UNIVERSIDAD CATOLICA DE SANTA MARIA
FACULTAD DE CIENCIAS FÍSICAS Y FORMALES
ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

Complejidad Algoritmica

Autores:
Cusirramos Chiri Santiago Jesus (2023601381)

Docente:
Marisol Cristel Galarza Flores

Curso:
Algoritmos y Estructura de Datos

AREQUIPA-PERÚ
2025

Índice

1. Reconocimientos

El autor de este trabajo reconoce con gratitud a los creadores de los lenguajes JAVA y otras personalidades y autores de libros de programación Bjarne Stroustrup, Dennis Ritchie, Herb Sutter, Herb Sutter, James Gosling, James Gosling, Brian Kernighan, Brian Kernighan, Ken Thompson.

2. Objetivo

- Comprender las reglas y métodos adecuados para evaluar la eficiencia de los algoritmos.
- Determinar la complejidad temporal de diferentes algoritmos.
- Aplicar la notación asintótica en la evaluación de la eficiencia de algoritmos

3. Palabras clave

- Eficiencia.
- Notación O.
- Complejidad de tiempo y memoria.
- Reglas Asintoticas.
- Complejidad de algoritmos.
- Recursividad.

4. Resumen

En el presente informe se explica la eficiencia temporal de los algoritmos y cómo se mide mediante la función de complejidad $T(n)$, considerando tres criterios: mejor caso, caso promedio y peor caso, siendo este último el más seguro para la evaluación. Luego, introduce la notación Big O, que permite analizar la complejidad asintótica de un algoritmo, proporcionando un límite superior para su tiempo de ejecución. La notación $O(f(n))$ ignora factores constantes y permite comparar distintos algoritmos en función de su eficiencia relativa, facilitando la elección de soluciones óptimas para problemas computacionales.

Posteriormente, se describen las funciones de complejidad más comunes, ordenadas de mayor a menor eficiencia, desde $O(1)$ (constante) hasta $O(2^n)$ (exponencial). Se presentan ejemplos como la búsqueda binaria con $O(\log n)$, algoritmos de ordenación como Quicksort con $O(n \log n)$ y algoritmos con ciclos anidados, cuya complejidad puede ser $O(n^2)$ o superior. Se destacan los algoritmos polinomiales como viables y los exponenciales como imprácticos en la mayoría de los casos debido a su alto costo computacional.

Finalmente, se explican reglas de notación asintótica y se analiza la complejidad de distintos tipos de instrucciones, desde asignaciones y secuencias hasta estructuras condicionales y bucles. También se aborda el impacto de las llamadas a procedimientos y el análisis de algoritmos recursivos mediante funciones de recurrencia, utilizando técnicas como la expansión de recurrencias para determinar su tiempo de ejecución.

5. Introducción

Cuando se trata de la eficiencia o la complejidad de un algortimo, esta hace referencia a la cantidad de recursos computacionales necesarios para su almacenamiento y ejecucion en un sistemas. Estos recursos se dividen principalmente en dos categorias.

- **Tiempo:** El periodo que transcurre entre el inicio y el fin de la ejecución del algoritmo.
- **Memoria:** La cantidad de espacio (que varia según el tipo de maquina) que el algoritmo necesita para llevar a cabo su proceso.

Cabe resaltar que cuando un algoritmo emplea menos recursos, se considera mas eficiente o de menor complejidad en comparación con otros. Tanto la capacidad como el diseño del sistema computacional pueden influir en la implementación del algoritmo.

Nos centraremos en el análisis de la eficiencia de los algoritmos, particularmente en relación con el tiempo de ejecución.

6. Marco Teórico

6.1. Eficiencia de tiempo.

La eficiencia temporal de un algoritmo se evalua mediante el analisis de su eficiencia teorica, utilizando medidas que expresan dicha eficiencia sin unidades concretas, a traves de funciones matematicas. Al determinar que la complejidad de un algoritmo es $T(n) = c * f(n)$, esto implica que el tiempo de ejecucion del algoritmo es proporcional a una funcion $f(n)$ y no una unidad de tiempo especifca.

Nosotros empleamos 3 criterios para calcula $T(n)$:

1. Considerar $T(n)$ como el tiempo correspondiente al mejor de los casos posibles.
2. Considerar $T(n)$ como el tiempo promedio de todos los casos posibles
3. Considerar $T(n)$ como el tiempo correspondiente al peor de los casos posibles.

Aunque las dos primeros medidas podrían parecer optimas, es importante tener en cuenta que no todos los casos posibles tiene la misma probabilidad de ocurrir. La 3ra medida, por otra parte, aunque aparentemente menos ajustada, es mas segura y facil de evaluar. En otras palabras, haremos la evalucion considerando el peor de los casos para asegurar mayor optimizacion.

6.2. Notacion GIB O().

El BIG O es una notacion matematica empleada para la descripcion de la complejidad de los algoritmos, específicamente para medir el tiempo de ejecución o el uso de espacio en función de tamaño de la entrada. Big O proporciona un limite superior asintotico para el comportamiento de un algoritmo en su peor caso, ayudando a comparar la eficiencia de diferentes algoritmos.

