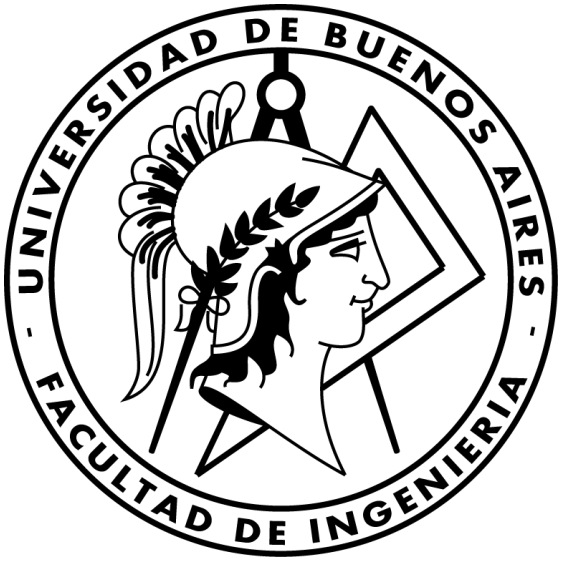


Universidad de Buenos Aires
Facultad de Ingeniería
Año 2018 - 1º Cuatrimestre

Teoría de Algoritmos 1 - Trabajo Práctico 2

NoName_4

Integrantes

Nombre	Padrón
Cabrera María Florencia	97864
Centurión Martín Leonardo	97874
Marino Gonzalo	97794
Prado María Florencia	96626

Parte 1: Spy Vs SpyEjercicio 1

1. Dado un mapa de una ciudad, las ubicaciones de los espías y el aeropuerto determine quién se quedará con los informes.
Se adjunta por mail los archivos correspondientes. Para ejecutar este problema por consola hay que poner:

```
# python problema1.py lineaEspia1 lineaEspia2 lineaAeropuerto
```

Observación: consideramos que si ambos espías llegan al mismo tiempo, quien se queda con los informes es el espía 2.

2. Repita el procedimiento pero introduciendo costos en los caminos.
Se adjunta por mail los archivos correspondientes. Para ejecutar este problema por consola hay que poner:

```
# python problema2.py lineaEspia1 lineaEspia2 lineaAeropuerto
```

Observación: consideramos que si ambos espías llegan al mismo tiempo, quien se queda con los informes es el espía 2.

3. Analice la complejidad añadida si se solicita retornar el camino realizado por cada espía en los pasos 1 y 2.
El algoritmo planteado para este problema, no varía demasiado al propuesto en los problemas 1 y 2. Esto se debe a que el querer obtener el camino que recorre cada uno de los espías es del orden $\Theta(2E)$ (E siendo la cantidad de aristas) ya que sólo se debe guardar en una estructura auxiliar, como una lista, el recorrido que debe hacer cada uno de los espías.
En el caso del grafo sin peso, el orden del problema 1 es de $\Theta(E + V)$, pero al agregar lo pedido, el orden nos queda $\Theta(3E + V)$. En dicho orden se puede redondear quitando las constantes, quedando $\Theta(E + V)$.
Para el caso del grafo con peso, el problema 2, es de orden $\Theta(E + V log V)$, pero agregando el orden que se genera para guardar el recorrido queda del orden $\Theta(3E + V log V)$, redondeamos al igual que lo hicimos anteriormente y vemos que es de orden $\Theta(E + V log V)$.

4. Realiza los pasos 1 y 2 nuevamente retornando como salida de ejecución los caminos realizados por cada espía.
Se adjunta por mail los archivos correspondientes. Para ejecutar este problema por consola hay que poner:

```
# python problema4a.py lineaEspia1 lineaEspia2 lineaAeropuerto
# python problema4b.py lineaEspia1 lineaEspia2 lineaAeropuerto
```

Observación: consideramos que si ambos espías llegan al mismo tiempo, quien se queda con los informes es el espía 2.

Parte 2

Ejercicio 1

Solución por fuerza bruta

El algoritmo desarrollado para determinar si dada dos cadenas de texto S1 y S2 de longitud n, la segunda es una rotación cíclica de la primera, se implementó realizando rotación de un caracter y evaluando si son iguales.

