1. Preguntas Opción Múltiple

1. Julieta encontró una caja con n piezas distinguibles de dominó, y se propuso encontrar el camino de mayor longitud que puede hacerse con las mismas*. Su estrategia para generar todos los candidatos a caminos consiste en probar cada posible ordenamiento de las piezas con cada posible orientación de cada pieza en la permutación, para luego buscar el camino más largo que quedó en esa configuración. Por suerte, conoce una técnica que le permite iterar por cada configuración, sin repetir, en tiempo constante. Tras unos minutos empleando esta estrategia se le ocurrió la siguiente idea: va a empezar de cero e ir agregando en cada paso una nueva pieza a la derecha del camino actual que está armando, probando una a una todas las piezas que sean compatibles con el camino armado hasta el momento (es decir, que se puedan orientar para que un lado coincida con la pieza más a la derecha del camino, considerando que la primera se puede orientar de las dos maneras). Cada vez que se quede sin piezas compatibles para seguir poniendo, volverá al último paso donde pueda poner alguna pieza que aún no haya probado en esa posición, y seguirá el camino utilizando esa pieza. Finalmente, se quedará con la mayor longitud que haya alcanzado formar durante este proceso. Decidir:

   - ☐ La segunda estrategia propuesta por Julieta no es correcta, existen casos para los que proveerá una longitud de cadena de dominós menor a la máxima posible.
   - ☐ La primera estrategia implica revisar $2^n \cdot n!$ soluciones candidatas, y toma $O(n)$ tiempo obtener el camino más largo en cada una.
   - ☐ En la segunda estrategia, para todo $1 \leq k \leq n-1$, una vez que Julieta llegó a armar un camino de longitud k, para la $(k+1)$-ésima pieza puede tener $2^{n-k}(n-k)$ opciones distintas posibles a considerar.
   - ☐ Una cota para la complejidad temporal de la segunda estrategia definida por Julieta es $O(2^n \cdot n)$.

   * Las piezas de dominó son rectangulares y tienen un número de cada lado. Una pieza se puede unir a otra si coinciden en algún número, y se unen a través ese mismo número. Por lo tanto, el número de ese lado ya no se puede usar para unir otra pieza. Por ejemplo, si tenemos las piezas $(2,3), (1,3)$ y $(1,4)$, podemos unirlas en el camino $(2,3), (3,1), (1,4)$. La longitud de un camino es igual a la cantidad de piezas en el camino.

Solución

La primera estrategia de Julieta es correcta. Prueba todos los ordenamientos posibles, con cada orientación de cada pieza del dominó. Tenemos $n!$ permutaciones distintas y para cada pieza tenemos 2 orientaciones distintas, para n piezas tenemos 2^n maneras de orientarlas. Además, para cada solución candidata, cuesta $O(n)$ buscar el largo del camino.

La segunda estrategia de Julieta chequea también todas las permutaciones distintas pero sin probar con aquellas configuraciones inválidas. Por ejemplo, para la primera pieza tiene n opciones, ya que sería poner una de las n piezas, luego en el paso k tiene dos opciones: pone una pieza si es que hay alguna que se pueda poner en ese lugar, o vuelve a la iteración anterior y se fija si puede probar poniendo otra.

Analicemos la complejidad de este algoritmo. Para la primera pieza tiene $2n$ opciones (por cada pieza, las dos orientaciones). A partir de ahí vale que en cada paso k le quedan $n-k$ opciones a lo sumo (y esto es porque después ya no hay libertad para elegir las orientaciones, solo se puede usar la que funciona, y si ambas funcionan es como si hubiera una sola opción). Entonces, una cota para la cantidad de permutaciones sería $O(n!)$. Ahora, dependiendo de cómo estén implementadas las piezas que se pueden poner en el paso k, la decisión de cuál poner se podría hacer en $O(1)$ o $O(n-k)$. Entonces, dependiendo de esto último la complejidad temporal del algoritmo puede ser $O(n!)$ o $O(n! \cdot n)$.

