

Taller Uno Estructura De Datos

Lisseth Tatiana Quilindo Patiño
Fundación Universitaria Konrad Lorenz

Abstract—Este informe analiza varios códigos y calcula su complejidad algorítmica utilizando la notación Big-O. Se explorarán códigos simples como bucles ‘for’, algoritmos de búsqueda y ordenamiento, así como algoritmos más complejos como el cálculo de Fibonacci y factoriales. El objetivo es comprender cómo evaluar la eficiencia de los algoritmos en términos de tiempo de ejecución en el peor caso.

I. INTRODUCCIÓN

Se evalúa la eficiencia de los algoritmos con términos Big-O, se ejecutan y explican los códigos del 1 al 15, esta notación proporciona una descripción asintótica del tiempo de ejecución en función del tamaño de entrada, algoritmos de búsqueda y ordenamiento, así como algoritmos más complejos como el cálculo de Fibonacci y factoriales.

II. RESULTADOS

A continuación, se presentan los resultados de analizar la complejidad algorítmica de los códigos proporcionados:

A. Código 1

El primer código es un bucle ‘for’ simple que recorre una matriz de longitud n . La complejidad algorítmica es $O(n)$.

Código 1:
`for (int i=0; i<n; i++) { // O(1)`
 constante

- Esta línea de código crea un bucle “for” que se ejecuta “n” veces, con “i” incrementándose de 0 a “n-1”. Dado que el cuerpo del bucle es constante.
- El incremento es constante y no depende del valor “n”, el número de iteraciones será el mismo.

Código 2:
`for (int i=0; i<n; i++) { // O(n)`
 `for (int j=0; j<m; j++) { // O(m)`
 } la complejidad total
 $\therefore O(n \cdot m)$

Fig. 1: Primer código.

B. Código 2

El segundo código contiene dos bucles ‘for’ anidados. Ambos bucles recorren matrices de longitud n y m , respectivamente. La complejidad algorítmica es $O(n \cdot m)$.

Código 2:
`for (int i=0; i<n; i++) { // O(n)`
 `for (int j=0; j<m; j++) { // O(m)`
 } la complejidad total
 $\therefore O(n \cdot m)$

- Crea dos bucles anidados el bucle exterior se ejecuta “n” veces y, en cada iteración el bucle interno se ejecuta “m” veces. Por lo tanto también debido a esto la complejidad es $O(n \cdot m)$.

Fig. 2: Segundo código.

C. Código 3

El tercer código también contiene dos bucles ‘for’ anidados, pero el segundo bucle comienza en el índice actual del primer bucle. La complejidad algorítmica es $O(\frac{n(n+1)}{2}) = O(n^2)$.

Código 3:
`for (int i=0; i<n; i++) { // O(n)`
 `for (int j=i; j<n; j++) { // O(n)`
 }
 $\therefore O(n) + O(n) = O(n^2)$

El bucle exterior se ejecuta “n” veces y en cada iteración el bucle interior se ejecuta desde “i” hasta “n”.

Fig. 3: Tercero código.

D. Código 4

Este código busca un elemento en una matriz de longitud n utilizando un bucle ‘for’. En el peor caso, el elemento no se encuentra y la complejidad algorítmica es $O(n)$.

Código 4:
`int index; // O(1)`
`for (int i=0; i<n; i++) { // O(n)`
 `if (matrix[i] == target) { // O(1)`
 `index = i; // O(1)`
 `}`
`} // O(1)`
 $\therefore O(n)$

El bucle for se ejecuta “n” veces y en cada iteración se realizan 3 operaciones constantes: la asignación y el break también son operaciones constantes.

5. int left = 0; right = n - 1; index = -1; inicializá las variables para marcar los límites del intervalo y para almacenar la posición del elemento en el bucle se creará matriz el intervalo

Fig. 4: Cuarto código.

E. Código 5

El código 5 implementa la búsqueda binaria en una matriz ordenada de longitud n . La complejidad algorítmica es $O(\log n)$ en el peor caso.

```

Código 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)
    }
}
if (index != -1) {
    cout << "Element found at index: " << index; //O(1)
} else {
    cout << "Element not found"; //O(1)
}
    
```

Fig. 5: Quinto código.

F. Código 6

Este código busca un elemento en una matriz bidimensional de tamaño $m \times n$. En el peor caso, recorre ambas dimensiones, por lo que la complejidad algorítmica es $O(m + n)$.

```

Código 6.

int col = 0, row = matrix[0].length - 1, indexRow = -1, indexCol = -1; //O(1)
while (row < matrix.length || col > 0) { //O(m+n)
    if (matrix[row][col] == target) { //O(1)
        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)
    }
}
cout << "Element found at index: (" << indexRow << ", " << indexCol); //O(1)
    
```

Fig. 6: Sexto código.