De manera formal, se dice que una función $f(n)$ es de orden $O(g(n))$ si y solo si, existen constantes positivas C y n_0, ambas independientemente de n, tales que se cumple la siguiente relación:

$$f(n) \leq c \cdot g(n), \forall n \geq n_0 \tag{1}$$

Por ejemplo, se puede comprobar que la funcion $t(n) = 3n^3 + 2n^2$ es de $O(n^3)$, aplicando la definición anterior:

$$t(n) = 3n^3 + 2n^2 \tag{2}$$
$$g(n) = n^3 \tag{3}$$

Si se elige $n_0 = 0, c = 5$, se cumple:

$$3n^3 + 2n^2 \leq 5n^3, \forall n \geq 0 \tag{4}$$

En general, el tiempo de ejecución de un algoritmo es proporcional a la función $f(n)$ es decir, se multiplica por una constante a uno de los tiempos de ejecución previamente propuestas, ademas de sumar algunas términos mas pequeños. Por ejemplo, un algoritmo cuyo tiempo de ejecución sea $T(n) = 3n^2 + 6n$ se considera proporcional a $2n^2$, por lo que se diría que el algoritmo tiene una complejidad de $O(n^2)$.

La notación $O(.)$ ignora los factores constantes, es decir, no considera si la implementación del algoritmo es mejor, y es independiente de los datos de entrada. El principal objetivo de aplicar esta notación a un algoritmo es encontrar un limite en el tiempo de ejecucion, es decir, el peor caso.

6.3. Funciones de complejidad mas usuales.

Las funciones de complejidad algorítmica mas frecuente depende principalmente del tamaño de la entrada n, y se ordenan de mayor a menos eficiencia de la siguiente manera:

- $O(1)$: **Complejidad Constante.** Se refiere a los algoritmos en los que la mayoria de las instruccion se ejecutan una sola vez o muy pocas veces, es la complejidad mas deseada.
- $O(\log n)$:**Complejidad logarítmica.** Aparece en ciertos algoritmos con iteración o recursión no estructural, como por ejemplo en la búsqueda binaria.
- $O(n)$: **Complejidad Lineal.** Es una de las complejidades mas comunes y eficientes. Se presenta en la evaluación de bucles simples cuando la complejidad de las operaciones dentro del bucle es constante.
- $O(n \log n)$: **Complejidad log-lineal.** Se encuentra en algoritmos de tipo "divide y vencerás", como en el caso del algoritmo de ordenación rápida (Quicksort). Se considera una buena complejidad porque, al duplicar n, el tiempo de ejecución aumenta ligeramente más del doble.
- $O(n^2)$:**Complejidad cuadrática.** Aparece en algoritmos con bucles o ciclos doblemente anidados. Si n se duplica, el tiempo de ejecución aumenta cuatro veces.
- $O(n^3)$:**Complejidad cubica.** Suele darse en algoritmos con bucles o ciclos triple-mente anidados. Si n se duplica, el tiempo de ejecución se multiplica por ocho. Para valores grandes de n, el tiempo de ejecución comienza a crecer de forma excesiva.
- $O(n^k)$: **Complejidad polinómica** (donde K >3). A medida que K aumenta, la complejidad del algoritmo se vuelve considerablemente mas eficiente.
- $O(n^k)$: **Complejidad exponencial.** Esta complejidad se caracterizaba por un tiempo de ejecución extremadamente alto, por lo que en la practica no suele ser útil. Se presenta en algoritmos recursivos que contienen dos o mas llamadas recursivas.
- **Algoritmos polinomiales:** Son aquellos cuya complejidad depende de n^k. Estos algoritmos son generalmente viables y prácticos.

- **Algoritmos exponenciales:** Son aquellos cuya complejidad depende de k^n. Estos algoritmos suelen no ser factibles, excepto cuando el tamaño de la entrada en muy pequeño.

Este enfoque proporciona una descripción mas fluida y clara de las diferentes clases de complejidad temporal que puede encontrarse en los algoritmos.

6.4. Reglas de notación asintótica.

1. **Regla de la suma:** Si $T1(n)$ y $T2(n)$ son las funciones que expresan los tiempos de ejecucion de datos fragmentados del programa que se acotan de forma que se tiene $T1(n) = O(f(n))$ y $T2(n) = O(g(n))$, se puede decir que $T1(n) + T2(n) = O(max(f(n), g(n)))$
2. **Regla del producto:** Si $T1(n)$ y $T2(n)$ son las funciones que expresan los tiempos de ejecucion de dos fragmentos de programa, y se acotan de forma que se tiene $T1(n) = O(f(n))$ y $T2(n) = O(g(n))$, se puede decir que $T1(n).T2(n) = O(f(n).g(n))$

6.5. Análisis de complejidad de algoritmos no recursivos.

- **Asignaciones y expresiones simples:** El tiempo de ejecución de toda instrucción de asignación simple, de la evaluación de una expresión formadas por términos simples o de toda constante es $O(1)$
- **Secuencia de instrucciones:** El tiempo de ejecucion de una secuencia de instrucciones es igual a la suma de sus tiempos de ejecuciones respectivos. Para una secuencia de dos instrumentos $I1$ e $I2$ se tiene que el tiempo de ejecuciones esta dado por la suma de los tiempos de ejecuciones de $I1$ e $I2$.

