Resolución primer parcial TN

July 11, 2024

Pregunta 1

Respuestas correctas

- $O(j!)$

Explicación

Hay dos formas de generar todas las posibles soluciones:

1. Elegir un cajón para cada juguete. Esto nos da c^j posibles asignaciones.
2. Ordenar los juguetes de alguna manera, y poner los primeros en el primer cajón hasta que no entren más, los siguientes en el segundo hasta que no entren más, etc. Aquí hay $j!$ combinaciones posibles.

En principio uno podría decir listo, estas son las complejidades del algoritmo. Pero **estaría equivocado**: estas cantidades describen cuántas **hojas** va a haber en el árbol de backtracking, no la cantidad total de nodos, ni cuánto tiempo se tarda por nodo. Vamos entonces a analizar los dos casos por separado.

1. Un algoritmo de backtracking que asigne un cajón a cada juguete construiría la solución parcial como una asignación de los primeros i juguetes a cajones. Pasar por todas las elecciones de cajones para un juguete no tarda más que $O(1)$ por cajón, así que cada nodo interno del backtracking toma $O(1)$. ¿Cuántos nodos tiene nuestro árbol? El primer nivel tendrá uno solo, el segundo tendrá c, el tercero c... es decir, la cantidad total de nodos es

$$O\left(\sum_{i=0}^{j} c^i\right) = O\left(\frac{c(c^j-1)}{c-1}\right) = O(c^j).$$

   Sólo faltaría ver cuánto trabajo hacemos en las c^j hojas del árbol. En principio, tendríamos que fijarnos para la asignación de j juguetes si entran en todos los cajones. Hacerlo bien naïf involucraría un costo de $O(j)$ en cada hoja, lo que nos lleva la complejidad a $O(c^j \cdot j)$. Por suerte podemos repartir ese trabajo entre todos los nodos. Vamos guardando un

arreglo con la suma de las alturas de los juguetes en cada cajón. Cuando agregamos un juguete, directamente le sumamos su altura al cajón correspondiente. Así, agregar un juguete sigue siendo $O(1)$. Además, vamos pasando un número que es la cantidad de cajones que están sobrepasados, el cual cambiamos cuando agregamos un juguete. En las hojas entonces solo necesitamos fijarnos si ese número es 0, lo cual es $O(1)$. Al final conseguimos la complejidad que queríamos de $O(c^j)$...?

NO. La igualdad de arriba solo vale si $c > 1$. Si $c = 1$, esto termina siendo $O(j)$. La O grande funciona raro con múltiples variables, lo que nos obliga a decir que esto en realidad es $O(c^j + j)$. Recontra molesto, y parte de la razón por la que fuimos más laxos con la corrección.

2. Para construir un orden de los juguetes podemos, por cada posición del orden, elegir un juguete de los que queden poner. Una solución parcial entonces va a ser un prefijo de un orden de juguetes. Agregar un juguete nuevo a este orden, si se hace inteligentemente, se puede hacer en $O(1)$. Eso es lo que vamos a tardar en cada nodo interno entonces en principio. ¿Cuántos nodos internos hay? Sabemos que hay $j!$ hojas en el último nivel. En el anteúltimo va a haber $j!/1!$, en el antepenúltimo $j!/2!$, en el anterior a ese $j!/3!$ ... Para ponerlo en una fórmula, sería

$$\sum_{i=0}^{j} \frac{j!}{i!} = j! \sum_{i=0}^{j} \frac{1}{i!} \leq j! \cdot 3.$$

Es decir, $O(j!)$ nodos internos.

Perfecto, ahora veamos cuánto se tarda en cada hoja. Si lo hacemos bien fuerza bruta, tendríamos que recorrer todos los juguetes en cada hoja, sumando en un contador la altura de cada uno hasta pasarnos de A. Esto nos llevaría la complejidad a $O(j! \cdot j)$, que es más que lo que queremos. Por suerte podemos pasar el trabajo a los nodos internos: cada vez que agregamos un juguete, calculamos la altura total de los juguetes en el cajón correspondiente. Si se pasa, comenzamos el siguiente cajón. Además, vamos acumulando cuántos cajones vamos usando. Así, el costo en cada nodo interno sigue siendo $O(1)$, y ahora fijarse en cada hoja es solo comparar si la cantidad de cajones usados es menor a c. Con esto reducimos el costo en cada hoja a $O(1)$.

