Ma trận là gì

Ma trận là một bảng số 2 chiều (một bảng gồm m hàng và n cột, mỗi ô của bảng là một con số)

Vector hàng là ma trận chỉ gồm 1 hàng (ma trận 1 \* m). Khi đó, ta gọi là vector hàng gồm n phần tử.

Vector cột là ma trận chỉ gồm 1 cột (ma trận n \* 1). Khi đó, ta gọi là vector cột gồm m phần tử.

Các phép toán trên ma trận

# Phép cộng ma trận

Phép cộng hai ma trận A và B (A + B) chỉ thực hiện được khi A và B có cùng số hàng và cùng số cột.

Khi đó, giá trị A + B là một ma trận C có kích thước giống như A và B.

A(m \* n) + B(m \* n) = C(m \* n)

c[i][j] = a[i][j] + b[i][j]

Tóm lại, phép cộng ma trận bản chất là cộng theo từng phần tử tương ứng.

# Phép nhân ma trận

Phép nhân ma trận A với ma trận B chỉ thực hiện được khi số cột của ma trận A trùng với số hàng của ma trận B. Khi đó, tích A \* B là một ma trận C có số hàng = số hàng của A, số cột = số cột của B

A(m \* n) \* B(n \* p) = C(m \* p)

c[i][k] = tổng a[i][j] \* b[j][k] với j = 1..n

Cách hiểu công thức: Với i = 1..m; k = 1..p

Phần tử hàng i cột k của ma trận C = A \* B, được xác định bởi hàng i của ma trận A và cột k của ma trận B

Hàng i của ma trận A có n phần tử: a[i][1] a[i][2] ... a[i][n]

Cột k của ma trận B có n phần tử: b[1][k] b[2][k] ... b[n][k]

Nhân tương ứng các vị trí lại với nhau rồi cộng các tích, ta được giá trị của c[i][k]

c[i][k] = a[i][1] \* b[1][k] + a[i][2] \* b[2][k] + ... + a[i][n] \* b[n][k]

## Các trường hợp đặc biệt của phép nhân ma trận

Trường hợp 1: A là vector cột (n \* 1), B là vector hàng (1 \* n) -> C = A \* B = ma trận vuông n \* n

c[i][k] = a[i][1] \* b[1][k]

Trường hợp 2: A là vector hàng (1 \* n), B là vector cột (n \* 1) -> C = A \* B = ma trận vuông 1 \* 1

c[1][1] = a[1][1] \* b[1][1] + a[1][2] \* b[2][1] + ... + a[1][n] \* b[n][1]

## Trường hợp quan trọng

A là vector hàng (1 \* n), B là ma trận vuông (n \* n)   
 -> C = A \* B = vector hàng (1 \* n)

C[1][i] = A[1][1] \* B[1][i] + A[1][2] \* B[2][i] + A[1][3] \* B[3][i]   
 + ... + A[1][n] \* B[n][i]

Do A và C chỉ là các vector hàng (chỉ gồm 1 hàng), ta bỏ đi toạ độ hàng, tức là  
 A = a[1] a[2] ... a[n]

C = c[1] c[2] ... c[n]

Rút gọn công thức trên bằng việc xoá bỏ chỉ số hàng của A và C, ta được

c[i] = a[1] \* b[1][i] + a[2] \* b[2][i] + a[3] \* b[3][i] + ... + a[n] \* b[n][i]

Quan sát công thức này, ta hiểu cách xác định c[i] (phần tử thứ i của vector hàng c) như sau:

c[i] là một tổng có hệ số của a[1] a[2] ... a[n]

Hệ số của a[1] là b[1][i]

Hệ số của a[2] là b[2][i]

Hệ số của a[3] là b[3][i]

...

Hệ số của a[n] là b[n][i]

b[1][i] b[2][i] ... b[n][i] = cột thứ i của ma trận B

Cột thứ i của ma trận B quy định các hệ số của các phần tử A đóng góp vào c[i]  
 **b[j][i] = hệ số đóng góp của a[j] vào c[i]**

# Phép toán luỹ thừa

Định nghĩa phép toán luỹ thừa của ma trận giống như luỹ thừa trên số

Ma trận A, số tự nhiên k: A^k = A \* A \* A \* ... \* A (k ma trận A nhân với nhau)

Phép luỹ thừa chỉ thực hiện được với ma trận vuông

Để phép toán A(m\*n) \* A(m\*n) có thể thực hiện được thì m = n

## Ma trận đơn vị

Ma trận đơn vị cấp n là một ma trận vuông gồm n hàng và n cột thoả mãn:  
 ô (i, j) = 1 nếu i = j

ô (i, j) = 0 nếu i khác j

Trong các phép nhân và luỹ thừa ma trận, ma trận đơn vị đóng vai trò như số 1 trong phép nhân/luỹ thừa trên số

Nếu I là ma trận đơn vị cấp n và A là ma trận vuông n \* n thì

A \* I = I \* A = A

A^0 = I

# Tính chất của các phép toán trên ma trận

Phép cộng ma trận có các tính chất của phép cộng thông thường: giao hoán, kết hợp

Phép nhân ma trận **nhìn chung không giao hoán**. Tức là với hai ma trận A và B, A \* B và B \* A không bằng nhau.

