**Práctica 5**

**Jerarquía temporal. P, NP y NPC. Reducciones poly(n).**

**Ejercicio 1. Responder y justificar brevemente las siguientes preguntas conceptuales:**

**a) ¿Por qué sólo tiene sentido tratar la complejidad temporal dentro de la clase R?**

Recordemos que la clase R contiene todos aquellos problemas computables para los cuales se puede construir una máquina de Turing que acepte el lenguaje y siempre pare. El hecho de que siempre pare es importante, no tendría sentido ver la complejidad temporal de un problema dentro de la clase RE, por ejemplo, ya que la máquina construida podría quedarse loopeando. La complejidad computacional temporal trata los lenguajes decidibles.

**b) Resolvimos de dos maneras el problema de los palíndromos, una con una MT de 1 cinta y otra con una MT con varias cintas. La primera tarda O(n2 ) pasos y la segunda O(n). Al igual que para otros problemas que manifiestan este comportamiento, ¿por qué es indistinta la cantidad de cintas utilizadas, considerando la jerarquía temporal que definimos?**

La cantidad de cintas que se utilizan es un factor importante en algunos problemas, ya que como vimos en el ejemplo de los palíndromos, pasar de una cinta a dos cintas permite disminuir notablemente la complejidad temporal. Sin embargo, llega un momento en el que agregar más cintas no genera una mejora notable en el tiempo de ejecución del problema. En particular esto ocurre una vez alcanzado un orden lineal:

*“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.”*

*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(T2 (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*

**c) Probar, utilizando la definición vista en clase, que n³** **= O(2^n).**

Según la notación O, que significa “orden de”: f(n) = O(g(n)) ⟷ (c > 0) (n ∈ N) (f(n) ≤ c.g(n))

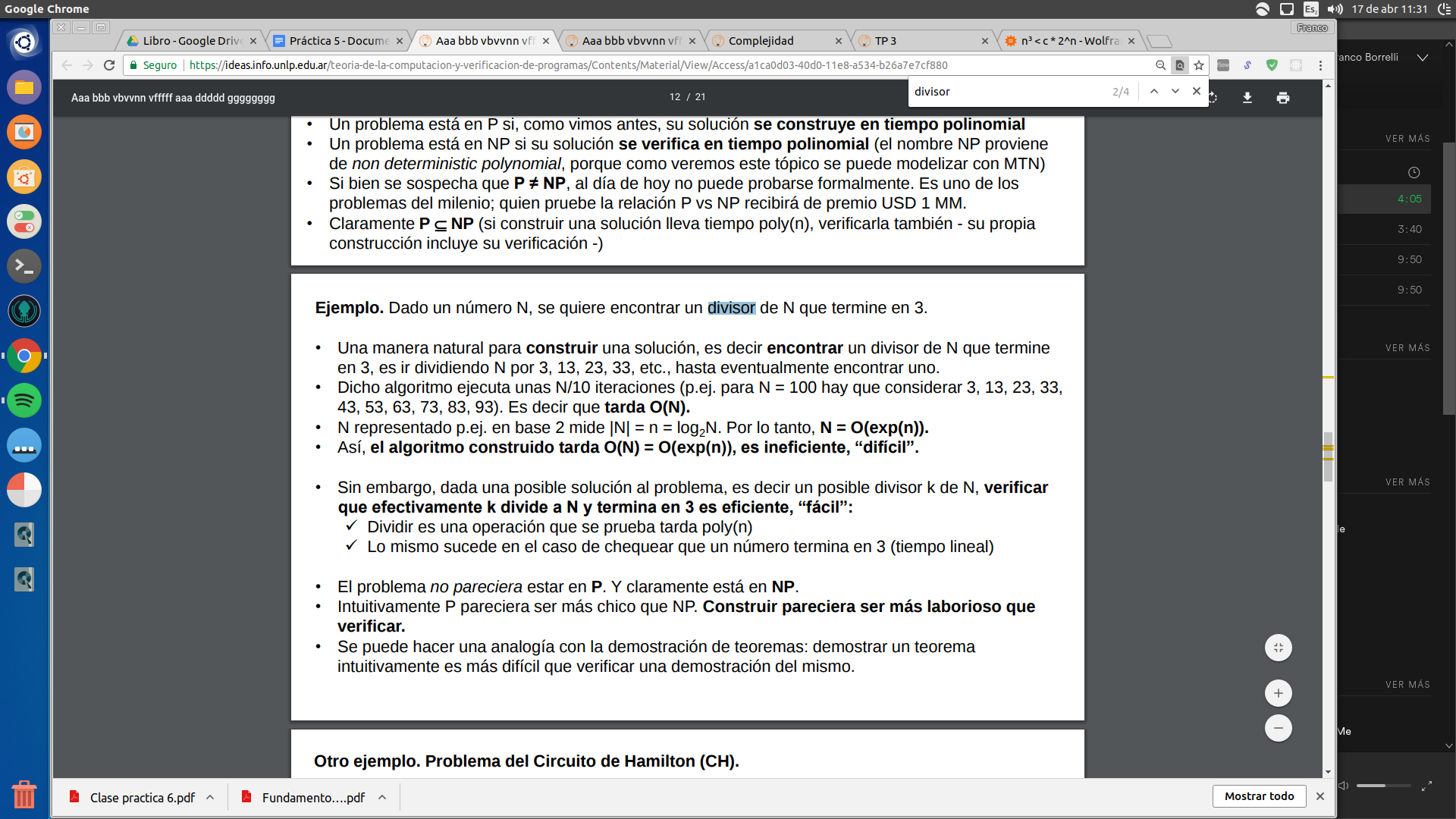
Es decir, f(n) está por debajo de c.g(n), para todo número natural n.

