Universidad Nacional Del Altiplano

Facultad De Ingeniería Mecánica Eléctrica, Electrónica Y Sistemas

Escuela Profesional De Ingeniería De Sistemas

Practica Nº2.

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

CURSO:

Algoritmos y Estructuras de Datos

DOCENTE:

Mg. Aldo Hernan Zanabria Galvez.

ESTUDIANTE:

Yefferson Miranda Josec

CODIGO: 216984

FECHA: 16/04/2025

SEMESTRE:

IV

Actividades N°2:

Analisis Teorico :

El **tiempo de complejidad** se refiere a la cantidad de pasos que requiere un algoritmo para completarse, y se usa para evaluar su eficiencia. Esta medida, también conocida

como **complejidad computacional**, fue introducida por Juris Hartmanis y Richard E. Stearns en 1965, quienes sentaron las bases de este campo.

La notación utilizada para expresar esta complejidad es la **notación Big O (O())**, que describe el comportamiento del algoritmo en función del tamaño de la entrada. Esta notación también permite representar gráficamente el crecimiento del tiempo de ejecución, lo que ayuda a identificar el tipo de algoritmo usado.

Evaluar la complejidad con distintos tamaños de entrada permite hacer un control de calidad del código. Aunque existan múltiples lenguajes de programación, todos los algoritmos están sujetos a estas reglas de complejidad.

En este análisis, se evaluará la complejidad teórica del algoritmo que **cuenta pares en una matriz**, implementado en Python y C++.

1. Explica paso a paso ambos algoritmos. ¿Qué estructura de datos se usa?

**ESTRUCTURA DE DATO-> LISTA DE LISTAS. C ++, O VECTOR DE VECTORES, CASI LO MISMO

```
//LISTA DE LISTAS
//EN VECTOR TRABAJO CON INDICES,


// Incluye la biblioteca para operaciones de entrada/salida
#include <iostream>
// Incluye la biblioteca para usar el contenedor vector
#include <vector>
// Incluye la biblioteca para funciones de tiempo
#include <ctime>
// Incluye la biblioteca para funciones generales (rand, srand)
#include <cstdlib>

// Usa el espacio de nombres estándar para evitar std::
using namespace std;

// Función que genera una matriz de números aleatorios
vector<vector<int>> generarMatriz(int filas, int columnas) {
    // Crea una matriz bidimensional con las dimensiones especificadas
    vector<vector<int>> matriz(filas, vector<int>(columnas));

    // Recorre cada fila de la matriz
    for (int i = 0; i < filas; ++i)
        // Recorre cada columna de la fila actual
        for (int j = 0; j < columnas; ++j)
            // Asigna un número aleatorio entre 0 y 100 a cada posición
            matriz[i][j] = rand() % 101;

    // Devuelve la matriz generada
    return matriz;
}

// Función que cuenta los números pares en una matriz
int contarPares(const vector<vector<int>>& matriz) {
    // Inicializa el contador de números pares en 0
    int conteo = 0;

    // Recorre cada fila de la matriz (usando referencia constante)
    for (const auto& fila : matriz)
        // Recorre cada valor en la fila actual
        for (int val : fila)
            // Si el valor es par (divisible por 2 sin residuo)
            if (val % 2 == 0)
```

```
                // Incrementa el contador de pares
                ++conteo;

    // Devuelve el total de números pares encontrados
    return conteo;
}

// Función principal del programa
int main() {
    // Establece la semilla para números aleatorios usando el tiempo actual
    srand(time(0));

    // Define las dimensiones de la matriz (100x100)
    int filas = 100, columnas = 100;

    // Genera la matriz llamando a la función generarMatriz
    vector<vector<int>> matriz = generarMatriz(filas, columnas);

    // Registra el tiempo de inicio antes de contar los pares
    clock_t inicio = clock();

    // Cuenta los números pares en la matriz
    int resultado = contarPares(matriz);

    // Registra el tiempo de finalización después de contar
    clock_t fin = clock();

    // Imprime la cantidad de números pares encontrados
    cout << "Números pares: " << resultado << endl;

    // Calcula e imprime el tiempo de ejecución en segundos
    // Convierte la diferencia de ticks a segundos dividiendo por
CLOCKS_PER_SEC
    cout << "Tiempo de ejecución: " << double(fin - inicio) / CLOCKS_PER_SEC
<< " segundos\n";

    // Retorna 0 indicando que el programa terminó correctamente
    return 0;
}
```