En total entonces tenemos complejidad $O(j!)$.

Bonus track: hay *otra* posible solución, si observamos que en la 1. asignar el primer juguete al cajón 1 o al cajón 26 es lo mismo. Lo que vamos a hacer entonces es asignar juguetes a cajones que ya tengan algún juguete, o a un cajón nuevo cualquiera. Cada nodo del árbol va a tener una cantidad de hijos variable, que va a depender de cuántos cajones había asignado su padre. La cantidad de nodos del árbol acá nos da

$$O\left(\sum_{i=0}^{j} \sum_{k=1}^{c} S(i,k)\right),$$

donde $S(i, k)$ es el número de Stirling de segundo tipo. Andá a saber eso.

Pregunta 2

Respuestas correctas

- Ninguna de estas respuestas es correcta.

Explicación

- ~~Utilizar memorización garantiza que solo se calculen los subproblemas necesarios, lo que siempre reduce el tiempo de ejecución del algoritmo.~~ No siempre se reduce el tiempo de ejecución. Además, a veces calculás subproblemas que después no usás, como ser muchas veces que hacemos bottom-up.
- ~~Cuando hay múltiples parámetros en la función implementada se necesita una matriz (de 2 o más dimensiones) para memorizar los resultados.~~ A veces algunos de los parámetros son constantes que no hace falta memroizar.
- ~~Utilizar memorización te obliga a utilizar una solución recursiva para cada problema.~~ Fijate cómo te sale más fácil hacer bottom-up y después hablamos.
- **Ninguna de estas respuestas es correcta.** Esta sí es correcta, así que medio mentirosa.
- ~~Hay más de un posible valor de retorno para cada subproblema.~~ No podríamos memorizar si este fuese el caso.

Pregunta 3

Respuestas correctas

- $O(\log(n) \cdot k)$

Explicación

Acá hay dos posibles algoritmos, los dos con la misma complejidad. Los dos se bancan en el hecho que nos podemos fijar rápidamente si un intervalo de valores tiene un hueco o no. Por ejemplo, si tenemos que el elemento de la posición 3 es 15, y el de la posición 5 es el 13, como el arreglo tiene todos elementos es decreciente, forzosamente la posición 4 tiene que tener el 14, y entonces no hay ningún hueco entre la posición 3 y la 5. En general, si tenemos que la diferencia entre el elemento en la posición i y el de la posición j es igual a $j - i$, no hay huecos entre las posiciones j e i.

El primer algoritmo consiste en partir el arreglo por la mitad, encontrar el mayor hueco a la izquierda y a la derecha recursivamente, y tomar el mayor de los dos. En principio, esto sería una recurrencia en la que en cada paso partimos en 2, y sumamos un costo constante por cada merge:

$$T(n) = 2T(n/2) + 1.$$

Esto es $O(n)$. Podemos sin embargo aplicar lo que notamos en el párrafo anterior para no hacer recurrencia cuando ya sabemos que en una de las mitades no puede haber ningún hueco. Como sabemos que hay k huecos, como máximo vamos a tener k caminos en el árbol de recurrencia que alcancen una hoja del último nivel. Podemos acotar la cantidad de nodos en este árbol como $O(\log(n) \cdot k)$, lo cual nos deja la complejidad que queríamos. Sin embargo, la cantidad de hijos de cada nodo en el árbol no es constante, y entonces no podemos usar el teorema maestro para calcular la complejidad.