Pseudocódigo:

```
esRotacionCiclica(S1,S2):
1     sea n = largo de ambas cadenas de texto
2     Para i de 0:n :
3           sea j = (n - i)
4           Si S1 es igual a la concatenación de S2[j:n] + S2[0:j]:
5                 return True
6     return False
```

Ejemplo:

```
esRotacionCiclica('ABRACADABRA', 'DABRAABRACA')

[D][A][B][R][A][A][B][R][A][C][A]
 0  1  2  3  4  5  6  7  8  9  10

Ejecución:
n = 11
i=0; j=11
ABRACADABRA==''+ 'DABRAABRACA' -> False
i=1; j=10
ABRACADABRA=='A'+ 'DABRAABRAC' -> False
i=2; j=9
ABRACADABRA=='CA' + 'DABRAABRA' -> False
i=3; j=8
ABRACADABRA=='ACA' + 'DABRAABR' -> False
i=4; j=7
ABRACADABRA=='RACA' + 'DABRAAB' -> False
i=5; j=6
ABRACADABRA=='BRACA' + 'DABRAA' -> False
i=6; j=5
ABRACADABRA=='ABRACA' + 'DABRA' -> True

esRotacionCiclica('ABRACADABRA', 'DABRAABRACA')-> True
```

Análisis de complejidad

La ejecución de la línea 4 tiene complejidad $\Theta(N)$, esto es porque para determinar si dos cadenas son iguales se debe ver para cada posición si comparten mismos caracteres, en el peor caso se van a realizar N comparaciones de $\Theta(1)$, esto pasa si comparten al menos $N-1$ caracteres en misma posición, es decir, se va a llegar hasta el n-ésimo caracter hasta afirmar la igualdad o desigualdad.

Ejemplo peor caso de comparación de dos cadenas:

```
Pseudocódigo:

sea n=largo de S1 y S2
Para i de 0:n
      Si S1[i]!=S2[i]:
            return False
return True

Ejecución:
Sea S1='ALGORITMO', S2='ALGORITMO', n=9
i=0 -> 'A'=='A'
i=1 -> 'L'=='L'
…
i=8 -> 'O'=='O'
```

Por otro lado en el peor caso el for principal se va a ejecutar N veces, esto sucede si S2 está rotado una vez a derecha con respecto a S1, por ejemplo S1=ALGORITMO Y S2=OALGORITM, o si no es una rotación cíclica.

Concluimos en que la complejidad de este algoritmo es $\Theta(N^2)$.

KMP - Knuth, Morris, Pratt

Es un algoritmo para "string-matching", busca la existencia de una subcadena, a la que llamaremos patrón, dentro de una cadena. Utiliza información del patrón a buscar y de los fallos previos para determinar en qué parte de la cadena seguir buscando, sin tener que examinar caracteres más de una vez.

Tabla de fallos

Primeramente se calcula la tabla de fallos. Está tabla es la que pretende evitar que cada caracter de la cadena sea examinado más de una vez. Para lograrlo se debe haber comparado alguna parte de la cadena con alguna parte del patrón, lo que nos va a decir las potenciales posiciones en donde podemos encontrar una nueva coincidencia sobre la parte de la cadena ya analizada.

Ejemplificado sería, supongamos que el patrón a buscar es "ABRACADABRA" y la cadena es "EL MAGO EJECUTÓ ABRACADAPA EN VEZ DE ABRACADABRA" cuando llegamos a la "P", posición 22 de la cadena, tenemos una falla. Lo que queremos ahora es buscar en dónde se encuentra la primera letra del patrón nuevamente, si existe, y hasta dónde logra repetirse. Si vemos nuestro patrón la letra "A", primera letra de "ABRACADABRA" se vuelve a repetir en la posición 4, antes de la "C", por lo tanto en la tabla de fallos vamos a poner un 1 en la posición siguiente a 4.
La tabla de fallos indica la el mayor shift, corrimiento, posible usando las comparaciones previamente realizadas. O bien, dicho de otro modo, tendrá la distancia que existe desde un punto del patrón hasta la última ocurrencia de la primer letra, si contar la primera aparición. Y mientras sigan coincidiendo se marca la distancia, cuando deja de coincidir se marca con 0 y se vuelve a buscar la aparición de la primera letra del patrón.

