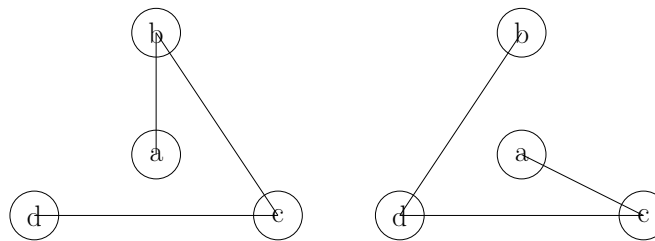




1. Dados dos árboles generadores T y R de una gráfica $G = (V, E)$, muestra cómo encontrar la secuencia más corta de árboles generadores T_0, T_1, \dots, T_k tal que $T_0 = T$, $T_k = R$ y cada árbol T_i difiere del árbol anterior T_{i-1} agregando y borrando una arista.

Primero que nada aclarar que como estamos considerando que son arboles generadores de una grafica general, el isomorfismo no vale, es decir, si 2 arboles son isomorfos no necesariamente son arboles generadores iguales, en particular digamos:



Estos 2 arboles se pueden comprobar como isomorfos, pero no son arboles generadores iguales, ya que la lista de adyacencias de la grafica es diferente. (o al menos eso me hace mas sentido xd igual ver si arboles son isomorfos cuesta $O(|V|)$ asi que se podria hacer antes del siguiente algoritmo)

Ahora si la secuencia de árboles generadores la vamos a hacer de la siguiente manera:

Diferencia simetrica:

Primero vamos a encontrar las diferencias entre T y R , identificamos las aristas que están en T y no en R , y las que están en R y no en T .

Sea E_T el conjunto de aristas de T y E_R el conjunto de aristas de R . Entonces, definimos a $Extras = E_T \setminus E_R$ (aristas que están en T y no en R) y a $Faltantes = E_R \setminus E_T$ (aristas que están en R y no en T).

Es evidente que vamos a realizar primero agregar uno de $Faltantes$ crear un ciclo y despues vamos a tener que hacer un corte de alguno de $Extras$ en T_0 para obtener un árbol generador de G , T_1 .

Este proceso lo vamos a hacer hasta que $Extras$ y $Faltantes$ sean vacíos, es decir, hasta que $T_k = R$; pero, lo importante radica en como saber que al hacer cada una de estas operaciones siga siendo arbol.

Bueno primero como estamos agregando una arista aqui solo podemos violar el agregar un ciclo y de hecho como es arbol siempre vamos a agregar un ciclo, entonces para que

siga siendo árbol, vamos a tener que cortar una arista de *Extras* que forme parte del ciclo que acabamos de agregar.

Ahora, como R es árbol (no contiene a este ciclo) podemos garantizar, que habrá una arista diferente a la que acabamos de agregar que forme parte del ciclo y que además este en *Extras* por tanto empezando desde un extremo de la arista que acabamos de agregar, vamos a buscar a esta otra arista y la vamos a cortar; (tenemos que verificar el ciclo es decir llegar al otro extremo de nuestra arista que agregamos, toma $O(|V|)$)

Alternativamente se puede pensar como, desconectar, crear 2 componentes conexas y de esas ver una arista de *Faltantes* que conecte las 2 componentes conexas, esta arista es la que vamos a agregar pero no vamos a ir por este camino.

En el peor de los casos, la diferencia simétrica entre los árboles es del orden de $O(|V|)$, por tanto el algoritmo tiene una complejidad de $O(|V|^2)$, se podría mejorar la complejidad si vamos usando alguna EDD para checar los ciclos.

Afortunadamente aquí piden que el algoritmo nos regrese la secuencia de árboles generadores mínima, sabemos que esto se va a cumplir pues en cada paso estamos acercándonos en 2 aristas a R , que es lo máximo que se puede acercar un árbol a otro, quitando y poniendo solo 1 arista por turno, es decir, si hubiera un camino más corto, este tendría que acercarse a R más rápido que 2 aristas por turno, lo cual no es posible.

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 más porque ni es un árbol 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:

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

Además 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:

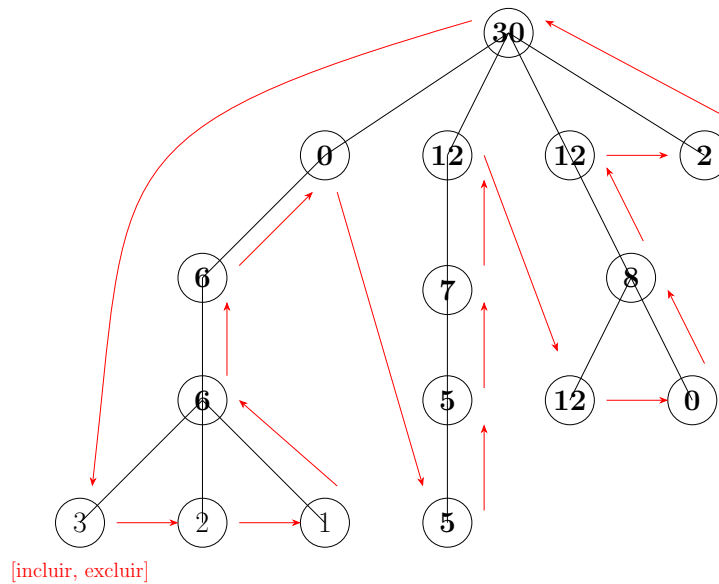
$$\begin{aligned}\text{incluir}(\text{nodo}) &= \text{nodo.diversion} \\ \text{excluir}(\text{nodo}) &= 0\end{aligned}$$

