Estructuras de Datos 2022-1
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: 1 de octubre de 2021
Hora límite de entrega: 23:59:59 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 y observes de forma práctica el tiempo que toma un algoritmo eficiente respecto a otro.

2. Actividad

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

2.1. Actividad 1 *(2.5 puntos)*

Dados dos arreglos ordenados de enteros, se mezcla de dos ambos arreglos desde la primera posición hasta una posición límite n y m para el arreglo A y el arreglo B respectivamente.

La mezcla da como resultado un arreglo ordenado de enteros de longitud `n+m` que contiene los primeros n elementos del arreglo A y los primeros m elementos de arreglo B.

- `mergeSortedArray([13,29,58,58,74,90,91], 3, [3,11,13,16,27,56,59,61,88,90], 5) = [3,11,13,13,16,27,29,58]`
- `mergeSortedArray([1,24,49,51,63,79,99,107,107,126,128,149], 5, [26,54,68,97,103,107, 109,123,126,132,155,175,187,208,230], 5) = [1,24,26,49,51,54,63,68,97,103]`

El siguiente algoritmo tiene complejidad *O(n+m)*. Mejora el método para que la complejidad sea *O(max(n, m))*.

```java
public static int[] mergeSortedArray(int[] array1, int n,
                                     int[] array2, int m){
    if(n > array1.length || m > array2.length)
            throw new RuntimeException("Límites no válidos");

    int[] result = new int[n + m];
    int pointer;
    for(pointer = 0; pointer < n; pointer++)
            result[pointer] = array1[pointer];
    for(int i = 0 ; i < m ; i++, pointer++)
            result[pointer] = array2[i];

    // Ordenamiento del arreglo result
    for(int j = 0; j < result.length - 1; j++){
            for(int k = j+1; k < result.length; k++){
                    if(result[k] < result[j]){
                            int aux = result[k];
                            result[k] = result[j];
                            result[j] = aux;
                    }
            }
    }

    return result;
}
```

2.2. Actividad 2 *(3 puntos)*

Dado un arreglo bidimensional de `nxn`, que representa un tablero, determina si el tablero contiene los elementos de 0 a n-1 en cada fila y columna. Como cada elemento es único en cada fila y columna, no pueden haber repetidos en una misma fila o columna.

- $$isValidBoard(\begin{pmatrix} 4 & 5 & 0 & 2 & 3 & 1 \\ 3 & 1 & 2 & 0 & 4 & 5 \\ 1 & 0 & 4 & 3 & 5 & 2 \\ 5 & 2 & 3 & 1 & 0 & 4 \\ 2 & 3 & 5 & 4 & 1 & 0 \\ 0 & 4 & 1 & 5 & 2 & 3 \end{pmatrix}) \rightarrow true$$

- $$isValidBoard(\begin{pmatrix} 4 & 5 & 0 & 2 & 3 & 1 \\ 3 & 1 & 2 & 0 & 4 & 5 \\ 2 & 0 & 4 & 3 & 5 & 2 \\ 5 & 2 & 3 & 1 & 0 & 4 \\ 1 & 3 & 5 & 4 & 1 & 0 \\ 0 & 4 & 1 & 5 & 2 & 3 \end{pmatrix}) \rightarrow false$$

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

```java
public static boolean isValidBoard(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*m)*, donde n es la longitud del arreglo y m es la cantidad de desplazamientos. Mejora el método para que la complejidad sea *O(m)*.

```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 una de las actividades.

mergeSortedArray(int[], int, int[], int)		
Entradas	Milisegundos algoritmo 1	Milisegundos algoritmo 2
ArrayA1.txt, 500, ArrayA2.txt, 700		
ArrayB1.txt, 2000, ArrayB2.txt, 3500		
ArrayC1.txt, 4000, ArrayC2.txt, 4000		
ArrayD1.txt, 7000, ArrayD2.txt, 8000		
ArrayE1.txt, 15000, ArrayE2.txt, 19000		
ArrayF1.txt, 30000, ArrayF2.txt, 25000		

isValidBoard(int[][])		
Entradas	Milisegundos algoritmo 1	Milisegundos algoritmo 2
BoardA.txt		
BoardB.txt		
BoardC.txt		
BoardD.txt		
BoardE.txt		
BoardF.txt		

rotateArray(int[], int)		
Entradas	Milisegundos algoritmo 1	Milisegundos algoritmo 2
ArrayA1.txt, 500		
ArrayB1.txt, 1000		
ArrayC1.txt, 2000		
ArrayD1.txt, 3000		
ArrayE1.txt, 10000		
ArrayF1.txt, 20000		

NOTA: Todos los archivos de texto están en el directorio *Tests*. Te puedes apoyar de la clase *ArrayReader.java* para leerlos. OJO! La lectura no forma parte del cálculo de milisegundos.

3. 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 que consideres necesarios.
- 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!