G. Códigos 7, 8 y 9

```

Código 7.

void bubbleSort(int[] array) { //O(n^2)
    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)
            }
        }
    }
}

O(n) for (int i = 0; i < n - 1; i++) { O(n) for (int j = 0; j < n - i - 1; j++) { O(1) 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) } } }

int n = array.length; obtiene la longitud del arreglo
el bucle exterior se ejecuta n veces siendo n la
longitud del arreglo, en bucle interno se
ejecuta n-1 veces. if (array[j] > array[j+1])
comprueba si el elemento actual es mayor que el siguiente
crea una variable temporal para realizar el swap
swap el valor de los elementos
    
```

Fig. 7: Septimo código.

```

Código 8.

void SelectionSort(int[] array) { //O(n^2)
    int n = array.length; //O(1)
    for (int i = 0; i < n - 1; i++) { //O(n)
        int minIndex = i; //O(1)
        for (int j = i + 1; j < n; j++) { //O(n)
            if (array[j] < array[minIndex]) { //O(1)
                minIndex = j; //O(1)
            }
        }
        int temp = array[i]; //O(1)
        array[i] = array[minIndex]; //O(1)
        array[minIndex] = temp; //O(1)
    }
}

O(n) for (int i = 0; i < n - 1; i++) { O(n) for (int j = i + 1; j < n; j++) { O(1) if (array[j] < array[minIndex]) { O(1) minIndex = j; O(1) } } }

int n = array.length; calcula la longitud del arreglo y el bucle
externo se ejecuta n-1 veces donde n es la longitud del
arreglo, el bucle interno es para encontrar el menor
entre los elementos en la posición i, el bucle interno
se ejecuta n-i veces
    
```

Fig. 8: Octavo código.

```

Código 9.

void insertionSort(int[] array) { //O(n^2)
    int n = array.length; //O(1)
    for (int i = 1; i < n; i++) { //O(n)
        int key = array[i]; //O(1)
        int j = i - 1; //O(1)
        while (j > 0 & array[j] > key) { //O(n)
            array[j + 1] = array[j]; //O(1)
            j -= 1; //O(1)
        }
        array[j + 1] = key; //O(1)
    }
}

O(n) for (int i = 1; i < n; i++) { O(n) while (j > 0 & array[j] > key) { O(1) array[j + 1] = array[j]; O(1) j -= 1; O(1) } array[j + 1] = key; O(1) }

int n = array.length; calcula la longitud del arreglo y los bucles
externos se ejecutan desde la segunda hasta la ultima
posición, almacena el valor del elemento en "key"
y como índice del elemento anterior, el bucle inferior se
ejecuta n-1 veces si este dentro del arreglo x el elemento
se cambia que "key"
    
```

Fig. 9: Noveno código.

Los códigos 10 y 11 son algoritmos de ordenamiento (Merge Sort y Quick Sort) con complejidades algorítmicas de $O(n \log n)$ en el peor caso.

Los códigos 10 y 11 son algoritmos de ordenamiento (Merge Sort y Quick Sort) con complejidades algorítmicas de $O(n \log n)$ en el peor caso.

Código 10

```
Void mergesort(int C[ ] array, int left, int right){//O(n)
    if(left > right) f
        mergeSort(array, left, mid, right); //O(n)
        mergeSort(array, mid+1, right); //O(n)
        merge(array, left, mid, right); //O(n)
    = O(n) + O(n) + log n * O(n) = O(n log n)

    if(left < right): verifica si hay más de un elemento en el segmento actual. Si no es así, se detiene la recursión.
    int mid = left + right - 1/2 : calcula el índice medio para dividir el arreglo en dos mitades.
    mergeSort(array, left, mid); llamada recursiva para ordenar la mitad izquierda del arreglo.
    mergeSort(array, mid+1, right); llamada recursiva para ordenar la mitad derecha del arreglo.
```

Fig. 10: Decimo código.

Código 11

```
Void quickSort(int C[ ] array, int low, int high){//O(n)
    if (low < high) {
        int pivotIndex = partition(array, low, pivotIndex-1); //O(n)
        quickSort(array, low, pivotIndex-1); //O(n)
        quickSort(array, pivotIndex+1, high); //O(n)
    }
    = O(n) + O(n) + log n * O(n) = O(n log n)

    if (low < high): verifica si la sección actual del arreglo tiene más de un elemento. Si no se detiene la recursión.
    int pivotIndex = partition(array, low, high); llama a la función partition para dividir la sección actual en dos partes alrededor del pivote.
    quickSort(array, low, pivotIndex); llamada recursiva para ordenar la mitad izquierda de la partición.
    quickSort(array, pivotIndex+1, high); llamada recursiva para ordenar la mitad derecha de la partición.
```

Fig. 11: Once código.

H. Código 12

El código 12 calcula el número de Fibonacci de manera iterativa utilizando un bucle ‘for’. La complejidad algorítmica es $O(n)$.

