Complejidad Algoritmica - Estructura de Datos

Daniel Osorio Valencia - 614222007

I. Introduccion

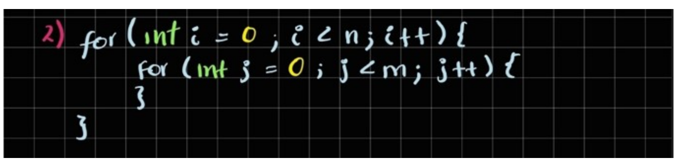
A continuación se realizará el análisis e interpretación para cada linea de código con el fin de deterinar el grado de complejidad. Lo anteriormente mencionado nos permitirá comprender como esta complejidad afecta el código y ”Big O”

A. CÓDIGO 1

- El for que encontramos tiene un límite ”n”, entonces tenemos O(n).

Respuesta: O(n)

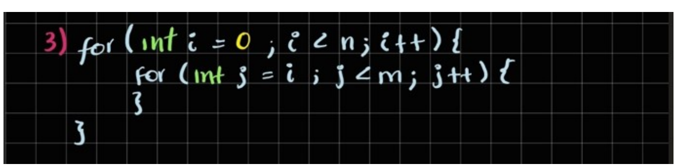
B. CÓDIGO 2



- En primer for que encontramos tiene un límite ”n”, entonces tenemos O(n).
- En segundo for el limite es ”m”, entonces tenemos O(m).

Respuesta: O(m) + O(n) = **O(n*m)**

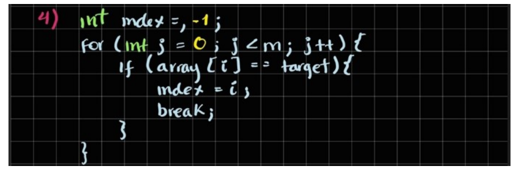
C. CÓDIGO 3



- En primer y segundo ”for” tiene un límite de ”n”, entonces tenemos: O(n) en cada ”for”.

Respuesta: O(n) + O(n) = O(n^2)

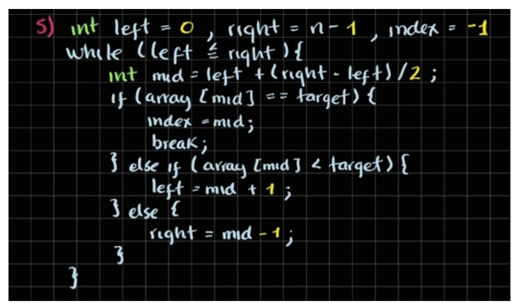
D. CÓDIGO 4



-La línea int index = -1 es una asignación, por lo cual, es de tiempo constante: O(1).
- En el for tiene un límite de ”n”,entonces tenemos: O(n).
- El if tiene una complejidad constante: O(1).
- La línea donde le da valor al index, es constante: O(1).
- Y el break, también es constate de O(1).

Respuesta: O(1) + O(n) + O(1) + O(1) + O(1) = **O(n)**

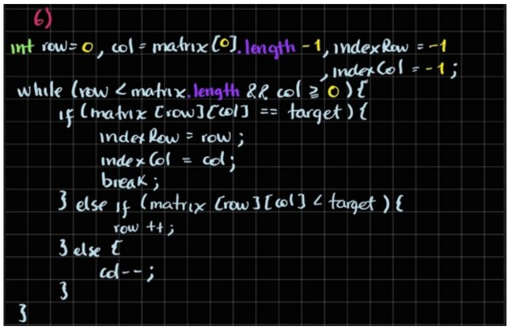
E. CÓDIGO 5



- En la primera linea es una asignación, por tanto tenemos: O(1). - El while es para hacer una búsqueda binaria, entonces tenemos: O(Log n).
- La tercera linea es una asignación y es constante: O(1).
- El if tiene una complejidad constante de O(1).
- La línea donde le da valor al index y el break, tienen una complejidad constante: O(1).
- El if de nuevo tiene una complejidad constante: O(1).
- La else también es constante O(1).

Respuesta: O(1) + O(Log n) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) = **O(Log n)**.

F. CÓDIGO 6

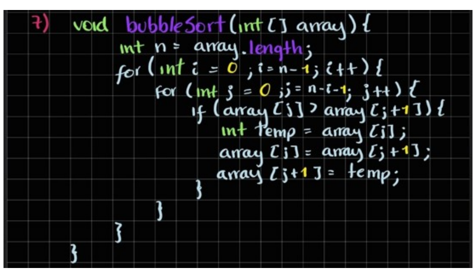


- En la linea primera línea tenemos inicializaciones, así que tenemos: O(1).
- En el while que encontramos que tiene una complejidad de O(n+m) donde n es el numero de filas y m el numero de columnas.
- En el if tenemos una complejidad de O(1).
- En las líneas donde se da valor a la variable indexRow y indexCol, tienen una complejidad constante de O(1).
- El else if y el aumento de 1 en la variable row, tienen la misma complejidad constante de O(1).
- Y el else también tiene complejidad constante de O(1).

