Complejidad Computacional

Alan Reyes-Figueroa
Teoría de la Computación **(Aula 28) 15.noviembre.2023**

Clases de Complejidad
P y NP
Máquinas de Turing Universales
Turing-*completeness*

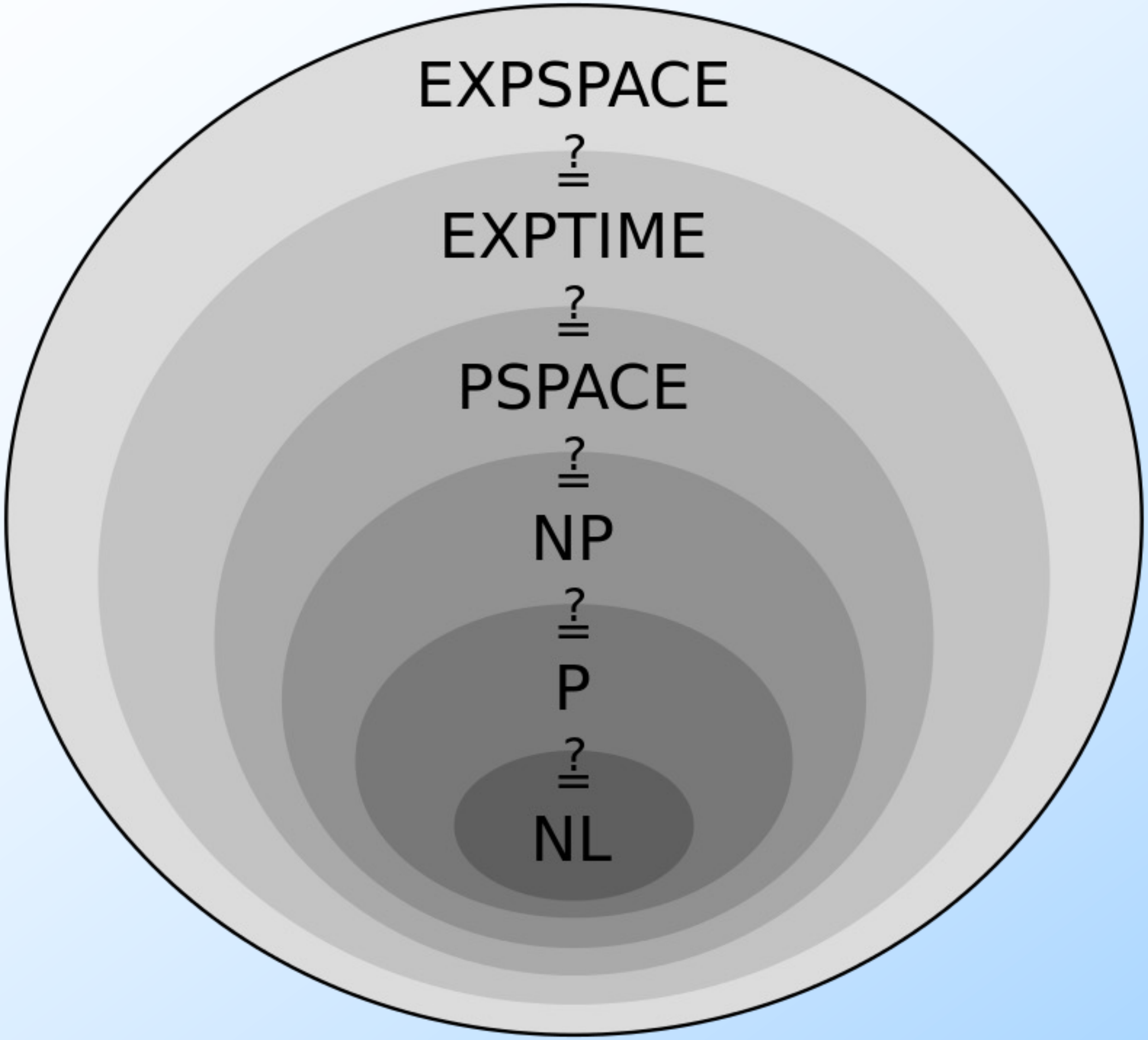
Complejidad de Algoritmos

- Clases:
  En la teoría de la complejidad computacional, es importante tener una medida (o estimación) precisa de qué tan complejo es un problema computacional.
- Esta medida está siempre relacionada a un recurso (tiempo o memoria)
- Nos sirve para comparar también entre dos soluciones (o algoritmos) para un problema.

