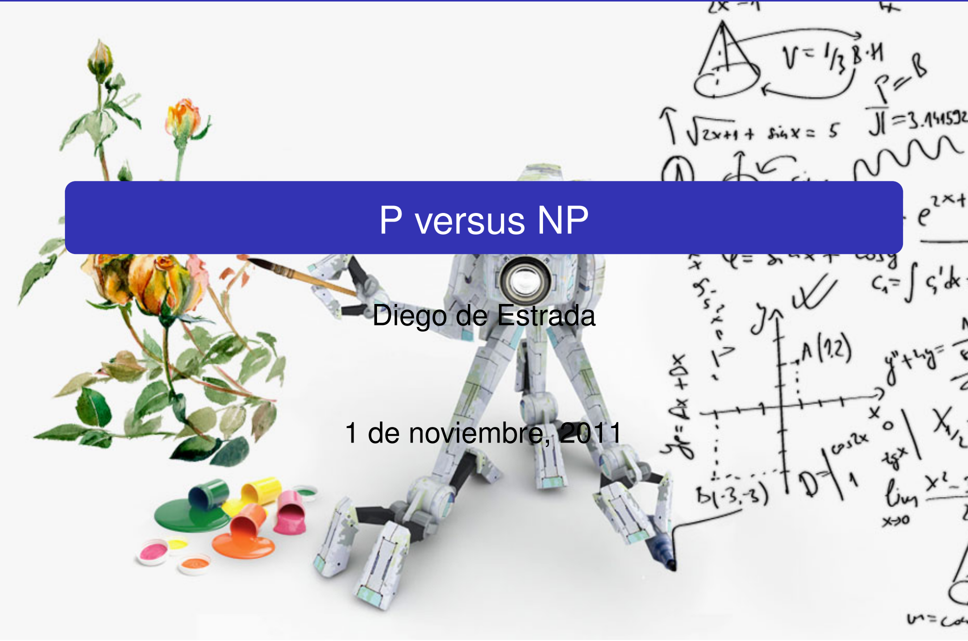
## UbaCs presenta. . .



# P versus NP

Diego de Estrada

1 de noviembre, 2011

# Introducción

Dado un problema cuya solución es fácil de verificar si es correcta, ¿eso implica que el problema es fácil de resolver?

- Es una pregunta filosófica, que acá la llevamos al dominio de la matemática sacrificando mucho en el medio.
- Pero la formulación no es arbitraria: tiene fundamentos empíricos que la hacen relevante en el mundo real.
- Trabajamos con el concepto de *máquina de Turing* (MT) para representar el cómputo, que es equivalente a un programa de computadora.

# Tesis de Church-Turing

*“Las máquinas de Turing pueden simular cualquier modelo computacional físicamente realizable.”*

- Es decir, el conjunto de problemas *computables* no depende del modelo de cómputo.
- Esto no es una proposición formal, sino una creencia sobre la naturaleza del mundo como hoy lo entendemos.
- En 2008 Dershowitz & Gurevich la formalizaron y demostraron. Pero no hay consenso sobre sus axiomas.

# Tesis fuerte de Church-Turing

*"Las máquinas de Turing pueden simular cualquier modelo computacional físicamente realizable con overhead polinomial."*

- Es decir, $t$ pasos en el modelo pueden simularse con $t^c$ pasos con una MT, donde $c$ es una constante que depende del modelo.
- Esto implica que el conjunto de problemas computables en $t^{O(1)}$ pasos no depende del modelo.
- Es controversial: las máquinas cuánticas posiblemente no se pueden simular con MTs con overhead polinomial, pero tampoco estamos seguros si son físicamente realizables.

