Pablo Gutiérrez Martínez, Víctor Jorge Sibaja

DESCRIPCIÓN DE ELABORACIÓN Y ANÁLISIS DE PRÁCTICA 2 DE ANÁLISIS Y DISEÑO DE ALGORITMOS

ALGORITMO INICIAL

(Al referirnos a posiciones de la matriz, contamos desde la posición (0,0) hasta la posición (n-1,m-1))

El algoritmo se basa en calcular cual es la combinación que permite tener una torre solución de mayor altura mediante la "acumulación". El algoritmo para cada pieza i, almacena la altura de la torre (contándose a sí misma como pieza de la solución), de esta forma podemos llegar a la torre solución tomando las piezas que generen una "subtorre" mayor.

Para cada caso el algoritmo recibe dos torres, denominadas: piezas1 y piezas2. Al número de piezas de la torre 1 lo hemos denominado m y al número de piezas de la torre 2 lo hemos denominado n. m >= n para todos los casos.

Una vez tenemos los datos formamos una matriz de n filas y m columnas inicialmente vacía. Por ejemplo, para el caso de prueba 1 de entrada.txt (el que figura en el pdf de descripción de la práctica) obtendríamos una matriz 6X7:

		TORRE 1						
		20	15	10	15	25	20	15
TORRE 2	15							
	25							
	10							
	20							
	15							
	20							

Las celdas de color oscuro son las piezas de la torre 1 (eje horizontal) y de la torre 2 (eje vertical), realmente no están en la matriz, pero con este esquema es más fácil comprender el funcionamiento del algoritmo. Las celdas claras son la matriz nxm en la que almacenaremos alturas de las "subtorres".

Ahora simplemente vamos iterando los elementos de la torre 1 y para cada uno recorremos su columna. En las posiciones que la pieza de la torre 1 coincida con la de la torre 2 almacenará 1 + el máximo almacenado hasta entonces.

En el caso de la primera pieza 20, como no había ninguna pieza anteriormente almacenará en las filas 3 y 5 el valor 1 (porque tomándolas permiten formar una torre de altura 1). Entonces tras la primera iteración tenemos la matriz:

		TORRE 1						
		20	15	10	15	25	20	15
TORRE 2	15							
	25							
	10							
	20	1						
	15							
	20	1						

Ahora repetiríamos el proceso con la primera pieza 15. En la fila 0 almacenará un 1, porque si tomamos esa pieza, no podríamos tomar la primera pieza 20 ya que se encuentra después en la torre 2. En la fila 4 si tomamos la pieza 15 de la fila 4 podremos formar una torre "acumulando" la pieza 20 de la iteración anterior (que ahora sí que podemos tomar porque la fila 4 está después de la 3) y obtendríamos una torre solución de altura 2. Resultando la matriz:

		TORRE 1						
		20	15	10	15	25	20	15
TORRE 2	15		1					
	25							
	10							
	20	1						
	15		2					
	20	1						

Repitiendo este proceso para cada elemento i de la torre 1, obtenemos al terminar de iterar la matriz:

		TORRE 1						
		20	15	10	15	25	20	15
TORRE 2	15		1		1			1
	25					2		
	10			2				
	20	1					3	
	15		2		3			4
	20	1					4	

El resultado más alto almacenado en la matriz es de altura 4 y por ello será la altura de la torre solución.

Para hallar la solución “desharemos la acumulación” recorriendo la matriz de forma inversa, buscando:

- Primera pieza (realmente es la última): celda que contenga la altura total de la torre solución.
- Segunda pieza: celda que contenga: el valor de la anterior pieza guardada – 1 y que esté en una fila < fila de la anterior pieza guardada y en una columna < columna de la anterior pieza guardada.
- ….
- Última pieza: contiene el valor 1 y que esté en una fila < fila de la anterior pieza guardada y en una columna < columna de anterior pieza guardada.

Después de esto ya no se busca más piezas ya que el único caso en que puede haber una torre de altura 0 es cuando la torre 1 y la torre 2 no compartan ninguna pieza.

