa imprime de 0 a 12.

- El peor caso no siempre está en el lenguaje: `00011` se rechaza recién en la tercera ronda.
- f no tiene por qué crecer: con n impar, M_2 rechaza en la primera pasada. La O es una cota superior.

Pasos contados, con 0^k1^k

terminal

```
Pasos con la entrada 0^k 1^k, de longitud n = 2k
    n   cuadrática   pasos/n^2     n log n   pasos/(n·log2 n)   dos cintas
   16          145       0.566         145              2.266           19
   64         2113       0.516         833              2.169           67
  256        33025       0.504        4353              2.125          259
 1024       525313       0.501       21505              2.100         1027
 4096      8392705       0.500      102401              2.083         4099
 8192     33562625       0.500      221185              2.077         8195
```

Seis de las diez filas que imprime `build/count_steps`. Son pasos, no segundos: salen iguales en cualquier computadora.

Pasos contados, con 0^k1^k

terminal

```
Pasos con la entrada 0^k 1^k, de longitud n = 2k
    n   cuadrática   pasos/n^2     n log n   pasos/(n·log2 n)   dos cintas
   16          145       0.566         145              2.266           19
   64         2113       0.516         833              2.169           67
  256        33025       0.504        4353              2.125          259
 1024       525313       0.501       21505              2.100         1027
 4096      8392705       0.500      102401              2.083         4099
 8192     33562625       0.500      221185              2.077         8195
```

Seis de las diez filas que imprime `build/count_steps`. Son pasos, no segundos: salen iguales en cualquier computadora.

- Con $n = 16$ empatan. Con $n = 8192$: 33 millones contra 221 mil.

Pasos contados, con 0^k1^k

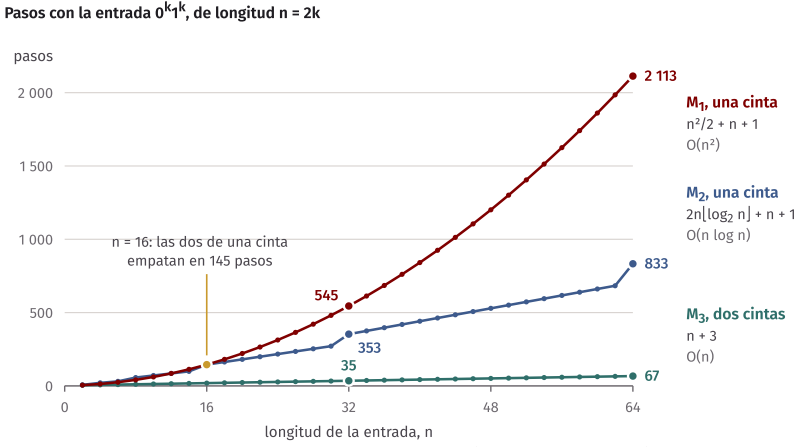
terminal

```
Pasos con la entrada 0^k 1^k, de longitud n = 2k
    n   cuadrática   pasos/n^2    n log n   pasos/(n·log2 n)   dos cintas
   16          145       0.566        145              2.266           19
   64         2113       0.516        833              2.169           67
  256        33025       0.504       4353              2.125          259
 1024       525313       0.501      21505              2.100         1027
 4096      8392705       0.500     102401              2.083         4099
 8192     33562625       0.500     221185              2.077         8195
```

Seis de las diez filas que imprime `build/count_steps`. Son pasos, no segundos: salen iguales en cualquier computadora.

- Con $n = 16$ empatan. Con $n = 8192$: 33 millones contra 221 mil.
- Las fórmulas exactas: $\frac{n^2}{2} + n + 1$, $\quad 2n\lfloor \log_2 n \rfloor + n + 1$, $\quad n + 3$.

Los mismos pasos, dibujados



La curva de M_2 da un salto en cada potencia de 2: ahí hace falta una pasada más.

SECCIÓN 03

El modelo importa, pero solo polinomialmente

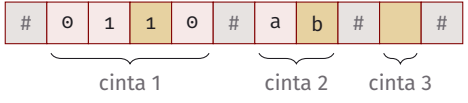
03

Varias cintas en una

Teorema

Sea $t(n) \geq n$. Toda máquina de varias cintas de tiempo $t(n)$ tiene una equivalente de **una** cinta de tiempo $O(t(n)^2)$.

La simulación de la semana 7: los k tramos en una cinta, separados por $\#$, con una marca por cabezal.

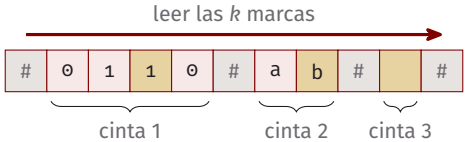


Varias cintas en una

Teorema

Sea $t(n) \geq n$. Toda máquina de varias cintas de tiempo $t(n)$ tiene una equivalente de **una** cinta de tiempo $O(t(n)^2)$.

La simulación de la semana 7: los k tramos en una cinta, separados por $\#$, con una marca por cabezal.

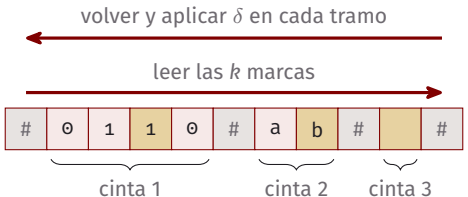


Varias cintas en una

Teorema

Sea $t(n) \geq n$. Toda máquina de varias cintas de tiempo $t(n)$ tiene una equivalente de **una** cinta de tiempo $O(t(n)^2)$.

La simulación de la semana 7: los k tramos en una cinta, separados por $\#$, con una marca por cabezal.

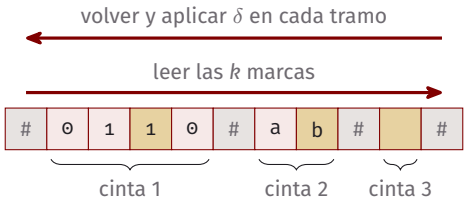


Varias cintas en una

Teorema

Sea $t(n) \geq n$. Toda máquina de varias cintas de tiempo $t(n)$ tiene una equivalente de **una** cinta de tiempo $O(t(n)^2)$.

La simulación de la semana 7: los k tramos en una cinta, separados por $\#$, con una marca por cabezal.



Si una marca se sale de su tramo, se abre un hueco desplazando todo lo que sigue una celda.

De dónde sale el cuadrado

1. En $t(n)$ pasos, cada cabezal de M se aleja a lo sumo $t(n)$ celdas.
   Cada tramo mide a lo sumo $t(n) + 1$: la parte usada de la cinta de S mide $O(t(n))$.

De dónde sale el cuadrado

1. En $t(n)$ pasos, cada cabezal de M se aleja a lo sumo $t(n)$ celdas.
   Cada tramo mide a lo sumo $t(n) + 1$: la parte usada de la cinta de S mide $O(t(n))$.
2. Dar el formato inicial: $O(n + k) = O(t(n))$, porque $n \leq t(n)$.

De dónde sale el cuadrado

1. En $t(n)$ pasos, cada cabezal de M se aleja a lo sumo $t(n)$ celdas.
   Cada tramo mide a lo sumo $t(n) + 1$: la parte usada de la cinta de S mide $O(t(n))$.
2. Dar el formato inicial: $O(n + k) = O(t(n))$, porque $n \leq t(n)$.
3. Simular **un** paso de M: dos recorridos y a lo sumo k desplazamientos, cada uno $O(t(n))$.

De dónde sale el cuadrado

1. En $t(n)$ pasos, cada cabezal de M se aleja a lo sumo $t(n)$ celdas.
   Cada tramo mide a lo sumo $t(n) + 1$: la parte usada de la cinta de S mide $O(t(n))$.
2. Dar el formato inicial: $O(n + k) = O(t(n))$, porque $n \leq t(n)$.
3. Simular **un** paso de M: dos recorridos y a lo sumo k desplazamientos, cada uno $O(t(n))$.
4. M da a lo sumo $t(n)$ pasos.

$$O(t(n)) + t(n) \cdot O(t(n)) = O(t(n)^2)$$

De dónde sale el cuadrado

1. En $t(n)$ pasos, cada cabezal de M se aleja a lo sumo $t(n)$ celdas.
   Cada tramo mide a lo sumo $t(n)+1$: la parte usada de la cinta de S mide $O(t(n))$.
2. Dar el formato inicial: $O(n+k) = O(t(n))$, porque $n \leq t(n)$.
3. Simular **un** paso de M: dos recorridos y a lo sumo k desplazamientos, cada uno $O(t(n))$.
4. M da a lo sumo $t(n)$ pasos.

$$O(t(n)) + t(n) \cdot O(t(n)) = O(t(n)^2)$$

En una frase

$t(n)$ pasos simulados, y cada uno cuesta recorrer $O(t(n))$ celdas: la máquina de una cinta tiene que ir a buscar sus cabezales.

De dónde sale el cuadrado

1. En $t(n)$ pasos, cada cabezal de M se aleja a lo sumo $t(n)$ celdas.
   Cada tramo mide a lo sumo $t(n)+1$: la parte usada de la cinta de S mide $O(t(n))$.
2. Dar el formato inicial: $O(n+k) = O(t(n))$, porque $n \leq t(n)$.
3. Simular **un** paso de M: dos recorridos y a lo sumo k desplazamientos, cada uno $O(t(n))$.
4. M da a lo sumo $t(n)$ pasos.

$$O(t(n)) + t(n) \cdot O(t(n)) = O(t(n)^2)$$

En una frase

$t(n)$ pasos simulados, y cada uno cuesta recorrer $O(t(n))$ celdas: la máquina de una cinta tiene que ir a buscar sus cabezales.

Es una cota superior. Para A promete $O(n^2)$ con una cinta, y ya vimos $O(n \log n)$.

La clase P

P

Los lenguajes que una máquina determinista de una cinta decide en tiempo polinomial: $\mathsf{P} = \bigcup_{k \geq 1} \mathsf{TIME}(n^k)$

La clase P

P

Los lenguajes que una máquina determinista de una cinta decide en tiempo polinomial: $\mathsf{P} = \bigcup_{k \geq 1} \mathsf{TIME}(n^k)$

- **Con varias cintas es la misma clase:** $(n^k)^2 = n^{2k}$, otro polinomio.

La clase P

P

Los lenguajes que una máquina determinista de una cinta decide en tiempo polinomial: $\mathsf{P} = \bigcup_{k \geq 1} \mathsf{TIME}(n^k)$

- **Con varias cintas es la misma clase:** $(n^k)^2 = n^{2k}$**,** otro polinomio.
- Lo mismo con cualquier modelo determinista razonable: son **polinomialmente equivalentes**.

