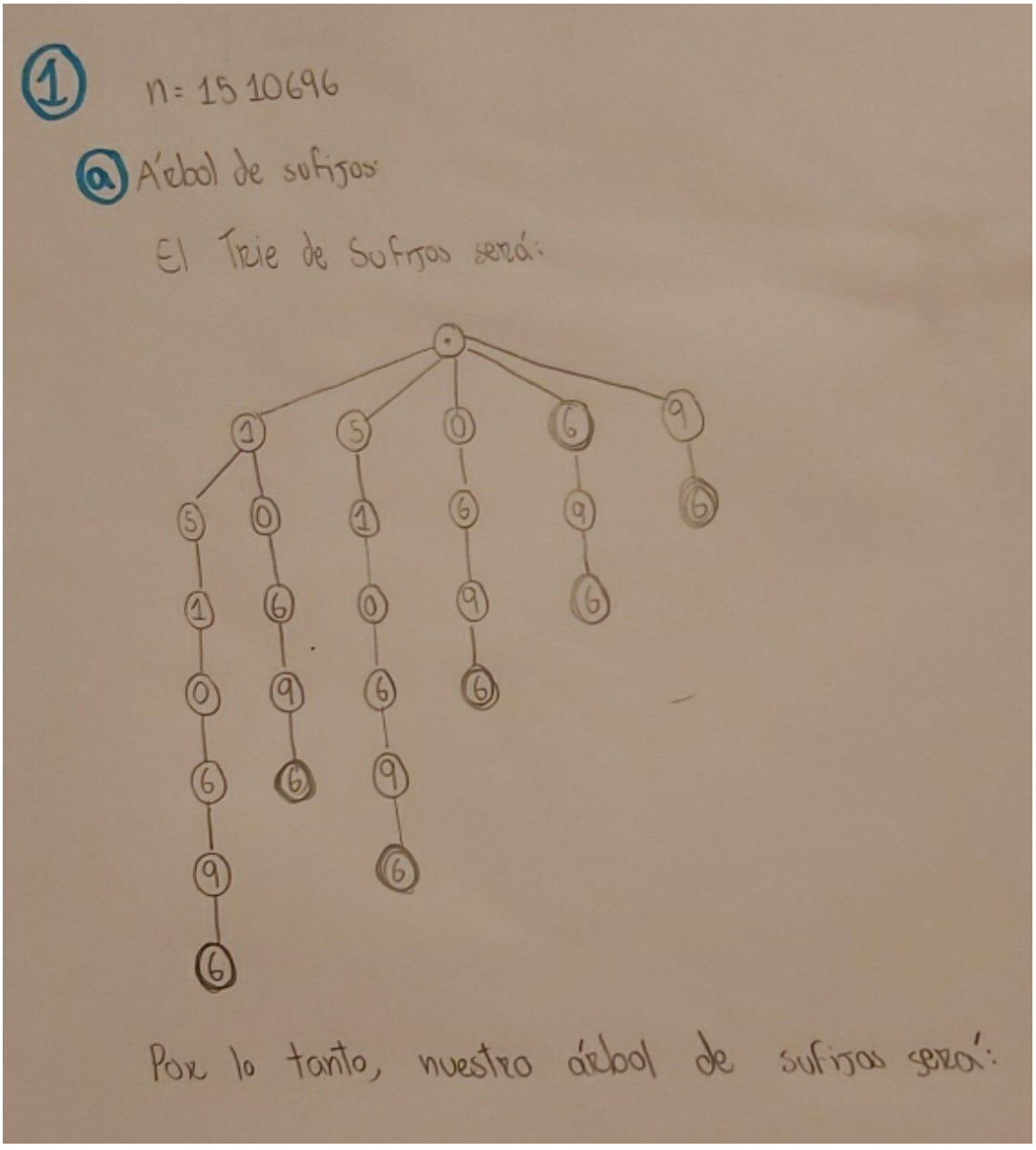
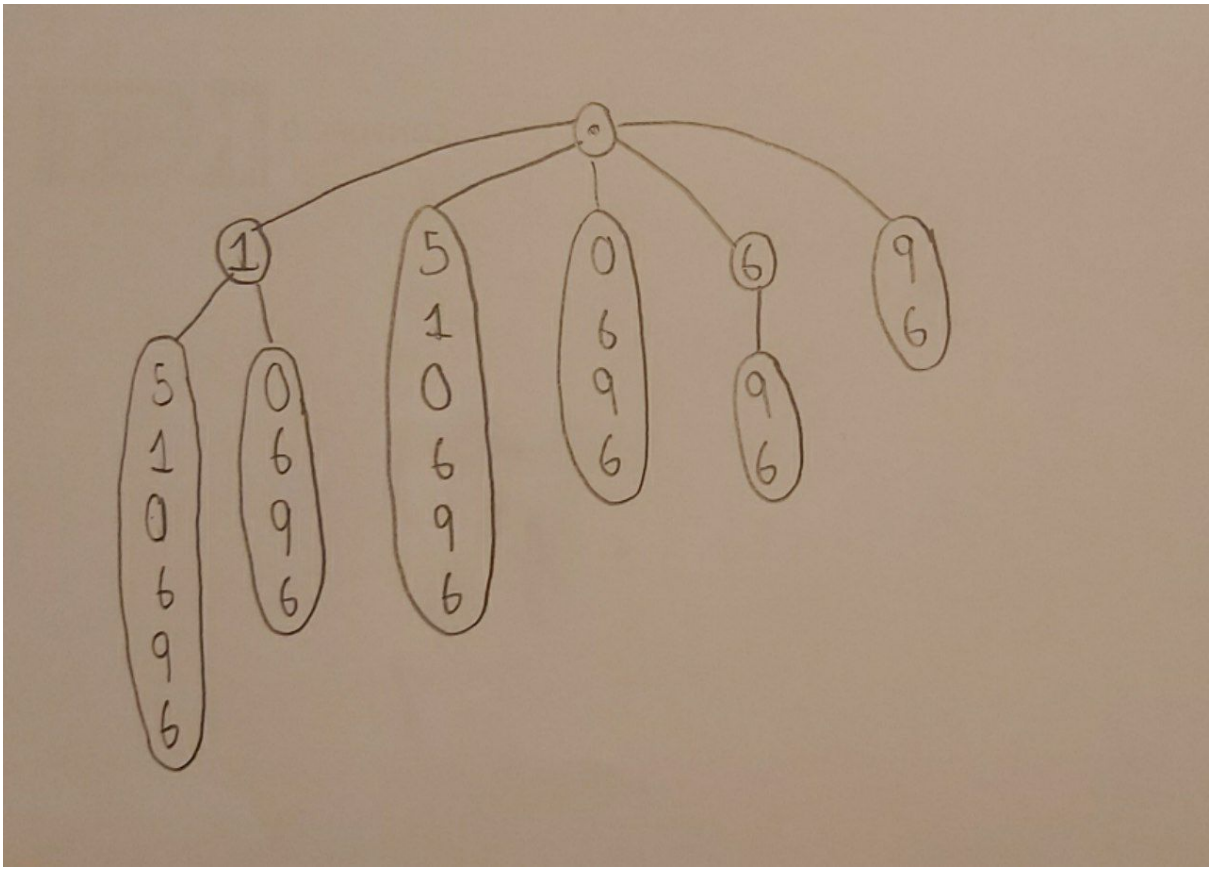
Viernes, 26 de Junio del 2020
CI5651 – Diseño de Algoritmos I
Maria Gabriela Hernandez
15-10696

