Seguimiento: Computación y estructuras discretas.

Estudiantes:

- Maria Paz Henao Bedoya
- María Juliana Marín Shek
- Valentina Arana Babativa

Instrucciones del Profesor:

El trabajo debe realizarse en los grupos de la tarea integradora y debe ser entregado el primer día de clase de la próxima semana. Si tienen alguna duda o inquietud, pueden hacérmela saber por correo electrónico. La entrega debe ser un proyecto de IntelliJ que se ejecute por consola. Dentro del proyecto debe haber una carpeta con documentación, en la que se incluya el análisis temporal y espacial de los algoritmos desarrollados. No es necesario incluir pruebas de ningún método.

4. Analiza la complejidad temporal y espacial, en el peor caso, de cada uno de los algoritmos.

   a) Realiza una tabla y describe línea por línea la cantidad de veces que se ejecuta cada una de ellas.

   b) Al final, realiza la suma de las líneas para hallar la complejidad del algoritmo y concluye cuál es la cota superior. Una manera formal de encontrar la cota es mediante una constante *c* que permita acotar por arriba la función encontrada.

1. Escribe un algoritmo que tenga como entrada una lista de *n* enteros en orden no decreciente y genere la lista de todos los valores que aparecen más de una vez.

a) Entrada: *L* = [1, 2, 2, 3, 4, 4, 4, 5, 6, 6]

b) Salida: *R* = [2, 4, 6]

Encontrar elementos duplicados de una lista ordena			
Número	Código	Costo	# de Iteraciones
1	public static ArrayList<Integer> encontrarRepetidos(int[] lista1){	O(1)	1
2	ArrayList<Integer> lista2= new ArrayList<>();	O(1)	1
3	for(int i=0; i < lista1.length; i++){	O(n)	n -1
4	if(i>0 && lista1[i] != lista1[i-1]){	O(n)	n
5	if(lista1[i] == lista1[i+1]){	O(n)	n-1 (máximo)
6	lista2.add(lista1[i]);	O(n)	elementos ≤ n
7	}		
8	}		
9	}		
10	return lista2;	O(1)	1
11	}		

Demostración formal

$T(n) = c_1 n + c_2$

$T(n) = O(n)$

$T(n) \leq c * g(n)$

$c_1 n + c_2 \leq c * n$

$c_1 + \frac{c_2}{n} \leq c$

$n_0 = 1$

$c = c_1 + c_2$

$T(n) = O(n)$

Cota superior:

$O(n)$

Complejidad temporal (Suma de las líneas)

$O(1) + O(1) + O(1) + O(n) + O(n) + O(n) + O(n) + O(1) = 4n + 3 = O(n)$

Demostración formal:

$T(n) = 4n + 3$

$T(n) = O(n)$

$4n + 3 \leq c * n$

$\frac{4n}{n} + \frac{3}{n} \leq \frac{c*n}{n}$

$4 + \frac{3}{n} \leq c$

$n_0 = 1$

$c = 4 + 3 = 7$

$7 \leq 7$

$T(n) = O(n)$

Conclusión:

Dado que encontramos un c y un n_0 que cumplen la definición de notación Big-O, podemos afirmar que la función está acotada por O(n).

Cota superior:

$O(n)$

Tipo	Variable	Tamaño de 1 valor atómico	Cantidad de valores atómicos
Entrada	`lista1`	*32 bits*	*n*
	n	*32 bits*	*1*
Auxiliar	`lista2`	*32 bits*	*1*
Salida	`lista2`	*32 bits*	*1*

Complejidad espacial total:

Entrada + Auxiliar + Salida = n + 1 + 1 + 1 = n + 3 = O(n)

Complejidad espacial auxiliar:

1 = Θ(1)

Complejidad Espacial Auxiliar + Salida:

1 + 1= Θ(1)

Conclusión

*El espacio dominante es **n**, correspondiente al arreglo de entrada, por lo que la complejidad espacial total es **Θ(n)**.*

2. Un palíndromo es una cadena que se lee igual de izquierda a derecha que al revés. Escribe un algoritmo para determinar si una cadena de *n* caracteres es un pal´ındro mo.