Necesitamos que cada pieza solución que almacenamos esté en al menos una fila y una columna menor a la anterior porque:

- Fila hacia arriba: si tenemos ya la cuarta pieza almacenada y la pieza i es la tercera pieza, la segunda pieza tendrá que estar antes en la torre 2 para poder respetar el orden inicial de las torres.
- Columna a la izquierda: si tenemos ya la cuarta pieza almacenada y la pieza i es la tercera pieza, la segunda pieza tendrá que estar antes en la torre 1 para poder respetar el orden inicial de las torres.

```
for( int j=auxiliar; j>=0;j-- ){
    if (matriz[j][i]==mayorMaximo){
        solucion[mayorMaximo - 1] = piezas1[i];
        mayorMaximo--;
        auxiliar = j - 1;
        break;

    }
}
```

Para no tener que recorrer la matriz entera, al encontrar una pieza de la solución en la posición (i,j), saltamos a la posición (i-1,j-1) y continuamos recorriendo.

		TORRE 1						
		20	15	10	15	25	20	15
TORRE 2	15		1		1			1
	25					2		
	10			2				
	20	1					3	
	15		2		3			4
	20	1					4	

De cada una de las celdas nos guardamos la pieza de la torre 1 a la que corresponde y así tenemos la combinación: 15 20 25 15, que dada la vuelta es la solución:

15 25 20 15

Y habríamos resuelto el caso.

PRIMERA MEJORA DE EFICIENCIA

(Al referirnos a la posición i hablamos de las filas de la matriz y al referirnos a la posición j hablamos de las columnas de la matriz)

Para la mejora de la eficiencia hemos utilizado un vector auxiliar para poder almacenar los máximos de cada fila de la matriz que formamos. Con el anterior algoritmo, para el elemento "a", en la posición (i,j) , tendríamos que recorrer toda la "submatriz" que llega hasta la posición: (i-1,j-1) y buscar el máximo.

Utilizando el vector máximos[] solo tenemos que recorrer desde 0 a i-1, mejorando la eficiencia ya que en el peor caso recorreríamos n-1 (siendo n el número de elementos de la torre 2) elementos en lugar de i-1 x j-1 elementos que tendría la "submatriz".

Para encontrar el máximo del vector de máximos utilizamos la función:

```
public static int buscarMaximo(int[] maximos, int extremoInferior){
    int maximo = 0;

    for (int i = 0; i<extremoInferior; i++){
        if (maximos[i] > maximo){
            maximo = maximos[i];
        }
    }
    return maximo;
}
```

```
if (piezas1[i] == piezas2[j]){
    maximo = buscarMaximo(maximos,j);
    maximo++;
    matriz[j][i] = maximo;
    if (maximo > maximos[j]){
        aux[j] = maximo;
    }
```

BuscarMaximo() es llamada en caso de que la pieza que estamos iterado de la torre 1 coincida con la que iteramos de la torre 2.

El máximo obtenido lo incrementamos en una unidad (debido a que se debe almacenar el máximo + otra pieza) y lo almacenamos en la matriz.

En caso de que el nuevo máximo de la fila i sea mayor que el que haya almacenado en máximos[i], se sustituirá el anterior por el nuevo obtenido.

Podemos usar este esquema para entender mejor la estructura del algoritmo:

		TORRE 1							MAXIMOS
		20	15	10	15	25	20	15	
TORRE 2	15		1		1			1	1
	25					2			2
	10			2					2
	20	1					3		3
	15		2		3			4	4
	20	1					4		4

Las búsquedas del máximo deben ser con el vector de máximos de la anterior iteración, ya que si por ejemplo para la tabla de arriba, cuando estamos completando la primera columna, al llegar a la segunda pieza 20 encontraremos un máximo: 1. Según el algoritmo si no utilizamos nada más debería haber un 2 para la segunda pieza 20.

Necesitaremos por lo tanto realizar las búsquedas en una copia del vector de máximos de la anterior iteración, por ello antes de recorrer cada columna, copiamos el vector máximos en el vector aux[] haciendo un clon:

```
aux = maximos.clone();
```

Este vector aux[] será sobre el que hacemos las modificaciones de los máximos y al terminar de recorrer cada columna, reemplazamos el vector de máximos por el vector aux[] en el que hemos hechos las modificaciones:

```
maximos = aux;
```

SEGUNDA MEJORA DE EFICIENCIA

Existen casos en los que una pieza puede estar en una torre y en la otra no, en estos casos si no realizamos ninguna modificación recorreríamos la columna igualmente.