La clase P

P

Los lenguajes que una máquina determinista de una cinta decide en tiempo polinomial: $\mathsf{P} = \bigcup_{k \geq 1} \mathsf{TIME}(n^k)$

- **Con varias cintas es la misma clase:** $(n^k)^2 = n^{2k}$, otro polinomio.
- Lo mismo con cualquier modelo determinista razonable: son **polinomialmente equivalentes**.

Idea clave

P no depende del modelo. Por eso se puede volver a escribir en alto nivel.

Cómo se muestra que un lenguaje está en P

Un algoritmo en alto nivel, más tres comprobaciones:

Cómo se muestra que un lenguaje está en P

Un algoritmo en alto nivel, más tres comprobaciones:

1. La **cantidad de etapas** es polinomial en la longitud n, contando las vueltas de los ciclos.

Cómo se muestra que un lenguaje está en P

Un algoritmo en alto nivel, más tres comprobaciones:

1. La **cantidad de etapas** es polinomial en la longitud n, contando las vueltas de los ciclos.
2. **Cada etapa** se hace en tiempo polinomial en un modelo determinista razonable.

Cómo se muestra que un lenguaje está en P

Un algoritmo en alto nivel, más tres comprobaciones:

1. La **cantidad de etapas** es polinomial en la longitud n, contando las vueltas de los ciclos.
2. **Cada etapa** se hace en tiempo polinomial en un modelo determinista razonable.
3. La entrada tiene una **codificación razonable**.

Cómo se muestra que un lenguaje está en P

Un algoritmo en alto nivel, más tres comprobaciones:

1. La **cantidad de etapas** es polinomial en la longitud n, contando las vueltas de los ciclos.
2. **Cada etapa** se hace en tiempo polinomial en un modelo determinista razonable.
3. La entrada tiene una **codificación razonable**.

Funciona porque la suma, el producto y la composición de polinomios son polinomios.

Cómo se muestra que un lenguaje está en P

Un algoritmo en alto nivel, más tres comprobaciones:

1. La **cantidad de etapas** es polinomial en la longitud n, contando las vueltas de los ciclos.
2. **Cada etapa** se hace en tiempo polinomial en un modelo determinista razonable.
3. La entrada tiene una **codificación razonable**.

Funciona porque la suma, el producto y la composición de polinomios son polinomios.

¿Por qué "polinomial" como sinónimo de "tratable"?

No porque n^{100} sea rápido. Es la clase más pequeña que contiene al tiempo lineal y no cambia con el modelo, es cerrada por composición, y con $n = 100$, n^3 es un millón y 2^n tiene 31 dígitos.

SECCIÓN 04

La codificación de la entrada

04

Binario y unario

terminal

```
$ build/encoding_demo
Un número x, en binario y en unario
        x   en binario             bits   símbolos en unario
        5   101                       3                    5
       12   1100                      4                   12
      100   1100100                   7                  100
     1000   1111101000               10                 1000
  1000000   11110100001001000000     20              1000000
```

- Grafos, fórmulas, gramáticas: todas las codificaciones naturales miden parecido.

Binario y unario

terminal

```
$ build/encoding_demo
Un número x, en binario y en unario
        x   en binario             bits   símbolos en unario
        5   101                       3                    5
       12   1100                      4                   12
      100   1100100                   7                  100
     1000   1111101000               10                 1000
  1000000   11110100001001000000     20              1000000
```

- Grafos, fórmulas, gramáticas: todas las codificaciones naturales miden parecido.
- Los **números** no: unario es exponencialmente más largo.

Binario y unario

terminal

```
$ build/encoding_demo
Un número x, en binario y en unario
        x   en binario             bits   símbolos en unario
        5   101                       3                    5
       12   1100                      4                   12
      100   1100100                   7                  100
     1000   1111101000               10                 1000
  1000000   11110100001001000000     20              1000000
```

- Grafos, fórmulas, gramáticas: todas las codificaciones naturales miden parecido.
- Los **números** no: unario es exponencialmente más largo.

La convención: los números se escriben en **binario**. Un número x ocupa $\lfloor \log_2 x \rfloor + 1$ símbolos.

Binario y unario

terminal

```
$ build/encoding_demo
Un número x, en binario y en unario
        x   en binario             bits   símbolos en unario
        5   101                       3                    5
       12   1100                      4                   12
      100   1100100                   7                  100
     1000   1111101000               10                 1000
  1000000   11110100001001000000     20              1000000
```

- Grafos, fórmulas, gramáticas: todas las codificaciones naturales miden parecido.
- Los **números** no: unario es exponencialmente más largo.

La convención: los números se escriben en **binario**. Un número x ocupa $\lfloor \log_2 x \rfloor + 1$ símbolos.

Consecuencia: un algoritmo que da "x pasos" para el número x es **exponencial**. La entrada mide $n = \log_2 x$, y da 2^n pasos.

¿Es primo? Probar divisores hasta la raíz

encoding_demo, tabla 2, sin su título

b	x	divisiones
8	251	14
16	65521	254
24	16777213	4094
32	4294967291	65534
40	1099511627689	1048574

x es el mayor primo de *b* bits.

- 8 bits más: 16 veces más divisiones. $2^{8/2} = 16$.

¿Es primo? Probar divisores hasta la raíz

encoding_demo, tabla 2, sin su título

b	x	divisiones
8	251	14
16	65521	254
24	16777213	4094
32	4294967291	65534
40	1099511627689	1048574

x es el mayor primo de b bits.

- 8 bits más: 16 veces más divisiones. $2^{8/2} = 16$.
- Son $\sqrt{x} \approx 2^{n/2}$ divisiones: **exponencial** en la longitud n.

Error común

Medir contra el **valor** de un número y no contra su **longitud**. ¿Polinomial en qué?

¿Es primo? Probar divisores hasta la raíz

encoding_demo, tabla 2, sin su título

b	x	divisiones
8	251	14
16	65521	254
24	16777213	4094
32	4294967291	65534
40	1099511627689	1048574

x es el mayor primo de *b* bits.

- 8 bits más: 16 veces más divisiones. $2^{8/2} = 16$.
- Son $\sqrt{x} \approx 2^{n/2}$ divisiones: **exponencial** en la longitud n.
- El algoritmo es exponencial; el problema no: PRIMES $\in$ P (2002), con otro algoritmo.

Error común

Medir contra el **valor** de un número y no contra su **longitud**. ¿Polinomial en qué?

Pseudopolinomial: la mochila otra vez

SUBSET-SUM: dados los números $x_1, \ldots, x_n$ y un objetivo t, ¿algún subconjunto suma exactamente t?

terminal

```
10 números de b bits, objetivo la mitad de la suma
 b   bits de la entrada   casillas de la tabla
 4                   46                    550
 8                   90                   8730
12                  134                 139530
16                  178                2232330
```

Tercera tabla de `build/encoding_demo`, título abreviado.

- La tabla de Algoritmos: $n \cdot (t + 1)$ casillas.

Pseudopolinomial: la mochila otra vez

SUBSET-SUM: dados los números $x_1, \ldots, x_n$ y un objetivo t, ¿algún subconjunto suma exactamente t?

terminal

```
10 números de b bits, objetivo la mitad de la suma
 b   bits de la entrada   casillas de la tabla
 4                   46                    550
 8                   90                   8730
12                  134                 139530
16                  178                2232330
```

Tercera tabla de `build/encoding_demo`, título abreviado.

- La tabla de Algoritmos: $n \cdot (t + 1)$ casillas.
- Siempre 10 números. La entrada crece 4 veces; la tabla, 4000.

Pseudopolinomial: la mochila otra vez

SUBSET-SUM: dados los números $x_1, \ldots, x_n$ y un objetivo t, ¿algún subconjunto suma exactamente t?

terminal

```
10 números de b bits, objetivo la mitad de la suma
 b   bits de la entrada   casillas de la tabla
 4                   46                    550
 8                   90                   8730
12                  134                 139530
16                  178                2232330
```

Tercera tabla de `build/encoding_demo`, título abreviado.

- La tabla de Algoritmos: $n \cdot (t + 1)$ casillas.
- Siempre 10 números. La entrada crece 4 veces; la tabla, 4000.
- n números de b bits: unas $n^2 \cdot 2^b$ casillas.

Pseudopolinomial: la mochila otra vez

SUBSET-SUM: dados los números $x_1, \ldots, x_n$ y un objetivo t, ¿algún subconjunto suma exactamente t?

terminal

```
10 números de b bits, objetivo la mitad de la suma
 b   bits de la entrada   casillas de la tabla
 4                   46                    550
 8                   90                   8730
12                  134                 139530
16                  178                2232330
```

Tercera tabla de `build/encoding_demo`, título abreviado.

- La tabla de Algoritmos: $n \cdot (t + 1)$ casillas.
- Siempre 10 números. La entrada crece 4 veces; la tabla, 4000.
- n números de b bits: unas $n^2 \cdot 2^b$ casillas.

Pseudopolinomial: polinomial en la longitud **y en el valor** de los números. Sería polinomial si se escribieran en unario.

Pseudopolinomial: la mochila otra vez

SUBSET-SUM: dados los números $x_1, \ldots, x_n$ y un objetivo t, ¿algún subconjunto suma exactamente t?

terminal

```
10 números de b bits, objetivo la mitad de la suma
 b   bits de la entrada   casillas de la tabla
 4                   46                    550
 8                   90                   8730
12                  134                 139530
16                  178                2232330
```

Tercera tabla de `build/encoding_demo`, título abreviado.

- La tabla de Algoritmos: $n \cdot (t + 1)$ casillas.
- Siempre 10 números. La entrada crece 4 veces; la tabla, 4000.
- n números de b bits: unas $n^2 \cdot 2^b$ casillas.

Pseudopolinomial: polinomial en la longitud **y en el valor** de los números. Sería polinomial si se escribieran en unario.

Por eso la mochila en $O(nW)$ no demuestra que P $=$ NP.

Polinomial, ¿en qué?

El número 12, escrito de dos maneras



Un algoritmo recibe el número x y da x pasos. ¿Cuánto es eso contra la longitud n de la entrada?

Con x en binario
$n \approx \log_2 x$, así que $x \approx 2^n$ pasos: exponencial en n

Con x en unario
$n = x$, así que son n pasos: lineal en n

La convención del curso: los números se escriben en binario.

