Clasificación de algoritmos

Programación de estructuras de datos y algoritmos fundamentales (TC1031)

Francisco J. Navarro B. BEng, MSc, PhD
fj.navarro.barron@tec.mx

Tecnológico de Monterrey

08-2022

Contenido

1. Clases de Complejidad, P y NP de un algoritmo
   - Cómputo
   - Historia
   - Problemas de clase **P**
   - Problemas de clase NP
   - P = NP?

2. Reducción Polinomial

Jerarquía de los algoritmos

Recapitulación

Las principales notaciones *Big O* que se tienen en los algoritmos son:

Notación O	**Nombre**
$O(1)$	Constante
$O(\log\log(n))$	log log
$O(\log(n))$	Logarítmica
$O(n)$	Lineal
$O(n\log(n))$	n log n
$O(n^2)$	Cuadrática
$O(n^3)$	Cúbica
$O(n^m)$	Polinomial
$O(m^n) m \geq 2$	Exponencial
$O(n!)$	Factorial

Cómputo

Las **ciencias computacionales** se encargan de estudiar el cómo resolver los problemas (no las computadoras). A esto se le conoce como **cómputo** y se relaciona directamente con todo lo que tiene que ver con cómo generar **cálculos**.

- ¿Podemos calcularlo todo?
- ¿Qué información necesitamos para poder calcular la respuesta a alguna pregunta?

Millenium Prize Problems

Introducción

1. Birch and Swinnerton-Dyer Conjecture
2. Hodge Conjecture
3. Navier–Stokes Equation
4. P vs NP Problem
5. Riemann Hypothesis
6. Poincaré Conjecture
7. Yang–Mills and Mass Gap

Millenium Prize Problems

Introducción

1. Birch and Swinnerton-Dyer Conjecture
2. Hodge Conjecture
3. Navier–Stokes Equation
4. **P vs NP Problem** (most recent: 1971) [1]
5. ~~Riemann Hypothesis~~ (solved in 2010)
6. Poincaré Conjecture
7. Yang–Mills and Mass Gap

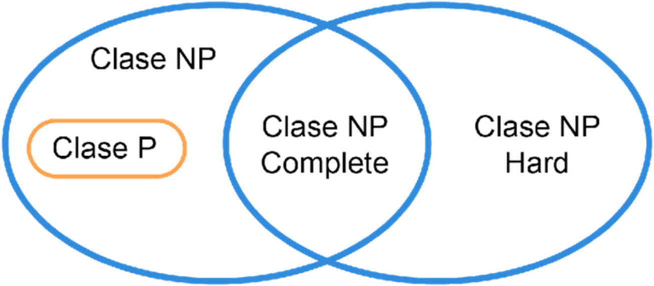
[1] http://www.claymath.org/millennium-problems/p-vs-np-problem

Historia

1970's

- Algoritmos para solucionar muchos problemas del mundo
- Algunas soluciones a problemas muy lentas → Implementaciones inteligentes.
- Ejemplo: *Traveling Salesman Problem* (TSP)
- ¿Cómo categorizarlos?
  - Ordenar → *Fast*
  - Ajedrez → *Slow*
  - TSP → ???

Clasificación de problemas



Clase *P*

Un problema es asignado a la clase **P** (tiempo polinomial) cuando el tiempo de esta solución está en una potencia constante del tamaño de la entrada n^k.

- Problemas que pueden ser resueltos por un programa razonablemente rápido.
- Ejemplos: multiplicación, ordenar alfabéticamente una lista de nombres, etc.

Determinismo vs No-determinismo

Introducción

La palabra **determinismo** hace referencia a que la solución a un problema se puede obtener después de un número **determinado** de pasos que depende del **estado** en el que se encuentra el problema. Algunas operaciones deterministas son:

- Sumar dos números.
- Multiplicar dos matrices.
- Ordenar una lista.
- Encontrar el número más pequeño en un arreglo.

Problemas P

Si en su lugar, nos limitamos a tener una máquina **determinista** (es decir que sólo puede estar en un estado), entonces sólo algunos de los problemas NP se pueden resolver en un tiempo *decente* ($O(n^m)$).

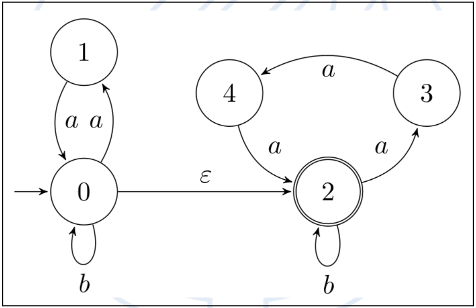
A lo mucho, en tiempo **polinomial**.

