# Autómatas y Lenguajes Formales
# Facultad de Ciencias UNAM

## Nota 15 sobre lenguajes, máquinas de Turing, *Halting Problem* . . .

Favio E. Miranda Perea  A. Liliana Reyes Cabello
Lourdes del Carmen González Huesca
luglzhuesca@ciencias.unam.mx

23 y 24 noviembre 2016

# Lenguajes recursivos

Un lenguaje $L$ es **recursivo** si es reconocido por una máquina de Turing **total**, es decir, si existe una máquina $M$ que se detiene con todas las cadenas de entrada y $L = L(M)$.

# Lenguajes recursivos

Un lenguaje $L$ es **recursivo** si es reconocido por una máquina de Turing **total**, es decir, si existe una máquina $M$ que se detiene con todas las cadenas de entrada y $L = L(M)$.

- Los lenguajes recursivos también se conocen como Turing-decidibles.
- Existe un algoritmo para decidir si cualquier cadena pertenece a $L$.

# Lenguajes recursivamente enumerables

Un lenguaje $L$ es **recursivamente enumerable**
si es reconocido por una máquina de Turing, es decir,
si existe una máquina de Turing $M$ tal que $L = L(M)$.

# Lenguajes recursivamente enumerables

Un lenguaje $L$ es **recursivamente enumerable**
si es reconocido por una máquina de Turing, es decir,
si existe una máquina de Turing $M$ tal que $L = L(M)$.

- Los lenguajes recursivamente enumerables también se conocen como Turing-reconocibles o semidecidibles.

# Lenguajes recursivamente enumerables

Un lenguaje $L$ es **recursivamente enumerable**
si es reconocido por una máquina de Turing, es decir,
si existe una máquina de Turing $M$ tal que $L = L(M)$.

- Los lenguajes recursivamente enumerables también se conocen como Turing-reconocibles o semidecidibles.
- Existe un procedimiento para decidir si una cadena pertenece a $L$, si no pertenece puede suceder que la máquina se cicle.

# Lenguajes recursivos

## Enumeración

- Si un lenguaje $L$ es recursivo entonces existe un proceso de enumeración para $L$.

# Lenguajes recursivos

## Enumeración

- Si un lenguaje $L$ es recursivo entonces existe un proceso de enumeración para $L$.
- Es decir, existe un proceso que genera **todas** las cadenas de $L$.

# Lenguajes recursivos

## Enumeración

- Si un lenguaje $L$ es recursivo entonces existe un proceso de enumeración para $L$.
- Es decir, existe un proceso que genera **todas** las cadenas de $L$.
- Una máquina que enumera a $L$ se construye componiendo dos máquinas:

# Lenguajes recursivos

## Enumeración

- Si un lenguaje $L$ es recursivo entonces existe un proceso de enumeración para $L$.
- Es decir, existe un proceso que genera **todas** las cadenas de $L$.
- Una máquina que enumera a $L$ se construye componiendo dos máquinas:
  1. una máquina $M$ que genera a todas las cadenas del alfabeto de entrada

# Lenguajes recursivos

## Enumeración

- Si un lenguaje $L$ es recursivo entonces existe un proceso de enumeración para $L$.
- Es decir, existe un proceso que genera **todas** las cadenas de $L$.
- Una máquina que enumera a $L$ se construye componiendo dos máquinas:
  1. una máquina $M$ que genera a todas las cadenas del alfabeto de entrada
  2. una máquina $N$ que reconozca a $L$

# Lenguajes recursivos

## Enumeración

- Si un lenguaje $L$ es recursivo entonces existe un proceso de enumeración para $L$.
- Es decir, existe un proceso que genera **todas** las cadenas de $L$.
- Una máquina que enumera a $L$ se construye componiendo dos máquinas:
  1. una máquina $M$ que genera a todas las cadenas del alfabeto de entrada
  2. una máquina $N$ que reconozca a $L$
- El proceso de enumeración consiste en repetir los siguientes pasos:

# Lenguajes recursivos

## Enumeración

- Si un lenguaje $L$ es recursivo entonces existe un proceso de enumeración para $L$.
- Es decir, existe un proceso que genera **todas** las cadenas de $L$.
- Una máquina que enumera a $L$ se construye componiendo dos máquinas:
  1. una máquina $M$ que genera a todas las cadenas del alfabeto de entrada
  2. una máquina $N$ que reconozca a $L$
- El proceso de enumeración consiste en repetir los siguientes pasos:
  1. $M$ genera una cadena $w$

# Lenguajes recursivos

## Enumeración

- Si un lenguaje $L$ es recursivo entonces existe un proceso de enumeración para $L$.
- Es decir, existe un proceso que genera **todas** las cadenas de $L$.
- Una máquina que enumera a $L$ se construye componiendo dos máquinas:
  1. una máquina $M$ que genera a todas las cadenas del alfabeto de entrada
  2. una máquina $N$ que reconozca a $L$
- El proceso de enumeración consiste en repetir los siguientes pasos:
  1. $M$ genera una cadena $w$
  2. $N$ verifica si $w \in L$, en caso positivo se imprime $w$, en otro caso se ignora a $w$.

# Lenguajes recursivos

## Enumeración

- Si un lenguaje $L$ es recursivo entonces existe un proceso de enumeración para $L$.
- Es decir, existe un proceso que genera **todas** las cadenas de $L$.
- Una máquina que enumera a $L$ se construye componiendo dos máquinas:
  1. una máquina $M$ que genera a todas las cadenas del alfabeto de entrada
  2. una máquina $N$ que reconozca a $L$
- El proceso de enumeración consiste en repetir los siguientes pasos:
  1. $M$ genera una cadena $w$
  2. $N$ verifica si $w \in L$, en caso positivo se imprime $w$, en otro caso se ignora a $w$.
- Este procedimiento funciona pues $N$ siempre se detiene.

# Lenguajes recursivamente enumerables

## Enumeración

Del proceso anterior podemos decir que:

Un lenguaje $L$ es recursivamente enumerable si y sólo si existe un proceso de enumeración para $L$.

# Lenguajes recursivamente enumerables

## Enumeración

Del proceso anterior podemos decir que:

> Un lenguaje $L$ es recursivamente enumerable si y sólo si existe un proceso de enumeración para $L$.

- El proceso de enumeración para un lenguaje recursivo enumerable no funciona de la misma forma. Se debe combinar la ejecución de $M$ y $N$ de otra manera, dado que $N$ puede ciclarse.

# Lenguajes recursivamente enumerables

## Enumeración

Del proceso anterior podemos decir que:

Un lenguaje $L$ es recursivamente enumerable si y sólo si existe un proceso de enumeración para $L$.

- El proceso de enumeración para un lenguaje recursivo enumerable no funciona de la misma forma. Se debe combinar la ejecución de $M$ y $N$ de otra manera, dado que $N$ puede ciclarse.
- El proceso de enumeración consiste en repetir los siguientes pasos:

# Lenguajes recursivamente enumerables

## Enumeración

Del proceso anterior podemos decir que:

Un lenguaje $L$ es recursivamente enumerable si y sólo si existe un proceso de enumeración para $L$.

- El proceso de enumeración para un lenguaje recursivo enumerable no funciona de la misma forma. Se debe combinar la ejecución de $M$ y $N$ de otra manera, dado que $N$ puede ciclarse.
- El proceso de enumeración consiste en repetir los siguientes pasos:
  1. $M$ genera a la $i$-ésima cadena $w_i$

# Lenguajes recursivamente enumerables

## Enumeración

Del proceso anterior podemos decir que:

Un lenguaje $L$ es recursivamente enumerable si y sólo si existe un proceso de enumeración para $L$.

- El proceso de enumeración para un lenguaje recursivo enumerable no funciona de la misma forma. Se debe combinar la ejecución de $M$ y $N$ de otra manera, dado que $N$ puede ciclarse.
- El proceso de enumeración consiste en repetir los siguientes pasos:
  1. $M$ genera a la $i$-ésima cadena $w_i$
  2. $N$ ejecuta un paso (transición) en $w_i$, dos pasos en $w_{i-1}$, tres pasos en $w_{i-2}$, y así sucesivamente.

# Lenguajes recursivamente enumerables

## Enumeración

Del proceso anterior podemos decir que:

Un lenguaje $L$ es recursivamente enumerable si y sólo si existe un proceso de enumeración para $L$.

- El proceso de enumeración para un lenguaje recursivo enumerable no funciona de la misma forma. Se debe combinar la ejecución de $M$ y $N$ de otra manera, dado que $N$ puede ciclarse.
- El proceso de enumeración consiste en repetir los siguientes pasos:
  1. $M$ genera a la $i$-ésima cadena $w_i$
  2. $N$ ejecuta un paso (transición) en $w_i$, dos pasos en $w_{i-1}$, tres pasos en $w_{i-2}$, y así sucesivamente.
  3. Si en algún momento $N$ acepta a $w_i$, se imprime dicha cadena.

# Lenguajes recursivamente enumerables

## Enumeración

Del proceso anterior podemos decir que:

Un lenguaje $L$ es recursivamente enumerable si y sólo si existe un proceso de enumeración para $L$.

- El proceso de enumeración para un lenguaje recursivo enumerable no funciona de la misma forma. Se debe combinar la ejecución de $M$ y $N$ de otra manera, dado que $N$ puede ciclarse.
- El proceso de enumeración consiste en repetir los siguientes pasos:
  1. $M$ genera a la $i$-ésima cadena $w_i$
  2. $N$ ejecuta un paso (transición) en $w_i$, dos pasos en $w_{i-1}$, tres pasos en $w_{i-2}$, y así sucesivamente.
  3. Si en algún momento $N$ acepta a $w_i$, se imprime dicha cadena.
- Este procedimiento funciona pues $N$ va procesando en tiempo finito fragmentos de cada cadena $w_i$

# ¿Qué es un algoritmo?

## Introducción

- ¿Qué es un algoritmo?

