

Teoría de la Computación

Autorreferencia

Quines, el teorema de recursión, la incompletitud de Gödel y la complejidad de Kolmogorov

Gerson Garrido

Semana 11

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A

1. La objeción a la semana 9
2. El intento ingenuo
3. La frase que se describe a sí misma
4. Dos partes: datos y código
5. Un quine en C++
6. El teorema de recursión y dos usos

Sesión B

7. El programa de Hilbert
8. Sistemas formales: verdadero y demostrable
9. El primer teorema de incompletitud
10. La afirmación de Gödel
11. Lo que el teorema no dice
12. Opcional: complejidad de Kolmogorov

> **Lectura**
>
> Apuntes de la semana 11. Apoyo: Sipser, sec. 6.1, 6.2 y 6.4; Barak, cap. 11. Código en `code/week-11`.

SECCIÓN 01

La objeción

01

Semana 9: la máquina que recibía su descripción

Si H decidiera A_{TM}:

```
D = "Con la entrada ⟨M⟩, donde M es una máquina de Turing:
  1. Correr H con la entrada ⟨M, ⟨M⟩⟩.
  2. Si H acepta, rechazar. Si H rechaza, aceptar."
```

- Con la entrada $\langle D \rangle$, D acepta si y solo si no acepta.
- La demostración es correcta: $\langle D \rangle$ es una cadena como cualquier otra.
- Pero a D la descripción se la dieron. Ella no la tenía.

La objeción

¿Puede una máquina **tener** su propia descripción y usarla en medio de un cálculo, sin que nadie se la entregue?

Dos respuestas malas y una buena

1. **“La lee de su archivo.”** Una máquina de Turing no tiene archivo: tiene estados, δ y una cinta con la entrada.
2. **“La lleva escrita dentro.”** Los estados que la escriben son parte de la máquina, y la descripción tendría que incluirlos.
3. **Lleva escrita una parte y calcula el resto.** Esa es la respuesta.

Quine

Programa que no lee nada (ni entrada ni archivos) y cuya salida es exactamente su propio código fuente.

El nombre es de Douglas Hofstadter (1979), por el filósofo W. V. O. Quine. La pregunta: ¿existe un quine?

SECCIÓN 02

El intento ingenuo

02

Un programa que imprime un programa

errors/prints_other.cpp

```cpp
#include <iostream>

int main() {
    std::cout << "#include <iostream>\n\nint main() {\n}\n";
}
```

terminal

```
$ build/prints_other
#include <iostream>

int main() {
}
$ build/prints_other | cmp - errors/prints_other.cpp
- errors/prints_other.cpp differ: byte 1, line 1
```

- Imprime un programa, pero no el suyo: falta la línea del `std::cout`.
- Si la agregamos al texto, la línea cambió, y hay que agregar la nueva.
- Las líneas 1 a 3 del archivo son un comentario, que tampoco sale.

La cadena no se cierra

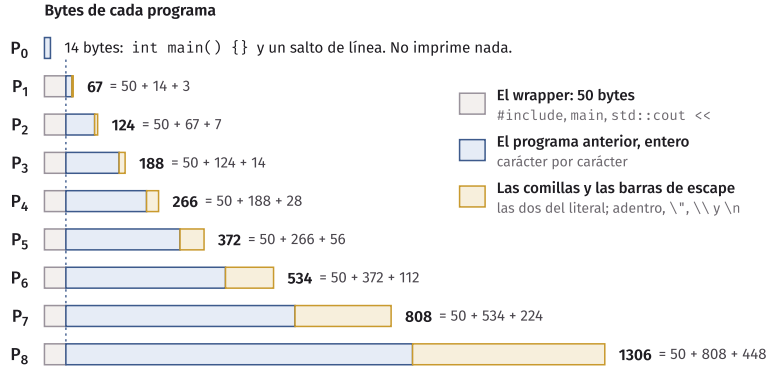
P_{k+1} es el programa que imprime el código de P_k.

terminal

```
$ build/naive_growth
 k   bytes de Pk   lo que imprime Pk
 0            14   nada
 1            67   P0, de 14 bytes
 2           124   P1, de 67 bytes
 3           188   P2, de 124 bytes
 4           266   P3, de 188 bytes
 5           372   P4, de 266 bytes
 6           534   P5, de 372 bytes
 7           808   P6, de 534 bytes
 8          1306   P7, de 808 bytes
```

- Cada uno lleva al anterior entre comillas.
- Las comillas del anterior se escriben \", y sus barras, \\: cada nivel es más largo.
- Cada uno imprime al anterior. Ninguno se imprime a sí mismo.

Cada uno lleva al anterior, y algo más



Cada programa lleva escrito al anterior y algo más, así que es más largo que lo que imprime.

P_{k+1} imprime a P_k, y ninguno se imprime a sí mismo: la cadena no se cierra.

Por qué no se cierra nunca

Proposición

Ningún programa de la forma “imprimir este literal” es un quine.

- La salida es el texto que el literal denota, y no es más larga que el literal.
- El literal es solo una parte del programa: faltan las comillas y la instrucción.
- La salida es más corta que el programa. No puede ser igual a él. ■

Idea clave

Un programa no puede llevar escrito su código completo. La salida de un quine tiene que salir, en parte, de un **cálculo**.

SECCIÓN 03

La frase que se describe a sí misma

03

Un quine en español

Instrucciones en una hoja. Al obedecerlas, ¿se escribe la misma hoja?

- **Escriban esta frase.** Trampa: “esta” señala desde afuera, como abrir el archivo.
- **Escriban lo que sigue: «hola»** Escribe *hola*. Es `prints_other`.