- Son problemas de **decisión**: ¿Puedes? SI o NO
- Se verifican **fácilmente**: si ya tengo una solución para comparar, puedo ver si voy por buen camino o si el problema está bien hecho.
- Se resuelven **siempre** con el mismo número de pasos (en el peor de los casos).

Determinismo vs No-determinismo

Introducción

El **no-determinismo** es lo contrario. Considera el siguiente *Autómata Finito No Determinista* donde ϵ es la transición vacía (o sea ''):



La máquina está en **dos estados distintos** a la vez y las acciones siguientes dependen del estado en el que estás... que es en ambos...

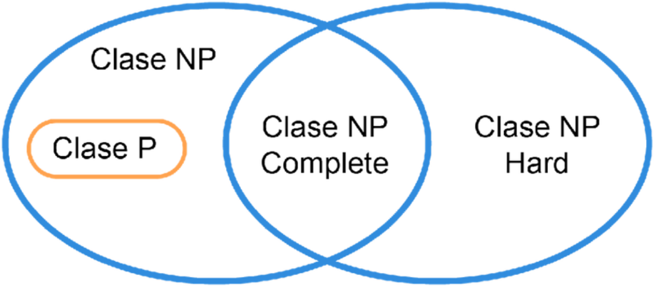
Clase *NP*

Un problema es *NP* si tiene un tiempo de ejecución polinomial en una máquina de Turing no determinística

- *NP* significa *non-deterministic polynomial*, y no *non-polynomial*
- Dada la solución correcta esta puede ser verificada en una cantidad de tiempo razonable (polinomial)
- Encontrar la solución (tiempo exponencial) es más complicado que verificarla

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

$P = NP$?



En ocasiones, se tenia suerte y se encontraban problemas NP que en realidad eran parte de la clase $P \rightarrow$ programa rápido.

$P = NP$?

La pregunta del millón: $P = NP$?

Si es fácil verificar la solución a un problema, ¿es también fácil encontrar su solución?

Otra manera de ver a *NP*

Si ambas clases de problemas son de decisión y fácilmente verificables, ¿cómo sé si algún problema es P o es NP?

- Los problemas en P los **resolvemos** con una máquina **determinista** y los **verificamos** con la **misma máquina** en tiempo **polinomial**.

Otra manera de ver a *NP*

Si ambas clases de problemas son de decisión y fácilmente verificables, ¿cómo sé si algún problema es P o es NP?

- Los problemas en P los **resolvemos** con una máquina **determinista** y los **verificamos** con la **misma máquina** en tiempo **polinomial**.
- Los problemas en NP los **resolvemos** con una máquina **no determinista** pero los **verificamos** con una máquina **determinista** en tiempo polinomial.

Para pensar...

Para cada uno de estos problemas pregúntate lo siguiente:

1. ¿Es un problema de decisión?
2. ¿Puedo verificarlo rápidamente?
3. ¿Qué complejidad me toma?
4. ¿Si uso otro tipo de máquina reduzco el tiempo de solución?

Ejemplos

- **Armar un rompecabezas**. Si comparo cada una de las n piezas contra todas las demás $(n-1)$ entonces sé qué pieza va en cada lugar $\rightarrow O(n^2)$ $\checkmark$

Ejemplos

- **Armar un rompecabezas**. Si comparo cada una de las n piezas contra todas las demás $(n-1)$ entonces sé qué pieza va en cada lugar $\rightarrow O(n^2)$ ✓
- **Sentar invitados en una mesa sin conflictos**. Por cada silla n checo que no tenga conflicto con las $n-1$ sillas restantes. Luego, por la siguiente silla, checo con las $n-2$ restantes. Y así, o sea $n \times (n-1) \times (n-2) \cdots = O(n!)$ ✗

Ejemplos

- **Armar un rompecabezas**. Si comparo cada una de las n piezas contra todas las demás $(n-1)$ entonces sé qué pieza va en cada lugar $\rightarrow O(n^2)$ ✓
- **Sentar invitados en una mesa sin conflictos**. Por cada silla n checo que no tenga conflicto con las $n-1$ sillas restantes. Luego, por la siguiente silla, checo con las $n-2$ restantes. Y así, o sea $n \times (n-1) \times (n-2) \cdots = O(n!)$ ✗
- **Empacar la mayor cantidad de valores sin exceder la capacidad de una bolsa dados ciertos objetos de valor**. No es de decisión siquiera ✗

