Resolución primer parcial algo3-2c2024

Santiago Cifuentes

Octubre 2024

1 Multiple choice

1.1 Ejercicio 1

La complejidad del algoritmo propuesto sigue una recurrencia de la forma

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

Usando el teorema maestro con $a = b = 2$ observamos que $O(n^{\log_b a}) = O(n)$ y por lo tanto el costo de cada llamado $f(n) = O(1)$ es estrictamente menor polinomialmente con respecto al costo de los llamados recursivos. Luego, $T(n) = O(n)$.

Por otro lado, el algoritmo es incorrecto debido a que dada una lista de longitud n devuelve n, independientemente del contenido de la lista.

1.2 Ejercicio 2

Veamos contraejemplos para algunas de las estrategias propuestas:

- Elegir la que maximice $w_i - l_i$: la entrada $\{(1, 2), (3, 5)\}$ sirve como contraejemplo.
- Elegir la que maximice w_i: podemos usar el mismo contraejemplo anterior.
- Elegir la que minimice l_i: esta estrategia falla en la entrada $\{(1, 1), (2, 4)\}$.

Probemos que la estrategia de elegir una de las tareas que maximiza w_i/l_i es correcta. Para esto, tomemos un ordenamiento de las tareas como $t_1 \dots t_n$ (de ahora en más notamos w_i y l_i a la penalización y duración de la tarea i-ésima ordenadas de esta forma) y veamos cómo cambia su valor de penalización si swapeamos dos tareas que están consecutivas. Si logramos probar que swapearlas mejora el valor total si y solamente si estaban desordenadas (por el cociente), entonces podremos concluir que la solución óptima debe estar ordenada (recordar `insertion sort`).

Llamemos $T = t_1 \dots t_j t_{j+1} \dots t_n$ al ordenamiento original, y $T' = t_1 \dots t_{j+1} t_j \dots t_n$ al nuevo ordenamiento swappeando estas dos tareas consecutivas. Llamemos c_i al momento en el cual se termina la tarea i de acuerdo al ordenamiento T. Vale que $c_i = \sum_{1 \leq j \leq i} l_i$, y que el costo de la solución T se puede escribir como

$$w(T) = \sum_{i=1}^{n} c_i w_i$$

Notemos que en T' las tareas $t_1 \dots t_{j-1}$ y las tareas $t_{j+2} \dots t_n$ se terminan en el mismo tiempo que en T. Luego, vale que

$$w(T') = \sum_{i=1}^{j-1} c_i w_i + (c_{j-1} + l_{j+1}) w_{j+1} + c_{j+1} w_j + \sum_{i=j+2}^{n} c_i w_i$$

Con estas ecuaciones podemos calcular directamente bajo qué condiciones $w(T') < w(T)$:

$$\begin{aligned} w(T') < w(T) &\iff (c_{j-1} + l_{j+1}) w_{j+1} + c_{j+1} w_j < c_j w_j + c_{j+1} w_{j+1} \\ &\iff l_{j+1} w_j < l_j w_{j+1} \\ &\iff w_j / l_j < w_{j+1} / l_{j+1} \end{aligned}$$

Es decir, este swappeo mejora el costo si las tareas t_j y t_{j+1} no están ordenadas por el cociente entre el factor de penalización y la duración. Por lo tanto, se sigue que una solución óptima **debe estar ordenada** por este cociente. Ergo, tomar en cada paso la tarea que lo maximiza es correcta (recordar `selection sort`).

1.3 Ejercicio 3

Primera parte

La función recursiva propuesta genera un árbol de recursión de profundidad n donde cada nodo realiza a lo sumo 2 llamados. Notemos que si $t = 0$ entonces la condición en el `if` siempre se cumple, y por lo tanto siempre hay dos llamados. En este caso, el árbol de recursión tendrá 2^n hojas.

Por otro lado, la función es incorrecta, dado que imprime todos los conjuntos que se obtienen en las hojas, independientemente de si tienen longitud mayor o igual a k.

