



1. Dados dos árboles generadores T y R de una gráfica $G = (V, E)$, muestra cómo encontrar la secuencia más corta de árboles generadores T_0, T_1, \dots, T_k tal que $T_0 = T$, $T_k = R$ y cada árbol T_i difiere del árbol anterior T_{i-1} agregando y borrando una arista.
2. Sea G una gráfica cuyas aristas tienen asignados pesos positivos. Sea T un árbol generador de peso mínimo de G . Pruebe que existen aristas $e \in T$ y $e' \notin T$ tales que $T - \{e \cup e'\}$ forman un árbol de peso mayor o igual que T , pero menor o igual a cualquier otro árbol generador de G , i.e. un segundo árbol generador de peso mínimo.
3. Una empresa está planeando una fiesta para sus empleados. Los organizadores de la fiesta quieren que sea una fiesta divertida, por lo que han asignado una calificación de “diversión” a cada empleado. Los empleados están organizados en una estricta jerarquía, es decir, un árbol enraizado en el presidente. Sin embargo, hay una restricción en la lista de invitados a la fiesta: tanto un empleado como su supervisor inmediato (padre en el árbol) no pueden asistir a la fiesta (porque eso no sería divertido). Diseñe un algoritmo de tiempo lineal que haga una lista de invitados para la fiesta y que maximice la suma de las calificaciones de “diversión” de los invitados.

Lo primero que hay que notar es que no hay restricciones sobre la cantidad de hijos que puede tener un nodo en el árbol ni tampoco si el valor de “diversión” es positivo o negativo; así que vamos a hacer unas aclaraciones previas.

Aclaraciones:

- Solo vamos a considerar valores de “diversión” positivos. (el ayudante Adrián autoriza aunque no importa tanto para la solución)
- El árbol puede ser un árbol binario o no.
- Solo vamos a considerar árboles con más de 2 nodos pues si no el problema es trivial. (igualmente se resuelve con el algoritmo propuesto)

La idea central del algoritmo va a ser utilizar el árbol que ya existe, agregarle información y recorrerlo de manera eficiente.

Usando el árbol que ya existe + recorridos:

En este algoritmo, nos interesa empezar a procesar por las hojas del árbol, ya que si un nodo es hoja, no tiene hijos y por lo tanto no tiene restricciones.

Podemos utilizar un recorrido en postorden para recorrer el árbol, de manera que procesamos primero los hijos de un nodo antes de procesar al nodo en sí, este recorrido tiene complejidad $O(n)$. (la implementación específica no nos interesa mas porque ni es un arbol binario xd)

Ahora, en cada nodo vamos a agregarle información, para ello vamos a usar 2 funciones que se pueden describir así:

- **Incluir:** Si incluimos al nodo en la fiesta, entonces sus hijos no pueden ir. Esto se ve así:

$$\text{Incluir}(\text{nodo}) = \text{nodo.diversion} + \sum_{h \in \text{hijos}} \text{excluir}(h)$$

- **Excluir:** Si excluimos al nodo en la fiesta, entonces podemos o no incluir a sus hijos. Esto se ve así:

$$\text{Excluir}(\text{nodo}) = \sum_{h \in \text{hijos}} \max\{\text{incluir}(h), \text{excluir}(h)\}$$

Ademas de estas funciones, vamos a agregar la base de la recurrencia, como ya dijimos cuando un nodo es hoja no tiene restricciones, por lo que podemos definir las funciones así:

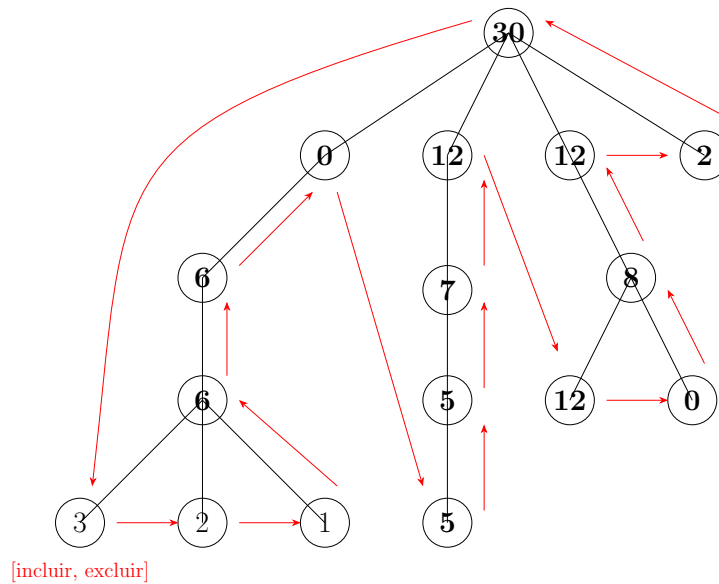
$$\begin{aligned}\text{incluir}(\text{nodo}) &= \text{nodo.diversion} \\ \text{excluir}(\text{nodo}) &= 0\end{aligned}$$

Adicionalmente, si queremos considerar negativos (aunque ya lo resuelve) podemos decir que si un valor es negativo simplemente no lo agregamos. (como en la vida real)

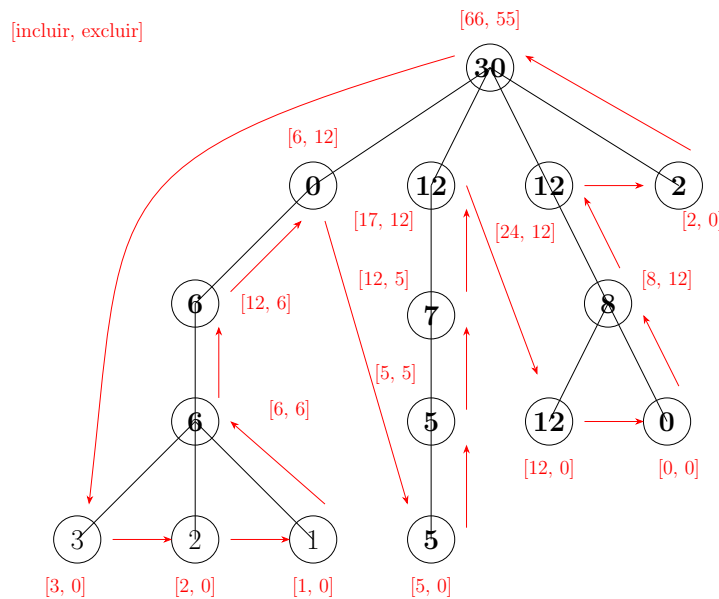
Entonces para cada nodo, empezando por las hojas agregamos esta información y recuperamos la información de sus hijos para poder calcular la información del nodo padre.

Importante: las funciones no llaman a la otra función y así recursivamente si no que llaman al valor que tenemos guardado en el nodo.

Entonces el arbol se va a ir llenando algo así en el recorrido:

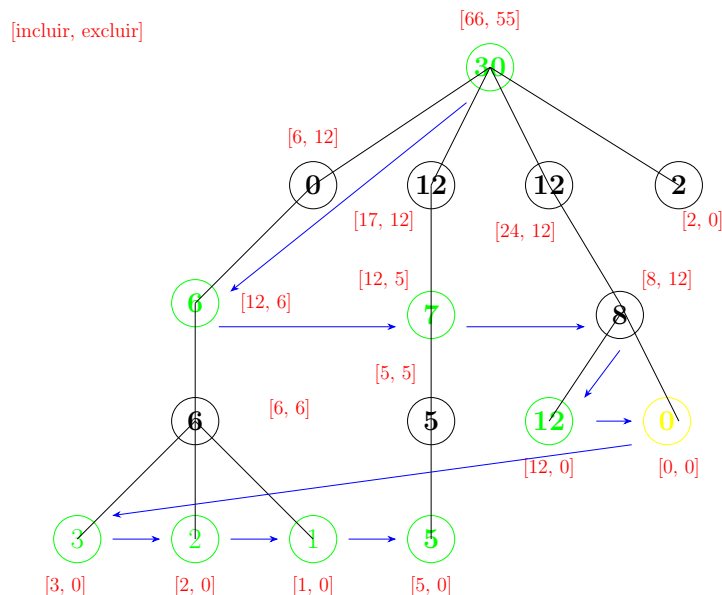


Seguimos el camino rojo y para cada nodo vamos calculando sus 2 valores, esto sucede en a lo mas $O(n)$ pues cada nodo puede tener a lo mas $n-1$ hijos. (las funciones de incluir y excluir tienen que checar a todos sus hijos) pero en el caso de las hojas y en general cuando tienen una cantidad razonable de hijos (hijos "constantes" relativo a n) se hace en $O(1)$.



De aqui ya podemos sacar que la respuesta es el maximo entre incluir y excluir de la raiz; pero el problema nos pide tambien la lista asi que vamos a crearla, para esto vamos a hacer un recorrido con bfs para checar por nivel viendo si en cada nodo nos convino incluirlo o excluirlo, ademas, si un nodo es incluido no tenemos que checar a cada uno de sus hijos, en ese caso solo hay que ir a checar a cada uno de sus nietos, finalmente si un valor queda como que da igual (y no tiene restricciones) podemos incluirlo o excluirlo, esto puede pasar si hay varios caminos o si el numero es 0.

Esto nos puede quedar algo asi:



El camino azul nos dice cuales tenemos que verificar pero hay que recalcar que todos los nodos entran a la fila en algun momento aunque sea para meter a sus hijos, por lo que la complejidad de este paso es $O(n)$ (bfs usualmente es la suma de aristas y vertices pero en arboles tenemos $n-1$ aristas).

Los nodos que agregamos los pinte de verde pero podemos agregarlos a una lista y los que excluyo son o hijos de incluidos o nodos cuyo valor de excluir es mayor que el de incluir y no quedan coloreados (o incluidos).

En total entonces el algoritmo tiene complejidad de recorrer el arbol con un recorrido postorden, agregar informacion por cada nodo mas el recorrerlo con bfs, o lo que es lo mismo $O(n) \cdot O(1) + O(n) = O(n)$ y el espacio adicional es $O(n)$ para guardar la lista de nodos que vamos a incluir. (notemos que en el caso de que un nodo tiene muchos hijos estos a su vez ya no pueden tener muchos hijos).

4. Supongamos que usted quiere marcar un número de n dígitos $\{r_1, r_2, \dots, r_n\}$ en un teléfono normal en el que los números están en un arreglo normal de 4×3 teclas utilizando sólo dos dedos. Supongamos que al comenzar a marcar, sus dedos están en las teclas “*” y “#”. Encuentre un algoritmo de tiempo lineal (programación dinámica) que minimiza la distancia Euclideana que tienen que recorrer sus dedos.

5. Sea S un conjunto de n puntos en el plano y en posición general, tales que $\forall (x_i, y_i) \in S$ se tiene que $x_i, y_i \in \mathbb{N}$ y $x_i, y_i \in [0, \dots, n^2]$. Describe un algoritmo que encuentre el cierre convexo de S es tiempo $O(n)$.

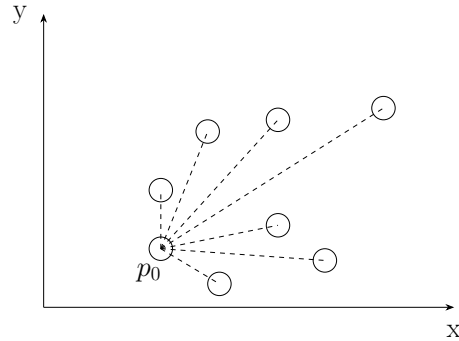
Lo primero a notar es que no podemos aplicar metodos como el de Graham o el de Jarvis, ya que estos tienen una complejidad de $O(n \log n)$ y en este caso se pide un algoritmo de complejidad $O(n)$.