a) Entrada: *cadena* = ”*reconocer*”

b) Salida: *verdadero*

c) Entrada: *cadena* = ”*hola*”

d) Salida: *f also*

Verificar si una palabra es un palíndromo

Número	Código	Costo	# de Iteraciones
1	private boolean esPalindromo(String palabra) {	O(1)	1
2	int left = 0;	O(1)	1
3	int right = palabra.length() - 1;	O(1)	1
4	while (left < right) {	O(n)	n/2
5	if (palabra.charAt(left) != palabra.charAt(right)) {	O(n)	n/2 (máximo)
6	return false;	O(1)	Solo si hay diferencia
7	}		
8	left++;	O(n)	n/2
9	right--;	O(n)	n/2
10	}		
11	return true;	O(1)	1
12	}		

Complejidad espacial:

O(1) + O(1) + O(1) + O(n/2) + O(n/2) + O(n/2) + O(n/2)+ O(1) = $\frac{4n}{2} + 4 = O(n)$

O(n) = n

$f(n) = \frac{4n}{2} + 4$

Demostración formal:

$f(n) = a * n + b$

$a * n + b \leq c * n, \forall n \geq n_0$

$\frac{an}{2} + b \leq c * n$

$\frac{a}{2} + \frac{b}{n} \leq c$

$c = \frac{a}{2} + b$

Demostración formal:

$\frac{4*n}{2} + 4 \leq c * n$

$\frac{2n}{n} + \frac{4}{n} \leq c$

$c = 2 + \frac{4}{n}$

$c = 2 + 4 = 6$

Para n_0 *=1*

Cota superior:

O(n)

Complejidad temporal (Suma de las líneas):

$O(n/2) = O(n)$

Complejidad espacial total:

Entrada + Auxiliar + Salida = n + 3 = Θ(n)

Complejidad espacial auxiliar:

1 + 1 = Θ(1)

Solo variables auxiliares (left, right) → O(1)

Complejidad espacial auxiliar + Salida:

1 + 1 + 1 = Θ(1)

Conclusión

El espacio dominante es ***n****, correspondiente a la cadena de entrada, por lo que la complejidad espacial total es* ***Θ(n)****.*

Tipo	Variable	Tamaño de 1 valor atómico	Cantidad de valores atómicos

Entrada	`palabra`	*16 bits (2 bytes)*	`n (tamaño de la palabra)`
Auxiliar	`left`	*32 bits (4 bytes)*	*1*
Auxiliar	`right`	*32 bits (4 bytes)*	*1*
Salida	`boolean`	*1 bit*	*1*

3. Escribe un algoritmo que cuente los bits 1 que aparecen en una cadena de bits, examinando cada bit para determinar si es un 1.

a) Entrada: cadenaBits = ”1011001”

b) Salida: 4

Análisis temporal Y análisis espacial

Contar bits “1” en una cadena			
Número	Código	Costo	# de Iteraciones
1	public void punto3(){	O(1)	1
2	String cadena = "1011001";	O(1)	1
3	int cont = 0;	O(1)	1
4	for(int i=0; i < cadena.length(); i++){	O(n)	n
5	if(cadena.charAt(i) == '1'){	O(n)	n (máximo)
6	cont++;	O(n)	k veces (donde k≤n), como es el peor de los casos entonces k = n
7	}		
8	}		
9	System.out.println(cont);	O(1)	1
10	}		
11	}		

Complejidad temporal (Suma de la complejidad de las líneas)

O(1) + O(1) + O(1)+ O(n) + O(n) + O(n) + O(1) = 3n + 4 = O(n)

Complejidad Espacial

String cadena → O(n) (almacena los caracteres)

int cont → O(1)

$O(n) + O(1) = O(n)$

Complejidad Final= O(n)

Cota superior: O(n)

a = 3 operaciones por iteración. n veces

b = 4 operaciones fuera del bucle

$f(n) = a * n + b$

$f(n) = O(n)$

$a * n + b \leq c * n, \forall n \geq n_0$

$a + \frac{b}{n} \leq c$

$c = a + b$

Peor caso Cota superior

Demostración Formal

$f(n) = 3n + 4$

$f(n) = O(n)$

$3 * n + 4 \leq c * n, \forall n \geq n_0$

$\frac{3n}{n} + \frac{4}{n} \leq \frac{cn}{n}$

$3 + \frac{4}{n} \leq c$

3 + 4 = 7

valor n

$n_0 = 1$

*3*1 +* $\frac{4}{1} \leq c$

3 + $4 \leq c$

$7 \leq c$ *cuando c = 7 (cumple que* $\forall n \geq n_0$*)*

cota superior : c = 7 n_0

Tipo	Variable	Tamaño de 1 valor atómico	Cantidad de valores atómicos
Entrada	`cadena`	*16 bits*	*n*
Auxiliar	`cont`	*32 bits*	*1*
Auxiliar	`i`	*32 bits*	*1*
Salida	`cont`	*32 bits*	*1*