# Problemas de decisión

- Dada $f : \{0,1\}^* \to \{0,1\}^*$, decimos que una MT $M$ computa $f$ si para todo $x$, la máquina con input $x$ para en finitos pasos y su output es $f(x)$.
- Dada $T : \mathbb{N} \to \mathbb{N}$, decimos que $M$ computa $f$ en tiempo $T(n)$ si el cómputo en cada input $x$ requiere a lo sumo $T(|x|)$ pasos.
- Por conveniencia nos enfocamos en funciones booleanas, o *problemas de decisión*:

$$f : \{0,1\}^* \to \{0,1\}$$

- Estas definen *lenguajes*:

$$L = \{x \in \{0,1\}^* : f(x) = 1\}$$

- Decimos que una MT decide a $L$ si computa $f$.

# Ejemplos de problemas de decisión computables

- Dado un grafo, ver si es conexo.
- Dado un número natural, ver si es primo.
- Dada una fórmula de lógica proposicional, ver si es satisfacible.
- Dados $x, y, i \in \mathbb{N}$, ver si el $i$-ésimo bit de $xy$ es 1.

**No** son computables:

- Dada una MT, ver si para con input 0 (halting problem).
- O en general, verificar cualquier propiedad no trivial de la función parcial que computa una MT (teorema de Rice).
- …(de los $2^{\aleph_0}$ lenguajes hay sólo $\aleph_0$ decidibles)

# Clases de complejidad

- Una *clase de complejidad* es un conjunto de lenguajes decidibles dentro de ciertas cotas de recursos.
- Un lenguaje está en $\mathbf{DTIME}(T(n))$ sii existe una MT que lo decide y corre en tiempo $O(T(n))$.
- Definimos:

$$\mathbf{P} = \cup_{c \geq 1} \mathbf{DTIME}(n^c)$$

**Tesis de Edmonds-Cobham / Cook-Karp**

**P** captura la noción de problemas de decisión computables eficientemente.

- Es controversial, porque $\mathbf{DTIME}(n^{100})$ no parece representar al cómputo eficiente, y más aún, las constantes de problemas en $\mathbf{DTIME}(n)$ pueden ser enormes.
- Pero en general, siempre que probamos que un problema *natural* está en **P**, encontramos tarde o temprano algoritmos eficientes.

# Soluciones verificables eficientemente: la clase **NP**

- Un lenguaje $L$ está en **NP** si existe una MT $M$ de tiempo polinomial y un polinomio $p$ tal que para todo $x$:

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ tal que } M(x,u) = 1$$

  (decimos que $u$ es un certificado)
- Notar que $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{DTIME}(2^{p(n)})$.
- Otra forma de verlo: agregar `fork()` para ir "pidiendo" los bits del certificado. Aceptar si alguna rama tiene output 1. (A estas máquinas se les dice *no determinísticas*.)
- Muchos problemas relevantes en la práctica están en **NP**, y queremos saber si podemos resolverlos eficientemente.

# Aprovechando el azar

- En vez de `fork()`, se puede agregar `flip()`, que devuelve 0 o 1 con probabilidad $1/2$.
- $L$ está en **BPP** si existe una MT probabilística de tiempo polinomial $M$ tal que:

$$x \in L \implies \Pr[M(x) = 1] \geq 2/3$$
$$x \notin L \implies \Pr[M(x) = 0] \geq 2/3$$

- Si repetimos $k$ veces el mismo algoritmo y nos quedamos con el resultado de la mayoría, la probabilidad de errar decrece exponencialmente en $k$ (cota de Chernoff).
- Esto nos dice que los problemas en **BPP** se resuelven eficientemente en la práctica.
- Es una pregunta abierta si $\mathbf{P} = \mathbf{BPP}$. Muchos problemas clásicos en **BPP** ahora se sabe que están en **P**, como Primalidad (Agrawal et al, 2002), pero no todos (por ej. ver si un polinomio multivariado es nulo).

