SAET 2024 - Maratona de Programação

29 de novembro de 2024

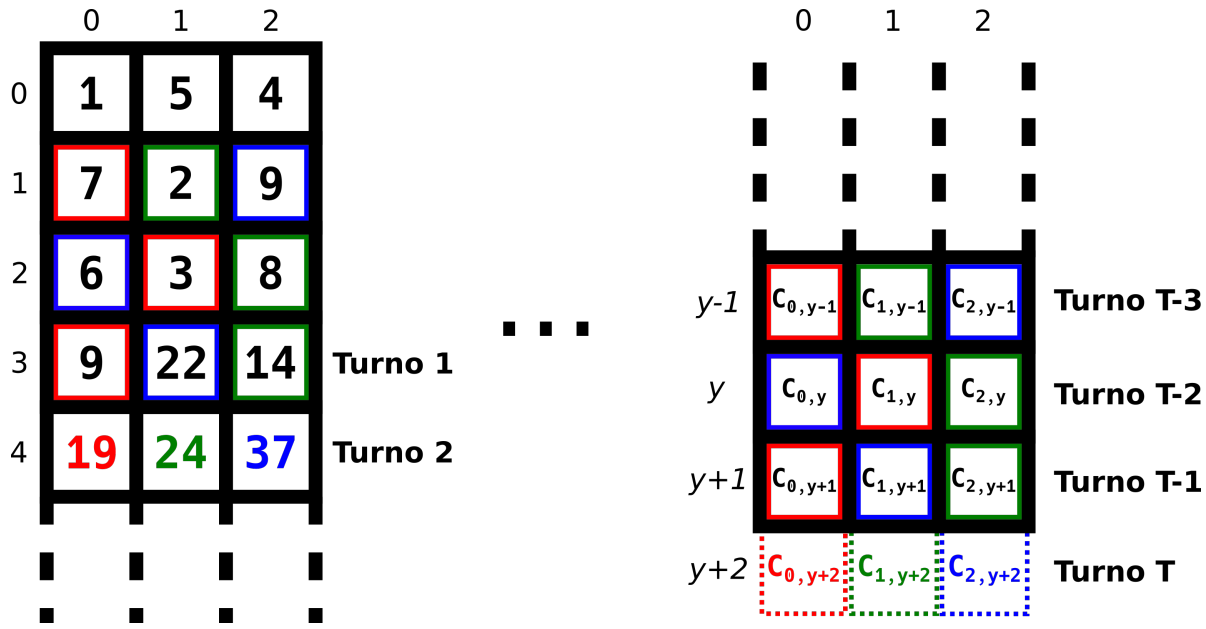
K: Kummirub: 2D++!

Simular os turnos do jogo envolveria realizar as somas iterando sobre o tabuleiro $N \times N$ para os T turnos. A complexidade seria $\mathcal{O}(N^2 \cdot T)$, o que seria impraticável para o tempo limite de 1 segundo.

No entanto, pode-se montar uma matriz que represente a recursão do problema, levando em consideração as escolhas dos jogadores. Nesse caso, é possível utilizar algum algoritmo de exponenciação de matrizes que seja suficientemente rápido. O mais comum é a exponenciação binária. Este não será explicado em detalhes aqui, mas consiste em se aproveitar da propriedade associativa da multiplicação de matrizes de uma forma recursiva. Sendo $\mathbf{M}$ uma matriz, pode-se calcular $\mathbf{M}^n$ com $\mathbf{M}^{\frac{n}{2}} \cdot \mathbf{M}^{\frac{n}{2}}$ se n for par, ou $\mathbf{M}^{n-1} \cdot \mathbf{M}$ para n ímpar.

Se tiver dificuldade em programar soluções que envolvem operações de matrizes, veja este recurso. Toda vez que pensar em uma solução que usa matrizes, é só copiar o código do seu caderno. Isso poupa **muito** tempo.

