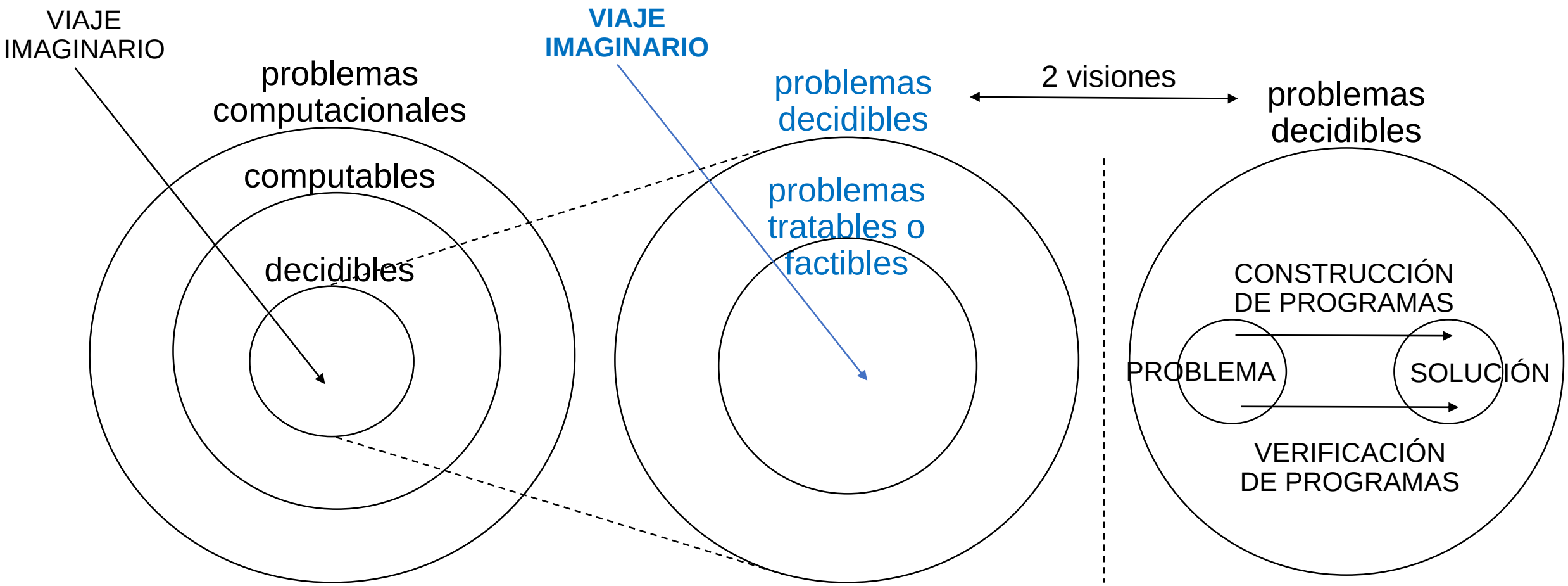
Clase 5. Complejidad Computacional. P y NP.



Jerarquía de la Computabilidad

Jerarquía de la Complejidad

Correctitud de Programas

Sigue el viaje al interior del universo de problemas

- Seguimos en el marco de los **problemas de decisión** (problemas = lenguajes).

- Seguimos utilizando las **máquinas de Turing (MT).**

- Nos focalizaremos en la **complejidad computacional temporal** (luego haremos una breve incursión a la **complejidad computacional espacial**).

- **Tiempo = cantidad de pasos ejecutados por una MT.**
- **Espacio = cantidad de celdas ocupadas por una MT.**

- Otras métricas **dinámicas** que se pueden considerar:
  - ✓ **cantidad de cambios de dirección** del cabezal de una MT.
  - ✓ **consumo de un recurso abstracto** (Blum).
  - ✓ etc.

- Otras métricas **estáticas** que se pueden considerar:
  - ✓ **el mínimo |Q|.|Γ|** de una MT que resuelva el problema analizado (Shannon).
  - ✓ **complejidad estructural de Chomsky** (de menor a mayor complejidad: autómatas finitos, autómatas con pila, autómatas linealmente acotados, máquinas de Turing).
  - ✓ etc.

Introducción a la complejidad computacional temporal

- Es natural que una MT, un algoritmo, un programa, **tarden más a medida que sus inputs sean más grandes**.

- Por eso se utilizan **funciones temporales crecientes** $T : N \rightarrow N^+$. Sea $n = |w|$, es decir la longitud del input, y sea k una constante. Funciones típicas que usaremos son:
  - ✓ $T(n) = k.n$ (lineal)
  - ✓ $T(n) = k.n^2$ (cuadrática)
  - ✓ $T(n) = k.n^c$ (polinomial, que a veces abreviaremos con poly(n))
  - ✓ $T(n) = k.c^n$ (exponencial, que a veces abreviaremos con exp(n)). En realidad, los exponentes son de la forma $d.n$ o n^d, con d constante.

  Por ejemplo, vimos que una MT con 2 cintas reconoce palíndromos en tiempo T(n) = 3.n (lineal).

