Taller 1

Andrés Salcedo Vera
Fundacion Universitaria Konrad Lorenz

***Resumen*—A continuación, se expondrán los resultados obtenidos mediante el uso de los algoritmos designados para su análisis en notación BigO. Además, se abordarán conceptos de optimización y se proporcionará una breve explicación de las actividades realizadas durante el taller.**

I. Introducción

Para el primer taller de estructuras de datos, se asignaron 15 códigos en Java que debían ser transcritos manualmente en un cuaderno. En el proceso, se debían resaltar las palabras reservadas y los operadores lógicos en rojo, las variables en azul y el resto del código en negro.

Posteriormente, después de transcribir cada uno de los ejercicios, se procedió a realizar un análisis BigO para comprender la complejidad de cada una de estas funciones. A continuación, se proporcionará una breve explicación de la notación BigO y se describirá punto por punto cómo se resolvieron los ejercicios.

II. Big O

La notación Big O se utiliza en el ámbito de la informática con el propósito de evaluar la complejidad de un algoritmo en términos de tiempo y eficiencia. En la programación, es fundamental comprender y analizar las diferentes complejidades, ya que un programa puede variar significativamente en cuanto a su eficacia.

Podemos dividir estas complejidades en tres categorías principales. En primer lugar, las complejidades como O(n!), $O(2^n)$, y $O(n^2)$ se asignan a algoritmos con la peor calificación en cuanto a eficiencia. Estos algoritmos tienden a experimentar un crecimiento vertical significativo a medida que aumenta la carga de trabajo, lo que se traduce en un mayor consumo de tiempo y recursos del sistema.

En segundo lugar, encontramos la complejidad O(n log n), que es más lineal en su comportamiento. Esta complejidad puede variar ampliamente en términos de eficiencia, ya que puede ser moderadamente efectiva o menos eficiente, pero generalmente se encuentra en un punto intermedio en términos de rendimiento.

Por último, tenemos las complejidades de mejor rendimiento, que incluyen O(n), O(1), o O(log n). Estas se caracterizan por un crecimiento horizontal insignificante o casi nulo. En otras palabras, representan los casos ideales de eficiencia en un programa, donde el tiempo y los recursos se utilizan de manera extremadamente eficiente.

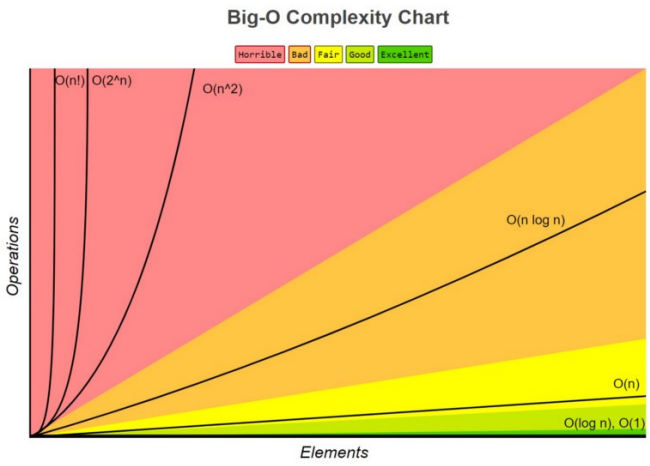

Figura 1. Algoritmo

II-A. *Punto 1*

En el primer punto, no hay nada notable; simplemente se tiene un bucle "for", iterando sobre "n", lo cual da una complejidad de O(n). Esto no proporciona mucha información debido a la simplicidad del código.

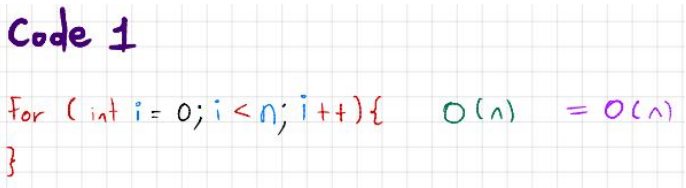

Figura 2. Ejecutar

II-B. *Punto 2*

En el segundo punto, podemos ver dos bucles "for", uno iterando sobre "n", y otro sobre "m". Esto nos daría una complejidad de $O(n \cdot m)$ o $O(n^2)$.