Examen 2

Pregunta 1:

1. Sea n una cadena de caracteres que tiene su número de carné (sin el guión)
   (a) Construya un árbol de sufijos para n
   (b) Construya un arreglo de sufijos a partir del árbol de sufijos para n





b)

Indice	Sufijo
0	1510696
1	510696
2	10696
3	0696
4	696
5	96
6	6
7	

⇒

Indice	Sufijo
7	
3	0696
2	10696
0	1510696
1	510696
6	6
4	696
5	96

Por lo tanto, el arreglo de Sufijos es:

[7, 3, 2, 0, 1, 6, 4, 5]

Pregunta 2:

2. Finalmente has cedido a la presión social y te propones ver todas las películas del universo cinematográfico de Marvel. ¡Pero no sabes en qué orden debes verlas! Tus amigos te recomiendan ver algunas películas antes que otras, para que puedas entender bien la trama, ya que cada película depende directamente de algunas de las películas anteriores. ¿En qué orden deberías ver las películas de tal forma que se respeten sus recomendaciones?

   Formalmente, dado un conjunto de n pares (p, q), representando el hecho de que la película p debe verse antes que la película q, diseñe un algoritmo que devuelva una secuencia $P = \langle p_1, p_2, \cdots, p_n \rangle$ de películas que represente el orden en que deben verse las películas de tal forma que no aparezca primero la película q que la película p

Podemos modelar el conjunto de n pares (p,q) como un grafo dirigido, en el cual el nodo p apunta al nodo q. Y luego, podemos construir para nuestro problema un orden topológico

```
global topo: pila

función topológico( G(N,C): grafo ) -> secuencia
    vis <- mapa de N a booleano con valor por defecto { i -> false }
    pila <- pila vacía
    para cada u en N
        si ¬vis[u]
            topológicoRec(G,u,vis)
    retornar desempilarTodosLosElementos(topo)

función topológicoRec( G(N,C): grafo, u: N, vis: mapa de N a booleano)
                                                                -> secuencia
    vis[u] <- cierto
    para v en N tal que {u,v} en C
        si ¬vis[v]
            topológicoRec(G,v,vis)
    empilar(topo,u)

función desempilarTodosLosElementos( pila:pila ) -> pila
    pilaARetornar <- pila vacía
    para i desde 1 hasta |pila|
        empilar(pilaARetornar, desempilar(pila))
    retornar pilaARetornar
```

Este pseudocódigo es muy parecido a un DFS. la diferencia es que utilizaremos una pila llamada topo, e iremos empilando nodos progresivamente en la función topológicoRec.

Para obtener el orden topológico, queremos los elementos en orden inverso a como se insertaron. Por lo tanto, retornaremos todos los nodos desempilados, usando la función desempilarTodosLosElementos.

Pregunta 3:

3. Sea $A = (N, C)$ un árbol (notemos que $|C| = |N| - 1$) y una función de costo $c : C \to \mathbb{N}$. Queremos responder consultas de la forma $costoTotal(x, y)$, para $x, y \in N$, que tenga la suma de los costos en el camino entre los nodos x e y.

   Diseñe un algoritmo que pueda responder Q consultas de esta forma en $O(|N| + Q \log |N|)$, usando memoria adicional $O(|N|)$

   Pista: Realice un precondicionamiento adecuado en $O(|N|)$, que le permita responder cada consulta en $O(\log |N|)$.

Podemos notar que como |C| = |N| - 1, entonces A es un árbol enraizado y no posee ciclos. Por lo tanto, solamente existirá una manera de llegar de un nodo a otro nodo.

Para obtener el camino de ir de un nodo a otro, debemos pasar por su ancestro en común más bajo.

Por lo tanto, realizaremos un recorrido del árbol en pre-order y asignaremos valores a cuatro arreglos:

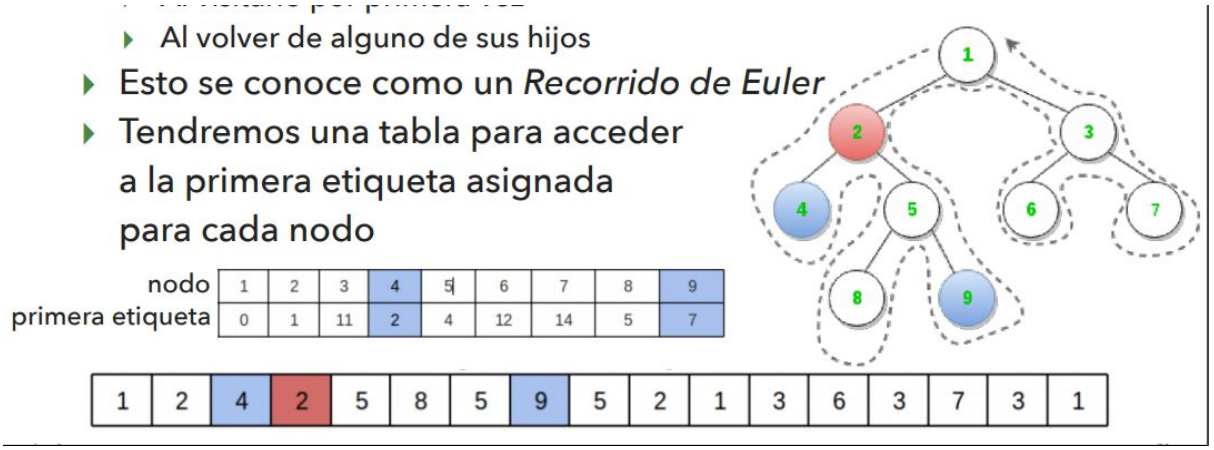
1. Un arreglo llamado ‘’nodo’’ que usa un etiquetado por niveles, que consume memoria adicional O(n).
2. Un arreglo llamado ‘’primera etiqueta’’ el cual nos servirá para acceder a la primera etiqueta asignada para cada nodo., que consume memoria adicional O(n)
3. Una arreglo para almacenar el Recorrido de Euler, que consume memoria adicional O(2n -1), es decir, O(n).
4. Un arreglo llamado ‘’distancia’’ que nos da la distancia de un nodo hasta su papá. El nodo raíz posee distancia 0.