# Oráculos

- Dado un lenguaje $A$, una MT con oráculo $A$ es una MT con una cinta extra en la que escribe cadenas $x$ y en un paso se entera si $x \in A$.
- Definimos $\mathbf{P}^A$ ($\mathbf{NP}^A$) como el conjunto de lenguajes decidibles por una MT de tiempo polinomial (no) determinística con oráculo $A$.
- Extendemos **P** vs **NP** de la siguiente manera: $L \in \Sigma_i^p$ si existe una MT de tiempo polinomial y un polinomio $q$ tal que:

$$x \in L \iff \exists u_1 \forall u_2 \ldots Q_i u_i \text{ tal que } M(x, u_1, \ldots, u_i) = 1,$$

  donde todo $u_j \in \{0,1\}^{q(|x|)}$.
- Notar que $\Sigma_0^p = \mathbf{P}$ y $\Sigma_1^p = \mathbf{NP}$. En general vale $\Sigma_{i+1}^p = \mathbf{NP}^{\Sigma_i^p}$
- Definimos $\mathbf{PH} = \cup_{i \geq 0} \Sigma_i^p$ (la *jerarquía polinomial*). Creemos que $\Sigma_i^p \neq \Sigma_{i+1}^p$ para todo $i$, (y si no fuera así $\mathbf{PH} = \Sigma_i^p$, *colapsa* al $i$-ésimo nivel).

# La primera gran dificultad

**Teorema (Baker-Gill-Solovay, 1975)**

*Existen oráculos $A$ y $B$ tales que* $\mathbf{P}^A = \mathbf{NP}^A$ *y* $\mathbf{P}^B \neq \mathbf{NP}^B$*.*

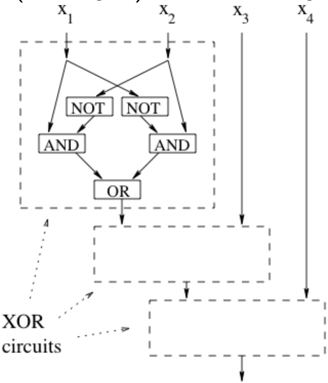
**Dem.** Tomar $A = \{(M, x, 1^n) \ : \ M(x) = 1 \text{ en } 2^n \text{ pasos}\}$. Es fácil ver que $\mathbf{EXP} \subseteq \mathbf{P}^A$ y $\mathbf{NP}^A \subseteq \mathbf{EXP}$.
Sea $U_B = \{1^n \ : \ B \cap \{0,1\}^n \neq \emptyset\}$. Para cualquier $B$, $U_B \in \mathbf{NP}^B$.
Construimos por etapas un $B$ tal que $U_B \notin \mathbf{P}^B$. Sean $M_1, M_2, \ldots$ todas las MT oraculares, en la etapa $i$ vemos qué meter y qué no en $B$ para que $M_i^B$ no decida $U_B$ en tiempo $o(2^n)$.

**Conclusión**

Para resolver **P** vs **NP** necesitamos usar un argumento que no *relativice* (no sea invariante por oráculos), que no use a las MT como *cajas negras* (por ej. no la diagonalización clásica).

Dada una función $f : \{0,1\}^n \to \{0,1\}$, un circuito para $f$ es un diagrama que representa cómo computarla utilizando compuertas OR, AND y NOT.

Por ejemplo, uno para
$f(x_1 x_2 x_3 x_4) = x_1 \oplus x_2 \oplus x_3 \oplus x_4$



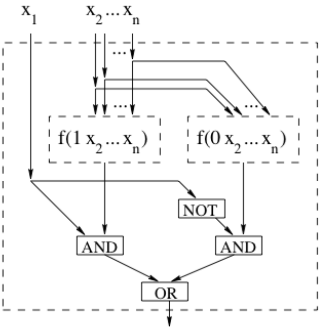

- El tamaño de un circuito lo definimos como la cantidad de compuertas AND y OR.
- Dado un lenguaje $L$ y $T : \mathbb{N} \to \mathbb{N}$, decimos que $L \in$ **SIZE**$(T(n))$ si existe una flia de circuitos $\{C_n\}_{n \in \mathbb{N}}$ donde $C_n$ tiene tamaño $\leq T(n)$, y $\forall x \in \{0,1\}^n$, $x \in L \iff C_n(x) = 1$.
- Se puede ver que **DTIME**$(T(n)) \subseteq$ **SIZE**$(O(T(n) \log T(n)))$.