# ¿Qué es un algoritmo?

## Introducción

- ¿Qué es un algoritmo?
- Una receta, una serie de pasos a seguir, etc., podemos reconocer cuando vemos un algoritmo.

# ¿Qué es un algoritmo?

## Introducción

- ¿Qué es un algoritmo?
- Una receta, una serie de pasos a seguir, etc., podemos reconocer cuando vemos un algoritmo.
- Pero, ¿podemos dar una definición precisa del concepto de algoritmo?

# ¿Qué es un algoritmo?

## Introducción

- ¿Qué es un algoritmo?
- Una receta, una serie de pasos a seguir, etc., podemos reconocer cuando vemos un algoritmo.
- Pero, ¿podemos dar una definición precisa del concepto de algoritmo?
- ¿Por qué es importante tener una definición precisa (matemática) de algoritmo?

# ¿Qué es un algoritmo?

## Introducción

- Un algoritmo es una colección de instrucciones simples para realizar una tarea o problema particular (procedimientos o recetas).

# ¿Qué es un algoritmo?

## Introducción

- Un algoritmo es una colección de instrucciones simples para realizar una tarea o problema particular (procedimientos o recetas).
- Si se tiene un algoritmo para un problema dado $P$ significa que se tiene una manera para resolver $P$ o calcular efectivamente su resultado.

# ¿Qué es un algoritmo?

## Introducción

- Un algoritmo es una colección de instrucciones simples para realizar una tarea o problema particular (procedimientos o recetas).
- Si se tiene un algoritmo para un problema dado $P$ significa que se tiene una manera para resolver $P$ o calcular efectivamente su resultado.
- Un algoritmo es un proceso **potencialmente** realizable ya que:

# ¿Qué es un algoritmo?

## Introducción

- Un algoritmo es una colección de instrucciones simples para realizar una tarea o problema particular (procedimientos o recetas).
- Si se tiene un algoritmo para un problema dado $P$ significa que se tiene una manera para resolver $P$ o calcular efectivamente su resultado.
- Un algoritmo es un proceso **potencialmente** realizable ya que:
  1. las operaciones del proceso se pueden realizar inequívocamente y

# ¿Qué es un algoritmo?

## Introducción

- Un algoritmo es una colección de instrucciones simples para realizar una tarea o problema particular (procedimientos o recetas).
- Si se tiene un algoritmo para un problema dado $P$ significa que se tiene una manera para resolver $P$ o calcular efectivamente su resultado.
- Un algoritmo es un proceso **potencialmente** realizable ya que:
  1. las operaciones del proceso se pueden realizar inequívocamente y
  2. el número de operaciones o pasos del proceso es finito.

# Existencia de algoritmos

## Introducción

- Décimo problema de Hilbert:

  Hallar un proceso de acuerdo al cual pueda determinarse en un número finito de pasos (un algoritmo) si un polinomio dado tiene una raiz entera.

# Existencia de algoritmos

## Introducción

- Décimo problema de Hilbert:

  Hallar un proceso de acuerdo al cual pueda determinarse en un número finito de pasos (un algoritmo) si un polinomio dado tiene una raiz entera.

- Se creía que todo problema $P$ tenia una solución algorítmica.

# Existencia de algoritmos

## Introducción

- Décimo problema de Hilbert:

  Hallar un proceso de acuerdo al cual pueda determinarse en un número finito de pasos (un algoritmo) si un polinomio dado tiene una raiz entera.
- Se creía que todo problema $P$ tenia una solución algorítmica.
- Más aún se pensaba en la existencia de un algoritmo universal $U$ que pudiera resolver todos los problemas matemáticos.

# Existencia de algoritmos

## Introducción

- Los intentos por hallar el algoritmo universal $U$ fallaron.

# Existencia de algoritmos

## Introducción

- Los intentos por hallar el algoritmo universal $U$ fallaron.
- Tal vez $U$ no existía.

# Existencia de algoritmos

## Introducción

- Los intentos por hallar el algoritmo universal $U$ fallaron.
- Tal vez $U$ no existía.
- ¿Cómo probar la no existencia de $U$?

# Existencia de algoritmos

## Introducción

- Los intentos por hallar el algoritmo universal $U$ fallaron.
- Tal vez $U$ no existía.
- ¿Cómo probar la no existencia de $U$?
- Era necesario definir el concepto de algoritmo de una manera precisa y hallar un formalismo para poder probar propiedades de los mismos.

# Existencia de algoritmos

## Introducción

- Los intentos por hallar el algoritmo universal $U$ fallaron.
- Tal vez $U$ no existía.
- ¿Cómo probar la no existencia de $U$?
- Era necesario definir el concepto de algoritmo de una manera precisa y hallar un formalismo para poder probar propiedades de los mismos.
- Un formalismo de algoritmos debería ser preciso y libre de ambigüedades, simple y general.

# Formalización del concepto de algoritmo

## Diferentes modelos

- Máquinas de Turing (Alan Turing, Cambridge 1936)

# Formalización del concepto de algoritmo

## Diferentes modelos

- Máquinas de Turing (Alan Turing, Cambridge 1936)
- Cálculo Lambda (Alonzo Church, Princeton 1936)

# Formalización del concepto de algoritmo

## Diferentes modelos

- Máquinas de Turing (Alan Turing, Cambridge 1936)
- Cálculo Lambda (Alonzo Church, Princeton 1936)
- Sistemas de Post (Emile Post)

# Formalización del concepto de algoritmo

## Diferentes modelos

- Máquinas de Turing (Alan Turing, Cambridge 1936)
- Cálculo Lambda (Alonzo Church, Princeton 1936)
- Sistemas de Post (Emile Post)
- Funciones $\mu$-recursivas (Gödel, Herbrand, Kleene)

# Formalización del concepto de algoritmo

## Diferentes modelos

- Máquinas de Turing (Alan Turing, Cambridge 1936)
- Cálculo Lambda (Alonzo Church, Princeton 1936)
- Sistemas de Post (Emile Post)
- Funciones $\mu$-recursivas (Gödel, Herbrand, Kleene)
- Lógica Combinatoria (Curry, Schönfinkel)

# Formalización del concepto de algoritmo

## Diferentes modelos

- Máquinas de Turing (Alan Turing, Cambridge 1936)
- Cálculo Lambda (Alonzo Church, Princeton 1936)
- Sistemas de Post (Emile Post)
- Funciones $\mu$-recursivas (Gödel, Herbrand, Kleene)
- Lógica Combinatoria (Curry, Schönfinkel)
- Máquinas de registro (Sheperdson, Sturgis)

# Equivalencia de los formalismos

- Sorprendentemente todos los formalismos han resultado equivalentes.

# Equivalencia de los formalismos

- Sorprendentemente todos los formalismos han resultado equivalentes.
- Un problema tiene solución en un formalismo si y sólo si tiene solución en cualquiera de los otros.

# Equivalencia de los formalismos

- Sorprendentemente todos los formalismos han resultado equivalentes.
- Un problema tiene solución en un formalismo si y sólo si tiene solución en cualquiera de los otros.
- Tal afirmación es un teorema, o una serie de teoremas rigurosamente demostrados.

# Equivalencia de los formalismos

- Sorprendentemente todos los formalismos han resultado equivalentes.
- Un problema tiene solución en un formalismo si y sólo si tiene solución en cualquiera de los otros.
- Tal afirmación es un teorema, o una serie de teoremas rigurosamente demostrados.
- Esta coincidencia nos lleva a conjeturar que existe una única noción de computabilidad.

# La Tesis de Church-Turing

Un problema es soluble algorítmicamente si y sólo si es soluble mediante una máquina de Turing.

# La Tesis de Church-Turing

Un problema es soluble algorítmicamente si y sólo si
es soluble mediante una máquina de Turing.

- La tesis de Church-Turing afirma que la noción intuitiva de algoritmo es capturada de manera exacta por la noción matemática de máquina de Turing.

# La Tesis de Church-Turing

Un problema es soluble algorítmicamente si y sólo si
es soluble mediante una máquina de Turing.

- La tesis de Church-Turing afirma que la noción intuitiva de algoritmo es capturada de manera exacta por la noción matemática de máquina de Turing.
- Es decir, las máquinas de Turing implementan a cualquier algoritmo.

# La Tesis de Church-Turing

Un problema es soluble algorítmicamente si y sólo si
es soluble mediante una máquina de Turing.

- La tesis de Church-Turing afirma que la noción intuitiva de algoritmo es capturada de manera exacta por la noción matemática de máquina de Turing.
- Es decir, las máquinas de Turing implementan a cualquier algoritmo.
- Equivalentemente, una función es computable si y sólo si es soluble mediante una máquina de Turing.

# La Tesis de Church-Turing

Un problema es soluble algorítmicamente si y sólo si
es soluble mediante una máquina de Turing.

- La tesis de Church-Turing afirma que la noción intuitiva de algoritmo es capturada de manera exacta por la noción matemática de máquina de Turing.
- Es decir, las máquinas de Turing implementan a cualquier algoritmo.
- Equivalentemente, una función es computable si y sólo si es soluble mediante una máquina de Turing.
- También se usa el término **Turing-completo** para referirse a un sistema que puede usarse para simular a una máquina de Turing.

# La Tesis de Church-Turing

Un problema es soluble algorítmicamente si y sólo si es soluble mediante una máquina de Turing.

# La Tesis de Church-Turing

Un problema es soluble algorítmicamente si y sólo si
es soluble mediante una máquina de Turing.

- La afirmación es una tesis indemostrable pues la noción de algoritmo es intuitiva.

# La Tesis de Church-Turing

Un problema es soluble algorítmicamente si y sólo si
es soluble mediante una máquina de Turing.

- La afirmación es una tesis indemostrable pues la noción de algoritmo es intuitiva.
- Por otro lado la tesis es refutable y se destruirá mostrando un algoritmo que no pudiera ser implementado en una máquina de Turing.

# Tesis de Church-Turing

## Evidencias a favor

- Existen fuertes evidencias a favor de la tesis.

