UNIVERSIDAD NACIONAL DE MOQUEGUA

Facultad de Ingeniería y Arquitectura

Escuela Profesional de Ingeniería de Sistemas e Informática

Análisis y Desarrollo de Algoritmos para Matrices: Implementación y Complejidad Computacional

Informe laboratorio N° 2

Estudiante:
Gersael Mathias Rojas Quijano

Profesores:
Honorio Apaza Alanoca

8 de diciembre de 2024

Índice

1. Introducción

1.1. Motivación y Contexto

El manejo de matrices es fundamental en múltiples disciplinas, desde la computación hasta la física y la economía. Este trabajo tiene como objetivo documentar y analizar seis ejercicios de matrices, proporcionando tanto la implementación de los algoritmos como el análisis de su complejidad computacional.

1.2. Objetivo general

Desarrollar una documentación exhaustiva que explique los ejercicios de matrices y analice la eficiencia de los algoritmos asociados, utilizando la notación Big O.

1.3. Objetivos específicos

- Implementar seis algoritmos que resuelvan problemas con matrices.
- Explicar detalladamente el razonamiento detrás de cada algoritmo.
- Analizar la complejidad computacional (tiempo y espacio) de cada solución.
- Comparar los resultados obtenidos y proponer mejoras.

1.4. Justificación

El análisis de la eficiencia de los algoritmos es crucial para optimizar recursos en aplicaciones reales. Este trabajo busca fortalecer las habilidades de diseño y análisis algorítmico, fundamentales en el desarrollo de software.

2. Marco teórico

2.1. Antecedentes

Las matrices, también conocidas como arreglos bidimensionales, son una estructura básica de datos utilizada para organizar información en forma tabular. En programación, las matrices son una de las herramientas más versátiles debido a su simplicidad y capacidad para modelar datos complejos.
El uso de matrices se remonta a la programación primitiva, donde servían para representar gráficos, resolver sistemas de ecuaciones, y realizar simulaciones numéricas. En la actualidad, son esenciales en áreas como:

- Análisis numérico: Solución de ecuaciones lineales, interpolación, etc.
- Inteligencia artificial: Representación de datos en redes neuronales.
- Gráficos computacionales: Transformaciones y modelado de imágenes.
- Bases de datos: Organización de datos relacionales.

2.2. Marco conceptual

2.2.1. Tema 1: Estructuras de Datos

Una matriz es una colección ordenada de datos distribuidos en filas y columnas. Cada elemento de una matriz se accede mediante un par de índices (i,j) donde i es el índice de la fila y j el de la columna.

Tipos comunes de matrices en programación:

- **Matrices estáticas:** Tienen un tamaño fijo definido en tiempo de compilación.
  - *Ventaja:* Menor consumo de memoria debido a la asignación directa.
  - *Desventaja:* Carecen de flexibilidad para crecer o reducirse.
- **Matrices dinámicas:** Pueden cambiar de tamaño durante la ejecución.
  - *Ventaja:* Adaptabilidad a entradas variables.
  - *Desventaja:* Gestión más compleja de la memoria.

Operaciones comunes en matrices:

- **Inicialización:** Asignación de valores por defecto o entrada del usuario.
- **Acceso:** Recuperar o modificar un elemento específico.
- **Recorrido:** Iteración a través de filas y columnas para procesar los datos.
- **Transformación:** Operaciones como transposición, rotación, etc.

2.2.2. Tema 2: Complejidad Computacional

La notación **Big O** es una herramienta matemática utilizada para describir el rendimiento de los algoritmos en función del tamaño de su entrada. Proporciona una forma de analizar:

- **Complejidad temporal:** Cuánto tiempo toma un algoritmo en ejecutarse.
- **Complejidad espacial:** Cuánta memoria utiliza un algoritmo durante su ejecución.