# Circuitos booleanos

Hay lenguajes que requieren tiempo arbitrariamente grande en MTs, por ej. decisión en aritmética de Presburger está en $\mathbf{DTIME}(2^{2^{\Omega(n)}})$, ¿pero podría pasar esto para circuitos?

Usando que

$f(x_1 x_2 \ldots x_n) = (x_1 \wedge f(1x_2 \ldots x_n)) \vee (\overline{x}_1 \wedge f(0x_2 \ldots x_n))$,




El tamaño del circuito es $s(n) = 3 + 2s(n-1)$ y $s(1) = 1$, lo que da $s(n) = 2^{n+1} - 3$. Con un poco más de trabajo (Lupanov) llegamos a que todo lenguaje está en $\mathbf{SIZE}(2^n/n)$.

La cota es ajustada y casi todos los lenguajes sólo tienen circuitos de tamaño $\Omega(2^n/n)$ (Shannon).

# Circuitos booleanos

- Definimos $\mathbf{P/poly} = \cup_{c \geq 1} \mathbf{SIZE}(n^c)$.
- Otra forma de verlo: $L \in \mathbf{P/poly}$ si existe una MT de tiempo polinomial y una función $F : \mathbb{N} \to \{0,1\}^*$ (*advice*) con $|F(n)| = n^{O(1)}$ tal que $x \in L \iff M(x, F(|x|)) = 1$. (Por esto se dice que es una clase *no uniforme*.)

**Teorema (Adleman, 1978)**

$\mathbf{BPP} \subset \mathbf{P/poly}$

- Notar que no representa una forma de computar útil en la práctica: por ej. contiene a todo lenguaje unario indecidible.
- Pero se usa en criptografía para modelar adversarios que pueden precomputar mucho (por ej. rainbow tables).
- Y es probable que $\mathbf{NP} \not\subseteq \mathbf{P/poly}$ (lo veremos luego), lo cual implicaría $\mathbf{P} \neq \mathbf{NP}$. Las demostraciones usando circuitos tienden a no relativizar.

# Separando clases

Hay problemas abiertos “ridículos”:

- separar $\mathbf{NC}^1$ de **PH**, donde $\mathbf{NC}^1$ es la subclase de $\mathbf{P}/\mathbf{poly}$ cuyos lenguajes tienen circuitos de *profundidad* $O(\log n)$.
- separar **BPP** de $\mathbf{NEXP} = \mathbf{NTIME}(2^{n^{O(1)}})$, notar que $\mathbf{NP} = \mathbf{NTIME}(n^{O(1)})$.
- la mejor cota inferior encontrada para el tamaño del menor circuito booleano de un problema en **NP** es de $5n - o(n)$ (Iwama et al, 2005).

Es por esto que estamos muy lejos de resolver **P** vs **NP**.

# Reducciones entre problemas

¿Cómo probar que un problema es al menos tan difícil como otro? usando reducciones. Hay distintos tipos:

- **Cook**: un lenguaje $A$ es *reducible Cook* a $B$ si $A \in \mathbf{P}^B$.
- **Karp**: un lenguaje $A$ es *reducible Karp* a $B$ si existe $f : \{0,1\}^* \to \{0,1\}^*$ computable en tiempo polinomial tal que $x \in A$ sii $f(x) \in B$ para todo $x$. (Se denota $A \leq_p B$). Son un caso particular de las reducciones Cook (admitiendo una sola pregunta al oráculo).
- ...

Si $A \leq_p B$ y $B \in \mathbf{P}$, entonces $A \in \mathbf{P}$, y por eso $B$ es *al menos tan difícil* como $A$.

Decimos que $B$ es **NP**-completo (**NPC**) si $A \leq_p B$ para todo $A \in \mathbf{NP}$, y $B \in \mathbf{NP}$. **NPC** contiene, entonces, a los problemas más difíciles de **NP**.