Ejemplo 1

- 1) un tipo de problema computacional,
  2) un modelo de computación y
  3) un recurso limitado (tiempo o memoria).
- En particular, la mayoría de las clases de complejidad consisten en problemas de decisión que se pueden resolver con una máquina de Turing y se diferencian por sus requisitos de tiempo o espacio (memoria).
- Estas clases se clasifican y jerarquizan.

Complejidad Computacional



La Clase **NL**

- **NL** = *Non-deterministic logarithmic space*, (espacio logarítmico no determinista).
- Es la clase de complejidad que contiene problemas de decisión que pueden resolverse mediante una máquina de Turing no determinista utilizando una cantidad logarítmica de espacio de memoria.
- **NL = NSPACE** O(log n).

- Ejemplos:
- ST-connectivity: Determinar en un grafo, si dos vértices S y T son conectados (T alcanzable desde S)
- 2-satisfiability: 2-SAT

La Clase **P**

- **P =** *(Deterministic) Polinomial Time*. PTIME o DTIME($n^{O(1)}$),
- Es una clase de complejidad fundamental. Contiene todos los problemas de decisión que pueden ser resueltos por una máquina de Turing determinista utilizando una cantidad polinomial de tiempo.
- $T = O(n^k)$

P = Easy to find

La Clase **P**

- Problemas notables en **P**:
  - Búsqueda de un máximo en un arreglo.
  - Ordenamiento de un arreglo.
  - Programación lineal.
  - Determinar si un número entero positivo es primo o no.
  - Saber si un número es primo:
    - Criba de Eratóstenes: O(n log log n)
    - En 2002, se construyó el algoritmo AKS (Agrawal-Kayal-Saxena), y este algoritmo es de tiempo polinomial $O((\log n)^{12})$

La Clase **NP**

- **NP =** *Non-deterministic Polinomial Time*. NDTIME($n^{O(1)}$),
- Es el conjunto de problemas de decisión para los cuales las instancias del problema, donde la respuesta es "sí", tienen pruebas verificables en tiempo polinomial por una máquina de Turing determinista.
- Alternativamente, es el conjunto de problemas que pueden resolverse en tiempo polinomial por una máquina de Turing no determinista.

NP = Easy to check

La Clase **NP**

- Problemas notables que no sabemos si están en **P** (sí están en **NP**):
  - TSP.
  - Determinar si un grafo es Euleriano.
  - Determinar si un grafo es Hamiltoniano.
  - Programación combinatoria:
    - Resolver el problema de las n-reinas
    - Resolver un Sudoku
    - Resolver un problema de optimización.
    - *Knapsack problem*
  - Hallar la factoración en primos de un entero positivo.

Problemas difíciles

- ¿Por qué hay problemas difíciles de resolver? (i.e. tardados) T = O(superpolinomial)
- Problemas de búsqueda (en espacios muy grandes).
- Problemas combinatorios:
  - 3-SAT
  - TSP
  - Ciclos y caminos en grafos
  - Optimización combinatoria

Otras Clases de Complejidad

- EXPTIME
  es el conjunto de todos los problemas de decisión que pueden resolverse mediante una máquina de Turing determinista en tiempo exponencial.
  $T = O(2^{p(n)})$, donde p(n) es un polinomio.

- EXPSPACE es el conjunto de todos los problemas de decisión que se pueden resolver con una máquina de Turing determinista en el espacio exponencial.
  $E = O(2^{p(n)})$.

- $NL \subset P \subset NP \subset PSPACE \subset EXPTIME \subset EXPSPACE$

Conjetura **P = NP**

- **¿es P = NP?**
- El problema P versus NP es un problema importante sin resolver en la informática teórica.
- En términos informales, pregunta si todos los problemas cuya solución puede verificarse rápidamente (NP) también pueden resolverse rápidamente (P).