- Considerar k.T(n) o T(n), con k constante, **será indistinto**:
  - ✓ Si una MT M_1 tarda k.T(n) pasos, se prueba que existe otra MT M_2 que hace lo mismo pero en sólo T(n) pasos, es decir k veces menos: en cada paso M_2 procesa juntos varios símbolos de M_1 (el alfabeto de M_2 tiene como elementos secuencias de símbolos del alfabeto de M_1). Este es el **teorema de aceleración lineal.**

  Veamos un ejemplo de esto en el siguiente slide:

- Sea nuevamente **el problema de los palíndromos**. Sea el input w = 3581991853.

- Una MT M con 1 cinta que lo resuelve de a **1 símbolo por vez** podría trabajar de esta manera:

  Al inicio se tiene: **3**581991853 (M arranca posicionada en el 1er 3)
  luego de 10 pasos queda: 581991853 (M tacha el 1er 3 y se posiciona en el último 3)
  luego de 9 pasos queda: **5**8199185 (M tacha el último 3 y se posiciona en el 1er 5)
  luego de 8 pasos queda: 819918**5** (M tacha el 1er 5 y se posiciona en el último 5)
  luego de 7 pasos queda: **8**19918 (M tacha el último 5 y se posiciona en el 1er 8)
  y así M sigue hasta llegar, en este caso, a la cinta vacía.

- En cambio, una MT M´ que resuelve el problema procesando de a **2 símbolos por vez** (los símbolos de su alfabeto son de la forma 00, 01, 02, …, 98, 99) podría trabajar de la siguiente manera:

  Al inicio se tiene: **35**81991853 (M´ arranca posicionada en el 35 de la izquierda)
  luego de 5 pasos queda: 81991**53** (M´ tacha el 35 y se posiciona en el 53 del final)
  luego de 4 pasos queda: **81**9918 (M´ tacha el 53 y se posiciona en el 81 del inicio)
  luego de 3 pasos queda 99**18** (M´ tacha el 81 y se posiciona en el 18 del final)
  y así M´ sigue hasta llegar, en este caso, a la cinta vacía.

- Es irrelevante ahora para nosotros calcular exactamente de a cuántos símbolos por vez hay que procesar para pasar de k.T(n) a T(n). **Lo cierto es que la constante se puede eliminar.**

- Por convención, respaldada por la experiencia:
  - ✓ Tiempo factible, tratable, razonable (“feasible”): $T(n) = k.n^c$ (**polinomial o poly(n)**)
  - ✓ Tiempo no factible, intratable, no razonable (“infeasible”): $T(n) = k.c^n$ (**exponencial o exp(n)**)

- Correspondientemente hablaremos de problemas de resolución factible o no factible (eficiente o ineficiente, fácil o difícil), o abreviando, directamente de **problemas factibles o no factibles.**

- Para simplificar, seremos binarios:
  - ✓ **El tiempo será polinomial o exponencial,** es decir que habrá tiempo **poly(n)** o **exp(n)**.
  - ✓ Por ejemplo, n^2 será un tiempo de resolución factible, mientras que 2^n no.

- Recordar que cuando $n \longrightarrow \infty$, poly(n) siempre está por debajo de exp(n).
  - ✓ Por ejemplo, para el caso de n^2 vs 2^n, esto se comprueba a partir de n = 5 (comprobarlo).

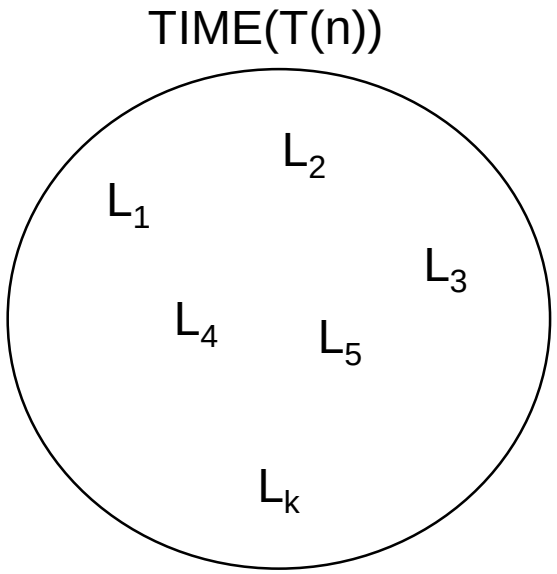
- Afortunadamente no existen casos reales de problemas con tiempos de resolución de la forma n^{100} o $2^{0,001.n}$
  - ✓ Habría problemas con la convención: ¿acaso n^{100} sería un tiempo de resolución factible y $2^{0,001.n}$ no?

Definiciones.

1. Una MT M **trabaja en tiempo T(n),** con $T : N \rightarrow N^+$, sii para todo input w, con $|w| = n$, M hace a lo sumo T(n) pasos.

