

DISEÑO DE SOFTWARE

**ACTIVIDAD 1 – GUIA 3.**

**ETRUCTURAS DE DATOS NO LINEALES**

JOAN SEBASTIAN GUERRA ARANGO

DILSA TRIANA MARTINEZ

2024

Contenido

Objetivos ………………………………………………………………………………….

Marco Teórico ..…………………………………………………………………………..

Ejercicios Cálculo de Complejidad .……………………………………………………

Ejercicios Árboles Binarios………...……………………………………………………

Link Repositorio ………………………………………………………………………….

Conclusiones ……………………………………………………………………………..

Bibliografía ………………………………………………………………………………..

# Objetivos:

* Entender el concepto de complejidad Algorítmica, calcularla en diversos algoritmos y comprender su importancia en el desarrollo de software.
* Comprender los árboles binários como estructuras de datos, su estructura, tipos recorridos y operaciones comunes.
* Utilizar métodos para reconstruir un Árbol binario a partir de sus recorridos.
* Aplicar las listas en un proyecto práctico (Central de pacientes)
* Almacenar el código fuente en un repositorio (GitHub, GitLab o BitBucket).

# Marco Teórico:

**La complejidad algorítmica** se refiere al estudio de la eficiencia de los algoritmos, evaluando los recursos computacionales que requieren para ejecutar una tarea. Esto incluye el tiempo y el espacio necesarios, y proporciona una base para comparar y seleccionar algoritmos en función de su rendimiento.

# Ejercicios Cálculo de complejidad:

1. Para cada uno de los tiempos que toma un algoritmo en terminar, demostrar que cada uno es de la complejidad dada
2. **TA​(n) = 2n3−3n2+1 es O(n3)**

Para demostrar que la función **TA(n)** sea **O(n3)**, necesitamos encontrar **c** y **n0** tales que:

2n3 - 3n2 + 1 <= c \* n3 para todo n >= n0

Reorganizamos la desigualdad:

2n3 - 3n2 + 1 <= c \* n3

Restamos 2n^3 de ambos lados:

-3n2 + 1 <= (c - 2) \* n3

Para obtener una cota superior válida, necesitamos que:

(-3n2 + 1)/n3 <= c - 2

Esto simplifica a:

(-3n2)/(n3) + (1)/(n3) <= c - 2 –(3)/(n) + (1)(n3) <= c - 2

A medida que **n** aumenta, (-3)/(n) y (1)/(n3) se acercan a 0. Entonces, para valores grandes de n, -(3)/(n) es muy pequeño, y podemos encontrar un **c** tal que la desigualdad sea verdadera.

Vamos a elegir valores para c y n0 :

1. Elegir un valor para c :

Vamos a asumir un valor razonable para **c** . Supongamos c = 3. Entonces queremos verificar si:

-(3)/(n) + (1)/(n3) <= 3 - 2 -(3)/(n) + (1)/(n3) <= 1

Para valores grandes de n, el término (1)/(n3) es muy pequeño en comparación con -(3)/(n), por lo que:

-(3)/(n) <= 1

Para que esto se cumpla, necesitamos:

-3 <= n & n >= 3

Por lo tanto, para n >= 3 , la desigualdad se mantiene.

2. Verificación con n0 = 3:

Verificamos si para n >= 3 , la desigualdad se cumple:

- Para n = 3:

TA(3) = 2 \* 3^3 - 3 \* 3^2 + 1 = 54 - 27 + 1 = 28

Y:

c \* 3^3 = 3 \* 27 = 81

Entonces:

28 <= 81

Esto es cierto.

Por lo tanto, los valores **c** = 3 y **n0** = 3 son adecuados para la demostración. La función TA(n) = 2n3 - 3n2 + 1 está efectivamente acotada superiormente por c \* n3 con estos valores, lo que confirma que TA(n) es O(n3).

1. **TA​(n) = n5 + 42 – (√n) + 1 es O(n5)**

Queremos encontrar valores para la constantes c y n0 tales que para todo n >= n0 , la siguiente desigualdad se cumpla:

TA(n) <= c \* n5

donde TA(n) = n5 + 42 – (√n) + 1.

