

Big-O notation

Juan Esteban Pinzon Preciado
Fundación Universitaria Konrad Lorenz

I. INTRODUCCIÓN

A fin de comprender el rendimiento y la eficiencia de un conjunto de algoritmos, es crucial llevar a cabo un análisis detallado de su complejidad temporal. Este análisis se realiza comúnmente utilizando la notación Big-O, que nos proporciona una visión general de cómo aumenta el tiempo de ejecución de un algoritmo a medida que crece el tamaño de entrada.

II. EJERCICIO DE ANÁLISIS DE COMPLEJIDAD DE ALGORITMOS

En esta sección se describirá la metodología utilizada.

II-A. Código número uno

La inicialización `int i = 0;` es una operación de tiempo constante, $O(1)$. La condición `i < n` se evalúa n veces y la operación `i++` se ejecuta n veces. Por lo tanto, la complejidad de este bucle `for` es $O(n)$.

Complejidad de $O(n)$

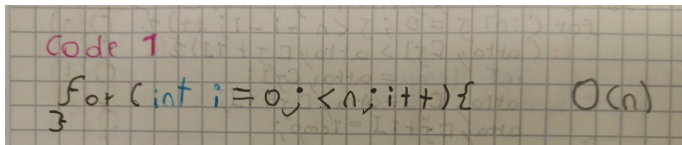


Figura 1. Ejemplo código 1

II-B. Código número dos

Para el bucle externo `for`, la complejidad es $O(n)$ debido a que la inicialización y la operación de incremento son de tiempo constante.

Para el bucle interno, la complejidad es $O(m)$ debido a que la inicialización y la operación de incremento también son de tiempo constante. En el contexto del bucle externo, esto resulta en una complejidad total de $O(n) + O(m)$.

Complejidad de $O(n)$:

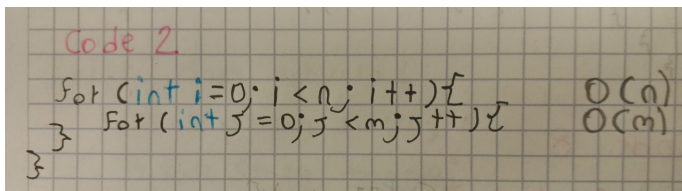


Figura 2. Ejemplo código 2

II-C. Código número tres

La línea representa un bucle que se ejecuta " n " veces, por lo que su complejidad es $O(n)$. El bucle interno se ejecuta en un patrón triangular, realizando aproximadamente n^2 iteraciones en total. Por lo tanto, la complejidad del bucle interno es $O(n^2)$. Esto resulta en una complejidad total de $O(n) + O(n^2)$.

Complejidad de $O(n^2)$:

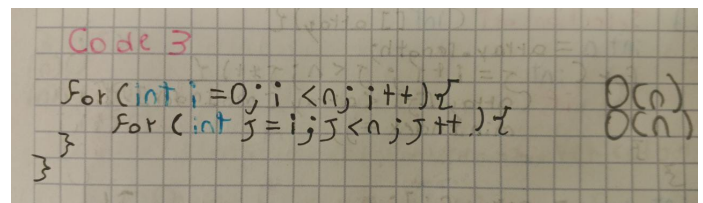


Figura 3. Ejemplo código 3

II-D. Código número cuatro

1. La línea `'int index = -1;'` es una asignación de tiempo constante: $O(1)$.

2. El bucle `'for (int i = 0; i < n; i++)'` itera sobre un arreglo `'array'` de longitud `'n'`, lo que da como resultado una complejidad en el peor caso de $O(n)$.

3. Dentro del bucle, la comparación `'if (array[i] == target)'` es una operación de tiempo constante y, en el peor caso, se realiza hasta `'n'` veces: $O(n)$.

4. La asignación `'index = i;'` es una operación de tiempo constante y se ejecuta una vez en el peor caso: $O(1)$.

5. El `'break;'` se ejecutará una vez cuando se encuentre el elemento buscado, lo que también es una operación de tiempo constante en el peor caso: $O(1)$.

