Universidad Nacional Autónoma de México
Facultad de Ciencias
Análisis de Algoritmos | 7083
Tarea 3 : | Programacion Dinamica
Sosa Romo Juan Mario | 320051926
25/09/24

1. **Dados dos árboles generadores T y R de una gráfica $G = (V, E)$, muestra cḿo encontrar la secuencia más corta de árboles generadores $T_0, T_1, \ldots, T_k$ 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; asi que vamos a ahcer unas aclaraciones previas.

   Aclaraciones:

   - Solo vamos a considerar valores de "diversión" positivos. (el ayudante Adrián autoriza aunque no ipmorta tanto para la solución)
   - El arbol puede ser un árbol binario o no.
   - Solo vamos a considerar arboles con mas 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 especifica 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 asi:

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

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

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

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

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 asi:

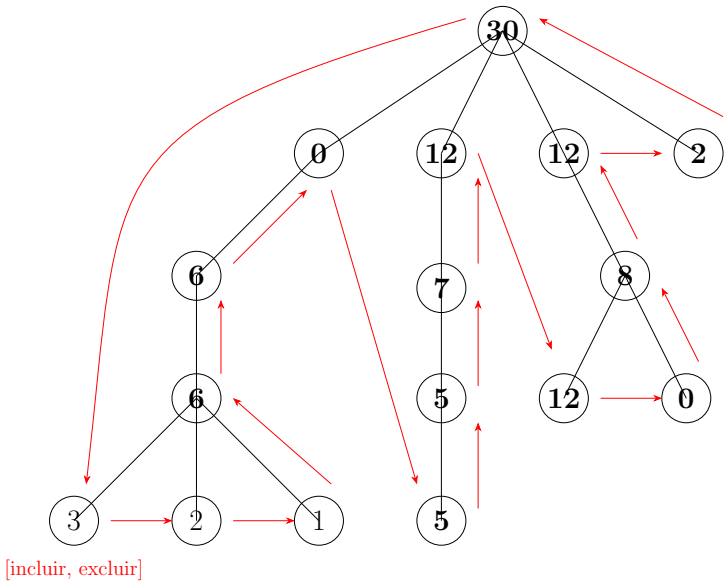
$$\text{incluir}(nodo) = nodo.diversion$$
$$\text{excluir}(nodo) = 0$$

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)

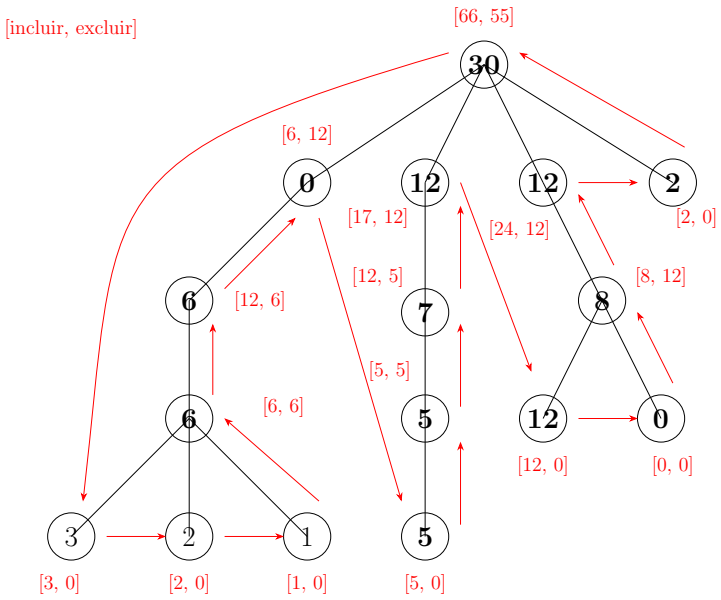
Entonces para cada nodo, empezando por las hojas agregamos esta información y recuperamos la información de sus hijos para poder calcular la información del nodo padre.