La tabla de SUBSET-SUM con 10 números de b bits (`encoding_demo`)

bits de cada número, b	4	8	12	16	
bits de la entrada	46	90	134	178	crece menos de 4 veces
casillas de la tabla	**550**	**8 730**	**139 530**	**2 232 330**	**crece 4000 veces**

SECCIÓN 05

Ejemplos en P, y el no determinismo

05



PATH está en P

$$\text{PATH} = \{ \langle G, s, t \rangle : G \text{ es un grafo dirigido con un camino de } s \text{ a } t \}$$

Intento: probar todos los caminos. Hay del orden de m^m secuencias de vértices.

PATH está en P

$$\text{PATH} = \{ \langle G, s, t \rangle : G \text{ es un grafo dirigido con un camino de } s \text{ a } t \}$$

Intento: probar todos los caminos. Hay del orden de m^m secuencias de vértices.

Algoritmo: marcar lo alcanzable.

1. Marcar s.
2. Repetir hasta no marcar nada nuevo:
3. por cada arco (a, b) con a marcado y b sin marcar, marcar b.
4. Aceptar si t está marcado.

PATH está en P

$$\text{PATH} = \{ \langle G, s, t \rangle : G \text{ es un grafo dirigido con un camino de } s \text{ a } t \}$$

Intento: probar todos los caminos. Hay del orden de m^m secuencias de vértices.

Algoritmo: marcar lo alcanzable.

1. Marcar s.
2. Repetir hasta no marcar nada nuevo:
3. por cada arco (a, b) con a marcado y b sin marcar, marcar b.
4. Aceptar si t está marcado.

La cuenta:

- Las etapas 1 y 4, una vez.
- La etapa 3, a lo sumo m veces: cada repetición, salvo la última, marca un vértice nuevo.
- Cada repetición recorre a lo sumo m^2 arcos.

PATH está en P

$$\text{PATH} = \{ \langle G, s, t \rangle : G \text{ es un grafo dirigido con un camino de } s \text{ a } t \}$$

Intento: probar todos los caminos. Hay del orden de m^m secuencias de vértices.

Algoritmo: marcar lo alcanzable.

1. Marcar s.
2. Repetir hasta no marcar nada nuevo:
3. por cada arco (a, b) con a marcado y b sin marcar, marcar b.
4. Aceptar si t está marcado.

La cuenta:

- Las etapas 1 y 4, una vez.
- La etapa 3, a lo sumo m veces: cada repetición, salvo la última, marca un vértice nuevo.
- Cada repetición recorre a lo sumo m^2 arcos.

Mucho más lenta que la búsqueda en anchura. Para estar en P basta *algún* polinomio.

Todo lenguaje libre de contexto está en P

- L libre de contexto tiene una gramática G en forma normal de Chomsky.

Todo lenguaje libre de contexto está en P

- L libre de contexto tiene una gramática G en forma normal de Chomsky.
- **Intento:** probar las derivaciones de $2n - 1$ pasos (semana 9). Decide, pero son exponencialmente muchas.

Todo lenguaje libre de contexto está en P

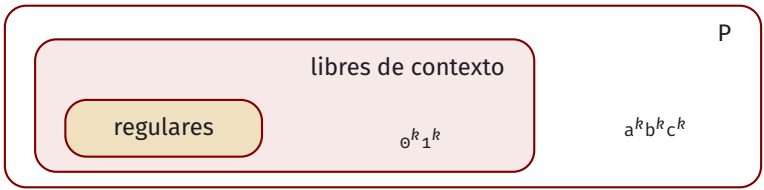
- L libre de contexto tiene una gramática G en forma normal de Chomsky.
- **Intento:** probar las derivaciones de $2n - 1$ pasos (semana 9). Decide, pero son exponencialmente muchas.
- **CYK** (semana 5): $T[i][j]$ = variables que generan $w[i..j]$. Tres ciclos sobre posiciones y uno sobre las reglas.

Todo lenguaje libre de contexto está en P

- L libre de contexto tiene una gramática G en forma normal de Chomsky.
- **Intento:** probar las derivaciones de $2n - 1$ pasos (semana 9). Decide, pero son exponencialmente muchas.
- **CYK** (semana 5): $T[i][j]$ = variables que generan $w[i..j]$. Tres ciclos sobre posiciones y uno sobre las reglas.
- **La gramática no es parte de la entrada.** L es fijo, G es fija: la cantidad de reglas es una constante. $O(n^3)$ operaciones.

Todo lenguaje libre de contexto está en P

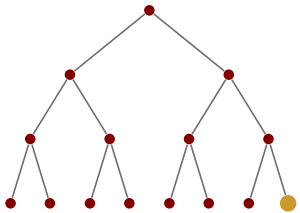
- L libre de contexto tiene una gramática G en forma normal de Chomsky.
- **Intento:** probar las derivaciones de $2n - 1$ pasos (semana 9). Decide, pero son exponencialmente muchas.
- **CYK** (semana 5): $T[i][j]$ = variables que generan $w[i..j]$. Tres ciclos sobre posiciones y uno sobre las reglas.
- **La gramática no es parte de la entrada.** L es fijo, G es fija: la cantidad de reglas es una constante. $O(n^3)$ operaciones.



El tiempo de una máquina no determinista

Definición

N decididor: todas sus ramas se detienen. Su tiempo $f(n)$ es el máximo número de pasos de **cualquier rama**, con cualquier entrada de longitud n.

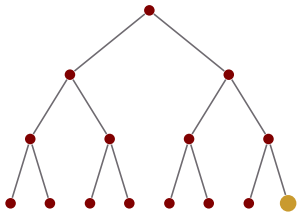


Tiempo: la altura del árbol.

El tiempo de una máquina no determinista

Definición

N decididor: todas sus ramas se detienen. Su tiempo $f(n)$ es el máximo número de pasos de **cualquier rama**, con cualquier entrada de longitud n.



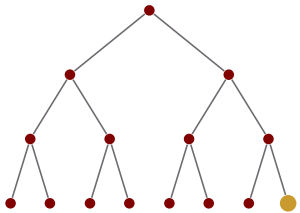
Tiempo: la altura del árbol.

- Es la **profundidad** del árbol, no la cantidad de ramas.

El tiempo de una máquina no determinista

Definición

N decididor: todas sus ramas se detienen. Su tiempo $f(n)$ es el máximo número de pasos de **cualquier rama**, con cualquier entrada de longitud n.



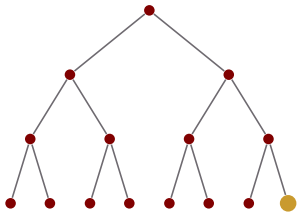
Tiempo: la altura del árbol.

- Es la **profundidad** del árbol, no la cantidad de ramas.
- Tiempo n puede ser 2^n ramas de n pasos.

El tiempo de una máquina no determinista

Definición

N decididor: todas sus ramas se detienen. Su tiempo $f(n)$ es el máximo número de pasos de **cualquier rama**, con cualquier entrada de longitud n.



Tiempo: la altura del árbol.

- Es la **profundidad** del árbol, no la cantidad de ramas.
- Tiempo n puede ser 2^n ramas de n pasos.
- Acepta si **alguna** rama acepta; rechaza si **todas** rechazan.

El no determinismo parece costar caro

Teorema

Sea $t(n) \geq n$. Toda máquina no determinista de tiempo $t(n)$ tiene una determinista equivalente de tiempo $2^{O(t(n))}$.

La simulación de la semana 7, que recorre el árbol por niveles, contando:

El no determinismo parece costar caro

Teorema

Sea $t(n) \geq n$. Toda máquina no determinista de tiempo $t(n)$ tiene una determinista equivalente de tiempo $2^{O(t(n))}$.

La simulación de la semana 7, que recorre el árbol por niveles, contando:

- Profundidad $t(n)$ y a lo sumo b opciones por paso: a lo sumo $2\,b^{t(n)}$ nodos.

El no determinismo parece costar caro

Teorema

Sea $t(n) \geq n$. Toda máquina no determinista de tiempo $t(n)$ tiene una determinista equivalente de tiempo $2^{O(t(n))}$.

La simulación de la semana 7, que recorre el árbol por niveles, contando:

- Profundidad $t(n)$ y a lo sumo b opciones por paso: a lo sumo $2\,b^{t(n)}$ nodos.
- Para cada nodo se vuelve a ejecutar desde la raíz: $O(t(n))$ pasos por nodo.

El no determinismo parece costar caro

Teorema

Sea $t(n) \geq n$. Toda máquina no determinista de tiempo $t(n)$ tiene una determinista equivalente de tiempo $2^{O(t(n))}$.

La simulación de la semana 7, que recorre el árbol por niveles, contando:

- Profundidad $t(n)$ y a lo sumo b opciones por paso: a lo sumo $2\,b^{t(n)}$ nodos.
- Para cada nodo se vuelve a ejecutar desde la raíz: $O(t(n))$ pasos por nodo.

$$t(n) \cdot b^{t(n)} = 2^{\log_2 t(n)} \cdot 2^{t(n) \log_2 b} = 2^{O(t(n))}$$

El no determinismo parece costar caro

Teorema

Sea $t(n) \geq n$. Toda máquina no determinista de tiempo $t(n)$ tiene una determinista equivalente de tiempo $2^{O(t(n))}$.

La simulación de la semana 7, que recorre el árbol por niveles, contando:

- Profundidad $t(n)$ y a lo sumo b opciones por paso: a lo sumo $2\,b^{t(n)}$ nodos.
- Para cada nodo se vuelve a ejecutar desde la raíz: $O(t(n))$ pasos por nodo.

$$t(n) \cdot b^{t(n)} = 2^{\log_2 t(n)} \cdot 2^{t(n) \log_2 b} = 2^{O(t(n))}$$

- Pasar de varias cintas a una eleva al cuadrado: sigue siendo $2^{O(t(n))}$.

Dos simulaciones, dos destinos

De	A una cinta determinista	Un polinomio se vuelve
Varias cintas, tiempo $t(n)$	$O(t(n)^2)$	otro polinomio
No determinista, tiempo $t(n)$	$2^{O(t(n))}$	una exponencial

Dos simulaciones, dos destinos

De	A una cinta determinista	Un polinomio se vuelve
Varias cintas, tiempo $t(n)$	$O(t(n)^2)$	otro polinomio
No determinista, tiempo $t(n)$	$2^{O(t(n))}$	una exponencial

- La segunda es una cota superior. Nadie ha demostrado que haga falta tanto, y nadie tiene una simulación mejor.

Dos simulaciones, dos destinos

De	A una cinta determinista	Un polinomio se vuelve
Varias cintas, tiempo $t(n)$	$O(t(n)^2)$	otro polinomio
No determinista, tiempo $t(n)$	$2^{O(t(n))}$	una exponencial

- La segunda es una cota superior. Nadie ha demostrado que haga falta tanto, y nadie tiene una simulación mejor.