El segundo algoritmo, en cambio, usa búsqueda binaria para encontrar la primer posición en la que se detecta que hay un hueco, usando la propiedad que vimos antes. Para la búsqueda binaria, se fija si entre la posición 0 y la i puede haber un hueco comparando las dos posiciones. Si hay un hueco, achica el i, sino, lo agranda. Una vez que encontró la primer posición en la que hay un hueco, reinicia la búsqueda desde esa posición. En resumen, hace una búsqueda binaria por cada hueco en el arreglo, es decir, $O(\log(n) \cdot k)$. El chiste acá es que el algoritmo de búsqueda binaria es divide and conquer, o sea que el $\log(n)$ puede salir del teorema maestro. Técnicamente no se puede calcular la complejidad del algoritmo total con el teorema maestro, y es discutible si esto es un algoritmo de divide and conquer, pero ninguna de las respuestas que dan razones por las que no se puede calcular su complejidad con el teorema maestro dan la razón correcta. Decidimos entonces poner como correctas las dos opciones.

Pregunta 4

Respuestas correctas

- La complejidad es ajustada, la demostración es incorrecta.

Explicación

La demo es incorrecta porque ϵ no puede ser negativo en el teorema maestro. Sin embargo, $5 \cdot n^{1/4}$ sí pertenece a $O(n^{\log_9 3-\epsilon})$ para un ϵ muy chico, ya que $O(n^{\log_9 3-\epsilon}) \approx O(n^{1/2})$, así que podemos arreglar la demo para que nos quede la misma complejidad.

Pregunta 5

Respuestas correctas

- Toda orientación acíclica de un grafo tiene al menos un sumidero.

- Todo grafo etiquetado con al menos una arista tiene una cantidad par de orientaciones acíclicas distintas.

Explicación

- *Toda orientación acíclica de un grafo tiene al menos un sumidero.* Si no hay sumidero, forzosamente existe un recorrido infinito, que como el grafo es finito debe repetir vértices, y por lo tanto tener un ciclo.
- *Todo grafo etiquetado con al menos una arista tiene una cantidad par de orientaciones acíclicas distintas.* Tomemos una orientación acíclica cualquiera y demos vuelta todas las aristas. Claramente sigue siendo acíclica, porque cada ciclo de la orientación nueva sería un ciclo de la orientación original. Emparejar las orientaciones acíclicas de esta manera demuestra que son todos grupitos de 2, y por lo tanto hay una cantidad par.
- ~~Todo grafo etiquetado con al menos una arista tiene una cantidad impar de orientaciones acíclicas distintas.~~ Es el contrario de la primera.
- ~~Todo grafo etiquetado tiene al menos un sumidero.~~ Un grafo etiquetado ni siquiera es dirigido.

Pregunta 6

Respuestas correctas

- Toda orientación acíclica de K_n tiene un único sumidero.
- Si a una orientación acíclica H de un grafo le agregamos un nuevo vértice v junto a una arista $u \to v$ para todo vértice u de H, entonces el grafo resultante es la orientación acíclica de un grafo.

Explicación

- *Toda orientación acíclica de K_n tiene un único sumidero.* Si tuviese dos u y v, serían adyacentes en el grafo original, y la arista (u, v) tendría que o salir de u o salir de v, contradiciendo que son los dos sumideros. Además, sabemos que tiene por lo menos uno por lo visto en la Pregunta 5.
- *Si a una orientación acíclica H de un grafo le agregamos un nuevo vértice v junto a una arista $u \to v$ para todo vértice u de H, entonces el grafo resultante es la orientación acíclica de un grafo.* Ningún ciclo puede contener a v, porque v es sumidero. Y tampoco ningún ciclo puede no contener a v, porque entonces sería un ciclo en H, que era una orientación acíclica.
- ~~Toda orientación acíclica de K_n tiene exactamente dos sumideros (para $n \geq 4$).~~ Ver el primer ítem.
- ~~El grafo K_n etiquetado tiene exactamente n^2 orientaciones acíclicas distintas.~~ Son $n!$, todos los posibles órdenes de los vértices.

Pregunta 7

Respuestas correctas

- Si G es un grafo conexo entonces no es un grafo junta.
- La existencia de un ciclo en un grafo asegura que el grafo es conexo.

