

Taller Big O

Yudy Carolina Guevara Cely

September 17, 2023

1 Introduction

La notación Big O es una manera de describir la rapidez o complejidad de un algoritmo dado, es decir te cuenta el número de operaciones que hará un algoritmo. Toma su nombre de la "O grande" en frente del número estimado de operaciones. Lo que la notación Big O no te dice es la rapidez del algoritmo en segundos. Hay demasiados factores que influyen en el tiempo que tarde en ejecutarse un algoritmo

2 Requirements

se debia analizar la complejidad algoritmica de el codigo linea por linea para poder entender el funcionamiento de cada codigo

3 Proces

En las siguientes figuras se puede ver evidenciado el proceso de evaluación de rendimiento para 15 algoritmos bajo la notación Big O

Code #1 Declaraciones de variables Operadores lógicos / palabras
for (int i=0; i<n; i++) { }
* ya que el tamaño es directamente proporcional al tamaño que quese ejecutar n veces su complejidad es de $O(n)$

Code #2
for (int i=0; i<n; i++) { - Línea 1 tenemos un ciclo For es decir se repite n veces = complejidad $O(n)$
 for (int j=0; j<m; j++) { Línea 2 tenemos un ciclo anidado al cuadrado $(O(nxm))$
 }
 }
Resultado
 $= O(n) + O(nxm)$
 $= O(nxm)$

Code #3
for (int i=0; i<n; i++) { linea 1 = Ciclo que se ejecuta n veces. Por lo que su complejidad sería $O(n)$
 for (int j=i; j<n; j++) { linea 2 = Ciclo anidado que se ejecuta n-i veces ya que cuando entra en el segundo ... lleva a restar 1 ya que en el primer for el i toma un valor por lo que en el segundo reduce el numero de iteraciones, $O(n \times n)$
 }
 }
Resultado
 $= O(n) + O(n^2)$
 $= O(n^2)$

Code #4
1 int index=-1;
2 for (int i=0; i<n; i++) { linea 1 = tiene complejidad constante $O(1)$ porque es asignación
3 if (array[i] == target) { linea 2 = complejidad $O(1)$ por que se ejecuta n veces
4 index = i; linea 3 = el if esta haciendo de condicional pero dentro de él hay scoping que para así asignarle un nuevo valor a i
5 break; linea 4 = asignando valor a i por lo tanto es $O(1)$
 }
Resultado
 $| O(n) + O(1) + O(1) = O(n)$

Powered by  CamScanner

Figure 1: códigos del 1 al 4 .

Code #5

```

① int left = 0, right = n-1, index = -1;
② while (left <= right) {
    ③     int mid = left + (right-left)/2;
    ④     if (array[mid] == target) {
        ⑤         index = mid;
        ⑥         break;
    ⑦     } else if (array[mid] < target) {
        ⑧         left = mid + 1;
    ⑨     } else {
        ⑩         right = mid - 1;
    ⑪     }
    ⑫ }
  
```

Resultado: cuando es un while para evaluar la complejidad referir consider como cuenta la variable contadora.

$$O(1) + O(n-j) + O(j) + O(1) + O(1) + O(1) + O(1)$$

$$+ O(1) = \text{Logarítmica}$$

$$O(\log n)$$

AND

Linea 1: asignación de valores a las variables. $O(1)$
 Linea 2: $O(\log n)$ por que las variables $left$ y $right$ crecen de forma logarítmica.
 Linea 3: asignación de valores por lo tanto su complejidad es constante $O(1)$
 Linea 4 al ser el if como condicional su complejidad es constante.
 Linea 5: asignación asignando el valor $O(1)$
 Linea 6: romper el ciclo while complejidad $O(1)$
 Linea 7: al ser el condicional tendría complejidad $O(1)$
 Linea 8: asignación de variables con complejidad $O(1)$
 Linea 9: es la continuación de la instrucción $O(1)$
 Linea 10: asignación de valores, complejidad $O(n)$
 Por eso debemos evaluar como crecen las variables en proceso de while.