Estos pasos se pueden realizar en tiempo O(n)

Usamos la tabla de ocurrencias para conseguir la primera ocurrencia entre ambos nodos. Luego, la distancia entre cada nodo la calcularemos de la siguiente manera:

1. El elemento mínimo entre el arreglo [i .. j] en donde i es el nodo 1 y j es el nodo 2, es el ancestro más bajo. Por lo tanto, nunca haremos nada con su valor en el arreglo ‘’distancia’’
2. Sumaremos la distancia de cada uno de los nodos hasta su papá usando el arreglo llamado ‘’distancia’’. Si uno de los dos nodos (o ambos) es su mismo ancestro más bajo, no se realiza esta suma.
3. Sumaremos todas las distancias en el arreglo [i + 1 .. j - 1] a menos que se repita un valor z de la forma: [i+1, … , z , … , z … j+1]. Los valores entre z + 1 (de su primera instancia) y z no se sumarán.

El punto 3 sucede porque no se necesita pasar entre z y z para poder ir de i a j. Podemos ver un ejemplo gráfico con:



Supongamos que queremos ir de 4 a 9, es decir, queremos costoTotal(4,9). Por lo tanto, eso es equivalente a:

costoTotal(4,9) = distancia[4] + distancia [5] + distancia[9].

Este algoritmo puede responder cada consulta en O(log(n)) si construimos un árbol de segmentos que lleve la suma de las distancias entre cada rango, tomando en cuenta los elementos que se repiten para no sumarlos.

Finalmente, el algoritmo puede responder Q consultas de esta forma en O(|N| + Qlog(|N|)) usando memoria adicional O(|N|).

Pregunta 4:

4. Sea un grafo $G = (N, C)$, dos nodos $s, t \in N$ y una función $cap : N \to \mathbb{N}$. Cada nodo tiene capacidad $cap(n)$, que corresponde a la cantidad de flujo que puede pasar a través de n por unidad de tiempo. Queremos hallar el flujo máximo de s hasta t. Llamaremos a este problema $MAX\text{–}FLOW_N$

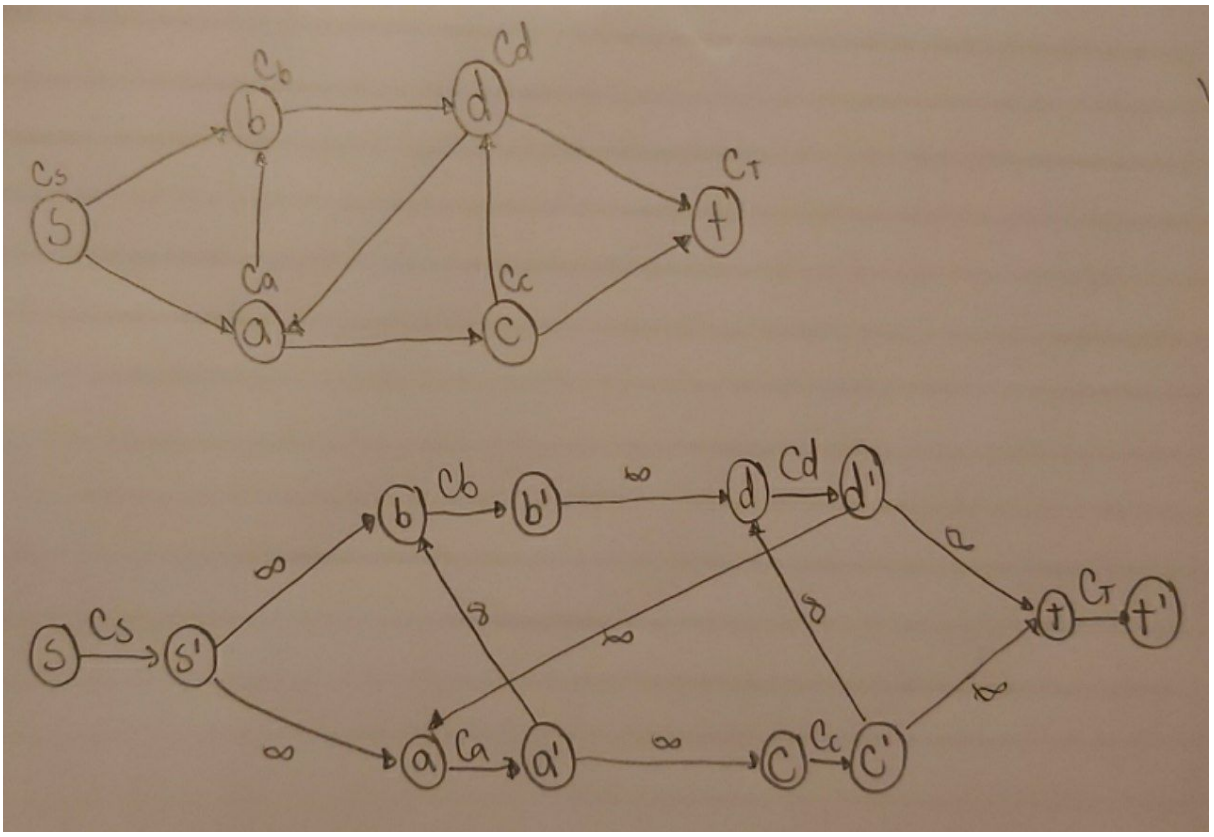
   El problema de flujo máximo cuando las capacidades están en las conexiones lo llamaremos $MAX\text{–}FLOW_C$. ¿Cómo haría una P–reducción de $MAX\text{–}FLOW_N$ a $MAX\text{–}FLOW_C$?