# Problemas **NP**-completos

- Sea $L = \{(M, x, 1^n, 1^t) \ : \exists u \in \{0,1\}^n$ tal que $M(x, u) = 1$ en $t$ pasos$\}$. Es fácil ver que $L$ es **NPC**, pero no es útil porque está muy ligado a las MT.
- Sea SAT el conjunto de fórmulas satisfacibles de lógica proposicional. Cook y Levin (1971) demostraron indepte. que SAT es **NPC**. Además, la reducción (de cualquier $L \in$ **NP** a SAT) ¡transforma certificados y mantiene cantidad de soluciones!
- Usando SAT, en 1973 Karp demostró que muchos problemas naturales de combinatoria y teoría de grafos son **NPC**.
- Ahora conocemos miles y de muchas áreas distintas. Si demostramos que uno de todos ellos está en **P** entonces **P** $=$ **NP**.

# Propiedades de **NP**

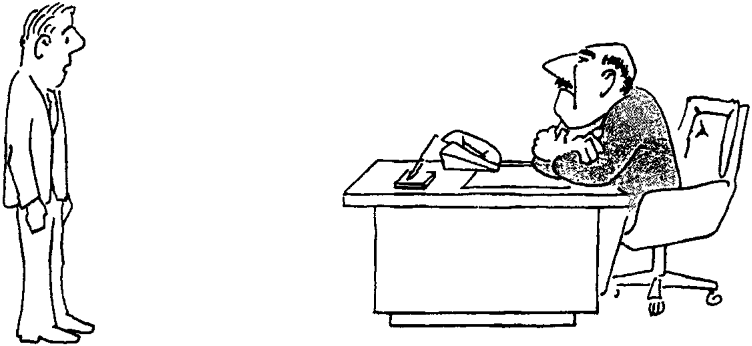
- **Teorema de Mahaney** (1982): sea $L$ esparso (tiene $n^{O(1)}$ cadenas de longitud $n$). Si $L$ es **NPC** entonces **P** $=$ **NP**.
- **Teorema de Ladner** (1975): si **P** $\neq$ **NP**, entonces existen problemas en **NP** que no están en **P** ni **NPC** (los llamamos *intermedios* o **NPI**).
- Se cree que, entre otros, Isomorfismo de Grafos y Factorizar Enteros son **NPI**. De hecho, si Isomorfismo de Grafos es **NPC** entonces **PH** colapsa al segundo nivel (Schöning, 1987).
- **Teorema de Karp-Lipton** (1980): si **NP** $\subseteq$ **P**/**poly** entonces **PH** colapsa al segundo nivel.

# Sobre máquinas cuánticas

- La clase análoga a **P** para máquinas cuánticas es **BQP** (aún no sabemos si son distintas). Se sabe que Factorizar Enteros está en **BQP** (Shor, 1994) como consecuencia de que el Problema del Subgrupo Escondido para grupos abelianos lo está. Si esto también vale para grupos simétricos, Isomorfismo de Grafos está en **BQP**.
- No se conocen problemas **NPC** en **BQP**, las máquinas cuánticas apenas aceleran algunos algoritmos de $2^{O(n)}$ a $2^{O(\sqrt{n})}$ (Grover, 1996).

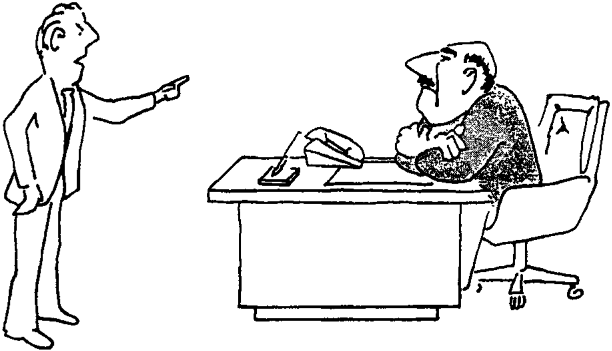
# Utilidad práctica de probar completitud

Nunca queremos que nos pase esto. . .