2. Será útil la **notación O**, que significa “orden de”:
   $f(n) = O(g(n)) \leftrightarrow (\exists c > 0)\ (\forall n \in N)\ (f(n) \leq c.g(n))$
   Es decir, f(n) está por debajo de c.g(n), para todo número natural n.
   Por ejemplo, si $f(n) = 5n^3 + 8n + 25$, podremos escribir simplemente $f(n) = O(n^3)$ (comprobarlo)
   Otro ejemplo: $n^2 = O(n^3)$ (comprobarlo)
   Otro ejemplo: $n^3 = O(2^n)$ (comprobarlo)

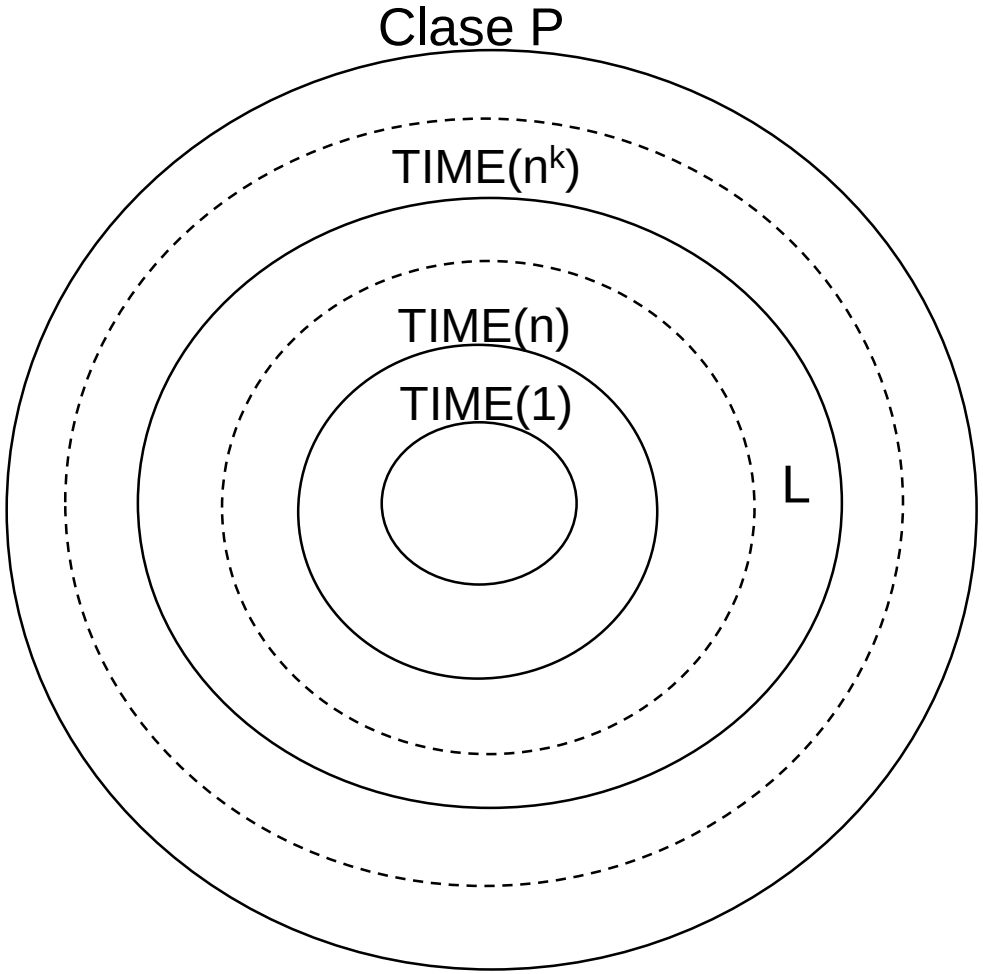
3. Un lenguaje L cumple que **$L \in$ TIME(T(n))** sii existe una MT que acepta L en tiempo O(T(n)).
   *Notar que se considera el **peor caso**, es decir el tiempo máximo que tarda una MT considerando todos los inputs posibles (algunos pocos inputs pueden perjudicar la **cota temporal superior**). Sería mejor usar el **caso promedio** (pero es técnicamente difícil). La misma dificultad aparece en la búsqueda de la **cota temporal inferior** (por ejemplo del tipo n.logn para el problema del ordenamiento o sort).*



4. La clase de los lenguajes con resolución temporal polinomial se conoce como P:

$$\mathbf{P = U_{k \geq 0}\ TIME(n^k)}$$

Un lenguaje L está en P si existe una MT que lo acepta en tiempo poly(n)



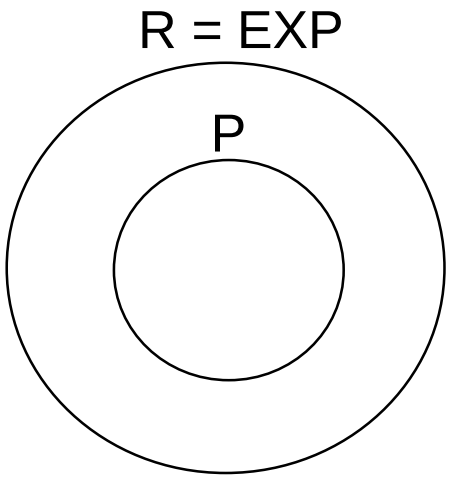
P tiene infinitos conjuntos. Su contorno es la asíntota que determina el tiempo polinomial.

