|  |
| --- |
| **UNIVERSIDAD NACIONAL DE GENERAL**  **SARMIENTO**  PROGRAMACION I (2° SEMESTRE DE 2020)  COMPLEJIDAD COMPUTACIONAL  **Fecha de entrega:** 10/11/2020  **Ubicación:** Los Polvorines, ciudad del partido de Malvinas Argentinas.  **Comisión:** COM-03 (Turno Noche)  **Docentes:** Waingarten, Leonardo y Niveyro, César  **Estudiante:** Barrientos, Lucas. |

1. El siguiente método detecta si un arreglo tiene alguna de las primeras potencias de 2.

**public static boolean tienePotencia(double[] arr) {**

int i = 0, j = 0; 2 asignaciones (se ignora porque su orden es menor al orden de complejidad)

F1= while (i < arr.length) { arr.length comparaciones = n (O(n))

j = 0; k veces (se ignora porque su orden es menor al orden de complejidad)

F2= while (j < arr.length) { arr.length comparaciones = n\*\*2 (O(g(n)))

if (arr[i] == Math.pow(2, j) { 4n iteraciones(if+vector+comparacion+calculo)

return true; 0

}

j++; k veces (se ignora)

}

i++; k veces (se ignora)

}

return false; 1 iteración (se ignora porque su orden es menor al orden de complejidad)

**}**

**Entrada:** arr

**Tamaño:** arr.length

**Función de Complejidad:** Sumamos los bloques F1 + F2 = 2 + n + 1 + n\*\*2 + 4n = 2 + 5n + n\*\*2

**Orden de Complejidad:** Dado que F1 = O(n) y F2 = O(g(n)) == O(n\*\*2) Nos quedamos con el valor más grande: O(n\*\*2) (Función cuadrática)

2. El siguiente método calcula una suma relacionada con el parámetro n.

**public static double rareza(int n) {**

double sum = 0.0; 1 asignación (se ignora porque su orden es menor al orden de complejidad)

F1= for (int i = n + 100; i >= 0; i−−) { n comparaciones (O(n))

F2= for (int j = n + 100; j >= n; j−−) { n(n\*100n)comparaciones= n\*\*2 + 100n (O(g(n)))

sum = sum + (double) j/(i + 100); 3n (suma total+double j + (i+100))

}

}

return sum; 1 iteración (se ignora porque su orden es menor al orden de complejidad) = O(1)

**}**

**Entrada:** n

**Tamaño:** n

**Función de Complejidad:**

Sumamos los bloques

F1 + F2 = 1 + n + n\*\*2 + 100n + 3n = 1+ 104n + n\*\*2

**Orden de Complejidad:** Dado que F1 = O(n) y F2 = O(g(n)) == O(n\*\*2) Nos quedamos con el valor más grande: O(n\*\*2) (Función cuadrática)

3. El siguiente método escribe un número en base 3.

**public static void base3(int n) {**

if(n <= 2) { 1 comparación

System.out.print(n); 1 iteración

} else {

base3(n / 3); n/3 iteraciones

System.out.print(n % 3); 1 iteración

}

**}**

**Entrada:** n

**Tamaño:** n

**Función de Complejidad:** 3(n/3) = 3n + 9

**Orden de Complejidad:** O(n) Función Lineal

4. El siguiente método suma los primeros n términos de la serie geométrica de razón 1/2.

**public static double serieGeom2(int n) {**

if (n < 1) { n comparaciones

return 1; 1 iteracion = O(1)

}

return 1 + 0.5 ∗ serieGeom2(n − 1); 1+ 0.5(T(n-1)) iteraciones

**}**

**Entrada:** n

**Tamaño:** n

**Función de Complejidad:** n + 1+ 1+ 0.5(T(n-1)) = T(n) = 2 + n + 0.5(T(n-1))

**Orden de Complejidad:**

Cuando tenemos una función al estilo: T(n) = 2 + n + 0.5(T(n-1)), se requiere llegar a k de la forma:

T(n) = 2+ n + 0.5(T(n-1)) => T(n) = 4+ 2n + 1.0(T(n-2)) => T(n) = 8 + 4n + 1.5(T(n-3)) => …

=> T(n) = k + k+ k(T(n-k))

Al ser k y n el mismo número da como resultado n = O(n) (Función Lineal)