Reordenamos la desigualdad:

n5 + 42 - (√n) + 1 <= c \* n^5

Restamos n5 de ambos lados:

42 - (√n) + 1 <= (c - 1) \* n5 43 - (√n) <= (c - 1) \* n5

Para grandes valores de n, (√n) es mucho menor comparado con n^5. Por lo tanto, el término ( -(√n)) será insignificante frente a (c - 1) \* n5 si elegimos una constante c suficientemente grande.

Elegimos Valores para c y n0.

Elegir un valor para c

Para simplificar, supongamos que c = 2. Entonces queremos que:

43 - (√n) <= (2 - 1) \* n5 \* 43 -(√n) <= n5

Encontrar n0

Necesitamos encontrar un valor **n0** tal que la desigualdad ( 43 - (√n) <= n5) se mantenga para todo n >= n0. Esto significa que para valores suficientemente grandes de n, el término (√n) es mucho menor que 43

y puede ser ignorado en comparación con n5.

Para encontrar el valor de n0, resolvamos la desigualdad:

(√n) <= 43

Al elevar ambos lados al cuadrado, obtenemos:

n <= 43^2 n <= 1849

Por lo tanto, podemos elegir n0 = 1849. Para todo ( n >= 1849 ), (√n) <= 43, y así:

43 - (√n) <= 43

Dado que (√n) es mucho menor que n5 cuando n es grande, 43 - (√n) será menor que n^5 para n >=1849.

Vamos a verificar para n = 1849:

Para n >=1849:

(√1849) = 43

Así:

43 - √1849 = 43 - 43 = 0

Y:

n5 para n = 1849 es enormemente grande en comparación con 0.

Por lo tanto, la elección de c = 2 y n0 = 1849 garantiza que:

43 - (√n) <= n5

para n >= 1849, confirmando que TA(n) = n^5 + 42 - (√n) + 1 es (O(n^5)).

1. **TA​(n) = n2 log n + 2n4 + √2n es O(n4)**

Queremos encontrar una constante c y un valor n0 tales que para todo n >= n0:

TA(n) <= c \* n4.

Dado que TA(n) = n2 log n + 2n4 + √2 n, necesitamos que:

n2 log n + 2n4 + √2 n <= c \* n4.

Desglose de la Cota Superior

1. Para el término n2 log n queremos que:

n2 log n <= (c - 2) \* n4

Dividimos ambos lados entre n4:

(n2 log n)/(n4) = (log n)/(n2) <= (c - 2)

Dado que **(log n)/(n2)** tiende a 0 cuando **n** crece, podemos encontrar un valor de **n1** suficientemente grande tal que:

(log n)/(n2) <= (c - 2)

Por ejemplo, para n >= 10, (log n)/(n2) se vuelve muy pequeño. Podemos calcular:

log 10/(10^2) aprox (1)/(100) = 0.01

Por lo tanto, para valores de n >= 1, (log n)/(n2) es mucho menor que 1, y podemos ajustar c para satisfacer esta desigualdad.

2. Para el término √2 \* n

Queremos que:

√2 \* n <= (c - 2) \* n4

Dividimos ambos lados entre n4:

√2 \* n /n4 = √2 \* n /n^3 <= (c - 2)

Dado que √2 /n3 tiende a 0 cuando n crece, podemos encontrar un valor de n2 suficientemente grande tal que:

√2 /n3 <= (c - 2)

Por ejemplo, para n >= 10:

√2 /10^3 aprox 1.414/1000 = 0.001414

Esto también es mucho menor que 1, por lo que podemos ajustar c para satisfacer esta desigualdad.

3. Para el término dominante 2n^4

Dado que este término es 2n^4, si elegimos c = 3, entonces:

2n^4 <= 3n^4

Esta desigualdad es siempre cierta.

Elegimos valores para c y n0

Hemos elegido **c** = 3, porque este valor es suficientemente grande para cubrir los términos menores cuando n es grande.

Encontramos n0: Para que n >=n0, hemos estimado valores en torno a 10 para simplificar. De hecho:

- Para n >= 10:

- Para log n/n^2:

log 10/10^2 aprox 0.01 <=1, lo cual es pequeño suficiente para ser acotado por

(c - 2) = 1.

- Para √2 /n3:

√2 /10 ^3 aprox 0.001414 < 1, lo cual es pequeño suficiente para ser acotado por (c - 2) = 1.

Por lo tanto, elegimos n0 = 10 para asegurar que las desigualdades se mantengan para todos los n >= n0.