Por lo tanto si f(n) = n³, y g(n)= 2^n, se cumple que f(n) = O(g(n)) si n³ ≤ c . 2^n, lo cual se cumple con c = 4, para todo n.

La notación O, nos permite obtener una cota superior para f(n).

La cátedra no necesariamente pide encontrar la constante c mínima, se podia probar utilizando cualquier c mayor que 4 (en este caso).

**d) Vimos que un algoritmo natural para encontrar un divisor que termine en 3 de un número N tarda O(N) pasos. ¿Esto significa que el problema está en P?**

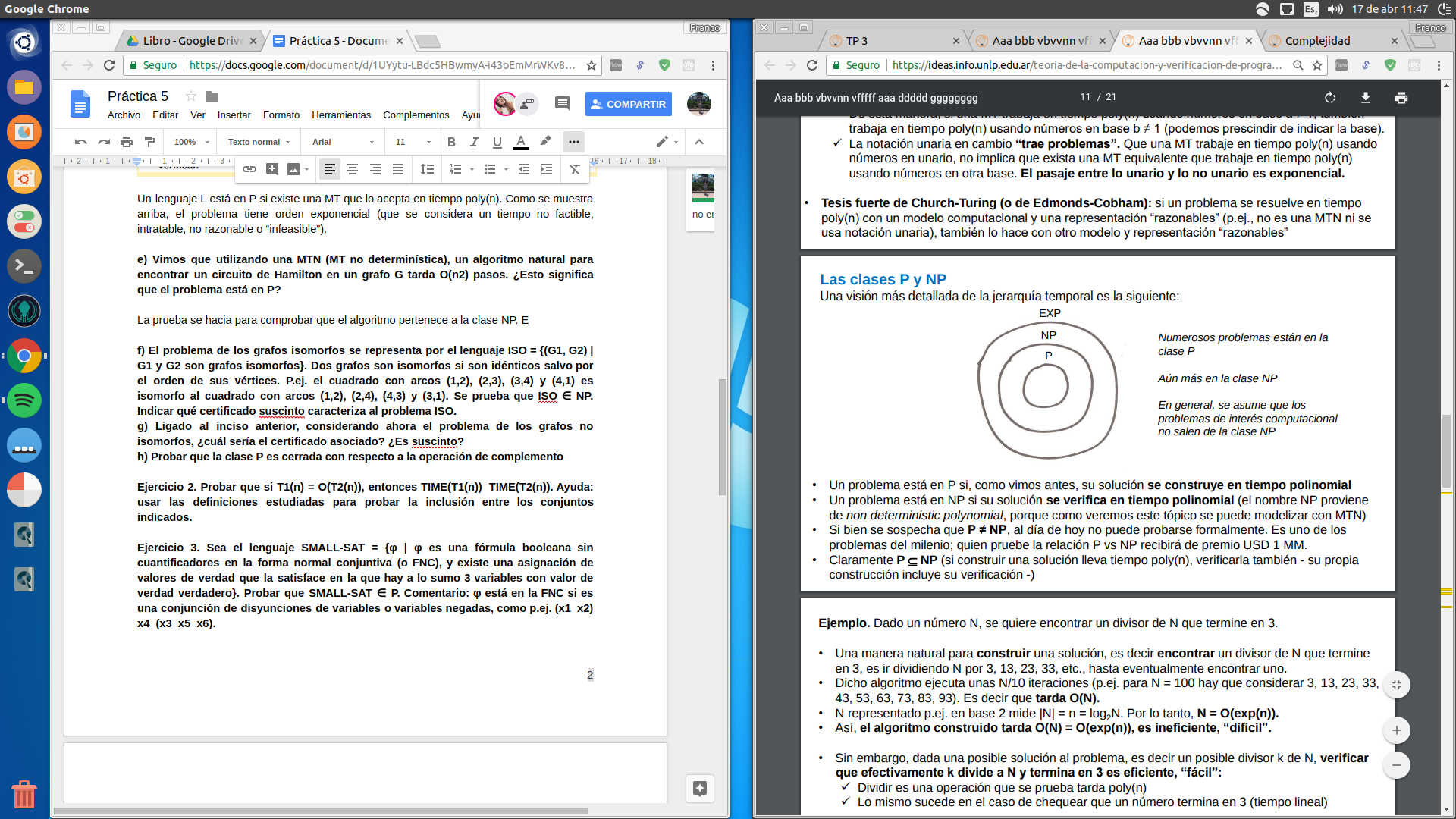


Un lenguaje L está en P si existe una MT que lo acepta en tiempo poly(n). Como se muestra arriba, el problema tiene orden exponencial (que se considera un tiempo no factible, intratable, no razonable o “infeasible”).

N (número introducido como input) => log2 (N) longitud de N. El valor N se presenta con notación log2. Este número es el que en verdad nos interesa ver.

Por lo que n = log2 (N) => N ~= 2^n => De acá se desprende que sea exponencial. Y por tanto no pertenece a P.