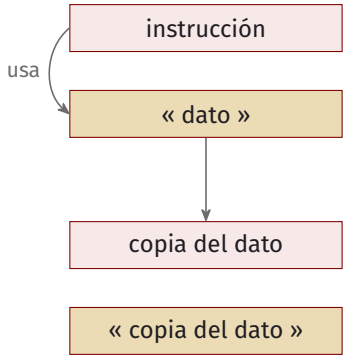
La frase

Escriban dos veces lo que sigue, la segunda vez entre comillas: «Escriban dos veces lo que sigue, la segunda vez entre comillas:»

- Lo que sigue: *Escriban dos veces lo que sigue, la segunda vez entre comillas:*
- Una vez tal cual, otra entre comillas: sale la frase completa, letra por letra.

Por qué funciona

- **Dos partes.** Afuera de las comillas, una instrucción: se obedece. Adentro, un dato: se copia.
- **El mismo texto** en las dos. Las comillas dicen cómo se usa: fuera se *usa*, dentro se *menciona*.
- La instrucción nunca habla de sí misma. Solo dice qué hacer con el dato.
- La frase contiene **la mitad** de sí misma. La instrucción produce las dos mitades: una copiando, la otra copiando y poniendo comillas.



SECCIÓN 04

Dos partes: datos y código

04

La herramienta: poner entre comillas

Lema

Hay una función computable q tal que, para toda cadena w, $q(w) = \langle P_w \rangle$: la descripción de una máquina P_w que, con cualquier entrada, escribe w en la cinta y se detiene.

```
Q = "Con la entrada w:
  1. Construir P_w = "Con cualquier entrada: borrar la cinta, escribir w y detenerse."
  2. Escribir ⟨P_w⟩ en la cinta y detenerse."
```

- q no escribe w: escribe **la descripción de** una máquina que escribe w.
- En C++ es `printer_of`: de un texto al código de un programa que lo imprime.

La máquina *SELF*

Primero B, que no menciona a A:

```
B = "Con la entrada ⟨M⟩, donde M es una máquina de Turing:
  1. Calcular q(⟨M⟩). Es la descripción de una máquina que escribe ⟨M⟩.
  2. Unir esa máquina con M: primero la que se acaba de calcular, después M.
  3. Escribir en la cinta la descripción de la máquina unida y detenerse."
```

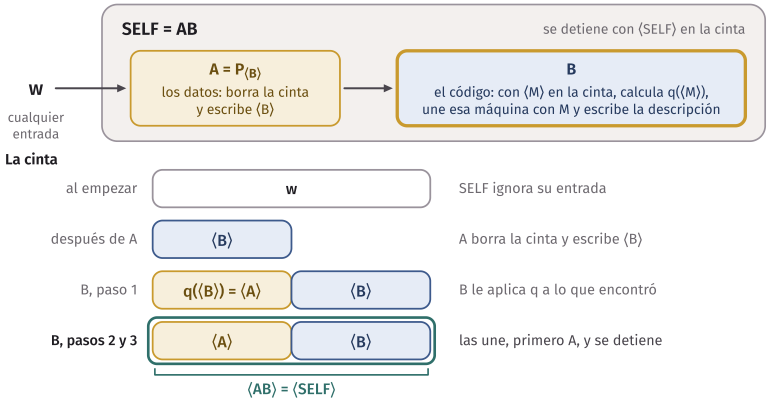
Después A, a partir de B: $A = P_{\langle B \rangle}$, la máquina que escribe $\langle B \rangle$. $\langle A \rangle = q(\langle B \rangle)$.

Y las dos en orden: $SELF = AB$.

El orden importa

B no depende de nada, y A solo de B. "A escribe $\langle SELF \rangle$" sería el intento ingenuo.

SELF escribe ⟨*SELF*⟩: la ejecución



B une la máquina calculada con la suya y escribe la descripción: ⟨AB⟩ = ⟨SELF⟩.

En la cinta quedó la descripción de la máquina de arriba. Ninguna parte se contiene a sí misma.

Qué lleva escrito cada parte

- ¿Cómo consigue *B* a la parte *A*, si no la conoce? Sabe **cómo se fabrica**: *A* es la máquina que escribe lo que *B* encuentra en la cinta. Para reconstruirla, le aplica *q* a lo que encontró.
- A sí misma, *B* no tiene que reconstruirse: su descripción es justamente lo que *A* le dejó.

> **Idea clave**
>
> *A* lleva escrita la descripción de *B*, y solo esa. *B* no lleva escrita ninguna: recibe la suya de *A* y **calcula** la de *A*. Ninguna se contiene a sí misma.

Tres veces la misma idea

	En la frase	**En *SELF***	**En el quine de C++**
Los datos	El texto entre comillas	*A*: escribe ⟨*B*⟩	El literal `text`
El código	La instrucción de afuera	*B*: reconstruye a *A* y la une consigo	Las líneas que imprimen
Las comillas	« »	*q*	Escribir `text` como literal

En la práctica

En una célula, el ADN es *A*: una descripción. La maquinaria es *B*: lee la descripción para construir, y aparte la copia sin interpretarla. Von Neumann llegó a esa división a fines de los años cuarenta.

SECCIÓN 05

Un quine en C++

05



El molde: el programa con un hueco

templates/quine.tpl

```
1  // Un quine: imprime exactamente su propio código fuente, sin abrir ningún archivo.
2  #include <cstddef>
3  #include <iostream>
4  #include <string>
5
6  int main() {
7      // Parte A: los datos. Es el texto de todo el programa, con una arroba en el lugar
8      // donde va este mismo literal.
9      const std::string text = @;
10
11     // Parte B: el código. Imprime lo que hay antes de la arroba, después el texto otra
12     // vez, ahora escrito como literal, y al final lo que hay después de la arroba.
13     const std::size_t hole = text.find('@');
14     std::cout << text.substr(0, hole);
15     std::cout << "R\"~(" << text << ")~\"";
16     std::cout << text.substr(hole + 1);
17     return 0;
18 }
```

El programa completo: en el lugar de la arroba de la línea 9, un literal con **este mismo texto**.