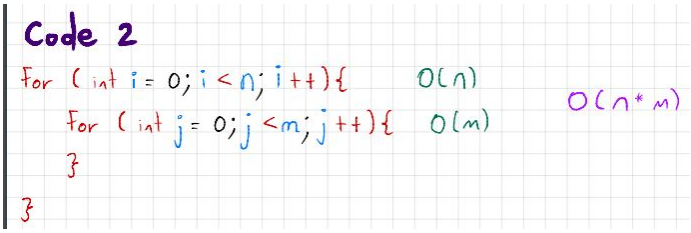

Figura 3. Punto 2

II-C. Punto 3

En el segundo punto, podemos ver dos bucles "for", los dos iterando sobre "n". Esto nos daría una complejidad de $O(n^2)$.

```
Code 3
for ( int i = 0; i < n; i++){        O(n)
    for ( int j = i ; j < n; j++){   O(n)        O(n²)
    }
}
```

Figura 4. Punto 3

II-D. Punto 4

En este caso, empezamos a observar asignación de variables, las cuales no representan ninguna dificultad para el sistema, por lo que su complejidad es O(1). Sin embargo, el bucle "for"genera cierta complejidad, como ya hemos discutido. En esta instancia, la complejidad en el peor de los casos sería O(n) debido al bucle "for".

Es importante aclarar que en la notación Big O no se consideran las asignaciones de variables constantes, por lo que si tenemos un O(1), este se cancelaría y quedarían solamente las variables más complejas que dependen del caso específico en el script.

```
Code 4
int  index = -1;          O(1)
for ( int i = 0; i < n; i++){  O(n)
    if (array[i] == target){   O(1)        O(n)
        index = i;             O(1)
        break;
    }
}
```

Figura 5. Punto 3

II-E. Punto 5

Ahora podemos observar ciclos "while". Este tipo de bucles también pueden variar según las circunstancias. En este caso, no se ha asignado ningún valor específico dentro de ellos, lo que implica que la complejidad es constante. Sin embargo, esta constancia está sujeta al control ejercido sobre el bucle. Una iteración deficiente o una planificación inadecuada pueden impactar directamente en el rendimiento del programa. En esta situación, nuestra complejidad sería exclusivamente de O(log n) ya que dependerá de los valores asignados en el array, left, right y el índice.

```
Code 5
int left = 0, right = n-1, index = -1;   O(1) → todas
while ( left <= right) {   O(1)
    int  mid = left + ( right - left) / 2;  O(log n)
    if ( array [ mid ] == target) { O(1)                = O(log n)
        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 6. Punto 5

II-F. Punto 6

En este caso tenemos la misma funcionalidad que en el anterior, por lo que en pocas palabras tambien vamos a tener una complejidad de O(log n).

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

Figura 7. Punto 6

II-G. Punto 7

Como ya hemos visto en puntos anteriores, el hecho de tener dos bucles for en una funcion puede generar mas problemas, aunque lo idoneo seria no tener este tipo de bucles, claramente es dependiente a lo que se haga en el codigo, asi que en este caso seria O(n^2).

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)
            }
        }
    }
}
```

O(n²)

Figura 8. Punto 7

II-H. Punto 8

Como hemos discutido en puntos anteriores, tener dos bucles "for."[en] una función puede ocasionar problemas de rendimiento. Aunque lo ideal sería evitar este tipo de bucles, su impacto depende en gran medida de la lógica implementada en el código. En este caso específico, la complejidad sería $O(n^2)$.

Code 8

```
void selectionSort(int[] array){ O(1)
  int n = array.length; O(1)
  for (int i = 0; i < n-1; i++){ O(n)
      int minIndex = i; O(1)
      for (int j = 0; j < n-i-1; 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(n²)

Figura 9. Punto 8

II-I. Punto 9

Como hemos mencionado previamente, la presencia de dos bucles "for."[en] una función puede tener un impacto negativo en el rendimiento. Aunque lo deseable sería evitar este tipo de bucles, su efecto puede variar significativamente según la lógica específica del código. En este caso, la complejidad sigue siendo $O(n^2)$.

Sin embargo, se ha introducido un bucle "while."[a]dicional que inicialmente podría llevar a una complejidad de O(log n). No obstante, debido a la inclusión de validaciones condicionales dentro del bucle, esto incrementa la complejidad general del programa. Por lo tanto, la complejidad en este caso se vería afectada y sería más alta que O(log n), pero la complejidad exacta dependería de la naturaleza y cantidad de validaciones dentro del bucle "while".

Code 9

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

O(n²)

Figura 10. Punto 9

II-J. Punto 10

En este punto, es evidente que la mayoría de los procesos son lineales, en gran parte debido a la recursividad. Sin embargo, la recursividad puede ser de doble filo, ya que en el peor de los casos podría llevarnos a una complejidad de $O(2^n)$. Esto se debe a que la recursión se ejecuta dos veces y se le agregan operaciones adicionales. La falta de control sobre la recursión podría ocasionarnos problemas y resultar en una eficiencia muy deficiente. Por lo tanto, es importante tener en cuenta y gestionar cuidadosamente la recursión en este tipo de situaciones para evitar el peor escenario de complejidad.

Code 10