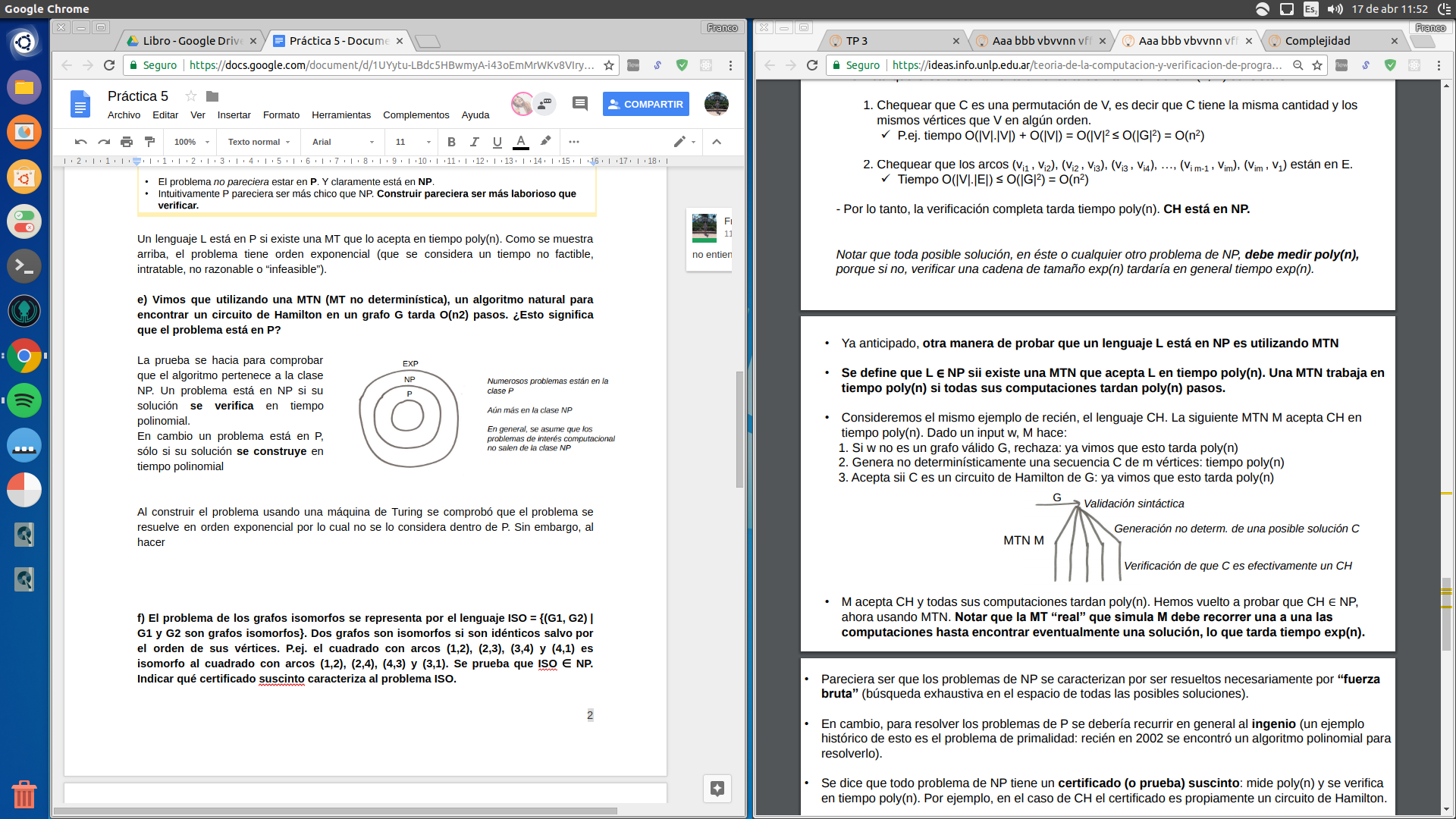
**e) Vimos que utilizando una MTN (MT no determinística), un algoritmo natural para encontrar un circuito de Hamilton en un grafo G tarda O(n2) pasos. ¿Esto significa que el problema está en P?**



La prueba se hacia para comprobar que el algoritmo pertenece a la clase NP. Un problema está en NP si su solución **se verifica** en tiempo polinomial.

En cambio un problema está en P, sólo si su solución **se construye** en tiempo polinomial

Al construir el problema usando una máquina de Turing se comprobó que el problema se resuelve en orden exponencial por lo cual no se lo considera dentro de P. Por lo contrario si se encuentra dentro de NP.



La MTD que simula esa MTN tendría que recorrer cada una de las ‘ramas’ o bien computaciones de MTN para encontrar eventualmente una solución, lo que tarda tiempo exp(n).