Podemos crear una P-reducción de MAX-FLOXn a MAX-FLOWc de la siguiente manera:

Primero, consideramos que las capacidades se encuentran en los nodos. Por lo tanto, debemos conseguir una forma de llevarlas a las aristas.

Podemos dividir un nodo X en dos: nodo X y nodo X', los cuales están conectados por una arista que vaya de X a X', tal que X -> X'. En esta arista dentro de los nodos tendrá la capacidad cap(n) que tenía en un principio el nodo X.

Para las demás conexiones que no posean cap(n) (es decir, para todas las conexiones restantes que no sean de la forma X -> X'), se les asignará infinito. De tal forma, que la P-reducción tendrá esta forma:



La cantidad que pueda pasar desde la arista X -> X' es la misma cantidad que pasaría por el el vértice X en el problema original.

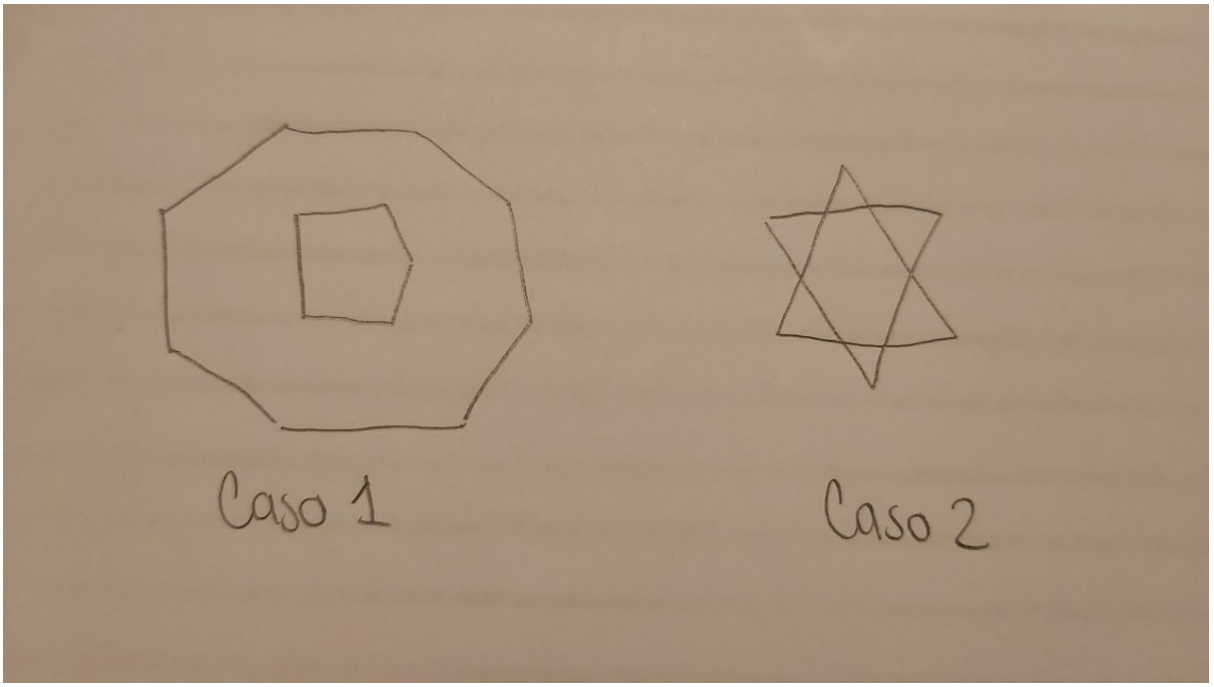
Al resolver el problema de flujo máximo con esta p-reducción, la solución sería equivalente a la solución del problema original.