Idea clave

Una cinta y varias: polinomialmente equivalentes. Determinista y no determinista: no se sabe. Por eso la definición de P dice “determinista”.

SECCIÓN 06

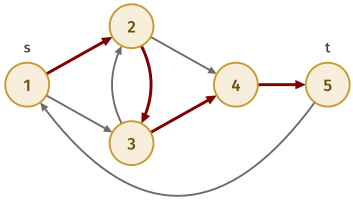
Encontrar frente a comprobar

06

Un camino hamiltoniano

HAMPATH: ¿hay en *G* un camino de *s* a *t* que pase por cada vértice exactamente una vez?

El grafo G de `verify_demo`, con s = 1 y t = 5



Un certificado que nos entregan

c = [1] [2] [3] [4] [5]

El verificador V comprueba:

1. c es una lista de 5 vértices de G
2. ningún vértice se repite
3. el primero es s y el último es t
4. cada paso es un arco de G: 1→2, 2→3, 3→4, 4→5

Las cuatro pasan: V acepta

Certificados que V rechaza, y la comprobación en la que fallan

- ⊗ [1] [3] [4] [5] — 1. tiene 4 vértices y no 5
- ⊗ [1] [2] [2] [4] [5] — 2. repite el vértice 2
- ⊗ [2] [3] [4] [5] [1] — 3. es hamiltoniano, pero no de s a t
- ⊗ [1] [2] [4] [3] [5] — 4. no hay arco 4→3

Encontrar frente a comprobar

terminal

```
$ build/verify_demo
  c = 1 2 3 4 5     válido
  c = 1 3 2 4 5     válido
  c = 1 3 4 5       inválido   tiene 4 vértices y no 5
  c = 1 2 2 4 5     inválido   repite el vértice 2
  c = 1 2 3 4 6     inválido   el vértice 6 no existe
  c = 1 2 4 3 5     inválido   no hay arco 4→3
  c = 2 3 4 5 1     inválido   es un camino hamiltoniano, pero no de s a t
```

Parte de la salida: los certificados para el grafo de la diapositiva anterior.

Encontrar frente a comprobar

terminal

```
$ build/verify_demo
  c = 1 2 3 4 5     válido
  c = 1 3 2 4 5     válido
  c = 1 3 4 5       inválido   tiene 4 vértices y no 5
  c = 1 2 2 4 5     inválido   repite el vértice 2
  c = 1 2 3 4 6     inválido   el vértice 6 no existe
  c = 1 2 4 3 5     inválido   no hay arco 4→3
  c = 2 3 4 5 1     inválido   es un camino hamiltoniano, pero no de s a t
```

Parte de la salida: los certificados para el grafo de la diapositiva anterior.

- **Encontrar:** nadie conoce un algoritmo polinomial. De 120 ordenamientos sirven dos.

Encontrar frente a comprobar

terminal

```
$ build/verify_demo
  c = 1 2 3 4 5     válido
  c = 1 3 2 4 5     válido
  c = 1 3 4 5       inválido   tiene 4 vértices y no 5
  c = 1 2 2 4 5     inválido   repite el vértice 2
  c = 1 2 3 4 6     inválido   el vértice 6 no existe
  c = 1 2 4 3 5     inválido   no hay arco 4→3
  c = 2 3 4 5 1     inválido   es un camino hamiltoniano, pero no de s a t
```

Parte de la salida: los certificados para el grafo de la diapositiva anterior.

- **Encontrar:** nadie conoce un algoritmo polinomial. De 120 ordenamientos sirven dos.
- **Comprobar** uno que nos entregan: cinco revisiones, tiempo polinomial.

Verificadores y la clase NP

Verificador *(verifier)*

Un **verificador** de un lenguaje A es un algoritmo V tal que

$$A = \{\, w : V \text{ acepta } \langle w, c \rangle \text{ para alguna cadena } c \,\}.$$

c es un **certificado** *(certificate)*. El tiempo de V se mide **solo en función de** $|w|$.

Verificadores y la clase NP

Verificador *(verifier)*

Un **verificador** de un lenguaje A es un algoritmo V tal que

$$A = \{ w : V \text{ acepta } \langle w, c \rangle \text{ para alguna cadena } c \}.$$

c es un **certificado** *(certificate)*. El tiempo de V se mide **solo en función de** $|w|$.

NP es la clase de los lenguajes que tienen un verificador de **tiempo polinomial**.

Verificadores y la clase NP

Verificador *(verifier)*

Un **verificador** de un lenguaje A es un algoritmo V tal que

$$A = \{\, w : V \text{ acepta } \langle w, c \rangle \text{ para alguna cadena } c \,\}.$$

c es un **certificado** *(certificate)*. El tiempo de V se mide **solo en función de** $|w|$.

NP es la clase de los lenguajes que tienen un verificador de **tiempo polinomial**.

- **Completitud:** si $w \in A$, *existe* un c que V acepta.

Verificadores y la clase NP

Verificador *(verifier)*

Un **verificador** de un lenguaje A es un algoritmo V tal que

$$A = \{\, w : V \text{ acepta } \langle w, c \rangle \text{ para alguna cadena } c \,\}.$$

c es un **certificado** *(certificate)*. El tiempo de V se mide **solo en función de** $|w|$.

NP es la clase de los lenguajes que tienen un verificador de **tiempo polinomial**.

- **Completitud:** si $w \in A$, *existe* un c que V acepta.
- **Solidez:** si $w \notin A$, V rechaza con *todo* c.

Verificadores y la clase NP

Verificador *(verifier)*

Un **verificador** de un lenguaje A es un algoritmo V tal que

$$A = \{ w : V \text{ acepta } \langle w, c \rangle \text{ para alguna cadena } c \}.$$

c es un **certificado** *(certificate)*. El tiempo de V se mide **solo en función de** $|w|$.

NP es la clase de los lenguajes que tienen un verificador de **tiempo polinomial**.

- **Completitud:** si $w \in A$, *existe* un c que V acepta.
- **Solidez:** si $w \notin A$, V rechaza con *todo* c.
- En NP, las respuestas "sí" tienen demostraciones cortas que se comprueban rápido.

Leer la definición despacio

- **El verificador no confía en el certificado.** *c* puede ser basura, o estar armado para engañar. Un certificado mal formado se rechaza.

Leer la definición despacio

- **El verificador no confía en el certificado.** c puede ser basura, o estar armado para engañar. Un certificado mal formado se rechaza.
- **El certificado es corto.** V da a lo sumo $p(|w|)$ pasos: no lee más de $p(|w|)$ símbolos de c. Si hay un certificado, hay uno de longitud polinomial.

Leer la definición despacio

- **El verificador no confía en el certificado.** c puede ser basura, o estar armado para engañar. Un certificado mal formado se rechaza.
- **El certificado es corto.** V da a lo sumo $p(|w|)$ pasos: no lee más de $p(|w|)$ símbolos de c. Si hay un certificado, hay uno de longitud polinomial.
- **Nadie dice de dónde sale.** La definición solo pide que exista.

Leer la definición despacio

- **El verificador no confía en el certificado.** c puede ser basura, o estar armado para engañar. Un certificado mal formado se rechaza.
- **El certificado es corto.** V da a lo sumo $p(|w|)$ pasos: no lee más de $p(|w|)$ símbolos de c. Si hay un certificado, hay uno de longitud polinomial.
- **Nadie dice de dónde sale.** La definición solo pide que exista.

Error común

Dar como "verificador" un algoritmo que **busca** la solución. El verificador *recibe* el certificado y solo comprueba. Si dice "para cada subconjunto", no es un verificador.

SECCIÓN 07

Cinco problemas de NP y sus certificados

07

HAMPATH está en NP

Certificado: una lista de vértices. **Verificador,** para G con m vértices:

1. Comprobar que c es una lista de exactamente m vértices de G.
2. Comprobar que ningún vértice se repite.
3. Comprobar que el primero es s y el último es t.
4. Para cada par consecutivo, comprobar que G tiene el arco.
5. Si todo pasa, aceptar. Si algo falla, rechazar.

HAMPATH está en NP

Certificado: una lista de vértices. **Verificador,** para G con m vértices:

1. Comprobar que c es una lista de exactamente m vértices de G.
2. Comprobar que ningún vértice se repite.
3. Comprobar que el primero es s y el último es t.
4. Para cada par consecutivo, comprobar que G tiene el arco.
5. Si todo pasa, aceptar. Si algo falla, rechazar.

- **Tiempo:** m^2 comparaciones en el paso 2, y $m - 1$ búsquedas en una lista de a lo sumo m^2 arcos. Polinomial.

HAMPATH está en NP

Certificado: una lista de vértices. **Verificador,** para G con m vértices:

1. Comprobar que c es una lista de exactamente m vértices de G.
2. Comprobar que ningún vértice se repite.
3. Comprobar que el primero es s y el último es t.
4. Para cada par consecutivo, comprobar que G tiene el arco.
5. Si todo pasa, aceptar. Si algo falla, rechazar.

- **Tiempo:** m^2 comparaciones en el paso 2, y $m - 1$ búsquedas en una lista de a lo sumo m^2 arcos. Polinomial.
- **Tamaño del certificado:** $m\lceil \log_2 m \rceil$ bits.

HAMPATH está en NP

Certificado: una lista de vértices. **Verificador,** para G con m vértices:

1. Comprobar que c es una lista de exactamente m vértices de G.
2. Comprobar que ningún vértice se repite.
3. Comprobar que el primero es s y el último es t.
4. Para cada par consecutivo, comprobar que G tiene el arco.
5. Si todo pasa, aceptar. Si algo falla, rechazar.

- **Tiempo:** m^2 comparaciones en el paso 2, y $m - 1$ búsquedas en una lista de a lo sumo m^2 arcos. Polinomial.
- **Tamaño del certificado:** $m\lceil \log_2 m \rceil$ bits.
- **Es un verificador:** un camino hamiltoniano pasa las revisiones, y una lista que las pasa *es* un camino hamiltoniano de s a t.

CLIQUE

CLIQUE: ¿tiene el grafo no dirigido G un conjunto de k vértices todos unidos entre sí?

- **Certificado:** k vértices.
- **Verificador:** son k, son vértices, son distintos, y cada par tiene su arista.

terminal

```
CLIQUE: grafo no dirigido con 6 vértices y las aristas
  1-2 1-3 1-4 2-3 2-4 3-4 4-5 5-6
  k = 4
  c = 1 2 3 4       válido
  c = 4 2 1 3       válido   el orden no importa
  c = 1 2 3 5       inválido   falta la arista 1-5
  c = 1 2 3 3       inválido   repite el vértice 3
  c = 1 2 3         inválido   son 3 vértices y se pedían 4
```

De la salida de `build/verify_demo`.

