Primer Parcial de Estructura de Datos y Algoritmos

Ejer 1	Ejer 2	Ejer 3	Nota
/4	/4	/2	/10

Duración: 2 horas

Condición Mínima de Aprobación. Deben cumplir estas 2 condiciones:

- Sumar **no menos de cuatro** puntos
- Sumar por lo menos 3 puntos entre el ejercicio 1 y 2.

Muy Importante

Al terminar el examen deberían subir los siguientes 2 grupos de archivos, según lo explicado en los ejercicios:

1) Clases Java: **MinPathFinder.java**, **IndexWithDuplicates.java** y **SingleListList.java** según lo pedido. Si hay código auxiliar, entregarlo también.

2) Para todos los ejercicios que no consistan en implementar código Java y pidan calcular complejidades, dibujar matrices, completar cuadros, hacer seguimientos, etc. pueden optar por alguna de estas estrategias:
   a. O completar este documento y subirlo también
   b. O directamente resolverlo en hojas de papel y sacarle fotos (formato jpg, png o pdf) y subir todas las imágenes.

Ejercicio 1

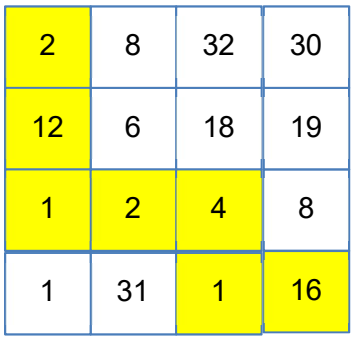
Dada una matriz de enteros donde cada celda representa el costo de la celda correspondiente en la matriz, se pide implementar un algoritmo que encuentre el **costo acumulado mínimo** que resulta de partir **desde la celda de arriba a la izquierda** y llegar finalmente a la **celda de abajo a la derecha**. Sólo se puede mover entre celdas adyacentes, pero con una importante **restricción de movimiento**: parados en una celda los únicos dos movimientos posibles son **o bien ir hacia abajo** (celda adyacente) **o bien ir hacia la derecha** (celda adyacente).

Se pide:

Implementar la clase **MinPathFinder** con una única función **findMinPath(int[][] weightMatrix)** que tome una matriz como entrada y devuelva el costo mínimo para llegar desde la celda de arriba a la izquierda a la celda de abajo a la derecha según lo explicado.

Ejemplo 1: Partiendo de la celda superior izquierda (marcada en el círculo) hay que llegar a la última a la derecha, **eligiendo cada vez una celda** en el camino, solo moviéndose o bien hacia la derecha o bien hacia abajo. Esa celda elegida tiene un costo que es el que se acumula hasta llegar a la final. El algoritmo debe calcular el mínimo costo acumulado posible.

2	8	32	30
12	6	18	19
1	2	4	8
1	31	1	16



2	8	32	30
12	6	18	19
1	2	4	8
1	31	1	16

El recorrido indicado tiene el menor costo acumulado posible**: 2 + 12 + 1 + 2 + 4 + 1 +16 = 38**

(no interesa conocer el recorrido, solo el costo mínimo que puede alcanzarse)

Al realizar la siguiente invocación, se obtiene **38** :

```
public static void main(String[] args) {
        int[][] v = new int [][]
        {{2, 8, 32, 30},
        {12, 6, 18, 19},
        {1, 2, 4, 8},
        {1, 31, 1, 16}};
        MinPathFinder minPathFinder = new MinPathFinder();
        int ans = minPathFinder.getMinPath(v);
        System.out.println(ans);
}
```

Ejemplo 2:

2	8	32	30
12	6	18	19
1	2	4	8

2	8	32	30
12	6	18	19
1	2	4	8

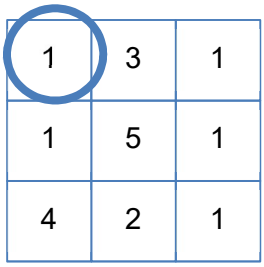
El recorrido indicado tiene el menor costo acumulado posible**: 2 + 12 + 1 + 2 + 4 + 8 = 29**

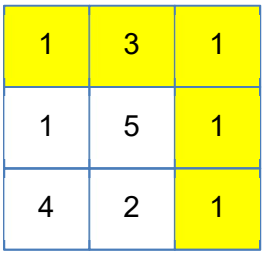
(no interesa conocer el recorrido, solo el costo mínimo que puede alcanzarse)

Al realizar la siguiente invocación, se obtiene **29** :

```
public static void main(String[] args) {
        int [][] v = new int [][]
        {{2, 8, 32, 30},
        {12, 6, 18, 19},
        {1, 2, 4, 8}};
        MinPathFinder minPathFinder = new MinPathFinder();
        int ans = minPathFinder.getMinPath(v);
        System.out.println(ans);
}
```

Ejemplo 3:





El recorrido indicado tiene el menor costo acumulado posible**: 1 + 3 + 1 + 1 + 1 = 7**