Pregunta 5:

5. Sean $P = \langle p_1, p_2, \cdots, p_n, p_1 \rangle$ y $Q = \langle q_1, q_2, \cdots, q_m, q_1 \rangle$ dos polígonos convexos de n y m vértices, respectivamente. Se dice que estos dos poligonos intersectan si existe un punto w tal que w esté en el interior de P y w esté en el interior de Q.

   Diseñe un algoritmo que decida si dos polígonos convexos P y Q intersectan en $O(nm)$, usando memoria adicional $O(1)$.

Para saber si dos polígonos se intersectan, tenemos dos casos:



- Caso 1: Un polígono se encuentra completamente dentro de otro. En este caso, debemos ver si los vértices del polígono interno se encuentran dentro del polígono externo. Para esto, debemos utilizar el algoritmo para saber si un punto está contenido en un polígono y los puntos serán cada vértice del polígono interno. Por lo tanto, este algoritmo toma tiempoO(nm).

- Caso 2: Un polígono A se intersecta con un polígono B y ninguno de sus vértices se encuentran dentro del otro polígono. Para esto debemos usar el algoritmo de Intersección de Segmentos, ya que p1-p2 es un segmento, p2-p3 es un segmento, y así sucesivamente, entonces debemos ver si se intersectan con los segmentos de Q, los cuales son q1-q2, q2-q3,... qm-q1. Por lo tanto, este algoritmo también toma tiempo O(nm).

En ambos casos se toma tiempo adicional constante (O(1)) para realizar estas operaciones.

Por lo tanto, para ver si dos polígonos P y Q se intersectan, se deben verificar tanto el caso 1 como el caso 2. Si ninguno de los casos se cumple, entonces los polígonos no se intersectan. De lo contrario, si se intersectan. Esto tomaría tiempo O(nm) + O(nm), es decir, O(nm).

Pregunta 6:

6. Sea A y B dos matrices $n \times n$, para algún entero $n > 0$.

   Sospechamos que $B = A^{-1}$. Esto es, que B es la matriz inversa de A.

   Diseñe un algoritmo de Monte Carlo que permita confirmar esta sospecha, con un cierto error permitido ε, usando tiempo $O(n^2 \log \frac{1}{\varepsilon})$.

Como existe la sospecha de que $B = A^{-1}$, entonces queremos verificar si $BA = I$ (Ya que $A^{-1}A = I$), en donde I es la matriz Identidad.

Una manera eficiente de verificar si $B = A^{-1}$ es utilizando el algoritmo de Freivalds, en donde generamos un vector X y verificamos si $XAB = XI$. Si es cierto, es posible que hayamos tenido suerte al seleccionar el vector. Si es falso, $B \neq A^{-1}$.

Este algoritmo se debe ejecutar $\log(1/\varepsilon)$ veces para que el error se encuentre por debajo de $\varepsilon > 0$ y poder disminuir la probabilidad de fallar.

Pregunta 7:

7. Sea un grafo $G = (N, C)$, decimos que $V \subseteq N$ es un cubrimiento de vértices para G si todas las conexiones tienen alguno de sus extremos en V.

$$(\forall a, b \in N : \{a, b\} \in C \Rightarrow a \in V \vee b \in V)$$

Sea $MIN{-}COVER$ el problema de hallar un cubrimiento de vértices de cardinalidad mínima. Sabemos que $MIN{-}COVER$ es NP–completo.

Diseñe un algoritmo para el problema 1–relativo–$MIN{-}COVER$ asociado. Esto es, diseñe un algoritmo eficiente (en tiempo polinomial) que resuelve el problema del mínimo cubrimiento de vértices, produciendo una respuesta que es a lo sumo el doble de la solución óptima. Debe demostrar que esto último es cierto para su algoritmo propuesto.

Podemos diseñar el siguiente algoritmo para este problema:

Primero, se selecciona uno de todos los arcos al azar, se seleccionan los dos vértices de los extremos y los colocamos en un conjunto. Luego, buscamos otro arco tal que ninguno de los dos vértices estén en nuestro conjunto y se guardan en él. Y se repite este último paso las veces que sean necesarias.

Si con el algoritmo nosotros seleccionamos K arcos, entonces dentro de nuestro conjunto vamos a obtener 2K vértices, (ya que siempre seleccionamos los dos vértices extremos de cada arco).

El cubrimiento de vértices que posee el MIN-COVER, debe poseer al menos uno de esos dos vértices que posea nuestro conjunto, porque si no posee al menos uno de esos dos vértices, entonces el arco no está contemplado en la definición, lo cual es una contradicción, ya que todos los arcos se tienen que contemplar. Por lo tanto, el cubrimiento de vértices que posee el MIN-COVER tiene al menos K vértices.