Complejidad de $O(n)$

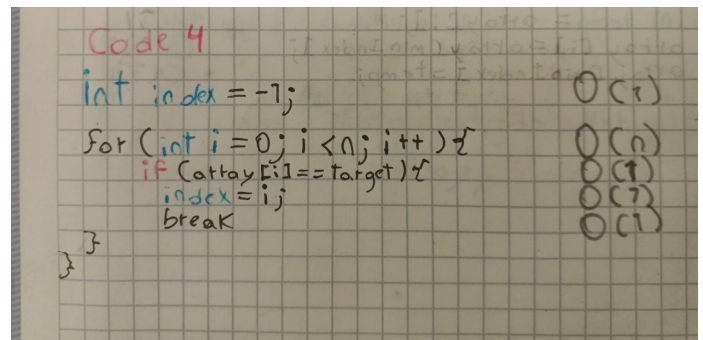
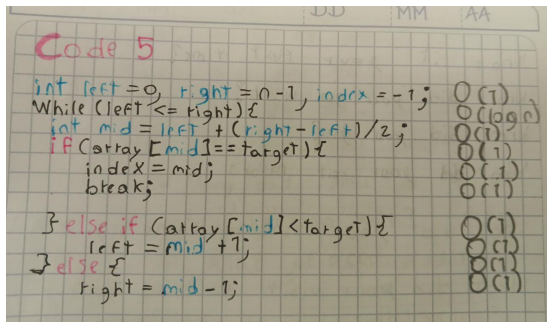


Figura 4. Ejemplo código 4

II-E. Código número cinco

1. La línea es una serie de asignaciones de tiempo constante: $O(1)$.
2. El bucle 'while (left <= right)' implementa la búsqueda binaria. La búsqueda binaria tiene una complejidad de tiempo de $O(\log n)$.
3. La línea calcula el índice medio del rango actual. Esta operación es de tiempo constante: $O(1)$.
4. La comparación es una operación de tiempo constante y se ejecuta en el peor caso una vez: $O(1)$.
5. Las asignaciones son operaciones de tiempo constante que también se ejecutan en el peor caso una vez: $O(1)$.
6. El 'break;' se ejecutará una vez cuando se encuentre el elemento buscado : $O(1)$.

Complejidad de $O(\log n)$



```

Code 5
int left=0, right=n-1, index=-1; O(1)
while (left <= right) { O(log n)
    int mid = left + (right-left)/2; O(1)
    if (array[mid] == target) { O(1)
        index = mid; O(1)
        break; O(1)
    } else if (array[mid] < target) { O(1)
        left = mid + 1; O(1)
    } else { O(1)
        right = mid - 1; O(1)
    }
}

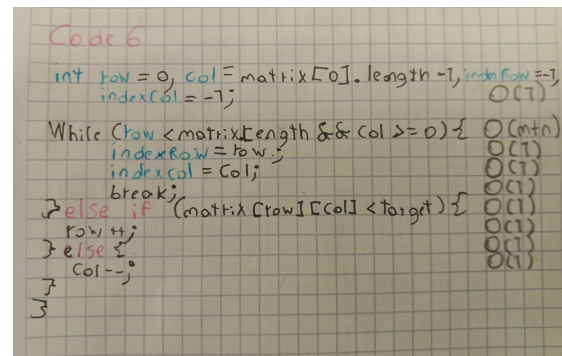
```

Figura 5. Ejemplo código 5

II-F. Código número seis

1. La línea es una serie de asignaciones de tiempo constante: $O(1)$.
2. El bucle 'while se ejecutará hasta que 'row' sea mayor o igual al número de filas de la matriz 'matrix' y 'col' sea mayor o igual a cero. En el peor caso, este bucle se ejecutará un máximo de 'm + n' veces, donde 'm' es el número de filas y 'n' es el número de columnas de la matriz. $O(n)$
3. La comparación if es una operación de tiempo constante y se ejecuta en el peor caso una vez: $O(1)$.
4. Las asignaciones son operaciones de tiempo constante que también se ejecutan una vez: $O(1)$.
5. Las asignaciones son operaciones de tiempo constante que se ejecutan en el peor caso una vez: $O(1)$.
6. El 'break;' se ejecutará una vez cuando se encuentre el elemento buscado: $O(1)$

Complejidad de $O(n)$



```

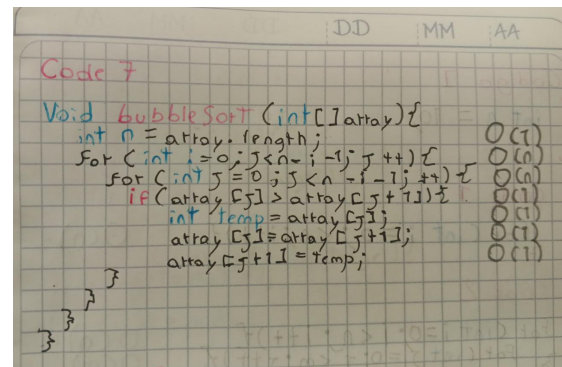
Code 6
int row=0, col=matrix[0].length-1, indexRow=-1, indexCol=-1; O(1)
while (row < matrix.length && col >= 0) { O(m+n)
    indexRow = row; O(1)
    indexCol = col; O(1)
    break; O(1)
} else if (matrix[row][col] < target) { O(1)
    row++; O(1)
} else { O(1)
    col--; O(1)
}
}