```
void mergeSort(int[] array, int left, int right){ O(1)
    if(left < right){ O(1)
        int mid = left + (right - left) / 2; O(1)
        mergeSort(array, left, mid); O(n log n)
        mergeSort(array, mid+1, right); O(n log n)
        merge(array, left, mid, right); O(n log n)
    }
}
```

O(n log n)

Figura 11. Punto 10

II-K. Punto 11

En este punto, es evidente que la mayoría de los procesos son lineales, en gran parte debido a la recursividad. Sin embargo, la recursividad puede ser de doble filo, ya que en el peor de los casos podría llevarnos a una complejidad de $O(2^n)$. Esto se debe a que la recursión se ejecuta dos veces y

se le agregan operaciones adicionales. La falta de control sobre la recursión podría ocasionarnos problemas y resultar en una eficiencia muy deficiente. Por lo tanto, es importante tener en cuenta y gestionar cuidadosamente la recursión en este tipo de situaciones para evitar el peor escenario de complejidad.

```
Code 11
void quickSort(int[]array, int low, int high){ O(nlogn)
  if(low < high){ O(1)
    int pivotIndex = partition(array, low, high); O(1)
    quickSort(array, low, pivotIndex-1); O(nlogn)
    quickSort(array,pivotIndex+1, high); O(nlogn)
  }
                                        O(nlogn)
}
```

Figura 12. Punto 11

II-L. Punto 12

En este punto lo unico que genera mas complejidad seria el ciclo for y esto nos daria complejidad O(n).

```
Code 12
int fibonacci(int n){ O(1)
  if(n <= 1){ O(1)
    return n; O(1)
  }
  int[] dp = new int[n+1]; O(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)
}
```

Figura 13. Punto 12

II-M. Punto 13

En este punto, lo único que añade complejidad es el bucle "for", lo cual nos lleva a una complejidad de O(n). Es importante recordar que los comentarios no afectan en absoluto la ejecución del programa, por lo que no tienen ningún impacto en la complejidad del mismo.

```
Code 13
void linearSearch(int[] array, int target){ O(1)
  for(int i = 0; i < array.length; i++){ O(n)
    if(array[i] == target){ O(1)
                                        O(n)
      // Encontrado
      return; O(1)
    }
  }
  // No Encontrado
}
```

Figura 14. Punto 13

II-N. Punto 14

En este punto, la complejidad se debe principalmente a las operaciones dentro del bloque ïf", pero la mayor complejidad proviene de la recursividad utilizada en la sentencia return". Esto puede causar problemas y resultar en una complejidad de O(n) o incluso complicarse aún más hasta llegar a $O(2^n)$, ya que el resultado se vuelve más extenso con cada iteración recursiva. La eficiencia del algoritmo dependerá de la gestión adecuada de la recursión y de cómo se maneje el crecimiento exponencial en la complejidad en función de los datos de entrada.

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

Figura 15. Punto 14

II-Ñ. Punto 15

En este punto solo se genera cierta complejidad por las opetaciones que se encuentran en el codigo, por lo que la complejidad seria de O(log n)

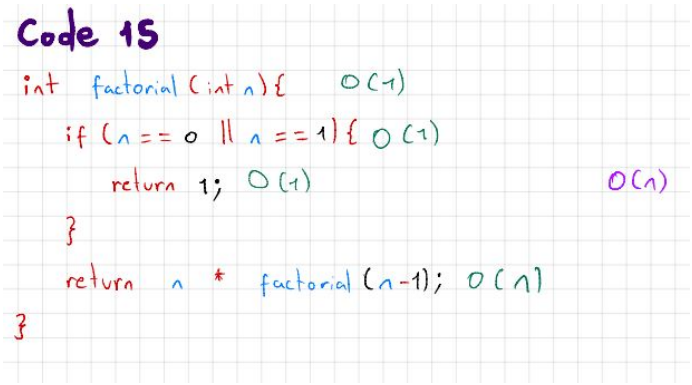

Figura 16. Punto 15

III. Bibliografia

OpenAI. (2021). ChatGPT: A large language model.
Andés Salcedo Vera. (2023). Taller 3-4.