Ejemplo de análisis en matrices:

- Un algoritmo que recorre todos los elementos de una matriz de tamaño $m \times n$ tiene una complejidad temporal de $\mathcal{O}(m \cdot n)$, ya que debe procesar m filas y n columnas.
- Si el algoritmo utiliza una matriz adicional para almacenar resultados, la complejidad espacial también será $\mathcal{O}(m \cdot n)$.

Clasificaciones comunes de complejidad temporal:

- **Constante $\mathcal{O}(1)$:** El tiempo de ejecución no depende del tamaño de la entrada. Ejemplo: Acceso a un elemento específico de una matriz.
- **Lineal $\mathcal{O}(n)$:** El tiempo de ejecución aumenta proporcionalmente al tamaño de la entrada. Ejemplo: Sumar todos los elementos de una fila.
- **Cuadrática $\mathcal{O}(n^2)$:** El tiempo de ejecución aumenta proporcionalmente al cuadrado del tamaño de la entrada. Ejemplo: Calcular el producto de dos matrices cuadradas.
- **Logarítmica $\mathcal{O}(\log n)$:** El tiempo de ejecución crece de manera logarítmica. Ejemplo: Algoritmos de búsqueda en matrices ordenadas utilizando divisiones binarias.

Relación entre matrices y complejidad: El uso eficiente de matrices depende de una correcta implementación y análisis de su complejidad. Por ejemplo:

- **Optimización de memoria:** Usar matrices dispersas si la mayoría de los elementos son ceros.
- **Paralelización:** Dividir operaciones en subprocesos para matrices grandes.

3. Metodología

La metodología describe el enfoque seguido para desarrollar y analizar los ejercicios de matrices, incluyendo la implementación, ejecución y evaluación de cada solución, con énfasis en el análisis de complejidad computacional.

3.1. Enfoque Metodológico

El desarrollo de los ejercicios sigue un enfoque práctico basado en programación y análisis teórico, estructurado en los siguientes pasos:

1. **Definición de ejercicios:** Utilizamos seis problemas representativos del uso de matrices en programación.
2. **Implementación de soluciones:** Codificación de cada ejercicio en el lenguaje de programación Java.
3. **Ejecución y pruebas:** Pruebas de funcionalidad utilizando diferentes tamaños y configuraciones de matrices para validar las soluciones.
4. **Análisis de complejidad:** Estimación teórica de la complejidad temporal y espacial de cada solución mediante notación Big O.
5. **Documentación:** Registro detallado de los algoritmos, pruebas y análisis en un informe estructurado en LaTeX.

3.2. Proceso de Resolución

3.2.1. Paso 1: Análisis del ejercicio

Para cada ejercicio, se analiza los aspectos a utilizar:

- **Operaciones básicas:** Suma, multiplicación.
- **Transformaciones:** Transposición, rotaciones.
- **Aplicaciones prácticas:** Búsqueda de elementos, caminos mínimos en grafos.

3.2.2. Paso 2: Implementación del algoritmo

Cada ejercicio se implementa utilizando un lenguaje de programación estructurado Java. La implementación sigue un flujo general:

1. Inicialización de la matriz con datos de entrada(dimension de la matriz).
2. Desarrollo del algoritmo para resolver el problema.
3. Validación de los resultados mediante casos de prueba predefinidos.

Ejemplo en pseudocódigo:

```
Inicio:
1. Leer tamaño de la matriz (d).
2. Inicializar matriz con valores random.
3. Ejecutar el algoritmo correspondiente.
4. Mostrar el resultado.
Fin.
```

3.2.3. Paso 3: Análisis de complejidad

Se analiza la complejidad temporal y espacial de cada algoritmo.

- **Complejidad temporal:** Determinada por el número de operaciones necesarias para procesar la matriz.
- **Complejidad espacial:** Determinada por el uso de memoria adicional.