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)

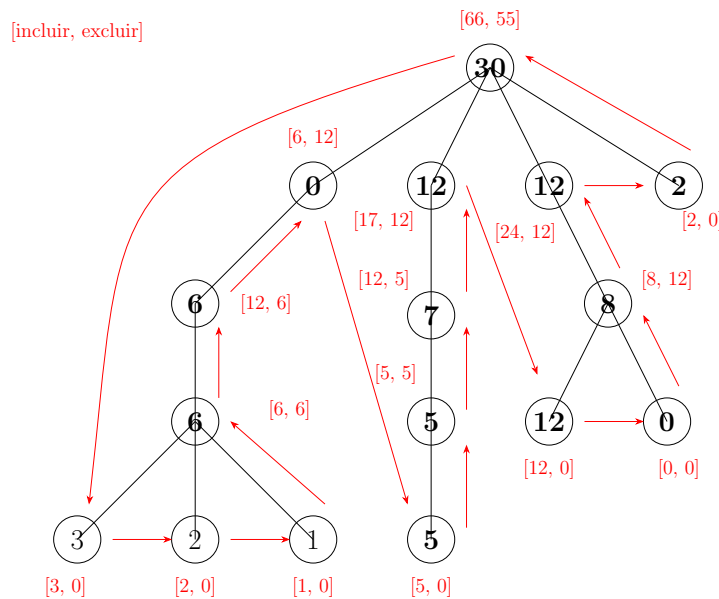
Entonces para cada nodo, empezando por las hojas agregamos esta información y recuperamos la información de sus hijos para poder calcular la información del nodo padre.

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 árbol 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:

- (0,1) para 2
- (0,2) para 3
- (1,0) para 4
- (1,1) para 5
- (1,2) para 6
- (2,0) para 7
- (2,1) para 8
- (2,2) para 9
- (3,0) para *
- (3,1) para 0
- (3,2) para #

Ademas de esto, la distancia euclideana entre dos teclas $t_1 = (x_1, y_1)$ y $t_2 = (x_2, y_2)$ es :

$$dist(t_1, t_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

En la ayudantia se dijo algo sobre no poder saltar por numeros pero no veo porque no se podria en base a lo que dice el problema.

El algoritmo que voy a presentar es increiblemente malo para numeros chicos pero para numeros grandes funciona muy bien ($O(n)$ digitos).

Programacion dinamica con matriz 3d:

Suena un poco raro pero basicamente queremos 3 cosas, el calculo del costo minimo hasta el digito i , la posicion del dedo 1 digamos j y la posicion del dedo 2 digamos k . Entonces vamos a hacer una matriz 3d, digamosle $M[i][j][k]$, sabemos que i va de 1 a n , j y k usan la represntacion de las teclas que dije antes. Entonces $M[i][j][k]$ va a ser el costo minimo de marcar los primeros i digitos con el dedo 1 en la tecla j y el dedo 2 en la tecla k .

Esta funcion se ve asi:

$$M[i][j][k] = \begin{cases} 0 & \text{si } i = 0 \\ \min\{M[i-1][j'][k] + dist(j', r_i), M[i-1][j][k'] + dist(k', r_i)\} & \text{si } i > 0 \end{cases}$$

Aclaraciones sobre la funcion, el estado inicial de j es (3,0) y el de k es (3,2), por lo que cuando estamos en el digito $i=0$, estos son sus valores, a la hora de calcular el digito actual, vamos a ver cual es el minimo entre el costo de mover el dedo 1, que estaba en un estado j' y el dedo 2 en k , y el costo de mover el dedo 2, que estaba en un estado k' y el dedo 1 en j . El digito actual i que estamos procesando es r_i , una vez movemos el dedo a esa posicion ese sera su nuevo estado.

Al final nuestro resultado va a ser el minimo entre $M[n][j][k]$ para todo j y k .

En escencia lo que esta pasando es que tenemos 12 estados posibles por dedo y vamos a ir calculando el costo minimo de marcar los primeros i digitos moviendo los dedos a otros estados, como tenemos n digitos y hay 12 estados posibles por dedo, la complejidad es $O(n*12*12) = O(n)$.

-
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 en tiempo $O(n)$.

