**Complejidad**

1. **Introducción**

Un algoritmo es un método para resolver un problema, debe presentarse como una secuencia ordenada de instrucciones que siempre se ejecutan en un tiempo finito y con una cantidad de esfuerzo también finito. En un algoritmo siempre debe haber un punto de inicio y uno de terminación, estos deben ser únicos y deben ser fácilmente identificables (*LUDA UAM-Azc.*, 2024).

Todo algoritmo debe cumplir con las siguientes características:

1. Debe ser ***Preciso***; es decir, debe especificar sin ambigüedad el orden en quese deben ejecutar las instrucciones.
2. Debe ser ***Definido***; esto es, que cada vez que se ejecute bajo las mismas condiciones, la secuencia de ejecución deberá ser la misma y entregar los mismos resultados.
3. Debe ser ***Finito***; es decir, que siempre se realizarán un número finito de instrucciones en un tiempo y esfuerzo finitos.
4. **Definición de Complejidad Algorítmica**

Existen muchas alternativas de solución para un problema, debemos seleccionar el algoritmo más eficiente con el mejor conjunto de pasos, que su tiempo de ejecución sea el menor y cuyas líneas de código sean las menos posibles.

A simple vista parece algo muy simple, pero a medida que un programa crece, se requiere una medición más exacta y apropiada, para esto se realizan ciertas operaciones matemáticas que establecen la eficiencia teórica del programa, al estudio de estos casos se denomina ***Complejidad Algorítmica***.

La ***complejidad algorítmica*** representa la cantidad de recursos (temporales y espaciales) que necesita un algoritmo para resolver un problema y por tanto permite determinar la eficiencia de dicho algoritmo, pero son medidas relativas al tamaño del problema.  
Un algoritmo será más eficiente comparado con otro, siempre que consuma menos recursos, como el tiempo y espacio de memoria necesarios para ejecutarlo.  
**1. Complejidad Temporal:** Tiempo de cómputo necesario para ejecutar algún programa.

**2. Complejidad Espacial:** Memoria que utiliza un programa para su ejecución, indica la cantidad de espacio requerido para ejecutar el algoritmo; es decir, el espacio en memoria que ocupan todas las variables propias al algoritmo. Para calcular la memoria estática de un algoritmo se suma la memoria que ocupan las variables declaradas en el algoritmo. Para el caso de la memoria dinámica, el cálculo no es tan simple ya que, este depende de cada ejecución del algoritmo.

Nosotros estudiaremos las complejidades temporales, con este fin, para cada problema determinaremos una medida N, que llamaremos tamaño de la entrada o número de datos a procesar por el programa, intentaremos hallar respuestas en función de dicha N.  
Esto dependerá de la naturaleza del problema, por ejemplo, si hablamos de un arreglo se puede ver a N como el rango del arreglo, para una matriz, el número de elementos que la componen; para un grafo, podría ser el número de nodos o arcos, no se puede establecer una regla para N, pues cada problema tiene su propia complejidad.

La familia ***O(f(n))*** representa un ***Orden de Complejidad*** de donde ***f(n)*** es la función más sencilla de la familia.

Las funciones de complejidad algorítmica más habituales en las cuales el único factor del que dependen es el tamaño de la muestra de entrada ***n***, ordenadas de mayor a menor eficiencia son:

Tabla

Descripción generada automáticamente

**Imagen 1** Tipos de Complejidad algorítmica

A continuación, se mostrará su definición de cada tipo de complejidad

|  |  |
| --- | --- |
| Complejidad | Definición |
| O(1) | Complejidad constante. Las instrucciones se ejecutan una vez. |
| O(log n) | Complejidad logarítmica. Aparece en algoritmos como búsqueda binaria. |
| O(n) | Complejidad lineal. Se encuentra en bucles simples con instrucciones constantes. |
| O(n log n) | Complejidad cuasi-lineal. Presente en algoritmos de tipo divide y vencerás. |
| O(n^2) | Complejidad cuadrática. Se da en bucles doblemente anidados. |
| O(n^3) | Complejidad cúbica. Se da en bucles con triple anidación. |
| O(n^a) | Complejidad polinómica (a > 3). El tiempo de ejecución aumenta con n^a. |
| O(2^n) | Complejidad exponencial. Tiempos de ejecución muy altos, poco prácticos. |

Gráfico, Gráfico de superficie

Descripción generada automáticamente

**Imagen 1** Grafica de comportamiento de ordenes de crecimiento de algoritmos

1. **Reglas elementales para el análisis de algoritmos**

Cuando se dispone de distintos algoritmos para resolver un problema, es necesario definir cuál de ellos es el más eficiente. Una herramienta esencial para este propósito es el análisis de algoritmos.  Este análisis suele efectuarse desde dentro del algoritmo hacia afuera, determinando el tiempo requerido por las instrucciones secuenciales y cómo se combinan estos tiempos con las estructuras de control que enlazan las distintas partes del algoritmo. A continuación, se presentan las reglas básicas para el análisis de algoritmos:

**Operaciones elementales:** el coste computacional de una operación elemental es 1 (asignación, en trata/salida, operación aritmética que no desborde el tamaño de la variable utilizada). La mayoría de las instrucciones de un algoritmo son operaciones elementales que están en el orden de *O*(1).