$$T(I_1; I_2) = T(I_1) + T(I_2) \tag{5}$$

Aplicando la regla de la suma:

$$O(T(I_1; I_2)) = max(O(T(I_1) + T(I_2))) \tag{6}$$

- **Instrucciones condicionales.** El tiempo de ejecución requerido por una instrucción condicional SI-ENTONCES es:

$$T(Si - Entonces) = T(Condicion) + t(RamaEntonces) \tag{7}$$

El tiempo para una instrucción condicional del tipo Si-Entonces-Sino es:

$$T(Si - Entonces - Sino) = T(Condicion) + max(t(RamaEntonces), T(RamaSino)) \tag{8}$$

El tiempo de ejecución de un condicional múltiple (SWITCH) es el tiempo necesario para evaluar la condición, mas el mayor de los tiempos de las secuencias a ejecutar en cada valor condicional.

- **Instrucciones condicionales.** La complejidad en tiempo de un nucle PARA(FOR) es el producto del numero de iteraciones por la complejidad de las instrucciones del cuerpo de bucle o ciclo. Para los ciclos del tipos $Mientras - Hacer(While - Do)$ y $Hacer - Mientras(Do - While)$ se sigle la regla anterior , pero se considera la estimación del numero de iteraciones para el peor caso posible. Si existen ciclos anidados, se realiza el análisis de adentro hacia afuera, considerando el tiempo de ejecución de un ciclo interior y la suma del resto de proposiciones como el tiempo de ejecución de una iteración del ciclo exterior.

6.6. Llamadas a procedimientos.

Su tiempo de ejecución esta dado por el tiempo requerido para ejecutar el cuerpo del procedimiento. Si un procedimiento hace llamadas a otros procedimientos, es posible calcular el tiempo de ejecución de cada procedimiento llamado, partiendo de aquellos que no llaman a ninguno. Al menos tiene que existir un procedimiento que no llama a otros, o se estaría en un caso recursivo. En las invocaciones a otros procedimientos se debe considerar el tiempo u orden de complejidad de los procedimientos invocados.

6.7. Análisis de complejidad de algoritmos recursivos.

Debemos especificar la función de recurrencia que represente el tiempo de ejecución recursivo $T(n)$. Es posible aplicar las pautas establecidas para el análisis de expresiones simples, secuencias, condicionales y bucles. Una vez definida la función de recurrencia, se pueden emplear varios métodos, como la expansión de recurrencias, que consiste en reemplazar las recurrencias por su igualdad sucesivamente hasta llegar a un valor conocido de $T(n_0)$.

7. Actividad

7.1. Obtener el orden de complejidad de algoritmos iterativos.

1. Analice los procedimeitnos, funcines y obtenga el orden BigO correspondiente aplicando las reglas asintóticas y las pautas para el análisis de algoritmos.

a) Función intercambia.

P3:Actividades:Intercambia

```
public static void intercambia(int x, int y){
    int aux;    // O(1)
    aux = x;    // O(1)
    x = y;      // O(1)
    y = aux;    // O(1)
}
```

En este caso, cada operacion tiene un costo de $O(1)$. Esto debido a que no hay bucles ni operaciones recursivas, la complejidad total del procedimeitno es:

$$O(1) + O(1) + O(1) + O(1) = O(1) \tag{9}$$

Llegamos a esta respuesta de $O(1)$ por la regla de la suma, donde maximizamos las expresiones y hallamos una general.

b) Función Max

P3:Actividades:Max

```
public static int max(int x, int y){
    int result;         // O(1)
    if(x > y){          // O(1)
        result = x;     // O(1)
    } else {
        result = y;     // O(1)
    }
    return result;      // O(1)
}
```

Para este caso, cada operacion presenta un $O(1)$.

$$O(1) + O(1) + O(1) + O(1) + O(1) = O(1) \tag{10}$$

Llegamos a esta conclusión, empleando la composición de instrucciones condicionales nos percatamos que esto es de $O(1)$

c) Función Suma

P3:Actividades:Max

```
public static int suma(int[] v, int size) {
    int result = 0;                     // O(1)
    for (int i = 0; i < size; i++) {    // O(1) + O(n)
        result += v[i];                 // O(n)
    }
    return result;                      // O(1)
}
```

Para esta funcion lo que aplicaremos es la regla de la suma. Partiendo de adentro hacia afuera(bucles y/o condiconales).

Nuestro primer FOR nos da un $= O(n)$ debido a que N veces se iterara la suma, posteriormente el mismo for se va a iterar N veces hasta que cumpla con la condicion del tamaño. En este FOR vamos a aplicar la regla de la suma, como resultado obtenemos $O(n)$.

Las operaciones fuera del FOR representan un $O(1)$.