El primer y segundo valor de la tabla son siempre -1 y 0 respectivamente, la función de fallo empieza examinando desde la tercera letra del patrón. Esto es así porque sí para la segunda letra se marca 1, no se va a saltar nunca porque se volvería a ese punto. Y el primero es -1 por necesidad del algoritmo, para no regresar más atrás.

Veamos como quedaría la tabla para el patrón usado en el ejemplo:

```
0 1 2 3 4 5 6 7 8 9 10
 A B R A C A D A B R A
-1,0,0,0,1
La primera A aparece en la posición 3, por lo que en la siguiente
letra, la C, en la posición 4, está a distancia 1 de dicha
aparición, es por eso que se coloca un 1.

 0 1 2 3 4 5 6 7 8 9 10
 A B R A C A D A B R A
-1,0,0,0,1,0,1
Luego no hay coincidencia con la segunda letra del patrón y
nuevamente, aparece una A en la posición 5, por eso la posición 6
lleva un 1.

 0 1 2 3 4 5 6 7 8 9 10
 A B R A C A D A B R A
-1,0,0,0,1,0,1,0,1
Mismo caso que en el anterior

 0 1 2 3 4 5 6 7 8 9 10
 A B R A C A D A B R A
-1,0,0,0,1,0,1,0,1,2,3

Como las dos siguientes letras coinciden, entonces se marca la
distancia, se suma de a uno.

Pseudocódigo para cálculo de la tabla de fallos
```

```
Algoritmo tablaKMP:
      Entrada:
            un array de caracteres, P (el patrón que va a ser analizada)
            un array de enteros, TF (la tabla de fallo a rellenar) debe
      tener el mismo tamaño que P.
      Salida:
            devuelve la tabla rellenada TF
      Variables que se usan:
            entero i ← 2 (posición actual donde se está calculando TF)
            entero j ← 0 (el índice en P del siguiente carácter del actual
      candidato en la subcadena)

      asignar TF[0] ← -1, TF[1] ← 0

      Hacer mientras i sea menor o igual que el tamaño de P:
            (caso 1º: siguiente candidato coincidente en la cadena)
            Si P[i - 1] = P[j] entonces
                  Asignar j ← j + 1, TF[i] ← j, i ← i + 1
            Pero si j > 0 entonces
                        Asignar cnd ← TF[j]
                  (caso 3º: no se halló candidatos coincidentes (otra
      vez))
            Y si no entonces
                Asignar TF[i] ← 0, i ← i + 1
            Fin si
      Repetir
```

Algoritmo KMP

Una vez que sabemos calcular la tabla de fallos podemos pasar a entender el algoritmo KMP. Será explicado siguiendo un ejemplo para comprender correctamente cada paso.

Sea P el patrón a encontrar en la cadena S, tenemos los punteros i y k, i para marcar la posición en P y k en S. Por otro lado tenemos la tabla de fallos que va a ir denotando los incrementos de k.

```
Inicialmente i=0; k=0

S := 'ABRACADAPA  ABRACADABRA'
P := 'ABRACADABRA'
```

Como vimos en la explicación del armado de la tabla de fallos, para el patrón 'ABRACADABRA' :

```
TF:= -1,0,0,0,1,0,1,0,1,2,3
```

Por lo tanto:

```
k :=  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
S := 'A B R A C A D A P A    A  B  R  A  C  A  D  A  B  R  A'
P := 'A B R A C A D A B R A'
i :=  0 1 2 3 4 5 6 7 8 9 10
TF:= -1,0,0,0,1,0,1,0,1,2,3

0 Mientras k+i es menor que tamaño de S:
1     Si P[i] == S[k+i] ; Esto inicialmente es 'A' == 'A'
```

2 **Si** i == (tamaño de P) - 1 => Se encontró P en S
3 **Si no** se incrementa i ; i -> i+1

Esto último sucede hasta que **i=8**, siendo S[8]='P' y P[8]='B'.
(En ese momento **k=0**)
Dado que la igualdad de la línea 1 no se cumple se debe actualizar el valor de k:

4 **Si no** P[i] == S[k+i]:
5 k <- k + i - TF[i];
En esta instancia k = 0 + 8 - 1; k=7