**Secuencia:** en un fragmento de código, su coste computacional será la suma del coste individual de cada una de sus operaciones o fragmentos de código, aplicando la regla anterior. Es decir, si se cuenta con un fragmento de código compuesto de cinco operaciones elementales, el coste de dicho fragmento será *O*(5).

**Condicional *if-else* (decisión):** si *g* es el coste de la evaluación de un condicional *if*, su valor es de  *O*(1) por cada condición que se deba evaluar (considerando siempre el peor caso, en el que se deba  realizar todas las comparaciones dependiendo del operador lógico usado). A esto se le suma el coste  de la rama de mayor valor, sea esta la verdadera o falsa (considerando las anidadas si corresponde),  aplicando las reglas anteriores.

Es decir el coste de un fragmento condicional es del orden de *O*(*g* + *máximo*(rama verdadera,  rama falsa)).

**Ciclo *for*:** El coste de dicho fragmento se considera como una constante. Y el mismo se considera despreciable o de *O* (n) respecto al número de iteraciones del ciclo.

Es decir el coste total estará en el orden de *O*(n \* *O*(1)) lo que significa que es de *O*(n). Si dentro del *F1* tenemos la presencia de otro ciclo del mismo tipo, el coste total estará en el orden de *O*(n \* n \* *O*(1))  lo que significa que es de *O*(n2).

**Ciclo *while*:** en este tipo de ciclos se aplica la regla anterior, solo se debe tener en cuenta que a veces no se cuenta con una variable de control numérica, sino que depende del tamaño de las estructuras con las que se esté trabajando y su dimensión, o del tipo de actividad que se realice dentro de dicho ciclo.

**Recursividad:** el cálculo del coste total de un algoritmo recursivo no es tan sencillo como los casos que vimos previamente. Para realizar este cálculo existen dos técnicas comúnmente utilizadas:  ecuaciones recurrentes y teoremas maestros. Para realizar la primera de ellas se busca una ecuación que elimine la recursividad para poder obtener el orden de dicha función, muchas veces esto no es una tarea sencilla. En cambio, el segundo utiliza funciones condicionales y condiciones de regularidad para realizar el cálculo del coste. Estas técnicas antes mencionadas son avanzadas y exceden los alcances de este libro por lo que solo se explicarán algunos ejemplos utilizando la técnica de ecuaciones recurrentes para determinar el orden de funciones recurrentes.

**3 ejemplos de complejidad**

Tenemos el siguiente ejemplo donde vamos a analizar la complejidad

**Ejemplo 1:  
Complejidad Lineal**

**void** numeros\_pares\_impares(**int** numero) {

**int** cont\_imp = 0; // O(1)

**for** (**int** i = 1; i <= numero; ++i) { // O(n)

**if** (i % 2 == 0) { // O(1)

std::cout << i << " Es Par" << std::endl; // O(1)

} **else** { // O(1)

std::cout << i << " Es Impar" << std::endl;// O(1)

cont\_imp++; // O(1)

}

}

std::cout << "Cantidad de Números Impares: " << cont\_imp << std::endl; // O(1)

}

Para calcular la complejidad algorítmica de este código, primero analicemos los diferentes bloques de código:

1. El bucle **for** itera desde **1** hasta **numero** inclusive, donde **numero** es el parámetro de entrada.
2. Dentro del bucle, se realiza una verificación condicional **if (i % 2 == 0)** para determinar si **i** es par o impar.
3. En función de la paridad de **i**, se imprime un mensaje correspondiente.
4. Además, se incrementa el contador **cont\_imp** cuando **i** es impar.
5. Finalmente, se imprime la cantidad de números impares encontrados.

Vamos a analizar la complejidad de cada uno de estos bloques:

* El bucle **for** ejecuta **numero** veces, por lo que su complejidad es O(numero).
* La verificación **if (i % 2 == 0)** se realiza en cada iteración del bucle, lo cual es una operación de tiempo constante O(1).
* La impresión de un mensaje es una operación de tiempo constante O(1).
* El incremento de **cont\_imp** es una operación de tiempo constante O(1).