CLIQUE

CLIQUE: ¿tiene el grafo no dirigido G un conjunto de k vértices todos unidos entre sí?

- **Certificado:** k vértices.
- **Verificador:** son k, son vértices, son distintos, y cada par tiene su arista.
- **Tiempo:** $k(k-1)/2$ pares, cada uno buscado en la lista de aristas.

terminal

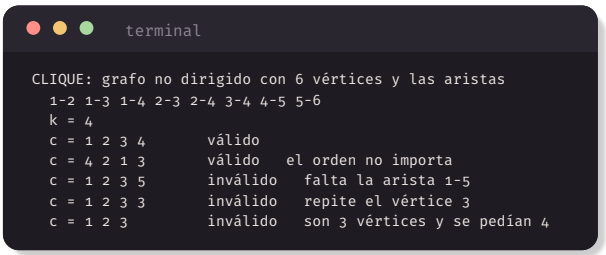
```
CLIQUE: grafo no dirigido con 6 vértices y las aristas
  1-2 1-3 1-4 2-3 2-4 3-4 4-5 5-6
  k = 4
  c = 1 2 3 4       válido
  c = 4 2 1 3       válido   el orden no importa
  c = 1 2 3 5       inválido   falta la arista 1-5
  c = 1 2 3 3       inválido   repite el vértice 3
  c = 1 2 3         inválido   son 3 vértices y se pedían 4
```

De la salida de `build/verify_demo`.

CLIQUE

CLIQUE: ¿tiene el grafo no dirigido G un conjunto de k vértices todos unidos entre sí?

- **Certificado:** k vértices.
- **Verificador:** son k, son vértices, son distintos, y cada par tiene su arista.
- **Tiempo:** $k(k-1)/2$ pares, cada uno buscado en la lista de aristas.
- k es un número de la entrada, pero $k \leq m$: si no, se rechaza sin mirar.



De la salida de `build/verify_demo`.

SUBSET-SUM

- **Certificado:** las posiciones de los números elegidos.
- **Verificador:** existen, son distintas, y sus números suman t.

terminal

```
SUBSET-SUM: los números 8 13 21 5 17 2, objetivo 28.
  c = 1 2 4 6       válido   8 + 13 + 5 + 2 = 28
  c = 3 4 6         válido   21 + 5 + 2 = 28
  c = 1 3           inválido   8 + 21 = 29
  c = 4 4 4 2       inválido   5 + 5 + 5 + 13 = 28,
                               pero repite la posición 4
  c = 1 2 4 7       inválido   no hay posición 7
```

De `build/verify_demo`; dos líneas recortadas o partidas.

SUBSET-SUM

- **Certificado:** las posiciones de los números elegidos.
- **Verificador:** existen, son distintas, y sus números suman *t*.
- **Tiempo:** a lo sumo *n* sumas. Polinomial *en la longitud*: no hay ninguna tabla que dependa del valor de *t*.

terminal

```
SUBSET-SUM: los números 8 13 21 5 17 2, objetivo 28.
  c = 1 2 4 6       válido   8 + 13 + 5 + 2 = 28
  c = 3 4 6         válido   21 + 5 + 2 = 28
  c = 1 3           inválido   8 + 21 = 29
  c = 4 4 4 2       inválido   5 + 5 + 5 + 13 = 28,
                               pero repite la posición 4
  c = 1 2 4 7       inválido   no hay posición 7
```

De `build/verify_demo`; dos líneas recortadas o partidas.

SUBSET-SUM

- **Certificado:** las posiciones de los números elegidos.
- **Verificador:** existen, son distintas, y sus números suman t.
- **Tiempo:** a lo sumo n sumas. Polinomial *en la longitud*: no hay ninguna tabla que dependa del valor de t.

terminal

```
SUBSET-SUM: los números 8 13 21 5 17 2, objetivo 28.
  c = 1 2 4 6       válido   8 + 13 + 5 + 2 = 28
  c = 3 4 6         válido   21 + 5 + 2 = 28
  c = 1 3           inválido   8 + 21 = 29
  c = 4 4 4 2       inválido   5 + 5 + 5 + 13 = 28,
                               pero repite la posición 4
  c = 1 2 4 7       inválido   no hay posición 7
```

De `build/verify_demo`; dos líneas recortadas o partidas.

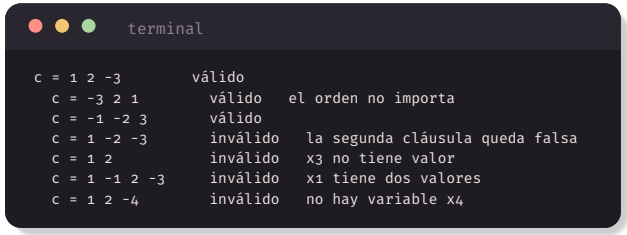
Por qué revisar las repeticiones

Con objetivo 4 no hay solución. Un verificador que solo suma aceptaría “posición 6, posición 6”: $2 + 2 = 4$.

SAT

SAT: ¿hay una asignación que satisfaga la fórmula?
$\varphi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_1)$

- **Certificado:** una asignación, un bit por variable.
- **Verificador:** cada variable tiene un valor, y la fórmula evaluada da verdadero.

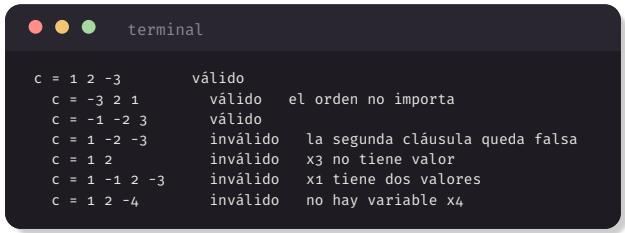


De la salida de `build/verify_demo`, sin la línea que imprime la fórmula.

SAT

SAT: ¿hay una asignación que satisfaga la fórmula?
$\varphi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_1)$

- **Certificado:** una asignación, un bit por variable.
- **Verificador:** cada variable tiene un valor, y la fórmula evaluada da verdadero.
- **Tiempo:** evaluar es recorrer la fórmula.

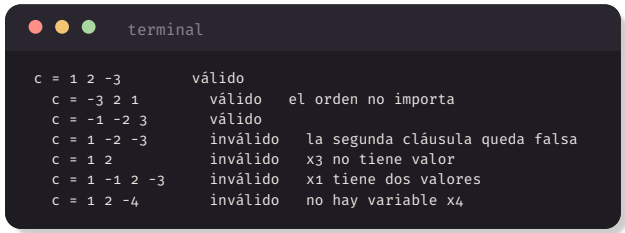


De la salida de `build/verify_demo`, sin la línea que imprime la fórmula.

SAT

SAT: ¿hay una asignación que satisfaga la fórmula?
$\varphi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_1)$

- **Certificado:** una asignación, un bit por variable.
- **Verificador:** cada variable tiene un valor, y la fórmula evaluada da verdadero.
- **Tiempo:** evaluar es recorrer la fórmula.
- En el código: CNF, y literales i o $-i$, como en DIMACS.

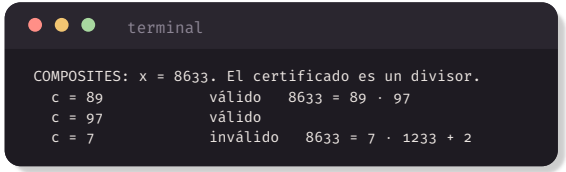


De la salida de `build/verify_demo`, sin la línea que imprime la fórmula.

COMPOSITES

COMPOSITES: ¿es $x = pq$ para algunos enteros $p, q > 1$? Con x en binario.

- **Certificado:** un divisor d.
- **Verificador:** $1 < d < x$, y el resto de dividir x entre d es 0.



De `build/verify_demo`; siguen los certificados 1 y 8633, inválidos.

COMPOSITES

COMPOSITES: ¿es $x = pq$ para algunos enteros $p, q > 1$? Con x en binario.

- **Certificado:** un divisor d.
- **Verificador:** $1 < d < x$, y el resto de dividir x entre d es 0.
- **Tiempo:** una división de números de n bits, $O(n^2)$.

terminal

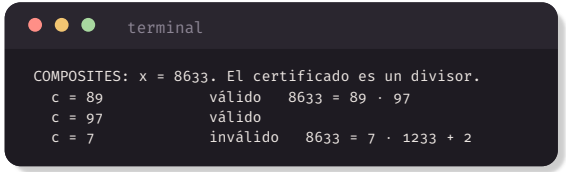
```
COMPOSITES: x = 8633. El certificado es un divisor.
  c = 89            válido   8633 = 89 · 97
  c = 97            válido
  c = 7             inválido   8633 = 7 · 1233 + 2
```

De `build/verify_demo`; siguen los certificados 1 y 8633, inválidos.

COMPOSITES

COMPOSITES: ¿es $x = pq$ para algunos enteros $p, q > 1$? Con x en binario.

- **Certificado:** un divisor d.
- **Verificador:** $1 < d < x$, y el resto de dividir x entre d es 0.
- **Tiempo:** una división de números de n bits, $O(n^2)$.
- Con $x = 8641$, que es primo, el verificador rechaza los 8642 certificados de 0 a 8641.



De `build/verify_demo`; siguen los certificados 1 y 8633, inválidos.

COMPOSITES

COMPOSITES: ¿es $x = pq$ para algunos enteros $p, q > 1$? Con x en binario.

- **Certificado:** un divisor d.
- **Verificador:** $1 < d < x$, y el resto de dividir x entre d es 0.
- **Tiempo:** una división de números de n bits, $O(n^2)$.
- Con $x = 8641$, que es primo, el verificador rechaza los 8642 certificados de 0 a 8641.

```
terminal

COMPOSITES: x = 8633. El certificado es un divisor.
  c = 89            válido   8633 = 89 · 97
  c = 97            válido
  c = 7             inválido   8633 = 7 · 1233 + 2
```

De `build/verify_demo`; siguen los certificados 1 y 8633, inválidos.

Estar en NP no es ser difícil

Hallar el 89 probando divisores es exponencial; comprobarlo es una división. Y aun así COMPOSITES $\in$ P: es el complemento de PRIMES.