`quine.cpp`: 36 líneas, el molde dos veces

- Líneas 1 a 9: el molde hasta `text = .`
- Líneas 9 a 27: `R"~( ... )~"`, el molde entero como datos. Es la **parte** *A*.
- Líneas 27 a 36: el resto del molde. La **parte** *B* son las líneas 31 a 34.

terminal

```
$ build/quine | cmp - quine.cpp && echo iguales
iguales
$ wc -c templates/quine.tpl quine.cpp
 672 templates/quine.tpl
1350 quine.cpp
2022 total
```

- $2 \cdot 672 - 1 + 7 = 1350$: dos veces el molde, menos la arroba, más las dos marcas.
- El literal no calcula nada: deja un texto en una variable. Como *A*.

La parte *B*: cuatro líneas

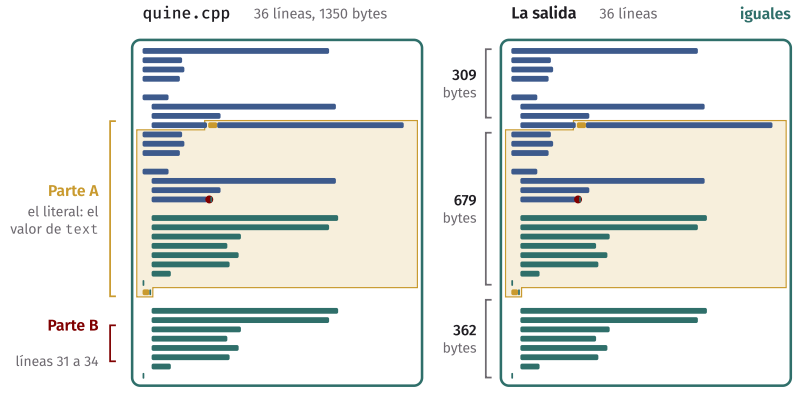
quine.cpp, líneas 31 a 34

```
1       const std::size_t hole = text.find('@');
2       std::cout << text.substr(0, hole);
3       std::cout << "R\"~(" << text << ")~\"";
4       std::cout << text.substr(hole + 1);
```

1. Busca la **primera** arroba de `text`: el hueco. La de `find('@')` está más adelante.
2. Imprime lo que hay antes del hueco: las líneas 1 a 8 y el comienzo de la 9.
3. Imprime `text` **como literal**: la marca, el texto, la marca. Es *q*: poner entre comillas.
4. Imprime lo que hay después del hueco: de la línea 27 al final.

En C++: entre `R"~(` y `)~"` no hay escapes. Poner entre comillas es pegar una marca adelante y otra atrás.

El quine corriendo



La salida tiene 36 líneas y 309 + 679 + 362 = 1350 bytes: es `quine.cpp`, byte por byte.

El molde salió dos veces, las dos del mismo `text`: una como código y otra como literal.

La construcción es mecánica

self.h

```
1  inline std::string fill_hole(const std::string& tpl) {
2      const std::size_t hole = tpl.find('@');
3      if (hole == std::string::npos) {
4          throw std::invalid_argument("el molde no tiene una arroba");
5      }
6      return tpl.substr(0, hole) + raw_literal(tpl) + tpl.substr(hole + 1);
7  }
```

terminal

```
$ build/make_self < templates/quine.tpl | cmp - quine.cpp && echo iguales
iguales
```

- `make_self` hace desde afuera lo que el quine hace cada vez que corre.
- Del molde al programa hay una **función computable**: de ahí sale el teorema.

La copia de la copia

terminal

```
$ build/quine > build/copy.cpp
$ g++ -Wall -Wextra -std=c++20 build/copy.cpp -o build/copy
$ build/copy | cmp - quine.cpp && echo iguales
iguales
```

- La salida del quine es un programa. Compilado, es otro quine igual al primero.
- `build/copy` se construyó sin `quine.cpp`: todo pasó por la salida.

Error común

Creer que un quine “hace trampa” leyendo su archivo. No abre nada: funciona igual si se borra el fuente.

Leer el archivo no es un quine

errors/reads_file.cpp: __FILE__ es la ruta del fuente

```cpp
int main() {
    std::ifstream source(__FILE__);
    if (!source) {
        std::cerr << "no encuentro mi código fuente\n";
        return 1;
    }
    std::cout << source.rdbuf();
    return 0;
}
```

terminal

```
$ build/reads_file | cmp - errors/reads_file.cpp && echo iguales
iguales
$ cd build && ./reads_file; cd ..
no encuentro mi código fuente
```

SECCIÓN 06

El teorema de recursión

06



El teorema de recursión *(recursion theorem)*

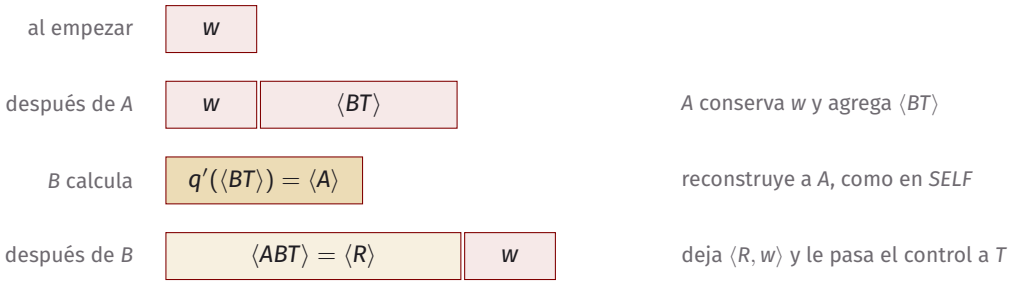
Un quine imprime su código y termina. ¿Se puede **seguir calculando** con él?