Para entender melhor a ideia, usaremos o exemplo dado no enunciado, onde temos o seguinte tabuleiro na entrada, expandido até o caso geral:



Indexando as coordenadas em x e y das casas a partir de 0, podemos chegar ao seguinte passo recursivo, para quando $y \geq 3$:

$$\begin{cases} C_{0,y+2} = C_{0,y+1} + C_{1,y} + C_{0,y-1} \\ C_{1,y+2} = C_{2,y+1} + C_{2,y} + C_{1,y-1} \\ C_{2,y+2} = C_{1,y+1} + C_{0,y} + C_{2,y-1} \end{cases} \tag{1}$$

Então, é possível montar a seguinte relação, utilizando matrizes:

$$\begin{bmatrix} C_{0,y+2} \\ C_{1,y+2} \\ C_{2,y+2} \\ C_{0,y+1} \\ C_{1,y+1} \\ C_{2,y+1} \\ C_{0,y} \\ C_{1,y} \\ C_{2,y} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} C_{0,y+1} \\ C_{1,y+1} \\ C_{2,y+1} \\ C_{0,y} \\ C_{1,y} \\ C_{2,y} \\ C_{0,y-1} \\ C_{1,y-1} \\ C_{2,y-1} \end{bmatrix}$$

Ao expandir a recursão, chega-se em:

$$\begin{bmatrix} C_{0,y+2} \\ C_{1,y+2} \\ C_{2,y+2} \\ C_{0,y+1} \\ C_{1,y+1} \\ C_{2,y+1} \\ C_{0,y} \\ C_{1,y} \\ C_{2,y} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}^T \begin{bmatrix} C_{0,2} \\ C_{1,2} \\ C_{2,2} \\ C_{0,1} \\ C_{1,1} \\ C_{2,1} \\ C_{0,0} \\ C_{1,0} \\ C_{2,0} \end{bmatrix}$$

Após realizar a exponenciação, basta realizar a multiplicação para obter as respostas esperadas.

Para N jogadores, é necessária uma matriz $N \times N$ como visto no exemplo acima. Assim, a multiplicação de matrizes tem complexidade $\mathcal{O}(N^6)$. Já a exponenciação binária da matriz tem complexidade $\mathcal{O}(\lg(T))$.

Complexidade total: $\mathcal{O}(N^6 \lg(T))$.

kummirub.cpp

```cpp
#include <bits/stdc++.h>
#define MOD 1000000007;

using namespace std;
using ll = long long;

struct Matrix {
    int row, col; // Dimensões
    vector<vector<ll>> data;

    Matrix(int rows, int cols, ll diag) : row(rows), col(cols) {
        data = vector<vector<ll>>(row, vector<ll>(col));
        for (int i = 0; i<row && i<col; ++i)
            data[i][i] = diag;
    }

    Matrix operator*(const Matrix& rhs) {
        Matrix saida(row, rhs.col, 0);
        for (int i = 0; i<row; ++i) {
```

```
            for (int k = 0; k<col; ++k) { // Invertido (chache hit)
                for (int j = 0; j<rhs.col; ++j) { // Invertido (chache hit)
                    saida.data[i][j] += (data[i][k] * rhs.data[k][j]) %MOD;
                    saida.data[i][j] %= MOD;
                }
            }
        }
        return saida;
    }

    Matrix pow(int exp) { // Ex:  22 = 0b10110 ⟹ M^16 * M^4 * M^2
        Matrix saida(row, col, 1), aux = *this;
        while (exp) {
            if (exp&1) saida = saida * aux;  // exp&1 == exp%2
            aux = aux*aux; // A cada iteração:  M, M^2, M^4, M^8, ...
            exp >>= 1; // exp /= 2;
        }
        return saida;
    }
};

int main() {
    int n, t;
    Matrix vet(n*n, 1, 0); // Vetor do tabuleiro
    Matrix mat(n*n, n*n, 0); // Matriz da operação

    scanf("%d", &n);

    for (int i = 0; i<n; ++i)
        for (int j = 0; j<n; ++j)
            scanf("%lld", &vet.data[(n-1-i)*n + j][0]);

    scanf("%d", &t);

    for (int i = 0; i<n; ++i) {
        for (int j = 0; j<n; ++j) {
            int aux;
            scanf("%d", &aux);
            mat.data[aux-1][(n-1-i)*n + j] = 1; // Montagem da matriz
        }
    }

    for (int i = 0; i<n*n-n; ++i)
        mat.data[n+i][i] = 1; // Montagem da matriz

    mat = mat.pow(t); // Exponenciação binária

    vet = mat*vet; // Aplicação da matriz para obter o resultado

    printf("%lld", vet.data[0][0]);
    for (int i = 1; i<n; ++i)
        printf(" %lld", vet.data[i][0]);
    printf("\n");

    return 0;
}
```