La complejidad en peor caso no es $O(2^n)$. Para ver esto, supongamos que $t = 0$ y estudiemos únicamente el trabajo que se realiza en las hojas. Cada una de ellas se procesa en $\Theta(|\texttt{sol}|)$, donde `sol` es la solución que debe imprimirse en la hoja. Luego, la suma de estos costos se puede calcular como

$$\sum_{i=0}^{n} \binom{n}{i} i$$

Esta fórmula tiene una expresión cerrada como $2^{n-1} n$ (la cual puede probarse por inducción), lo cual implica que la complejidad en peor caso es $\Omega(2^{n-1} n) = \Omega(2^n n)$. También se puede acotar la fórmula directamente:

$$\begin{aligned} \sum_{i=0}^{n} \binom{n}{i} i &\geq \sum_{i=n/2}^{n} \binom{n}{i} i \\ &\geq \frac{n}{2} \sum_{i=n/2}^{n} \binom{n}{i} \\ &\geq 2^{n-1} \frac{n}{2} \end{aligned}$$

Segunda parte

La nueva función genera el mismo árbol que la función anterior, dado que la **única** diferencia es que la condición `seleccionado_en_ultimos` se calcula más rápido. Es cierto que la complejidad temporal en la cual se procesaban los nodos internos mejoró (ahora se procesan en $O(1)$, mientras

que antes tomaban $\Theta(t)$), pero vale que las hojas siguen costando $\Theta(|sol|)$, y por lo tanto sigue valiendo la cota inferior a la complejidad en peor caso que derivamos antes.

1.4 Ejercicio 4

Sea F el bosque, y sean $F_1 \dots F_k$ sus componentes conexas, con $n_1 \dots n_k$ sus tamaños. Como cada una es un árbol podemos calcular directamente la suma de los grados de los nodos de F como

$$\begin{aligned}\sum_{v \in V_F} d(v) &= \sum_{i=1}^{k} \sum_{v \in V_{F_i}} d(v) \\ &= \sum_{i=1}^{k} 2(n_i - 1) = 2n - 2k\end{aligned}$$

Por lo tanto, al agregar dos ejes vale que la suma de los grados es $2n - 2k + 4 \leq 2n + 2$.

El segundo inciso nos pide decidir si al agregar dos eje aumenta el diámetro. Esto es claramente falso si los eje que se agregan no conectan dos componentes distintas, ya que en general agregar ejes a un grafo conexo no puede aumentar el diámetro.

Sobre el inciso 3, F' no es fuertemente conexo (en particular puede no ser conexo si $k > 1$).

Sobre el inciso 4, se pueden dibujar árboles tales que al agregar dos aristas se creen más de dos ciclos. Sin ir más lejos, tomando P_5 con nodos ordenados como $v_1 \dots v_5$ y agregando las aristas v_1v_4 y v_2v_5 se obtiene un contraejemplo.

2 Ejercicios a desarrollar

2.1 Octavio y el ski

Inciso a) Definamos, siguiendo la sugerencia, una función recursiva $f(i, j)$ que devuelva el mayor puntaje al que puede llegar Octavio si comienza su recorrido en la posición $(1, N/2)$ y lo termina en (i, j):

$$f(i,j) = \begin{cases} -\infty & j > N \vee j < 1 \\ 0 & i = 1, j = N/2 \\ -\infty & i = 1, j \neq N/2 \\ \max_{-k \leq t \leq k}(f(i-1, j+t)/2) & (i,j) \in H \\ \max(\max_{-k \leq t \leq k}(f(i-1, j+t) + P_{i,j}), 100 + f(i-L, j)) & (i-L, j) \in R \\ \max_{-k \leq t \leq k}(f(i-1, j+t) + P_{i,j}) & \text{caso contrario} \end{cases} \quad (1)$$

El primer caso indica cuando la función está fuera de rango. De forma similar, el tercer caso indica cuando se llegó a la cima, pero sin estar en el centro de la pista. El segundo caso es el caso base satisfactorio, que asumimos vale 0 puntos (es decir, no damos puntaje por el primer casillero).

El cuarto caso se corresponde con la situación donde hay un hueco en la posición (en cuyo caso estamos asumiendo que no se suma el puntaje). El quinto se corresponde con la existencia de una rampa L casilleros más arriba, mientras que el sexto es el caso más simple donde no hay ni rampa ni hueco. Observar que se está asumiendo que no hay un hueco "al final" de ninguna rampa. De todas formas, este caso se puede manejar fácilmente con las mismas ideas.

Para resolver el problema se debe calcular $\max_{1 \leq j \leq N}(f(N, j))$, que siguiendo la interpretación de la función denota el mayor puntaje que se puede tener estando en la base de la montaña.

Esta función recursiva, si se implementa sin memorización, tiene una complejidad de peor caso $\Omega(k^N)$: notemos que al llamar como $f(N, j)$ para cualquier j **siempre** se hacen por lo menos k llamados a posiciones válidas (asumiendo que $k \leq N$). Esto implica que el árbol de recursión contiene al menos un subárbol de tamaño k^N.