Verificamos el valor de TF[i] para ver si actualizar el valor de i

6 **Si** TF[i] > -1 ; TF[8]=1
7 i <- TF[i]; En este caso **i = 1**

```
k :=  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
S := 'A B R A C A D A P A    A  B  R  A  C  A  D  A  B  R  A'
P :=               'A B R A  C  A  D  A  B  R  A'
i :=                0 1 2 3  4  5  6  7  8  9  10
TF:=               -1,0,0,0, 1, 0, 1, 0, 1, 2, 3
```

Se vuelve a analizar la línea 1, en este caso: P[1] == S[7+1]
Como 'B' no es igual a 'P', se actualiza nuevamente k.
línea 5: k = 7 + 1 - 0; k=8
línea 6: TF[1] = 0 > -1
línea 7: i=0

```
k :=  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
S := 'A B R A C A D A P A    A  B  R  A  C  A  D  A  B  R  A'
P :=                 'A B R  A  C  A  D  A  B  R  A'
i :=                  0 1 2  3  4  5  6  7  8  9  10
TF:=                 -1,0,0, 0, 1, 0, 1, 0, 1, 2, 3
```

línea 1: P[0] == S[8+0]
Como 'A' no es igual a 'P', se actualiza nuevamente k.
línea 5: k = 8 + 0 - (-1); k=9
línea 6: TF[0] = -1, no pasa a la línea 7

```
k :=  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
S := 'A B R A C A D A P A    A  B  R  A  C  A  D  A  B  R  A'
P :=                 'A B R  A  C  A  D  A  B  R  A'
i :=                  0 1 2  3  4  5  6  7  8  9  10
TF:=                 -1,0,0, 0, 1, 0, 1, 0, 1, 2, 3
```

línea 1: P[0] == S[9+0]

Como 'A' es igual a 'A' y 0 no es ((tamaño de P) -1), se incrementa i

```
k :=  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
S := 'A B R A C A D A P A    A  B  R  A  C  A  D  A  B  R  A'
P :=                   'A B  R  A  C  A  D  A  B  R  A'
i :=                    0 1  2  3  4  5  6  7  8  9  10
TF:=                   -1,0, 0, 0, 1, 0, 1, 0, 1, 2, 3
```

línea 1: P[1] == S[9+1]
Como 'B' no es igual a ' ', se actualiza nuevamente k.
línea 5: k = 9 + 1 - 0
línea 6: TF[1] = 0
línea 7: i=0

```
k :=  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
S := 'A B R A C A D A P A    A  B  R  A  C  A  D  A  B  R  A'
P :=                     'A  B  R  A  C  A  D  A  B  R  A'
i :=                      0  1  2  3  4  5  6  7  8  9  10
TF:=                     -1, 0, 0, 0, 1, 0, 1, 0, 1, 2, 3
```

línea 1: P[0] == S[10+0]
Como 'A' no es igual a ' ', se actualiza nuevamente k.
línea 5: k = 10 + 0 - (-1); k=11
línea 6: TF[0] = -1, no pasa a la línea 7

```
k :=  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
S := 'A B R A C A D A P A    A  B  R  A  C  A  D  A  B  R  A'
P :=                       'A  B  R  A  C  A  D  A  B  R  A'
i :=                        0  1  2  3  4  5  6  7  8  9  10
TF:=                       -1, 0, 0, 0, 1, 0, 1, 0, 1, 2, 3
```

línea 1: P[0] == S[11+0]
Como 'A' es igual a 'A' y 0 no es (tamaño de P -1), se incrementa i
línea 6: TF[0] = -1, no pasa a la línea 7.
i se va a incrementar hasta i=10 con k=11.

```
k :=  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
S := 'A B R A C A D A P A    A  B  R  A  C  A  D  A  B  R  A'
P :=                       'A  B  R  A  C  A  D  A  B  R  A'
i :=                        0  1  2  3  4  5  6  7  8  9  10
TF:=                       -1, 0, 0, 0, 1, 0, 1, 0, 1, 2, 3
```

línea 0: (10 + 11) es menor a 22
línea 1: P[10] == S[21], 'A'=='A'
línea 2: 10 == (11-1) => Se encontró P en S.