En base a esto, la única respuesta correcta es la opción dos. De la tercera estaría bueno señalar que para la $(k+1)$-ésima pieza le faltaría explorar $(n-k)!$ opciones distintas.

2. Román quiere resolver el problema SUBSET-SUM: dada una lista de números naturales $A = \{a_1, \ldots, a_n\}$ de longitud par mayor a cero y un número natural $k \in \mathbb{N}_0$, quiere decidir si existe un subconjunto $S \subseteq A$ tal que $\sum_{a_i \in S} a_i = k$. Para esto, propone inicialmente un algoritmo que implementa la siguiente recursión:

$$\text{ss}(i,j) = \begin{cases} \text{True} & i = n+1 \wedge j = 0 \\ \text{False} & (i = n+1 \wedge j > 0) \vee (j < 0) \\ \text{ss}(i+1,j) \vee \text{ss}(i+1, j-a_i) & \text{caso contrario} \end{cases}$$

y dice que $\text{ss}(1,k)$ le dará la solución al problema. Decidir:

- ☐ Una cota ajustada para la complejidad temporal de este algoritmo es $O(n2^n)$.
- ☐ El algoritmo propuesto por Román es correcto.
- ☐ Para cualquier lista A de números naturales de longitud par vale que al llamar a la función *ss* con parámetros 1 y k habrá por lo menos 2^n subllamados.

Solución

Respuestas correctas

- El algoritmo propuesto por Román es correcto.

Explicación

- *El algoritmo propuesto por Román es correcto.* Ya lo demostraron en la guía y en el ejercicio entregable si es que lo hicieron.
- ~~Una cota ajustada para la complejidad temporal de este algoritmo es $O(n2^n)$.~~ Cada llamado nos cuesta $O(1)$ resolverlo, por lo que la complejidad va a estar dada por la cantidad de nodos del árbol de llamadas. En este caso $O(n2^n)$ no es una cota ajustada, puesto que nuestro árbol binario tiene n niveles y 2^n hojas, debería ser más fina la cota. Una cota más fina sería $O(2^n)$, ya que $2^{n+1} - 1$ es la cantidad de nodos de un árbol binario completo. En este caso, debido a nuestros casos base, puede ser que tenga menos nodos que $2^{n+1} - 1$ ya que la recursión se cortaría antes.
- ~~Para cualquier lista A de números naturales de longitud par vale que al llamar a la función *ss* con parámetros 1 y k habrá por lo menos 2^n subllamados.~~ Esto podemos mostrar que es falso con un contraejemplo. Por ejemplo, si A es una lista de 1's de tamaño n y $k = 0$, entonces la función va a hacer $O(n)$ llamados, no 2^n.

3. Para mejorar su algoritmo, Román propone la siguiente estrategia: dividirá el conjunto A en dos mitades $A_1 = \{a_1, \ldots, a_{n/2}\}$ y $A_2 = \{a_{n/2+1}, \ldots, a_n\}$, y para $i \in \{1, 2\}$ calculará, con backtracking, el conjunto T_i, que denota todas las sumas que se pueden obtener usando subconjuntos de A_i (formalmente, $T_i = \{m \in \mathbf{N}_0 : \exists S \subseteq A_i, \sum_{a \in S} a = m\}$). Luego, ordenará T_1 y T_2 e iterará sobre los elementos de T_1 haciendo búsqueda binaria sobre T_2 para para detectar si existen dos números $t_1 \in T_1$ y $t_2 \in T_2$ tales que $t_1 + t_2 = k$. Decidir:

- ☐ La complejidad espacial de este segundo algoritmo es $O(n^2)$.
- ☐ El segundo algoritmo no es correcto cuando el valor k se obtiene usando elementos de sólo una de A_1 o A_2.
- ☐ Una cota para la complejidad temporal de este algoritmo es $O(n2^{\frac{n}{2}})$.
- ☐ El segundo algoritmo propuesto por Román es correcto.

Solución

Respuestas correctas

- El segundo algoritmo propuesto por Román es correcto.
- Una cota para la complejidad temporal de este algoritmo es $O(n2^{\frac{n}{2}})$.