Ejemplos

- **Armar un rompecabezas**. Si comparo cada una de las n piezas contra todas las demás $(n-1)$ entonces sé qué pieza va en cada lugar $\rightarrow O(n^2)$ ✓
- **Sentar invitados en una mesa sin conflictos**. Por cada silla n checo que no tenga conflicto con las $n-1$ sillas restantes. Luego, por la siguiente silla, checo con las $n-2$ restantes. Y así, o sea $n \times (n-1) \times (n-2) \cdots = O(n!)$ ✗
- **Empacar la mayor cantidad de valores sin exceder la capacidad de una bolsa dados ciertos objetos de valor**. No es de decisión siquiera ✗
- **Ordenar una lista de números**. El peor de los casos es que vengan en el orden contrario así que a lo mucho comparo todos contra todos los demás $\rightarrow O(n^2)$✓

Ejemplos

- **Saber si un programa finalizará dada su instrucción inicial**. Es de decisión pero es imposible de resolver ✗

¿Notas algún patrón?

Ejemplos

- **Saber si un programa finalizará dada su instrucción inicial**. Es de decisión pero es imposible de resolver ✗
- **¿Puedes visitar todos los *Starbucks* de Querétaro de tal modo que la distancia que recorras sea la menor posible?**. Empiezo en uno, y me voy a alguno, y luego a otro y luego a otro. . . y anoto su distancia. Después empiezo en otro, y reviso las otras $n-1$ posibilidades, otras $n-2$ veces $\cdots = O(n!)$✗

¿Notas algún patrón?

Ejemplos

- **Saber si un programa finalizará dada su instrucción inicial**. Es de decisión pero es imposible de resolver ✗
- **¿Puedes visitar todos los *Starbucks* de Querétaro de tal modo que la distancia que recorras sea la menor posible?**. Empiezo en uno, y me voy a alguno, y luego a otro y luego a otro. . . y anoto su distancia. Después empiezo en otro, y reviso las otras $n-1$ posibilidades, otras $n-2$ veces $\cdots = O(n!)$✗
- **Asignar salones de clase a profesores**. Pruebo para uno de los n profesores alguno de los m salones disponibles y veo si no tiene problema a esa hora en ese salón. Luego reviso que no haya conflictos con los demás $n-1$ profesores en sus $m-1$ salones. Y asigno otro $\cdots = O(n!)$✗

¿Notas algún patrón?

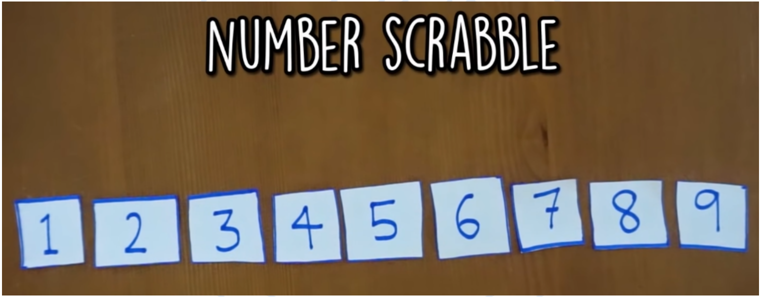
El no-determinismo de *NP*

Algunos de los que no se pueden resolver en tiempo polinomial podrían resolverse fácilmente si de manera no determinista generamos una posible solución (*good guess*) y la **verificamos** de manera determinista.

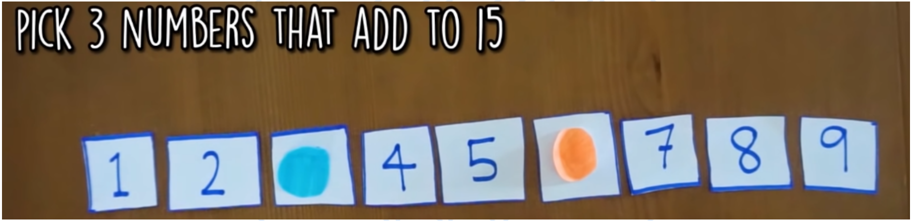
Esos son los *NP*.