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

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

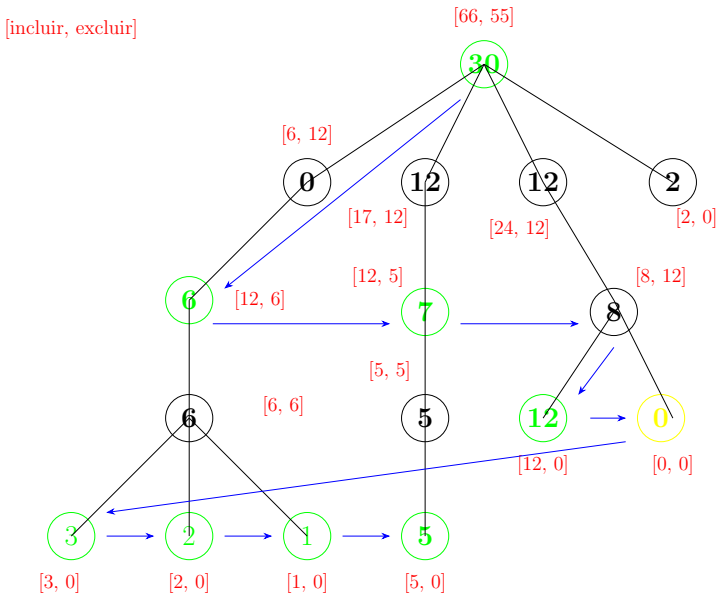


Seguimos el camino rojo y para cada nodo vamos calculando sus 2 valores, esto sucede en a lo mas O(n) pues cada nodo puede tener a lo mas n-1 hijos. (las funciones de incluir y excluir tienen que checar a todos sus hijos) pero en el caso de las hojas y en general cuando tienen una cantidad razonable 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 exlcuirlo, 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)*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 $\{r_1, r_2, \ldots, r_n\}$ 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 $x_i, y_i \in [0, \ldots, n^2]$. Describe un algoritmo que encuentre el cierre convexo de S es tiempo $O(n)$.**

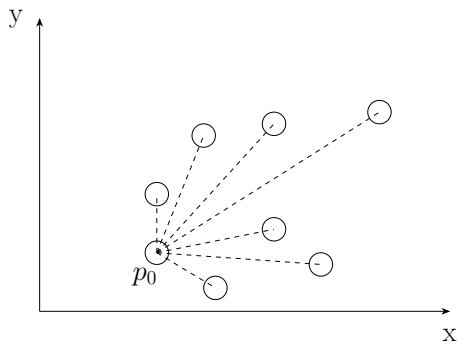
   Lo primero a notar es que no podemos aplicar metodos como el de Graham o el de Jarvis, ya que estos tienen una complejidad de $O(n \log n)$ y en este caso se pide un algoritmo de complejidad $O(n)$.

   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:

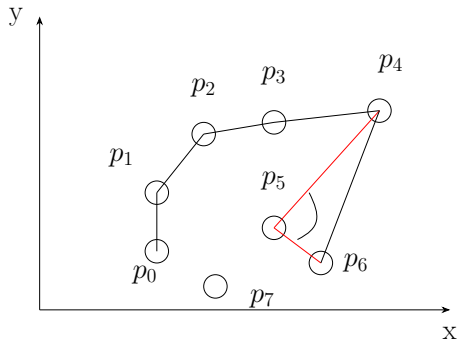
Primero, vamos a buscar el mas chico en la coordenada x, lo vamos a llamar p_0. Luego, vamos a ordenar los puntos de S en orden decreciente de angulo abarcado entre el segmento que une a p_0 con el punto y el eje x, se puede utilizar la cotangente para agilizar este proceso, ademas por la reestriccion de que los puntos estan en el rango $[0, \ldots, n^2]$ 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 $O(1)$ $((x2-x1)(y3-y1)-(y2-y1)(x3-x1)$ 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 $O(n)$.

□

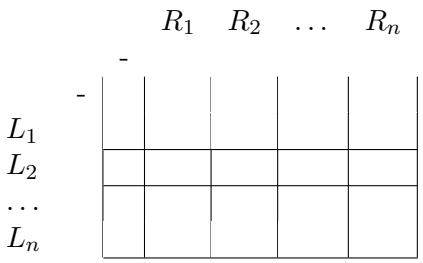
6. **Considera que un río fluye de norte a sur con caudal constante. Suponga que hay n ciudades en ambos lados del río, es decir n ciudades a la izquierda del río y n ciudades a la derecha. Suponga también que dichas ciudades fueron numeradas de 1 a n, pero se desconoce el orden. Construye el mayor número de puentes entre ciudades con el mismo número, tal que dos puentes no se intersecten.**

Como ya vimos con la profesora, este problema es similar al problema de encontrar la subsecuencia creciente mas larga.