Tenemos:

O(1) + O(n) \* (O(1) + O(1) + O(1) + O(1)) + O(1) = O(n)

Por lo tanto, la complejidad total del programa es O(n), donde n es el número ingresado como parámetro. Esto significa que el tiempo de ejecución del algoritmo crecerá linealmente con el valor de entrada numero.

**Ejemplo 2:**

**Complejidad Cuadrática**

**void** imprimir\_pares(**int** n) {

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

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

std::cout << "(" << i << ", " << j << ") "; // O(1)

}

std::cout << std::endl; // O(1)

}

}

Este código tiene un bucle anidado dentro de otro bucle, lo que resulta en una complejidad cuadrática. Ahora, hagamos el análisis de la complejidad:

* El bucle externo ejecuta n veces, donde n es el valor ingresado por el usuario.
* Dentro de cada iteración del bucle externo, el bucle interno también se ejecuta n veces.
* Por lo tanto, el número total de iteraciones del bucle interno es n \* n, lo que resulta en una complejidad cuadrática.

Tenemos:

O(n) \* (O(n) \* O(1) + O(1)) = O(n^2)

Por lo tanto, la complejidad total del programa es O(n^2), donde n es el valor de entrada. Esto significa que el tiempo de ejecución del algoritmo crecerá cuadráticamente con el valor de entrada n.

**Ejemplo 3:**

**Complejidad algorítmica**

**int** potencia\_logaritmica(**int** base, **int** exponente) { // O(1)

**if** (exponente == 0) // O(1)

**return** 1; // O(1)

**int** mitad\_potencia = potencia\_logaritmica(base, exponente / 2); // O(log n)

**int** resultado = mitad\_potencia \* mitad\_potencia; // O(1)

**if** (exponente % 2 == 1) // O(1)

resultado \*= base; // O(1)

**return** resultado; // O(1)

}

El análisis de complejidad de tu función **potencia\_logaritmica** es diferente al ejemplo del bucle anidado, pero vamos a desglosarlo:

1. La verificación **if (exponente == 0)** y la operación de retorno **return 1;** son operaciones de tiempo constante O(1). Esto se debe a que no importa el valor de **exponente**, estas operaciones siempre toman una cantidad constante de tiempo.
2. La llamada recursiva **int mitad\_potencia = potencia\_logaritmica(base, exponente / 2);** tiene una complejidad de O(log n). En cada llamada recursiva, el exponente se reduce a la mitad, lo que resulta en aproximadamente logarítmicamente muchas llamadas recursivas.
3. La multiplicación **int resultado = mitad\_potencia \* mitad\_potencia;** es una operación de tiempo constante O(1).
4. La verificación **if (exponente % 2 == 1)** y la multiplicación adicional **resultado \*= base;** también son operaciones de tiempo constante O(1).
5. La operación de retorno **return resultado;** es una operación de tiempo constante O(1).

Entonces, podemos resumir la complejidad de esta función como la suma de todas estas operaciones:

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

Por lo tanto, la complejidad total de tu función **potencia\_logaritmica** es O(log n), donde n es el exponente. Esto significa que el tiempo de ejecución del algoritmo aumentará logarítmicamente con el valor del exponente.

**Ejemplo 4**

**Complejidad factorial**

**unsigned** **long** **long** factorial(**unsigned** **int** n) {

**if** (n == 0 || n == 1) // O(1)

**return** 1; // O(1)

**else**

**return** n \* factorial(n - 1); // O(n) \* O(n-1)

}

Ahora, hagamos el análisis de la complejidad:

1. La llamada recursiva factorial(n - 1) se realiza n veces, ya que el algoritmo calcula el factorial de n multiplicando n por el factorial de n-1.
2. Cada llamada recursiva implica una multiplicación, que es una operación de tiempo constante O(1).
3. Por lo tanto, la complejidad total del algoritmo es O(n!), donde n! es el factorial de n.

Por lo tanto, la complejidad factorial significa que el tiempo de ejecución del algoritmo crecerá factorialmente con el valor de entrada n. Esto hace que los algoritmos con complejidad factorial sean muy ineficientes para valores de entrada grandes.

**Ejemplo 5**

**Complejidad cubica**

**void** imprimir\_cubo(**int** n) {

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

**for** (**int** j = 0; j < n; ++j) { // O(n) \* O(n)

**for** (**int** k = 0; k < n; ++k) { // O(n) \* O(n) \* O(n)

std::cout << "(" << i << ", " << j << ", " << k << ") ";

}

std::cout << std::endl; // O(1)

}

std::cout << std::endl; // O(1)

}

}

1. El bucle externo se ejecuta n veces.
2. Dentro de cada iteración del bucle externo, hay un bucle interno que se ejecuta n veces.
3. Dentro de cada iteración del bucle interno, hay otro bucle interno que también se ejecuta n veces.
4. Por lo tanto, el número total de iteraciones del bucle más interno es n \* n \* n = n^3.