Explicación

- *El segundo algoritmo propuesto por Román es correcto.* Nos podemos convencer de que es correcto viendo que al mergear ambas mitades vamos a tener todas las soluciones. También lo pueden demostrar cómo el anterior, aunque cómo no tienen que justificar en este MP con convencerse y tener una idea general de la demostración alcanza.
- ~~El segundo algoritmo no es correcto cuando el valor k se obtiene usando elementos de sólo una de A_1 o A_2.~~ Cómo la primera es correcta, esta respuesta es incorrecta.
- *Una cota para la complejidad temporal de este algoritmo es* $O(n2^{\frac{n}{2}})$. Podemos calcular cuánto nos cuesta cada operación: generar los conjuntos T_1 y T_2, ordenarlos y efectuar la búsqueda binaria. Así llegamos a que la complejidad está dada por generar T_1 y T_2, lo cuál está acotado por $O(n2^{\frac{n}{2}})$.
- ~~La complejidad espacial de este segundo algoritmo es $O(n^2)$.~~ En el peor caso, en el cual la suma de todos los subconjuntos den números distintos, deberíamos guardar un valor por cada subconjunto. En ese caso, la complejidad espacial sería $O(2^{n/2})$, por lo que $O(n^2)$ no es una cota correcta.

4. Un grafo etiquetado G es un grafo cuyos vértices tienen cada uno una etiqueta distinta en el conjunto $\{1, ..., n\}$, donde n es la cantidad de vértices de G. En G, cada arista se etiqueta utilizando las etiquetas de sus vértices incidentes. Dos grafos etiquetados de n vértices son iguales cuando tienen el mismo conjunto de aristas etiquetadas. Una orientación acíclica de un grafo (etiquetado o no) G es un grafo orientado H que no tiene ciclos dirigidos y que resulta de asignarle una dirección a cada arista de G. Ciertamente, si G es etiquetado, entonces H mantiene las mismas etiquetas que G. ¿Cuál/es de las siguientes afirmaciones son verdaderas?:

   ☐ Todo grafo etiquetado con al menos una arista tiene una cantidad impar de orientaciones acíclicas distintas.

   ☐ Todo grafo etiquetado tiene al menos un sumidero*.

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

   ☐ Toda orientación acíclica de un grafo tiene al menos un sumidero*.

   * Un sumidero es un vértice con grado de salida 0.

Solución

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.

5. Seguimos en el contexto del grafo etiquetado de la pregunta anterior. ¿Cuál/es de las siguientes afirmaciones son verdaderas? Aclaración: K_n es el grafo completo de n vértices.

   ☐ Toda orientación acíclica de K_n tiene un único sumidero.

   ☐ Toda orientación acíclica de K_n tiene exactamente dos sumideros (para $n \leq 4$).

   ☐ El grafo K_n etiquetado tiene exactamente n^2 orientaciones acíclicas distintas.

   ☐ 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.

Solución

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.

6. ¿Cuál/es de las siguientes afirmaciones son correctas sobre el recorrido BFS?

   ☐ Se necesita que el grafo de entrada sea representado con una matriz de adyacencias para alcanzar la complejidad óptima.

   ☐ Para toda arista (u, v) del grafo original, o u es ancestro de v en el árbol BFS, o v es ancestro de u.

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

   ☐ Para cada grafo G hay un único árbol BFS posible.

Solución

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.

2. Ejercicios a Desarrollar

1. Se tiene un array A de n elementos (por simplicidad, consideramos n potencia de 2) del cual se quiere obtener su máximo y su mínimo. Ciertamente, una manera de lograrlo es recorriendo A secuencialmente y comparando cada elemento contra el máximo y el mínimo que hayamos encontrado. Si el tipo de los elementos en A tiene una comparación costosa, tiene sentido considerar una manera alternativa de obtener estos valores. Proponga un algoritmo que solucione este problema buscando optimizar la cantidad de comparaciones y que no empeore la complejidad temporal de recorrer secuencialmente. Justifique.