Programacion dinamica parecida a la de la subsecuencia creciente mas larga.

Comenzamos caraterizando el problema, en nuestra solucion tenemos que si conectamos dos ciudades con un puente, entonces no podemos conectar una ciudad anterior de un lado con una posterior del otro lado respecto al puente que conecta las dos ciudades, esto significa que podemos ir procesando siempre para adelante, vamos a definir una funcion.

Vamos a utilizar un arreglo de (n+1)(n+1) para guardar la subsecuencia comun mas larga, donde la posicion (i, j) guarda la cantidad de puentes que se pueden construir entre las ciudades de la izquierda hasta la ciudad i y las ciudades de la derecha hasta la ciudad j, esto se puede ver algo asi:



Ahora para llenar la matriz, vamos a utilizar la siguiente funcion de recurrencia:

$$\text{puentes}[i][j] = \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0 \\ puentes[i-1][j-1] + 1 & \text{si } L[i] = R[j] \\ \max(puentes[i-1][j], puentes[i][j-1]) & \text{si } L[i] \neq R[j] \end{cases}$$

Donde L y R son los arreglos de las ciudades de la izquierda y de la derecha respectivamente. La maxima cantidad de puentes entre ciudades del mismo numero sin intersecciones se vera en la entrada [n+1,n+1]. Este algoritmo tiene complejidad $O(n^2)$ pues cada celda de la matriz se llena en tiempo constante y complejidad espacial $O(n^2)$ pues la matriz tiene dimensiones $n+1$ por $n+1$.

Para entenderlo mejor veamos un ejemplo:

			R_1	R_2	R_3	R_4	R_5
		-	4	1	2	3	5
	-	*0*	*0*	*0*	*0*	*0*	0
L_1	5	*0*					
L_2	1	0					
L_3	3	0					
L_4	2	*0*					
L_5	4	0					

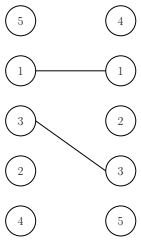
Usando el primer caso llenamos de 0 la primera fila y la primera columna, luego usamos la recurrencia para llenar el resto de la matriz.

			R_1	R_2	R_3	R_4	R_5
		-	*4*	*1*	*2*	*3*	*5*
	-	*0*	*0*	*0*	*0*	*0*	*0*
L_1	*5*	*0*	*0*	*0*	*0*	*0*	*1*
L_2	*1*	*0*	*0*	*1*	*1*	*1*	*1*
L_3	*3*	*0*	*0*	*1*	*1*	*2*	*2*
L_4	*2*	*0*	*0*	*1*	*2*	*2*	*2*
L_5	*4*	*0*	*1*	*1*	*2*	*2*	*2*

Vemos que la maxima cantidad de puentes que se pueden construir en este caso son 2, para recuperar el camino empezamos desde la esquina inferior derecha, y checamos, si los valores de las ciudades coinciden entonces son parte de la solucion, la agregamos a una lista de solucion y nos movemos a la diagonal superior izquierda, si no coinciden entonces, nos movemos al mayor entre el de arriba y el de la izquierda, si son iguales nos movemos al de arriba.

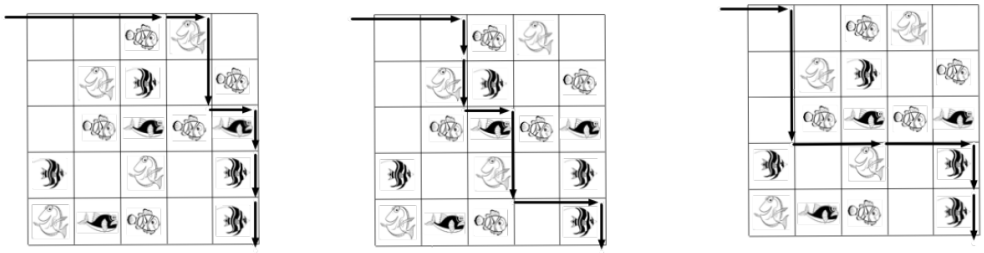
			R_1	R_2	R_3	R_4	R_5
		-	*4*	*1*	*2*	*3*	*5*
	-	*0*	*0*	*0*	*0*	*0*	*0*
L_1	*5*	*0*	*0*	*0*	*0*	*0*	*1*
L_2	*1*	*0*	*0*	*1*	*1*	*1*	*1*
L_3	*3*	*0*	*0*	*1*	*1*	*2*	*2*
L_4	*2*	*0*	*0*	*1*	*2*	*2*	*2*
L_5	*4*	*0*	*1*	*1*	*2*	*2*	*2*