"I can't find an efficient algorithm, I guess I'm just too dumb."

Lo ideal sería poder decir...



"I can't find an efficient algorithm, because no such algorithm is possible!"

# Utilidad práctica de probar completitud

Probando que el problema es **NP**-completo, ¡es casi tan bueno!

"I can't find an efficient algorithm, but neither can all these famous people."

# Funciones one-way

Una función $f : \{0,1\}^* \to \{0,1\}^*$ computable en tiempo polinomial es one-way si para todo algoritmo probabilístico de tiempo polinomial $A$ (*el adversario*) existe $\epsilon(n) = n^{-\omega(1)}$ tal que $\forall n$,

$$\Pr_{x \in_R \{0,1\}^n}[A(f(x)) = x' \text{ tal que } f(x') = f(x)] < \epsilon(n)$$

(es decir, $f$ es fácil de computar pero difícil de invertir).

## Conjetura

Existen las funciones one-way. (Notar que implica $\mathbf{P} \neq \mathbf{NP}$)

Decimos que $f$ es one-way *fuerte* si resiste adversarios con tiempo $2^{n^{\epsilon}}$ para algún $\epsilon > 0$ fijo.

# Funciones one-way

Creemos que las siguientes funciones son one-way:

- **Multiplicación**: dados $A$ y $B$ de $n/2$-bits, devolver $A \cdot B$. Invertirla es el problema de factorizar enteros.
- **Función de Rabin**: invertirla es el problema de residuos cuadráticos, que es equivalente a factorizar.
- **Función de RSA**: el famoso sistema de clave pública basa su seguridad en ella. Invertirla es tan o más fácil que factorizar (no se sabe si es equivalente).
- **Función universal de Levin**: es one-way si y sólo si existen las funciones one-way.

La existencia de funciones one-way es equivalente a la de generadores pseudoaleatorios: algoritmos polinomiales para generar cadenas que *parezcan* aleatorias para *todo* algoritmo probabilístico de tiempo polinomial (Håstad et al, 1999).

# La segunda gran dificultad

Dados $f : \{0,1\}^n \to \{0,1\}$ y $c \geq 1$, demostrar que $f$ no tiene circuitos de tamaño $n^c$ sería exhibir un predicado $\mathcal{P}$ tal que $\mathcal{P}(f) = 1$ y

$$\mathcal{P}(g) = 0 \text{ para toda } g \text{ que tiene circuitos de tamaño } n^c. \qquad (1)$$

(Un $\mathcal{P}$ que cumple (1) se dice $n^c$-útil).
Decimos que $\mathcal{P}$ es *natural* si satisface:

1. se computa $\mathcal{P}(g)$ en tiempo $2^{O(n)}$ (polinomial en el tamaño de la tabla de verdad de $g$)
2. $\Pr_g[\mathcal{P}(g) = 1] \geq 1/n$

**Teorema (Razborov-Rudich, 1994)**

*Si existen las funciones one-way fuertes, entonces existe un c tal que no hay predicados naturales y $n^c$-útiles.*

# La segunda gran dificultad

**¿Por qué es jodido esto?**

Nos dice que, bajo hipótesis razonables, no se pueden usar demostraciones naturales para probar que $\mathbf{NP} \not\subseteq \mathbf{P}/\mathbf{poly}$.

¿Qué tienen de natural las demostraciones naturales?

1. Casi todas las demostraciones conocidas de cotas inferiores para circuitos usan técnicas de combinatoria que tienden a ser computables en tiempo polinomial.
2. Demostrar que una función $f$ no tiene circuitos de tamaño $S$ implica que al menos la mitad de las $g : \{0,1\}^n \to \{0,1\}$ no tienen circuitos de tamaño $S/2-2$: si tomo una $g$ al azar, luego $f \oplus g$ también es aleatoria, y si ambas tuvieran circuitos de tamaño $S/2-2$ entonces, como $f = (f \oplus g) \oplus g$, $f$ también lo tendría.