Análisis de complejidad

El tiempo de ejecución es proporcional al tiempo necesario para leer los caracteres en la cadena y en el patrón. En otras palabras el peor caso de tiempo de ejecución es, sea N la longitud de la cadena y M la del patrón $\Theta(N + M)$, con $M \leq N$.

Mirando la tabla de fallos se puede deducir cuáles serán los mejores y peores casos. Cuantos más ceros, mejor será nuestro caso, no se repiten caracteres desde el principio del patrón, por lo que tendremos saltos más grandes. Siendo el peor caso cuando el patrón se compone de un único caracter, lo que nos da una tabla de fallos igual a $[-1][0][1][2]...[M-3]$, mientras que para el mejor caso, un patrón con todos caracteres distintos tiene como resultante $[-1][0][0][0]...[0]$, en este último los saltos serán tan grandes como el patrón, M.

Solución utilizando KMP

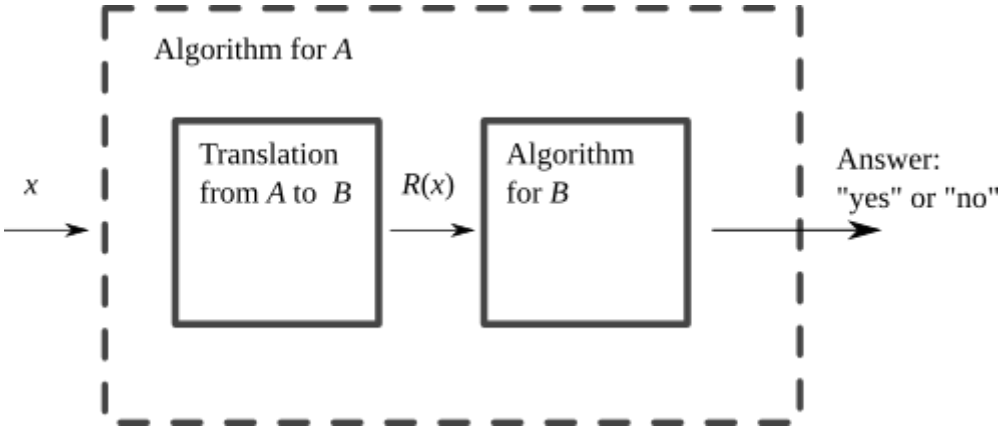
Veremos cómo modificar el algoritmo por fuerza bruta para que utilice KMP. Convertiremos el problema de ver si es una rotación cíclica en uno de "string matching", problema que resuelve el algoritmo KMP.

Se seguirá realizando rotación de un caracter pero, a través de KMP, se determinará si hay una igualdad, lo que se traduce en decir que hay un match.

Pseudocódigo:

```
esRotacionCiclica(S1,S2):
1     sea n = largo de ambas cadenas de texto
2     Para i de 0:n :
3           sea j = (n - i)
4           Si (KMP(S1, S2[j:n] + S2[0:j]):
5                 return True
6     return False
```

Estamos ante una reducción, podemos ver como en la línea 4 se realiza la transformación del problema inicial, se convierte la determinación de si es una rotación cíclica en encontrar una subcadena dentro de otra. Tras resolver este problema hay una transformación de la solución para convertirla en solución del problema inicial. Esto último es, sí se encontró el patrón en la cadena entonces es una rotación cíclica.



En esta ilustración sacada de un texto de Computational Complexity por C.H. Papadimitriou, vemos que A sería el problema de determinar si una cadena S2 es rotación cíclica de otra S1. Seguidamente tenemos la transformación del problema A en B, siendo B encontrar una subcadena dentro de otra.

Análisis de complejidad

Al igual que en la solución por fuerza bruta, en el peor caso el for principal se va a ejecutar N veces y la ejecución del algoritmo KMP, como explicamos anteriormente, tiene un costo de $\Theta(N)$, resultando en que la complejidad de esta solución es $\Theta(N^2)$.

Solución con única ejecución KMP

Como entrada al algoritmo se esperan dos cadenas de tamaño n, S1 y S2. Se tomará como patrón S2 , ya que se quiere ver si S2 es una rotación cíclica de S1.
Transformamos el problema de analizar si es rotación cíclica en ver si el patrón se encuentra en la concatenación de S1 con S1. Dado a que si existe la rotación cíclica deberá aparecer S2 en dicha concatenación.