Graham Scan + Radix Sort

Como bien dice el titulo, vamos a usar el algoritmo de Graham Scan para encontrar el cierre convexo de S . Ademas, vamos a usar el algoritmo de Radix Sort para ordenar los puntos de S en tiempo $O(n)$. A continuación se describe el algoritmo:

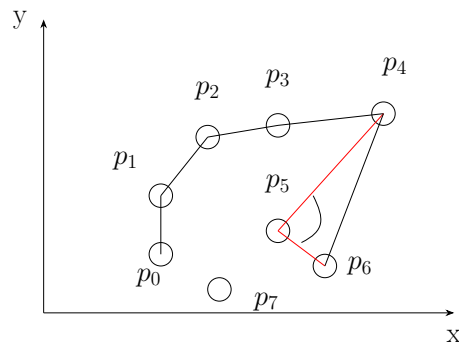
Primero, vamos a buscar el mas chico en la coordenada x, lo vamos a llamar p_0 . Luego, vamos a ordenar los puntos de S en orden decreciente de angulo abarcado entre el segmento que une a p_0 con el punto y el eje x, se puede utilizar la cotangente para agilizar este proceso, ademas por la reestriccion de que los puntos estan en el rango $[0, \dots, n^2]$ podemos usar Radix Sort para ordenar los puntos en tiempo $O(n)$.

Eso nos va a dar algo de este estilo:



Entonces buscar el mas chico en una coordenada nos tomo $O(n)$ y ordenar los puntos nos tomo $O(n)$, por lo que hasta ahora llevamos $O(n)$. Ahora, vamos a aplicar el algoritmo de Graham Scan para encontrar el cierre convexo de S .

Ahora Graham va a tomar una pila, meter a p_0 y a p_1 , luego va a ir tomando los puntos de S de a uno y va a ir viendo si el giro que forma el punto actual con los dos ultimos puntos de la pila es a la izquierda o a la derecha. Si el movimiento es a la derecha entonces continua y mete a el siguiente punto en la pila, si el movimiento es a la izquierda entonces saca el ultimo punto de la pila y vuelve a hacer el giro con el nuevo ultimo punto de la pila. Algo asi:

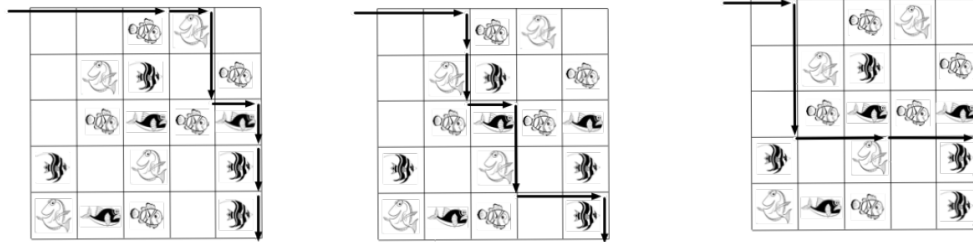


Para calcular si un giro es a la izquierda o a la derecha, se puede usar el producto vectorial, que tiene una complejidad de $O(1)$ $((x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1))$ si es 0 entonces es colineal (este caso no pasa en posicion general), si es positivo es a la izquierda y si es negativo es a la derecha).

Acabamos cuando regresamos a p_0 y la pila tiene n elementos, por lo que la complejidad de Graham Scan es $O(n)$ (porque solo procesa en la pila el elemento una vez). Por lo tanto, la complejidad total del algoritmo es $O(n)$.

□

6. Considera que un río fluye de norte a sur con caudal constante. Suponga que hay n ciudades en ambos lados del río, es decir n ciudades a la izquierda del río y n ciudades a la derecha. Suponga también que dichas ciudades fueron numeradas de 1 a n , pero se desconoce el orden. Construye el mayor número de puentes entre ciudades con el mismo número, tal que dos puentes no se intersecten.
7. Un pescador está sobre un océano rectangular. El valor del pez en el punto (i, j) está dado por un arreglo A de dimensión $2 \times n \times m$. Diseña un algoritmo que calcule el máximo valor de pescado que un pescador puede atrapar en un camino desde la esquina superior izquierda a la esquina inferior derecha. El pescador solo puede moverse hacia abajo o hacia la derecha, como se ilustra en la siguiente figura.