Solución

El ejercicio se puede resolver usando D&C. **Dividimos** recursivamente el arreglo en dos mitades, **resolvemos** para cada una de estas y luego **combinamos** lo obtenido. La manera de hacerlo sería la siguiente:

- Si el arreglo tiene un solo elemento, ese será tanto el mínimo como el máximo.
- Si el arreglo tiene dos elementos, se comparan entre sí, y uno será el mínimo, mientras que el otro será el máximo.
- Si el arreglo tiene más de dos elementos, se divide en dos mitades, se calcula recursivamente el mínimo y el máximo de cada mitad, y luego se toma el menor de los mínimos y el mayor de los máximos.

El pseudocódigo sería algo como esto:

```
function ObtenerMaxMin(A, inicio, fin)
    if inicio = fin then                          ▷ Caso base: un solo elemento
        return (A[inicio], A[inicio])
    end if
    medio ← ⌊(inicio + fin)/2⌋
    (min_izq, max_izq) ← ObtenerMaxMin(A, inicio, medio)
    (min_der, max_der) ← ObtenerMaxMin(A, medio + 1, fin)
    min_global ← mín(min_izq, min_der)
    max_global ← máx(max_izq, max_der)
    return (min_global, max_global)
end function
```

La complejidad del algoritmo queda de la siguiente manera:

- $C(n) = 2C\left(\frac{n}{2}\right) + O(1)$

Ya que son dos los llamados recursivos y se divide el arreglo en dos en cada uno de ellos. Además, hacer la comparación para ver con qué mínimo y máximo se queda se hace en $O(1)$. Finalmente, si usamos el Teorema Maestro, nos da que la complejidad es $O(n)$. Además, al ver la cantidad de comparaciones de cada algoritmo, vemos que son asintóticamente del mismo orden.

2. Dado un grafo $G = (V, E)$ decimos que G está *conectadísimo* si es conexo y no tiene puentes. Aparte, dada una arista $e \in E$ decimos que e es *clave* si $G' = (V, E \backslash \{e\})$ no está conectadísimo (es decir, al sacar e obtenemos un grafo que tiene algún puente).

   a) Dar un algoritmo que dado un grafo conectadísimo G indique cuáles aristas de G son clave. El algoritmo debe tener complejidad $O(m^2)$. Justificar su diseño.

   b) Sea $G = (V, E)$ un grafo conectadísimo, $T = (V, E_T)$ un árbol DFS de G, y vw una backedge de T (es decir, $vw \in E \setminus E_T$). Probar que vw es clave si y solamente si hay una arista de E_T que solo está cubierta por vw.

   c) Dar un algoritmo que, dado el árbol DFS $T = (V, E_T)$ y una backedge vw, decida si vw es clave en $O(n)$. Se puede asumir que el árbol DFS tiene información adicional, siempre y cuando se pueda calcular en $O(n + m)$ y esta no dependa del eje vw.[1]

   d) Dar un algoritmo de complejidad $O(nm)$ que devuelva el conjunto de aristas clave de un grafo conectadísimo.

   e) Suponiendo que para cualquier árbol DFS ya sabemos qué tree-edges son claves. ¿Cómo harías un algoritmo que obtenga las aristas clave de un grafo conectadísimo en $O(m)$? **Ayuda**: Combinar los incisos a) y c).

   Criterio de aprobación: Dos incisos deben estar bien resueltos.

   Recordatorio: Se puede usar cualquier algoritmo visto en clase sin necesidad de justificación (tanto de correctitud como de complejidad).

Solución

Cómo G está conectadísimo, entonces $O(n+m) = O(m)$. Esto es necesario de aclarar, puesto que si no nuestra justificación de la complejidad sería incorrecta.

a) ¿Qué algoritmos conocemos para saber si un grafo tiene puentes? Correcto! El algoritmo de puentes que vimos en la práctica, que es el ejercicio 2 de la Práctica 4. ¿Lo podemos utilizar en el parcial? Sí, aunque siempre que vayan a utilizar un algoritmo/propiedad visto en la clase o en la práctica, consulten si lo pueden tomar cómo sabido o lo tienen que implementar/demostrar en el parcial. Ahora que ya tenemos el algoritmo de puentes, nuestro algoritmo para este primer inciso nos queda así.