# Tesis de Church-Turing

## Evidencias a favor

- Existen fuertes evidencias a favor de la tesis.
- Intuitivamente cualquier algoritmo detallado para el cálculo manual puede programarse en una Máquina de Turing.

# Tesis de Church-Turing

## Evidencias a favor

- Existen fuertes evidencias a favor de la tesis.
- Intuitivamente cualquier algoritmo detallado para el cálculo manual puede programarse en una Máquina de Turing.
- La equivalencia con otros formalismos más modernos.

# Tesis de Church-Turing

## Evidencias a favor

- Existen fuertes evidencias a favor de la tesis.
- Intuitivamente cualquier algoritmo detallado para el cálculo manual puede programarse en una Máquina de Turing.
- La equivalencia con otros formalismos más modernos.
- Existen demasiados ejemplos a favor y por supuesto ningún contraejemplo.

# Tesis de Church-Turing

## Evidencias a favor

- Existen fuertes evidencias a favor de la tesis.
- Intuitivamente cualquier algoritmo detallado para el cálculo manual puede programarse en una Máquina de Turing.
- La equivalencia con otros formalismos más modernos.
- Existen demasiados ejemplos a favor y por supuesto ningún contraejemplo.
- La comunidad tanto en matemáticas como en ciencias de la computación acepta ampiamente la tesis.

# Máquinas de Turing vs. Computadoras

- Una computadora es capaz de interpretar algoritmos arbitrarios y obtener la misma respuesta que cada algoritmo particular.

# Máquinas de Turing vs. Computadoras

- Una computadora es capaz de interpretar algoritmos arbitrarios y obtener la misma respuesta que cada algoritmo particular.
- Entonces una computadora es una máquina útil para propósitos generales.

# Máquinas de Turing vs. Computadoras

- Una computadora es capaz de interpretar algoritmos arbitrarios y obtener la misma respuesta que cada algoritmo particular.
- Entonces una computadora es una máquina útil para propósitos generales.
- Las Máquina de Turing en cambio son diseñadas para propósitos particulares.

# Máquinas de Turing vs. Computadoras

- Una computadora es capaz de interpretar algoritmos arbitrarios y obtener la misma respuesta que cada algoritmo particular.
- Entonces una computadora es una máquina útil para propósitos generales.
- Las Máquina de Turing en cambio son diseñadas para propósitos particulares.
- Conclusión: el poder computacional de las Máquina de Turing no puede ser equiparable al de las computadoras actuales.

# Máquinas de Turing vs. Computadoras

- Una computadora es capaz de interpretar algoritmos arbitrarios y obtener la misma respuesta que cada algoritmo particular.
- Entonces una computadora es una máquina útil para propósitos generales.
- Las Máquina de Turing en cambio son diseñadas para propósitos particulares.
- Conclusión: el poder computacional de las Máquina de Turing no puede ser equiparable al de las computadoras actuales.
- Las computadoras son programables, las Máquina de Turing no.

# La Máquina Universal de Turing

- ¿Sería factible pensar en la existencia de una Máquina de Turing que se comporte de la misma forma que una computadora real?

# La Máquina Universal de Turing

- ¿Sería factible pensar en la existencia de una Máquina de Turing que se comporte de la misma forma que una computadora real?
- Es decir, una máquina que sea útil para propósitos múltiples.

# La Máquina Universal de Turing

- ¿Sería factible pensar en la existencia de una Máquina de Turing que se comporte de la misma forma que una computadora real?
- Es decir, una máquina que sea útil para propósitos múltiples.
- Dicha máquina sería capaz de programar y ejecutar máquinas de Turing.

# La Máquina Universal de Turing

- ¿Sería factible pensar en la existencia de una Máquina de Turing que se comporte de la misma forma que una computadora real?
- Es decir, una máquina que sea útil para propósitos múltiples.
- Dicha máquina sería capaz de programar y ejecutar máquinas de Turing.
- Tal máquina existe y se conoce como máquina universal de Turing $\mathcal{M}$.

# La Máquina Universal de Turing

- ¿Sería factible pensar en la existencia de una Máquina de Turing que se comporte de la misma forma que una computadora real?
- Es decir, una máquina que sea útil para propósitos múltiples.
- Dicha máquina sería capaz de programar y ejecutar máquinas de Turing.
- Tal máquina existe y se conoce como máquina universal de Turing $\mathcal{M}$.
- Esta máquina recibe como entrada una descripción de una Máquina de Turing $M$ y una cadena $w$ y simula el comportamiento de $M$ sobre $w$.

# La Máquina Universal de Turing

- ¿Sería factible pensar en la existencia de una Máquina de Turing que se comporte de la misma forma que una computadora real?
- Es decir, una máquina que sea útil para propósitos múltiples.
- Dicha máquina sería capaz de programar y ejecutar máquinas de Turing.
- Tal máquina existe y se conoce como máquina universal de Turing $\mathcal{M}$.
- Esta máquina recibe como entrada una descripción de una Máquina de Turing $M$ y una cadena $w$ y simula el comportamiento de $M$ sobre $w$.
- Los datos de entrada $M$ y $w$ deben ser codificados de manera adecuada.

# Codificación de Máquina de Turing

## Observaciones

- La codificación de una Máquina de Turing no es única, puesto que el orden de las transiciones no importa y un orden distinto genera una codificación distinta.

# Codificación de Máquina de Turing

## Observaciones

- La codificación de una Máquina de Turing no es única, puesto que el orden de las transiciones no importa y un orden distinto genera una codificación distinta.
- De hecho si $M$ tiene $n$ transiciones, existen $n!$ codificaciones distintas para $M$.

# Codificación de Máquina de Turing

## Observaciones

- La codificación de una Máquina de Turing no es única, puesto que el orden de las transiciones no importa y un orden distinto genera una codificación distinta.
- De hecho si $M$ tiene $n$ transiciones, existen $n!$ codificaciones distintas para $M$.
- El proceso de codificación puede revertirse, no es díficil definir un algoritmo que decida si una secuencia binaria representa un código válido para Máquina de Turing y en tal caso lo decodifique.

# Codificación de Máquina de Turing

## Observaciones

- La codificación de una Máquina de Turing no es única, puesto que el orden de las transiciones no importa y un orden distinto genera una codificación distinta.
- De hecho si $M$ tiene $n$ transiciones, existen $n!$ codificaciones distintas para $M$.
- El proceso de codificación puede revertirse, no es díficil definir un algoritmo que decida si una secuencia binaria representa un código válido para Máquina de Turing y en tal caso lo decodifique.

N.B. La codificación y el uso de esta máquina se estudiaron en la nota anterior.

# Enumerabilidad de las Máquina de Turing

- En conclusión toda Máquina de Turing puede representarse como una cadena binaria.

# Enumerabilidad de las Máquina de Turing

- En conclusión toda Máquina de Turing puede representarse como una cadena binaria.
- No todas las cadenas binarias representan Máquina de Turing válidas, por ejemplo, las cadenas que empiezan o terminan con 1 o las que tienen más de dos ceros consecutivos.

# Enumerabilidad de las Máquina de Turing

- En conclusión toda Máquina de Turing puede representarse como una cadena binaria.
- No todas las cadenas binarias representan Máquina de Turing válidas, por ejemplo, las cadenas que empiezan o terminan con 1 o las que tienen más de dos ceros consecutivos.
- Cada cadena binaria, representa por otra parte un número natural y viceversa. Es decir, hay tantas cadenas binaria como números naturales.

# Enumerabilidad de las Máquina de Turing

- En conclusión toda Máquina de Turing puede representarse como una cadena binaria.
- No todas las cadenas binarias representan Máquina de Turing válidas, por ejemplo, las cadenas que empiezan o terminan con 1 o las que tienen más de dos ceros consecutivos.
- Cada cadena binaria, representa por otra parte un número natural y viceversa. Es decir, hay tantas cadenas binaria como números naturales.
- De lo anterior se concluye que hay sólo un número numerable de Máquina de Turing.

# Enumerabilidad de las Máquina de Turing

- Las cadenas binarias pueden enumerarse en orden lexicográfico considerando $0 < 1$:

$$0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \ldots$$

# Enumerabilidad de las Máquina de Turing

- Las cadenas binarias pueden enumerarse en orden lexicográfico considerando $0 < 1$:

$$0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \ldots$$

- Cada Máquina de Turing figura varias veces en esta lista.

# Enumerabilidad de las Máquina de Turing

- Las cadenas binarias pueden enumerarse en orden lexicográfico considerando $0 < 1$:

$$0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \ldots$$

- Cada Máquina de Turing figura varias veces en esta lista.
- Por motivos prácticos si una cadena no codifica directamente a una Máquina de Turing entonces acordamos que codifica a la máquina sin transiciones que acepta el lenguaje vacío.

# Enumerabilidad de las Máquina de Turing

- Las cadenas binarias pueden enumerarse en orden lexicográfico considerando $0 < 1$:

$$0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \ldots$$

- Cada Máquina de Turing figura varias veces en esta lista.
- Por motivos prácticos si una cadena no codifica directamente a una Máquina de Turing entonces acordamos que codifica a la máquina sin transiciones que acepta el lenguaje vacío.
- De esta manera cada cadena binaria codifica a una Máquina de Turing.

# Existencia de funciones no computables

## Enumerabilidad de Máquinas de Turing

- Dado que las cadenas binarias son tantas como los números naturales y cada cadena codifica a una Máquina de Turing concluimos que sólo hay un número infinito numerable de Máquinas de Turing.

# Existencia de funciones no computables

## Enumerabilidad de Máquinas de Turing

- Dado que las cadenas binarias son tantas como los números naturales y cada cadena codifica a una Máquina de Turing concluimos que sólo hay un número infinito numerable de Máquinas de Turing.
- Por otro lado si consideramos las funciones $f : \mathbb{N} \to \mathbb{N}$ es bien sabido que son un número **no** numerable (tantas como números reales).

