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, lo que queda corresponde al número faltante. Complejidad $O(N)$.

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) {             //O(n)
5          for(int j = 0; j < n; ++j) {          //O(N)
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 Tarea 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++) {           //O(N)
7          sumaN += i;
8      }
9      int x;
10     for(int i = 0; i < N-1; i++) {           //O(N)
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)           //O(N)
5          dps_sum += dps[i];
6      for(int i = 0; i < n; ++i){          //O(N)
7          sum += (dps_sum * ps[i]);
8          dps_sum -= dps[i];
9      }
10     return sum;
11 }
```

La complejidad es $O(N)$ en el peor caso, 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 2.

```
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)       //O(N^2)
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.

Listing 7: Solución 1 y Diagnóstico.

```
1
2  static int dmgRecibido(Dragon[] dragones, int N){
3      int totalDmg = 0;
4      int received = 0;
5      for(int i = 0; i < N; i++){
6          totalDmg += dragones[i].getDPS();
7      }
8      for(int i = 0; i < N; i++){
9          received += (totalDmg * dragones[i].getPS());
10         totalDmg = totalDmg - dragones[i].getDPS();
11     }
12     return received;
13 }
```

Mi código es exactamente igual a la solución 1, la única diferencia son los nombres de variables.

Unos y Ceros

Listing 8: Solución 1.

```
1  static void recursive(char[] str, int index) {
2      if (index == str.length) {
3          System.out.println(str);
4          return;
5      }
6      if (str[index] == '?') {
7          str[index] = '0';
8          recursive(str, index + 1);
9          str[index] = '1';
10         recursive(str, index + 1);
11         str[index] = '?';
12     }
13     else{
14         recursive(str, index + 1);
15     }
16 }
```

Este algoritmo en cada llamada recursiva revisa una posición del array, cuando encuentra un '?' se llama a si misma remplazando el signo por un '0' o '1' y dandole a esos llamados la siguiente posición (index+1). Se termina cuando llega al final del array.

En el peor caso se tiene un string muy grande compuesto de '?' solamente, donde se imprimen y retornan al llegar a su ultimo nivel (nivel del arbol que se forma), por lo que la complejidad debería ser $Log(N)$.

Listing 9: Solución Tarea 1.

```
static int bin(String s, String fin, int i) {
    if(i == s.length()) {
        System.out.println(fin);
        return;
    }
    while(s.charAt(i) != '?') {
        fin += s.charAt(i);
        i++;
    }
    bin(s, fin+'0', i+1);
    bin(s, fin+'1', i+1);
}
```

El código que implementé consiste en pasar 2 string, el binario incompleto y un string vacío para almacenar el resultado. A diferencia del código anterior, este algoritmo salta las llamadas recursivas cuando no hay un '?' usando un while, lo cual debe haber aumentado la complejidad.

Cuando es un string de sólo '?' la complejidad debería ser $Log(N)$, igual que el codigo propuesto anteriormente. Creo que el peor caso sería intercalando '?' con 1 y 0, porque entraría al while cuando no es '?' y llamaría recursividad cuando es. Por lo que entonces en el peor caso (que podría ser el average) se tiene $NLog(N)$, siendo N el correspondiente al while.

Algoritmo a

Listing 10: Algoritmo a.

```
static void algoa(int arr[]) {
        int n = arr.length;
        for(int i = 0; i < n-1; i++) {
            int min_idx = i;
            for(int j = i+1; j < n; j++)
                if(arr[j] < arr[min_idx]) min_idx = j;
            int temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
}
```

Este algoritmo ordena números en forma creciente, para esto usa bucles anidados. Se busca la posición del menor elemento y luego se intercambia en la primera posición del array, esto se repite hasta que el array queda ordenado.

Tiene complejidad $O(N^2)$

Algoritmo b

Listing 11: Algoritmo b.

```
1  static void algo(int[] array, int n) {
2      int i, j, size_table;
3      //int max = ~0, min = -(~0);
4      int max = -999999999;
5      int min = 999999999;
6      for(i = 0; i < n; ++i) {
7          if (max < array[i]) max = array[i];
8          if (min > array[i]) min = array[i];
9      }
10     size_table = max - min + 1;
11     int[] table = new int[size_table];
12     for(i = 0; i < n; ++i)
13         ++table[array[i] - min];
14     for(i = 0, j = 0; i < size_table; ++i) {
15         if ((table[i]--) > 0){
16             array[j] = (i + min);
17             j++;
18         }
19     }
20 }
```

Este código también ordena números de forma creciente, siendo mucho más eficiente. Primero busca el máx y el min en el array, usándolos para hacer un rango correspondiente al tamaño de un nuevo array "table". Las posiciones de table corresponden al valor numérico, y estas son devueltas en orden como número en el array inicial.

Posee $O(N)$ de complejidad, ya que consiste en recorrer el array de manera lineal.

Euclidean Algorithm

Listing 12: Algoritmo c.

```
1 static int algoc(int a, int b){
2     if (b == 0) return a;
3     return algoc(b, a%b);
4 }
```

Este algoritmo busca el máximo común divisor entre a y b, es conocido como el "Euclidean Algorithm", consiste en llamados recursivos donde se obtiene el módulo de a entre b. En palabras más simples, a es dividido por b manteniendo de resultado enteros, si no se puede obtener un entero a pasa a ser el divisor y b el resto. Se sale de la recursividad cuando es encontrado un divisor de ambos números.

La complejidad es $O(Log(N))$ ya que es dividido en cada llamada recursiva, y el peor caso es cuando a y b son números primos muy grandes.