Ejemplo: Para calcular la suma de los elementos de una matriz de tamaño $m \times n$:

- **Complejidad temporal:** $\mathcal{O}(m \cdot n)$.
- **Complejidad espacial:** $\mathcal{O}(1)$ (sin uso de estructuras auxiliares).

3.3. Instrumentos y Herramientas

- **Lenguajes de programación:** Python para graficación y Java para codificación.
- **Entornos de desarrollo:** VSCode, NetBeans o Jupyter Notebook.

3.4. Resultados Esperados

Al finalizar el desarrollo de cada ejercicio, se espera obtener:

- Algoritmos funcionales que resuelvan los problemas propuestos.
- Un análisis claro de la eficiencia de cada solución en términos de tiempo y espacio.
- Una documentación bien estructurada que permita replicar los resultados.

4. Propuesta

4.1. Ejercicio 1: Suma de las diagonales de una matriz

4.1.1. Análisis del ejercicio

El objetivo es calcular la suma de los elementos de las diagonales principal y secundaria de una matriz cuadrada. La diagonal principal se extiende desde la esquina superior izquierda a la inferior derecha, mientras que la diagonal secundaria va desde la esquina superior derecha a la inferior izquierda.

4.1.2. Implementación del algoritmo

```
public static void sumardiagonal(int [][] matriz,int d){
        int sum=0,sum2=0;
        for(int i=0;i<d;i++){
            sum+=matriz[i][i];
            sum2+=matriz[i][d-1-i];
        }
        System.out.println("La suma de la diagonal principal: "+sum);
        System.out.println("La suma de la diagonal secundaria: "+sum2);
}
```

4.1.3. Análisis de complejidad

- **Complejidad temporal:** $\mathbf{O(n)}$, donde n es la dimensión de la matriz.
- **Complejidad espacial:** $\mathbf{O(1)}$, ya que solo se usan variables auxiliares.

4.2. Ejercicio 2: Rotación de una matriz 90° en sentido horario

4.2.1. Análisis del ejercicio

Este ejercicio implica rotar una matriz cuadrada n×n 90° en sentido horario. Cada elemento de la matriz debe cambiar de posición de manera que la fila se convierta en la columna, pero en orden inverso.

4.2.2. Implementación del algoritmo

```java
public static int [][] RotarMatriz(int [][] matriz,int d){
        int [][]matriz2=new int[d][d];
        for(int i=0;i<d;i++){
            for(int j=0;j<d;j++){
                matriz2[j][d-i-1]=matriz[i][j];
            }
        }
        return matriz2;
}
```

4.2.3. Análisis de complejidad

- **Complejidad temporal:** $\mathbf{O(n^2)}$, ya que cada elemento de la matriz se procesa una vez.
- **Complejidad espacial:** $\mathbf{O(n^2)}$, debido a la creación de una nueva matriz de salida.

4.3. Ejercicio 3: Suma de los elementos en el perímetro de la matriz

4.3.1. Análisis del ejercicio

El objetivo es calcular la suma de los elementos que forman el perímetro de una matriz cuadrada n×n. Los elementos de la primera fila, la última fila, la primera columna y la última columna deben sumarse, excluyendo las esquinas que se suman dos veces.

4.3.2. Implementación del algoritmo

```java
public static int sumarperimetro(int[][]matriz,int d){
        if(d==1) return matriz[0][0];
        int sum=0;
        for(int i=0;i<d;i++){
            sum+=matriz[0][i]+matriz[d-1][i]+matriz[i][0]+matriz[i][d
    -1];
        }
        sum-=matriz[0][0]+matriz[0][d-1]+matriz[d-1][0]+matriz[d-1][d
    -1];
        return sum;
}
```

4.3.3. Análisis de complejidad

- **Complejidad temporal:** $\mathbf{O(n)}$, ya que se recorren las filas y columnas de la matriz.
- **Complejidad espacial:** $\mathbf{O(n)}$, se utilizan solo variables auxiliares.