[1] Es decir, la información adicional tiene que poder usarse para responder la consulta para cualquier backedge.

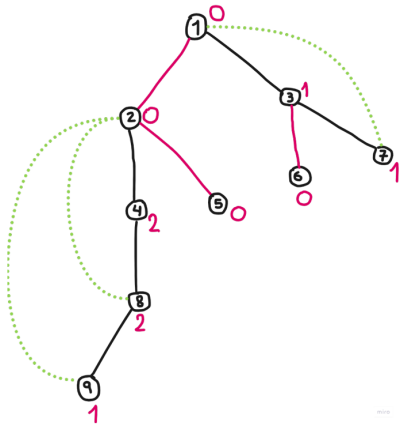

Figura 1: Ejemplo de un árbol DFS con los valores de *cubren* en sus nodos

a) Creamos un conjunto vacío llamado *claves*.

b) Iterar por cada una de las m aristas del grafo. m iteraciones

1) Obtenemos la arista e.
2) Removemos la arista del grafo G, obteniendo $G' = (V, E \setminus \{e\})$. ¿Cómo implementarían esto?
3) Corremos el algoritmo de puentes sobre G' para revisar si tiene puentes.
4) Si encontramos un puente en G' entonces agregamos e a *claves*.
5) Todas estas operaciones están acotadas por $O(m)$

c) Retornamos *claves*.

Nuestro algoritmo realiza m iteraciones de costo $O(m)$, por lo que la complejidad temporal de nuestro algoritmo es $O(m^2)$.

b)

Sea $G = (V, E)$ un grafo conectadísimo, $T = (V, E_T)$ un árbol DFS de G, y vw una backedge de T (es decir, $vw \in E \setminus E_T$). Probar que vw es clave $\Leftrightarrow$ hay una arista de E_T que solo está cubierta por vw.

Sea $G' = (V, E \setminus \{vw\})$. Sabemos que T va a ser un árbol DFS de G' también, puesto que la arista vw es una backedge, así que no afecta el resultado de DFS. De esta forma, las únicas aristas que van a ver su cobertura afectada son las del camino entre v y w (esto queda más claro más adelante con la fórmula). Este no sería el caso si vw fuera una tree-edge de T, puesto que ahí deberíamos volver a generar un árbol DFS nuevo o intentar rearmarlo a partir de T.

$\Rightarrow$ Partimos de que vw es clave. Por lo tanto, G' tiene un puente. Basándose en la definición de *cubren*, esto significa que un vértice $noCub$ de T' cumple que $cubren_{G'}[noCub] == 0$. También sabemos que ningún nodo cumple que $cubren_G[v] == 0$, pues G está conectadísimo y no tiene puentes. Pero la única diferencia entre G y G' es vw, y cubren se define cómo

$$\text{cubren}(v) = \text{backEdgesConExtremoInferiorEn}(v) - \text{backEdgesConExtremoSuperiorEn}(v) + \sum_{w \in \text{hijos}(v)} \text{cubren}(w)$$

Así que cómo la diferencia entre $cubren_{G'}$ y $cubren_G$ es a lo sumo 1, significa que $cubren_G[noCub] == 1$ (pues no puede ser 0). Por lo que $t = (noCub, parent[noCub])$ sólo está cubierta por vw.

$\Leftarrow$ Partimos de que hay una arista de E_T, $e = vw$ que solo está cubierta por vw. Por lo tanto, $cubren_G[v] == 1$. Así que al remover vw nos va a quedar $cubren_{G'}[v] == 0$, por la definición de arriba. Así que G' tendría un puente, por lo que vw es clave.

c)

Asumo que ya corrí el algoritmo de *cubren puentes* sobre G y, por lo tanto, tengo el valor de *cubren* de cada nodo. Esto se puede correr en $O(m)$ y no depende dé vw.

a) Obtenemos nuestra backedge vw, se cumple que w está ubicado más arriba en el árbol DFS.

b) Iteramos sobre los nodos que están en el camino de v a w en el árbol DFS, sin incluir a w.

c) Utilizando la propiedad del inciso *b*) sabemos que si alguno de los nodos n del camino, cumplen que $cubren[n] == 1$, entonces vw es clave. Puesto que va a generar un puente en la arista $(n, parent[n])$.

d) Si recorrí todos los nodos y ninguno cumplía la condición, entonces vw no es una arista clave.

