Tiempo de Ejecución IV

Tiempo y Orden de ejecución

- Enunciado:
  - Calcular el **T(n)** de **countOverlap** para el peor caso, detallando los pasos seguidos para llegar al resultado.
  - Calcular el **O(n)** de la función del punto anterior justificando usando la definición de big-OH.

- Puntos importantes:
  - Considerar el “**peor de los casos**”
  - Calcular **O(n)** usando **Big-OH**

Tiempo y Orden de ejecución

- Puntos importantes:
  - hay dos métodos: uno **recursivo** y uno **iterativo**
  - **NO** hay un **N** explícito

```
public static int recu(int[] array, int count, int len) {
    if (len == 0)
        return 0;
    else
    if (array[len - 1] == count)
        return 1 + recu(array, count, len - 1);
    else
        return recu(array, count, len - 1);
}

public static void countOverlap(int[] arrayA, int[] arrayB) {
    int count = 0, calc = 0, tam = 0;
    if (arrayA.length == arrayB.length) {
        tam = arrayA.length;
        for (int i = 0; i < arrayA.length; i++)
            for (int j = 0; j < arrayB.length; j++)
                if (arrayA[i] == arrayB[j]) {
                    count++;
                    calc = calc + recu(arrayA, count, tam)
                            + recu(arrayB, count, tam);
                }
    }
    System.out.println("count:" + count + "-calc:" + calc);
}
```

Tiempo y Orden de ejecución

```
public static int recu(int[] array, int count, int len) {
    if (len == 0)
        return 0;
    else
    if (array[len - 1] == count)
        return 1 + recu(array, count, len - 1);
    else
        return recu(array, count, len - 1);
}

public static void countOverlap(int[] arrayA, int[] arrayB) {
    int count = 0, calc = 0, tam = 0;
    if (arrayA.length == arrayB.length) {
        tam = arrayA.length;
        for (int i = 0; i < arrayA.length; i++)
            for (int j = 0; j < arrayB.length; j++)
                if (arrayA[i] == arrayB[j]) {
                    count++;
                    calc = calc + recu(arrayA, count, tam)
                            + recu(arrayB, count, tam);
                }
    }
    System.out.println("count:" + count + "-calc:" + calc);

}
```

Tiempo constante

Tiempo expresado como sumatorias

Tiempo constante + 2 veces el tiempo de “recu”

T(n) del algoritmo “recu”

- **len** se reduce en 1 hasta llegar al caso base, por lo tanto **len** es nuestro N

$$\boxed{T'(n) = \begin{cases} c & , n = 0 \\ d + T'(n-1), & n > 0 \end{cases}}$$

(paso 1) T’(n) = d+T’(n-1)

(paso 2) T’(n) = d+d+T’(n-2)

(paso 3) T’(n) = d+d+d+T’(n-3)

…………

(paso i) T’(n) = id+T’(n-i)

n – i = 0 → i = n

= nd + T’(n-n) → nd + T’(0) → nd + c

$$\boxed{T'(n) = \begin{cases} c & , n = 0 \\ c + nd, & n > 0 \end{cases}}$$

T(n) del algoritmo "countOverlap"

- Debemos considerar que ambos arrays son de igual tamaño (**peor caso**).
- El **tamaño** del array es nuestro **N**

$$T(n) = b + \sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-1} (f + 2(dn + c)) \right) (sea\ g = f + 2c)$$

Las sumatorias evalúan N veces. El contenido de sumatoria no se ve afectado por el índice, puedo o no reescribir los índices de la sumatoria

$$= b + \sum_{i=1}^{n} \left(n(2dn + g) \right)$$

$$= b + \ n(n)(2dn + g)$$

$$= b + 2dn^3 + gn^2$$

O(n)

¿Es $b + d2n^3 + gn^2 <= kO(n^3)$ para todo $n>= n_0$?

$b <= k_0n^3$, con $k_0=b$ y para $n_0 = 1$

$2dn^3 <= k_1n^3$, con $k_1=2d$ y para $n_0 = 0$

$gn^2 <= k_2n^3$, con $k_2=g$ y para $n_0 = 0$

Luego…

$b + 2dn^3 + gn^2 <= (k_0 + k_1 + k_2) n^3$

Por lo tanto…

$b + 2dn^3 + gn^2 <= k O(n^3)$,

con $k = k_0+k_1+k_2 = b + 2d + g$ y para $n_0 =1$

Por lo tanto, $b + 2dn^3 + gn^2$ es $O(n^3)$

Otro cálculo de O(n)

- ¿Cuál es el O(n) de la siguiente expresión?

$$T(n) = 4n + 100Log_2(n) - Log_4(n)$$

- Utilizando Big-Oh debemos probar que

$$4n + 100Log_2(n) - Log_4(n) \leq kO(n), \quad \forall n \geq n_0$$

Otro cálculo de O(n)

$$4n + 100Log_2(n) - Log_4(n) \leq kO(n), \forall n \geq n_0$$

- Debemos verificar la desigualdad para **cada uno** de los términos

Otro cálculo de O(n) – primer término

$$\boldsymbol{4n} + 100Log_2(n) - Log_4(n) \leq kO(n), \forall n \geq n_0$$

$$4n \leq 4n, k_1 = 4 \ y \ \forall n_0$$

Otro cálculo de O(n) – segundo término

$$4n + \mathbf{100Log_2(n)} - Log_4(n) \leq kO(n), \forall n \geq n_0$$

$$100Log_2(n) \leq 100n,\ k_2 = 100\ y\ n_0 = 1$$

- Otra opción hubiera sido

$$100Log_2(n) \leq n, k_2 = 1\ y\ n_0 = 1000$$

Otro cálculo de O(n) – tercer término

$$4n + 100Log_2(n) - \boldsymbol{Log_4(n)} \leq kO(n), \forall n \geq n_0$$

- No es necesario considerar $-Log_4(n)$ puesto que es negativo
- Si logramos acotar el resto de la expresión, obviamente estaremos acotando el resto de la expresión “–” algo.

Otro cálculo de O(n) – sumando desigualdades

$$4n \leq 4n, k_1 = 4 \; y \; \forall n_0$$
$$100Log_2(n) \leq 100n, k_2 = 100 \; y \; n_0 = 1$$
$$4n + 100Log_2(n) \leq 4n + 100n$$
$$4n + 100Log_2(n) \leq 104n, n_0 = 1$$
$$4n + 100Log_2(n) \leq kO(n), k = 104 \; y \; n_0 = 1$$
$$4n + 100Log_2(n) - Log_4(n) \leq kO(n), k = 104 \; y \; n_0 = 1$$

Llegamos a demostrar que es O(n)