Code #6 *

```

① int row = 0, col = matrix[0].length - 1, indexRow = -1, indexCol = -1;
② while (row < matrix.length && col >= 0) {
    ③     if (matrix[row][col] == target) {
        ④         indexRow = row;
        ⑤         indexCol = col;
        ⑥         break;
    ⑦     } else if (matrix[row][col] < target) {
        ⑧         row++;
    ⑨     } else {
        ⑩         col--;
    ⑪     }
  
```

Resultado:

$$O(n) + O(matrix.length \times Col > 0) + O(n) + O(1) + O(1) + O(1) + O(1)$$

$$+ O(1) = \text{Complejidad lineal}$$

$$O(n \times \text{matrix.length} \times Col > 0)$$

AND

Linea 1: asignación de variables $O(1)$
 Linea 2: Por lo que es un bucle, q el término q # de ejecuciones depende de la matriz y q depende por lo q es serca.
 Linea 3: $O(\text{matrix.length} \times Col > 0) = O(n)$
 Linea 4: condicional compara el valor q tiene la matriz en esa posición compuesta con el target
 $O(1)$
 Linea 5, 6: en la linea 4 y 5 asigna valores q en el 6 rompe el bucle todo son complejidad $O(1)$
 Linea 7: condicional q evalua en donde minada posición para bajar q mas q target para entrar $O(1)$
 Linea 8: asigna valor $O(1)$
 Linea 9, 10: else q $col--$ $O(1)$

Powered by CamScanner

Figure 2: códigos de el 5 y 6.

Code #7

```
① void bubbleSort (int []array) {  
②     int n = array.length;  
③     for (int i = 0; i < n - 1; i++) {  
④         for (int j = 0, j < n - i, j++) {  
⑤             if (array [j] > array [j + 1]) {  
⑥                 int temp = array [j];  
⑦                 array [j] = array [j + 1];  
⑧                 array [j + 1] = temp;  
            }  
        }  
    }  
}
```

(1 linea ①) = está haciendo que el método reciba un arreglo entero creando el método
O(n)
(2 linea 2) = está asignando a n el valor de el arreglo O(1)
(3 linea 3) = complejidad $O(n - 1)$ linea 1
(4 linea 4) = el bucle un for anidado
Se le suma la complejidad del for anterior ($O(n)$) depende de el otro
(5 linea 5) = complejidad O(n)
(6 linea 6) y 8 = está realizando asignaciones O(1)
 $O_1 + O_2 + O_{(n-1)} + O(n) + O_1$
+ O(1) =
 $O(n^2)$

Code #8

```
① void selectionSort (int []array) {  
②     int n = array.length;  
③     for (int i = 0, i < n - 1, i++) {  
④         int minIndex = i;  
⑤         for (int j = i + 1; j < n, j++) {  
⑥             if (array [j] < array [minIndex]) {  
⑦                 minIndex = j;  
            }  
        }  
        int temp = array [i];  
        array [i] = array [minIndex];  
        array [minIndex] = temp;  
    }  
}
```

la complejidad de este algoritmo sería de tipo $O(n^2 - 1)$

(1) como la linea 2 es un método sería constante O(1)
(2) como declarando una variable es constante O(1)
(3) tenemos un ciclo for que tendría complejidad linea 1 $O(n - 1)$
(4) en la linea 4 encontramos una declaración de constantes por lo tanto es constante O(1)
(5) vemos el otro for anidado con la misma variable n por lo que su complejidad sería $O(n)$ (ya que comienza desde el elemento siguiente al actual)
(6) se encarga de comparar elementos actual con el mínimo actual al estar comparando sería complejidad constante O(1)
(7) constante O(1)
(8)

log - las dos variables nose modifican el tiempo
explicar - ciclos anidados va a depender de la cantidad del ciclo interno * el el ciclo exterior

Powered by  CamScanner

Figure 3: codigos de el 7 y 8..

Code #10.

```

① void mergesort (int[] array, int left, int right) {
    ② if (left < right) {
        ③ int mid = left + (right - left) / 2;
        ④ → mergesort (array, left, mid);
        ⑤ mergesort (array, mid + 1, right);
        ⑥ merge (array, left, mid, right);
    }
}

```

linea 1 = creación del método con parámetros
 incremento el método por lo que sería $O(n)$
 ② deseable que tiene una condición
 comparar variables. Poco en este caso
 vemos que al final se está fragmentando
 en 2 ademas de usar un merge sort lo que
 sugiere que es $O(1)$ dependiente de el $O(n)$

Resultado
 $O(1) + O(1) + O(1) + O(\log n) + O(n) = O(n)$

Code #9

```

① void insertionSort (int[] array) {
    ② int n = array.length;
    ③ for (int i = 1; i < n; i++) {
        ④ int key = array[i];
        ⑤ int j = i - 1;
        ⑥ while (j >= 0 && array[j] > key) {
            ⑦ array[j + 1] = array[j];
            ⑧ j--;
        }
        ⑨ array[j + 1] = key;
    }
}