- Notar que en el marco de la clase P, es indistinta la cantidad de cintas que tengan las MT consideradas: vimos antes que si una MT con k cintas tarda tiempo T(n), entonces existe una MT equivalente con 1 cinta que tarda tiempo $O(T^2(n))$, es decir que tiene un retardo cuadrático. **Por lo tanto, si una MT con k cintas tarda tiempo polinomial, también lo hace una MT equivalente con 1 cinta.**

- **¡Cuidado que no sucede lo mismo con las MTN (MT no determinísticas)!** La simulación de una MTN mediante una MTD (MT determinística) vimos que tiene retardo exponencial. **Es decir que si una MTN tarda tiempo polinomial, no significa que exista una MTD equivalente que tarde también tiempo polinomial.** TIME se refiere a MTD.

- Así las cosas, **una primera versión de la jerarquía de la complejidad temporal** es la siguiente (dada nuestra división binaria entre tiempo exponencial y polinomial, igualamos a R con EXP, la clase de problemas con resolución temporal exponencial):

Sólo dentro de P ubicamos a los problemas factibles



EXP es la clase de lenguajes aceptados por MT en tiempo exponencial. Por nuestra convención coincide con la clase R.

Se prueba que efectivamente existen lenguajes en EXP – P (diagonalizando)

- Ejemplo sencillo de problema en P: volvemos al problema de los palíndromos. Vimos que una MT que lo resuelve, con 2 cintas, hace lo siguiente:
  1. Dado el input w en la cinta 1, lo copia en la cinta 2 (tiempo O(n))
  2. Posiciona el cabezal de la cinta 1 a la izquierda (tiempo O(n)), dejando el de la cinta 2 a la derecha
  3. Compara símbolo a símbolo yendo a derecha en la cinta 1 y a izquierda en la cinta 2 (tiempo O(n))

  Por lo tanto, el tiempo total es O(n) + O(n) + O(n) = O(n), es decir es tiempo lineal.

- Otros ejemplos clásicos de problemas en P: búsqueda de un camino en un grafo, búsqueda del camino mínimo, cálculo del máximo común divisor, búsqueda del máximo flujo, etc.

- La jerarquía temporal es **densa**: dada una clase TIME($T_1(n)$), siempre se puede encontrar una clase TIME($T_2(n)$) que cumple TIME($T_1(n)$) $\subset$ TIME($T_2(n)$). Ya vimos que debe ser $T_2(n) > k.T_1(n)$.

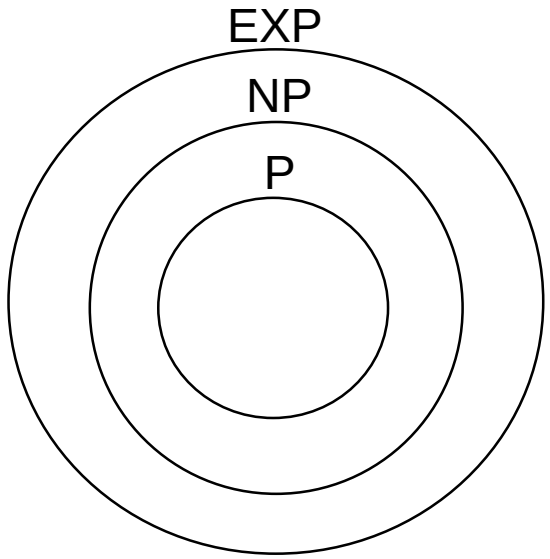
- Se trabaja con funciones T(n) **tiempo-construibles**: **las T(n) se computan en tiempo T(n),** se usan como "relojes" para justamente determinar si las MT tardan efectivamente tiempo T(n). Todas las funciones temporales habitualmente utilizadas son tiempo-construibles: n^k , c^n , n! , n.logn , etc.

- Los números se representan en notación **no unaria**:
  - Sea por ejemplo k. **En base 2 ocupa unas $\log_2 k$ celdas** (p.ej. 32 ocupa $\log_2 32 + 1 = 6$ celdas). El mismo k **en base 10 ocupa unas $\log_{10} k$ celdas**. Es decir, **k en base i ≠ 1 ocupa unas $\log_i k$ celdas.**
  - Por lo tanto, la relación entre la longitud de un número k representado en bases a ≠ 1 y b ≠ 1 es **constante**: se cumple que $\log_a k / \log_b k = \log_a b$ (p.ej. $\log_2 64 / \log_4 64 = \log_2 4 = 2$).
  - De esta manera, si una MT trabaja en tiempo poly(n) usando números en base a ≠ 1, también trabaja en tiempo poly(n) usando números en base b ≠ 1 (podemos prescindir de indicar la base).
  - La notación unaria en cambio **"trae problemas".** Que una MT trabaje en tiempo poly(n) usando números en unario, no implica que exista una MT equivalente que trabaje en tiempo poly(n) usando números en otra base. **El pasaje entre lo unario y lo no unario es exponencial.**

- **Tesis fuerte de Church-Turing (o de Edmonds-Cobham):** si un problema se resuelve en tiempo poly(n) con un modelo computacional y una representación "razonables" (p.ej., no es una MTN ni se usa notación unaria), también lo hace con otro modelo y representación "razonables".