Entonces esto con una maximizacion de valores nos quiere decir que esta función es de $O(N)$

d) Función Ordenar

P3:Actividades:Ordenar

```
public static void ordenar(int[] v, int size) {
    int i, j, aux;                          // O(1)
    for (i = 0; i < size - 1; i++) {        // O(1) + O(n) + O(n)
        for (j = 0; j < size - 1; j++) {    // O(1) + O(n) + O(n)
            if (v[j] > v[j + 1]) {          // O(1)
                aux = v[j];                 // O(1)
                v[j] = v[j + 1];            // O(1)
                v[j + 1] = aux;             // O(1)
            }
        }
    }
}
```

Partimos del condicional, este orden considerando todos los factores nos da $O(1)$. Para el FOR $O(n) + O(n)$ nos da un resultado de $O(n)$. Esto lo maximizamos con el orden del codicional. Esto nos da como un orden $O(n)$ para las operaciones del 1er for.

Para el 2do FOR, nos da un resultado de $O(n)$.

Para este caso, en lugar de aplicar la regla de la suma, debemos de aplicar la regla del producto la cual juntas las dos funciones.

$$T1(n).T2(n) = O(f(n).g(n)) \tag{11}$$

Siendo nuestro For1 T1 y nuestro FOR2 T2.

$$T1(n).T2(n) = O(for1(n).for2(n)) \tag{12}$$
$$T1(n).T2(n) = O(n^2) \tag{13}$$

Este siendo nuestro resultado final para esta función.

7.2. Obtener el orden de complejidad de algoritmos no iterativos.

1. Para la siguiente función recursiva y aplicando las reglas asintóticas obtenga.
   - La función de recurrencia correspondiente.
   - A partir de la función de recurrencia determine el orden de complejidad aplicando el método de expansión de recurrencias.

P3:Actividades:Potencia

```
public static double potencia(double x, double y) {
        double t;
        if (y == 0) {                           // O(1)
            return 1.0;                         // O(1)
        }
        if (y % 2 == 1) {                       // O(1)
            return x * potencia(x, y - 1);   // O(y - 1)
        }
        else {
            t = potencia(x, y / 2);          // O(y / 2)
            return t * t;                       // O(1)
        }
    }
```

La funcion potencial, calcula de manera recursiva diferentes casos.
Casos/condiciones:

- Si $y = 0$, el valor que retorna es 1.
- Si es impar, realiza una recursiva con $y - 1$ multipllicando por x.
- Si es par, realiza una recursiva con $y/2$, multiplicandose por si mismo.

$$T(y) = \begin{cases} O(1) & \text{si } y = 0 \\ T(y-1) + O(1) & \text{si } y \text{ es impar} \\ T(y/2) + O(1) & \text{si } y \text{ es par} \end{cases} \tag{14}$$

1. Caso: Cuando es impar

$$T(y) = T(y-1) + O(1) \tag{15}$$
$$T(y) = T(y-2) + 2 \tag{16}$$
$$T(y) = T(y-3) + 3 \tag{17}$$
$$T(y) = T(y-4) + 4 \tag{18}$$
$$\tag{19}$$

Para K:

$$T1(y) = T2(y-k) + k \tag{20}$$

Entonces $K = Y$

$$T(y) = T(y-y) + y \tag{21}$$
$$T(y) = O(T(y)) \tag{22}$$

Aqui obtenemos como resultado que nuestra funcion $T(y)$ es igual a $O(y)$, en otras palabras la complejidad es lineal

2. Caso: Cuando es par

$$T(y) = T(y/2) + O(1) \tag{23}$$
$$T(y) = T(y/4) + 2 \tag{24}$$
$$T(y) = T(y/8) + 3 \tag{25}$$
$$T(y) = T(y/16) + 4 \tag{26}$$
$$\tag{27}$$

Para K:

$$T(y) = T(y/2^k) + k \tag{28}$$

Entonces, para este caso... Cual es el valor de K?

$$y/2^k = 1 \tag{29}$$
$$y = 2^k \tag{30}$$
$$\tag{31}$$
$$k? = y \tag{32}$$
$$\tag{33}$$
$$k = \log_2 y \tag{34}$$

Entonces decimos de K es igual a $\log_2 y$.

$$T(y) = T(y/2^k) + k \tag{35}$$
$$T(y) = T(y/2^{\log_2 y}) + \log_2 y \tag{36}$$
$$T(y) = T(1) + \log_2 y \tag{37}$$
$$T(y) = O(T(\log_2 y)) \tag{38}$$

Aqui obtenemos como resultado que nuestra funcion $T(y)$ es igual a $O(T(\log_2 y))$, en otras palabras la complejidad es logaritmica.

Conclusión sobre la complejidad.

- **Par:** Este es el mejor caso ya que presente una complejidad logarítmica.
- **Impar:** Este es nuestro peor caso al presentar una complejidad lineal.

8. Ejercicio

8.1. Halle el orden de complejidad de los siguientes algoritmos no recursivos:

1. Ejercicio BM

P3:Ejercicios:BM

```
public static int BM(int[] v, int n) {
    int m = v[0];                          // O(1)
    for (int i = 1; i < n - 1; i++) {      // O(1) + O(n) + O(n)
        if (v[i] > m) {                    // O(1)
            m = v[i];                      // O(1)
        }
    }
    return m;                              // O(1)
}
```

Para esta función lo que aplicaremos es la regla de la suma. Partiendo de adentro hacia afuera(bucles y/o condicionales).