**f) El problema de los grafos isomorfos se representa por el lenguaje ISO = {(G1, G2) | G1 y G2 son grafos isomorfos}. Dos grafos son isomorfos si son idénticos salvo por el orden de sus vértices. P.ej. el cuadrado con arcos (1,2), (2,3), (3,4) y (4,1) es isomorfo al cuadrado con arcos (1,2), (2,4), (4,3) y (3,1). Se prueba que ISO ∈ NP. Indicar qué certificado sucinto caracteriza al problema ISO.**

1

2

3

4

1

2

4

3

G1

G2

* sucinto → Que está expresado de manera breve, concisa y precisa. Que tarda polinomia
* certificado → comprobación

|  |
| --- |
| *“Todo lenguaje que está dentro de NP tiene un certificado sucinto”* |

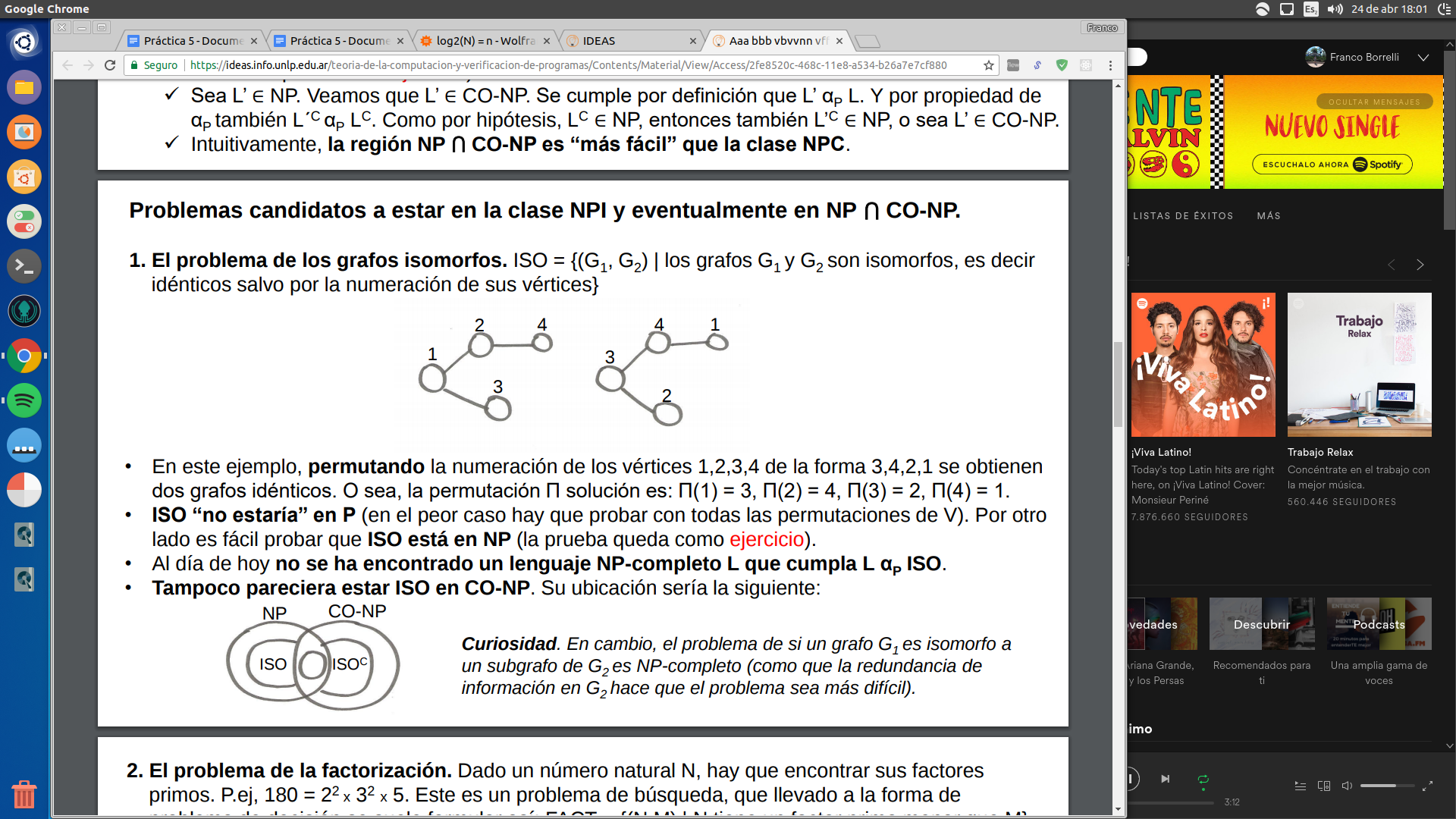
Se dice que todo problema de NP tiene un certificado (o prueba) sucinto: mide poly(n) y se verifica en tiempo poly(n). Por ejemplo, en el caso de CH el certificado es propiamente un circuito de Hamilton.