Código 12

```
int Fibonacci(int n){//O(n)
    if (n<2) {
        return n; //O(1)
    }
    int dp[ ] = new int [n+1]; //O(n)
    dp[0] = 0; //O(1)
    dp[1] = 1; //O(1)
    for (int i=2; i<n; i++){ //O(n)
        dp[i] = dp[i-1] + dp[i-2]; //O(1)
    }
    return dp[n]; //O(1)
}
= O(n) + O(n) + log n * O(n) = O(n)

    if (n<2): verifica si el valor de "n" es 0 o 1. Si es así, retorna el valor.
    new int [dp]: almacena los datos calculados.
    dp[0]: inicializa el primer valor de fibonacci.
    dp[1]: inicializa el segundo valor de fibonacci.
    for (int i=2; i<n; i): el bucle itera "n" veces, cada vez calculando el valor de fibonacci para:
    dp[i] = dp[i-1] + dp[i-2]: calcula el valor de fibonacci para:
    sumando los dos valores anteriores en dp[i-1] y dp[i-2].
    return dp[n]: devuelve el valor de fibonacci calculado para "n".
```

Fig. 12: Doce código.

I. Código 13

Este código realiza una búsqueda lineal en una matriz de longitud n . En el peor caso, la complejidad algorítmica es $O(n)$.

Código 13

```
Void linearSearch (int C[ ] array, int target){//O(n)
    for (int i=0; i<array.length; i++){ //O(n)
        if (array[i] == target) { //O(1)
            return i; //O(1)
        }
    }
    = O(n) + O(n) = O(n)
```

El bucle se ejecuta n veces; comienza si el elemento actual del bucle es igual al elemento objetivo. Si se encuentra el elemento, se devuelve inmediatamente.

Fig. 13: Trece código.

J. Código 14

El código 14 implementa la búsqueda binaria en una matriz ordenada de longitud n . La complejidad algorítmica es $O(\log n)$ en el peor caso.

Código 14

```
int binarySearch (int C[ ] sortedArray, int target){//O(n)
    int left = 0, right = sortedArray.length-1; //O(1)
    while (left <= right){ //O(n)
        int mid = left + (right-left)/2; //O(1)
        if (sortedArray[mid] == target) { //O(1)
            return mid;
        } else if (sortedArray[mid] < target){ //O(1)
            left = mid+1; //O(1)
        } else{ //O(1)
            right = mid-1; //O(1)
        }
    }
    = O(n) + O(n) + log n * O(1) = O(n log n)

    Inicializa los índices left y right para definir el rango de la búsqueda. El bucle se ejecuta hasta que left sea menor o igual que right. En cada iteración, el rango se reduce en la mitad. Calcula el índice medio para dividir el rango de la búsqueda en dos mitades.
```

Fig. 14: Catorce código.

K. Código 15

El código 15 calcula el factorial de un número de manera recursiva. La complejidad algorítmica es $O(n)$.

Código 15

```
int factorial (int n){//O(n)
    if (n==0 || n==1) { //O(1)
        return 1; //O(1)
    }
    return n * factorial (n-1); //O(n)
}
= O(n) + O(n) + log n * O(n) = O(n)

    Si n es igual a 0 o 1, se detiene la recursión y retorna el factorial (n-1). Realiza una multiplicación entre "n" y el resultado de la llamada recursiva.
    La multiplicación es constante en tiempo, pero debido a la recursión se llama a la función factorial "n" veces.
```

Fig. 15: Quince código.

III. CONCLUSIONES

En este informe, hemos analizado varios códigos y calculado sus complejidades algorítmicas utilizando la notación Big-O. Hemos observado que diferentes algoritmos tienen diferentes niveles de eficiencia en términos de tiempo de ejecución en el peor caso.

Los códigos que utilizan bucles ‘for’ simples (Códigos 1, 4, 6, 13 y 15) tienen una complejidad lineal $O(n)$ en el peor caso. Por otro lado, los algoritmos de búsqueda binaria (Códigos 5

y 14) tienen una complejidad logarítmica $O(\log n)$ en el peor caso, lo que los hace eficientes para listas ordenadas.

Los algoritmos de ordenamiento (Códigos 7, 8, 9, 10 y 11) tienen complejidades cuadráticas $O(n^2)$ o n logarítmica $O(n \log n)$ en el peor caso. Los algoritmos de ordenamiento más eficientes son Merge Sort y Quick Sort.

Finalmente, los códigos que calculan el número de Fibonacci (Código 12) y el factorial (Código 15) tienen complejidades lineales $O(n)$ en el peor caso, lo que los hace adecuados para valores pequeños de n .

En resumen, la elección del algoritmo adecuado depende de la naturaleza del problema y el tamaño de entrada. Es importante considerar la complejidad algorítmica al diseñar y seleccionar algoritmos para garantizar un rendimiento óptimo en una variedad de situaciones.