Por ejemplo:
Sea S1 = 'ABRACADABRA' y S2='DABRAABRACA'
S1+S1='ABRACADABRAABRACADABRA'
Vemos cómo S2 se encuentra en la concatenación.

Entonces con esto logramos una única ejecución del algoritmo KMP teniendo como patrón a S2 y como cadena S a S1+S1.

Pseudocódigo:

```
esRotacionCiclicaKMP(S1,S2):
1     sea S = S1.concat(S1)
2     return (KMP(S,S2) > 0)
```

Análisis de complejidad

La ejecución del algoritmo es de orden $\Theta(N)$ ya que como se dijo anteriormente KMP está en ese orden y podemos pensar en un algoritmo concat del mismo orden.

Ejercicio 2

1. Hallar un ciclo de mayores pesos. Demostrar que el problema es difícil (NP-Completo o NP-Hard)

Problema a resolver: dado un grafo no dirigido y con pesos, G(E,V), hallar el ciclo Hamiltoniano tal que la suma de sus pesos sea máxima.
Problema conocido: Problema del viajante.

Proponemos que el problema a resolver es difícil , en particular que es NP-Completo.
Demostración:
El grafo G que tenemos se corresponde a las ciudades a visitar como vértices, y las aristas unen a las ciudades, las mismas tienen como peso a la cantidad de millas que separan una ciudad con otra. Este grafo se supone completo y no dirigido.
Proponemos la siguiente reducción de orden lineal con respecto a la cantidad de vértices. Cambiamos los pesos de las aristas. Invertimos los pesos, haciendo que las aristas que antes tenían mayor peso, ahora tengan el mínimo. Si el peso de una arista es P, ahora será P'=1/(P+1) ; a P se le suma 1 advirtiendo que P sea igual a cero.
Ya tenemos el grafo G' para que pueda ser resuelto por el algoritmo del problema del viajante. El algoritmo nos devolverá el ciclo correspondiente, este es el ciclo de mayor peso de grafo G original.
Dado esto se puede afirmar que el problema resulta NP-Hard.
Aemás dado un ciclo, un peso, y un grafo se puede, aplicando la reducción anterior, determinar en tiempo polinómico si el ciclo dado es válido y si se corresponde con el peso propuesto. Se resuelve así la variante de decisión del problema propuesto en tiempo polinomial; luego se concluye que el problema pertenece a NP
Se puede decir entonces, que el problema es NP-Completo.

Ejercicio 3

1. Dar un algoritmo eficiente (pseudocódigo de alto nivel) que resuelva el problema.

Pseudocódigo:

```
def cantidad_aulas(C):
    #C = conjunto de cursos
    Si(C==Ø):
       return 0
    I = [ ] #cursos incompatibles con el curso i
    mientras C!=Ø :
       i = curso en C con tiempo de finalización mínimo
       I.append(todos los cursos incompatibles con i)
       Borrar de C todos los cursos incompatibles con i
```

```
    #Notar que cada vez que se ejecute cantidad_aulas, si C!=Ø, se
necesitará un aula para acomodar todos los cursos compatibles en
horario
    return 1 + cantidad_aulas(I)
```

2. Reducir el problema a una instancia de coloreo de grafos, en su versión de optimización (minimizar la cantidad de colores).

Problema a resolver: cantidad mínima de aulas
Problema conocido: coloreo de grafos
Se puede hacer un grafo tal que sus vértices sean los cursos que hay, y están unidos entre ellos a través de una arista si estos son incompatibles en horarios. Coloreamos este grafo guardando la cantidad de colores que se utilizó para colorearlo. Finalmente, la cantidad de colores utilizados es la cantidad mínima de aulas que se necesitan..

3. A partir de los dos puntos anteriores, ¿podemos asegurar que P = NP? ¿Por qué?
   Si A $\leqslant$B se puede decir lo siguiente:

- si B $\in$ P entonces A $\in$ P
- si A $\in$ NP-Completo entonces B$\in$ NP-Hard

   En el punto 2 se hizo A $\leqslant$B con A= organización de cursos, B=coloreo del grafo, y A$\in$ P entonces lo único que se puede decir es que B es tán o más difícil que A .