# Existencia de funciones no computables

## Enumerabilidad de Máquinas de Turing

- Dado que las cadenas binarias son tantas como los números naturales y cada cadena codifica a una Máquina de Turing concluimos que sólo hay un número infinito numerable de Máquinas de Turing.
- Por otro lado si consideramos las funciones $f : \mathbb{N} \to \mathbb{N}$ es bien sabido que son un número **no** numerable (tantas como números reales).
- De donde se concluye que existen funciones, que **no** pueden calcularse mediante una Máquina de Turing.

# Existencia de funciones no computables

## Enumerabilidad de Máquinas de Turing

- Dado que las cadenas binarias son tantas como los números naturales y cada cadena codifica a una Máquina de Turing concluimos que sólo hay un número infinito numerable de Máquinas de Turing.
- Por otro lado si consideramos las funciones $f : \mathbb{N} \to \mathbb{N}$ es bien sabido que son un número **no** numerable (tantas como números reales).
- De donde se concluye que existen funciones, que **no** pueden calcularse mediante una Máquina de Turing.
- Lo cual bajo la tesis de Church-Turing equivale a que existen funciones que no pueden ser calculadas mediante una computadora.

# El lenguaje diagonal $\mathcal{L}_D$

- Consideremos la enumeración de las máquinas de Turing $M_1, M_2, \ldots$ así como la enumeración de todas las cadenas de $\Sigma^\star$, digamos $w_1, w_2, \ldots, w_n, \ldots$

# El lenguaje diagonal $\mathcal{L}_D$

- Consideremos la enumeración de las máquinas de Turing $M_1, M_2, \ldots$ así como la enumeración de todas las cadenas de $\Sigma^\star$, digamos $w_1, w_2, \ldots, w_n, \ldots$
- Podemos entonces dar como entrada la $i$-ésima palabra $w_i$ a la $i$-ésima máquina $M_i$.

# El lenguaje diagonal $\mathcal{L}_D$

- Consideremos la enumeración de las máquinas de Turing $M_1, M_2, \ldots$ así como la enumeración de todas las cadenas de $\Sigma^\star$, digamos $w_1, w_2, \ldots, w_n, \ldots$
- Podemos entonces dar como entrada la $i$-ésima palabra $w_i$ a la $i$-ésima máquina $M_i$.
- Definimos el lenguaje diagonal se define como:

$$\mathcal{L}_D = \{w_i \mid w_i \text{ no es aceptada por } M_i\}$$

# El lenguaje diagonal $\mathcal{L}_D$

- Consideremos la enumeración de las máquinas de Turing $M_1, M_2, \ldots$ así como la enumeración de todas las cadenas de $\Sigma^\star$, digamos $w_1, w_2, \ldots, w_n, \ldots$
- Podemos entonces dar como entrada la $i$-ésima palabra $w_i$ a la $i$-ésima máquina $M_i$.
- Definimos el lenguaje diagonal se define como:

$$\mathcal{L}_D = \{w_i \mid w_i \text{ no es aceptada por } M_i\}$$

- Es decir $\mathcal{L}_D$ contiene a la $i$-ésima cadena si y sólo si ésta **no** es aceptada por la $i$-ésima máquina.

# $\mathcal{L}_D$ **no** es recursivamente enumerable

- Si $\mathcal{L}_D$ fuera recursivamente enumerable sería aceptado por una Máquina de Turing, digamos la $k$-ésima máquina $M_k$.

# $\mathcal{L}_D$ **no** es recursivamente enumerable

- Si $\mathcal{L}_D$ fuera recursivamente enumerable sería aceptado por una Máquina de Turing, digamos la $k$-ésima máquina $M_k$.
- En tal caso $\mathcal{L}_D = L(M_k)$.

# $\mathcal{L}_D$ **no** es recursivamente enumerable

- Si $\mathcal{L}_D$ fuera recursivamente enumerable sería aceptado por una Máquina de Turing, digamos la $k$-ésima máquina $M_k$.
- En tal caso $\mathcal{L}_D = L(M_k)$.
- Podemos preguntarnos entonces si $w_k \in \mathcal{L}_D$.

# $\mathcal{L}_D$ **no** es recursivamente enumerable

- Si $\mathcal{L}_D$ fuera recursivamente enumerable sería aceptado por una Máquina de Turing, digamos la $k$-ésima máquina $M_k$.
- En tal caso $\mathcal{L}_D = L(M_k)$.
- Podemos preguntarnos entonces si $w_k \in \mathcal{L}_D$.
  - $w_k \in \mathcal{L}_D \Rightarrow w_k$ no es aceptada por $M_k \Rightarrow w_k \notin L(M_k) = \mathcal{L}_D$.

# $\mathcal{L}_D$ **no** es recursivamente enumerable

- Si $\mathcal{L}_D$ fuera recursivamente enumerable sería aceptado por una Máquina de Turing, digamos la $k$-ésima máquina $M_k$.
- En tal caso $\mathcal{L}_D = L(M_k)$.
- Podemos preguntarnos entonces si $w_k \in \mathcal{L}_D$.
  - $w_k \in \mathcal{L}_D \Rightarrow w_k$ no es aceptada por $M_k \Rightarrow w_k \notin L(M_k) = \mathcal{L}_D$.
  - $w_k \notin \mathcal{L}_D \Rightarrow w_k$ es aceptada por $M_k \Rightarrow w_k \in L(M_k) = \mathcal{L}_D$.

# $\mathcal{L}_D$ **no** es recursivamente enumerable

- Si $\mathcal{L}_D$ fuera recursivamente enumerable sería aceptado por una Máquina de Turing, digamos la $k$-ésima máquina $M_k$.
- En tal caso $\mathcal{L}_D = L(M_k)$.
- Podemos preguntarnos entonces si $w_k \in \mathcal{L}_D$.
  - $w_k \in \mathcal{L}_D \Rightarrow w_k$ no es aceptada por $M_k \Rightarrow w_k \notin L(M_k) = \mathcal{L}_D$.
  - $w_k \notin \mathcal{L}_D \Rightarrow w_k$ es aceptada por $M_k \Rightarrow w_k \in L(M_k) = \mathcal{L}_D$.
- Por lo que se tendría

$$w_k \in \mathcal{L}_D \text{ si y sólo si } w_k \notin \mathcal{L}_D$$

# $\mathcal{L}_D$ **no** es recursivamente enumerable

- Si $\mathcal{L}_D$ fuera recursivamente enumerable sería aceptado por una Máquina de Turing, digamos la $k$-ésima máquina $M_k$.
- En tal caso $\mathcal{L}_D = L(M_k)$.
- Podemos preguntarnos entonces si $w_k \in \mathcal{L}_D$.
  - $w_k \in \mathcal{L}_D \Rightarrow w_k$ no es aceptada por $M_k \Rightarrow w_k \notin L(M_k) = \mathcal{L}_D$.
  - $w_k \notin \mathcal{L}_D \Rightarrow w_k$ es aceptada por $M_k \Rightarrow w_k \in L(M_k) = \mathcal{L}_D$.
- Por lo que se tendría

$$w_k \in \mathcal{L}_D \text{ si y sólo si } \; w_k \notin \mathcal{L}_D$$

- Lo cual es absurdo.

# El lenguaje universal $\mathcal{L}_{\mathcal{M}}$

## Definición

- El lenguaje aceptado por la máquina universal $\mathcal{M}$ se conoce como lenguaje universal, denotado $\mathcal{L}_{\mathcal{M}}$.

# El lenguaje universal $\mathcal{L}_{\mathcal{M}}$

## Definición

- El lenguaje aceptado por la máquina universal $\mathcal{M}$ se conoce como lenguaje universal, denotado $\mathcal{L}_{\mathcal{M}}$.

$$\mathcal{L}_{\mathcal{M}} = \{\langle M \rangle \# \langle w \rangle \mid M \text{ acepta a } w\}$$

$\langle M \rangle \# \langle w \rangle$ es la cadena de entrada con la máquina y la cadena codificadas.

# El lenguaje universal $\mathcal{L}_{\mathcal{M}}$

## Definición

- El lenguaje aceptado por la máquina universal $\mathcal{M}$ se conoce como lenguaje universal, denotado $\mathcal{L}_{\mathcal{M}}$.

$$\mathcal{L}_{\mathcal{M}} = \{\langle M \rangle \# \langle w \rangle \mid M \text{ acepta a } w\}$$

$\langle M \rangle \# \langle w \rangle$ es la cadena de entrada con la máquina y la cadena codificadas.

- Y es recursivamente enumerable, analicemos a continuación porqué no es recursivo.

# El lenguaje universal $\mathcal{L}_{\mathcal{M}}$ no es recursivo

Supongamos lo contrario y sea $M$ una máquina de Turing que siempre se detiene y tal que $\mathcal{L}_{\mathcal{M}} = L(M)$.
Veamos que a partir de $M$ podemos construir una máquina de Turing $M'$ que acepte al lenguaje diagonal $\mathcal{L}_D$, lo cual es absurdo.
$M'$ funciona como sigue para la entrada $w \in \Sigma^\star$:

- Enumerar las cadenas de $\Sigma^\star$ hasta encontrar $k \in \mathbb{N}$ tal que $w = w_k$.

# El lenguaje universal $\mathcal{L}_{\mathcal{M}}$ no es recursivo

Supongamos lo contrario y sea $M$ una máquina de Turing que siempre se detiene y tal que $\mathcal{L}_{\mathcal{M}} = L(M)$.
Veamos que a partir de $M$ podemos construir una máquina de Turing $M'$ que acepte al lenguaje diagonal $\mathcal{L}_D$, lo cual es absurdo.
$M'$ funciona como sigue para la entrada $w \in \Sigma^\star$:

- Enumerar las cadenas de $\Sigma^\star$ hasta encontrar $k \in \mathbb{N}$ tal que $w = w_k$.
- Llamar a $M$ con entrada $\langle M_k \rangle \# \langle w_k \rangle$.