El algoritmo consiste en iterar sobre las aristas en el camino de v a w y chequear una condición. La cantidad de iteraciones es $O(n)$ puesto que el camino no puede tener más de n vértices, luego cada operación me cuesta $O(1)$. Por lo que el costo de mi algoritmo es $O(n)$.

d)

a) Creamos un conjunto vacío llamado *claves*.

b) Corro DFS, luego separo las aristas en tree-edges y backedges. También corremos *puentes*. $O(m)$

c) Itero por cada una de las backedges. $O(m)$ iteraciones

1) Obtenemos la arista e.
2) Utilizamos el algoritmo del inciso *c*) para ver si es una arista clave o no. Si lo es, la agregamos a *claves*. $O(n)$

d) Itero por cada una de las tree-edges. $O(n)$ iteraciones

1) Obtenemos la arista e.
2) Utilizamos el algoritmo del inciso *a*) para ver si es una arista clave o no, corriendo el algoritmo de puentes. Si lo es, la agregamos a *claves*. $O(m)$

e) Retornamos *claves*.

El primer ciclo tiene $O(m)$ iteraciones que cuestan $O(n)$ cada una, por lo que en total tiene una complejidad de $O(nm)$. El segundo ciclo tiene $O(n)$ iteraciones (pues hay $n-1$ tree-edges), que cuestan $O(m)$ cada una, por lo que en total tiene una complejidad de $O(nm)$. Por lo que el algoritmo completo tiene una complejidad de $O(nm)$.

e)

Ahora sabemos cuáles de nuestras tree-edges son claves. Por lo que de nuestro algoritmo del inciso *d*) solo nos restaría modificar dada una backedge vw cómo definimos si es clave o no en $O(1)$. Viendo el algoritmo del inciso *c*), lo que necesitamos saber es si hay un nodo n que cumpla que $cubren[n] == 1$, *casi puente* los podemos llamar, en el camino entre v y w. ¿Qué cuenta podemos hacer?

$$cantCasiPuentesArriba(n) = esCasiPuente(n) + \ cantCasiPuentesArriba(parent[n])$$

Podemos implementar esta función utilizando programación dinámica y calcular estos valores en $O(n)$ (habiendo calculado *cubren* previamente). Ahora nos queda hacer el chequeo en $O(1)$ para una backedge vw para ver si es *clave*. Pero esto se resuelve viendo si $cantCasiPuentesArriba(v) - cantCasiPuentesArriba(w) > 0$. Pues esto significa que en el camino entre v y w hay al menos una arista *casi puente*. Cómo la cantidad de estados de *cantCasiPuentesArriba* es n y procesar cada uno cuesta $O(1)$ entonces el costo de pre calcular esto es $O(n)$. Así podemos reescribir nuestro algoritmo del inciso *d*.

a) Creamos un conjunto vacío llamado *claves*.

b) Corro DFS, luego separo las aristas en tree-edges y backedges. También corremos *puentes* y $cantCasiPuentesArriba$. $O(m)$

c) Itero por cada una de las backedges. $O(m)$ iteraciones

1) Obtenemos la arista $e = vw$.
2) Chequeamos $cantCasiPuentesArriba(v) - cantCasiPuentesArriba(w) > 0$ para ver si e es una arista clave o no. Si lo es, la agregamos a *claves*. $O(1)$

d) Retornamos *claves*.