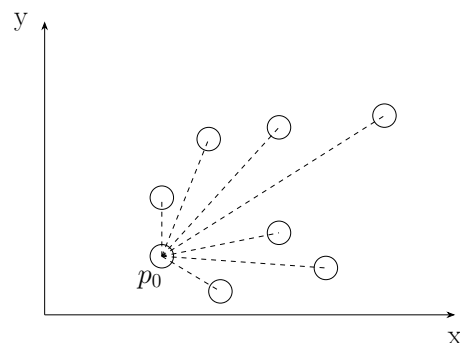
Lo primero a notar es que no podemos aplicar metodos como el de Graham o el de Jarvis, ya que estos tienen una complejidad de $O(n \log n)$ y en este caso se pide un algoritmo de complejidad $O(n)$.

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:

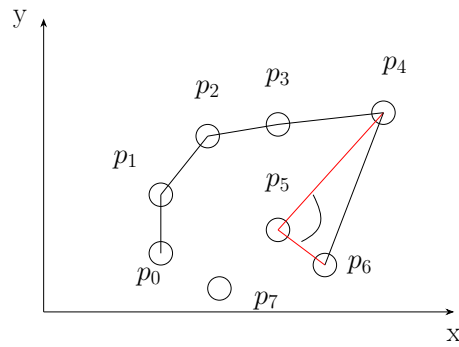
Primero, vamos a buscar el mas chico en la coordenada x, lo vamos a llamar p_0 . Luego, vamos a ordenar los puntos de S en orden decreciente de angulo abarcado entre el segmento que une a p_0 con el punto y el eje x, se puede utilizar la cotangente para agilizar este proceso, ademas por la reestriccion 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 posición 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.

Programación dinámica parecida a la de la subsecuencia creciente mas larga.

Comenzamos caracterizando el problema, en nuestra solución 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 función.

Vamos a utilizar un arreglo de $(n+1)(n+1)$ para guardar la subsecuencia común mas larga, donde la posición (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 así:

	R_1	R_2	\dots	R_n
L_1				
L_2				
\dots				
L_n				

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

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

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
L_1	5	0	0	0	0	0
L_2	1	0				
L_3	3	0				
L_4	2	0				
L_5	4	0				

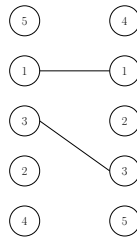
Usando el primer caso llenamos de 0 la primera fila y la primera columna, luego usamos la recurrencia para llenar el resto de la matriz.

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

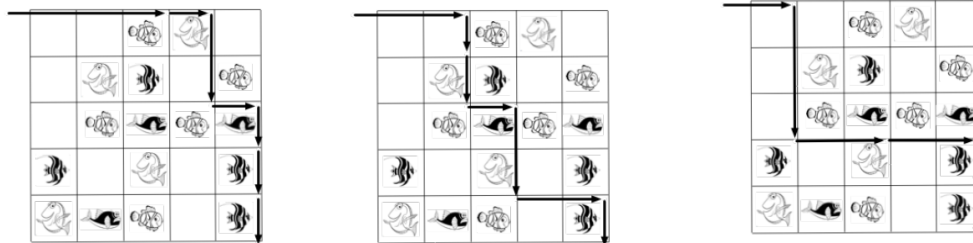
		R_1	R_2	R_3	R_4	R_5
	-	4	1	2	3	5
L_1	5	0	0	0	0	0
L_2	1	0	0	1	1	1
L_3	3	0	0	1	1	2
L_4	2	0	0	1	2	2
L_5	4	0	1	1	2	2

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



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 $2 \times n \times m$. 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.



Como en básicamente toda la tarea, vamos a usar programación dinámica. El punto es calcular el máximo valor de pescado que se puede atrapar en un camino desde la esquina superior izquierda a la esquina inferior derecha. Además, 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, además 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 más larga, en este caso ya tenemos el arreglo auxiliar y le vamos a meter información a las celdas, lo podemos hacer poniéndole un valor de valorM o algo por el estilo. (podemos usar otra matriz con las mismas dimensiones también)

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 más el máximo valor

entre las 2 que puede llegar. Ademias, 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, ademias, 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 (chechar 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:

		<div>012345678</div>									
		<div>c h o c o l a t e</div>									
i/j		0	1	2	3	4	5	6	7	8	9
0	c	1	1	0	0	0	0	0	0	0	0
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 $M[0][0] \text{ AND } X[0]=c==c=Z[0]$, 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	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	0	0	0	0	
1	h	2	0	1	0	0	1	0	0	0	0	
2	i	3	0	0	0	0	0	1	1	1	0	
3	p	4	0	0	0	0	0	0	1	1	1	
4	s	5	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	0	0	0	0	
1	h	2	0	1	0	0	0	1	0	0	0	
2	i	3	0	0	0	0	0	1	1	1	0	
3	p	4	0	0	0	0	0	0	1	1	1	
4	s	5	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	c	1	1	1	1	1	1	0	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	\dots	x_n	
i/j		0	1	2	\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 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.