Conclusión

Con **c** = 3 y **n0** = 10, tenemos que:

TA(n) = n^2 log n + 2n^4 + √2 \* n <= 3 \* n^4

para todo n >= 10. Por lo tanto, TA(n) es (O(n^4)).

1. Calcule paso a paso la complejidad del siguiente algoritmo:

void XXXXXX(int n) {

int x = 0;

for (int i = 1; i <= n; i \*= 5) {

int j = 1;

for (int j = 1; j <= n; j += 2) {

x = x + j;

}

for (int k = n; k >= 1; k /= 2) {

x = x + 1;

}

}

}

Para calcular la complejidad del algoritmo usaremos la notación sumatoria, primero identificaremos la cantidad de trabajo realizado por cada bucle en términos de sumatorias. Luego, combinaremos estos resultados para obtener la complejidad total.

1. Análisis del Primer Bucle for

for (int i = 1; i <= n; i \*= 5) {

Este bucle multiplica `i` por 5 en cada iteración. El número de iteraciones es el valor k tal que 5^k <= n . El número de iteraciones se puede expresar como:

Dado que log\_5(n) es log n/log 5, el número de iteraciones es (O(log n)), ya que los logaritmos con diferentes bases solo difieren por un factor constante.

2. Análisis del Segundo Bucle for

for (int j = 1; j <= n; j += 2) {

Este bucle itera desde 1 hasta n en incrementos de 2. El número de iteraciones es aproximadamente n/2, que es (O(n)). La cantidad de trabajo en este bucle por iteración del primer bucle se puede expresar como:

Esto se simplifica a n/2, que es (O(n)).

3. Análisis del Tercer Bucle

for (int k = n; k >= 1; k /= 2) {

Este bucle divide k por 2 en cada iteración. El número de iteraciones es el valor m tal que ( n/2^m >= 1. El número de iteraciones se puede expresar como:

Dado que (log2(n)) es (O(log n)), el número de iteraciones es (O(log n)).

Complejidad Total

Para calcular la complejidad total del algoritmo, consideremos lo siguiente:

- El primer bucle for se ejecuta (O(log n)) veces.

- Dentro de cada iteración del primer bucle:

- El segundo bucle for se ejecuta (O(n)) veces.

- El tercer bucle for se ejecuta (O(log n)) veces.

La complejidad del segundo bucle dentro del primer bucle es:

O(n) por cada iteración del primer bucle

La complejidad del tercer bucle dentro del primer bucle es:

O(log n) por cada iteración del primer bucle

Así que, combinando estos:

1. El trabajo total realizado por el segundo bucle dentro de todas las iteraciones del primer bucle es:

O(log n) \* O(n) = O(n log n)

2. El trabajo total realizado por el tercer bucle dentro de todas las iteraciones del primer bucle es:

O(log n) \* O(log n) = O((log n)^2)

Sin embargo, dado que O(n log n)) es el término dominante y crece más rápido que ( O((log n)^2)) para valores grandes de n, la complejidad total del algoritmo está dominada por el término (O(n log n)).

Así, la complejidad total del algoritmo es (O(n log n)).

1. Para cada uno de los tiempos que toma un algoritmo en terminar, demostrar que cada uno es de la complejidad dada:

Para decidir cuál de los dos algoritmos, `Ex1` o `Ex2`, es más eficiente para resolver el problema, analizaremos la complejidad temporal de cada uno.

Análisis de `Ex1`

boolean Ex1(int[] a, int elem) {

int pos = buscar(a, elem);

int n = a.length;

int x = pos;

for (int i = 0; i < n; ++i) {

x += 2;

for (int j = 0; j < n; ++j) {

if (a[j] > a[pos]) {

x++;

}

}

}

return x > elem;

}

1. Costo de la función `buscar`:

La función `buscar` tiene una complejidad de O(n log n). Esta operación se realiza una vez al principio.

2. Complejidad del primer bucle externo:

El bucle externo for (int i = 0; i < n; ++i) se ejecuta n veces.

3. Complejidad del segundo bucle interno:

Dentro del bucle externo, hay otro bucle for (int j = 0; j < n; ++j) que también se ejecuta n veces. En cada iteración del bucle interno, se realiza una comparación if (a[j] > a[pos]), lo que toma O(1) tiempo.