**ESTRUCTURA DE DATO-> LISTA DE LISTAS PYTON

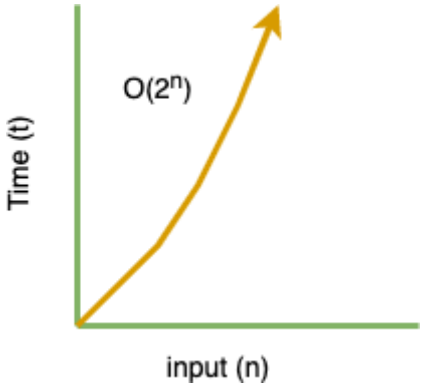
```
# Importar el módulo random para generar números aleatorios
import random
# Importar el módulo time para medir el tiempo de ejecución
import time

# Definir función para crear una matriz de números aleatorios
def generar_matriz(filas, columnas):
    # Retornar una matriz usando comprensión de listas anidadas:
    # - La lista externa crea 'filas' elementos
    # - Cada fila es una lista de 'columnas' números aleatorios entre 0 y
100
    return [[random.randint(0, 100) for _ in range(columnas)] for _ in
range(filas)]

# Definir función para contar números pares en una matriz
def contar_pares(matriz):
    # Inicializar contador de números pares en 0
    conteo = 0

    # Recorrer cada fila de la matriz
    for fila in matriz:
```

```
        # Recorrer cada elemento (valor) dentro de la fila
        for valor in fila:
            # Verificar si el valor es par (divisible por 2 sin residuo)
            if valor % 2 == 0:
                # Incrementar el contador si es par
                conteo += 1
    # Retornar el total de números pares encontrados
    return conteo

# Definir tamaño de la matriz (100 filas x 100 columnas)
filas, columnas = 100, 100

# Generar la matriz llamando a la función generar_matriz
matriz = generar_matriz(filas, columnas)

# Registrar el tiempo de inicio antes de contar los pares
inicio = time.time()

# Llamar a la función para contar los números pares en la matriz
resultado = contar_pares(matriz)

# Registrar el tiempo de finalización después de contar los pares
fin = time.time()

# Mostrar el resultado del conteo de números pares
print(f"Números pares: {resultado}")

# Mostrar el tiempo transcurrido (diferencia entre fin e inicio)
# Formateado a 6 decimales para mayor precisión
print(f"Tiempo de ejecución: {fin - inicio:.6f} segundos")
```

2. Calcula su complejidad temporal y espacial.

La complejidad temporal ya ha sido definida anteriormente, por ello.

Como se ha mencionado anteriormente, el tiempo de complejidad exponencial se basa en algoritmos que, su tiempo para resolver el problema crece exponencialmente según el tamaño de entrada, siendo en base 2, por ello, el tiempo crece exponencialmente según su entrada.



Su tipo de crecimiento es exponencial, denotado por la siguiente expresión:

$$O(2^n)$$

Por qué se define las matrices como tiempo exponencial, tenemos que, primero recorrer todos los datos de esa fila, luego, saltar a la siguiente fila, revisar cada dato en la fila, y así hasta acabar con la matriz.