3. Alfredo se está tomando unos días de descanso, así que va a recorrer diferentes pueblos de la Argentina y ha decidido que cuando regrese les regalará un alfajor a cada uno de los k demás docentes de la materia. Para esto ha decidido que los comprará en cada uno de los n pueblos que recorrerá. Al llegar al i-ésimo pueblo tiene tres opciones:

   - Comprar media docena (6) de alfajores locales por c_i pesos.
   - Comerse un alfajor en lugar de ir a la tienda a comprar.
   - No hacer ninguna de las dos anteriores.

   Y es que a Alfredo le gustan mucho los alfajores, y siente que si pasa g ciudades sin comerse al menos uno, se estresará mucho, lo cual repercutiría en sus clases. Además, sabemos qué $g \leq n$.

   a) Sea c la lista de precios de los alfajores $c_1, c_2, \ldots, c_n$, definir de forma recursiva $f_{c,n,k,g} : \mathbb{N}^3 \to \mathbb{N}$ donde

   $$f_{c,n,k,g}(i, a, h)$$

   calcula la cantidad mínima de pesos que Alfredo gastará al partir de la ciudad i con a alfajores y h ciudades que le quedan por recorrer sin comer alfajores antes que le agarre mal humor. ¿Qué llamado(s) se necesitan para resolver el problema? ¿Qué operación se debe realizar sobre esos llamados? Importante: acompañen a la definición recursiva con una explicación en castellano.

   b) Diseñe e implemente un algoritmo empleando dicha función, mencionando las estructuras que utilizaría. La complejidad del algoritmo debe ser $O(n \cdot \text{mín}(n, k + \frac{n}{g}) \cdot g)$ o mejor. (Ayuda: argumentar que no necesita comprar más de $k + \frac{n}{g} + 6$ alfajores).

   c) Demuestre que el algoritmo cumple con la propiedad de superposición de problemas, comparando con la solución que emplea backtracking.

   d) ¿Es posible usar el algoritmo anterior para determinar en qué pueblos tendría que comprar y en cuáles comerse un alfajor? ¿Es posible hacerlo sin empeorar la complejidad temporal? De ser posible, explique cómo lo haría.

Solución

a)

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

Para resolver el problema hace falta llamar a $f_{c,n,k,g}(1, 0, g)$. Además, habría que justificar cada llamado y los casos bases en palabras, con una explicación en lenguaje natural.

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:

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

Ahora sí, podemos hacer que la matriz de memorización tenga solo $(n+1) \cdot \text{mín}\{6n, k + \frac{n}{g}\} \cdot g$ entradas. Como cada llamado recursivo toma $O(1)$, nos queda que la complejidad es $O(n \cdot \text{mín}\{n, k + \frac{n}{g}\} \cdot g)$, 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:
  - *a*) o un solo llamado recursivo que no cae en un caso base, si se quedó sin alfajores y tiene que comprar;
  - *b*) o un solo llamado recursivo que no cae en un caso base, si tiene que comer sí o sí un alfajor;

c) 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.

4. Aye está remodelando su casa, y quiere cubrir el piso de su cuarto de n centímetros de largo con hileras de placas de madera. Compró un tablón de n centímetros, el cual quiere usar para la primera hilera. Ella sabe que no puede poner un segmento de madera de más de ℓ centímetros en el piso, porque se rompe. Por esto, quiere recortar su tablón de forma que cada segmento tenga a lo sumo tamaño ℓ. Además, en algunos lugares de la habitación va a poner muebles que cubran algunas partes del piso, así que se le ocurrió que si corta el tablón

sólo en lugares que vayan a quedar cubiertos, va a quedar todo mucho más lindo. Como ya tiene proyectado dónde va a poner los muebles, hizo marcas en la madera que corresponden a lugares en donde podría hacer los cortes (y sabe que esas marcas son suficientes como para que se puedan dejar tablas de la longitud requerida). Aye quiere resolver el tema rápido, por lo que propone la siguiente estrategia para minimizar la cantidad de cortes para un tablón dado:

a) Si el tablón tiene longitud menor o igual a ℓ, no hace nada.

b) En caso contrario, desde el borde izquierdo del tablón, elige la marca más lejana que esté a distancia menor o igual que ℓ posible, y corta allí.

c) Realiza el mismo procedimiento con el tablón de madera restante a la derecha del corte.