Por esta razón, nuestro conjunto tendría a lo sumo el doble de elementos del MIN-COVER.

Pregunta 8:

8. Sea $R = \{r_1, r_2, \cdots, r_n\}$ un conjunto de n rectángulos, donde cada rectángulo es un par ordenado $(alto, ancho)$.

   Para cualquier subconjunto $C \subseteq R$, definimos el costo de C como la multiplicación del alto del rectángulo más alto en C por el ancho del rectángulo más ancho en C (nótese que no necesariamente es el mismo rectángulo quien tiene estos máximos).

$$costo(C) = \left(\max_{(alto,ancho) \in C} alto \right) \times \left(\max_{(alto,ancho) \in C} ancho \right)$$

   Queremos realizar una partición de R en subconjuntos $C_1, C_2, \cdots, C_m$ de tal forma que

   - $C_1 \cup C_2 \cup \cdots \cup C_m = R$
   - $C_1 \cap C_2 \cap \cdots \cap C_m = \emptyset$

   La cantidad m de subconjuntos que forma la partición es libre, entre 1 y n. El costo de la partición es la suma de los conjuntos que lo conforman.

   Diseñe un algoritmo que permita hallar una partición de costo mínimo en $O(n \log n)$, usando memoria adicional $O(n)$.

Si analizamos el problema, podemos observar que tenemos en un principio un conjunto con todos los rectángulos.

Si existe un rectángulo A cuyas dimensiones sean ambas menores o iguales que las de otro rectángulo B, ese rectángulo A no aportaría nada a la solución final. Ya que costo(C) = max alto X max ancho. Entonces, podríamos resolver el problema sin él. Por lo tanto, lo eliminamos de la solución.

Por lo tanto, el conjunto que nos queda cumple una característica interesante. La cual es que, si por ejemplo, ordenamos los rectángulos por altura, mientras aumenta la altura, disminuye la anchura.

Podemos reducir el problema a conseguir subarreglos, de un arreglo en donde se encuentren estos rectángulos. Esto se puede solucionar seleccionando un primer subarreglo del arreglo, y luego resolviendo el resto recursivamente.

Queremos calcular:

f(n) = Suma mínima de los costos de los subconjuntos si consideramos todos los rectángulos desde 0 hasta n.

Por lo tanto, primero intentaremos hallar f(i), en donde i es arbitrario y i < n.

Hay que escoger todas las opciones donde se puede picar el subarreglo, por lo tanto, calcularemos f(j) = Suma mínima de los costos de los subconjuntos si consideramos todos los rectángulos desde 0 hasta j, con 0 <= j < i.

La solución del subarreglo [j...i] sería la anchura de j X altura de i. Al tener subarreglo, nos faltaría picar el arreglo que va de [0..j-1], y esto se resuelve recursivamente.

Sin embargo este problema tomaría tiempo $O(n^2)$ ya que tenemos N estados y en cada uno de los estados debemos iterar por las posibilidades donde podemos dividir el subarreglo. Podemos crear una optimización de la siguiente manera:

Lo podemos ver como Y(Xi) = mi x Xi + bj, tratando de obtener el mínimo de todos. Además, podemos observar que se cumple que mj > mi si j < i porque las anchuras decrecen a medida que aumenta la altura. Se puede ver como una ecuación de la recta.

Por lo tanto, finalmente podemos aplicar la optimización de Convex Hull ya que podría resolver este problema en tiempo O(nlog(n)).

Pregunta 9:

9. Sea $G = (A \cup B, C)$ un grafo bipartito, tal que $|A| = |B|$ y $(\forall a \in A, n \in B : \{a, b\} \in C)$. Notemos que siempre es posible crear apareamientos de cardinalidad $|A|$ (o, equivalentemente, de cardinalidad B).

   Definimos una función de costo $c : C \to \mathbb{N}$

   Diseñe un algoritmo usando *Branch & Bound* que halle un apareamiento de máxima cardinalidad y mínimo costo (según la función c)

Podemos resolver este problema utilizando Branch & Bound como un problema de asignación, ya que |A| = |B|.

Tenemos na nodos que pertenecen al conjunto A, los cuales deben estar conectados a nb nodos que pertenecen al conjunto B. Podemos observar que, para crear un apareamiento de máxima cardinalidad y mínimo costo, solamente un nodo de na puede estar conectado con un nodo de nb. La asignación óptima será aquella en donde la suma de los costos de la función de costos c sea mínima.

En un principio calculamos la cota inferior y superior para la solución. Luego, en cada paso del problema, se irá asignando una arista que conecte un nodo que pertenezca a A, con un nodo que pertenezca a B, y seguiremos explorando solamente si existen posibilidades que se encuentren dentro del rango que previamente calculamos.