```

Figura 6. Ejemplo código 6

II-G. Código número siete

Aquí se explicará el proceso A.



```

Code 7
void bubbleSort(int[] array) { O(1)
    int n = array.length; O(1)
    for (int i=0; i<n-1; i++) { O(n)
        for (int j=0; j<n-i-1; j++) { O(n)
            if (array[j] > array[j+1]) { O(1)
                int temp = array[j]; O(1)
                array[j] = array[j+1]; O(1)
                array[j+1] = temp; O(1)
            }
        }
    }
}

```

Figura 7. Ejemplo código 7

II-H. Código número ocho

1. La línea calcula la longitud del arreglo 'array', lo cual es una operación de tiempo constante: $O(1)$.
2. El bucle externo veces en el peor caso, ya que en cada iteración se coloca el elemento más pequeño en la posición correcta. La complejidad de este bucle es $O(n)$.
3. Dentro del bucle externo, se inicializa 'minIndex' en cada iteración, lo cual es una operación de tiempo constante: $O(1)$.
4. El bucle interno desde 'i + 1' hasta 'n-1', lo que significa que se ejecuta n veces en la primera iteración del bucle externo, luego 'n-i-2' veces en la segunda iteración, y así sucesivamente. En el peor caso lo que resulta en una complejidad de $O(n^2)$.
5. Dentro del bucle interno, la comparación es una operación de tiempo constante: $O(1)$.
6. Si se encuentra un nuevo mínimo en el bucle interno, se actualiza 'minIndex', lo cual es una operación de tiempo constante: $O(1)$.
7. Fuera del bucle interno, se realiza el intercambio de elementos en el arreglo, lo cual consiste en tres operaciones de tiempo constante: $O(1)$.

complejidad total de $O(n^2)$.

```

Code 9

void insertionsort (int C[] array) {
    int n = array.length;
    for (int i = 1; i < n; i++) {
        int key = array[i];
        int j = i - 1;
        while (j >= 0 && array[j] > key) {
            array[j+1] = array[j];
            j--;
        }
        array[j+1] = key;
    }
}

```

```

Code 10

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

```

```

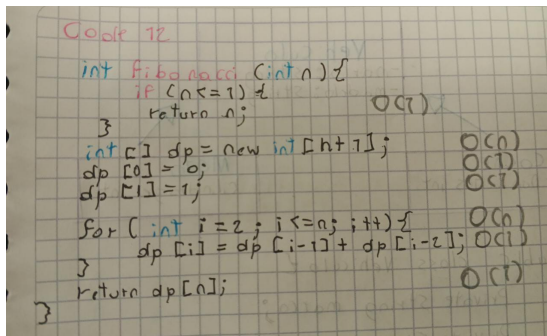
Code 11
void quickSort (int larray, int low, int high) {
    if (low < high) {
        int pivotIndex = Partition (array, low, high); // O(1)
        quickSort (array, low, pivotIndex - 1); // O(n)
        quickSort (array, pivotIndex + 1, high); // O(n)
    }
}

```

1. La línea 'if (n <= 1)' es una comparación de tiempo constante y el retorno en ese caso también es de tiempo constante: $O(1)$.

2. La creación del arreglo 'dp' de longitud 'n + 1' es una operación de tiempo lineal en relación con 'n': $O(n)$.
3. Las asignaciones 'dp[0] = 0;' y 'dp[1] = 1;' son de tiempo constante: $O(1)$.
4. El bucle 'for (int i = 2; i <= n; i++)' itera 'n-1' veces en el peor caso, ya que comienza en 'i = 2' y va hasta 'n'. La complejidad de este bucle es $O(n)$.
5. Dentro del bucle, la asignación 'dp[i] = dp[i - 1] + dp[i - 2];' es de tiempo constante: $O(1)$.
6. Finalmente, el retorno 'dp[n]' es una operación de tiempo constante: $O(1)$.

Complejidad de $O(n)$



```

Code 12
int Fibonacci (int n) {
    if (n <= 1) {
        return n;
    }
    int[] dp = new int[n+1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}

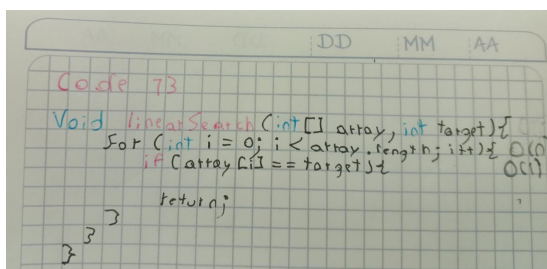
```

Figura 12. Ejemplo código 12

II-M. Código número trece

1. El bucle 'for (int i = 0; i < array.length; i++)' itera a través de cada elemento en el arreglo 'array'. En el peor caso, si el valor 'target' no se encuentra en el arreglo y es necesario recorrer todo el arreglo hasta el final, este bucle se ejecutará 'n' veces, donde 'n' es la longitud del arreglo. La complejidad de este bucle es $O(n)$ en el peor caso.
2. Dentro del bucle, la comparación 'if (array[i] == target)' es una operación de tiempo constante, ya que simplemente compara dos valores: $O(1)$.