Explicación
Esta era la que había que elegir las **falsas**.

- *Si G es un grafo conexo entonces no es un grafo junta.* Contraejemplo: K_2.
- *La existencia de un ciclo en un grafo asegura que el grafo es conexo.* Contraejemplo: $K_3 \cup K_1$.
- ~~El digrafo resultado de darle una orientación a un grafo completo tiene un camino de longitud $n-1$.~~ Lo vamos a demostrar por inducción en la cantidad de vértices. Claramente si el digrafo tieme un sólo vértice podemos encontrar un camino de 0 aristas. Pasemos al paso inductivo. Sea D una orientación de un grafo completo de $n+1$ vértices. Tomemos v un vértice cualquiera de D. Consideremos $D' := D - v$. Por HI, hay un camino P' de longitud $n-1$ en D'. Si el primer vértice de P' tiene un arco desde v en D, agregamos a v al principio de P' para crear P, un camino de tamaño n en D. Asumamos entonces que el arco entre v y el primer vértice de P' está dirigido hacia v. Tomemos entonces el primer vértice u de P' tal que la arista entre u y v está dirigida hacia u, si existe. Sea w el vértice anterior a u en P'. Sabemos que w tiene un arco que va hacia v. Modificamos entonces P' cambiando la arista $w \to u$ por la secuencia $w \to v \to u$ para conseguir un camino P de tamaño n en D.

  Nos queda un caso en el que ese u no existe. Esto quiere decir que todas las aristas desde D' están dirigidas hacia v. En particular, hay un arco saliendo del último vértice de P' hacia v. Armamos entonces P agregando a v al final de P', y conseguimos el camino que queríamos en D.
- ~~Si G tiene exactamente 2 vértices con grado impar, tiene que haber un camino entre ellos.~~ Tomemos 2 vértices con grado impar u y v. Supongamos que no hay camino entre ellos, o sea, que pertenecen a distintas componentes conexas. Sabemos que la suma de los grados es par en cada componente conexa. Además, no puede haber otro vértice con grado impar en ninguna de las dos componentes conexas, porque u y v eran los únicos vértices de grado impar. Pero entonces hay un sólo vértice de grado impar en cada componente conexa, y la suma tiene que dar impar. **¡ABSURDO!**

Pregunta 8

Respuestas correctas

- Todos los vértices a distancia k de la raíz son visitados antes que los de distancia $k+1$.

Explicación

- *Todos los vértices a distancia k de la raíz son visitados antes que los de distancia $k+1$.* Así funciona BFS.
- ~~Se necesita que el grafo de entrada sea representado con una matriz de adyacencias para alcanzar la complejidad óptima.~~ Nop, se necesita que sea lista de adyacencias.
- ~~Para cada grafo G hay un único árbol BFS posible.~~ Simplemente cambiar el vértice de donde empezás te da un árbol potencialmente diferente.
- ~~Para toda arista (u, v) del grafo original, o u es ancestro de v en el árbol BFS, o v es ancestro de u.~~ Esto era para DFS nomás. Contraejemplo: K_3.

Pregunta 9

Respuestas correctas

- $O(|V| + |E|)$

Explicación

Cada arco $u \to v$ es visitado una sola vez: ni bien se visita u. Cada vértice también es visitado una vez, que es cuando se marca como visitado. Ergo, se hace una cantidad de operaciones proporcional a la suma de los dos.

Pregunta 10

Respuestas correctas

- Ninguna de estas respuestas es correcta.

Explicación

- ~~*La cantidad de vértices de D con grado de entrada 0.*~~ Contraejemplo: el grafo $u \to v$, si se empieza el primer DFS desde v.
- ~~*La cantidad de componentes conexas de D.*~~ Contraejemplo: el grafo $u \to v$, si se empieza el primer DFS desde v.