Finalmente, después de asignar n aristas a los nodos, encontraremos un apareamiento de máxima cardinalidad y mínimo costo según la función c.

Pregunta 10:

10. Considere un número entero positivo X. Definimos la función $decomp(X)$ como la cantidad de enteros positivos, a, b, c y d de tal forma que $ab + cd = X$.

$$decomp(X) = |\{(a, b, c, d) : a > 0 \wedge b > 0 \wedge c > 0 \wedge d > 0 \wedge ab + cd = X\}|$$

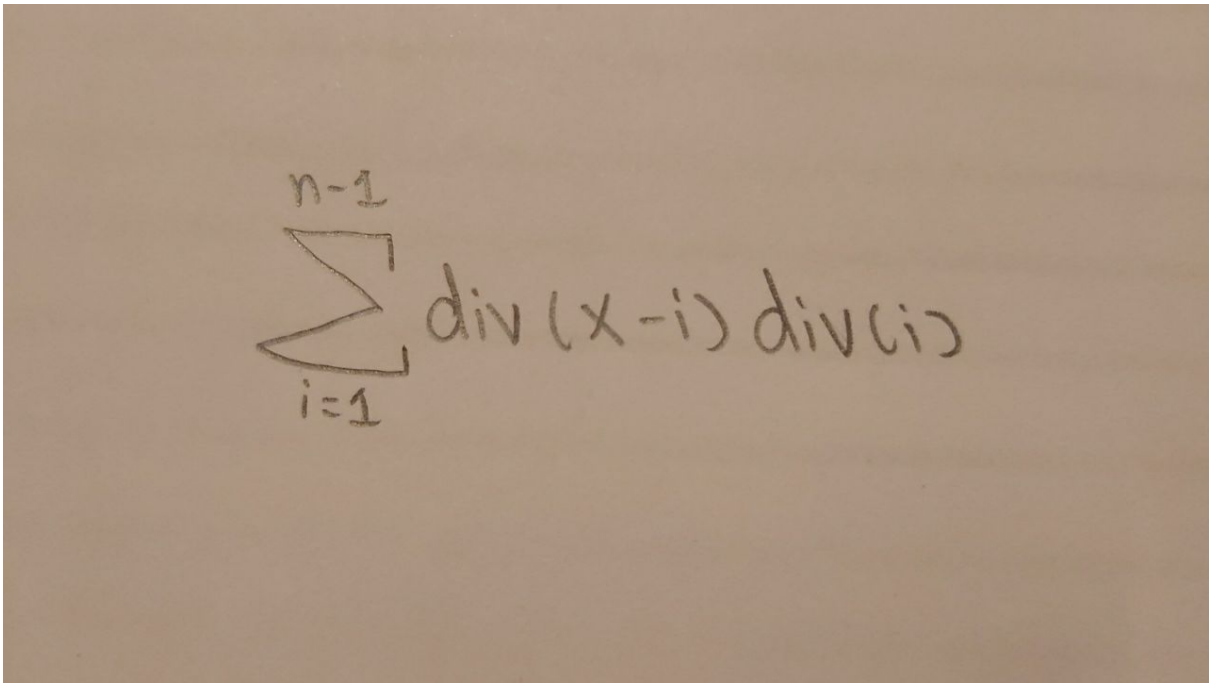
    Dado un número N, queremos hallar el máximo valor para $decomp(X)$ donde $1 \leq X \leq N$.

    Diseñe un algoritmo que permita encontrar la respuesta en $O(N \log N)$

    Pista: Considere investigar los diferentes usos de la Criba de Eratóstenes

Si analizamos paso por paso el problema, tenemos que:

Un número n se puede escribir de la forma ab tantas veces como número de divisores tenga n. Por lo tanto, X = ab + cd lo puedo escribir de la siguiente manera:



Por lo tanto, esa sumatoria sería igual a decomp(X) y se puede calcular en tiempo O(n).

Luego, tenemos que saber cómo calcular div(n). Utilizando la Criba de Eratóstenes podemos obtener los números primos entre [1..n]. Por lo tanto, n se puede descomponer en p1^e1 * p2^e2 .... * pk^ek, y poseerá (e1 + 1)(e2 + 1)...(ek + 1) divisores.

Por lo tanto, div(n) = (e1 + 1)(e2 + 1)...(ek + 1) y se puede calcular en O(log(n)).

Luego, calcular cada decomp(N) toma tiempo O(n), ya que precalcular todos los div(x) que necesitamos tarda O(nlog(n)) y los podemos almacenar.