Las clases P y NP

Una visión más detallada de la jerarquía temporal es la siguiente:



Numerosos problemas están en la clase P

Aún más en la clase NP

En general, se asume que los problemas de interés computacional no salen de la clase NP

- Un problema está en P si, como vimos antes, su solución **se construye en tiempo polinomial.**
- Un problema está en NP si su solución **se verifica en tiempo polinomial** (el nombre NP proviene de *non deterministic polynomial*, porque como veremos, esta clase se puede modelizar usando MTN).
- Si bien se sospecha que **P ≠ NP**, al día de hoy no puede probarse formalmente. Es uno de los **problemas del milenio**; quien pruebe la relación P vs NP recibirá de premio USD 1 MM.
- Claramente **P** $\subseteq$ **NP** (si construir una solución lleva tiempo poly(n), verificarla también - su propia construcción incluye su verificación -).

Ejemplo. Dado como input w un número N, se quiere encontrar un divisor de N que termine en 3.

- Una manera natural para **construir** una solución, es decir para **encontrar** un divisor de N que termine en 3, es ir dividiendo N por 3, 13, 23, 33, etc., hasta eventualmente encontrar uno.
- Dicho algoritmo ejecuta unas N/10 iteraciones (p.ej., si N = 100, hay que chequear con los divisores 3, 13, 23, 33, 43, 53, 63, 73, 83, 93).
- Así, el algoritmo hace **O(N) iteraciones**. Como el tiempo del algoritmo debe medirse en términos de **la longitud del input N**, y N (en binario) mide $n = O(\log_2 N)$, y así $N = 2^n$, queda:

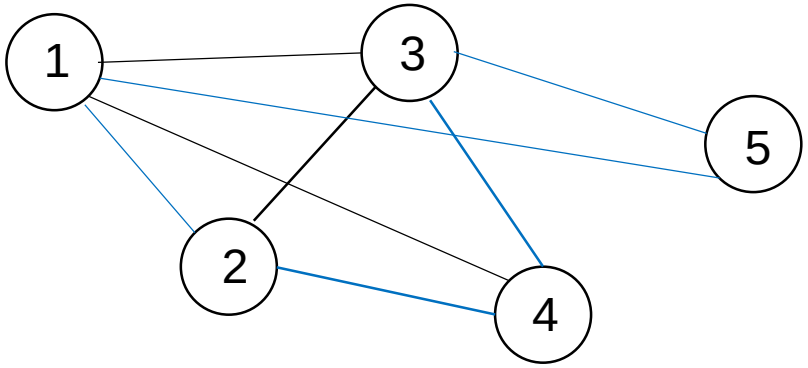
  el algoritmo tarda T(n) = $O(2^n)$, es ineficiente, "difícil".

- Sin embargo, dada una posible solución al problema, es decir un posible divisor k de N, **verificar que efectivamente: (a) k divide a N, y (b) k termina en 3, es eficiente, "fácil":**
  - (a) Dividir es una operación que se prueba tarda tiempo poly(n).
  - (b) Chequear que un número termina en 3 tarda tiempo O(n).

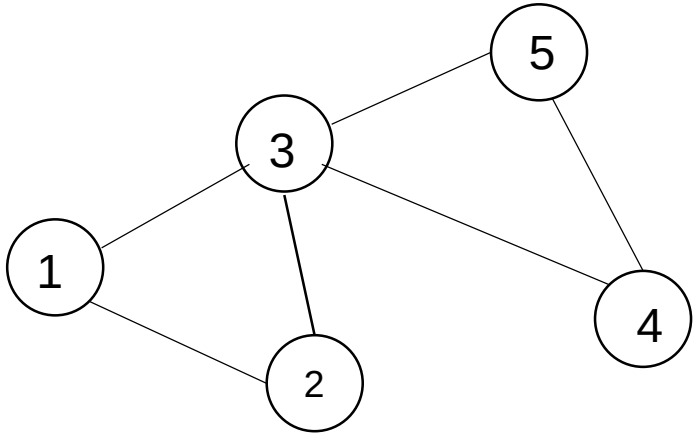
- El problema ***no pareciera estar en P***. Y claramente **está en NP**.
- Intuitivamente **P pareciera ser más chico que NP**. **Construir pareciera ser más laborioso que verificar.**
- Se puede hacer una analogía con la demostración de teoremas: demostrar un teorema intuitivamente es más difícil que verificar una demostración del mismo.

Otro ejemplo. Problema del Circuito de Hamilton (CH).