Cinco certificados

Lenguaje	Certificado	Bits	El verificador
HAMPATH	Los m vértices en orden	$m\lceil \log_2 m \rceil$	Todos, una vez, de s a t; cada arco
CLIQUE	k vértices	$k\lceil \log_2 m \rceil$	Distintos; cada par unido
SUBSET-SUM	Un subconjunto	n	Suma y compara con t
SAT	Una asignación	Uno por variable	Evalúa la fórmula
COMPOSITES	Un divisor	A lo sumo los de x	Divide

Cinco certificados

Lenguaje	Certificado	Bits	El verificador
HAMPATH	Los m vértices en orden	$m\lceil \log_2 m \rceil$	Todos, una vez, de s a t; cada arco
CLIQUE	k vértices	$k\lceil \log_2 m \rceil$	Distintos; cada par unido
SUBSET-SUM	Un subconjunto	n	Suma y compara con t
SAT	Una asignación	Uno por variable	Evalúa la fórmula
COMPOSITES	Un divisor	A lo sumo los de x	Divide

Para mostrar que un lenguaje está en NP se dan tres cosas:

1. qué es el certificado;
2. qué hace el verificador, incluido rechazar lo mal formado;
3. por qué corre en tiempo polinomial en la longitud de la instancia.

SECCIÓN 08

Máquinas no deterministas: la misma clase

08

La definición original

NTIME($t(n)$): los lenguajes que decide alguna máquina de Turing **no determinista** de tiempo $O(t(n))$.
NP viene de *Nondeterministic Polynomial time*. Una máquina para HAMPATH:

1. Escribir una lista $p_1, \ldots, p_m$, eligiendo cada número **de forma no determinista** entre 1 y m.
2. Si algún número se repite, o $p_1 \neq s$, o $p_m \neq t$, rechazar.
3. Si falta algún arco de p_i a p_{i+1}, rechazar. Si no, aceptar.

La definición original

NTIME($t(n)$): los lenguajes que decide alguna máquina de Turing **no determinista** de tiempo $O(t(n))$.
NP viene de *Nondeterministic Polynomial time*. Una máquina para HAMPATH:

1. Escribir una lista $p_1, \ldots, p_m$, eligiendo cada número **de forma no determinista** entre 1 y m.
2. Si algún número se repite, o $p_1 \neq s$, o $p_m \neq t$, rechazar.
3. Si falta algún arco de p_i a p_{i+1}, rechazar. Si no, aceptar.

- m^m ramas, cada una corta. Acepta si **alguna** llega al final.

La definición original

NTIME($t(n)$): los lenguajes que decide alguna máquina de Turing **no determinista** de tiempo $O(t(n))$.
NP viene de *Nondeterministic Polynomial time*. Una máquina para HAMPATH:

1. Escribir una lista $p_1, \ldots, p_m$, eligiendo cada número **de forma no determinista** entre 1 y m.
2. Si algún número se repite, o $p_1 \neq s$, o $p_m \neq t$, rechazar.
3. Si falta algún arco de p_i a p_{i+1}, rechazar. Si no, aceptar.

- m^m ramas, cada una corta. Acepta si **alguna** llega al final.
- Los pasos 2 y 3 son el verificador. El paso 1 es “adivinar el certificado”.

La definición original

NTIME($t(n)$): los lenguajes que decide alguna máquina de Turing **no determinista** de tiempo $O(t(n))$.
NP viene de *Nondeterministic Polynomial time*. Una máquina para HAMPATH:

1. Escribir una lista $p_1, \ldots, p_m$, eligiendo cada número **de forma no determinista** entre 1 y m.
2. Si algún número se repite, o $p_1 \neq s$, o $p_m \neq t$, rechazar.
3. Si falta algún arco de p_i a p_{i+1}, rechazar. Si no, aceptar.

- m^m ramas, cada una corta. Acepta si **alguna** llega al final.
- Los pasos 2 y 3 son el verificador. El paso 1 es "adivinar el certificado".

Error común

Leer "adivina" como suerte o inteligencia. Es un "existe una rama", como en los NFA.

Las dos definiciones coinciden

Teorema

Un lenguaje tiene un verificador de tiempo polinomial si y solo si alguna máquina no determinista de tiempo polinomial lo decide:

$$\mathrm{NP} = \bigcup_{k \geq 1} \mathrm{NTIME}(n^k)$$

De un verificador V (a lo sumo $p(n)$ pasos) a una máquina N:

1. Escribir de forma no determinista una cadena c de longitud a lo sumo $p(n)$.
2. Ejecutar V con $\langle w, c \rangle$. Aceptar si V acepta.

Las dos definiciones coinciden

Teorema

Un lenguaje tiene un verificador de tiempo polinomial si y solo si alguna máquina no determinista de tiempo polinomial lo decide:

$$\mathrm{NP} = \bigcup_{k \geq 1} \mathrm{NTIME}(n^k)$$

De un verificador V (a lo sumo $p(n)$ pasos) a una máquina N:

1. Escribir de forma no determinista una cadena c de longitud a lo sumo $p(n)$.
2. Ejecutar V con $\langle w, c \rangle$. Aceptar si V acepta.

- Si N acepta, alguna rama escribió un c que V aceptó: $w \in A$.

Las dos definiciones coinciden

Teorema

Un lenguaje tiene un verificador de tiempo polinomial si y solo si alguna máquina no determinista de tiempo polinomial lo decide:

$$\mathrm{NP} = \bigcup_{k \geq 1} \mathrm{NTIME}(n^k)$$

De un verificador V (a lo sumo $p(n)$ pasos) a una máquina N:

1. Escribir de forma no determinista una cadena c de longitud a lo sumo $p(n)$.
2. Ejecutar V con $\langle w, c \rangle$. Aceptar si V acepta.

- Si N acepta, alguna rama escribió un c que V aceptó: $w \in A$.
- Si $w \in A$, algún c sirve. ¿Y si es más largo que $p(n)$? En $p(n)$ pasos V no pasa de la celda $p(n)$: con el prefijo de c de esa longitud hace exactamente lo mismo.

Las dos definiciones coinciden

Teorema

Un lenguaje tiene un verificador de tiempo polinomial si y solo si alguna máquina no determinista de tiempo polinomial lo decide:

$$\mathrm{NP} = \bigcup_{k \geq 1} \mathrm{NTIME}(n^k)$$

De un verificador V (a lo sumo $p(n)$ pasos) a una máquina N:

1. Escribir de forma no determinista una cadena c de longitud a lo sumo $p(n)$.
2. Ejecutar V con $\langle w, c \rangle$. Aceptar si V acepta.

- Si N acepta, alguna rama escribió un c que V aceptó: $w \in A$.
- Si $w \in A$, algún c sirve. ¿Y si es más largo que $p(n)$? En $p(n)$ pasos V no pasa de la celda $p(n)$: con el prefijo de c de esa longitud hace exactamente lo mismo.
- Cada rama: tiempo polinomial para escribir, más $p(n)$ pasos de V.

La otra dirección: el certificado es la lista de decisiones

De una máquina N (ramas de a lo sumo $p(n)$ pasos, a lo sumo b opciones) a un verificador V:

1. Con $\langle w, c \rangle$, donde $c = c_1 c_2 \cdots$ sobre $\{1, \ldots, b\}$: simular N con w, tomando en el paso i la opción número c_i.
2. Si la opción no existe, o c se acaba antes de que N se detenga, rechazar.
3. Si esa rama acepta, aceptar. Si rechaza, rechazar.

La otra dirección: el certificado es la lista de decisiones

De una máquina N (ramas de a lo sumo $p(n)$ pasos, a lo sumo b opciones) a un verificador V:

1. Con $\langle w, c \rangle$, donde $c = c_1 c_2 \cdots$ sobre $\{1, \ldots, b\}$: simular N con w, tomando en el paso i la opción número c_i.
2. Si la opción no existe, o c se acaba antes de que N se detenga, rechazar.
3. Si esa rama acepta, aceptar. Si rechaza, rechazar.

- Si $w \in A$: alguna rama acepta. La lista de sus decisiones es un c que V acepta.

La otra dirección: el certificado es la lista de decisiones

De una máquina N (ramas de a lo sumo $p(n)$ pasos, a lo sumo b opciones) a un verificador V:

1. Con $\langle w, c \rangle$, donde $c = c_1 c_2 \cdots$ sobre $\{1, \ldots, b\}$: simular N con w, tomando en el paso i la opción número c_i.
2. Si la opción no existe, o c se acaba antes de que N se detenga, rechazar.
3. Si esa rama acepta, aceptar. Si rechaza, rechazar.

- Si $w \in A$: alguna rama acepta. La lista de sus decisiones es un c que V acepta.
- Si V acepta $\langle w, c \rangle$: siguió una rama real de N que acepta. Entonces $w \in A$.

La otra dirección: el certificado es la lista de decisiones

De una máquina N (ramas de a lo sumo $p(n)$ pasos, a lo sumo b opciones) a un verificador V:

1. Con $\langle w, c \rangle$, donde $c = c_1 c_2 \cdots$ sobre $\{1, \ldots, b\}$: simular N con w, tomando en el paso i la opción número c_i.
2. Si la opción no existe, o c se acaba antes de que N se detenga, rechazar.
3. Si esa rama acepta, aceptar. Si rechaza, rechazar.

- Si $w \in A$: alguna rama acepta. La lista de sus decisiones es un c que V acepta.
- Si V acepta $\langle w, c \rangle$: siguió una rama real de N que acepta. Entonces $w \in A$.
- V es determinista y simula a lo sumo $p(n)$ pasos, sea cual sea la longitud de c.

La otra dirección: el certificado es la lista de decisiones

De una máquina N (ramas de a lo sumo $p(n)$ pasos, a lo sumo b opciones) a un verificador V:

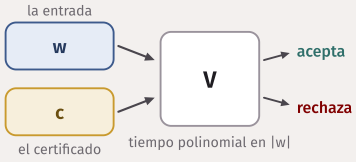
1. Con $\langle w, c\rangle$, donde $c = c_1 c_2 \cdots$ sobre $\{1, \ldots, b\}$: simular N con w, tomando en el paso i la opción número c_i.
2. Si la opción no existe, o c se acaba antes de que N se detenga, rechazar.
3. Si esa rama acepta, aceptar. Si rechaza, rechazar.

- Si $w \in A$: alguna rama acepta. La lista de sus decisiones es un c que V acepta.
- Si V acepta $\langle w, c\rangle$: siguió una rama real de N que acepta. Entonces $w \in A$.
- V es determinista y simula a lo sumo $p(n)$ pasos, sea cual sea la longitud de c.

Idea clave. "Existe un certificado que el verificador acepta" y "existe una rama que acepta" son el mismo *existe*.