```

① Declaración del método constante $O(1)$
 ② Declaración de variable n $O(1)$
 ③ Ciclo for iniciando en i hasta n . $O(n)$
 Por que se repite n veces
 ④ Declaración j para almacenar el array $O(1)$
 ⑤ Declaración variable j
 ⑥ es un bucle que si se cumple su parámetro.
 que se puede ejecutar hasta j veces
 $O(j)$
 ⑦ Asignación dentro de el bucle. $O(1)$
 ⑧ Variable $O(1)$,
 ⑨ asignación final después del bucle. $O(1)$

Resultado
 $O(1) + O(1) + O(n) + O(1) + O(n) + O(1) + O(n) = O(n^2)$.

Code #11

```

① void quicksort (int[] array, int low, int high) {
    ② if (low < high) {
        ③ int pivotIndex = partition (array, low, high);
        ④ quickSort (array, low, pivotIndex - 1);
        ⑤ quickSort (array, pivotIndex + 1, high);
    }
}

```

① Creación del método con
 Parámetros $O(n)$ función
 ② partición
 ③ condicional if constante $O(1)$
 ④ se encarga de dividir el
 array en subarreglos. en
 función de y pivot ($\frac{1}{2}$ en ambos
 arreglos $O(n)$) para que
 depende de la implementación
 partición.

4 u 5 bases llamadas recursivas
 particiones las sublistas
 más pequeñas de izq a derecha
 $O(n)$

Powered by  CamScanner

Figure 4: códigos de el 9 10 y 11..

Code # 12

```

int fibonacci (int n) {
    if (n <= 1) {           ① inicialización del método
        return n;           ② condicional if con parámetros n O(1)
    }
    int [] dp = new int [n+1]; ④ creación del arreglo O(n)
    dp[0] = 0;                ⑤ asignar valores iniciales en el arreglo
    dp[1] = 1;                ⑥ de tipo constante O(1)
    for (int i = 2; i <= n; i++) { ⑦ se ejecuta hasta que lleguen O(n)
        dp[i] = dp[i-1] + dp[i-2]; ⑧ O(n)
    }
    return dp[n];             ⑨ como retorna el valor n es constante O(1)
}
    
```

Code # 13

```

void linearSearch (int [] array, int target) {
    for (int i = 0; i < array.length; i++) { ① creación de método con parámetros O(1)
        if (array[i] == target) { ③ O(1)
            return; // encontrado. ④ O(1)
        }
    }
    ⑤ //no encontrado. ⑥ Resultado O(n)
}
    
```

Code # 14

```

int binarySearch (int [] sortedArray, int target) { O(1)
    int left = 0, right = sortedArray.length - 1; ⑦ O(0)
    while (left <= right) { ⑧ O(log n)
        int mid = left + (right - left) / 2; ⑨ O(1)
        if (sortedArray[mid] == target) { ⑩ O(1)
            return mid; // índice del elemento encontrado. O(1)
        } else if (sortedArray[mid] < target) { ⑪ O(1)
            left = mid + 1; ⑫ O(1)
        } else { ⑬ O(1)
            right = mid - 1; ⑭ O(1)
        }
    }
}
    
```

Powered by  CamScanner

Figure 5: codigos de el 12 al 14..

```
return -1; // elementos no encontrados.  
?  
Code # 15 .  
int factorial (int n) { O(1)  
    if (n==0 || n==1) {  $\rightarrow$  si no es 0 ó 1 es una comparación O(n)  
        return 1; O(1)  
    }  
    return n * factorial (n-1); O(1),  
}  
Resultado O(n)
```

Powered by  CamScanner

Figure 6: código 15 .

4 Conclusions

En conclusión se puede ver como las variables son constantes por lo tanto estas siempre tendrá una complejidad de $O(1)$ mientras que cuando no sabemos cuándo la condición esta(n) abierta es decir n nuestra complejidad será de complejidad lineal o (n) ya que a medida que crecen las iteraciones crece el tiempo de ejecución por ultimo queda entendido que cuando en un Código encontramos divisiones (split)este será de tipo logarítmico ya que divide la complejidad o tiempo del problema y queda entendido que cuando un Código es factorial que puede existir este será inejecutable en una maquina