Teorema

Sea T una máquina de Turing cuya entrada es un par $\langle M, w \rangle$. Existe una máquina R tal que, para toda w, R con la entrada w hace lo mismo que T con la entrada $\langle R, w \rangle$: acepta, rechaza, se detiene con el mismo resultado, o no se detiene.

- Si T calcula una función t, entonces R calcula r con $r(w) = t(\langle R \rangle, w)$.
- T es una máquina común: para ella la descripción es un dato más.
- R es T con su **propia** descripción en ese lugar.

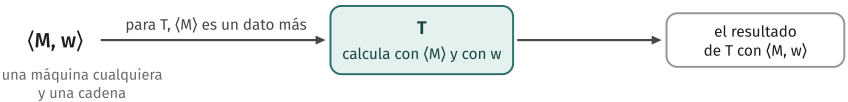
La demostración: $R = ABT$



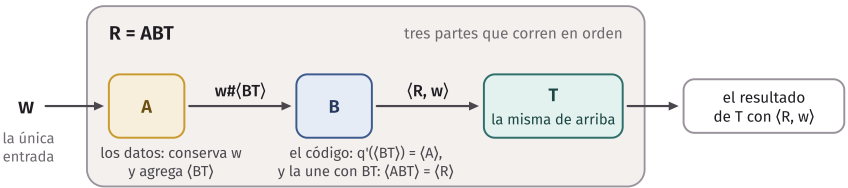
- Dos cambios respecto de *SELF*: la entrada w no se pierde, y al final corre T.
- Los datos, A, describen a **todo el resto**: BT. En `quine.cpp`, el literal contiene todo lo que no es el literal.

T tiene dos entradas; *R*, una sola

Se escribe T, una máquina común con dos entradas



El teorema entrega R, con una sola entrada



R con la entrada w hace lo mismo que T con la entrada ⟨R, w⟩: acepta, rechaza, deja la misma cadena o no se detiene.

Nadie le dio ⟨R⟩ a R. A lleva escrita la descripción del resto, ⟨BT⟩, y B calcula la de A.

Cómo se usa

El teorema autoriza un paso nuevo en las descripciones de alto nivel:

```
M = "Con la entrada w:
  1. Obtener, por el teorema de recursión, la propia descripción ⟨M⟩.
  2. ... seguir calculando con ⟨M⟩ y con w ..."
```

- La justificación: T recibe $\langle X, w \rangle$ y hace los pasos desde el 2 usando X. El teorema da la R que los hace con $\langle R \rangle$.
- Con su descripción, una máquina puede medirla, compararla, modificarla.
- Y puede **simularse a sí misma**, con la máquina universal de la semana 9.

Idea clave

"Obtener la propia descripción" es un paso tan válido como "simular otra máquina" o "copiar la entrada".

En C++: obtener el propio código

self_check.cpp (del molde)

```cpp
// Devuelve el código de este programa. Es la construcción del quine, sin imprimir.
std::string own_source() {
    const std::string text = @;
    const std::size_t hole = text.find('@');
    return text.substr(0, hole) + "R\"~(" + text + ")~\"" + text.substr(hole + 1);
}
```

- Es la parte *B* del quine, que ahora **devuelve** el texto en vez de imprimirlo.
- El resto del programa recibe su código en una variable y hace con él lo que quiera.
- Ese resto es la máquina *T* del teorema. Y `make_self` es el paso de *T* a *R*: el mismo para cualquier *T*.

En C++: el programa que se reconoce

self_check.cpp (del molde)

```cpp
int main() {
    const std::string self = own_source();
    const std::string input{std::istreambuf_iterator<char>(std::cin), std::istreambuf_iterator<char>()};
    if (input == self) {
        std::cout << "es mi código: " << self.size() << " bytes\n";
    } else {
        std::cout << "no es mi código: yo tengo " << self.size() << " bytes y la entrada " << input.size() << "\n";
    }
```

terminal

```
$ build/self_check < self_check.cpp
es mi código: 1890 bytes
$ build/self_check < quine.cpp
no es mi código: yo tengo 1890 bytes y la entrada 1350
```

Primer uso: A_{TM} es indecidible, en cuatro líneas

Supongamos que H decide A_{TM}.

```
B = "Con la entrada w:
  1. Obtener, por el teorema de recursión, la propia descripción ⟨B⟩.
  2. Correr H con la entrada ⟨B, w⟩.
  3. Si H acepta, rechazar. Si H rechaza, aceptar."
```

- Si B acepta w: H acepta $\langle B, w\rangle$, y por el paso 3, B rechaza w.
- Si B no acepta w: H rechaza $\langle B, w\rangle$, y por el paso 3, B acepta w.
- Contradicción con **toda** entrada. H no existe. ■

La respuesta a la objeción

Nadie le da nada a B: consigue su descripción, pregunta qué va a hacer, y hace lo otro.

Segundo uso: ¿es esta la máquina más pequeña?

Máquina mínima

M es **mínima** si ninguna máquina con descripción más corta reconoce el mismo lenguaje. $\mathrm{MIN}_{\mathrm{TM}} = \{ \langle M \rangle : M \text{ es mínima} \}$

- Para los DFA hay un algoritmo: la minimización de la semana 4.
- **Un intento:** revisar todas las máquinas más cortas que M (son finitas) y ver si alguna reconoce el mismo lenguaje.
- Falla en "ver si reconoce el mismo lenguaje": es $\mathrm{EQ}_{\mathrm{TM}}$, que no es ni reconocible.
- Eso solo dice que *este* algoritmo falla. ¿Habrá otro?
- Rice no sirve: ser mínima no es una propiedad del lenguaje. Dos máquinas con el mismo lenguaje pueden ser una mínima y la otra no.