# El lenguaje universal $\mathcal{L}_{\mathcal{M}}$ no es recursivo

Supongamos lo contrario y sea $M$ una máquina de Turing que siempre se detiene y tal que $\mathcal{L}_{\mathcal{M}} = L(M)$.
Veamos que a partir de $M$ podemos construir una máquina de Turing $M'$ que acepte al lenguaje diagonal $\mathcal{L}_D$, lo cual es absurdo.
$M'$ funciona como sigue para la entrada $w \in \Sigma^\star$:

- Enumerar las cadenas de $\Sigma^\star$ hasta encontrar $k \in \mathbb{N}$ tal que $w = w_k$.
- Llamar a $M$ con entrada $\langle M_k \rangle \# \langle w_k \rangle$.
- Como $M$ siempre se detiene entonces decide si $M_k$ acepta a $w_k$.

# El lenguaje universal $\mathcal{L}_{\mathcal{M}}$ no es recursivo

Supongamos lo contrario y sea $M$ una máquina de Turing que siempre se detiene y tal que $\mathcal{L}_{\mathcal{M}} = L(M)$.
Veamos que a partir de $M$ podemos construir una máquina de Turing $M'$ que acepte al lenguaje diagonal $\mathcal{L}_D$, lo cual es absurdo.
$M'$ funciona como sigue para la entrada $w \in \Sigma^\star$:

- Enumerar las cadenas de $\Sigma^\star$ hasta encontrar $k \in \mathbb{N}$ tal que $w = w_k$.
- Llamar a $M$ con entrada $\langle M_k \rangle \# \langle w_k \rangle$.
- Como $M$ siempre se detiene entonces decide si $M_k$ acepta a $w_k$.
- En tal caso forzamos a que $M'$ acepte también a $w_k$.

# El lenguaje universal $\mathcal{L}_{\mathcal{M}}$ no es recursivo

Supongamos lo contrario y sea $M$ una máquina de Turing que siempre se detiene y tal que $\mathcal{L}_{\mathcal{M}} = L(M)$.
Veamos que a partir de $M$ podemos construir una máquina de Turing $M'$ que acepte al lenguaje diagonal $\mathcal{L}_D$, lo cual es absurdo.
$M'$ funciona como sigue para la entrada $w \in \Sigma^\star$:

- Enumerar las cadenas de $\Sigma^\star$ hasta encontrar $k \in \mathbb{N}$ tal que $w = w_k$.
- Llamar a $M$ con entrada $\langle M_k \rangle \# \langle w_k \rangle$.
- Como $M$ siempre se detiene entonces decide si $M_k$ acepta a $w_k$.
- En tal caso forzamos a que $M'$ acepte también a $w_k$.
- Esto implica que $L(M') = \mathcal{L}_D$ que es un absurdo.

# Preguntas y Problemas

- Preguntas:

# Preguntas y Problemas

- Preguntas:
  - ¿Qué $x$ número real cumple que $2x^2 - 3x + 5 = 0$?

# Preguntas y Problemas

- Preguntas:
  - ¿Qué $x$ número real cumple que $2x^2 - 3x + 5 = 0$?
  - Dadas las ciudades $a, b, c, d$ ¿Cual es la forma óptima de visitarlas sin pasar dos veces por la misma ciudad?

# Preguntas y Problemas

- Preguntas:
  - ¿Qué $x$ número real cumple que $2x^2 - 3x + 5 = 0$?
  - Dadas las ciudades $a, b, c, d$ ¿Cual es la forma óptima de visitarlas sin pasar dos veces por la misma ciudad?
- Una problema es una clase de preguntas:

# Preguntas y Problemas

- Preguntas:
  - ¿Qué $x$ número real cumple que $2x^2 - 3x + 5 = 0$?
  - Dadas las ciudades $a, b, c, d$ ¿Cual es la forma óptima de visitarlas sin pasar dos veces por la misma ciudad?
- Una problema es una clase de preguntas:
  - ¿Cuales son las soluciones de ¿$ax^2 + bx + c = 0$?

# Preguntas y Problemas

- Preguntas:
  - ¿Qué $x$ número real cumple que $2x^2 - 3x + 5 = 0$?
  - Dadas las ciudades $a, b, c, d$ ¿Cual es la forma óptima de visitarlas sin pasar dos veces por la misma ciudad?
- Una problema es una clase de preguntas:
  - ¿Cuales son las soluciones de ¿$ax^2 + bx + c = 0$?
  - ¿ Dados $n$ vértices en un grafo existirá un camino hamiltoniano?

# Preguntas y Problemas

- Preguntas:
  - ¿Qué $x$ número real cumple que $2x^2 - 3x + 5 = 0$?
  - Dadas las ciudades $a, b, c, d$ ¿Cual es la forma óptima de visitarlas sin pasar dos veces por la misma ciudad?
- Una problema es una clase de preguntas:
  - ¿Cuales son las soluciones de ¿$ax^2 + bx + c = 0$?
  - ¿ Dados $n$ vértices en un grafo existirá un camino hamiltoniano?
- Cada caso particular de un problema es un ejemplo, ejemplar o instancia de éste.

# Tipos de Problemas

## Introducción

La mayoría de los problemas de interés en ciencias de la computación son de dos tipos.

1. Problemas de cómputo: obtener el valor de una función en un argumento dado.

# Tipos de Problemas

## Introducción

La mayoría de los problemas de interés en ciencias de la computación son de dos tipos.

1. Problemas de cómputo: obtener el valor de una función en un argumento dado.

   Por ejemplo obtener la raíz cuadrada de un número dado $x$ con exactitud de milésimas.

# Tipos de Problemas

## Introducción

La mayoría de los problemas de interés en ciencias de la computación son de dos tipos.

1. Problemas de cómputo: obtener el valor de una función en un argumento dado.

   Por ejemplo obtener la raíz cuadrada de un número dado $x$ con exactitud de milésimas.

2. Problemas de decisión: problemas cuya respuesta es si o no.

# Tipos de Problemas

## Introducción

La mayoría de los problemas de interés en ciencias de la computación son de dos tipos.

1. Problemas de cómputo: obtener el valor de una función en un argumento dado.

   Por ejemplo obtener la raíz cuadrada de un número dado $x$ con exactitud de milésimas.

2. Problemas de decisión: problemas cuya respuesta es si o no.

   Por ejemplo, decidir la existencia de un camino óptimo en costos para recorrer varias ciudades.

# Tipos de Problemas

## Introducción

La mayoría de los problemas de interés en ciencias de la computación son de dos tipos.

1. Problemas de cómputo: obtener el valor de una función en un argumento dado.

   Por ejemplo obtener la raíz cuadrada de un número dado $x$ con exactitud de milésimas.

2. Problemas de decisión: problemas cuya respuesta es si o no.

   Por ejemplo, decidir la existencia de un camino óptimo en costos para recorrer varias ciudades.

# Tipos de Problemas

## Introducción

La mayoría de los problemas de interés en ciencias de la computación son de dos tipos.

1. Problemas de cómputo: obtener el valor de una función en un argumento dado.

   Por ejemplo obtener la raíz cuadrada de un número dado $x$ con exactitud de milésimas.

2. Problemas de decisión: problemas cuya respuesta es si o no.

   Por ejemplo, decidir la existencia de un camino óptimo en costos para recorrer varias ciudades.

Si bien los problemas de decisión también podrían considerarse problemas de cómputo, es útil hacer la distinción.

# Algoritmos y problemas

- Un algoritmo para un problema consiste de una serie de instrucciones capaces de responder cualquier instancia del problema dado.

# Algoritmos y problemas

- Un algoritmo para un problema consiste de una serie de instrucciones capaces de responder cualquier instancia del problema dado.
- Un problema $P$ se dice soluble si existe un algoritmo para $P$. En otro caso se dice insoluble.

# Algoritmos y problemas

- Un algoritmo para un problema consiste de una serie de instrucciones capaces de responder cualquier instancia del problema dado.
- Un problema $P$ se dice soluble si existe un algoritmo para $P$. En otro caso se dice insoluble.
- Si un problema de decisión $P$ es soluble entonces decimos que $P$ es **decidible**. En caso contrario el problema se dice **indecidible**.

# Algoritmos y problemas

- Un algoritmo para un problema consiste de una serie de instrucciones capaces de responder cualquier instancia del problema dado.
- Un problema $P$ se dice soluble si existe un algoritmo para $P$. En otro caso se dice insoluble.
- Si un problema de decisión $P$ es soluble entonces decimos que $P$ es **decidible**. En caso contrario el problema se dice **indecidible**.
- Un proceso o algoritmo para un problema de decisión se conoce como un proceso o algoritmo de decisión.

# Algoritmos y funciones

- Cualquier algoritmo puede verse como una función:

$$\text{entrada } x \rightarrow \quad \text{algoritmo} \quad \rightarrow \quad \text{salida } y$$

# Algoritmos y funciones

- Cualquier algoritmo puede verse como una función:

$$\text{entrada } x \rightarrow \quad \text{algoritmo} \quad \rightarrow \quad \text{salida } y$$

- La salida está en función de la entrada $f(x) = y$

# Algoritmos y funciones

- Cualquier algoritmo puede verse como una función:

$$\text{entrada } x \rightarrow \quad \text{algoritmo} \quad \rightarrow \quad \text{salida } y$$

- La salida está en función de la entrada $f(x) = y$
- Una función es computable si existe un algoritmo que la calcule.

# Algoritmos y funciones

- Cualquier algoritmo puede verse como una función:

$$\text{entrada } x \rightarrow \quad \text{algoritmo} \quad \rightarrow \quad \text{salida } y$$

- La salida está en función de la entrada $f(x) = y$
- Una función es computable si existe un algoritmo que la calcule.
- La teoría de la computabilidad se encarga esencialmente a contestar la pregunta **¿Qué funciones son computables?**

# Algoritmos y funciones

- Cualquier algoritmo puede verse como una función:

$$\text{entrada } x \rightarrow \quad \text{algoritmo} \quad \rightarrow \quad \text{salida } y$$

- La salida está en función de la entrada $f(x) = y$
- Una función es computable si existe un algoritmo que la calcule.
- La teoría de la computabilidad se encarga esencialmente a contestar la pregunta **¿Qué funciones son computables?**
- Lo cual equivale entonces a responder que problemas son solubles.