(no interesa conocer el recorrido, solo el costo mínimo que puede alcanzarse)

realizar la siguiente invocación, se obtiene **7** :

```
public static void main(String[] args) {
        int[][] v = new int [][]
        {{1, 3, 1},
        {1, 5, 1},
        {4, 2, 1}};
        MinPathFinder minPathFinder = new MinPathFinder();
        int ans = minPathFinder.getMinPath(v);
        System.out.println(ans);
}
```

Ejercicio 2

Descargar de campus los archivos **IndexWithDuplicates.java y SingleListList.java (deben usar estas versiones y agregar lo que crean necesario).**

La clase **IndexWithDuplicates** representa un **índice paramétrico con repeticiones** (no compactado)

La clase **SingleLinkedList representa una lista lineal simplemente encadenada sin orden**

Queremos agregarle a la **clase IndexWithDuplicates el método de instancia**:

void **repeatedValues**(T[] values,
SimpleLinkedList<T> repeatedLst,
SimpleLinkedList<T> singleLst,
SimpleLinkedList<T> notIndexedLst)

El método recibe un arreglo **de valores desordenados** y debe generar las **tres listas simplemente encadenadas** de la siguiente manera:

- En la lista **repeatedLst** deberán estar todos los elementos del arreglo **values** que se encuentren repetidos en el índice.
- En la lista **singleLst** deberán estar todos los elementos del arreglo **values** que se encuentren una sola vez en el índice
- En la lista **notIndexedLst** deberán estar los elementos del arreglo **values** que no se encuentran presentes en el índice.

Existe cuatro importantes restricciones que debe cumplirse:

a) en cada una de las listas generadas los valores deben respetar el orden en que se encuentran en el arreglo **values**.

b) La implementación del método **repeatedValues** debe tener una **complejidad temporal** de **O(M log N)** siendo M el tamaño del array **values** y N la cantidad de elementos del índice.

c) Si alguno de los parámetros viene en null lanzar excepción con mensaje correspondiente.

d) No se pueden usar colecciones auxiliares que ya vengan implementadas en Java. Todo lo deben implementar Uds.

Ejemplo 1:
Si se invocara

```
public static void main(String[] args) {
IndexWithDuplicates<Integer> idx = new IndexWithDuplicates<>();
idx.initialize( new Integer[] {100, 50, 30, 50, 80, 10, 100, 30, 20, 138} );
```

```
SimpleLinkedList<Integer> repeatedLst = new SimpleLinkedList();
SimpleLinkedList<Integer> singleLst = new SimpleLinkedList();
SimpleLinkedList<Integer> notIndexedLst = new SimpleLinkedList();

idx.repeatedValues( new Integer[] { 10, 80, 10, 35, 80, 80 , 1111},
repeatedLst, singleLst, notIndexedLst );

System.out.println("Repeated Values");
repeatedLst.dump();

System.out.println("Single Values");
singleLst.dump();

System.out.println("Non Indexed Values");
notIndexedLst.dump();
}
```

Se obtendría:

```
Repeated Values

Single Values
10
80
10
80
80

Non Indexed Values
35
1111
```

La lista **repeatedLst** está vacía porque no había ningún elemento en el arreglo **values** entre los elementos repetidos del índice.

En la lista **singleLst** deben aparecer los valores del arreglo que tenían una sola aparición en el índice y conservar el orden en que estaban en el arreglo **values**. Por eso aparecen 10, 80, 10, 80, 80 conservando el orden.

En la lista **notIndexedLst** aparecen los valores del arreglo **values** que no están presentes en el índice y preservando el orden de aparición. Por eso está el 35 precediendo al 1111

Ejemplo 2:

Si se invocara

```
public static void main(String[] args) {
IndexWithDuplicates<Integer> idx = new IndexWithDuplicates<>();
idx.initialize( new Integer[] {100, 50, 30, 50, 80, 10, 100, 30, 20, 138} );

SimpleLinkedList<Integer> repeatedLst = new SimpleLinkedList();
SimpleLinkedList<Integer> singleLst = new SimpleLinkedList();
SimpleLinkedList<Integer> notIndexedLst = new SimpleLinkedList();

idx.repeatedValues( new Integer[] { 100, 70, 40, 120, 33, 80, 10, 50 }, repeatedLst,
singleLst, notIndexedLst );

System.out.println("Repeated Values");
repeatedLst.dump();
System.out.println("Single Values");
singleLst.dump();
System.out.println("Non Indexed Values");
notIndexedLst.dump();
}
```

Se obtendría:

```
Repeated Values
100
50

Single Values
80
10

Non Indexed Values
70
40
120
33
```

Ejercicio 3

¿Qué orden de complejidad temporal tiene este código? ¿Y espacial? Justificar detalladamente.

```
public static int maxSubarraySum(int[] arr) {
    int n = arr.length;
    int maxSum = Integer.MIN_VALUE;
    int currentSum = 0;
    for (int i = 0; i < n; i++) {
        currentSum = Math.max(arr[i], currentSum + arr[i]);
        maxSum = Math.max(maxSum, currentSum);
    }
    return maxSum;
}
```