$\mathrm{MIN}_{\mathrm{TM}}$ no es reconocible

Teorema

Ningún subconjunto infinito de $\mathrm{MIN}_{\mathrm{TM}}$ es reconocible. En particular, $\mathrm{MIN}_{\mathrm{TM}}$ no es reconocible ni decidible.

Sea $S \subseteq \mathrm{MIN}_{\mathrm{TM}}$ infinito y reconocible, y E un enumerador de S (semana 7).

```
C = "Con la entrada w:
  1. Obtener, por el teorema de recursión, la propia descripción ⟨C⟩.
  2. Correr el enumerador E hasta que imprima una descripción ⟨D⟩ más larga que ⟨C⟩.
  3. Simular D con la entrada w. Si D acepta, aceptar. Si D rechaza, rechazar."
```

C es corta y fija. Espera a que aparezca una máquina más grande que ella, y se hace pasar por esa.

$\mathrm{MIN}_{\mathrm{TM}}$ no es reconocible: la demostración

1. **El paso 2 termina.** S es infinito, y solo hay finitas cadenas de cada longitud: S tiene una cadena más larga que $\langle C \rangle$, y E la imprime.
2. D **es la misma para toda** w**.** El paso 2 no mira la entrada.
3. $L(C) = L(D)$**.** Por el paso 3, C acepta w si y solo si D acepta w. Si D no se detiene, C tampoco.
4. **Contradicción.** $|\langle C \rangle| < |\langle D \rangle|$ y reconocen lo mismo: D no es mínima. Pero $\langle D \rangle \in S \subseteq \mathrm{MIN}_{\mathrm{TM}}$. ■

- $\mathrm{MIN}_{\mathrm{TM}}$ es infinito: hay infinitos lenguajes reconocibles, y cada uno tiene una máquina de descripción más corta.
- En la práctica: no existe el optimizador que entrega siempre el programa más corto equivalente.

Un quine con payload

quine_payload.cpp (del molde)

```cpp
int main() {
    const std::string self = own_source();
    // La copia.
    std::cout << self;
    // El payload.
    const std::ptrdiff_t lines = std::count(self.begin(), self.end(), '\n');
    std::cerr << "payload: me copié completo, " << lines << " líneas y " << self.size() << " bytes\n";
    return 0;
}
```

terminal

```
$ build/quine_payload > build/copy2.cpp
payload: me copié completo, 48 líneas y 1898 bytes
$ cmp build/copy2.cpp quine_payload.cpp && echo iguales
iguales
```

Virus: un quine con algo más

- **Replicación:** el programa se copia. **Payload:** lo que hace además de copiarse.
- La copia es exacta, incluido el payload: compilada, vuelve a copiarse y a ejecutarlo.
- Para un virus real, la copia va dentro de otro programa y el payload hace daño. No hay teoría nueva.
- El teorema de recursión garantiza que la parte que se copia siempre se puede escribir.

En la práctica

Ken Thompson, *Reflections on Trusting Trust* (1984): un compilador que, al compilarse, vuelve a agregarse una puerta trasera que no está en su código fuente. La conferencia empieza con un quine.

SECCIÓN 07

El programa de Hilbert

07

El programa de Hilbert

Escribir toda la matemática en un **sistema formal**: axiomas y reglas tan precisos que comprobar una demostración sea mecánico. Hilbert esperaba tres cosas:

1. **Completo:** toda afirmación se puede demostrar o refutar.
2. **Consistente:** nunca demuestra una afirmación y su negación. Y que eso se pueda demostrar.
3. **Decidible:** un procedimiento dice si una afirmación tiene demostración. El *Entscheidungsproblem* (1928).

Las tres resultaron falsas

Las dos primeras, con Gödel en 1931. La tercera, con Church y Turing en 1936. Para demostrar que *ningún* procedimiento decide, Turing definió sus máquinas.

Nosotros vamos al revés: ya tenemos el problema de la parada, y la incompletitud sale en media página.

Afirmaciones de la aritmética

Números naturales con $+$ y $\times$, variables, $=$, conectores y cuantificadores. Una afirmación es una **cadena**.

- $\forall x\, \exists y\, (y = x + 1)$ — verdadera
- $\exists x\, (x + x = 1 + 1 + 1)$ — falsa
- $\text{primo}(p) = (1 < p) \wedge \neg \exists a\, \exists b\, (1 < a \wedge 1 < b \wedge a \times b = p)$
- $\forall n\, \exists p\, (n < p \wedge \text{primo}(p))$ — verdadera (Euclides)
- $\forall n\, \exists p\, (n < p \wedge \text{primo}(p) \wedge \text{primo}(p + 1 + 1))$ — nadie sabe

Verdadera o falsa

Cada afirmación habla de los números naturales, y ahí las cosas son de una manera. La verdad no depende de ningún sistema de axiomas ni de que alguien la haya demostrado.

SECCIÓN 08

Sistemas formales

08

Sistema formal: el comprobador

- **Axiomas** y **reglas de inferencia**. Una demostración: una lista finita de afirmaciones, cada una un axioma o consecuencia de las anteriores.
- Ejemplos: la aritmética de Peano, la teoría de conjuntos ZFC.

Lo único que usaremos

Un **sistema formal** S está dado por un **comprobador** V: una máquina que siempre se detiene y que, con una afirmación φ y una cadena π, acepta si π es una demostración de φ en S.

φ es **demostrable** si alguna π lo es.

- Se exige que **comprobar** sea mecánico. No que **encontrar** la demostración lo sea.

Verdadero no es lo mismo que demostrable

Propiedad	Qué dice	De qué habla
Correcto *(sound)*	Todo lo demostrable es verdadero	Del sistema y de los números
Consistente	Nunca demuestra φ y también $\neg\varphi$	Solo del sistema
Completo	Para toda φ, demuestra φ o demuestra $\neg\varphi$	Solo del sistema

- Correcto $\Rightarrow$ consistente: φ y $\neg\varphi$ no son las dos verdaderas. Al revés no vale.

