Secuencia creciente dentro de una matriz cuadrada

María José Niño Rodríguez[1] David Santiago Quintana Echavarria[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
{ma.nino, quintanae-david}@javeriana.edu.co

29 de septiembre de 2022

Resumen

En este documento se presenta la formalización del problema de la secuencia creciente más larga dentro de una matriz cuadrada implementando la estrategia de programación dinámica. **Palabras clave:** programación dinámica, matrices, secuencia, algoritmo, optimización, maximizar.

Índice

1. Introducción **1**

2. Formalización del problema **1**
2.1. Definición del problema de "Secuencia creciente dentro de una matriz cuadrada" 1

3. Algoritmo de solución **2**
3.1. Algoritmo de programación dinámica 2

4. Experimentación del algoritmo **2**

1. Introducción

Los algoritmos de programación dinámica nos ayudan a optimizar procesos, evita el recalculo de operaciones ya calculadas previamente, evita la concurrencia y se implementa en problemas comúnmente de optimización, ya sea maximizando o minimizando. En este documento se expone un problema que busca maximizar el tamaño de una secuencia dentro de una matriz cuadrada, con el objetivo de mostrar: la formalización del problema (sección 2), la escritura formal de este algoritmo (sección 3).

2. Formalización del problema

A partir de una matriz cuadrada natural ($\mathbb{N}$) de tamaño NxN que contiene los números únicos en el rango $[1, NxN]$ que no tiene ningún orden, se quiere obtener la secuencia más larga de vecinos que tengan la relación de orden parcial $a < b$ y que además tengan una diferencia de 1.

2.1. Definición del problema de "Secuencia creciente dentro de una matriz cuadrada"

Así, el problema se define a partir de:

1. Una matriz cuadrada natural ($\mathbb{N}$) de tamaño NxN

para producir una nueva secuencia de números ordenados.

- Entradas:
  - Una matriz $Mde \in \mathbb{N}$ de NxN.
- Salidas:
  - Una secuencia $s' \in \mathbb{N}$ que tengan una relación de orden parcial $a < b$ con una diferencia de 1 entre a y b.

3. Algoritmo de solución

3.1. Algoritmo de programación dinámica

- Formula de Recurrencia:

$$BC(i,j) = \begin{cases} 0; i < 0 \vee i >= n \vee j < 0 \vee j >= n \\ \\ max(1 + BC(i, j+1), \\ (1 + BC(i, j-1), \\ (1 + BC(i-1, j), \\ (1 + BC(i+1, j)) \end{cases}$$

Algoritmo 1 Secuencia creciente dentro de una matriz cuadrada

```
1:  procedure SecuenciaLarga(m, n)
2:      maxsecuencia ← 1
3:      memo : [1...|m| + 1] × [1...|n| + 1] ← −1
4:      tabIndices : [1...|m| + 1] × [1...|n| + 1] ← [0, 0]
5:      for i ← 1 to n do
6:          for j ← 1 to n do
7:              if memo[i][j] = -1 then
8:                  aux ← buscCamino(i, j, m, memo, tabIndices, n)
9:              end if
10:             if memo[i][j] ¿maxsecuencia then
11:                 maxsecuencia ← memo[i][j]
12:                 maxIndiceX ← i
13:                 maxIndiceY ← j
14:             end if
15:         end for
16:     end for
17:     while maxIndiceX != 0  maxIndiceY != 0 do
18:         aux ← maxInidicesX
19:         maxIndiceX ← maxIndiceX + tabIndices[maxInidicesX, maxIndicesY].x
20:         maxIndiceY ← maxIndiceY + tabIndices[aux, maxIndicesY].y
21:
```

4. Experimentación del algoritmo

Se valido la solución del problema con matrices cuya solución ya era conocida, se comparaba la salida del programa con esta respuesta, ambos resultados fueron iguales. Se experimentaron con matrices variando el tamaño, desde $10x10$, hasta $100x100$, en saltos de 10. Cada matriz en el experimento en un principio fue llenada en orden con números desde 1 hasta n, seguido se intercambiaban posiciones de manera aleatoria,