- ~~*La cantidad de componentes fuertemente conexas de D.*~~ Contraejemplo: el grafo $u \to v$, si se empieza el primer DFS desde u. *Fua este grafo sirve para todo.*
- ~~*La cantidad total de vértices de D.*~~ Contraejemplo: el grafo $u \to v$, si se empieza el primer DFS desde u. Para qué pensar en más vértices que 2.
- *Ninguna de estas respuestas es correcta.* Excepto esta.

Problema A

a)

$$f_{c,n,k,g}(i,a,h) = \begin{cases} \infty & \text{si } a < 0 \text{ o } h = 0 \text{ o } (i = n+1 \text{ y } a < k) \\ 0 & \text{si } i = n+1 \text{ y } a \geq k \text{ y } h > 0 \\ \min\{ & \\ \quad f_{c,n,k,g}(i+1, a-1, g), & \\ \quad f_{c,n,k,g}(i+1, a+6, h-1) + c_i, & \\ \quad f_{c,n,k,g}(i+1, a, h-1) & \\ \} & \text{sino} \end{cases}$$

Para resolver el problema hace falta llamar a $f_{c,n,k,g}(1,0,g)$.

b) Vamos a usar programación dinámica para no tener que calcular el resultado de un llamado dos veces. En principio creamos una matriz A de enteros con una entrada por cada posible combinación de argumentos de la función f. La matriz A va a tener $(n+1) \cdot 6n \cdot g$ entradas, lo cual es demasiado para la complejidad que nos piden. Vamos a tener que hacer algo mejor. Lo que vamos a hacer es acotar un poco más la cantidad de alfajores para los que necesitamos calcular la función.

Observemos que si tenemos muchísimos alfajores en la posición i, no vamos a necesitar comprar más en el resto del viaje. En este caso, el costo restante de hacer el recorrido va a ser 0. ¿Cuántos alfajores vamos a necesitar para recortar así? Seguro que por lo menos k, porque al final tenemos que llegar con k alfajores. Pero además necesitamos algunos alfajores para ir comiendo en el viaje. Tenemos que comer un alfajor en las próximas h ciudades, y después de eso comer uno cada g ciudades. Es decir, en total necesitamos tener $k + 1 + \frac{(n-i+1)-h}{g}$ alfajores para poder podar de esta manera. Este número siempre va a ser menor o igual a $k + \frac{n}{g}$, así que simplificamos un poco el código usando esta cuenta en cambio. Modificamos la funcioncita que hicimos antes de esta manera:

$$f_{c,n,k,g}(i,a,h) = \begin{cases} \infty & \text{si } a < 0 \text{ o } h = 0 \text{ o } (i = n+1 \text{ y } a < k) \\ 0 & \text{si } (i = n+1 \text{ y } a \geq k \text{ y } h > 0) \textbf{ o } \mathbf{a \geq k + \frac{n}{g}} \\ \min\{ & \\ \quad f_{c,n,k,g}(i+1, a-1, g), & \\ \quad f_{c,n,k,g}(i+1, a+6, h-1) + c_i, & \\ \quad f_{c,n,k,g}(i+1, a, h-1) & \\ \} & \text{sino} \end{cases}$$

Ahora sí, podemos hacer que la matriz de memorización tenga solo $(n+1) \cdot \min\{6n, k + \frac{n}{g}\} \cdot g$ entradas. Como cada llamado recursivo toma $O(1)$, nos queda que la complejidad es $O(n \cdot \min\{n, k + \frac{n}{g}\} \cdot g)$, que es lo que queríamos.

c) Tenemos dos opciones: o analizamos la función que creamos en el punto a), o la que hicimos en el punto b). La del punto b) es un **QUILOMBO**, así que vamos a empezar por la del a).

"Pero no entiendo, la función hace tres llamados recursivos en la mayoría de los casos, así que es suficiente decir que es $O(3^n)$ si hacemos backtracking." **NO.** Usar la O grande es dar una cota *superior* a la cantidad de llamados recursivos, pero nosotros queremos una cota *inferior*.

