PRÁCTICA 02: NOTACION ASITNOTICA: Algoritmos de búsqueda y conteo en matrices (C++ y Python)

DOCENTE: Mg. Aldo Hernán Zanabria Gálvez

CURSO: Algoritmos y Estructuras de Datos (SIS210)

CICLO: IV - Escuela Profesional de Ingeniería de Sistemas

Objetivo: Analizar el rendimiento de algoritmos utilizando notación Big-O, y comprender su impacto en la escalabilidad computacional.

Marco teórico: El análisis de algoritmos es una disciplina fundamental en la ciencia de la computación, ya que permite evaluar la eficiencia de las soluciones propuestas a problemas computacionales. Entre las herramientas más relevantes para este análisis se encuentra la **notación asintótica**, que permite describir el comportamiento de un algoritmo a medida que crece el tamaño de su entrada (Cormen et al., 2022; Goodrich & Tamassia, 2014).

La notación asintótica no mide el tiempo de ejecución en unidades físicas (como segundos), sino que representa la cantidad de operaciones básicas ejecutadas por un algoritmo en función del tamaño de la entrada, utilizando funciones matemáticas. Esta abstracción permite comparar algoritmos de manera independiente al hardware, al sistema operativo o al lenguaje de programación utilizado (Weiss, 2017).

Las principales notaciones utilizadas son:

- **O(f(n)) (Big-O):** Representa la **cota superior** del crecimiento de un algoritmo. Describe el peor caso, es decir, el máximo tiempo de ejecución posible ante un conjunto de entradas.
- **Ω(f(n)) (Omega):** Representa la **cota inferior**, describiendo el mejor caso posible.
- **Θ(f(n)) (Theta):** Describe la **cota ajustada** o exacta del algoritmo, cuando se conoce tanto el límite superior como el inferior con exactitud (Brassard & Bratley, 1997).

Estas notaciones permiten clasificar los algoritmos en **funciones de crecimiento** de acuerdo con su comportamiento asintótico. Entre las más comunes se encuentran:

Notación	Tipo de crecimiento	Ejemplo de algoritmo
O(1)	Constante	Acceso a un elemento de un arreglo
O(log n)	Logarítmica	Búsqueda binaria
O(n)	Lineal	Recorrido secuencial
O(n log n)	Lineal-logarítmica	Merge Sort, Quick Sort (en promedio)
$O(n^2)$	Cuadrática	Bubble Sort, Selection Sort
$O(2^n)$	Exponencial	Fuerza bruta en problemas combinatorios

Estas clasificaciones permiten prever la **escalabilidad** de los algoritmos, especialmente en contextos donde el volumen de datos es muy grande, como ocurre en el análisis de big data, inteligencia artificial o simulación. Además, el análisis asintótico sirve como criterio para

optimizar soluciones y seleccionar las estructuras de datos más apropiadas según el contexto del problema (Aho, Hopcroft & Ullman, 1988; Sedgewick & Flajolet, 1996).

Práctica programada:

1. Analizar teóricamente la siguiente función:

```
void imprimirPares(int n) {
  for (int i = 0; i < n; i++) {
    if (i % 2 == 0) {
      cout << i << endl;
    }
  }
}
```

   - ¿Cuál es su complejidad y por qué?

2. Comparar dos implementaciones para calcular la suma de los primeros N números naturales:
   - O(n): bucle acumulativo
   - O(1): fórmula directa n*(n+1)/2
   - Implementar ambas en C++ o Python.
   - Medir tiempos de ejecución con valores de n crecientes (1,000 a 1,000,000).
3. Crear una tabla comparativa y graficar los resultados.

Tarea domiciliaria: Investigar y explicar, con ejemplos prácticos, los siguientes tipos de complejidad:

- O(1), O(log n), O(n), O(n log n), O(n^2), O(2^n)
- Para cada caso, proponer un algoritmo real donde se aplique.

Bibliografía:

- Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
- Google. (s.f.). *Tech Dev Guide: Data Structures & Algorithms*. Recuperado de https://techdevguide.withgoogle.com/paths/data-structures-and-algorithms/
- W3Schools. (s.f.). *Data Structures and Algorithms Tutorial*. Recuperado de https://www.w3schools.com/dsa
- Khan Academy. (s.f.). *Algorithms*. Recuperado de https://www.khanacademy.org/computing/computer-science/algorithms

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1988). *Data structures and algorithms*. Addison-Wesley Iberoamericana.
Brassard, G., & Bratley, P. (1997). *Fundamentos de algoritmia*. Prentice Hall.
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press.
Goodrich, M. T., & Tamassia, R. (2014). *Data structures and algorithms in Python*. Wiley.

Sedgewick, R., & Flajolet, P. (1996). *An introduction to the analysis of algorithms*. Addison-Wesley.
Weiss, M. A. (2017). *Data structures and algorithm analysis in C++* (4th ed.). Pearson.

CODIGO FUENTE

CODIGO EN PYTHON

```python
import random
import time

def generar_matriz(filas, columnas):
    return [[random.randint(0, 100) for _ in range(columnas)] for _ in
range(filas)]

def contar_pares(matriz):
    conteo = 0
    for fila in matriz:
        for valor in fila:
            if valor % 2 == 0:
                conteo += 1
    return conteo

filas, columnas = 100, 100
matriz = generar_matriz(filas, columnas)

inicio = time.time()
resultado = contar_pares(matriz)
fin = time.time()

print(f"Números pares: {resultado}")
print(f"Tiempo de ejecución: {fin - inicio:.6f} segundos")
```

CODIGO EN C++

```cpp
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>
using namespace std;

vector<vector<int>> generarMatriz(int filas, int columnas) {
    vector<vector<int>> matriz(filas, vector<int>(columnas));
    for (int i = 0; i < filas; ++i)
        for (int j = 0; j < columnas; ++j)
            matriz[i][j] = rand() % 101;
    return matriz;
}

int contarPares(const vector<vector<int>>& matriz) {
    int conteo = 0;
    for (const auto& fila : matriz)
        for (int val : fila)
            if (val % 2 == 0) ++conteo;
    return conteo;
}

int main() {
    srand(time(0));
    int filas = 100, columnas = 100;
    vector<vector<int>> matriz = generarMatriz(filas, columnas);
```

```
    clock_t inicio = clock();
    int resultado = contarPares(matriz);
    clock_t fin = clock();

    cout << "Números pares: " << resultado << endl;
    cout << "Tiempo de ejecución: " << double(fin - inicio) /
CLOCKS_PER_SEC << " segundos\n";
    return 0;
}
```

Actividades

1. Explica paso a paso ambos algoritmos. ¿Qué estructura de datos se usa?
2. Calcula su complejidad temporal y espacial.
3. Ejecuta los códigos en ambos lenguajes con matrices de tamaño: 50x50, 100x100, 200x200.
4. Compara tiempos de ejecución y justifica las diferencias.
5. Agrega una función que cuente **números primos** en la matriz. Implementa en ambos lenguajes.
6. ¿Cómo optimizarías la verificación de primalidad?

Requisitos de Entrega

1. Informe con:
    - Análisis teórico.
    - Códigos comentados línea por línea.
    - Resultados experimentales (tablas o gráficos).
    - Capturas de ejecución.
2. Video corto explicativo.
3. Códigos subidos en OnlineGDB o GitHub.