4.4. Ejercicio 4: Transpuesta de una matriz

4.4.1. Análisis del ejercicio

El objetivo es generar la matriz transpuesta, que es una matriz donde las filas se convierten en columnas y las columnas en filas.

4.4.2. Implementación del algoritmo

```
public static int [][] transpuestamatriz(int[][] matriz){
        int [][] transpuesta=new int [matriz[0].length][matriz.length];
        for(int i=0;i<matriz.length;i++){
            for(int j=0;j<matriz[0].length;j++){
                transpuesta[j][i]=matriz[i][j];
            }
        }
        return transpuesta;
}
```

4.4.3. Análisis de complejidad

- **Complejidad temporal: O(nxm)**, donde n es el número de filas y m el número de columnas.
- **Complejidad espacial: O(nxm)**, se necesita una nueva matriz para la transpuesta.

4.5. Ejercicio 5: Verificar simetría de una matriz

4.5.1. Análisis del ejercicio

El objetivo es verificar si una matriz cuadrada es simétrica, es decir, si M[i][j]=M[j][i] para todos los i y j.

4.5.2. Implementación del algoritmo

```
public static boolean verificarsimetria(int [][] matriz,int d){
        for(int i=0;i<d;i++){
            for(int j=i+1;j<d;j++){
                if(matriz[i][j]!=matriz[j][i]) return false;
            }
        }
        return true;
}
```

4.5.3. Análisis de complejidad

- **Complejidad temporal:** $\mathbf{O(n^2)}$, se comparan todos los elementos.
- **Complejidad espacial:** $\mathbf{O(1)}$, solo se usan variables auxiliares.

4.6. Ejercicio 6: Recorrer una matriz en espiral

4.6.1. Análisis del ejercicio

El objetivo es recorrer una matriz de manera espiral, desde el borde exterior hacia el centro.

4.6.2. Implementación del algoritmo

```java
public static List<Integer> recorrerMatrizEnEspiral(int[][] matriz) {
        List<Integer> resultado = new ArrayList<>();

        if (matriz==null || matriz.length==0) {
            return resultado;
        }

        int finicio=0,ffin=matriz.length-1;
        int cinicio=0,cfin=matriz[0].length-1;

        while (finicio<=ffin && cinicio<=cfin) {
            for (int i=cinicio;i<=cfin;i++) {
                resultado.add(matriz[finicio][i]);
            }
            finicio++;
            for (int i=finicio;i<=ffin;i++) {
                resultado.add(matriz[i][cfin]);
            }
            cfin--;
            if (finicio<=ffin) {
                for (int i=cfin;i>=cinicio;i--) {
                    resultado.add(matriz[ffin][i]);
                }
                ffin--;
            }
            if (cinicio<=cfin) {
                for (int i=ffin;i>=finicio;i--) {
                    resultado.add(matriz[i][cinicio]);
                }
                cinicio++;
            }
        }
        return resultado;
    }
```

4.6.3. Análisis de complejidad

- **Complejidad temporal: $O(n^2)$**, ya que cada elemento se visita una vez.
- **Complejidad espacial: $O(n^2)$**, para almacenar la secuencia de elementos.

5. Resultados

5.1. Resultado del Ejercicio 1: Suma de la diagonal principal y secundaria

5.1.1. Validación mediante prueba

- **Caso de prueba:** Matriz 3x3:

$$\begin{bmatrix} -47 & -14 & 44 \\ -20 & 99 & 66 \\ -75 & -24 & -18 \end{bmatrix}$$

$$\text{Suma de la diagonal principal} = (-47) + 99 + (-18) = 34$$

$$\text{Suma de la diagonal secundaria} = 44 + 99 + (-75) = 68$$

5.1.2. Optimización

No es necesaria una optimización adicional, ya que la implementación es eficiente para el tamaño de matriz habitual.