5. El siguiente método usa bloques de código de los cuales solo conocemos su complejidad.

**public static void funcionBloques(int n) {**

for (int i = 1; i <= n + 1000; i++) { n+1000 iteraciones

F1 = if (i < n/3) { n iteraciones

/∗ bloque de código de orden O(n) ∗/ (n-1)/3 iteraciones = O(n)

F2 = } else if (i >= n/3 && i < 2 ∗ n/3) { 7n = if+(n/3)+(i>=n/3)+(n/2)+(i<2)+(i<2\*n/3)+&&

/∗ bloque de código de orden O(nˆ2) ∗/ (n\*\*2 - 1)/3 iteraciones = O(n\*\*2)

F3 = } else if (i >= 2 ∗ n/3 && i <= n) { 6n = (i>=n/3)+ (n/3)+ (&&) + (i<2) + (n/3) + (else if)

/\* bloque de código de orden O(log(n)) ∗/ (log n – 1)/3 iteraciones = O(log n)

F4 =} else { n iteraciones

/∗ bloque de código de orden O(nˆ3) ∗/ (n\*\*3)/3 iteraciones = O(n\*\*3)

}

}

**}**

**Entrada:** n

**Tamaño:** n

**Función de Complejidad:**

1000 + n + n\*((n-1)/3) + 7n \* ((n\*\*2-1)/3) + 6n \* ((log(n)-1)/3) + n \* ((n\*\*3-1)/3)

**Orden de Complejidad:**

Teniendo todos los órdenes verificamos cual es el más grande y nos quedamos con él:

F1= O(n\*\*2) F2 = O(n\*\*3) F3 = O(n log (n)) F4 = O(n\*\*3)

Nos quedamos con O(n\*\*4) (Función polinómica) porque: O(log n) c O(n\*\*2) c O(n\*\*3) c O(n\*\*4)

6. El siguiente método devuelve una matriz cuadrada armada a partir un arreglo que se pasa

como parámetro. Se asume que f y c son menores al largo del arreglo.

**public static double[][] matrizReducida(double[] array, int f, int c){**

int n = array.length; 1 asignación (se ignora porque su orden es menor al orden de complejidad)

double [] B = new double[n − 1][n − 1]; 1 asignación (se ignora porque su orden es menor al orden de complejidad)

F1 = for (int k = 0; k < f; k++) { f iteraciones

J1= for (int j = 0; j < c; j++) { f(c) iteraciones = O(g(n))

B[k][j] = array[k] + j; 3n (1 asignacion, vector, suma) = O(n)

}

J2= for (int j = c+1; j < array.length; j++) { f(c+1-array.length) iteraciones= (O(g(n)))

B[k][j − 1] = array[k] + j; 3n (1 asignación, vector y suma) = O(n)

}

}

F2 = for (int k = f + 1; k < array.length; k++) { (array.length – f + 1)) iteraciones

K1 = for (int j = 0; j < c; j++) { ((array.length – f + 1) \*(c) = O(g(n))

B[k − 1][j] = array[k] + j; 5n(1 resta, 3 vectores, 1 asignación, 1 suma) = O(n)

}

K2 = for (int j = c + 1; j < array.length; j++){ (array.length-f+1)\*c+1- array.length iteraciones

B[k − 1][j − 1] = array[k] + j; 5n(2 resta,3 vectores,1 asignación, 1 suma)= O(n)

}

}

return B; 1 iteracion (se ignora porque su orden es menor al orden de complejidad) = O(1)

**}**

**Entrada:** array, f, c

**Tamaño:** array.length

**Función de Complejidad:** Para realizar la función separé los términos en:

((F1 (J1)) + (F1 (J2))) + ((F2 (K1)) + (F1 (K2)))

((n\*g(n) + 3n) + (n\* g(n)+ 3n)) + ((n\*g(n) + 5n) + (n\* g(n)+ 5n))

(n\*\*2 + 3n + n\*\*2 + 3n) + (n\*\*2 + 5n + n\*\*2 + 5n)

6n + 2n\*\*2 + 10n 2n\*\*2

16n + 4n\*\*2

**Orden de Complejidad:**

El orden de complejidad es O(n\*\*2) (Función cuadrática) porque: O(1) c O(n) c O(n\*\*2)