Las dos definiciones, lado a lado

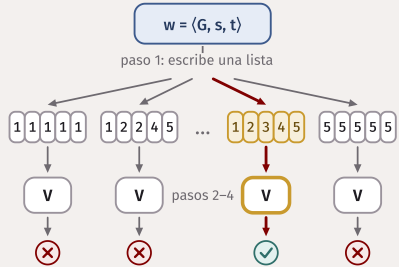
Con un verificador



Completitud
Si $w \in A$, existe algún c con el que V acepta $\langle w, c \rangle$.

Solidez
Si $w \notin A$, V rechaza $\langle w, c \rangle$ para toda cadena c.

Con una máquina no determinista



$5^5 = 3125$ ramas, una por lista. Alguna acepta: N acepta w.

«Existe un certificado que V acepta» y «existe una rama de N que acepta» son el mismo «existe».

De V a N: la máquina adivina el certificado. De N a V: el certificado es la lista de decisiones de la rama.

SECCIÓN 09

P, NP y EXPTIME

09



P está contenida en NP

Teorema

$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXPTIME}, \quad \text{donde} \quad \mathsf{EXPTIME} = \bigcup_{k \geq 1} \mathsf{TIME}\left(2^{n^k}\right).$

Primera inclusión. Sea M una máquina determinista que decide A en tiempo polinomial. Un decididor es un verificador que no necesita ayuda:

1. Con $\langle w, c \rangle$: ignorar c y ejecutar M con w.
2. Si M acepta, aceptar. Si rechaza, rechazar.

P está contenida en NP

Teorema

$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{EXPTIME}, \quad$ donde $\quad \mathrm{EXPTIME} = \bigcup_{k \geq 1} \mathrm{TIME}\left(2^{n^k}\right).$

Primera inclusión. Sea M una máquina determinista que decide A en tiempo polinomial. Un decididor es un verificador que no necesita ayuda:

1. Con $\langle w, c \rangle$: ignorar c y ejecutar M con w.
2. Si M acepta, aceptar. Si rechaza, rechazar.

- Si $w \in A$: acepta con cualquier c; en particular, con alguno.
- Si $w \notin A$: rechaza con todos.

P está contenida en NP

Teorema

$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{EXPTIME}$, donde $\mathrm{EXPTIME} = \bigcup_{k \geq 1} \mathrm{TIME}\left(2^{n^k}\right)$.

Primera inclusión. Sea M una máquina determinista que decide A en tiempo polinomial. Un decididor es un verificador que no necesita ayuda:

1. Con $\langle w, c \rangle$: ignorar c y ejecutar M con w.
2. Si M acepta, aceptar. Si rechaza, rechazar.

- Si $w \in A$: acepta con cualquier c; en particular, con alguno.
- Si $w \notin A$: rechaza con todos.
- Con la otra definición: una máquina determinista es una no determinista que nunca tiene más de una opción.

NP está contenida en EXPTIME

Sea V un verificador de A que da a lo sumo $p(n)$ pasos, con certificados sobre un alfabeto de g símbolos. Un decididor que prueba **todos** los certificados:

1. Para cada cadena c de longitud a lo sumo $p(n)$:
2. ejecutar V con $\langle w, c \rangle$. Si acepta, aceptar.
3. Si ninguna sirvió, rechazar.

NP está contenida en EXPTIME

Sea V un verificador de A que da a lo sumo $p(n)$ pasos, con certificados sobre un alfabeto de g símbolos. Un decididor que prueba **todos** los certificados:

1. Para cada cadena c de longitud a lo sumo $p(n)$:
2. ejecutar V con $\langle w, c \rangle$. Si acepta, aceptar.
3. Si ninguna sirvió, rechazar.

- Decide A: si algún certificado sirve, uno de longitud a lo sumo $p(n)$ sirve.

NP está contenida en EXPTIME

Sea V un verificador de A que da a lo sumo $p(n)$ pasos, con certificados sobre un alfabeto de g símbolos. Un decididor que prueba **todos** los certificados:

1. Para cada cadena c de longitud a lo sumo $p(n)$:
2. ejecutar V con $\langle w, c \rangle$. Si acepta, aceptar.
3. Si ninguna sirvió, rechazar.

- Decide A: si algún certificado sirve, uno de longitud a lo sumo $p(n)$ sirve.
- Hay a lo sumo $g^{p(n)+1}$ cadenas, y cada una cuesta $O(p(n))$:

$$O\left(p(n) \cdot g^{p(n)+1}\right) = 2^{O(p(n))}$$

NP está contenida en EXPTIME

Sea V un verificador de A que da a lo sumo $p(n)$ pasos, con certificados sobre un alfabeto de g símbolos. Un decididor que prueba **todos** los certificados:

1. Para cada cadena c de longitud a lo sumo $p(n)$:
2. ejecutar V con $\langle w, c \rangle$. Si acepta, aceptar.
3. Si ninguna sirvió, rechazar.

- Decide A: si algún certificado sirve, uno de longitud a lo sumo $p(n)$ sirve.
- Hay a lo sumo $g^{p(n)+1}$ cadenas, y cada una cuesta $O(p(n))$:

$$O\left(p(n) \cdot g^{p(n)+1}\right) = 2^{O(p(n))}$$

Es un programa conocido

La fuerza bruta. Está en `search.h`, para suma de subconjuntos, y llama al verificador.

Buscar frente a comprobar, contado

terminal

```
$ build/find_vs_check
Números 8 13 21 5 17 2, objetivo 28: certificado 1 2 4 6, hallado después de probar 44 de 64
Números 8 13 21 5 17 2, objetivo 4: ningún certificado sirve; se probaron los 64

Instancias sin solución: los números 2, 4, ..., 2n y un objetivo impar, cerca de la mitad de la suma
 n   certificados que prueba la búsqueda   sumas del verificador con un certificado
 4                      16                                    4
 8                     256                                    8
12                    4096                                   12
16                   65536                                   16
```

Buscar frente a comprobar, contado

terminal

```
$ build/find_vs_check
Números 8 13 21 5 17 2, objetivo 28: certificado 1 2 4 6, hallado después de probar 44 de 64
Números 8 13 21 5 17 2, objetivo 4: ningún certificado sirve; se probaron los 64

Instancias sin solución: los números 2, 4, ..., 2n y un objetivo impar, cerca de la mitad de la suma
 n   certificados que prueba la búsqueda   sumas del verificador con un certificado
 4                      16                                    4
 8                     256                                    8
12                    4096                                   12
16                   65536                                   16
```

- Si hay solución, la búsqueda puede tener suerte. Si no la hay, no hay atajo: para decir “no” hay que descartar los 2^n.

Buscar frente a comprobar, contado

terminal

```
$ build/find_vs_check
Números 8 13 21 5 17 2, objetivo 28: certificado 1 2 4 6, hallado después de probar 44 de 64
Números 8 13 21 5 17 2, objetivo 4: ningún certificado sirve; se probaron los 64

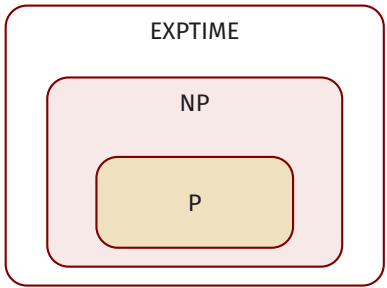
Instancias sin solución: los números 2, 4, ..., 2n y un objetivo impar, cerca de la mitad de la suma
 n   certificados que prueba la búsqueda   sumas del verificador con un certificado
 4                      16                                    4
 8                     256                                    8
12                    4096                                   12
16                   65536                                   16
```

- Si hay solución, la búsqueda puede tener suerte. Si no la hay, no hay atajo: para decir “no” hay que descartar los 2^n.
- Un número más: el doble de trabajo para buscar, una suma más para comprobar.

La pregunta P contra NP

¿P = NP?

¿Todo lenguaje con un verificador de tiempo polinomial es decidido por alguna máquina **determinista** de tiempo polinomial?



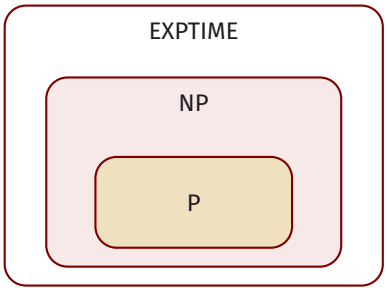
Alguna frontera es real. ¿Cuál?

La pregunta P contra NP

¿$P = NP$?

¿Todo lenguaje con un verificador de tiempo polinomial es decidido por alguna máquina **determinista** de tiempo polinomial?

- $P \subseteq NP \subseteq EXPTIME$: demostrado hoy.



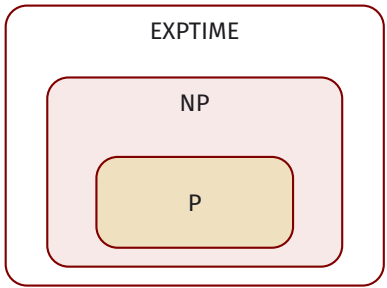
Alguna frontera es real. ¿Cuál?

La pregunta P contra NP

¿$P = NP$?

¿Todo lenguaje con un verificador de tiempo polinomial es decidido por alguna máquina **determinista** de tiempo polinomial?

- $P \subseteq NP \subseteq EXPTIME$: demostrado hoy.
- $P \neq EXPTIME$: teorema de jerarquía, semana 15.



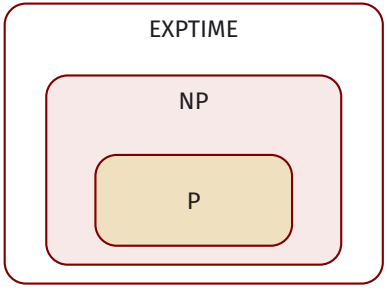
Alguna frontera es real. ¿Cuál?

La pregunta P contra NP

¿$P = NP$?

¿Todo lenguaje con un verificador de tiempo polinomial es decidido por alguna máquina **determinista** de tiempo polinomial?

- $P \subseteq NP \subseteq EXPTIME$: demostrado hoy.
- $P \neq EXPTIME$: teorema de jerarquía, semana 15.
- Entonces **alguna** de las dos inclusiones es estricta. No se sabe cuál.



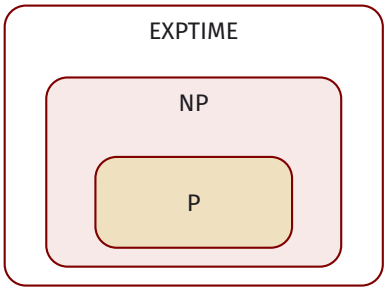
Alguna frontera es real. ¿Cuál?

La pregunta P contra NP

¿P = NP?

¿Todo lenguaje con un verificador de tiempo polinomial es decidido por alguna máquina **determinista** de tiempo polinomial?