- CH = {G : G es un grafo no dirigido y tiene un circuito de Hamilton (también abreviado CH), es decir, en G se pueden recorrer todos sus vértices a través de sus arcos, arrancando y terminando en un vértice v, y sin repetir ningún otro}



Este grafo tiene un CH,
p.ej. el circuito 1,2,4,3,5,1



Este grafo no tiene un CH

- Vamos a representar los grafos G como pares (V, E) de vértices y arcos. Asumiremos en general que los arcos, y así los grafos, no son dirigidos u orientados. Usando el 2do ejemplo:
  - ✓ V = {1, 2, 3, 4, 5}, en notación binaria. Usaremos m para indicar la cantidad de vértices.
  - ✓ E = {(1,2), (1,3), (2,3), (3,4), (3,5), (4,5)}, en notación binaria.
  - ✓ Utilizaremos ## para separar V de E, # para separar los arcos de E, y la coma para separar los vértices de V y los vértices de cada arco. Volviendo al 2do ejemplo, quedaría así (en binario):

1,2,3,4,5##1,2#1,3#2,3#3,4#3,5#4,5

- **No pareciera que CH ∈ P.** Una MT natural para aceptar CH sería, dado un input G:

  1. Validar la sintaxis de G. Si es incorrecta, rechazar.
  2. Generar una permutación de V (un ordenamiento determinado de los vértices de V).
  3. Chequear si hay arcos que unen los vértices generados en (2) en el orden dado, incluyendo uno del último al primero. Si existen, aceptar.
  4. Si no quedan permutaciones de V por probar, rechazar.
  5. Volver a (2).

  Validar la sintaxis de G tarda tiempo **poly(n).** Las validaciones más importantes son:

  - Si V tiene m vértices, entonces debe tener números del 1 a m: tiempo $O(|V|) \leq O(|G|) = O(n)$
  - Los arcos de E deben tener vértices pertenecientes a V: tiempo $O(|E|.|V|) \leq O(|G|^2) = O(n^2)$

  En definitiva, el tiempo total del paso 1 es $O(n) + O(n^2) = O(n^2)$, es decir poly(n).

  En cambio, el resto del algoritmo tarda tiempo **exp(n)**:

  - Independientemente de cuánto tarda la generación de una permutación en (2), notar que en el peor caso se generan |V|! permutaciones, donde $|V|! = |V|.(|V| – 1).(|V| – 2)…2.1$. Y como |V|! no tiene la forma $|V|^k$ con k cte, entonces el tiempo será $\exp(|V|) \leq \exp(|G|) = \exp(n)$.

  En definitiva, en conjunto los pasos 2 a 5 tardan tiempo exp(n).

  Así, esta MT no tarda tiempo poly(n). **CH no estaría en P (esto no es prueba sino sospecha).**

- Por otra parte, **se prueba fácilmente que CH ∈ NP:**

  - Una posible solución, es decir un posible circuito de Hamilton C de un grafo G, tiene la forma:

    $C = (v_{i1} , v_{i2} , v_{i3} , \dots , v_{im})$

  - Verificar que C es efectivamente un circuito de Hamilton de G = (V, E) consiste en:

    1. Chequear que C es una permutación de V, es decir que C tiene la misma cantidad y los mismos vértices que V en algún orden.
       - ✓ Tiempo $O(|V|.|V|) = O(|V|^2) \leq O(|G|^2) = O(n^2)$

    2. Chequear que los arcos (v_{i1} , v_{i2}), (v_{i2} , v_{i3}), (v_{i3} , v_{i4}), …, $(v_{i\ m-1} , v_{im})$, (v_{im} , v_{i1}) están en E.
       - ✓ Tiempo $O(|V|.|E|) \leq O(|G|^2) = O(n^2)$

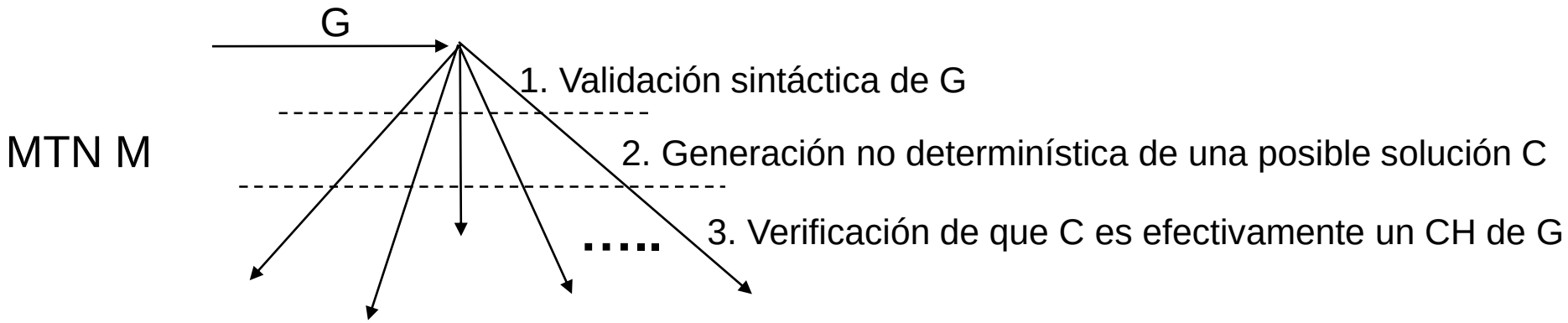
  - Por lo tanto, la verificación completa tarda tiempo poly(n). **CH está en NP.**

Notar que toda posible solución, en éste o cualquier otro problema de NP, ***debe medir poly(n)*** *(verificar una cadena de tamaño exp(n) tarda, salvo casos especiales, tiempo exp(n)).*

- Ya anticipado, **otra manera de probar que un lenguaje L está en NP es utilizando MTN.**

- **Se define que L ∈ NP sii existe una MTN que acepta L en tiempo poly(n). Una MTN trabaja en tiempo poly(n) si todas sus computaciones tardan poly(n) pasos.**

- Consideremos el mismo ejemplo de recién, el lenguaje CH de los grafos con circuitos de Hamilton. La siguiente MTN M acepta CH en tiempo poly(n). Dado un input w, M hace:
  1. Si w no es un grafo válido G, rechaza: ya vimos que esto tarda tiempo poly(n).
  2. Genera **no determinísticamente** una secuencia C de m vértices: tiempo O(n).
  3. Acepta sii C es un circuito de Hamilton de G: ya vimos que esto tarda tiempo poly(n).