El IF() al ser un condicional simple, sin ningún tipo de iteración o recursividad dentro, nos retorna/ indica que su complejidad es constante, en otras palabras su pero es $O(1)$.

El FOR, por otro lado, nos da un $= O(n)$ debido a que N veces se iterara la operaciones dentro del mismo, posteriormente el mismo for se va a iterar N veces hasta que cumpla con la condicion del tamaño. En este FOR vamos a aplicar la regla de la suma con los datos que contenga, como resultado de esta suma obtenemos $O(n)$.

Las operaciones fuera del FOR representan un $O(1)$.

Entonces esto con una maximizacion de valores nos quiere decir que esta función es de $O(N)$

2. Ejercicio ConteoL

P3:Ejercicios:ConteoL

```
public static int ConteoL(int[] v, int n) {
    int conteo = 0;
    for (int i = 0; i < n - 2; i++) {
        for (int j = i + 1; j < n - 1; j++) {
            if (v[i] == v[j]) {
                conteo += 1;
            }
        }
    }
    return conteo;
}
```

Vamos a examinar parte por parte, inciando con nuestro 1er FOR.

1er For

```
for (int j = i + 1; j < n - 1; j++) {   // O(n)
            if (v[i] == v[j]) {   // O(1)
                conteo += 1;      // O(1)
            }
        }
```

1ro la condicional de adentro.

$$T(cond) + T(then) = max(O(Cond, Then)) \tag{39}$$
$$T(cond) + T(then) = max(O(01, 01)) \tag{40}$$
$$T(cond) + T(then) = O(01) \tag{41}$$

Ahora el 1er FOR.

$$T(cond) + T(then) = max(O(Cond, Then)) \quad (42)$$
$$T(for) + T(if) = max(O(n, 01)) \quad (43)$$
$$T(for) + T(if) = O(n) \quad (44)$$

Posteriormente vamos a examinar el sigueinte FOR.

2do For

```
for (int i = 0; i < n - 2; i++) {
...  <- O(n)
}
```

Este for tambien es un $O(n)$, ahora he de juntarlo con nuestros resultados anteriores.

$$T_1 * T_2 = O(f_1(1erFor) * f_2(2doFor)) \quad (45)$$
$$T_1 * T_2 = O((n) * (n)) \quad (46)$$
$$T_1 * T_2 = O(n^2) \quad (47)$$
$$(48)$$

Para este caso, nuestra complejidad seria una cuadrática.

3. Ejercicio Moda

P3:Ejercicios:Moda

```
public static int moda(int[] v) {
    Map<Integer, Integer> frecuencia = new HashMap<>(); // O(1)
    int maxFrecuencia = 0;                              // O(1)
    int moda = v[0];                                    // O(1)

    for (int num : v) {                                 // O(n)
        int f = frecuencia.getOrDefault(num, 0) + 1; // O(1)
        frecuencia.put(num, f);                         // O(1)

        if (f > maxFrecuencia) {                        // O(1)
            maxFrecuencia = f;                          // O(1)
            moda = num;                                 // O(1)
        }
    }
    return moda;
}
```

Vamos a analizar nuestro condicional.

Condicional

```
if (f > maxFrecuencia) {                    // O(1)
    maxFrecuencia = f;                      // O(1)
    moda = num;                             // O(1)
}
```

Regla de la Suma, iniciando con las acciones de dentro.

$$T1(n) + T2(n) = O(max(f(1), g(1))) \tag{49}$$
$$T1(n) + T2(n) = O(1) \tag{50}$$

Comprobando la condición con el resultado obtenido.

$$T_{selección}(n) = T_{condición}(n) + T_{then}(n) \tag{51}$$
$$T_{selección}(n) = T_{condición}(1) + T_{then}(1) \tag{52}$$
$$T_{selección}(n) = O(1) \tag{53}$$

Ahora escalaremos al siguiente nivel, comparar con el FOR.

P3:Actividades:Moda

```
for (int num : v) {                                 // O(n)
    int f = frecuencia.getOrDefault(num, 0) + 1;    // O(1)
    frecuencia.put(num, f);                         // O(1)

    if (f > maxFrecuencia) {                        // O(1)
        ...
    }
}
```

Iniciando con las acciones de dentro

$$T1(n) + T2(n) + T3(n) = O(max(f(1), g(1), h(1))) \tag{54}$$
$$T1(n) + T2(n) + T3(n) = O(1) \tag{55}$$

Comentar que las funciones empleadas dentro de este for, son $O(1)$ ya que se emplean para designar datos, tienen sus respectivas comprobaciones pero al final terminaran siendo $O(1)$.

Funciones

```
// 1ra funcion
default V getOrDefault(Object key, V defaultValue) {
    V v;
    return (((v = get(key)) != null) || containsKey(key))
            ? v
            : defaultValue;
}

// 2da funcion
V put(K key, V value);
```

Ahora vamos a utilizar los valores para la comprobacion con el FOR, empleando la regla de la suma.

$$T1(for) + T2(n) = O(max(t(n), t(1))) \tag{56}$$
$$T1(for) + T2(n) = O(n) \tag{57}$$

Validando los valores con las acciones restantes, obtenemos que la función es de orden $O(n)$.