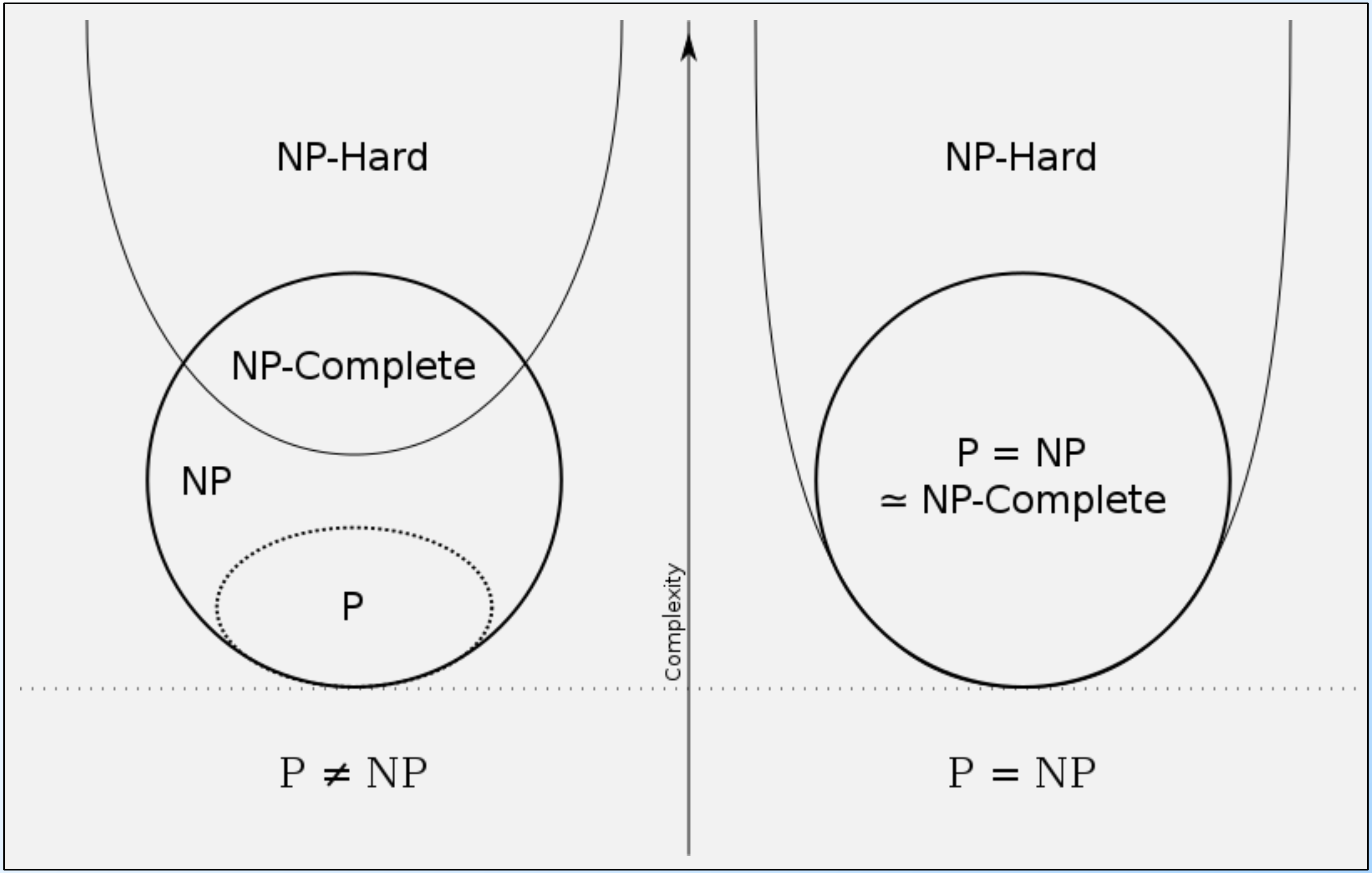
Ejemplo

- S = {-7, -3, -2, 5, 8}.
- Hallar un subconjunto de elementos en S que sume 0.

- <u>Verificación</u> (sí o no)
- A = {-3, -2, 5}, ¿ es la suma(A) = 0?

- <u>Resolver el problema</u> (hallar el subconjunto).
- Podemos revisar, uno por uno, cada subconjunto de A de S y ver si suma(A) = 0.