Complejidad espacial total:

Entrada + Auxiliar + Salida = n + 3 = Θ(n)

Complejidad espacial auxiliar:

1 + 1 = Θ(1)

Complejidad espacial auxiliar + Salida:

1 + 1 + 1 = Θ(1)

Conclusión

El espacio dominante es ***n****, correspondiente a la cadena de entrada, por lo que la complejidad espacial total es* ***Θ(n)****.*

Conclusión:

Algoritmo	*Complejidad Temporal*	*Complejidad Espacial*	*Cota Superior*

Encontrar duplicados	*O(n)*	*O(n)*	*O(n)*
Verificar palíndromo	*O(n)*	*O(n)*	*O(n)*
Contar bits '1'	*O(n)*	*O(n)*	*O(n)*

Todos los algoritmos tienen una complejidad temporal de O(n) en el peor caso.

Repositorio:

https://github.com/Majumashe/Seg3AnalisisComplejidad/tree/main

```
package ui;
import java.util.ArrayList;
public class Main {
   public static void main(String[] args) {
        System.out.println("Segumiento 3 Complejidad");
        Main main = new Main();
          // Prueba 1: Encontrar valores repetidos en una lista
ordenada
        int[] lista1 = {1, 2, 2, 3, 4, 4, 4, 5, 6, 6};
                              ArrayList<Integer>    repetidos    =
encontrarRepetidos(lista1);
        System.out.println("Lista de repetidos: " + repetidos);

        // Prueba 2: Verificar si una palabra es un palíndromo
        String palindromoTrue = "reconocer";
        String palindromoFalse = "hola";
           System.out.println("¿'reconocer' es palíndromo?: " +
main.esPalindromo(palindromoTrue));
               System.out.println("¿'hola' es palíndromo?: " +
main.esPalindromo(palindromoFalse));

        // Prueba 3: Contar bits '1' en una cadena binaria
```

```
        String cadenaBits = "1011001";
        int cantidadBits1 = contarBits1(cadenaBits);
                System.out.println("Número de bits '1': " +
cantidadBits1);
    }

     public static ArrayList<Integer> encontrarRepetidos(int[]
lista1){  //O(1)
        ArrayList<Integer> lista2= new ArrayList<>(); //O(1)
        for(int i=0; i < lista1.length; i++){ //O(n)
            if(i>0 && lista1[i] != lista1[i-1]){ //O(n)
                if(lista1[i] == lista1[i+1]){  //O(n)
                    lista2.add(lista1[i]); //O(n)
                }
            }
        }
        return lista2; //O(1)
    }


    public boolean esPalindromo(String palabra) {
        //apunta al primer carácter de la palabra
        int left = 0; //O(1)
        //apunta al último carácter de la palabra
        int right = palabra.length() - 1; //O(1)

          //Se ejecuta hasta la mitad de la palabra. Se repite
hasta llegar a left == right
        while (left < right) { //O(n/2)
                  //Si en cualquier puntosabemos que NO es un
palíndromo y devolvemos false de inmediato
              if (palabra.charAt(left) != palabra.charAt(right))
{ //O(n/2)
                return false; //O(1) mejor caso
            }
```

```
            //Avanzamos left hacia la derecha y retrocedemos
right hacia la izquierda
            left++; //O(n/2)
            right--; //O(n/2)
            //Esto hace que el bucle se repita n/2 veces.
        }
         //Si terminamos el bucle sin encontrar diferencias, la
palabra es un palindromo y devolvemos true
        return true; //O(1)
    }

    public static int contarBits1(String cadena){
        //contara cuantos '1' hay en la cadena
        int cont = 0; //O(1)
        for(int i=0; i < cadena.length(); i++){ //O(n)
             //En cada iteracion se compara si el caracter en la
posición i es '1'
            if(cadena.charAt(i) == '1'){ //O(n)
                    //Si el caracter es '1', se incrementa el
contador
                cont++; //O(n)
            }
        }
        return cont; //O(1)
    }

}
```