Bueno, suponete que tratamos de decir que la cantidad de llamados recursivos es $\Omega(3^n)$. ¿Qué pasa, por ejemplo, si $g = 1$? Sí o sí Alfredo va a tener que comerse un alfajor en cada ciudad y nada más. Como Alfredo empieza sin alfajores, se pone de mal humor inmediatamente. Hubo $O(1)$ llamados recursivos, lo cual es una función muuucho más chica que 3^n. O sea, seguro que entonces no podemos decir que la función es $\Omega(3^n)$. Ni siquiera podemos decir que la función depende solo de n, porque si $g = 1$ no nos importa el valor del resto de los argumentos para la cantidad de llamados.

Vamos a tener que partir en casos entonces. Y sabemos qué pasa cuando $g = 1$. ¿Qué otros casos hay? Vamos a partir en casos.

- $g = 1$: $O(1)$
- $g = 2$: Alfredo tiene que comer un alfajor cada dos ciudades. Lo primero que tiene que hacer es sí o sí comprar un alfajor, y en la siguiente ciudad comérselo. Después está más libre, aparte de tener que comer un alfajor cada dos ciudades. De alguna forma, cada llamado recursivo hace:
  1. o un solo llamado recursivo que no cae en un caso base, si se quedó sin alfajores y tiene que comprar;
  2. o un solo llamado recursivo que no cae en un caso base, si tiene que comer sí o sí un alfajor;
  3. o tres llamados recursivos que no caen en casos base, si no está obligado a comer ni a comprar alfajores.

El primer caso va a pasar cada 12 ciudades como máximo, ya que puede comprar de a 6 alfajores nomás, y se come uno cada dos ciudades. El segundo va a pasar cada dos ciudades, ya que $g = 2$. El último va a pasar en todos los otros casos.

La cantidad de llamados recursivos entonces va a ser por lo menos algo así como

$$1 + 1 + 3 + 3 + 3^2 + 3^2 + 3^3 + 3^3 + 3^4 + 3^4 + 3^5 + 3^5 + 3^5 + 3^5 + 3^6 + \dots$$

Notar que el 3^5 se repite 4 veces, porque en el segundo 3^5 se comió el último alfajor, y Alfredo tuvo que comprar más. Esta cuentita ya está siendo una cota inferior no muy ajustada, pero para lidiar con el tema de que cada 12 ciudades tenemos que repetir el exponente, vamos a ser más laxos todavía: vamos a decir que esta función es $\Omega(3^{n/4})$, porque la máxima cantidad de niveles seguidos sin aumentar el exponente es 4.

- $g > 2$: Por suerte lo que dijmos antes fue una cota inferior, y ahora Alfredo puede haber tomado esas mismas decisiones (comer un alfajor cada 2 ciudades), u otras más, así que sigue siendo una cota inferior. Usamos en este caso entonces la misma cota de $\Omega(3^{n/4})$.

Genial, entonces nos queda que si $g = 1$, tenemos $\Omega(1)$ llamados recursivos, y sino tenemos $\Omega(3^{n/4})$. *"Pero pará, no quiere decir esto que es siempre $\Omega(3^{n/4})$? O sea, el Ω nos dice cosas cuando la función tiende a infinito."* **NO.** La Ω nos dice cosas tomando en cuenta que cualquiera de los parámetros de la función puede quedarse constante. No es cierto que si g queda constante la función de cantidad de llamados recursivos crece más rápido que $3^{n/4}$ con respecto a n, y entonces sería falso decir que es $\Omega(3^{n/4})$.

Resumiendo, nos queda que la cantidad de llamados recursivos es $\Omega(\mathsf{llamados}(n, g))$, donde

$$\mathsf{llamados}(n, g) = \begin{cases} 1 & \text{si } g = 1 \\ 3^{n/4} & \text{sino} \end{cases}$$

Comparemos esto con la función de estados, que dijimos que es $O(n \cdot n \cdot g)$. Claramente si $g = 1$ no hay ninguna superposición de estados. Y si $g > 1$, con n suficientemente grande, seguro que sí, porque $n \cdot n \cdot g \leq n^3 \in O(3^{n/4})$.