El certificado sería la comprobación de isomorfismo entre ambos grafos.

Crearía una MTN que ante la entrada G1 y G2, haga permutaciones de los vértices de G1 para ver si ese es isomorfo con G2 . Cada permutación se corresponde con una rama de la MTN. Hacer una permutación tendria un costo de n, siendo n = |V| en G1, y verificar que esa permutación se corresponda con el orden de los vértices en G2 también tarda n. Cada computacion en cada rama de esta MTN tendrá O(n). Con que una rama acepte digo ‘estos grafos G1 y G2 son isomorfos’.

El certificado será encontrar la permutación que hace que se cumpla el isomorfismo en los grafos, y como vimos, al tener ésta un costo polinomial, el certificado es sucinto.

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

****

**g) Ligado al inciso anterior, considerando ahora el problema de los grafos no isomorfos, ¿cuál sería el certificado asociado? ¿Es sucinto?**

Comprobar que dos grafos no son isomorfos entre sí, consistiría en definir cada posible combinación de arcos que harían que G2 sea isomorfo respecto a G1. Cada una de estas debería chequearse con el conjunto de arcos de G2, comprobandose que no son isomorfos entre sí, sólo si ninguna de estas combinaciones formadas coincide. Este problema supone una complejidad más grande, ya que se deben generar cada una de las posibles alternativas. No puede resolverse en tiempo polinomial, por lo tanto no tiene sucinto o certificado.

**h) Probar que la clase P es cerrada con respecto a la operación de complemento**

P es cerrada no únicamente respecto al complemento, sino también respecto a la unión y la intersección.

Prueba:

1. Idea General:

Que L ∈P significa que existe una MT M que acepta el lenguaje en tiempo polinomial y siempre para. La idea es construir una MT Mc que acepte Lc y para siempre en tiempo polinomial. Esta MT simulará la máquina M cambiando sus estados finales, es decir:

* Aceptará (se detendrá en qA) si M rechaza.
* Rechazará (se detendrá en qR) si M acepta.

1. Construcción:

Si M = (Q, Ʃ, Γ, δ, q0 , qA , qR ),

entonces M C = (Q, Ʃ, Γ, δ’, q0 , qA , qR ), tal que δ y δ’ son idénticas salvo que con los estados qA y qR permutados

1. Prueba de correctitud de la construcción:

* M C para siempre:

M para siempre y M C sólo difiere de M en que para en el estado opuesto.

* L(M C) = LC (vamos a probarlo por doble inclusión de conjuntos):

w ∈L(M C) ⟷

con entrada w, ML C para en qA ⟷

con entrada w, ML para en qR ⟷

w ∉ L(M ) ⟷

w ∉ L ⟷

w L C

Como sabemos la MT M acepta L en poly(n), la MT Mc también lo hará (negar los estados se hace de forma lineal).

**Ejercicio 2. Probar que si T1(n) = O(T2(n)), entonces TIME(T1(n)) ⊆ TIME(T2(n)). Ayuda: usar las definiciones estudiadas para probar la inclusión entre los conjuntos indicados.**

Un lenguaje L cumple que L ∈ TIME(T(n)) si y sólo si existe una MT que acepta L en tiempo O(T(n)), es decir que a lo sumo es MT realiza T(n) pasos. 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).

Por definición de Orden, dijimos que T1(n) es de O(T2(n)) si se verifica que T1(n) está siempre por debajo de c . T2(n), para todo n natural. Siendo c un valor > 0.

A partir de esto sabemos que T1(n) siempre tardará lo mismo o menos que T2(n).