- P $\subseteq$ NP $\subseteq$ EXPTIME: demostrado hoy.
- P $\neq$ EXPTIME: teorema de jerarquía, semana 15.
- Entonces **alguna** de las dos inclusiones es estricta. No se sabe cuál.
- Se cree que P $\neq$ NP. No hay demostración.

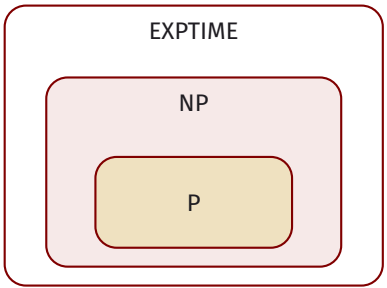


Alguna frontera es real. ¿Cuál?

La pregunta P contra NP

¿$P = NP$?

¿Todo lenguaje con un verificador de tiempo polinomial es decidido por alguna máquina **determinista** de tiempo polinomial?



Alguna frontera es real. ¿Cuál?

- $P \subseteq NP \subseteq EXPTIME$: demostrado hoy.
- $P \neq EXPTIME$: teorema de jerarquía, semana 15.
- Entonces **alguna** de las dos inclusiones es estricta. No se sabe cuál.
- Se cree que $P \neq NP$. No hay demostración.

Para $P = NP$ bastaría un algoritmo. Para $P \neq NP$ hay que descartar *todas* las máquinas polinomiales, incluidas las que nadie ha imaginado.

SECCIÓN 10

Lo que NP no es. coNP

10



Lo que NP no es

1. **No significa “no polinomial”.** La N es de *nondeterministic*. Todo problema de P está en NP.

Lo que NP no es

1. **No significa "no polinomial".** La N es de *nondeterministic*. Todo problema de P está en NP.
2. **No es "tiempo exponencial".** NP $\subseteq$ EXPTIME es una cota *superior*. De ningún problema de NP se ha demostrado que necesite tiempo exponencial.

Lo que NP no es

1. **No significa "no polinomial".** La N es de *nondeterministic*. Todo problema de P está en NP.
2. **No es "tiempo exponencial".** NP $\subseteq$ EXPTIME es una cota *superior*. De ningún problema de NP se ha demostrado que necesite tiempo exponencial.
3. **No es "difícil".** Los difíciles de NP son los NP-completos. COMPOSITES está en NP y en P.

Lo que NP no es

1. **No significa "no polinomial".** La N es de *nondeterministic*. Todo problema de P está en NP.
2. **No es "tiempo exponencial".** NP $\subseteq$ EXPTIME es una cota *superior*. De ningún problema de NP se ha demostrado que necesite tiempo exponencial.
3. **No es "difícil".** Los difíciles de NP son los NP-completos. COMPOSITES está en NP y en P.
4. **Es una clase de problemas de decisión.** "Hallar el camino más corto" no es un lenguaje.

Lo que NP no es

1. **No significa "no polinomial".** La N es de *nondeterministic*. Todo problema de P está en NP.
2. **No es "tiempo exponencial".** NP $\subseteq$ EXPTIME es una cota *superior*. De ningún problema de NP se ha demostrado que necesite tiempo exponencial.
3. **No es "difícil".** Los difíciles de NP son los NP-completos. COMPOSITES está en NP y en P.
4. **Es una clase de problemas de decisión.** "Hallar el camino más corto" no es un lenguaje.
5. **El certificado solo existe para las respuestas "sí".**

> **Pregunta**
>
> ¿Qué probaría que una fórmula **no** tiene ninguna asignación que la satisfaga?

coNP: el lado del no

- Una asignación que falla no prueba nada. Mil tampoco.

coNP: el lado del no

- Una asignación que falla no prueba nada. Mil tampoco.
- Lo único evidente: las 2^m asignaciones, todas fallando. No es de longitud polinomial.

coNP: el lado del no

- Una asignación que falla no prueba nada. Mil tampoco.
- Lo único evidente: las 2^m asignaciones, todas fallando. No es de longitud polinomial.
- No se conoce ningún certificado corto. Tampoco se ha demostrado que no exista.

coNP: el lado del no

- Una asignación que falla no prueba nada. Mil tampoco.
- Lo único evidente: las 2^m asignaciones, todas fallando. No es de longitud polinomial.
- No se conoce ningún certificado corto. Tampoco se ha demostrado que no exista.

coNP

La clase de los lenguajes cuyo complemento está en NP: $\text{coNP} = \{ A : \overline{A} \in \text{NP} \}$.
Son los lenguajes donde las respuestas **"no"** tienen demostraciones cortas.

coNP: el lado del no

- Una asignación que falla no prueba nada. Mil tampoco.
- Lo único evidente: las 2^m asignaciones, todas fallando. No es de longitud polinomial.
- No se conoce ningún certificado corto. Tampoco se ha demostrado que no exista.

coNP

La clase de los lenguajes cuyo complemento está en NP: $\text{coNP} = \{ A : \overline{A} \in \text{NP} \}$.
Son los lenguajes donde las respuestas **"no"** tienen demostraciones cortas.

Error común

coNP **no** es "lo que no está en NP". Es la clase de los complementos de los lenguajes de NP. Todo lenguaje de P está en las dos.

¿Por qué no basta intercambiar aceptar y rechazar?

Determinista: funciona.

- M decide A; al revés decide $\overline{A}$, en el mismo tiempo.
- P es cerrada bajo complemento.
- $\mathrm{P} \subseteq \mathrm{NP} \cap \mathrm{coNP}$.

¿Por qué no basta intercambiar aceptar y rechazar?

Determinista: funciona.

- M decide A; al revés decide $\overline{A}$, en el mismo tiempo.
- P es cerrada bajo complemento.
- $\mathrm{P} \subseteq \mathrm{NP} \cap \mathrm{coNP}$.

No determinista: falla.

- N' acepta si *alguna* rama de N rechaza.
- Pero $w \notin L(N)$ pide que *todas* rechacen.
- Para SAT: N' acepta las fórmulas con alguna asignación que falla. Casi todas.

¿Por qué no basta intercambiar aceptar y rechazar?

Determinista: funciona.

- M decide A; al revés decide $\overline{A}$, en el mismo tiempo.
- P es cerrada bajo complemento.
- $\mathrm{P} \subseteq \mathrm{NP} \cap \mathrm{coNP}$.

No determinista: falla.

- N' acepta si *alguna* rama de N rechaza.
- Pero $w \notin L(N)$ pide que *todas* rechacen.
- Para SAT: N' acepta las fórmulas con alguna asignación que falla. Casi todas.

Es lo que pasaba con los NFA en la semana 2. El “existe” de NP se vuelve un “para todo” al complementar.

¿Por qué no basta intercambiar aceptar y rechazar?

Determinista: funciona.

- M decide A; al revés decide $\overline{A}$, en el mismo tiempo.
- P es cerrada bajo complemento.
- $\mathrm{P} \subseteq \mathrm{NP} \cap \mathrm{coNP}$.

No determinista: falla.

- N' acepta si *alguna* rama de N rechaza.
- Pero $w \notin L(N)$ pide que *todas* rechacen.
- Para SAT: N' acepta las fórmulas con alguna asignación que falla. Casi todas.

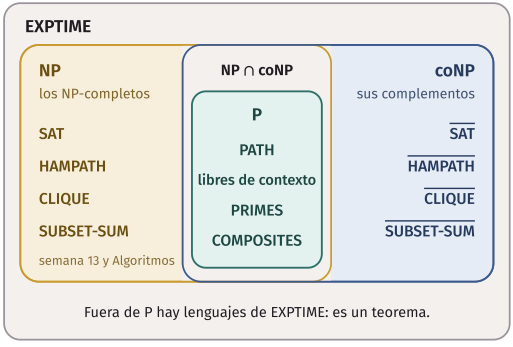
Es lo que pasaba con los NFA en la semana 2. El "existe" de NP se vuelve un "para todo" al complementar.

Lo que se sabe

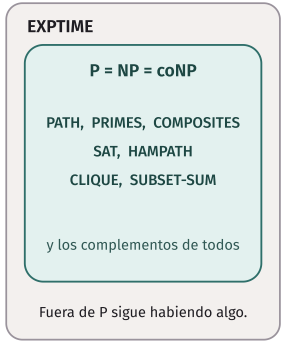
Si $\mathrm{P} = \mathrm{NP}$, entonces $\mathrm{NP} = \mathrm{coNP}$. Al revés: demostrar $\mathrm{NP} \neq \mathrm{coNP}$ demostraría $\mathrm{P} \neq \mathrm{NP}$.
No se sabe si $\mathrm{NP} = \mathrm{coNP}$. Se cree que no. Sigue en la semana 13.

El mapa: dos mundos posibles

Si P ≠ NP y NP ≠ coNP: lo que se cree



Si P = NP



Demostrado, y cierto en los dos dibujos: $P \subseteq NP \cap coNP$, $NP \subseteq EXPTIME$ y $P \neq EXPTIME$.

No se sabe cuál es el dibujo verdadero. Bastaría un algoritmo polinomial para SAT para estar en el de la derecha.

SECCIÓN 11

Laboratorio

11

Laboratorio: escribir verificadores

Tres problemas, en `code/week-12/lab/verifiers`: `hampath`, `sat` (formato DIMACS) y `subset_sum`.

1. Leer una instancia y varios certificados.
2. Responder `válido` o `inválido` por cada uno.
3. Rechazar los mal formados: un vértice repetido, una variable sin valor, un índice fuera de rango.
4. Pasar los casos grandes en tiempo lineal o casi lineal.
5. Responder cuántos bits tiene cada certificado.

Ojo: un verificador cuadrático sigue siendo de tiempo polinomial, pero da `TLE`. Y en `subset_sum` las sumas llegan a $5 \cdot 10^{18}$: `long long`.

subset_sum/casos/01.in

```
6 28
8 13 21 5 17 2
5
4 1 2 4 6
3 3 4 6
2 1 3
4 4 4 4 2
4 1 2 4 7
```

casos/01.out

```
válido
válido
inválido
inválido
inválido
```

Para la próxima clase

- Leer los apuntes de la semana 12 y terminar el laboratorio.
- Preguntas de repaso de la guía: por qué P es la misma con una cinta y con varias; si probar divisores hasta $\sqrt{x}$ es polinomial, y en función de qué; el certificado y el verificador de CLIQUE.
- Próxima semana: los problemas más difíciles de NP. El teorema de Cook-Levin, demostrado.

Pregunta

Todos los problemas de NP tienen algo en común: una máquina no determinista de tiempo n^k. ¿Cómo se podría describir con una fórmula booleana "esta máquina acepta w"?