Idea clave

Verdadero habla de los números. **Demostrable** habla de cadenas: existe una que el comprobador acepta. Un sistema correcto garantiza demostrable $\Rightarrow$ verdadero. Hilbert preguntaba por la otra dirección.

Lo demostrable es reconocible

Lema

Para todo sistema formal S, el conjunto de las afirmaciones demostrables en S es reconocible.

```
B_S = "Con la entrada φ:
  1. Para cada cadena π, en orden de longitud:
  2.     Correr el comprobador V con ⟨φ, π⟩. Si acepta, aceptar."
```

- Si φ tiene demostración, aparece en algún momento. Si no, la búsqueda no termina.
- Es la búsqueda con la que reconocimos PCP en la semana 10, con el mismo defecto.
- Si además el sistema fuera correcto y *completo*: buscando una demostración de φ o de $\neg\varphi$ se **decidiría** qué es verdadero.

La aritmética puede hablar de máquinas

Lema (solo la idea)

Hay una función computable que, de $\langle M, w \rangle$, produce una afirmación de la aritmética $\varphi_{M,w}$ con

$$\varphi_{M,w} \text{ es verdadera} \iff M \text{ no se detiene con la entrada } w.$$

1. **Las cadenas son números**: una configuración, leída en una base adecuada.
2. **Un paso es una relación entre números**: comprobar que una configuración sigue a otra es local (semana 10), y los dígitos se sacan con cocientes y restos.
3. **Detenerse es que exista una historia de cómputo**, que es un número h:
   $\varphi_{M,w} = \neg\exists h$ (h es una historia de M con w que termina)

"Se detiene": *existe* un número que se puede comprobar. "No se detiene": *ninguno* sirve. Ahí está la dificultad.

SECCIÓN 09

El primer teorema de incompletitud

09

El primer teorema de incompletitud

Teorema (para sistemas correctos)

Sea S un sistema formal que

1. es **efectivo**: un comprobador que siempre se detiene revisa las demostraciones;
2. **expresa la aritmética**: entre sus afirmaciones están las $\varphi_{M,w}$;
3. es **correcto**: toda afirmación de la aritmética demostrable en S es verdadera.

Entonces hay una afirmación de la aritmética verdadera y sin demostración en S.

- Más preciso: hay M y w tales que M no se detiene con w, y $\varphi_{M,w}$, que lo dice, no es demostrable.
- **Corolario:** S no es completo. $\varphi_{M,w}$ no se demuestra, y $\neg\varphi_{M,w}$ es falsa, así que tampoco.

La demostración: buscar demostraciones decidiría la parada

Supongamos que no: siempre que M no se detiene con w, $\varphi_{M,w}$ tiene demostración.

```
P = "Con la entrada ⟨M, w⟩:
  1. Construir la afirmación φ_{M,w}.
  2. Para k = 1, 2, 3, ...:
  3.     Simular M con w durante k pasos. Si se detiene, aceptar.
  4.     Correr V con φ_{M,w} y cada una de las primeras k cadenas.
        Si alguna es una demostración de φ_{M,w}, rechazar."
```

- M **se detiene:** $\varphi_{M,w}$ es falsa; S es correcto, así que no tiene demostración. El paso 3 acepta.
- M **no se detiene:** $\varphi_{M,w}$ es verdadera y, por lo supuesto, demostrable. El paso 4 rechaza.
- P decide $\mathrm{HALT}_{\mathrm{TM}}$. Contradicción. ■

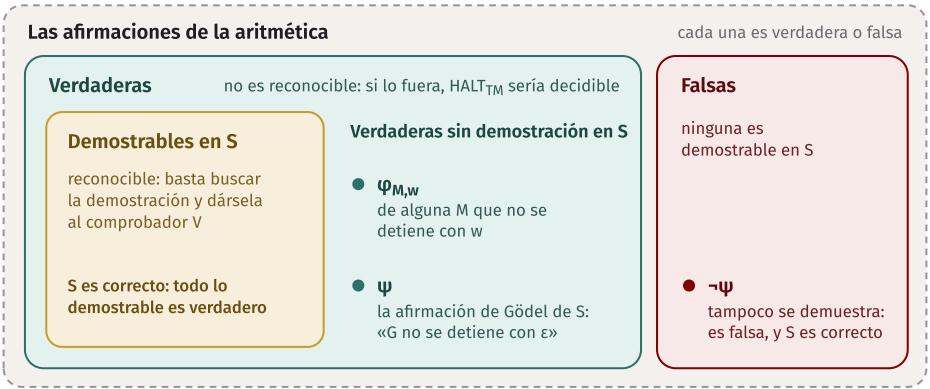
El argumento en tres líneas

Idea clave

Lo demostrable es **reconocible**: basta buscar la demostración.
Lo verdadero **no lo es**: si lo fuera, se reconocería cuándo una máquina *no* se detiene.
En un sistema correcto, lo demostrable está dentro de lo verdadero. Sobran verdades.

- Un conjunto reconocible y uno que no lo es no pueden ser iguales.
- Si S es correcto, la diferencia solo puede estar de un lado: verdades sin demostración.

Demostrable, dentro de verdadero



Un conjunto reconocible dentro de uno que no lo es: no pueden ser iguales.

Lo que sobra son verdades que S no demuestra. S no es completo.

Dos observaciones

- **Agregar axiomas no lo arregla.** Si φ es verdadera y no demostrable, se la puede agregar como axioma. El sistema nuevo sigue siendo efectivo, sigue expresando la aritmética y sigue siendo correcto: tiene su propia verdad sin demostración.
- **El teorema habla de programas.** Para cada sistema que cumple las condiciones hay un programa concreto que no termina, y cuya no terminación ese sistema no puede demostrar.
- La demostración prueba que la afirmación existe, pero no la muestra. Ahora construimos una.