# Problemas no computables

## Introducción

- ¿Por qué nos interesa averiguar qué problemas no son computables mediante un algoritmo o proceso?

# Problemas no computables

## Introducción

- ¿Por qué nos interesa averiguar qué problemas no son computables mediante un algoritmo o proceso?
- Tales problemas son aquellos que **no** podemos resolver.

# Problemas no computables

## Introducción

- ¿Por qué nos interesa averiguar qué problemas no son computables mediante un algoritmo o proceso?
- Tales problemas son aquellos que **no** podemos resolver.
- Son resultados fundamentales y debemos conocerlos para tener una visión general de las ciencias de la computación.

# Problemas no computables

## Introducción

- ¿Por qué nos interesa averiguar qué problemas no son computables mediante un algoritmo o proceso?
- Tales problemas son aquellos que **no** podemos resolver.
- Son resultados fundamentales y debemos conocerlos para tener una visión general de las ciencias de la computación.
- Debemos conocerlos para evitar intentar resolverlos.

# Problemas no computables

## Introducción

- ¿Por qué nos interesa averiguar qué problemas no son computables mediante un algoritmo o proceso?
- Tales problemas son aquellos que **no** podemos resolver.
- Son resultados fundamentales y debemos conocerlos para tener una visión general de las ciencias de la computación.
- Debemos conocerlos para evitar intentar resolverlos.
- Debemos entender que tales problemas son insolubles independientemente del desarrollo futuro del hardware.

# Recursividad y decidibilidad

## Problemas insolubles

- Considérese una propiedad $\mathcal{P}$ acerca de máquinas de Turing.

# Recursividad y decidibilidad

## Problemas insolubles

- Considérese una propiedad $\mathcal{P}$ acerca de máquinas de Turing.
- $\mathcal{P}$ genera el problema de decisión siguiente:

  ¿Satisface la máquina $M$ la propiedad $\mathcal{P}$ ?

# Recursividad y decidibilidad

## Problemas insolubles

- Considérese una propiedad $\mathcal{P}$ acerca de máquinas de Turing.
- $\mathcal{P}$ genera el problema de decisión siguiente:

  ¿Satisface la máquina $M$ la propiedad $\mathcal{P}$ ?
- Asumiendo la tesis de Church-Turing, tal problema de decisión será decidible (soluble) si y sólo si el lenguaje

  $L = \{\langle M \rangle \mid \langle M \rangle$ es el código de una Máquina de Turing que satisface $\mathcal{P}\}$

  es recursivo.

# El problema de la detención

*Halting problem*

Dada una máquina $M$ y una cadena $w$
¿Se detendrá $M$ al procesar $w$?

# El problema de la detención

*Halting problem*

Dada una máquina $M$ y una cadena $w$
¿Se detendrá $M$ al procesar $w$?

- El problema será soluble si pudieramos hallar una máquina $H$ tal que al recibir como entrada a cualquier cadena $\langle M \rangle \# \langle w \rangle$ se detuviera si y sólo si $M$ se detiene al procesar $w$.

# El problema de la detención

*Halting problem*

Dada una máquina $M$ y una cadena $w$
¿Se detendrá $M$ al procesar $w$?

- El problema será soluble si pudieramos hallar una máquina $H$ tal que al recibir como entrada a cualquier cadena $\langle M \rangle \# \langle w \rangle$ se detuviera si y sólo si $M$ se detiene al procesar $w$.
- Podría pensarse que una máquina universal puede hacer el trabajo.

# El problema de la detención

*Halting problem*

- El problema de la detención resulta indecidible, es decir, no existe tal máquina $H$.

# El problema de la detención

*Halting problem*

- El problema de la detención resulta indecidible, es decir, no existe tal máquina $H$.
- Además es quizás el problema indecidible más relevante en la teoría de la computabilidad.

# El problema de la detención

*Halting problem*

- El problema de la detención resulta indecidible, es decir, no existe tal máquina $H$.
- Además es quizás el problema indecidible más relevante en la teoría de la computabilidad.
- Una consecuencia inmediata de su indecidibilidad es que no puede existir un programa que verifique si cualquier programa dado se cicla.

# El problema universal

## Problemas no computables

Dada una máquina de Turing cualquiera $M$ y una cadena $w$
¿Acepta $M$ a $w$?

# El problema universal

## Problemas no computables

Dada una máquina de Turing cualquiera $M$ y una cadena $w$
¿Acepta $M$ a $w$?

- El problema universal equivale a que el lenguaje universal

$$\mathcal{L}_{\mathcal{M}} = \{\langle M \rangle \# \langle w \rangle \mid M \text{ acepta a } w\}$$

sea recursivo.

# El problema universal

## Problemas no computables

Dada una máquina de Turing cualquiera $M$ y una cadena $w$
¿Acepta $M$ a $w$?

- El problema universal equivale a que el lenguaje universal

$$\mathcal{L}_{\mathcal{M}} = \{\langle M \rangle \# \langle w \rangle \mid M \text{ acepta a } w\}$$

  sea recursivo.
- Ya demostramos que $\mathcal{L}_{\mathcal{M}}$ no es recursivo.

# El problema universal

## Problemas no computables

Dada una máquina de Turing cualquiera $M$ y una cadena $w$
¿Acepta $M$ a $w$?

- El problema universal equivale a que el lenguaje universal

$$\mathcal{L}_{\mathcal{M}} = \{\langle M \rangle \# \langle w \rangle \mid M \text{ acepta a } w\}$$

  sea recursivo.
- Ya demostramos que $\mathcal{L}_{\mathcal{M}}$ no es recursivo.
- Por lo tanto el problema universal es indecidible.

# Reducibilidad de problemas

- La reducibilidad de problemas es una técnica de gran utilidad para probar la indecidibilidad de un problema dado a partir de la indecidibilidad de un problema conocido.

# Reducibilidad de problemas

- La reducibilidad de problemas es una técnica de gran utilidad para probar la indecidibilidad de un problema dado a partir de la indecidibilidad de un problema conocido.
- Dados dos problemas $\mathcal{P}_1, \mathcal{P}_2$ decimos que $\mathcal{P}_1$ se reduce a $\mathcal{P}_2$ si un algoritmo para decidir $\mathcal{P}_2$ puede emplearse para decidir $\mathcal{P}_1$.

# Reducibilidad de problemas

Formalmente decimos que $\mathcal{P}_1$ se reduce a $\mathcal{P}_2$, denotado $\mathcal{P}_1 \prec \mathcal{P}_2$ si existe una Máquina de Turing $M$ tal que:

# Reducibilidad de problemas

Formalmente decimos que $\mathcal{P}_1$ se reduce a $\mathcal{P}_2$, denotado $\mathcal{P}_1 \prec \mathcal{P}_2$ si existe una Máquina de Turing $M$ tal que:

- $M$ recibe como entrada una instancia $I_1$ de $\mathcal{P}_1$

# Reducibilidad de problemas

Formalmente decimos que $\mathcal{P}_1$ se reduce a $\mathcal{P}_2$, denotado $\mathcal{P}_1 \prec \mathcal{P}_2$ si existe una Máquina de Turing $M$ tal que:

- $M$ recibe como entrada una instancia $I_1$ de $\mathcal{P}_1$
- $M$ devuelve como salida una instancia $I_2$ de $\mathcal{P}_2$.

# Reducibilidad de problemas

Formalmente decimos que $\mathcal{P}_1$ se reduce a $\mathcal{P}_2$, denotado $\mathcal{P}_1 \prec \mathcal{P}_2$ si existe una Máquina de Turing $M$ tal que:

- $M$ recibe como entrada una instancia $I_1$ de $\mathcal{P}_1$
- $M$ devuelve como salida una instancia $I_2$ de $\mathcal{P}_2$.
- $M$ decide a $I_1$ de la misma manera que $I_2$.

# Reducibilidad de problemas

Formalmente decimos que $\mathcal{P}_1$ se reduce a $\mathcal{P}_2$, denotado $\mathcal{P}_1 \prec \mathcal{P}_2$ si existe una Máquina de Turing $M$ tal que:

- $M$ recibe como entrada una instancia $I_1$ de $\mathcal{P}_1$
- $M$ devuelve como salida una instancia $I_2$ de $\mathcal{P}_2$.
- $M$ decide a $I_1$ de la misma manera que $I_2$.
- Es decir, $M$ responde con *sí* a $I_1$ si y sólo si responde con *sí* a $I_2$.

# Reducibilidad de problemas

Formalmente decimos que $\mathcal{P}_1$ se reduce a $\mathcal{P}_2$, denotado $\mathcal{P}_1 \prec \mathcal{P}_2$ si existe una Máquina de Turing $M$ tal que:

- $M$ recibe como entrada una instancia $I_1$ de $\mathcal{P}_1$
- $M$ devuelve como salida una instancia $I_2$ de $\mathcal{P}_2$.
- $M$ decide a $I_1$ de la misma manera que $I_2$.
- Es decir, $M$ responde con *sí* a $I_1$ si y sólo si responde con *sí* a $I_2$.

# Reducibilidad de problemas

Formalmente decimos que $\mathcal{P}_1$ se reduce a $\mathcal{P}_2$, denotado $\mathcal{P}_1 \prec \mathcal{P}_2$ si existe una Máquina de Turing $M$ tal que:

- $M$ recibe como entrada una instancia $I_1$ de $\mathcal{P}_1$
- $M$ devuelve como salida una instancia $I_2$ de $\mathcal{P}_2$.
- $M$ decide a $I_1$ de la misma manera que $I_2$.
- Es decir, $M$ responde con *sí* a $I_1$ si y sólo si responde con *sí* a $I_2$.

De esta manera se tiene que $\mathcal{P}_1$ es decidible si y sólo si $\mathcal{P}_2$ es decidible.