Complejidad de $O(n)$



```

Code 13
void linearSearch (int[] array, int target) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == target) {
            return;
        }
    }
}

```

Figura 13. Ejemplo código 13

II-N. Código número catorce

1. Las asignaciones 'int left = 0' y 'int right = sortedArray.length - 1' son operaciones de tiempo constante: $O(1)$.
2. El bucle 'while (left <= right)' implementa la búsqueda binaria. En el peor caso, este bucle se ejecutará hasta que

'left' sea mayor que 'right'. La búsqueda binaria tiene una complejidad de tiempo de $O(\log n)$ en el peor caso, donde 'n' es la longitud del arreglo.

3. La línea 'int mid = left + (right - left) / 2;' calcula el índice medio 'mid' del rango actual. Esta operación es de tiempo constante: $O(1)$.

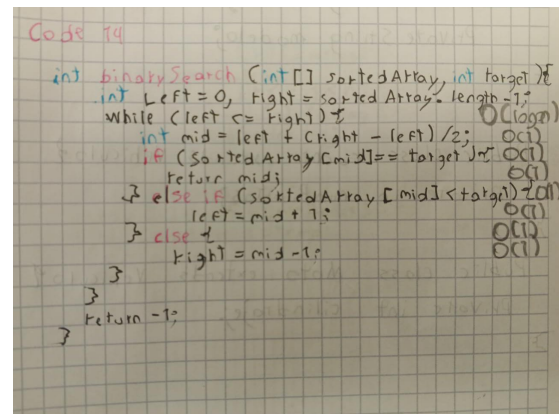
4. La comparación 'if (sortedArray[mid] == target)' es una operación de tiempo constante y se ejecuta en el peor caso una vez: $O(1)$.

5. Las asignaciones 'left = mid + 1;' y 'right = mid - 1;' son operaciones de tiempo constante que también se ejecutan en el peor caso una vez: $O(1)$.

6. El retorno 'mid' cuando se encuentra el elemento es de tiempo constante: $O(1)$.

7. Si el elemento no se encuentra y el bucle se ejecuta hasta que 'left' sea mayor que 'right', el retorno '-1' indicando que el elemento no se encontró también es de tiempo constante: $O(1)$.

Complejidad de $O(n \log n)$



```

Code 14
int binarySearch (int[] sortedArray, int target) {
    int left = 0, right = sortedArray.length - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (sortedArray[mid] == target) {
            return mid;
        } else if (sortedArray[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}

```

Figura 14. Ejemplo código 14

II-Ñ. Código número quince

1. La condición 'if (n == 0 || n == 1)' es una operación de tiempo constante, ya que simplemente verifica si 'n' es igual a 0 o 1: $O(1)$.

2. El retorno '1' cuando 'n' es 0 o 1 también es una operación de tiempo constante: $O(1)$.

3. La llamada recursiva 'return n * factorial(n - 1);' es el corazón de la función y es donde ocurre la recursión. Cada vez que se realiza una llamada recursiva, 'n' se decrementa en 1 hasta llegar a 1 o 0, momento en el cual se detiene la recursión. En el peor caso, esto implica que la función se llama 'n' veces antes de alcanzar el caso base. Por lo tanto, la complejidad de tiempo de esta llamada recursiva es $O(n)$ en el peor caso.

Complejidad de $O(n)$

Handwritten code on a grid background. The title 'Code 15' is written in red at the top left. The code is written in blue and red ink. To the right of the code, the time complexity for each line is written in black ink.

```

Code 15

int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    return n * factorial(n-1);
}

```

Complexity analysis written to the right of the code:

- $O(1)$ (next to the if condition)
- $O(1)$ (next to the return 1 statement)
- $O(n)$ (next to the recursive call return statement)

Figura 15. Ejemplo código 15

III. CONCLUSIONES

En este estudio, evaluamos la complejidad de varios algoritmos mediante la notación Big-O. Descubrimos que su eficiencia varía ampliamente: algunos son lineales ($O(n)$), otros cuadráticos ($O(n^2)$), y algunos logarítmicos ($O(\log n)$).

Este análisis resalta la importancia de seleccionar algoritmos adecuados para diferentes tareas, considerando el tamaño de los conjuntos de datos. La notación Big-O proporciona una herramienta esencial para tomar decisiones fundamentadas sobre la eficiencia de los algoritmos.