Aye no se anda con vueltas y se pone manos a la obra sin demostrar la correctitud de su estrategia.

Betina se enteró del problema de Aye pero cree que la estrategia para minimizar la cantidad de cortes es distinta:

a) Si el tablón tiene longitud menor o igual a ℓ, no hace nada.

b) En caso contrario, elige las dos marcas con máxima distancia entre ellas sin pasarse de ℓ (es decir, dos marcas distintas a distancia $d \leq \ell$ tal que $\ell - d$ sea mínimo), y corta en ellas.

c) Realiza el mismo procedimiento con los dos tablones restantes a la izquierda y a la derecha del que se acaba de cortar.

Debemos:

a) Demostrar que la estrategia de Aye funciona o dar un contraejemplo si no es el caso.

b) Dar un algoritmo de complejidad lineal o mejor.

c) Hacer lo mismo para el algo de Betina (complejidad cuadrática o mejor).

Solución

a) Podemos representar una solución como una lista ordenada de números donde cada número es en donde se hace un corte. Sea P(i, A) la proposición *Existe una óptimo O tal que A[0:i] = O[0:i]* veamos que para G nuestra solución Greedy:

- P(0, G)
- P(i, G) $\implies$ P(i+1, G)

El primer punto es trivial pues G[0:0] = O[0:0] para cualquier óptimo.
El segundo punto lo probamos del siguiente modo. Si G[i+1] = O[i+1] no hay nada que probar. Es imposible que O[i+1] >G[i+1], pues por factibilidad O[i+1] - O[i] (=G[i]) <=l y en ese caso nuestro greedy no habría tomado el mayor elemnto a distancia <= a l del último. Si suponemos O[i+1] <G[i+1] tenemos por una parte que G[i+1] - O[i](=G[i]) <= l y, si i+1 no es el último elemento, entonces O[i+2] - G[i+1] <= O[i+2] - O[i+1] <= l. Luego, si reemplazamos O[i+1] por G[i+1] tenemos una nueva solución factible con la misma longitud de O (es decir, es óptimo) y que cumple P(i+1, G) Esto prueba la correctitud del algoritmo.

b) Dado un array con los posibles cortes, inicializamos un valor último en 0 y recorremos el array hasta que Array[posición] - último >l; luego guardamos Array[posición - 1] en una lista y actualizamos último a ese mismo valor. Continuamos con la misma lógica hasta recorrer todo el array. El algoritmo tiene claramente complejidad M con M la cantidad de cortes

c) Un contraejemplo puede ser n = 24, l = 11, A = [6, 12, 15, 22] (verlo) Si la lista no nos la dan ordenada, la podemos ordenar en Mlog(M). Luego, en O(M) podemos crear una nueva lista donde el primer elemento sea 0 y el último n (es decir, pegamos 0 a la izquierda de M

y n a la derecha). Finalmente podemos encontrar los cortes corriendo el siguiente código con esa lista de entrada

```
cortes = []
cortesBetina(A):
    if len(A) <= 1 return;
    int maxDistancia = -1;
    int primerCorte = -1;
    int segundoCorte = -1;
    int i = 0;
    int j = 0;
    while j <len(M):
        distancia = A[j] - A[i];
        while j <len(A) and distancia <= l:
                if distancia >maxDistancia:
                    primerCorte = i;
                    segundCcorte = j;
                    maxDistancia = distancia;
        i++;
    if i != A[0] cortes.append(i);
    if j != A[-1] cortes.append(j);
    cortesBetina(M');
    return;
```

M' es una sublista de M que saca los elementos entre i y j inclusive que claramente podemos crear en O(M).
Cada llamado claramente es O(M) y es evidente que no haremos más de M llamados. Luego podemos acotar la complejidad como $O(M^2)$.

Notar que ambos algoritmos asumen que el problema tiene solución. Podemos modificar los algoritmos para que "se den cuentaçuando no hay solución si así fuere necesario o alternativamente podemos chequear en O(M) que no haya 2 cortes adyacentes a distancia $>$ a l.