8. Sean tres cadenas de caracteres X, Y y Z , con $|X| = n$, $|Y| = m$ y $|Z| = n + m$. Diremos que Z es un *shuffle* de X y Y si Z puede ser formado por caracteres intercalados de X y Y manteniendo el orden de izquierda a derecha de cada cadena.

- (a) Muestra que *cchocohilaptes* es un *shuffle* de *chocolate* y *chips*, pero *chocochilatspe* no lo es.

Primero voy a solucionar el siguiente inciso:

- (b) Diseña un algoritmo de programación dinámica eficiente que determine si Z es un *shuffle* de X y Y . *Hint*: Los valores de la matriz de programación dinámica que construyas, podrían ser valores booleanos y no numéricos.

El problema se parece bastante a la subcadena creciente mas grande. La idea es usar una matriz de $|X| + 1$ filas y $|Y| + 1$ columnas, donde la celda (i, j) indica si los primeros $i + j$ caracteres de Z son un *shuffle* de los primeros j caracteres de X y los primeros i caracteres de Y . (podemos cambiar cual es quien pero asi lo hice)

Usando matriz y programación dinámica:

Podemos empezar verificando que $|X| + |Y| = |Z|$, si no es así entonces no puede ser un *shuffle* de X y Y .

Tenemos entonces que nuestra matriz se va a ver algo así:

		x_0	x_1	\dots	x_n
i/j		0	1	\dots	n
	0	1			
y_0	1				
y_1	2				
\dots	\dots				
y_m	m				

Ahora vamos a ver como llenarla, primero por la definición que hicimos, el índice (i,j) sera 1 pues los primeros 0+0 caracteres de Z siempre serán shuffle de los primeros 0 caracteres de X y los primeros 0 caracteres de Y .

Para la primera fila, hay que checar que los primeros j caracteres de X sean iguales a los primeros j caracteres de Z , si es así, entonces la celda $(0, j)$ sera 1, si no entonces sera 0; para checar esto podemos checar si el caracter j es igual en ambos y después checar si a la izquierda ya tenemos un 1. (esto es equivalente a: $M[0][j]=M[0][j-1]$ AND $X[j-1]=Z[j-1]$ tenemos que restar 1 porque las cadenas tienen índice en 0 pero también lo puedes entender con cantidad de caracteres)

Para la primera columna, es lo mismo que la fila pero con Y y Z , esto se puede entender como $M[i][0]=M[i-1][0]$ AND $Y[i-1]=Z[i-1]$.

Ahora para el caso de en medio hay que checar que los primeros $i + j$ caracteres de Z sean shuffle de los primeros j caracteres de X y los primeros i caracteres de Y , esto se puede entender como $M[i][j]=M[i-1][j]$ AND $Y[i-1]=Z[i+j-1]$ OR $M[i][j-1]$ AND $X[j-1]=Z[i+j-1]$.

De manera general nuestra función quedaría algo así:

$$M[i][j] = \begin{cases} 1 & \text{si } i = 0 \text{ y } j = 0 \\ (M[i-1][j] \text{ AND } Y[i-1] == Z[i+j-1]) \text{ OR } (M[i][j-1] \text{ AND } X[j-1] == Z[i+j-1]) & \text{si } i > 0 \text{ y } j > 0 \end{cases}$$

Al final sabremos si Z es un *shuffle* de X y Y si $M[|Y|][|X|] = 1$, además el camino para ir de la celda $(|Y|, |X|)$ a la celda $(0, 0)$ nos dirá cuáles son los caracteres que se usaron.

Este algoritmo tiene complejidad $O(|X| * |Y|)$ y usa $O(|X| * |Y|)$ memoria, esto pues la matriz es de tamaño $(|X| + 1) * (|Y| + 1)$ y se llena en cada celda una vez tomando $O(1)$ tiempo (chechar arriba y abajo, tomar un índice de una cadena y compararlo con el índice en otra cadena que puede ser $O(1)$).

Para entender mejor checar el ejemplo de arriba.