# Probabilistically Checkable Proofs

Decimos que $L \in \mathbf{PCP}$ si existe una MT probabilística de tiempo polinomial $V$ tal que:

- dados $x \in \{0,1\}^n$ y una cadena $\pi$ (la *prueba*), $V$ usa $O(\log n)$ bits aleatorios y lee 3 bits de la prueba (llamamos $V^\pi(x)$ al resultado)
- si $x \in L$ existe $\pi$ tal que $\Pr[V^\pi(x) = 1] = 1$
- si $x \notin L$ entonces $\forall \pi$ vale $\Pr[V^\pi(x) = 1] \leq 1/2$

**Teorema (Arora et al 1992, Håstad 1997, Dinur 2005)**

$\mathbf{NP} = \mathbf{PCP}$

**Corolario**: dado un sistema axiomático $\mathcal{A}$ cuyas demos puedan ser verificadas en tiempo polinomial (como PA o ZF),

$$L = \{(\varphi, 1^n) : \varphi \text{ tiene demo de long} \leq n \text{ en } \mathcal{A}\}$$

está en **NP**, y luego toda demo se puede reescribir de manera que con sólo ver $O(1)$ bits, con alta probabilidad sabremos si es correcta.

# Dificultad de aproximar

- Muchos problemas de optimización sabemos que no pueden ser resueltos en tiempo polinomial si $\mathbf{P} \neq \mathbf{NP}$, pero para algunos existen algoritmos polinomiales para aproximar una solución óptima hasta cierto factor constante.
- Una formulación alternativa del **Teorema PCP** es muy útil para demostrar que algunos problemas no pueden ser aproximados eficientemente.
- La **Conjetura de los Juegos Únicos** (Khot, 2002), que postula que cierto problema de grafos es **NPC**, implicaría que los factores de aproximación conocidos para Vertex Cover (2) y Max Cut (0.878), entre otros, son óptimos.

# Los mundos posibles según Impagliazzo

- **Algorithmica**: SAT se resuelve eficientemente en la práctica, ya sea porque **P** $=$ **NP** o algo "moralmente equivalente" como **NP** $\subseteq$ **BPP**.
- **Heuristica**: los problemas **NP** son difíciles en el peor caso pero fáciles en promedio.
- **Pessiland**: los problemas **NP** son difíciles en promedio pero no existen las funciones one-way. Es el peor de todos los mundos, porque además de no poder resolver los problemas tampoco obtenemos ventaja criptográfica de ello.
- **Cryptomania**: existe la criptografía de clave pública (dos desconocidos pueden intercambiar secretos por un canal abierto).
- **Minicrypt**: existen las funciones one-way pero no la criptografía de clave pública.
- **Weirdland**: SAT se resuelve en $n^{100}$ o $n^{\log n}$, o la complejidad varía mucho entre tamaños de input.

# Referencias


- Arora & Barak - Computational Complexity, a modern approach (2009)
- Goldreich - Computational Complexity, a conceptual perspective (2008)
- Blogs de Dick Lipton, Scott Aaronson, Luca Trevisan, Lance Fortnow, etc.
- Complexity Zoo: `qwiki.stanford.edu/index.php/Complexity_Zoo`
- Intentos fallidos: `www.win.tue.nl/~gwoegi/P-versus-NP.htm`
- CSTheory Q&A: `cstheory.stackexchange.com`

# Problemas que reclutan matemáticos

- Caracterizar modelos de computación distribuida con topología algebraica (Herlihy & Shavit).
- Hacer práctico el *cifrado completamente homomórfico* (Craig Gentry).
- Separar clases de complejidad usando geometría algebraica (Mulmuley & Sohoni).
- Buscar una lógica que capture **P**, análoga al teorema de Fagin para **NP** (Martin Grohe).
- Multiplicación rápida de matrices usando teoría de grupos (Cohn & Umans).
- Analizar algoritmos usando análisis complejo (Flajolet).
- Cotas inferiores para algoritmos probabilísticos usando teoría de juegos (Yao).