En este caso la solucion es unir L_3, R_4 pues ambos tienen la ciudad numero 3 y unir L_2, R_2 pues ambas tienen la ciudad numero 1, y la cantidad de puentes es 2, esto se ve algo asi:



A notar que podemos dejar de buscar puentes en cuanto lleguemos a un 0 en la matriz, pues no podremos encontrar mas puentes.

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 $n \times m$. 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.**



Como en basicamente toda la tarea, vamos a usar programacion dinamica. El punto es calcular el maximo valor de pescado que se puede atrapar en un camino desde la esquina superior izquierda a la esquina inferior derecha. Ademas, solo vamos a considerar los movimientos hacia abajo y hacia la derecha. (uno a la vez).

Nota: Voy a asumir que la matriz tiene al menos una fila y una columna, si no, no tiene sentido, ademas no voy a considerar que un pez pueda tener valor negativo aunque me parece no afecta al algoritmo.

Programación dinámica:

De hecho este problema se parece bastante a la subcadena mas larga, en este caso ya tenemos el arreglo auxiliar y le vamos a meter informacion a las celdas, lo podemos hacer poniendole un valor de valorM o algo por el estilo. (podemos usar otra matriz con las mismas dimensiones tambien)

Empezando por caracterizar el problema, vemos que el pescador solo puede llegar a la celda (i,j) si antes estuvo en la celda (i-1,j) o en la celda (i,j-1). Por lo tanto, el valor en la celda (i,j) se puede entender como el valor del pez en esa celda mas el maximo valor entre las 2 que puede llegar. Ademas, el pescador siempre empieza en la celda (0,0) por tanto esta celda solo sera el valor del pez que haya o no ahi.

Entonces la funcion de recurrencia es la siguiente: (estamos guardando valorM por celda)

$$valorM[i][j] = \begin{cases} valorPez[0,0] & \text{si } i = 0 \text{ y } j = 0 \\ valorM[0][j-1] + valorPez[0,j] & \text{si } i = 0 \\ valorM[i-1][0] + valorPez[i,0] & \text{si } j = 0 \\ max(valorM[i-1][j], valorM[i][j-1]) + valorPez[i,j] & \text{si } i > 0 \text{ y } j > 0 \end{cases}$$

Entonces comenzamos a llenar la matriz, primero la celda (0,0) sera el valor del pez en esa celda. Despues llenamos la primera fila y la primera columna con la recurrencia que

pusimos arriba. Finalmente llenamos el resto de la matriz con la recurrencia y al final el valor que buscamos es el valor en la celda (n-1,m-1).

Veamos un ejemplo:

[5,]*	*[0,*]*	*[0,*]*	*[8,*]*	*[4,*]*	*[2,*]*
[1,]*	*[2,*]*	*[3,*]*	*[0,*]*	*[0,*]*	*[7,*]*
[5,]*	*[1,*]*	*[0,*]*	*[0,*]*	*[3,*]*	*[6,*]*
[6,]*	*[0,*]*	*[0,*]*	*[9,*]*	*[1,*]*	*[1,*]*

Aqui, comenzamos con un arreglo A de dimension 6x4, cada celda puede o no tener un pez con un valor asociado, ademas, simbolice [valorPez,valorM] siendo que no hemos calculado ningun valorM, esta como * para indicar que no lo hemos calculado.

Empezamos sabiendo que el valor en la celda (0,0) es 5, y de ahi usamos los 2 primeros casos para llenar la primera fila y la primera columna.

[5,5]	*[0,5]*	*[0,5]*	*[8,13]*	*[4,17]*	*[2,19]*
[1,6]	*[2,*]*	*[3,*]*	*[0,*]*	*[0,*]*	*[7,*]*
[5,11]	*[1,*]*	*[0,*]*	*[0,*]*	*[3,*]*	*[6,*]*
[6,17]	*[0,*]*	*[0,*]*	*[9,*]*	*[1,*]*	*[1,*]*

Ahora toca lo interesante, utilizar el tercer caso para llenar el resto de la matriz.