- M acepta CH y todas sus computaciones tardan poly(n). Hemos vuelto a probar que CH ∈ NP, ahora usando MTN. **Notar que la MT "real" que simula M debe recorrer una a una las computaciones hasta encontrar eventualmente una solución, lo que tarda tiempo exp(n).**

- Pareciera ser que los problemas de NP se caracterizan por tener que ser resueltos necesariamente por **“fuerza bruta”** (búsqueda exhaustiva en el espacio de todas las posibles soluciones).

- En cambio, para resolver los problemas de P se debería recurrir en general al **ingenio.** Un ejemplo histórico de esto es el problema de primalidad (¿acaso el input N es un número primo?): recién en 2002 se encontró un algoritmo polinomial para resolverlo.

- Se dice que todo problema de NP tiene un **certificado (o prueba) suscinto**: mide poly(n) y se verifica en tiempo poly(n). Por ejemplo, en el caso de CH el certificado es propiamente un circuito de Hamilton.

- Con una técnica similar a la prueba de que $P \subset EXP$, se puede probar que **$NP \subset EXP$ (teorema de la unión).** Así las cosas, los problemas de EXP – NP no tienen certificados suscintos.

- Claramente **P es cerrado con respecto al complemento** (ejercicio), y **pareciera ser que NP no**. Fijémonos por ejemplo en el lenguaje CH: no pareciera que existe un certificado suscinto para determinar si un grafo NO tiene un circuito de Hamilton. Un certificado adecuado en este caso debería incluir todas las permutaciones de V (con sólo algunas no alcanzaría para el chequeo). Pero ya vimos que esta cadena no mide poly(n), es exponencial respecto del tamaño del input.

- A propósito, un camino factible para probar $P \neq NP$ podría ser encontrar una **propiedad algebraica** de la clase P que no se cumpla para la clase NP (también podría recurrirse a la lógica).

Ejemplo. El problema del clique está en NP.

- El problema consiste en determinar si un grafo tiene un clique de tamaño K. Un clique de tamaño K en un grafo G es un subgrafo completo de G con K vértices. El lenguaje que representa el problema es CLIQUE = {(G, K) | G es un grafo que tiene un clique de tamaño K}.

- La MT natural para aceptar CLIQUE consiste en recorrer, en el peor caso, todos los subconjuntos de K vértices de V, para detectar si uno de ellos es un clique de G. Esta validación tarda tiempo exponencial (**por lo que CLIQUE no estaría en P**):

  - ✓ Existen m! / ((m – K)! . K!) subconjuntos de K vértices en V, y así, el tiempo de ejecución es al menos O(m.(m – 1).(m – 2)…(m – K + 1) / K!), acotado (groseramente) por $O(m^n) = \exp(n)$.

- Por otra parte, la siguiente MTN M acepta CLIQUE en tiempo polinomial, lo que prueba que **CLIQUE está en NP**. Dada una entrada w = (G, K), M hace:
  1. Si w es inválida, rechaza.
  2. Genera no determinísticamente un conjunto C de K vértices.
  3. Acepta sii C determina un subgrafo completo incluido en G.

Queda como ejercicio probar que L(M) = CLIQUE y que la MTN M trabaja en tiempo polinomial.

Ejemplo. Una variante del problema del clique que en este caso está en la clase P.

- El enunciado del problema es el mismo que antes, pero con la salvedad de que ahora el tamaño K del clique es una constante, es decir que no forma parte de las instancias del problema.

- En otras palabras, el lenguaje que representa el problema es CLIQUE´ = {G | G es un grafo que tiene un clique de tamaño K}, con K constante.

- Notar que si K no forma parte de las entradas del algoritmo descripto previamente, el problema tiene resolución polinomial, es decir que **está en P**:

  - ✓ Hay $m! / ((m – K)! . K!) = O(m^K) \leq O(n^K)$ subconjuntos de K vértices en V.
  - ✓ El chequeo de que los mismos constituyen cliques se puede hacer en tiempo polinomial.

  Nota: podría discutirse de todos modos si para un K muy grande, por ejemplo 1000, el tiempo de la MT construida puede considerarse aceptable.

Queda como ejercicio probar que efectivamente CLIQUE´ está en P, desarrollando un poco más los párrafos anteriores (construcción de la MT y medición del tiempo de cada uno de sus pasos).

Ejemplo. Otra variante del problema del clique, ahora posiblemente fuera de NP.

- El complemento del lenguaje CLIQUE visto antes **no parece estar ni siquiera en NP**.