4. Ejercicio Potencia Rapida.

P3:Ejercicios:PotenciaRapida

```
public static int potenciaRapida(int x, int y) {
    if (y == 0) {
        return 1;
    }
    if (y % 2 == 0) {
        int mitad = potenciaRapida(x, y / 2);
        return mitad * mitad;
    }
    else {
        return x * potenciaRapida(x, y - 1);
    }
}
```

La funcion potencial, calcula de manera recursiva diferentes casos.

Casos/ Condiciones.

- Si $y = 0$, el valor que retorna es 1.

- Si es par, realiza una recursiva con $y/2$, para finalizar recibiendo el valor y multiplicándose por si mismo.
- Si es impar, realiza una recursiva con $y-1$ multiplicando por x.

$$T(y) = \begin{cases} O(1) & \text{si } y = 0 \\ T(y/2) + O(1) & \text{si } y \text{ es par} \\ T(y-1) + O(1) & \text{si } y \text{ es impar} \end{cases} \tag{58}$$

a) Caso: Cuando es par

$$T(y) = T(y/2) + O(1) \tag{59}$$
$$T(y) = T(y/4) + 2 \tag{60}$$
$$T(y) = T(y/8) + 3 \tag{61}$$
$$T(y) = T(y/16) + 4 \tag{62}$$
$$\tag{63}$$

Para K:

$$T(y) = T(y/2^k) + k \tag{64}$$

Entonces, para este caso... Cual es el valor de K?

$$y/2^k = 1 \tag{65}$$
$$y = 2^k \tag{66}$$
$$k? = y \tag{67}$$
$$k = \log_2 y \tag{68}$$

Entonces decimos de K es igual a $\log_2 y$.

$$T(y) = T(y/2^k) + k \tag{69}$$
$$T(y) = T(y/2^{\log_2 y}) + \log_2 y \tag{70}$$
$$T(y) = T(1) + \log_2 y \tag{71}$$
$$T(y) = O(T(\log_2 y)) \tag{72}$$

Aquí obtenemos como resultado que nuestra función $T(y)$ es igual a $O(T(\log_2 y))$, en otras palabras la complejidad es logarítmica.

b) Caso: Cuando es impar

$$T(y) = T(y-1) + O(1) \tag{73}$$
$$T(y) = T(y-2) + 2 \tag{74}$$
$$T(y) = T(y-3) + 3 \tag{75}$$
$$T(y) = T(y-4) + 4 \tag{76}$$
$$\tag{77}$$

Para K:

$$T1(y) = T2(y-k) + k \tag{78}$$

Entonces $K = Y$

$$T(y) = T(y-y) + y \tag{79}$$
$$T(y) = O(T(y)) \tag{80}$$

Aqui obtenemos como resultado que nuestra funcion $T(y)$ es igual a $O(y)$, en otras palabras la complejidad es lineal

Conclusión sobre la complejidad.

- **Par:** Este es el mejor caso ya que presente una complejidad logarítmica.
- **Impar:** Este es nuestro peor caso al presentar una complejidad lineal.

5. A partir de la siguiente funcino de recurrencia de un algoritmo, halle el orden de complejidad correspondiente:

$$T(n) = \begin{cases} T(n/2) + 3 & \text{si } n \geq 2 \\ 7 & \text{si } n = 1 \end{cases} \tag{81}$$

Analizando la recursividad:

$$T(n) = T(n/2) + 3 \tag{82}$$
$$T(n) = T(n/4) + 6 \tag{83}$$
$$T(n) = T(n/8) + 9 \tag{84}$$
$$T(n) = T(n/16) + 12 \tag{85}$$
$$\tag{86}$$

Para K:

$$T(n) = T(n/2^k) + 3 * K \tag{87}$$

Entonces, para este caso... Cual es el valor de K?

$$n/2^k = 1 \tag{88}$$
$$n = 2^k \tag{89}$$
$$k? = n \tag{90}$$
$$k = \log_2 n \tag{91}$$

Entonces decimos que K es igual a $\log_2 n$.

$$T(n) = T(n/2^k) + k \tag{92}$$
$$T(n) = T(n/2^{\log_2 n}) + 3 * \log_2 n \tag{93}$$
$$T(n) = T(1) + 3 * \log_2 n \tag{94}$$
$$T(n) = 7 + T(\log_2 n) \tag{95}$$

Con su maximizacion respectiva, obtenemos que el orden de esta funcion recursiva es logaritmica $T(\log_2 n)$

6. Investigue acerca del **algoritmo de ordenación por fusión (Merge Sort)** y cómo puede ser utilizado para ordenar un arreglo de enteros. En base a la información obtenida, escriba la implementación en Java de un algoritmo de **O(n log n)** que permita ordenar un arreglo de enteros utilizando el algoritmo de ordenación por fusión. Explique utilizando las reglas de la notación asintótica el por qué su orden de complejidad