[5,5]	*[0,5]*	*[0,5]*	*[8,13]*	*[4,17]*	*[2,19]*
[1,6]	*[2,8]*	*[3,11]*	*[0,13]*	*[0,17]*	*[7,26]*
[5,11]	*[1,12]*	*[0,12]*	*[0,13]*	*[3,20]*	*[6,32]*
[6,17]	*[0,17]*	*[0,17]*	*[9,26]*	*[1,27]*	*[1,33]*

Realmente el proceso aun manual es bastante facil solo es sumar el valor del pez en esa posicion mas el maximo valor que puede llevar de arriba o de la izquierda, en este caso el maximo total que puede conseguir es 33, y mas aun podemos recuperar el camino viendo desde que lado vino:

[5,5]	*[0,5]*	*[0,5]*	*[8,13]*	*[4,17]*	*[2,19]*
[1,6]	*[2,8]*	*[3,11]*	*[0,13]*	*[0,17]*	*[7,26]*
[5,11]	*[1,12]*	*[0,12]*	*[0,13]*	*[3,20]*	*[6,32]*
[6,17]	*[0,17]*	*[0,17]*	*[9,26]*	*[1,27]*	*[1,33]*

La complejidad de este algoritmo es de $O(n*m)$ y usa $O(n*m)$ memoria (es equivalente agregar un valor por cada valor a usar otro arreglo pero asi se ve mas bonito uwu), esto pues la matriz es de tamaño $n*m$ y se llena en cada celda una vez tomando $O(1)$ tiempo (checar arriba y abajo, tomar un indice de un arreglo).

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.**

Para entender esta solución, primero checar el ejercicio 8.b, voy a usar a chocolate como X y a chips como Y.

Usando el algoritmo de 8.b:

Primero checamos que $|X| + |Y| = |Z|$, si no es asi entonces no puede ser un *shuffle* de X y Y, en este caso $|9| + |5| = |14|$ entonces continuamos, creamos nuestra tablita:

			0	1	2	3	4	5	6	7	8
			c	h	o	c	o	l	a	t	e
		i/j	0	1	2	3	4	5	6	7	8	9
		0	1									
0	c	1										
1	h	2										
2	i	3										
3	p	4										
4	s	5										

La tabla que guardamos es unicamente lo marcado con recuadros, lo que puse justo afuera es los indices, en la horizontal van los i y en la vertical los j, ademas inclui las cadenas que estamos comparando en sus lugares con sus indices para que sea mas facil de entender.

Usamos la funcion que definimos en el ejercicio 8.b, vamos a empezar por llenar la primera fila:

			0	1	2	3	4	5	6	7	8
			c	h	o	c	o	l	a	t	e
		i/j	0	1	2	3	4	5	6	7	8	9
		0	1	1	0	0	0	0	0	0	0	0
0	c	1										
1	h	2										
2	i	3										
3	p	4										
4	s	5										

Vemos que la (0,1) se llena con un 1 pues cumple que M[0][0] AND X[0]=c==c=Z[0], despues la siguiente checa la h y la compara con la c de Z y como no son iguales ahora el resto de la fila se llena con 0 pues no hay nada mas que comparar.

Algo interesante es que si checa arriba y a la izquierda y tiene 0s entonces no tiene que verificar nada mas para esa celda.

Podemos seguir llenando por filas pero me gusta mas llenar primero la primera columna para que se vea mas bonito:

				0	1	2	3	4	5	6	7	8
				c	h	o	c	o	l	a	t	e
		i/j	0	1	2	3	4	5	6	7	8	9
		0	1	1	0	0	0	0	0	0	0	0
0	c	1	1									
1	h	2	0									
2	i	3	0									
3	p	4	0									
4	s	5	0									

Aqui igual la unica que tiene un 1 es la primera c pues igual empieza como nuestra Z pero las demas no, ahora podemos llenar el resto de la tabla fila a fila:

				0	1	2	3	4	5	6	7	8
				c	h	o	c	o	l	a	t	e
		i/j	0	1	2	3	4	5	6	7	8	9
		0	1	1	0	0	0	0	0	0	0	0
0	c	1	1	1	1	1	1	1	0	0	0	0
1	h	2	0									
2	i	3	0									
3	p	4	0									
4	s	5	0									

Podemos ver que si tiene un uno arriba y uno a la izquierda aun se tiene que usar la funcion, en este caso vemos como podemos empezar con cualquiera de las 2 'c' (hasta ahora) y despues seguir por X hasta llegar a la 'l' que nuestra Z quiere una 'h' entonces ya no encaja y ponemos 0s.