3. Ejecuta los códigos en ambos lenguajes con matrices de tamaño: 50x50, 100x100, 200x200.

Prueba de 50x50	PYTHON	C++
TIEMPO	0.000998seg	3.7e-05seg

Prueba de 100x100	PYTHON	C++
TIEMPO	0.000999seg	0.000166seg

Prueba de 200x200	PYTHON	C++
TIEMPO	0.002998seg	0.000759seg

4. Compara tiempos de ejecución y justifica las diferencias.

1. Tiempo de ejecución y explicación

Prueba de 50x50	PYTHON	C++
TIEMPO	0.000998seg	3.7e-05seg
Prueba de 100x100	PYTHON	C++
TIEMPO	0.000999seg	0.000166seg
Prueba de 200x200	PYTHON	C++
TIEMPO	0.002998seg	0.000759seg

5. Agrega una función que cuente números primos en la matriz. Implementa en ambos lenguajes.

**PYTON

```python
# Definición de función para verificar si un número es primo
def primal_check(e_value):
    # Por defecto asumimos que el número es primo
    truth = True

    # Un número menor que 2 no es primo
    if e_value < 2:
        return False

    # Verificar divisores desde 2 hasta la raíz cuadrada del número
    for n in range(2, int(e_value ** 0.5) + 1):
        if e_value % n == 0:
            # Si tiene algún divisor (distinto de 1 y de sí mismo), no es primo
            truth = False
            break

    return truth

# Función para contar números primos en una matriz
def count_primal(matriz):
    conteo = 0  # Inicializar el contador de primos

    # Recorrer cada fila de la matriz
    for fila in matriz:
        # Recorrer cada valor en la fila
        for valor in fila:
            # Verificar si el número es primo
            if primal_check(valor):
                conteo += 1  # Incrementar el contador si es primo

    return conteo  # Retornar el total de primos encontrados
```

**C++

```cpp
// Función que verifica si un número es primo
bool primal_check(int e_num) {
    // Por defecto, asumimos que el número es primo
    bool truth = true;

    // Un número menor que 2 no es primo
    if (e_num < 2)
        return false;

    // Verificar divisores desde 2 hasta la raíz cuadrada del número
    for (int n = 2; n * n <= e_num; ++n) {
        // Si es divisible por algún número distinto de 1 y de sí mismo, no es primo
        if (e_num % n == 0) {
            truth = false;
```

```
            break;
        }
    }

    return truth;
}

// Función para contar números primos en una matriz
int contarprimos(const vector<vector<int>>& matriz) {
    int conteo = 0;  // Inicializar contador

    // Recorrer cada fila de la matriz
    for (const auto& fila : matriz) {
        // Recorrer cada valor en la fila
        for (int val : fila) {
            // Verificar si es primo
            if (primal_check(val))
                ++conteo;  // Incrementar si es primo
        }
    }

    return conteo;  // Retornar el total de primos
```

6. ¿Cómo optimizarías la verificación de primalidad?

La verificación de si un número es primo puede optimizarse utilizando dos observaciones clave:

1. **Solo verificar divisores hasta la raíz cuadrada del número.**
   Si un número tiene un divisor mayor que su raíz cuadrada, el otro divisor debe ser menor, por lo tanto, basta con revisar hasta $\sqrt{n}$.
2. **Excluir los pares, salvo el número 2.**
   Todos los números pares mayores a 2 no son primos. Esto reduce las iteraciones a casi la mitad.
3. **Criba de Eratóstenes** (cuando se requiere verificar muchos primos a la vez):
   Es más eficiente para generar una lista de primos, no para verificar uno solo.

PYTON

```
4. from math import isqrt  # isqrt evita usar float y es más rápido
5.
6. def es_primo(val):
7.     if val < 2:
8.         return False
9.     if val == 2:
10.          return True
11.      if val % 2 == 0:
12.          return False
13.
14.      # Verifica solo números impares hasta la raíz cuadrada
15.      for i in range(3, isqrt(val) + 1, 2):
16.          if val % i == 0:
```

```
            return False
    return True
```

C++

```cpp
#include <cmath>

bool es_primo(int val) {
    if (val < 2)
        return false;
    if (val == 2)
        return true;
    if (val % 2 == 0)
        return false;

    // Verificar divisores impares hasta la raíz cuadrada de val
    for (int i = 3; i <= sqrt(val); i += 2) {
        if (val % i == 0)
            return false;
    }
    return true;
}
```

7. Código GitHub.

C++ y Python: https://github.com/yefferson12355/Algoritmos-y-Estructuras-de-Datos/tree/master/Actividades%20N%C2%B02