Analisis de algoritmos tarea 1

Daniel Dazarola, Universidad Diego Portales 31/08/2020

Número faltante

Listing 1: Solución 1.

```
1  static int solution1(int[] arr, int n) {
2      int sum_n = (n * n + n) / 2;              //O(1)
3      for(int i = 0; i < n; ++i)                //O(N)
4          sum_n -= arr[i]                       //O(1)
5      return sum_n;                             //O(1)
6  }
```

Se determina el valor de la sumatoria en O(1). Se le resta cada elemento ingresado O(n) lo que queda corresponde al número faltante.

Listing 2: Solución 2.

```
1  static int solution2(int[] arr, int n) {
2      boolean flag = false;
3      int num_faltante = 0;
4      for(int  i = 1; i < n; ++i) {
5          for(int j = 0; j < n; ++j) {
6              if(arr[j] == i){
7                  flag = true;
8                  break;
9              }
10         }
11         if(flag == false) {
12             num_faltante = i;
13             break;
14         }
15         flag = false;
16     }
17     return num_faltante;
18 }
```

En este code, para encontrar el número faltante, se recurre al viejo truco de 2 loops anidados, lo cual le da una complejidad O(N^2). El primer bucle recorre los enteros y el segundo el array, buscando que esté cada entero en el array, devolviendo así el entero no encontrado en el array.

Listing 3: Solución 3.

```
1  static int solution3(int[] arr, int n){
2          boolean[] table = new boolean[n];
3          int number = 0;
4          for (int i = 0; i < n - 1; ++i)      //O(n)
5              table[arr[i] - 1] = true;
6          for(int i = 0; i < n; ++i)           //O(n)
7              if(table[i] == false)
8                  number = i + 1;
9          return number;
10 }
```

Este código tiene complejidad $O(N)$, debido a que son 2 bucles separados. El algoritmo consiste en recorrer el array agregando los números en él como posiciones en un array de booleanos, luego devuelve la posición donde es falso, siendo este el número faltante.

Listing 4: Solución Diagnóstico.

```
1  static void solution(){
2      Scanner scan = new Scanner(System.in);
3      int N = scan.nextInt();
4      int sumaN = 0;
5      int sumaX = 0;
6      for(int i = 1; i < N+1; i++) {
7          sumaN += i;
8      }
9      int x;
10     for(int i = 0; i < N-1; i++) {
11         x = scan.nextInt();
12         sumaX += x;
13     }
14     System.out.println(sumaN - sumaX);
15 }
```

El presente código tiene complejidad $O(N)$, ya que usa 2 for loops para recorrer N. $O(N) + O(N) = O(N)$. Consiste en tener el resultado de la sumatoria de 1 hasta N, y luego restarle la suma de los números ingresados al programa.

Se puede optimizar quitando un $O(N)$ al cambiar el primer for por la formula de sumatoria, y en vez de sumar cada número, restarlos a la sumatoria. Quedaría igual a la solución 1.

Daño recibido por el mago (El mago y los dragones)

Listing 5: Solución 1 y Diagnóstico.

```
1  static int solution1(int ps[], int dps[], int n) {
2      int sum = 0;
3          dps_sum = 0;
4      for(int i = 0; i < n; ++i)
5          dps_sum += dps[i];
6      for(int i = 0; i < n; ++i){
7          sum += (dps_sum * ps[i]);
8          dps_sum -= dps[i];
9      }
10     return sum;
11 }
```

Mi código es exactamente igual a este, la única diferencia son los nombres de variables, los cuales corresponden a los getDPS, getPS, etc..., por lo que no veo conveniente colocarlo aquí. La complejidad es $O(N)$, porque son 2 for loops separados.

Consiste en guardar el daño bruto que hacen los dragones vivos en cada turno. Luego se suma el daño bruto multiplicado por la vida de cada dragón, ya que mientras vive hace dmg cada turno. Al morir un dragón no se le considera más, por lo que su dps es borrado del daño bruto. Finalmente se retorna el daño que recibe el mago.

Listing 6: Solución 1.

```
1  static int solution1(int ps[], int dps[], int n) {
2      int sum = 0;
3      for(int i = 0; i < n; ++i)
4          for(int j = i; j < n; ++j)
5              sum += (dps[j] * ps[i]);
6      return sum;
7  }
```

Este code tiene complejidad $O(N^2)$ ya que por cada dragón, recorre todos los dragones. Se suma por cada dragón el dps de cada dragón por los ps del dragón.