Estructuras de Datos 2021-2
Práctica 1: Complejidad Computacional

Pedro Ulises Cervantes González
confundeme@ciencias.unam.mx

Emmanuel Cruz Hernández
emmanuel_cruzh@ciencias.unam.mx

Yessica Janeth Pablo Martínez
yessica_j_pablo@ciencias.unam.mx

América Montserrat García Coronado
ame_coronado@ciencias.unam.mx

Adrián Felipe Vélez Rivera
adrianf_velez@ciencias.unam.mx

Fecha límite de entrega: 17 de marzo de 2021
Hora límite de entrega: 23:59 hrs

1. Objetivo

La complejidad de un algoritmo es muy importante para hacer eficiente el funcionamiento de un programa, ya sea en tiempo o en espacio. Con esta práctica se espera que analices la cantidad de operaciones que realiza un algoritmo para resolver un problema, de tal forma que se pueda mejorar el tiempo en que se resuelve la misma tarea.

2. Actividad

Dados los siguientes métodos, mejora el tiempo de ejecución de cada uno.

2.1. Actividad 1 *(2.5 puntos)*

Dado un arreglo de enteros *num*, encuentra el índice del primer y último entero llamado *value*. Si no existe el elemento, se regresa [-1,-1].

- `findFirstAndLast([1,4,2,1,6,2,9], 2) = [2, 5]`
- `findFirstAndLast([4,2,7,5,4,3,7,2,5,3,4,1], 15) = [-1, -1]`
- `findFirstAndLast([3,2,1,4,2], 1) = [2, 2]`

El siguiente algoritmo tiene complejidad *O(n)*. Mejora el método para que la complejidad sea *O(n/2)*.

```java
public int[] findFirstAndLast(int[] num, int value){
    int[] result = new int[2];
    result[0] = -1;
    result[1] = -1;

    for(int i = 0; i < num.length ; i++){
        if(num[i] == value){
            result[0] = i;
            break;
        }
    }

    for(int j = num.length-1 ; j >= 0 ; j--){
        if(num[j] == value){
            result[1] = j;
            break;
        }
    }

    return result;
}
```

2.2. Actividad 2 *(3 puntos)*

Dado un arreglo bidimensional de 6x6, que representa un tablero de sudoku, determina si el tablero es válido o no (**sólo considera verticales y horizontales**).

- $$isSudokuValid(\begin{pmatrix} 4 & 5 & 6 & 2 & 3 & 1 \\ 3 & 1 & 2 & 6 & 4 & 5 \\ 1 & 6 & 4 & 3 & 5 & 2 \\ 5 & 2 & 3 & 1 & 6 & 4 \\ 2 & 3 & 5 & 4 & 1 & 6 \\ 6 & 4 & 1 & 5 & 2 & 3 \end{pmatrix}) \rightarrow true$$

- $$isSudokuValid(\begin{pmatrix} 4 & 5 & 6 & 2 & 3 & 1 \\ 3 & 1 & 2 & 6 & 4 & 5 \\ 2 & 6 & 4 & 3 & 5 & 2 \\ 5 & 2 & 3 & 1 & 6 & 4 \\ 1 & 3 & 5 & 4 & 1 & 6 \\ 6 & 4 & 1 & 5 & 2 & 3 \end{pmatrix}) \rightarrow false$$

El siguiente algoritmo tiene complejidad $O(n^3)$. Mejora el método para que la complejidad sea $O(n^2)$.

```java
public static boolean isSudokuValid(int[][] board){
    int length = board.length;
    for (int i = 0; i < length ; i++) {
        for (int j = 1; j <= length ; j++ ) {
            boolean verificador = false;
            // Verifica sobre las filas
            for(int k = 0 ; k < length; k++){
                if(board[i][k] == j){
                verificador = true;
                break;
                }
            }
            if(!verificador){
                return false;
            }
            verificador = false;
            // Verifica sobre las columnas
            for(int k = 0 ; k < length; k++){
                if(board[k][i] == j){
                    verificador = true;
                    break;
                }
            }
            if(!verificador){
                return false;
            }
        }
    }
    return true;
}
```

2.3. Actividad 3 *(2.5 puntos)*

Dado un arreglo *num* y un entero *positions* ≥ 0, se regresa un arreglo rotado *position* cantidad de posiciones hacía la izquierda.