La complejidad total del algoritmo es O(n^3), donde n es el número ingresado por el usuario. Esto significa que el tiempo de ejecución del algoritmo aumentará cúbicamente con el valor de entrada n, lo que lo hace menos eficiente para valores grandes de n.

**Ejemplo practico de graficar el tipo de complejidad**

**7. Referencias**

*LUDA UAM-Azc.* (2024). Aniei.org.mx. <http://aniei.org.mx/paginas/uam/CursoAA/curso_aa_01.html>

**8.Anexo**

**Ejemplo 1:**

#include <iostream>

**void imprimir\_pares(int n)** {

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

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

std::cout << "(" << i << ", " << j << ") ";

}

std::cout << std::endl;

}

}

**int main()** {

int numero;

std::cout << "Ingrese un número: ";

std::cin >> numero;

imprimir\_pares(numero);

return 0;

}

**Ejemplo 2**

#include <iostream>

**void imprimir\_pares(int n)** {

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

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

std::cout << "(" << i << ", " << j << ") ";

}

std::cout << std::endl;

}

}

**int main()** {

int numero;

std::cout << "Ingrese un número: ";

std::cin >> numero;

imprimir\_pares(numero);

return 0;

}

**Ejemplo 3**

#include <iostream>

**int** potencia\_logaritmica(**int** base, **int** exponente) {

**if** (exponente == 0)

**return** 1;

**int** mitad\_potencia = potencia\_logaritmica(base, exponente / 2);

**int** resultado = mitad\_potencia \* mitad\_potencia;

**if** (exponente % 2 == 1)

resultado \*= base;

**return** resultado;

}

**int** main() {

**int** base, exponente;

std::cout << "Ingrese la base: ";

std::cin >> base;

std::cout << "Ingrese el exponente: ";

std::cin >> exponente;

**int** resultado = potencia\_logaritmica(base, exponente);

std::cout << "El resultado de " << base << " elevado a " << exponente << " es: " << resultado << std::endl;

**return** 0;

}

**Ejemplo 4**

#include <iostream>

**unsigned** **long** **long** factorial(**unsigned** **int** n) {

**if** (n == 0 || n == 1)

**return** 1;

**else**

**return** n \* factorial(n - 1);

}

**int** main() {

**unsigned** **int** numero;

std::cout << "Ingrese un número: ";

std::cin >> numero;

**unsigned** **long** **long** resultado = factorial(numero);

std::cout << "El factorial de " << numero << " es: " << resultado << std::endl;

**return** 0;

}

**Ejemplo 5**

#include <iostream>

**void** imprimir\_cubo(**int** n) {

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

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

**for** (**int** k = 0; k < n; ++k) {

std::cout << "(" << i << ", " << j << ", " << k << ") ";

}

std::cout << std::endl;

}

std::cout << std::endl;

}

}

**int** main() {

**int** numero;

std::cout << "Ingrese un número: ";

std::cin >> numero;

imprimir\_cubo(numero);

**return** 0;

}

**9.** **Guía de ejercicios prácticos**

A continuación, se plantean una serie de fragmentos de códigos. Habrá que realizar un análisis de la complejidad de cada una de las diferentes funciones.

**Ejercicio 1:**

**int** suma\_arreglo(**const** std::vector<**int**>& arr) {

**int** suma = 0;

**for** (**int** num : arr) {

suma += num;

}

**return** suma;

}

**Ejercicio 2:**

**bool** buscar\_elemento(**const** std::vector<**int**>& lista, **int** elemento) {

**for** (**int** num : lista) {

**if** (num == elemento)

**return** **true**;

}

**return** **false**;

}

**Ejercicio 3:**

**int** fibonacci\_recursivo(**int** n) {

**if** (n <= 1)

**return** n;

**return** fibonacci\_recursivo(n - 1) + fibonacci\_recursivo(n - 2);

}

**Ejercicio 4:**

**void** ordenamiento\_seleccion(std::vector<**int**>& arr) {

**int** n = arr.size();

**for** (**int** i = 0; i < n - 1; ++i) {

**int** min\_idx = i;

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

**if** (arr[j] < arr[min\_idx])

min\_idx = j;

}

std::swap(arr[i], arr[min\_idx]);

}

}

**Ejercicio 5:**

**unsigned** **long** **long** factorial\_iterativo(**unsigned** **int** n) {

**unsigned** **long** **long** resultado = 1;

**for** (**unsigned** **int** i = 2; i <= n; ++i) {

resultado \*= i;

}

**return** resultado;

}