¿Puedes usar el mismo método para los *P*?
Por supuesto, porque $P \subseteq NP$

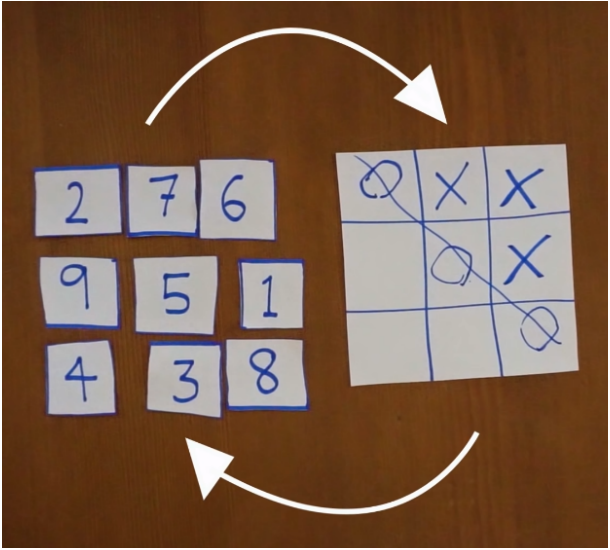
Number Scrabble



Number Scrabble

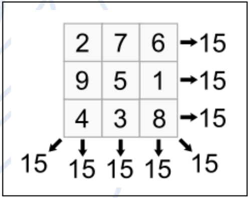


Number Scrabble



Number Scrabble

Muchos problemas en CS pueden ser reducidos al mismo problema exacto $\rightarrow$ **misma estrategia** (algoritmo).



Resolver vs Verificar

Ya vimos que **resolver** es más *difícil* que **verificar**:

- **Resolver un sudoku**. Tendrías que ir de cuadro en cuadro, probar con un número, y luego checar que dé la suma; y cambiar de uno por uno, para evitar conflictos con los $n-1$ restantes, que no tenga conflictos con los $n-2$ restantes $\cdots = O(n!)$

Resolver vs Verificar

Ya vimos que **resolver** es más *difícil* que **verificar**:

- **Resolver un sudoku**. Tendrías que ir de cuadro en cuadro, probar con un número, y luego checar que dé la suma; y cambiar de uno por uno, para evitar conflictos con los $n-1$ restantes, que no tenga conflictos con los $n-2$ restantes $\cdots = O(n!)$
- **Verificar la solución de un sudoku**. Recibo una solución propuesta y me voy de una por una en las n casillas para revisar que todo cuadre. Si cuadra, perfecto. Si no, reporto que está mal. Esto toma $O(n)$ operaciones.

Una reducción absurda

Reducción

Una reducción es una transformación de un problema *easy* a uno *harder*. Por ejemplo:

- *easy* = "No puedo levantar este auto porque pesa demasiado"
- *harder* = "Me pregunto si podré levantar este barco de carga"

Por medio de una reducción, podemos convertir el problema *easy* a un problema *harder*, por ejemplo pensando que metemos el auto dentro del barco de carga. Si podemos levantar el barco de carga, entonces podemos levantar el auto.
De este modo, resolver el problema *harder* ayudó a resolver el problema *easy*.

Otra reducción

Reducción

Una forma de simplificar las cosas es demostrar que un problema se "reduce" a otro. Por ejemplo: el problema de **hallar la mediana** de un conjunto de n números.

- Hallar el número tal que existen $n/2$ números menores o iguales que él y otros tantos $n/2$ mayores o iguales.
- Se reduce a poner los objetos en un vector y ordenarlo.
- Una vez ordenado, toma el elemento de la posición media $\rightarrow$ cota superior $=$ complejidad del algoritmo de ordenamiento

Reducción en tiempo polinomial

Haciendo una serie de transformaciones a cualquiera de nuestros problemas *NP* podemos hacer la siguiente *máquina* (algoritmo):

1. Probemos todas las posibles soluciones para nuestro problema.
2. Si una de ellas termina, entonces detente. Si no, entra en un bucle infinito.

Claramente, si el programa de 2 líneas se detiene significa que nuestro problema *NP* tenía una solución óptima.

Reducción en tiempo polinomial

- Esta reducción de nuestro problema *NP* a este otro problema de *detener la máquina* nos dice entonces que **decidir si la máquina** se detendrá es más difícil que resolver el problema *NP* (o sea, es el barco de carga de nuestro auto).
- También podemos asegurar que resolver el problema *NP* (cargar nuestro auto) es *al menos* tan difícil como resolver el problema de **decidir si la máquina se detendrá**.

Gracias!