Inciso b) Si bien hay $\Omega(k^N)$ llamados, vale que solo hay $O(N^2)$ llamados **distintos**. Por lo tanto, si los memorizamos con una matriz de $N \times N$ solo calcularemos a lo sumo N^2 estados. Para calcular un estado se deben hacer $2k$ llamados recursivos, y se debe verificar si la posición es un hueco o bien si es la posición en donde se aterriza luego de un salto. Esto último se puede hacer en $O(1)$ si precalculamos una matriz booleana que nos permita responder las queries: es decir, armamos una matriz M_H para los huecos tal que $M_H[i, j] = 1$ sii hay un hueco en la posición (i, j), y hacemos lo mismo para las rampas. Con estas estructuras el costo de cada llamado es $O(k)$, y por lo tanto el algoritmo completo de programación dinámica tiene una complejidad $O(kN^2)$, la cual es mejor que el $\Omega(k^N)$ de la versión sin memorización.

Inciso c) Lo escribimos en el pizarrón.

2.2 Rabeirou y sus amigos

Inciso a) Podemos armar un grafo G donde los nodos se coresponden con los amigos de Raibeirou (y él mismo), y donde conectamos dos nodos si los correspondientes amigos son amigos posta. Luego, dos amigos son casuales si están en la misma componente conexa en este grafo.

Notemos que dos amigos v, w son inseparables si y solamente si no hay ningún eje del grafo tal que al removerlo v y w queden en dos componentes distintas. Observar que este eje que los separa debe ser un puente, ya que originalmente v y w estaban en la misma componente conexa. Esto ya nos da un algoritmo para resolver el problema: recorremos los puentes, y por cada puente calculamos qué pares no son inseparables eliminando el puente y calculando componentes conexas. Bien implementado, este algoritmo es $O(mn^2)$.

Un mejor algoritmo se obtiene de la siguiente observación: dos amigos son separables si y solamente si al quitar todos los puentes de G estos dos quedan en componentes conexas distintas. La ida de esta proposición es inmediata de lo dicho anteriormente. Para probar la vuelta, llamamos $C_1, \ldots, C_k$ a las componentes conexas que nos quedaron tras remover los puentes. Notar que los puentes unían nodos de una componente C_i a otra distinta C_j. Si pensamos las componentes como nodos, y ponemos ejes entre dos componentes si había un puente entre ellas, debe valer entonces este grafo nuevo no tiene ciclos, dado que caso contrario habría un ciclo en el grafo original que contiene a una de las aristas puente (contradiciendo que sea puente). De esto se deduce que si dos nodos quedaron en componentes distintas C_i, C_j hay algún puente e tal que todos los caminos de nodos de C_i que van a nodos de C_j deben pasar por e. Ergo, e separa ambos nodos.

Gracias a esta propiedad se concluye que la respuesta al problema debe ser $\sum_{i=1}^{k} \binom{|C_i|}{2}$. Para calcular el tamaño de estas componentes en $O(n + m)$ se pueden calcular los puentes de G en $O(n + m)$, quitarlos del grafo y luego calcular las componentes conexas del grafo resultante junto a su tamaño también en $O(n + m)$.

Inciso b) Notemos que en base al modelado que armamos el problema consiste en decidir cuáles aristas de C hacen que no haya más puentes entre los nodof de de Raibeirou y Oboca.

Trabajemos una vez más con el grafo H de las componentes conexas $C_1, \ldots, C_k$ que armamos antes, y sean C_R y C_O las componentes que tienen respectivamente a Raibeirou y a Obocanegra. Este grafo es un árbol (dijimos que no tenía ciclos), y por lo tanto podemos enraizarlo en C_R, y llamar T al subárbol donde está contenido C_O.

Sea $e \in C$ una arista candidata a agregarse al grafo. Al agregar la arista e debe ocurrir que todos puentes entre C_R y C_O ahora pertenezcan a un ciclo. Esto implica que la arista e tiene que tener un lado en el subárbol por debajo de C_O, y el otro lado por fuera del subárbol T. Caso contrario, alguno de los puentes del grafo original seguirá siendo puente.

Esta idea se puede implementar de muchas formas. Una es calculando explícitamente el grafo de componentes conexas $C_1, \ldots, C_k$, enraizarlo en C_R y luego procesar cada arista en $O(1)$ decidiendo si está o no en cada uno de los subárboles[1]. Otra opción consiste en quitar del grafo todas las aristas puente entre Raibeirou y Oboca, hacer componentes conexas, y ver qué aristas conectan la componente en donde quedó Oboca con la componente donde está Raibeirou.

[1]Observar igualmente que no hace falta calcular explícitamente el grafo, sino lograr identificar a partir de qué puentes comienza la componente C_O, y donde termina la componente C_R.