# PD $\prec$ PU

## Reducibilidad

**PD:** Dada una máquina $M$ y una cadena $w$ ¿Se detendrá $M$ al procesar $w$?
**PU:** Dada una máquina de Turing cualquiera $M$ y una cadena $w$ ¿$M$ acepta $w$?

# PD $\prec$ PU

## Reducibilidad

**PD:** Dada una máquina $M$ y una cadena $w$ ¿Se detendrá $M$ al procesar $w$?
**PU:** Dada una máquina de Turing cualquiera $M$ y una cadena $w$ ¿$M$ acepta $w$?

El problema universal puede reducirse al problema de la detención:

- Supongamos que existe una máquina $H$ que decide el problema de la detención.

# PD $\prec$ PU

## Reducibilidad

**PD:** Dada una máquina $M$ y una cadena $w$ ¿Se detendrá $M$ al procesar $w$?
**PU:** Dada una máquina de Turing cualquiera $M$ y una cadena $w$ ¿$M$ acepta $w$?

El problema universal puede reducirse al problema de la detención:

- Supongamos que existe una máquina $H$ que decide el problema de la detención.
- En tal caso $H$ decide también al problema universal.

# PD $\prec$ PU

## Reducibilidad

**PD:** Dada una máquina $M$ y una cadena $w$ ¿Se detendrá $M$ al procesar $w$?
**PU:** Dada una máquina de Turing cualquiera $M$ y una cadena $w$ ¿$M$ acepta $w$?

El problema universal puede reducirse al problema de la detención:

- Supongamos que existe una máquina $H$ que decide el problema de la detención.
- En tal caso $H$ decide también al problema universal.
- Al recibir una entrada $\langle M \rangle 0 \langle w \rangle$, $H$ decide si $M$ se detiene o no con entrada $w$.

# PD $\prec$ PU

## Reducibilidad

**PD:** Dada una máquina $M$ y una cadena $w$ ¿Se detendrá $M$ al procesar $w$?
**PU:** Dada una máquina de Turing cualquiera $M$ y una cadena $w$ ¿$M$ acepta $w$?

El problema universal puede reducirse al problema de la detención:

- Supongamos que existe una máquina $H$ que decide el problema de la detención.
- En tal caso $H$ decide también al problema universal.
- Al recibir una entrada $\langle M \rangle 0 \langle w \rangle$, $H$ decide si $M$ se detiene o no con entrada $w$.
- Si $M$ no se detiene con $w$ entonces $M$ no acepta a $w$.

# PD $\prec$ PU

## Reducibilidad

**PD:** Dada una máquina $M$ y una cadena $w$ ¿Se detendrá $M$ al procesar $w$?
**PU:** Dada una máquina de Turing cualquiera $M$ y una cadena $w$ ¿$M$ acepta $w$?

El problema universal puede reducirse al problema de la detención:

- Supongamos que existe una máquina $H$ que decide el problema de la detención.
- En tal caso $H$ decide también al problema universal.
- Al recibir una entrada $\langle M \rangle 0 \langle w \rangle$, $H$ decide si $M$ se detiene o no con entrada $w$.
- Si $M$ no se detiene con $w$ entonces $M$ no acepta a $w$.
- Si $M$ se detiene con $w$ entonces $M$ procesa a $w$ y decide si la acepta o no.

# PD $\prec$ PU

## Reducibilidad

**PD:** Dada una máquina $M$ y una cadena $w$ ¿Se detendrá $M$ al procesar $w$?
**PU:** Dada una máquina de Turing cualquiera $M$ y una cadena $w$ ¿$M$ acepta $w$?

El problema universal puede reducirse al problema de la detención:

- Supongamos que existe una máquina $H$ que decide el problema de la detención.
- En tal caso $H$ decide también al problema universal.
- Al recibir una entrada $\langle M \rangle 0 \langle w \rangle$, $H$ decide si $M$ se detiene o no con entrada $w$.
- Si $M$ no se detiene con $w$ entonces $M$ no acepta a $w$.
- Si $M$ se detiene con $w$ entonces $M$ procesa a $w$ y decide si la acepta o no.
- De manera que el problema universal se decide, lo cual es absurdo.

# Otros problemas indecidibles

## Reducibilidad

- Detención con cinta en blanco: ¿ se detiene la máquina $M$ al iniciar con la cinta en blanco?

# Otros problemas indecidibles

## Reducibilidad

- Detención con cinta en blanco: ¿ se detiene la máquina $M$ al iniciar con la cinta en blanco?
- Impresión de un símbolo: Dada $M$ y $s \in \Sigma$ ¿Escribirá $M$ en algún momento a $s$ sobre la cinta?

# Otros problemas indecidibles

## Reducibilidad

- Detención con cinta en blanco: ¿ se detiene la máquina $M$ al iniciar con la cinta en blanco?
- Impresión de un símbolo: Dada $M$ y $s \in \Sigma$ ¿Escribirá $M$ en algún momento a $s$ sobre la cinta?
- Dada una gramática libre de contexto $G$ ¿ $G$ es ambigua?

# Otros problemas indecidibles

## Reducibilidad

- Detención con cinta en blanco: ¿ se detiene la máquina $M$ al iniciar con la cinta en blanco?
- Impresión de un símbolo: Dada $M$ y $s \in \Sigma$ ¿Escribirá $M$ en algún momento a $s$ sobre la cinta?
- Dada una gramática libre de contexto $G$ ¿ $G$ es ambigua?
- Determinar si dos GLC son equivalentes.

# Otros problemas indecidibles

## Reducibilidad

- Detención con cinta en blanco: ¿ se detiene la máquina $M$ al iniciar con la cinta en blanco?
- Impresión de un símbolo: Dada $M$ y $s \in \Sigma$ ¿Escribirá $M$ en algún momento a $s$ sobre la cinta?
- Dada una gramática libre de contexto $G$ ¿ $G$ es ambigua?
- Determinar si dos GLC son equivalentes.
- Cualquier propiedad no trivial acerca de Máquina de Turing (Teorema de Rice).

# Complejidad

## Introducción

- Las Máquinas de Turing son el formalismo más útil en el análisis de algoritmos, debido a que modelan de manera precisa los conceptos centrales de cómputo, almacenamiento, espacio y tiempo.

# Complejidad

## Introducción

- Las Máquinas de Turing son el formalismo más útil en el análisis de algoritmos, debido a que modelan de manera precisa los conceptos centrales de cómputo, almacenamiento, espacio y tiempo.
- La noción formal de cómputo permite precisar sin ambigüedades el tiempo de computación de un problema.

# Complejidad

## Introducción

- Las Máquinas de Turing son el formalismo más útil en el análisis de algoritmos, debido a que modelan de manera precisa los conceptos centrales de cómputo, almacenamiento, espacio y tiempo.
- La noción formal de cómputo permite precisar sin ambigüedades el tiempo de computación de un problema.
- Las celdas de la cinta formalizan de manera clara la noción de espacio de almacenamiento (memoria)

# Complejidad

## Introducción

- Las Máquinas de Turing son el formalismo más útil en el análisis de algoritmos, debido a que modelan de manera precisa los conceptos centrales de cómputo, almacenamiento, espacio y tiempo.
- La noción formal de cómputo permite precisar sin ambigüedades el tiempo de computación de un problema.
- Las celdas de la cinta formalizan de manera clara la noción de espacio de almacenamiento (memoria)
- Así las nociones de tiempo y espacio se modelan de forma muy realista mediante una Máquina de Turing, lo cual permite analizar la complejidad computacional de un problema.

# Complejidad

## Introducción

- La teoría de complejidad se interesa por el estudio de la complejidad necesaria para resolver un problema.

# Complejidad

## Introducción

- La teoría de complejidad se interesa por el estudio de la complejidad necesaria para resolver un problema.
- En particular por el consumo de recursos (espacio y tiempo).

# Complejidad

## Introducción

- La teoría de complejidad se interesa por el estudio de la complejidad necesaria para resolver un problema.
- En particular por el consumo de recursos (espacio y tiempo).
- **No le conciernen los problemas insolubles.**

# Complejidad

## Introducción

- La teoría de complejidad se interesa por el estudio de la complejidad necesaria para resolver un problema.
- En particular por el consumo de recursos (espacio y tiempo).
- **No le conciernen los problemas insolubles.**
- Pero tampoco todos los problemas solubles, en la práctica aquellos problemas solubles que no pueden resolverse en un tiempo razonable son tan despreciables como un problema insoluble.

# Complejidad

## Introducción

- La teoría de complejidad se interesa por el estudio de la complejidad necesaria para resolver un problema.
- En particular por el consumo de recursos (espacio y tiempo).
- **No le conciernen los problemas insolubles.**
- Pero tampoco todos los problemas solubles, en la práctica aquellos problemas solubles que no pueden resolverse en un tiempo razonable son tan despreciables como un problema insoluble.
- Los problemas se clasifican en clases de complejidad de acuerdo a qué tan difícil es resolverlos.

# La función de tiempo de ejecución de una Máquina de Turing

## Complejidad

La función tiempo de ejecución de una Máquina de Turing $M$ se define como

$$t_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$$

# La función de tiempo de ejecución de una Máquina de Turing

## Complejidad

La función tiempo de ejecución de una Máquina de Turing $M$ se define como

$$t_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$$

$t_M(n) :=$ máximo número de pasos de ejecución de $M$ para una entrada de longitud $n$.

# Máquinas que corren en tiempo polinomial

## Complejidad

Una máquina de Turing $M$ corre en tiempo polinomial si:

Existe un polinomio con coeficientes enteros no negativos

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

# Máquinas que corren en tiempo polinomial

## Complejidad

Una máquina de Turing $M$ corre en tiempo polinomial si:

Existe un polinomio con coeficientes enteros no negativos

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

tal que $t_M(n) \leq p(n)$ para toda $n \in \mathbb{N}$.

# Máquinas que corren en tiempo polinomial

## Complejidad

Una máquina de Turing $M$ corre en tiempo polinomial si:

Existe un polinomio con coeficientes enteros no negativos

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

tal que $t_M(n) \leq p(n)$ para toda $n \in \mathbb{N}$.