- `rotateArray([1,4,2,1,6,2,9], 5)` $\rightarrow$ `[2,9,1,4,2,1,6]`
- `rotateArray([4,2,7,5,4,3,7,2,5,3,4,1], 0)` $\rightarrow$ `[4,2,7,5,4,3,7,2,5,3,4,1]`
- `rotateArray([3,2,1,4,2], 2)` $\rightarrow$ `[1,4,2,3,2]`

El siguiente algoritmo tiene complejidad $O(n^2)$. Mejora el método para que la complejidad sea $O(n)$.

```java
public static void rotateArray(int[] num, int position){
    for(int i = 0; i < position ; i++){
        int aux = num[0];
        for(int j = 0; j < num.length -1 ; j++){
            num[j] = num[j+1];
        }
    num[num.length-1] = aux;
    }
}
```

2.4. Actividad 4 *(2 puntos)*

Crea un archivo *Test.pdf*, donde llenes las tablas con los resultados obtenidos y explica brevemente (de 2 a 4 renglones) porqué el algoritmo que diseñaste mejora la complejidad en tiempo de cada uno de los algoritmos de las actividades.

findFirstAndLast		
Entradas	Milisegundos algoritmo 1	Milisegundos algoritmo 2
[1,4,2,1,6,2,9], 2		
[4,2,7,5,4,3,7,2,5,3,4,1], 15		
[3,2,1,4,2], 1		

isSudokuValid		
Entradas	Milisegundos algoritmo 1	Milisegundos algoritmo 2
ejemplo2a		
ejemplo2b		

findFirstAndLast		
Entradas	Milisegundos algoritmo 1	Milisegundos algoritmo 2
[1,4,2,1,6,2,9], 5		
[4,2,7,5,4,3,7,2,5,3,4,1], 0		
[3,2,1,4,2], 2		

3. Extra *(1 punto)*

El Juego de la vida es un autómata celular diseñado por el matemático británico John Horton Conway en 1970 [1].

El tablero está formado por una cuadrícula de celdas de m x n, donde cada celda tiene un estado inicial: vivo (representado por un 1) o muerto (representado por un 0). Cada celda interactúa con sus ocho vecinos (horizontal, vertical, diagonal) utilizando las siguientes cuatro reglas (tomadas del artículo de Wikipedia):

- Cualquier célula viva con menos de dos vecinos vivos muere a causa de una población insuficiente.
- Cualquier célula viva con dos o tres vecinos vivos vive para la próxima generación.
- Cualquier célula viva con más de tres vecinos vivos muere, por sobrepoblación.
- Cualquier célula muerta con exactamente tres vecinos vivos se convierte en una célula viva, por reproducción.

El siguiente estado se crea aplicando las reglas anteriores simultáneamente a cada celda en el estado actual, donde los nacimientos y muertes ocurren simultáneamente. Dado el estado actual de la placa de cuadrícula m x n, devuelve el siguiente estado.

- $$gameOfLife(\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}) \rightarrow \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

[1] `https://es.wikipedia.org/wiki/Juego_de_la_vida`

- $$gameOfLife(\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}) \rightarrow \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

Crea un método que implemente el juego de la vida y menciona en el PDF *Test.pdf* la complejidad del método que implementaste.

4. Reglas Importantes

- **No se recibirán prácticas en las que estén involucrados más de dos integrantes**.
- Cumple con los lineamientos de entrega.
- Todos los archivos deberán contener nombre y número de cuenta.
- Tu código debe estar comentado. Esto abarca clases, atributos, métodos y comentarios extra.
- Para cada clase solicitada, crea un nuevo archivo.
- Utiliza correctamente las convenciones para nombrar variables, constantes, clases y métodos.
- El programa debe ser 100 % robusto.
- En caso de no cumplirse alguna de las reglas especificadas, se restará 0.5 puntos en tu calificación obtenida.

¡Éxito!