Conjetura **P = NP**



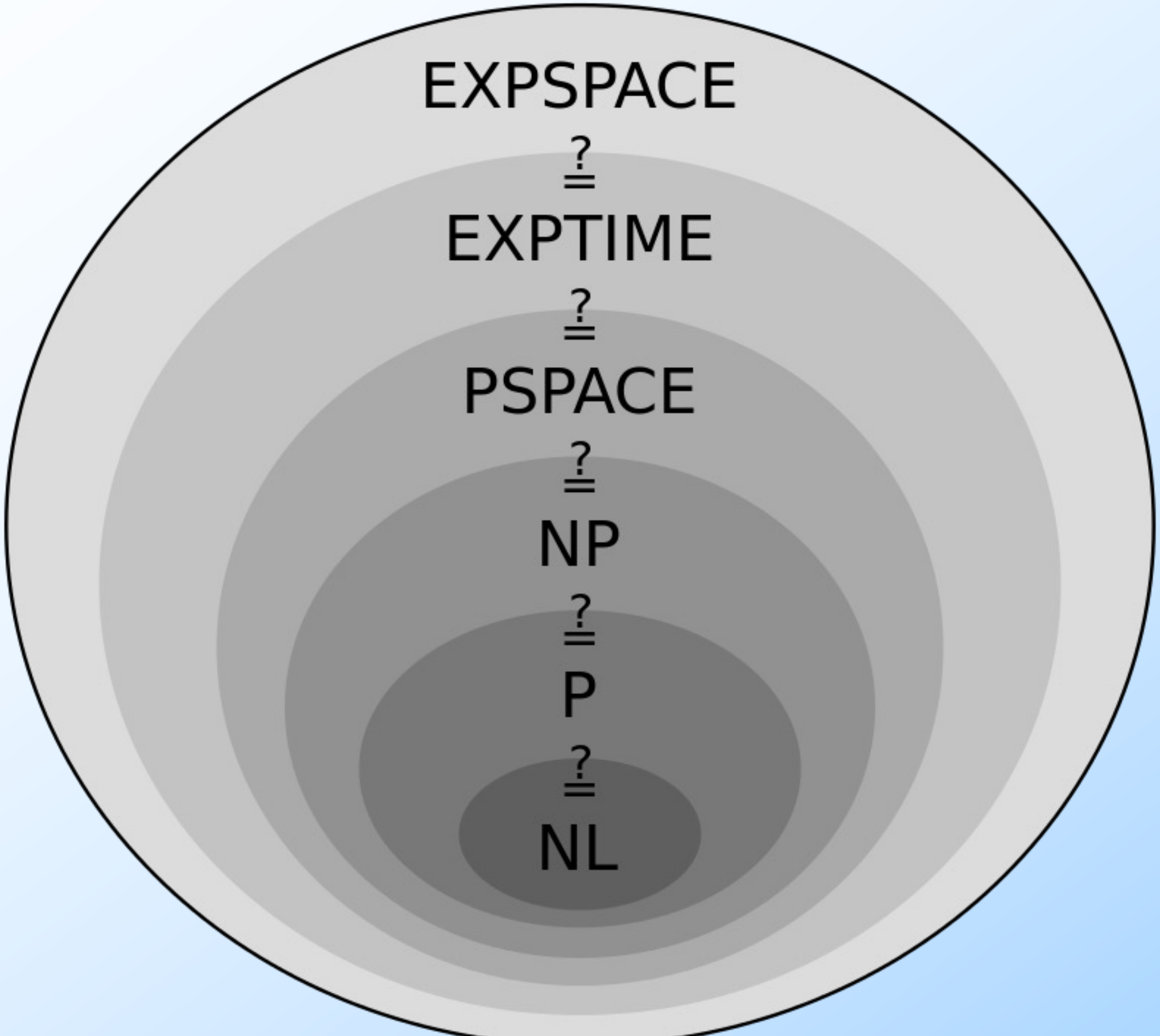
Importancia de ¿**P = NP**?

- <u>Ejemplo</u>: desarrollar una mano robótica.
- P ≠ NP representa problemas que no se pueden resolver. (Aunque sabemos cuál debería ser la solución, en este caso, crear una mano robótica similar a la humana), la solución para crear una mano robótica similar a la humana completamente funcional no se puede cumplir por completo.

Importancia de ¿**P = NP**?

- Si P = NP, podríamos encontrar soluciones a los problemas de búsqueda tan fácilmente como comprobar si esas soluciones son buenas.
- Básicamente, esto resolvería todos los desafíos algorítmicos que enfrentamos hoy y las computadoras podrían resolver casi cualquier tarea.