- Es decir, si la función de tiempo de ejecución de $M$ está **acotada superiormente** por un polinomio.

# Máquinas que corren en tiempo polinomial

## Complejidad

Una máquina de Turing $M$ corre en tiempo polinomial si:

Existe un polinomio con coeficientes enteros no negativos

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

tal que $t_M(n) \leq p(n)$ para toda $n \in \mathbb{N}$.

- Es decir, si la función de tiempo de ejecución de $M$ está **acotada superiormente** por un polinomio.
- Una máquina que corre en tiempo polinomial siempre se detiene.

# Clases de complejidad

## P

- Clase de problemas que pueden ser resueltos eficientemente.

# Clases de complejidad

## P

- Clase de problemas que pueden ser resueltos eficientemente.
- Eficientemente significa que existe un algoritmo que corre en **tiempo polinomial**.

# Clases de complejidad

## P

- Clase de problemas que pueden ser resueltos eficientemente.
- Eficientemente significa que existe un algoritmo que corre en **tiempo polinomial**.
- **P** se conoce también como la clase de problemas **tratables**.

# Clases de complejidad

## P

- Clase de problemas que pueden ser resueltos eficientemente.
- Eficientemente significa que existe un algoritmo que corre en **tiempo polinomial**.
- **P** se conoce también como la clase de problemas **tratables**.
- En contraste, un problema es **intratable** si es soluble pero cualquier solución algorítmica corre en tiempo exponencial en el peor de los casos.

# Clases de complejidad

## P

- Clase de problemas que pueden ser resueltos eficientemente.
- Eficientemente significa que existe un algoritmo que corre en **tiempo polinomial**.
- **P** se conoce también como la clase de problemas **tratables**.
- En contraste, un problema es **intratable** si es soluble pero cualquier solución algorítmica corre en tiempo exponencial en el peor de los casos.
- Un problema intratable es prácticamente insoluble, excepto para entradas muy pequeñas, a no ser que el caso promedio sea mucho mejor que el peor caso.

# Clases de complejidad

## NP

- Clase de problemas que pueden ser resueltos en tiempo polinomial pero sólo por un algoritmo **no determinista**.

# Clases de complejidad

## NP

- Clase de problemas que pueden ser resueltos en tiempo polinomial pero sólo por un algoritmo **no determinista**.
- Es decir son los problemas solubles por una Máquina de Turing no-determinista en tiempo polinomial.

# Clases de complejidad

## NP

- Clase de problemas que pueden ser resueltos en tiempo polinomial pero sólo por un algoritmo **no determinista**.
- Es decir son los problemas solubles por una Máquina de Turing no-determinista en tiempo polinomial.
- Sabemos que las Máquina de Turing no-deterministas equivalen a las Máquina de Turing deterministas.

# Clases de complejidad

## NP

- Clase de problemas que pueden ser resueltos en tiempo polinomial pero sólo por un algoritmo **no determinista**.
- Es decir son los problemas solubles por una Máquina de Turing no-determinista en tiempo polinomial.
- Sabemos que las Máquina de Turing no-deterministas equivalen a las Máquina de Turing deterministas.
- Sin embargo no se sabe en general si un problema soluble mediante un algoritmo no-determinista tendrá una solución determinista.

# P vs NP

- Dado un problema $\mathcal{P} \in \mathbf{NP}$ en general no sabemos si existirá una solución determinista.

# P vs NP

- Dado un problema $\mathcal{P} \in \mathbf{NP}$ en general no sabemos si existirá una solución determinista.
- Si $\mathbf{P} = \mathbf{NP}$ entonces la respuesta es afirmativa.

# P vs NP

- Dado un problema $\mathcal{P} \in \mathbf{NP}$ en general no sabemos si existirá una solución determinista.
- Si $\mathbf{P} = \mathbf{NP}$ entonces la respuesta es afirmativa.
- Aun no se ha probado ni refutado si $\mathbf{P}=\mathbf{NP}$.

# P vs NP

- Dado un problema $\mathcal{P} \in \mathbf{NP}$ en general no sabemos si existirá una solución determinista.
- Si $\mathbf{P} = \mathbf{NP}$ entonces la respuesta es afirmativa.
- Aun no se ha probado ni refutado si $\mathbf{P}=\mathbf{NP}$.
- En caso positivo siempre habrí un algoritmo determinista para un problema cuya solución no-determinista se conoce.

# P vs NP

- Dado un problema $\mathcal{P} \in \mathbf{NP}$ en general no sabemos si existirá una solución determinista.
- Si $\mathbf{P} = \mathbf{NP}$ entonces la respuesta es afirmativa.
- Aun no se ha probado ni refutado si $\mathbf{P}=\mathbf{NP}$.
- En caso positivo siempre habrí un algoritmo determinista para un problema cuya solución no-determinista se conoce.
- Se sospecha que $\mathbf{P}\neq\mathbf{NP}$ y esta es la pregunta más importante en la teoria de la complejidad (la respuesta vale 1 millón de dólares).

# Problemas **NP**-completos

- Los problemas **NP**-completos son los más difíciles de la clase **NP**.

# Problemas **NP**-completos

- Los problemas **NP**-completos son los más difíciles de la clase **NP**.
- Cualquier problema de la clase **NP** puede reducirse a cualquier problema **NP**-completo.

# Problemas **NP**-completos

- Los problemas **NP**-completos son los más difíciles de la clase **NP**.
- Cualquier problema de la clase **NP** puede reducirse a cualquier problema **NP**-completo.
- Existen muchos problemas **NP**-completos, el más relevante es probablemente el problema **SAT** (Teorema de Cook).

# Problemas **NP**-completos

- Un problema $\mathcal{P}$ es **NP**-completo ($\mathcal{P} \in$ **NPC**) si:

# Problemas **NP**-completos

- Un problema $\mathcal{P}$ es **NP**-completo ($\mathcal{P} \in$ **NPC**) si:
  - $\mathcal{P} \in$ **NP**.

# Problemas **NP**-completos

- Un problema $\mathcal{P}$ es **NP**-completo ($\mathcal{P} \in$ **NPC**) si:
  - $\mathcal{P} \in$ **NP**.
  - $\mathcal{Q} \prec \mathcal{P}$ en tiempo polinomial para cualquier problema $\mathcal{Q} \in$ **NP**.

# Problemas **NP**-completos

- Un problema $\mathcal{P}$ es **NP**-completo ($\mathcal{P} \in$ **NPC**) si:
  - $\mathcal{P} \in$ **NP**.
  - $\mathcal{Q} \prec \mathcal{P}$ en tiempo polinomial para cualquier problema $\mathcal{Q} \in$ **NP**.
- Todos los problemas en la clase **NPC** son equivalentes, es decir, si $A, B \in$ **NPC** entonces $A \prec B$ y $B \prec A$.

# Problemas **NP**-completos

- Un problema $\mathcal{P}$ es **NP**-completo ($\mathcal{P} \in$ **NPC**) si:
  - $\mathcal{P} \in$ **NP**.
  - $\mathcal{Q} \prec \mathcal{P}$ en tiempo polinomial para cualquier problema $\mathcal{Q} \in$ **NP**.
- Todos los problemas en la clase **NPC** son equivalentes, es decir, si $A, B \in$ **NPC** entonces $A \prec B$ y $B \prec A$.
- La segunda condición de la definición se puede intercambiar por: $C \prec \mathcal{P}$ para algún problema **NP**-completo $C$.

# El problema de satisfacción **SAT**

## Algunos problemas **NP**-completos

- Dada una fórmula de la lógica proposicional en forma normal conjuntiva:

$$P := C_1 \vee C_2 \vee C_3 \ldots C_n$$

# El problema de satisfacción **SAT**

## Algunos problemas **NP**-completos

- Dada una fórmula de la lógica proposicional en forma normal conjuntiva:

$$P := C_1 \vee C_2 \vee C_3 \ldots C_n$$

- ¿Existe una asignación de verdad que satisfaga a $P$?

# El problema de satisfacción **SAT**

## Algunos problemas **NP**-completos

- Dada una fórmula de la lógica proposicional en forma normal conjuntiva:

$$P := C_1 \vee C_2 \vee C_3 \ldots C_n$$

- ¿Existe una asignación de verdad que satisfaga a $P$?
- Teorema de Cook (1971): **SAT** es **NP**-completo.

# El problema de satisfacción **SAT**

## Algunos problemas **NP**-completos

- Dada una fórmula de la lógica proposicional en forma normal conjuntiva:

$$P := C_1 \vee C_2 \vee C_3 \ldots C_n$$

- ¿Existe una asignación de verdad que satisfaga a $P$?
- Teorema de Cook (1971): **SAT** es **NP**-completo.
- Este es el primer problema **NP**-completo.

# Circuito Hamiltoniano **PCH**

## Algunos problemas **NP**-completos

- Un circuito hamiltoniano es un camino que inicia y termina en un mismo vértice de un grafo conexo y que visita a todos los vértices exactamente una vez (el vértice de inicio y fin cuenta solo una vez).

# Circuito Hamiltoniano **PCH**

## Algunos problemas **NP**-completos

- Un circuito hamiltoniano es un camino que inicia y termina en un mismo vértice de un grafo conexo y que visita a todos los vértices exactamente una vez (el vértice de inicio y fin cuenta solo una vez).
- **PCH**: ¿Dado un grafo conexo $G$, tiene $G$ un circuito hamiltoniano?

# Problema del agente viajero **PAV**

## Algunos problemas **NP**-completos

- Se tienen dadas $m$ ciudades, las distancias entre cualesquiera dos de ellas y un entero $N$.

# Problema del agente viajero **PAV**

## Algunos problemas **NP**-completos

- Se tienen dadas $m$ ciudades, las distancias entre cualesquiera dos de ellas y un entero $N$.
- **PAV**: ¿Existe un recorrido que visite cada ciudad exactamente una vez y cuya longitud sea menor o igual que $N$ ?