Las tres hipótesis hacen falta

Hipótesis	Dónde se usó	Sin ella
Efectivo	P busca demostraciones	Axiomas: todas las verdades. Completo, e inútil: nadie comprueba qué es un axioma
Expresa la aritmética	Escribir $\varphi_{M,w}$	Hay sistemas completos para lenguajes más pobres: la aritmética con suma y sin producto
Correcto	Lo falso no tiene demostración	Un sistema contradictorio demuestra todo: es completo

SECCIÓN 10

La afirmación de Gödel

10

La máquina que busca su propia demostración

```
G = "Con cualquier entrada:
  1. Obtener, por el teorema de recursión, la propia descripción ⟨G⟩.
  2. Construir la afirmación ψ = φ_{G,ε}: "G no se detiene con la entrada ε".
  3. Para cada cadena π, en orden de longitud:
  4.     Correr V con ⟨ψ, π⟩. Si π es una demostración de ψ, detenerse."
```

- Por los pasos 3 y 4: G se detiene con $\varepsilon \iff \psi$ tiene demostración en S.
- Por cómo se hizo ψ: ψ es verdadera $\iff$ G no se detiene con ε.

La afirmación de Gödel

$$\psi \text{ es verdadera} \iff \psi \text{ no tiene demostración en } S.$$

Es “esta afirmación no tiene demostración”. La palabra *esta* la puso el teorema de recursión.

ψ es verdadera y no tiene demostración

Teorema

Si S cumple las tres condiciones, ψ es verdadera y no es demostrable en S.

- Si ψ tuviera demostración, G la encontraría y se detendría.
- Entonces ψ, que dice que G no se detiene, sería falsa: S demostraría algo falso. Pero S es correcto.
- Luego ψ no tiene demostración. G busca para siempre, no se detiene, y ψ es verdadera. ■

¿No acabamos de demostrar ψ?

Sí, pero no en S: usamos que S es correcto. "Indemostrable" siempre es **en un sistema dado**.

Qué demostró Gödel exactamente

- Nuestra hipótesis: S es **correcto**. Habla de la verdad en los números naturales.
- El teorema original pide menos: que S sea **consistente** y que *demuestre* los hechos elementales de la aritmética. La conclusión es la misma: S no es completo.
- Gödel pedía algo más que consistencia; Rosser lo redujo a la consistencia sola en 1936. Esa demostración es más fina y no la vemos.

Segundo teorema de incompletitud

Un sistema formal efectivo y consistente que contiene la aritmética de Peano no puede demostrar la afirmación que expresa su propia consistencia.

SECCIÓN 11

Lo que el teorema no dice

11

Lo que el teorema no dice

- **No dice que "nada se puede demostrar".** Lo demostrado sigue demostrado. Ningún sistema fijo agota las verdades de la aritmética.
- **No hay verdades "absolutamente indemostrables".** ψ depende de S, y se demuestra en sistemas más fuertes.
- **No habla de todos los sistemas.** La aritmética de Presburger (suma, sin producto) es completa y decidible. Su lenguaje no describe máquinas.
- **No habla de la mente humana.** Vimos algo condicional: *si* S es correcto, ψ es verdadera. El teorema no dice que una persona pueda saber eso de todo sistema.
- **No son solo rarezas autorreferentes.** El teorema de Goodstein no se demuestra en la aritmética de Peano; la hipótesis del continuo no se decide en ZFC.
- **No es el teorema de completitud** (Gödel, 1929), donde "completo" significa otra cosa.

Incompleto no es contradictorio

Error común

Usar “incompleto” como si quisiera decir “contradictorio”. Son propiedades distintas, y el teorema las enfrenta.

- **Inconsistente:** demuestra φ y $\neg\varphi$. Demuestra *de más*.
- **Incompleto:** para alguna φ, no demuestra ni φ ni $\neg\varphi$. Demuestra *de menos*.
- Un sistema inconsistente demuestra todo, así que es completo.
- Para un sistema efectivo que contiene la aritmética hay que elegir: **si es consistente, es incompleto**.

Las tres esperanzas de Hilbert, hoy

Para un sistema efectivo que contiene la aritmética:

Esperanza	Resultado	De dónde sale
Completo	No, si es consistente	Primer teorema de incompletitud
Demuestra su propia consistencia	No, si es consistente	Segundo teorema de incompletitud
Un algoritmo decide qué es demostrable	No, si es consistente	Church y Turing, 1936: es el problema de la parada

La incompletitud no delata una contradicción

Es el precio de no tenerla.

SECCIÓN 12

Opcional: complejidad de Kolmogorov

12



Opcional: ¿cuánta información tiene una cadena?

dos cadenas de 40 símbolos

```
0101010101010101010101010101010101010101
1010001100011110010010111100110000001000
```

- La primera se dicta en cuatro palabras: “veinte veces cero uno”. La segunda hay que leerla entera.

Complejidad de Kolmogorov *(Kolmogorov complexity)*

Una **descripción** de x es un par $\langle M, w\rangle$ tal que M con la entrada w se detiene con x en la cinta.
$K(x)$ es la longitud de la descripción más corta de x.

- Un programa y unos datos, como en el quine. $\quad |\langle M, w\rangle| = 2\,|\langle M\rangle| + 2 + |w|$.

Opcional: cotas superiores e invariancia

- $K(x) \leq |x| + c$: la máquina que no hace nada, con x como datos.
- $K(\mathtt{o}^n) \leq \log_2 n + c$: la máquina que lee n en binario y escribe n ceros.

Teorema de invariancia

Para todo lenguaje de descripción computable p hay una constante c_p tal que, para toda x, $\quad K(x) \leq K_p(x) + c_p$.