Respuesta: O(1) + O(n+m) + O(1) + O(1) + O(1) + O(1) + O(1) + O(1) = **O(n+m)**

G. CÓDIGO 7

-

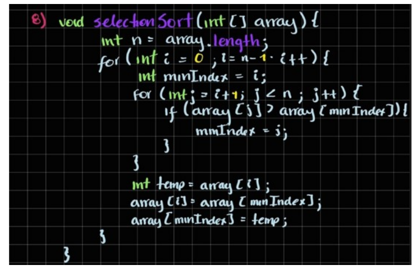


- En la primera línea donde se inicializa la variable n, es constante: O(1).
- El for se ejecuta hasta n-1, entonces tenemos: O(n).
- El otro for será de n-i-1 veces, así que también será: O(n).
- El if tiene una complejidad constante: O(1).
- las úlitmas 3 líneas tiene una complejidad constante: O(1)

Respuesta: O(1) + O(n) + O(n) + O(1) + O(1) + O(1) + O(1) = O(n*n) = On^2

H. CÓDIGO 8

- En la primera línea con la variable n inicializada, tenemos: O(1).
- El for va hasta n-1, así que será de: O(n).

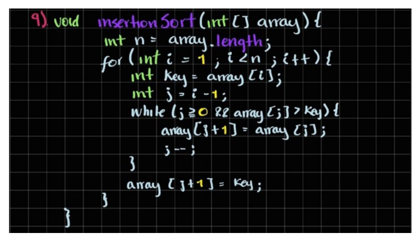


- En la línea donde se inicializa la variable minIndex, respecto al tamaño del i, tiene una complejidad constante: O(1).
- El segundo for que esta anidado se va a ejecutar hasta n, entonces también es: O(n).
- El if que evalúa que la posición del array tiene una complejidad constante: O(1).
- las úlitmas 3 líneas tiene una complejidad constante: O(1)

Respuesta: O(1) + O(n) + O(1) + O(n) + O(1) + O(1) + O(1) = O(n^2).

I. CÓDIGO 9

-

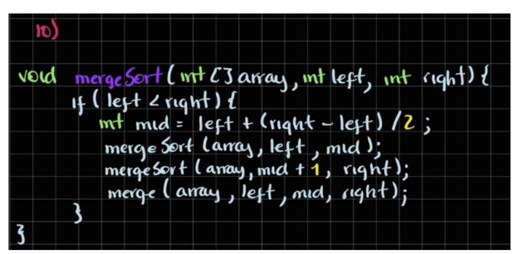


- En la primera línea con la variable n inicializada, tenemos: O(1).
- El primer for se va a ejecutar hasta n, entonces es: O(n).
- Las inicializaciones de Key y j tiene una complejidad constante: O(1).
- El while será hasta n veces, entonces será: O(n).

Respuesta: O(1) + O(n) + O(1) + O(n) = O(n^2)

J. CÓDIGO 10

-



- En la primera línea se define mergeSort recibiendo 3 parámetros, un array y 2 números, siendo constante: O(1).

- El if compara si derecha es mayor que izquierda así que es constante: O(1).

- La declaración del int mid, tiene una complejidad constante de O(1).

- Al llamar recursivamente a la función mergeSort hace que su complejidad sea dependiente de la profundidad de la reclusión y también está relacionada con el resultado de la operación mid es por ello que esta línea tiene una complejidad de O(Logn).

- Al volver a llamar a la función tendremos que su complejidad es la misma: O(Logn).

- Y la función merge, que se encarga de fusionar los 2 al arrays, por lo tanto serpa: O(n).

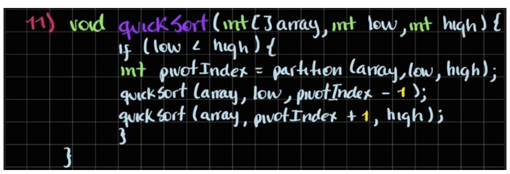
-

Respuesta: O(1) + O(1) + O(1) + O(Logn) + O(Logn) + O(n) = **O(Log n)**

REFERENCES

K. *CÓDIGO 11*

-



- En la primera línea tenemos la definición de la función quickSort y tiene una complejidad constante: O(1).

- El if evalúa que high sea mayor que low, así que es: O(1).

- Inicializamos una variable llamada pivotindex, la cual es elegida dentro de los elementos del array para poder dividir el array, la complejidad de esta línea va a depender de la elección del pivote, por lo tanto tiene una complejidad de O(n).

- En la penúltima línea se llama recursivamente a la función para que organice el sub-array izquierdo generado del principal, su complejidad es de O(n log n).