El costo de la parte interna del segundo bucle es (O(n)), y como este bucle se encuentra dentro de un bucle externo que también se ejecuta (n) veces, la complejidad total del bucle anidado es:

O(n) \* O(n) = O(n^2)

4. Complejidad total de `Ex1`:

La complejidad total del algoritmo `Ex1` es la suma de la complejidad de `buscar` más la complejidad del bucle anidado:

O(n log n) + O(n^2) = O(n^2)

- Análisis de `Ex2`

boolean Ex2(int[] a, int elem) {

int n = a.length;

int x = 0;

for (int i = 0; i < n; ++i) {

int pos = buscar(a, elem);

x += pos + 2;

for (int j = 0; j < n; ++j) {

if (a[j] > a[pos]) {

x++;

}

}

}

return x > elem;

}

1. Costo de la función buscar:

La función buscar tiene una complejidad de O(n log n)), y esta operación se realiza dentro del primer bucle. Dado que el primer bucle se ejecuta (n) veces, el costo total de las llamadas a `buscar` es:

n \* O(n log n) = O(n^2 log n)

2. Complejidad del primer bucle externo:

El primer bucle for (int i = 0; i < n; ++i)` se ejecuta \( n \) veces.

3. Complejidad del segundo bucle interno:

Dentro del bucle externo, hay un bucle for (int j = 0; j < n; ++j) que se ejecuta n veces. La comparación if (a[j] > a[pos]) toma O(1) tiempo.

Como en `Ex1`, el costo de la parte interna del segundo bucle es O(n). Dado que este bucle está dentro del bucle externo que se ejecuta n veces, la complejidad total del bucle anidado es:

O(n) \* O(n) = O(n^2)

4. Complejidad total de `Ex2`:

La complejidad total del algoritmo `Ex2` es la suma del costo de las llamadas a `buscar` y el costo del bucle anidado:

O(n^2 log n) + O(n^2)

Dado que (O(n^2 log n) domina a (O(n^2)), la complejidad total es:

O(n^2 log n)

Comparación de `Ex1` y `Ex2`

- `Ex1` tiene una complejidad de (O(n^2)).

- `Ex2` tiene una complejidad de (O(n^2 log n)).

- El algoritmo `Ex1` es más eficiente que `Ex2` debido a que su complejidad es menor (( O(n^2)) frente a (O(n^2 log n)).

- Por lo tanto, se recomienda usar el algoritmo `Ex1` para resolver el problema.

La razón es que el costo adicional causado por las llamadas repetidas a buscar en `Ex2` hace que su complejidad sea mayor que la de `Ex1`, que realiza la búsqueda solo una vez.

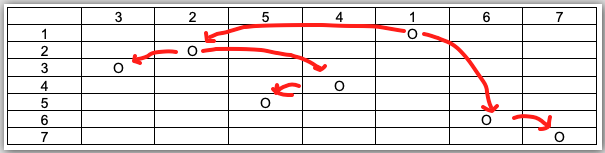
# Ejercicios sobre Árboles Binarios:

1. Para el Árbol binario (1):
2. **Peso:** 11
3. **Altura:** 5
4. **Hojas:** 5
5. **Una rama:** 64, 55, 48, 33, 39
6. **Recorrido en inorder:** 33, 39, 48, 50, 55, 57, 64, 74, 76, 78, 81
7. **Recorrido en preorder:** 64, 55, 48, 33, 39, 50, 57, 76, 74, 78, 81
8. **Recorrido en postorder:** 39, 33, 50, 48, 57, 55, 74, 81, 78, 76, 64
9. Para el Árbol binario (2):
10. **Numero de niveles:** 4
11. **Ancestro común de la E** **y la A:** B
12. **Altura:** 4
13. **Peso del árbol Izquierdo de la F:** 6
14. **Recorrido en inorder:** A, B, C, D, E, F, G, H, I
15. **Recorrido en preorder:** F, B, A, D, C, E, G, I, H
16. **Recorrido en postorder:** A, C, E, D, B, H, I, G, F
17. Reconstruya el Árbol binario que posee los siguientes recorridos:

Preorder: 1-2-3-4-5-6-7

Inorder: 3-2-5-4-1-6-7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **3** | **2** | **5** | **4** | **1** | **6** | **7** |
| **1** |  |  |  |  | O |  |  |
| **2** |  | O |  |  |  |  |  |
| **3** | O |  |  |  |  |  |  |
| **4** |  |  |  | O |  |  |  |
| **5** |  |  | O |  |  |  |  |
| **6** |  |  |  |  |  | O |  |
| **7** |  |  |  |  |  |  | O |



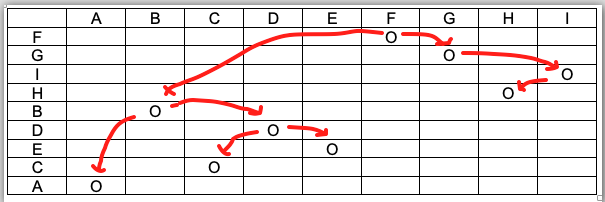
*Elaboración propia*

1. Reconstruya el Árbol binario que posee los siguientes recorridos:

Postorder: A-C-E-D-B-H-I-G-F

Inorder: A-B-C-D-E-F-G-H-I

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** |
| **F** |  |  |  |  |  | O |  |  |  |
| **G** |  |  |  |  |  |  | O |  |  |
| **I** |  |  |  |  |  |  |  |  | O |
| **H** |  |  |  |  |  |  |  | O |  |
| **B** |  | O |  |  |  |  |  |  |  |
| **D** |  |  |  | O |  |  |  |  |  |
| **E** |  |  |  |  | O |  |  |  |  |
| **C** |  |  | O |  |  |  |  |  |  |
| **A** | O |  |  |  |  |  |  |  |  |



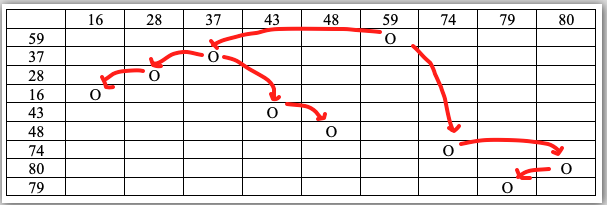
*Elaboración propia*

1. Reconstruya el Árbol binario que posee los siguientes recorridos:

Preorder: 59-37-28-16-43-48-74-80-79

Inorder: 16-28-37-43-48-59-74-79-80

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 16 | 28 | 37 | 43 | 48 | 59 | 74 | 79 | 80 |
| 59 |  |  |  |  |  | O |  |  |  |
| 37 |  |  | O |  |  |  |  |  |  |
| 28 |  | O |  |  |  |  |  |  |  |
| 16 | O |  |  |  |  |  |  |  |  |
| 43 |  |  |  | O |  |  |  |  |  |
| 48 |  |  |  |  | O |  |  |  |  |
| 74 |  |  |  |  |  |  | O |  |  |
| 80 |  |  |  |  |  |  |  |  | O |
| 79 |  |  |  |  |  |  |  | O |  |



*Elaboración propia*

# Link Repositorio:

https://github.com/jsguerra07/DDS\_Guia3

# Conclusiones:

* El análisis de la complejidad algorítmica es esencial para diseñar algoritmos eficientes y escalables. Permite a los desarrolladores entender el comportamiento de sus algoritmos en términos de tiempo y espacio, y tomar decisiones informadas sobre cuál utilizar en diferentes contextos.
* Los árboles binarios son estructuras de datos jerárquicas en las que cada nodo tiene, a lo sumo, dos hijos. Esta estructura permite una organización eficiente de los datos, facilitando operaciones como búsqueda, inserción y eliminación.
* A través de este trabajo, hemos comprendido cómo se puede construir un árbol binario a partir de sus recorridos.
* las listas enlazadas son una estructura de datos versátil y eficiente para una amplia serie de aplicaciones que requieren inserciones y eliminaciones rápidas. A pesar de sus ventajas, como la flexibilidad y la eficiencia en la gestión dinámica de memoria, también presentan desventajas como la necesidad de acceso secuencial y el uso adicional de memoria para referencias. La elección de usar listas enlazadas en lugar de arrays u otras estructuras de datos debe basarse en los requisitos específicos de la aplicación y las operaciones predominantes.

# Bibliografía

Villalobos, J. & Casallas R. (2006). *Fundamentos de Programación*. Editorial Pearson, Prentice Hall.