Complejidad Computacional



Máquina de Turing Universal

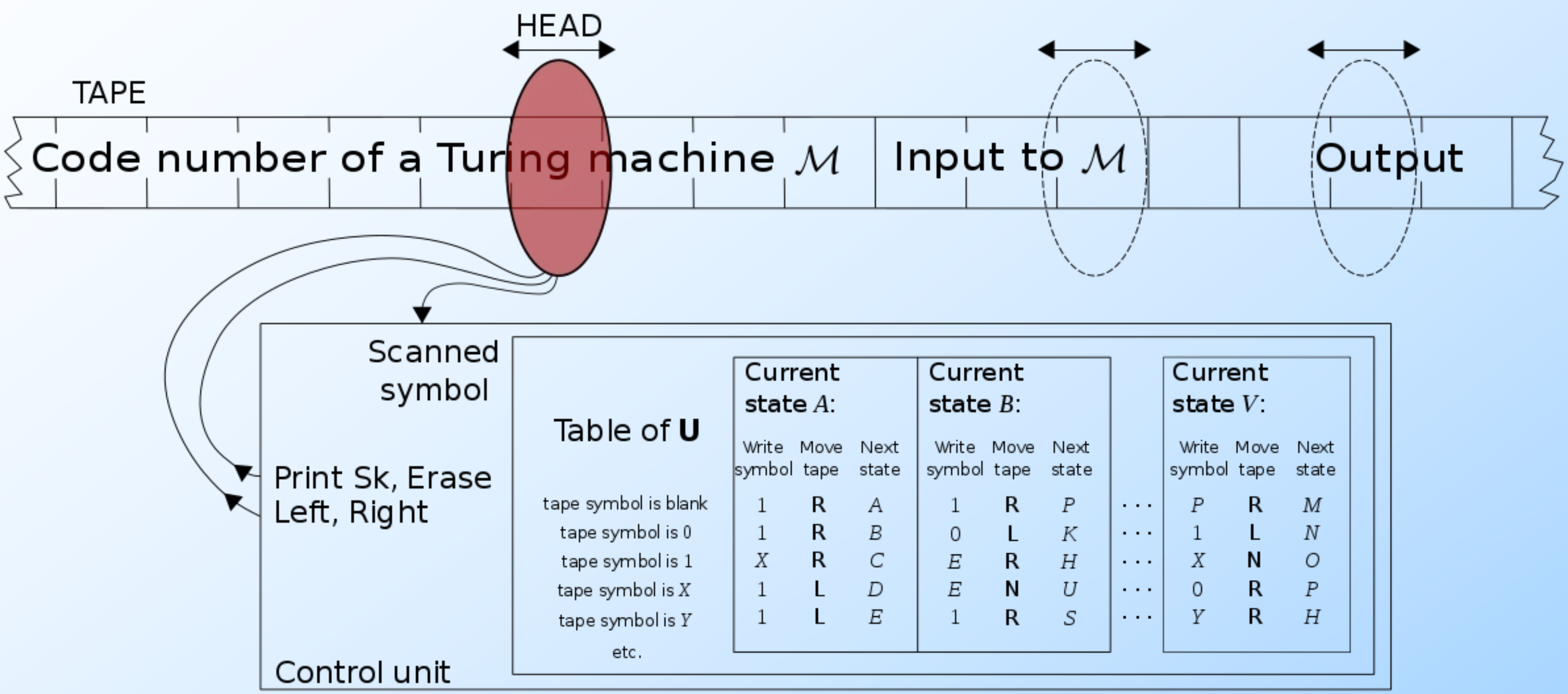
- Una máquina de Turing universal (UTM) es una máquina de Turing que puede simular una máquina de Turing arbitraria con una entrada arbitraria.
- La máquina universal esencialmente logra esto leyendo tanto la descripción de la máquina a simular como la entrada a esa máquina desde su propia cinta.
- Introducidas por Turing (1936-1937).

Máquina de Turing Universal

- Se considera que este principio es el origen de la idea de una computadora con programa almacenado utilizada por John von Neumann en 1946

¿Para qué se usan?

- Para simular cualquier otra máquina de Turing.



Turing-*completeness*

- Hoy por hoy, el mejor modelo computacional que tenemos (el de mayor capacidad) son las máquinas de Turing.

- Las máquinas universales (UTM), en su sentido más amplio, abarcan a cualquier dispositivo que sea capaz de simular cualquier otra máquina de Turing.
- Esto es, capaz de resolver cualquier algoritmo.

Turing-*completeness*

- Decimos que un dispositivo es **Turing-completo** o **Turing-equivalente** si es capaz de simular a cualquier máquina de Turing.
- (Básicamente, si tiene el mismo poder computacional que una UTM).
- Pueden calcular o resolver cualquier función computable.

Turing-*completeness*

- Existen muchos ejemplos de dispositivos Turing-completos:
  - Lenguajes de programación procedimentales de propósito general: C, Pascal
  - Lenguajes de programación orientados a objetos: Java, C#
  - Lenguajes multi-paradigna: Ada, C++, Lisp, Fortran, JavaScript, Perl, Python, R, …
  - La mayoría de lenguajes de programación: Lisp, Haskell, Prolog, m4, TeX, *esoteric languages, …*

Turing-*completeness*

- Otros ejemplos (no intencionados):
  - Excel
  - Power-Point
  - Juegos: Dwarf Fortress, Cities, Opus Magnum, Minecraft.
  - El juego de la vida de Conway.