5.2. Resultado del Ejercicio 2: Rotación de una matriz 90°

5.2.1. Validación mediante prueba

Ingrese la dimensión de la matriz: 3
Matriz Original:

$$\begin{bmatrix} 97 & 28 & -51 \\ -9 & 73 & -1 \\ 3 & 71 & 41 \end{bmatrix}$$

Matriz rotada:

$$\begin{bmatrix} 3 & -9 & 97 \\ 71 & 73 & 28 \\ 41 & -1 & -51 \end{bmatrix}$$

5.2.2. Optimización

Para una rotación in-place que minimice el uso de memoria, se peude utilizar un algoritmo de rotación por capas.

5.3. Resultado del Ejercicio 3: Perímetro de la matriz

5.3.1. Validación mediante prueba

- **Caso de prueba:** Matriz 3x3:

$$\begin{bmatrix} 49 & 60 & 32 \\ 80 & -50 & 26 \\ 6 & 37 & 41 \end{bmatrix}$$

La suma del perímetro es: 331

5.3.2. Optimización

No es necesaria una optimización adicional, ya que la solución es eficiente.

5.4. Resultado del Ejercicio 4: Transpuesta de una matriz

5.4.1. Validación mediante prueba

Ingrese la cantidad de filas: 3
Ingrese la cantidad de columnas: 2
Matriz Original:

$$\begin{bmatrix} -98 & -48 \\ 25 & 41 \\ 44 & -97 \end{bmatrix}$$

Matriz Transpuesta:

$$\begin{bmatrix} -98 & 25 & 44 \\ -48 & 41 & -97 \end{bmatrix}$$

5.4.2. Optimización

Si la matriz es cuadrada, puedes realizar la transposición in-place.

5.5. Resultado del Ejercicio 5: Verificar simetría de una matriz

5.5.1. Validación mediante prueba

Ingrese la dimensión de la matriz: 3
Matriz:

$$\begin{bmatrix} -87 & 77 & 23 \\ -37 & -74 & -72 \\ 4 & 19 & 95 \end{bmatrix}$$

Resultado: No es simétrica

5.5.2. Optimización

No es necesaria una optimización adicional, ya que la solución es eficiente.

5.6. Resultado del Ejercicio 6: Verificar simetría de una matriz

5.6.1. Validación mediante prueba

Matriz:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{bmatrix}$$

Espiral: [1, 2, 3, 4, 5, 10, 15, 20, 19, 18, 17, 16, 11, 6, 7, 8, 9, 14, 13, 12]

5.6.2. Optimización

No es necesaria una optimización adicional para la mayoría de los casos prácticos.

6. Conclusiones

1. **Coclusion numero uno : Confirmación de la efectividad de los algoritmos** Todos los algoritmos implementados han demostrado ser efectivos y eficientes para las matrices de prueba proporcionadas. Los ejercicios de suma de diagonales, rotación de matrices, y transposición, entre otros, han sido validados exitosamente con matrices de tamaño 3x3, y sus resultados coinciden con los cálculos manuales.

2. **Coclusion numero dos : Necesidad de optimización en algunos casos** Aunque se mencionó que no es necesaria una optimización adicional, algunos algoritmos como la transposición de matrices podrían beneficiarse de una implementación inplace en matrices cuadradas. Esta optimización podría ayudar a reducir el uso de espacio, un aspecto relevante en matrices grandes.

3. **Coclusion numero tres : Validación de la complejidad de los algoritmos** La complejidad de los algoritmos se mantiene dentro de los estándares esperados para operaciones en matrices. La mayoría de los algoritmos tienen una complejidad temporal de $\mathbf{O(n^2)}$, lo cual es esperado cuando se recorre cada elemento de la matriz. La complejidad espacial, en general, es $\mathbf{O(1)}$ o $\mathbf{O(n^2}$, dependiendo de si se requiere espacio adicional para almacenar los resultados, lo que está bien justificado.

Repositorio de Github