- Sea NOCLIQUE = {(G, K) | G es un grafo que no tiene un clique de tamaño K}.

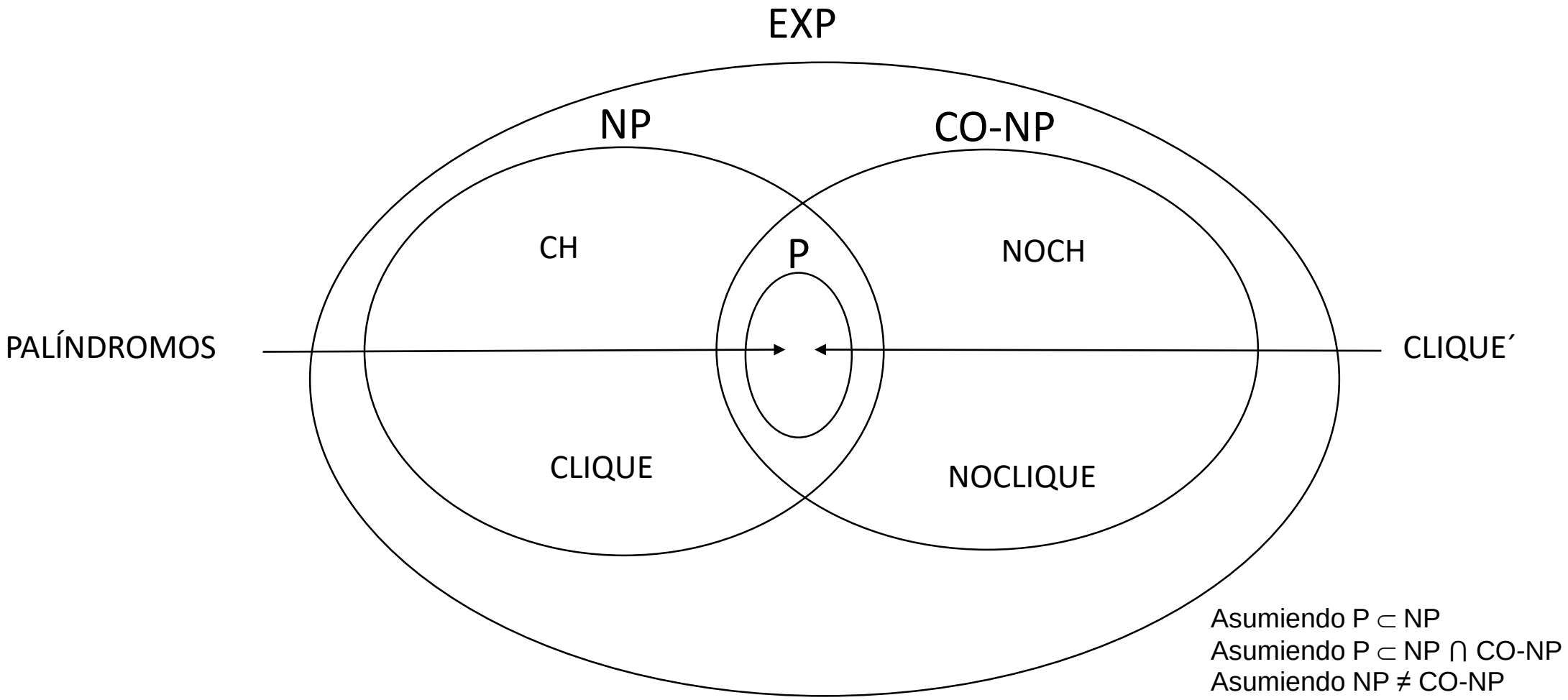
- Notar que aún utilizando una MTN, en cada una de sus computaciones deberían recorrerse todos los subconjuntos de K vértices del grafo G para aceptar o rechazar adecuadamente. En otras palabras, NOCLIQUE no pareciera tener un certificado suscinto.

- Ya dicho, **se conjetura que la clase NP no es cerrada con respecto al complemento** (p.ej., CLIQUE está en NP y NOCLIQUE pareciera que no).

- Si CO-NP es la clase de los lenguajes complemento de los lenguajes de la clase NP, se cumple claramente que **P $\subseteq$ NP $\cap$ CO-NP**:
  - ✓ P $\subseteq$ NP por definición (usando MT: una MTD es un caso particular de una MTN).
  - ✓ P $\subseteq$ CO-NP: dado L en P, entonces L^C está en P, entonces L^C está en NP, entonces L está en CO-NP.

- **Nada se puede decir de la inversa, es decir de si se cumple NP $\cap$ CO-NP $\subseteq$ P.** La conjetura es que existen lenguajes en (NP $\cap$ CO-NP) – P (ver el slide siguiente):

A diferencia del mapa de la computabilidad, **en el mapa de la complejidad temporal hay muchas asunciones** (preguntas abiertas que al día de hoy no pueden probarse).



Sí se prueba que NP ⋃ CO-NP ⊂ EXP. También que P ⊆ NP ⋂ CO-NP. Luego desglosaremos más este mapa, y lo iremos poblando con distintos ejemplares en las distintas clases de problemas.