				0	1	2	3	4	5	6	7	8
				c	h	o	c	o	l	a	t	e
		i/j	0	1	2	3	4	5	6	7	8	9
		0	1	1	0	0	0	0	0	0	0	0
0	c	1	1	1	1	1	1	1	0	0	0	0
1	h	2	0	1	0	0	0	1	0	0	0	0
2	i	3	0	0	0	0	0	1	1	1	0	0
3	p	4	0	0	0	0	0	0	0	1	1	1
4	s	5	0	0	0	0	0	0	0	0	0	1

Vemos que la ultima celda tiene un 1 entonces si es un *shuffle* de X y Y; para verificar vamos a checar el camino y ver si tiene sentido:

			0	1	2	3	4	5	6	7	8
				c	h	o	c	o	l	a	t	e
		i/j	0	1	2	3	4	5	6	7	8	9
		0	1	1	0	0	0	0	0	0	0	0
0	c	1	1	1	1	1	1	1	0	0	0	0
1	h	2	0	1	0	0	0	1	0	0	0	0
2	i	3	0	0	0	0	0	1	1	1	0	0
3	p	4	0	0	0	0	0	0	0	1	1	1
4	s	5	0	0	0	0	0	0	0	0	0	1

Entonces seguimos el caminito amarillo, (si es verde da igual cual tomemos) y checamos, quitamos una por una letra de Z segun el caminito y quitamos una de X o de Y :

cchocohilaptes	chocolate	chips
chocohilaptes	chocolate	hips
hocohilaptes	hocolate	hips
hilaptes	late	hips
laptes	late	ps
ptes	te	ps
tes	te	s
es	e	s
s		s

Entonces si es un *shuffle* de X y Y. (quitamos en orden letras de X y Y y pudimos eliminar Z en orden)

Probando que chocochilatspe no es *shuffle*:

Literalmente vamos a aplicar exactamente lo mismo asi que solo me voy a saltar a tener la tablita completa y ver si llegamos a un 1 en la ultima celda:

			0	1	2	3	4	5	6	7	8
				c	h	o	c	o	l	a	t	e
		i/j	0	1	2	3	4	5	6	7	8	9
		0	1	1	1	1	1	1	0	0	0	0
0	c	1	1	0	0	1	0	1	0	0	0	0
1	h	2	1	0	0	0	0	1	0	0	0	0
2	i	3	0	0	0	0	0	1	1	1	1	0
3	p	4	0	0	0	0	0	0	0	0	0	0
4	s	5	0	0	0	0	0	0	0	0	0	0

Aqui es claro que no hay un 1 en la ultima celda, de hecho el caminito toma el primer "choco" de la cadena X luego "chi" de Y, luego "lat" de X pero busca una "s" pero como no ha pasado la p de chips entonces no puede acceder a la p y por tanto no es un *shuffle* de X y Y.

Veamoslo con el otro metodo:

chocochilatspe chocolate chips
chilatspe late chips
latspe late ps
spe e ps

(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 asi 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	...	x_n
	i/j	0	1	2	...	n
	0	1				
y_0	1					
y_1	2					
...	...					
y_m	m					

Ahora vamos a ver como llenarla, primero por la definición que hicimos, el indice (i,j) sera 1 pues los primeros 0+0 caracteres de Z siempre seran 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 asi, 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 despues 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 indice en 0 pero tambien 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 funcion quedaria algo asi:

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

Al final sabremos si Z es un *shuffle* de X y Y si $M[|Y|][|X|] = 1$, ademas el camino para ir de la celda $(|Y|, |X|)$ a la celda $(0, 0)$ nos dira cuales 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 (checar arriba y abajo, tomar un indice de una cadena y compararlo con el indice en otra cadena que puede ser O(1)).

Para entender mejor checar el ejemplo de arriba.