Sin embargo, si utilizamos una tabla de Hash en la que almacenamos como clave las piezas de la torre 2 (se almacenan los de la 2 para tener que almacenar un menor número de piezas según hemos diseñado el algoritmo), podemos saber si una pieza está en las dos torres y en caso de no estarlo, pasar directamente a la siguiente pieza evitando recorrer la columna de la matriz.

Tomamos una pieza de la torre 1 y consultando si está contenida en la tabla de Hash (lo cual es una operación con O(1)) sabemos si debemos recorrer la columna o no:

```
if (tabla.containsKey(piezas1[i])){
```

Podríamos pensar que rellenar la tabla de Hash empeora la eficiencia, pero solo nos llevará un O(n) (ya que iteramos los n elementos de la torre 2 y los metemos en la tabla de Hash cada uno con una operación O(1)) lo cual es menor que el orden del algoritmo completo:

```
for (int i = 0; i<n; i++){
    tabla.put(piezas2[i],i);
}
```

ANÁLISIS DE EFICIENCIA

Para hacer este análisis se tiene en cuenta el peor caso: que la torre 1 y la torre 2 sean exactamente iguales, ya que esto provoca que se tenga que rellenar toda la matriz.

```
for (int i = 0; i<m; i++){
    aux = maximos.clone();
    /*
    Solo en caso de que el elemento de la torre 1 esté en la torre 2
    interesará recorrer
     */
    if (tabla.containsKey(piezas1[i])){
        for (int j = 0; j<n; j++){
            if (piezas1[i] == piezas2[j]){
                maximo = buscarMaximo(maximos,j);
                maximo++;
                matriz[j][i] = maximo;
                if (maximo > maximos[j]){
                    aux[j] = maximo;
                }

                if (maximo>mayorMaximo){
                    mayorMaximo=maximo;
                }
            }
        }
        maximos = aux;
    }

}
```

Si nos fijamos en el código tenemos dos bucles anidados que provocan que se ejecuten las operaciones elementales n x m veces, ya que la comprobación en la tabla de Hash no tiene efecto al tener que rellenar toda la matriz.

Si nos fijamos en el segundo bucle, vemos que se invocaría la función buscarMaximo() n x m veces, ya que al ser todas las piezas iguales, la condición de que las piezas de cada torre que estamos iterando sean iguales se va a cumplir siempre.

Si ahora analizamos la función buscarMaximo():

```
public static int buscarMaximo(int[] maximos, int extremoInferior){
    int maximo = 0;

    for (int i = 0; i<extremoInferior; i++){
        if (maximos[i] > maximo){
            maximo = maximos[i];
        }
    }
    return maximo;
}
```

Vemos que cada vez que es invocada en este caso, al estar recorriendo la matriz por columnas, si estamos en la fila i (la cual pasamos como el parámetro el extremo Inferior) esta función iterará hasta la fila i-1. Esto provoca que cada iteración tenga un coste distinto que se irá incrementando en 1 a medida que aumente i, llegando como máximo a iterar n-1 elementos (correspondiente a las celdas de la matriz que se encuentran en la fila n-1). Es decir, en la primera iteración se iterará 0 veces, en la segunda 1 vez, en la tercera 2 veces, … , y en la llamada n iterará n-1 veces.

Esto nos quedaría $\Sigma^{n-1}_{i=1}$ i = n(n-1) / 2 para cada columna de la matriz.

Tras este análisis podemos concluir que en el orden obtenido en el peor caso es:
O(n x m x (n (n-1)/2)) . Como hemos dicho que en el peor caso n = m, el orden pasa a ser:
$O(n^2 \times (n^2/2 - n/2)) = O(n^4/2 - n^3/2) \cong O(n^4)$.

No se tiene en cuenta el recorrido hacia atrás para sacar la solución en el análisis porque siempre va a ser de coste inferior a lo que hemos analizado antes.

REPARTO DE TRABAJO

Ambos participantes hemos realizado la misma carga de trabajo al realizar la práctica, por lo tanto cada uno ha realizado un 50%, realizando múltiples reuniones y colaborando continuamente.