- En la última línea se llama nuevamente de manera recursiva a la función para que organice el sub-array derecho generador del principal, su complejidad es de O(n log n).

Respuesta: O(1) + O(1) + O(n) + O(n log n) + O(n log n) =
O(n log n)

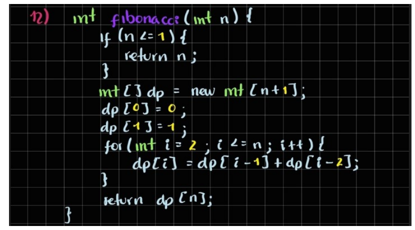
L. *CÓDIGO 12*

-

- En la primera línea a fibonacci tomando un número entero n, siendo constante: O(1).

- En el if sólo se calcula que si n cero o uno, por ende, su complejidad es constante de O(1).

- En la siguiente sección se usa programación dinámica, y como en la raíz se declara n, esta parte tendrá una complejidad: O(n).



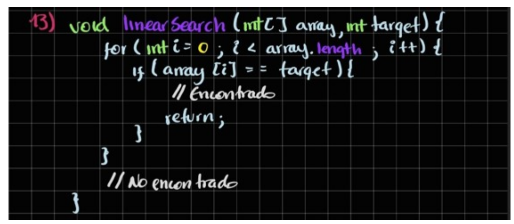
- El for se ejecutará hasta n, por lo cual, será complejidad: O(n).

- Y en la úlitma se devuelve el valor de la posición n del array, siendo: O(1).

Respuesta: O(1) + O(1) + O(n) + O(n) + O(1) = **O(n)**

M. *CÓDIGO 13*

-



- La sunción linearSearch, que toma un array y un valor entero a buscar, siendo complejidad: O(1).

- En la siguiente línea al tener que recorrer todo el array, será de complejidad O(n).

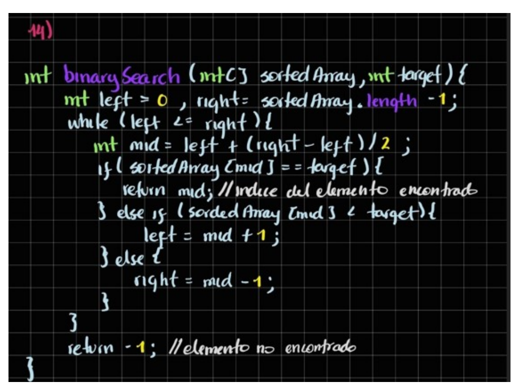
- Dentro del for encontramos un if que sólo evalua si el array en la posición i es igual al dato ingresado, entonces tenemos que: O(1).

- Y tenemos un return constante: O(1).

Respuesta: O(1) + O(n) + O(1) + O(1) = **O(n)**

N. *CÓDIGO 14*

-



- La función binarysearch, recibe por parámetros por lo cual es constante: O(1)

- En la siguiente línea se inicializan variables, siendo también: O(1).

- El while se va a ejecutar dependiendo de la longitud del array, así que será O(log n)

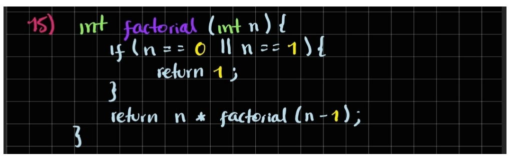
- Se calcula el índice medio dentro de la sección actual del array, tiene una complejidad constante de O(1).

- Dentro del bucle y los if, else if y else se realizan comparaciones siemples, por ende, sus complejidades son: O(1).

Respuesta: O(1) + O(1) + O(log n) + O(1) + O(1) + O(1) = **O(log n)**

O. CÓDIGO 15

-



- En la línea uno inicialización de una función; siendo su complejidad constante: O(1).

- En el if también tendrá una complejidad constante: O(1).

- Finalmente se devuelve la variable n multiplicado por la función factorial,obteniendo una complejidad: O(n).

Respuesta: O(1) + O(1) + O(n) = **O(n)**

II. CONCLUSION

Con la realización de este trabajo podemos concluir la importancia de analizar, comprender y manejar una buena estructura en el desarrollo de algoritmos que nos permitan generar códigos mas limpios y eficientes para las tareas, trabajos o proyectos que se esten creando, permitiendo así, tener visibilidad de todo el panorama y tomar deciciones en cuanto a tiempo y dinero.

III. BIBLIOGRAFÍA

Joel Ayala de la Vega, Irene Aguilar Juárez, Farid García Lamont Hipólito Gómez Ayala (2019). Introducción al análisis de algoritmos.http://ri.uaemex.mx/bitstream/handle/20.500.11799/105198/Libro

José A. Mañas. (2017). Análisis de Algoritmos – Complejidad.https://www.dit.upm.es/ pepe/doc/adsw/tema1/ Complejidad.pdf