- La constante es el intérprete de p, y se paga una sola vez.
- Un compresor es un lenguaje de descripción: el tamaño comprimido es una **cota superior** de K.

Opcional: cadenas incompresibles

Teorema

Para todo n hay una cadena de longitud n con $K(x) \geq n$: una cadena **incompresible**.

- Cadenas de longitud n: 2^n. Descripciones más cortas que n: $2^0 + \cdots + 2^{n-1} = 2^n - 1$.
- Cada descripción describe a lo sumo una cadena. Alguna de las 2^n se queda sin descripción corta. ■
- El mismo conteo: menos de una fracción 2^{1-c} cumple $K(x) \leq n - c$.

Error común

Ningún compresor achica todos los archivos: si unas cadenas salen más cortas, otras salen más largas.

Opcional: K no es computable

La paradoja de Berry

"El menor número natural que no se puede describir con menos de veinte palabras." Tiene catorce.

Si una máquina calculara K:

```
N = "Con la entrada m, un número escrito en binario:
  1. Para cada cadena x, en orden de longitud:
  2.     Calcular K(x).
  3.     Si K(x) ≥ m, escribir x en la cinta y detenerse."
```

- N se detiene: hay cadenas con $K(x) \geq m$. Sea x_m su salida: $K(x_m) \geq m$.
- Pero $\langle N, m \rangle$ describe a x_m: $K(x_m) \leq c + \log_2 m$.
- $m \leq c + \log_2 m$ para todo m: falso. ■

Opcional: lo que hace un compresor

terminal

```
$ make compress
cadenas de 1000000 símbolos (0 y 1); tamaños comprimidos en bits
cadena           gzip        bzip2           xz
pattern          8136          584         2272
count         1089400       958288       130304
prng          1199768      1281800      1091104
el programa que las genera: 14136 bits de código fuente
```

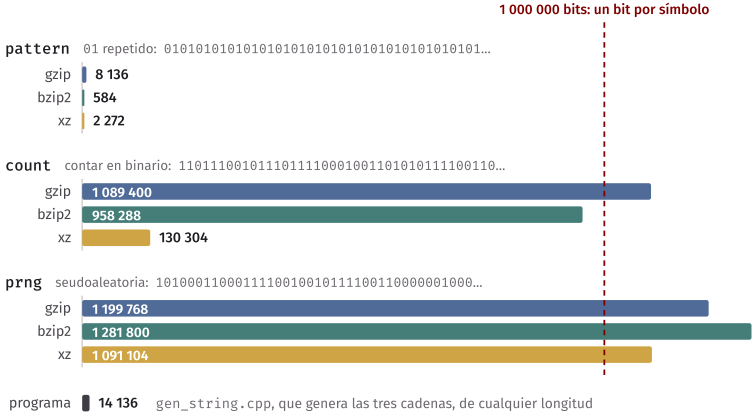
`count`: 1, 10, 11, 100… pegados. `prng`: `std::mt19937_64`, semilla fija.

- `pattern`: todos ven la regularidad.
- `count`: `gzip` entrega *más* de un millón de bits.
- `prng`: incompresible para los tres. Y sale de un programa de 1767 bytes.

Idea clave

El tamaño comprimido es una cota superior de K. Un compresor busca *algunas* regularidades; K, todas.

Opcional: la tabla, contra un bit por símbolo



Lo que pasa de la línea salió peor que no comprimir. Para estos compresores, `prng` es incompresible.

Las tres cadenas son simples: el tamaño comprimido es una cota superior de K, nunca K.

SECCIÓN 13

Cierre

13

Ejercicio opcional: un quine en C++

La sesión es la **Práctica 3**. El ejercicio opcional está en `lab/quine`.

1. Imprimir exactamente el propio fuente, sin abrir archivos y **sin raw string literals**.
2. `make test`: compila, ejecuta y compara con `cmp`.
3. `make test-length`: además imprime cuántos bytes tiene.

terminal

```
$ make test
WA: la salida no es igual a lo esperado
  esperado: 508 bytes; salida: 38 bytes
  primera diferencia: byte 1, línea 1
  para verla: diff build/output.txt build/expected.txt | head
make: *** [Makefile:26: test] Error 1
```

La regla es el espíritu

`make test` rechaza `fstream`, `__FILE__` y similares. La regla es no pedirle el código a nadie.

Resumen

- Un **quine** tiene datos y código. Los datos describen al código; el código usa los datos dos veces.
- **Teorema de recursión**: “obtener la propia descripción” es un paso válido en cualquier máquina.
- Con él: A_{TM} es indecidible en cuatro líneas, y $\mathrm{MIN}_{\mathrm{TM}}$ no es reconocible.
- **Demostrable** no es **verdadero**. Lo demostrable es reconocible; lo verdadero no.
- **Primer teorema de incompletitud**: todo sistema efectivo y correcto que exprese la aritmética deja verdades sin demostrar. Si no, se decidiría la parada.
- Incompleto no es inconsistente. El teorema no habla de la mente ni dice que nada se puede demostrar.
- Opcional: $K(x)$, la descripción más corta. Hay cadenas incompresibles, y K no es computable.

Para la próxima clase

- Leer los apuntes de la semana 11. El ejercicio del quine es opcional.
- Preguntas de repaso de la guía: por qué el intento ingenuo no termina; A_{TM} con el teorema de recursión; por qué lo demostrable es reconocible, y qué pasaría si todo lo verdadero fuera demostrable.
- Aquí termina la parte de computabilidad.
- Próxima semana: de “¿se puede?” a “¿cuánto cuesta?”. Complejidad en tiempo, las clases P y NP.

Pregunta

Encontrar un camino que pase una vez por cada vértice de un grafo parece difícil. Comprobar que un camino dado lo hace es fácil. ¿Qué tienen en común los problemas donde pasa eso?