Algoritmo 2 Caso Base

```
1: procedure BuscCamino(i, j, m, memo, tabIndices, n)
2:     if i ¡0 — i ¿= n — j ¡0 — j ¿= n then
3:         return 0
4:     end if
5:     if memo[i][j] != -1 then
6:         memo[i][j];
7:     end if
8:     der ← −1
9:     izq ← −1
10:    arriba ← −1
11:    abajo ← −1
12:    if !(j + 1 = n) then
13:        if ((m[i][j] + 1) = m[i][j + 1]) then
14:            tabIndices[i][j].x ← 1
15:            tabIndices[i][j].y ← 0
16:            der ← 1 + buscCamino(i, j + 1, m, memo, tabInidces, n)
17:        end if
18:    end if
19:    if !(j - 1 = -1) then
20:        if ((m[i][j] + 1) = m[i][j - 1]) then
21:            tabIndices[i][j].x ← −1
22:            tabIndices[i][j].y ← 0
23:            izq ← 1 + buscCamino(i, j − 1, m, memo, tabInidcesn)
24:        end if
25:    end if
26:    if !(i - 1 = -1) then
27:        if ((m[i][j] + 1) = m[i - 1][j]) then
28:            tabIndices[i][j].x ← 0
29:            tabIndices[i][j].y ← −1
30:            arriba ← 1 + buscCamino(i − 1, j, m, memo, tabInidces, n)
31:        end if
32:    end if
33:    if !(i + 1 = n) then
34:        if ((m[i][j] + 1) = m[i + 1][j]) then
35:            tabIndices[i][j].x ← 0
36:            tabIndices[i][j].y ← 1
37:            abajo ← 1 + buscCamino(i + 1, j, m, memo, tabInidces, n)
38:        end if
39:    end if
40: end procedure=0
```

$n*4$ veces. Con la matriz de prueba generada se llamaba a la función para generar la secuencia, analizando las matrices original, memoización y backtracking en cada caso los resultados generados por el algoritmo fueron los mismos que se esperaban, esto en cada caso genero correctamente la subsecuencia como se evidencia en la siguiente figura.

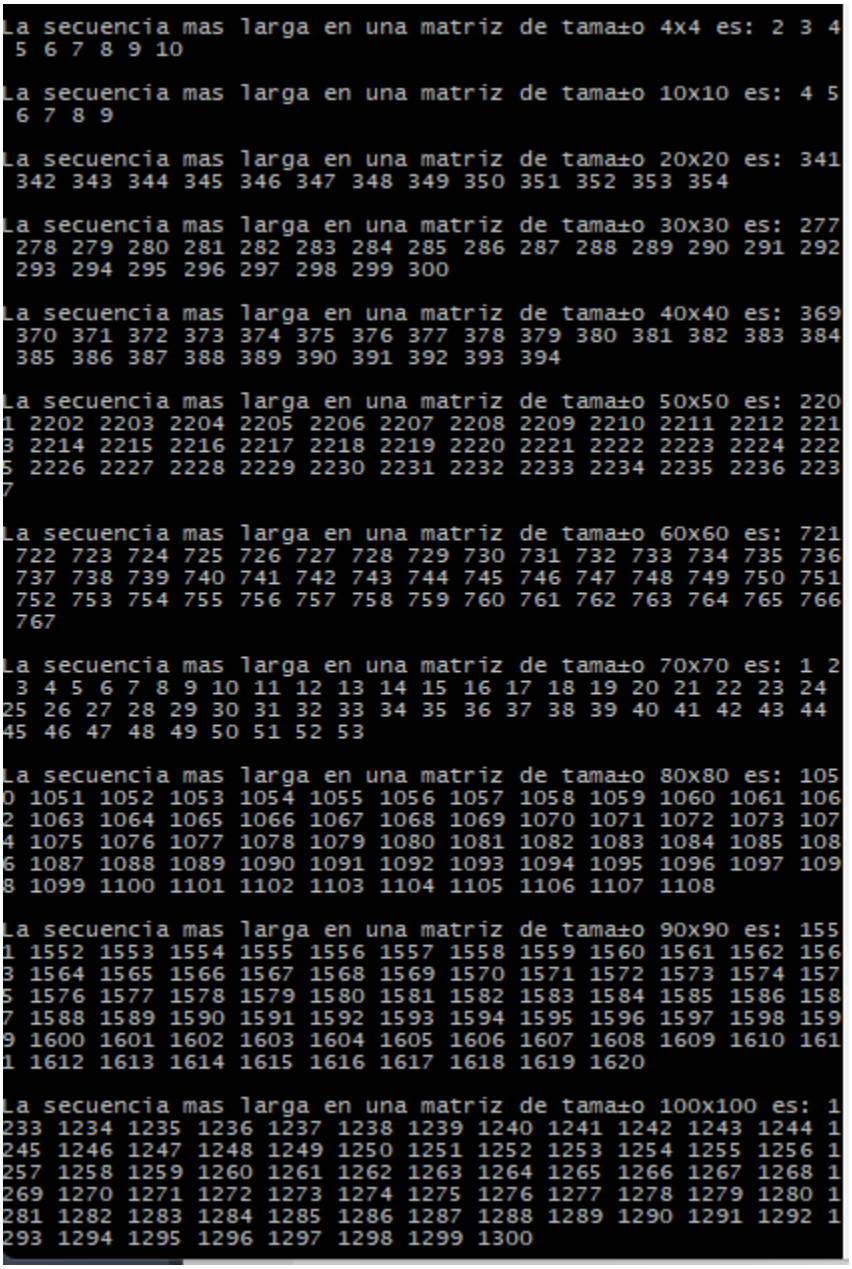

Figura 1: Resultados del experimento con matrices de tamaño 10x10 hasta 100x100