**Ejercicio 3. Sea el lenguaje SMALL-SAT = {φ | φ es una fórmula booleana sin cuantificadores en la forma normal conjuntiva (o FNC), y existe una asignación de valores de verdad que la satisface en la que hay a lo sumo 3 variables con valor de verdad verdadero}. Probar que SMALL-SAT ∈ P. Comentario: φ está en la FNC si es una conjunción de disyunciones de variables o variables negadas, como p.ej.**

**(x1 v x2) ^ x4 ^ (¬x3v x5 v x6).**

Para probar que SMALL-SAT ∈ P debería construir una MT que acepte SMALL-SAT en tiempo poly(n). Esta MT aceptaría (para en estado qA) si se encuentra alguna asignación de valores de verdad para la fórmula booleana recibida con a lo sumo 3 variables con valor verdadero.

La MT hará distintas iteraciones, primero probando todas las combinaciones con variables con valores de verdad negativo. Luego todas las que tengan solo una variable con valor verdadero y así sucesivamente hasta probar todas las que tengan 3 variables con este valor. Si luego probar todas estas combinaciones diferentes (que son un conjunto finito) vemos que no acepta ninguna entonces la MT se detendrá en qR. Si por lo contrario, si durante estas iteraciones encuentra una combinación que satisface, entonces se detiene en qA.

Esta máquina, en el peor de los caso deberá hacer 4 iteraciones generando:

1. nC0 = 1 combinación.
2. nC1 = n combinaciones.
3. nC2 combinaciones.
4. nC3 combinaciones.

Siendo n la cantidad total de variables que tiene la fórmula booleana.

La MT tardará tiempo lineal en verificar que la fórmula sea válida ( O(n) ). Suponiendo que esta tenga 3 o más variables generará 1 + n + ½ (n - 1) n + ⅙ (n - 2) ( n - 1) n. Es decir que tardará O(n³).

Por lo cual la MT acepta L en tiempo poly(n).

**Ejercicio 4. El problema del conjunto dominante de un grafo se representa por el lenguaje DOM-SET = {(G, K) | G es un grafo que tiene un conjunto dominante de K vértices}. Un subconjunto de vértices de un grafo G es un conjunto dominante de G, si todo otro vértice de G es adyacente a algún vértice de dicho subconjunto. Probar que DOM-SET ∈ NP. Justificar las respuestas.**

Para probar que DOM-SET pertenece a NP voy a construir una MTN, la cual debe aceptar DOM-SET en tiempo poly(n).

Construcción de la MTN:

1. Validar el input recibido y rechazar si es inválido.
2. Generar cada una de las diferentes secuencias de K vértices. Cada una de estas secuencias se hace en una computación diferente dentro de la MTN.
3. Verificar en cada computación de la MTN si se formó un conjunto dominando. Si alguna computación satisface entonces la MTN acepta, sino rechaza.

Además se debe comprobar que la MTN acepta en tiempo poly(n):

* Validar un grafo como vimos en la teoría, se hace en tiempo poly(n).
* Generar cada secuencia de K vértices se hace también en poly(n). Se harán |V| tomadas de K computaciones diferentes.
* El paso de verificar una secuencia se hace en tiempo O(n²).

Por lo tanto se comprueba que DOM-SET ∈ NP

**¿Se cumple que DOM-SET ∈ P?**

No, la construcción de esta MTN solo permite probar que DOM-SET ∈ NP. Se puede llegar a suponer que DOM-SET no pertenece a P dado que simular esta MTN requerirá un tiempo exponencial.

**¿Se cumple que DOM-SETC ∈ NP?**

No, porque debería pasar por cada una de las posibles combinaciones posibles para probar que ninguna de ellas forma un conjunto dominante. Que se hace en tiempo exponencial.

Ejercicio 5. Sea M una MTN que si acepta una cadena w, al menos en una de sus  
computaciones lo hace en tiempo polinomial con respecto a |w|, digamos p(|w|). Probar que  
L(M) ∈ NP. Ayuda: Se sabe que toda función polinomial p(n) es tiempo-construible, es decir quese computa en tiempo p(n).