Práctica 0

Material de apoyo:

- código de ejemplo de recorrido de matrices: EjemploMatrices.java.
- además, todos los ejercicios obligatorios cuentan con un archivo de pruebas automáticas para correr desde Eclipse con JUnit.

Índice

{:.no_toc}

- TOC {:toc}

Nota: Los ejercicios marcados con ★ son de mayor dificultad.

Acumuladores booleanos

{: #accumuladores}

accum-pertenecen

{: #accum-pertenecen}

Implementar una función que determine si un arreglo es subconjunto de otro:

```
public static boolean pertenecenTodos(int[] elems,
                                      int[] arreglo) ...
```

Casos borde a tener en cuenta:

- *elems* está vacío (y la función devuelve verdadero)
- *arreglo* está vacío (y la función devuelve falso)
- alguno de los arreglos contiene duplicados (no influye, es suficiente con que estén una vez)

Algunos ejemplos:

```
[1, 2] ⊆ [3, 2, 1]
[4, 1] ⊈ [1, 2, 3]
[2, 2] ⊆ [1, 2, 3]
```

Pruebas automáticas: TestAccumPertenecen.java.

accum-matriz

{: #accum-matriz}

Implementar una función que, dada una matriz de enteros, verifique que:

1. todas las filas están en orden estrictamente ascendente
2. todas las columnas tienen al menos un elemento impar, y otro par

Signatura y documentación:

```
// Pre-condición: "int[][] mtx" es una matriz N × M, esto es:
// todas las filas tienen longitud N y todas las columnas, M;
// con N, M > 0.
//
// No es necesario verificar explícitamente la pre-condición:
// de no cumplirse, el código puede devolver cualquier valor, o
// lanzar una excepción (p.ej. ArrayIndexOutOfBoundsException).
public static boolean mayorDiversidad(int[][] mtx) ...
```

Algunos ejemplos:

```
[[1, 2, 3], [4, 5, 6]] → Verdadero
[[1, 2, 3], [4, 5, 5]] → Falso
[[1, 2, 3], [2, 4, 6]] → Falso

No cumplen la pre-condición:

    [[1, 2], [3, 4], [5, 6, 7]]
    [[1], [2, 3]]
    [[1], [2, 3], []]
    []
    [[]]
```

Pruebas automáticas: TestAccumMatriz.java.

accum-matriz2 ★

{: #accum-matriz2}

Implementar una nueva versión de la función anterior, eliminando la pre-condición:

```
public static boolean mayorDiversidad2(int[][] arr) ...
```

Arreglos estáticos

{: #arreglos}

arr-combinar

{: #arr-combinar}

Implementar una función que reciba dos arreglos ordenados y devuelva un tercer arreglo ordenado que sea la unión de ambos:

```
public static int[] combinarOrdenados(int[] a, int[] b) ...
```

No se permite modificar ninguno de los arreglos originales, ni usar estructuras auxiliares (excepto el nuevo arreglo a devolver).[^estraux]

Ejemplo:

```
Arreglo A: [4 7 9 15 35 39]
Arreglo B: [1 2 5 9 14 30 50]

Resultado ( new int[13] ):

    [1 2 4 5 7 9 9 14 15 30 35 39 50]
```

Pruebas automáticas: TestArrCombinar.java.

[^estraux]: Siempre se permite usar métodos auxiliares *privados*, y tantas variables de tipos primitivos como sean necesarias. El término "estructuras auxiliares" hace referencia a objectos creados con `new`. En este caso, por ejemplo, puede usarse una sola instrucción **`new`**, para crear el arreglo que será devuelto.

arr-pico

{: #arr-pico}

Un arreglo se dice unimodal o en forma de pico si es estrictamente creciente hasta una posición P, y estrictamente decreciente a partir de ella.

Implementar una función que devuelva el índice del pico en un arreglo unimodal, o -1 si el arreglo no tiene forma de pico.

```
public static int indicePico(int[] arreglo) ...
```

Ejemplos:

```
[2, 4, 6, 19, 15, 8, -2] → se devuelve 3 (19 es el pico)
[10, 20, 30, 40, 50, 15] → se devuelve 4 (50 es el pico)
[50, 100, 75]            → se devuelve 1 (100 es el pico)
[50, 75, 100]            → se devuelve -1 (no es unimodal)
[1, 4, 7, 6, 5, 2, 3, 1] → se devuelve -1 (no es unimodal)
```

Nota: siempre se cumple que 0 < P < N-1, por lo que todos los arreglos unimodales tienen al menos tres elementos.

Pruebas automáticas: TestArrPico.java.

arr-picolog ★

{: #arr-picolog}

Implementar un método estático `indicePicoLog()` que, con complejidad logarítmica, encuentre el pico en un arreglo que se sabe es unimodal.

```
// Pre-condición: "arreglo" es unimodal (tiene forma de pico).
//
// Si se cumple la pre-condición:
//   (1) la función calcula el resultado correcto
//   (2) en tiempo proporcional a log₂(arreglo.length)
//
// No es necesario verificar la pre-condición: de no cumplirse, la
// función no garantiza ningún valor de retorno en particular.
public static int indicePicoLog(int[] arreglo) ...
```