MergeSort

```java
public static int[] mergeSort(int[] arr) {
    if (arr.length == 1) {
        return arr;
    }

    int middle = arr.length / 2;
    int[] leftArray = new int[middle];
    int[] rightArray = new int[arr.length - middle];

    for (int i = 0; i < middle; i++) {
        leftArray[i] = arr[i];
    }
    for (int i = middle; i < arr.length; i++) {
        rightArray[i - middle] = arr[i];
    }

    int[] sortedLeftArray = mergeSort(leftArray);
    int[] sortedRightArray = mergeSort(rightArray);

    return merge(sortedLeftArray, sortedRightArray);
}
```

Merge

```
private static int[] merge(int[] leftArray, int[] rightArray) {
       int[] result = new int[leftArray.length + rightArray.length];
       int leftIndex = 0, rightIndex = 0, resultIndex = 0;

       while (leftIndex < leftArray.length && rightIndex < rightArray.length) {
           if (leftArray[leftIndex] <= rightArray[rightIndex]) {
               result[resultIndex++] = leftArray[leftIndex++];
           } else {
               result[resultIndex++] = rightArray[rightIndex++];
           }
       }

       while (leftIndex < leftArray.length) {
           result[resultIndex++] = leftArray[leftIndex++];
       }

       while (rightIndex < rightArray.length) {
           result[resultIndex++] = rightArray[rightIndex++];
       }

       return result;
   }
```

La complejidad algorítmica de la Funcion MergeSort es:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + n & \text{sino} \end{cases} \tag{96}$$

Este con el siguiente desarrollo.

Analizando la recursividad:

$$T(n) = 2T(n/2) + n \tag{97}$$
$$T(n) = 4T(n/4) + 2n \tag{98}$$
$$T(n) = 8T(n/8) + 3n \tag{99}$$
$$T(n) = 16T(n/16) + 4n \tag{100}$$
$$\tag{101}$$

Para K:

$$T(n) = 2^k T(n/2^k) + n * K \tag{102}$$

Entonces, para este caso... Cual es el valor de K?

$$n/2^k = 1 \tag{103}$$
$$n = 2^k \tag{104}$$
$$k? = n \tag{105}$$
$$k = \log_2 n \tag{106}$$

Entonces decimos que K es igual a $\log_2 n$.

$$T(n) = T(n/2^k) + k \tag{107}$$
$$T(n) = T(n/2^{\log_2 n}) + 3 * \log_2 n \tag{108}$$
$$T(n) = T(1) + 3 * \log_2 n \tag{109}$$
$$T(n) = 1 + T(\log_2 n) \tag{110}$$

Con su maximizacion respectiva, obtenemos que el orden de esta función recursiva es logarítmica $T(\log_2 n)$

7. Analice el algoritmo propuesto del ejercicio anterior de modo que encuentra una versión mejorada. Explique haciendo uso de las reglas asintóticas el orden de complejidad encontrado, así mismo, explique las diferencias con el algoritmo propuesto anteriormente, haciendo énfasis en aquellos aspectos que redundaron en la mejora del algoritmo.

   Hay manera que codificar mas simple, esta nos ahorra algunas lineas de codigo.

MergeSortPeque

```java
public class MergeSortSimple {
    public static void mergeSort(int[] arr) {
        if (arr.length < 2) {
            return;
        }

        int mid = arr.length / 2;
        int[] left = Arrays.copyOfRange(arr, 0, mid);
        int[] right = Arrays.copyOfRange(arr, mid, arr.length);

        mergeSort(left);
        mergeSort(right);
        merge(arr, left, right);
    }

    private static void merge(int[] arr, int[] left, int[] right) {
        int i = 0, j = 0, k = 0;

        while (i < left.length && j < right.length) {
            if (left[i] <= right[j]) {
```

```
                arr[k++] = left[i++];
            } else {
                arr[k++] = right[j++];
            }
        }

        while (i < left.length) {
            arr[k++] = left[i++];
        }

        while (j < right.length) {
            arr[k++] = right[j++];
        }
    }
}
```

Este código en temas de tiempo es un poco mas efectivo, en temas de memoria sigue representanta el mismo peso. Se han probado diferentes algoritmos que se pueden encontrar en el GitHub, sin embargo, no se ha notado una clara mejora, mas que en el tiempo.

Como tal, el producto es complicado de mejorar.

Por otra parte, hay un ordenamiento del algoritmo Sort que esta empleando Hilos, este fue generado por IA.