Terminamos con el análisis para la función que definimos en el a). Para la del b) la idea sería separar en casos según el valor de $\frac{n}{g}$ y de k; si $k + \frac{n}{g} \leq 6$ ya sabemos que la primera compra que haga Alfredo va a caer en un caso base, y como sí o sí va a tener que hacer una compra antes de comer y en una de las primeras g ciudades, van a haber máximo g llamados recursivos que no sean a casos base. Si, en cambio, $k + \frac{n}{g} > 6$, Alfredo siempre va a poder comprar, comer un alfajor, y después elegir entre comer un alfajor o no hacer nada en cada ciudad hasta que tenga que comprar de nuevo, o sea que la cantidad de llamados recursivos no va a ser constante. Si quieren hacerla la dejamos de tarea.

d) Si, es posible. Una vez que terminamos de calcular los valores de la matriz de memorización, nos fijamos en el elemento en la posición $(i = 0, a = 0, h = g)$. Desde ahí, revisamos los elementos en las posiciones $(i = 1, a = 6, h = g)$ y $(i = 1, a = 0, h = g-1)$. Elegimos el que haya elegido el algoritmo de dinámica, comparando si el elemento en $(i = 1, a = 6, h = g)$ más c_i es menor al de $(i = 1, a = 0, h = g - 1)$. Guardamos en un arreglo esa decisión tomada, sea de compra o de no hacer nada, y pasamos a hacer lo mismo parándonos en la posición correspondiente. De alguna manera, estamos tomando las mismas decisiones que tomó el algoritmo anterior, pero ahora almacenando los pasos exactos que tomamos.

Problema B

Primero, traduzcamos los conceptos que nos presentan.

- **Mantenible:** sin ciclos.
- **Orden de actualización seguro:** orden topológico en el digrafo traspuesto.

a) Nos piden demostrar que si D es acíclico, entonces tiene un orden topológico en el traspuesto. Sabemos que si un digrafo es acíclico entonces tiene por lo menos un sumidero. Lo hacemos entonces por inducción en la cantidad de vértices: si tiene uno solo, claramente el orden que tiene el único vértice es topológico, porque no hay loops. Si tiene más de uno, sacamos un vértice v que sea sumidero, obtenemos un orden seguro σ de $D - v$, y le enchufamos v al principio. Este sigue siendo un orden seguro, porque v no depende de nadie, todos los que dependen de v están después de v, y para todo otro par de vértices ya era seguro. QED.

Otra forma era demostrar que el post-order armaba un orden topológico del digrafo, lo dejamos para ustedes.

b) Primero, vemos si el digrafo tiene algún ciclo. Creo que vieron en la teórica que un digrafo tiene ciclos si y sólo si tiene back edges en un recorrido DFS, así que esto es fácil y se hace en $O(n + m)$. Si tiene ciclos, listo, decimos que no es mantenible. Si no los tiene, hacemos lo que dijimos en el punto a): agarramos un vértice sumidero cualquiera, lo ponemos al principio del orden, lo sacamos del grafo, y repetimos hasta quedarnos sin vértices. Hay que tener cuidado con la complejidad acá: si buscamos un sumidero nuevo cada vez se nos va a $O(n^2)$. Lo que hacemos entonces es marcar cada vértice con su respectivo grado de salida al principio, y ponemos todos los que tengan grado 0 en una cola. Cuando sacamos un vértice v, restamos uno al grado de salida de todos los vértices que llegaban a v, y agregamos a la cola a todos aquellos que ahora tengan grado cero en consecuencia. Así, hacemos $O(m + n)$ para marcar todos los grados al principio, después $O(\deg_{\text{in}}(v))$ por cada vértice de D, o sea, $O(m)$, y $O(n)$ para agregar y sacar cada vértice de la cola. En total, $O(n + m)$.

El otro algoritmo sería hacer el post-order del digrafo y punto, pero quizá era más difícil de demostrar en el a).

c) El pseudocódigo se los dejo a ustedes.