MergeSort - Generado por IA

```
package Fase1.P3.Ordenamiento;

import java.util.Arrays;

public class Hilos {
    // Array auxiliar global para evitar creaciones repetidas
    private static int[] auxiliar;

    public static void mergeSort(int[] arr) {
        // Inicializar el array auxiliar una sola vez
        auxiliar = new int[arr.length];
        mergeSortInternal(arr, 0, arr.length - 1);
    }

    private static void mergeSortInternal(int[] arr, int izquierda, int
    ↪  derecha) {
        if (izquierda < derecha) {
            int medio = izquierda + (derecha - izquierda) / 2;
```

```
            // Usar un umbral para cambiar a insertion sort en arrays pequeños
            if (derecha - izquierda <= 10) {
                insertionSort(arr, izquierda, derecha);
            } else {
                mergeSortInternal(arr, izquierda, medio);
                mergeSortInternal(arr, medio + 1, derecha);
                merge(arr, izquierda, medio, derecha);
            }
        }
    }

    private static void merge(int[] arr, int izquierda, int medio, int derecha)
    ↪  {
        // Copiar solo la sección necesaria al array auxiliar
        for (int i = izquierda; i <= derecha; i++) {
            auxiliar[i] = arr[i];
        }

        int i = izquierda;
        int j = medio + 1;
        int k = izquierda;

        while (i <= medio && j <= derecha) {
            if (auxiliar[i] <= auxiliar[j]) {
                arr[k++] = auxiliar[i++];
            } else {
                arr[k++] = auxiliar[j++];
            }
        }

        // Solo necesitamos copiar los elementos restantes de la parte izquierda
        // (los de la derecha ya están en su posición)
        while (i <= medio) {
            arr[k++] = auxiliar[i++];
        }
    }

    // Insertion sort para arrays pequeños
    private static void insertionSort(int[] arr, int inicio, int fin) {
        for (int i = inicio + 1; i <= fin; i++) {
            int valorActual = arr[i];
            int j = i - 1;

            while (j >= inicio && arr[j] > valorActual) {
```

```
                arr[j + 1] = arr[j];
                j--;
            }

            arr[j + 1] = valorActual;
        }
    }

    // Versión multihilo del MergeSort
    public static void parallelMergeSort(int[] arr) {
        // Inicializar el array auxiliar una sola vez
        auxiliar = new int[arr.length];

        // Determinar el número de hilos disponibles
        int numThreads = Runtime.getRuntime().availableProcessors();

        // Crear y ejecutar hilos
        Thread[] threads = new Thread[numThreads];
        int segmentSize = arr.length / numThreads;

        for (int i = 0; i < numThreads; i++) {
            final int threadId = i;
            threads[i] = new Thread(() -> {
                int start = threadId * segmentSize;
                int end = (threadId == numThreads - 1) ? arr.length - 1 :
                ↪  (threadId + 1) * segmentSize - 1;
                mergeSortInternal(arr, start, end);
            });
            threads[i].start();
        }

        // Esperar a que todos los hilos terminen
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // Combinar los segmentos ordenados
        for (int size = segmentSize; size < arr.length; size *= 2) {
            for (int left = 0; left < arr.length; left += 2 * size) {
                int mid = Math.min(left + size - 1, arr.length - 1);
```

```
                int right = Math.min(left + 2 * size - 1, arr.length - 1);
                merge(arr, left, mid, right);
            }
        }
    }

    public static void main(String[] args) {
        int size = 100000;
        int[] arr = new int[size];
        int[] arrParallel = new int[size];

        for (int i = 0; i < size; i++) {
            arr[i] = (int) (Math.random() * size);
            arrParallel[i] = arr[i]; // Copiar para comparación justa
        }

        System.out.println("Ordenando array de " + size + " elementos con
        ↪  MergeSort optimizado...");
        long startTime = System.currentTimeMillis();
        mergeSort(arr);
        long endTime = System.currentTimeMillis();
        System.out.println("Tiempo de ejecución MergeSort optimizado: " +
        ↪  (endTime - startTime) + " ms");

        System.out.println("\nOrdenando array de " + size + " elementos con
        ↪  MergeSort paralelo...");
        startTime = System.currentTimeMillis();
        parallelMergeSort(arrParallel);
        endTime = System.currentTimeMillis();
        System.out.println("Tiempo de ejecución MergeSort paralelo: " +
        ↪  (endTime - startTime) + " ms");

        // Verificar que ambos resultados sean iguales
        boolean sonIguales = Arrays.equals(arr, arrParallel);
        System.out.println("\n¿Los resultados son iguales? " + sonIguales);

        System.out.println("\nPrimeros 5 elementos: " +
        ↪  Arrays.toString(Arrays.copyOfRange(arr, 0, Math.min(5,
        ↪  arr.length))));
        System.out.println("Últimos 5 elementos: " +
        ↪  Arrays.toString(Arrays.copyOfRange(arr, Math.max(0, arr.length -
        ↪  5), arr.length)));
    }
```

```
}
```

Las diversas pruebas que se han llevado a cabo, nos indican que en temas de tiempo se logra optimizar con este ultimo algoritmo, para la memoria, se optimiza un poco debido a las variables auxiliares con las que cuenta.

9. Conclusiones y Observaciones

- Se ha logrado comprender las diferentes reglas y métodos para la evaluacion de la eficiencia de los logaritmos, incluyendo las el metodo por expansion como los metodos de recursion.
- Se logro determinar la complejidad de los diferentes algoritmos que van desde O(1) siendo la complejidad simple hasta la complejidad $O(n^n)$.
- Ademas de haber lograda entender la notacion asintotica

A. Anexos

Los codigos fuente de las actividades y ejercicios están en la carpeta: Algoritmos-P3