A(2 \* 3); B(3 \* 4): A \* B = ma trận 2 hàng 4 cột; B \* A không thực hiện được

A(6 \* 9); B(9 \* 6): A \* B = ma trận 6 \* 6, B \* A = ma trận 9 \* 9

A(3 \* 3); B(3 \* 3): Cả A \* B và B \* A đều thực hiện được và đều cho ra ma trận vuông 3 \* 3. Nhưng hai ma trận A \* B và B \* A khi đó vẫn không chắc bằng nhau.

( 3 6 9 ) ( 1 3 2 ) ( ? ? ? )

( 2 2 7 ) \* ( 0 1 2 ) = ( 37 ? ? )

( 9 9 7 ) ( 5 0 1 ) ( ? ? ? )

( 1 3 2 ) ( 3 6 9 ) ( ? ? ? )

( 0 1 2 ) \* ( 2 2 7 ) = ( 20 ? ? )

( 5 0 1 ) ( 9 9 7 ) ( ? ? ? )

Tuy nhiên, có các trường hợp đặc biệt, khi đó phép nhân ma trận vẫn "giao hoán":

A \* I = I \* A = A (nhân với ma trận đơn vị)

A^i \* A^j = A^j \* A^i = A^(i+j)

Cài đặt

Lưu ý về cài đặt:

- Ta cài đặt bằng mảng để tối ưu về mặt thời gian

- Các hàng được đánh số từ 0 đến m - 1, các cột được đánh số từ 0 đến n - 1

- Các phép toán cộng nhân và luỹ thừa được giả sử rằng các điều kiện thực hiện phép toán đều hợp lệ.

- Chú ý, khi tính A \* B^k, ta phải gọi A \* (B^k) (có dấu ngoặc bao quanh B^k) vì nếu không có dấu ngoặc, phép \* sẽ được thực hiện trước phép ^

- Độ phức tạp:

Phép cộng: O(m \* n)

Phép nhân: O(m \* n \* p); trong trường hợp ma trận vuông n \* n: O(n^3)

Phép luỹ thừa: O(n^3 \* log(k))

# Cài đặt đơn giản

Cài đặt struct Matrix mô phỏng ma trận và các phép toán cộng nhân luỹ thừa như định nghĩa ở trên

#define MAX\_SIZE 20

struct Matrix {

int m, n; // m = số hàng, n = số cột  
 int d[MAX\_SIZE][MAX\_SIZE];   
  
 Matrix(int \_m = 0, int \_n = 0) {  
 m = \_m; n = \_n;  
 memset(d, 0, sizeof d);  
 }  
  
 Matrix operator + (const Matrix &a) const { // phép cộng ma trận  
 Matrix res(m, n);  
 for (int i = 0; i < m; i++) for (int j = 0; j < n; j++)  
 res.d[i][j] = d[i][j] + a.d[i][j];  
 return res;  
 }  
  
 Matrix operator \* (const Matrix &a) const { // phép nhân ma trận  
 int x = m, y = n, z = a.n;  
 Matrix res(x, z);  
 for (int i = 0; i < x; i++) for (int j = 0; j < y; j++)   
 for (int k = 0; k < z; k++) res.d[i][k] += d[i][j] \* a.d[j][k];  
 return res;  
 }  
  
 Matrix operator ^ (long long k) const { // phép luỹ thừa ma trận  
 Matrix res(n, n);  
 for (int i = 0; i < n; i++) res.d[i][i] = 1;  
 Matrix mul = \*this;  
 while (k > 0) {  
 if (k & 1) res = res \* mul;  
 mul = mul \* mul;  
 k >>= 1;  
 }  
 return res;  
 }

};

# Ma trận tính kết quả theo modulo

const int MOD = 998244353;

#define MAX\_SIZE 20

struct Matrix {

int m, n; // m = số hàng, n = số cột  
 int d[MAX\_SIZE][MAX\_SIZE];   
  
 Matrix(int \_m = 0, int \_n = 0) {  
 m = \_m; n = \_n;  
 memset(d, 0, sizeof d);  
 }  
  
 Matrix operator + (const Matrix &a) const { // phép cộng ma trận  
 Matrix res(m, n);  
 for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) {  
 res.d[i][j] = d[i][j] + a.d[i][j];  
 if (res.d[i][j] >= MOD) res.d[i][j] -= MOD;  
 }  
   
 return res;  
 }  
  
 Matrix operator \* (const Matrix &a) const { // phép nhân ma trận  
 int x = m, y = n, z = a.n;  
 Matrix res(x, z);  
 for (int i = 0; i < x; i++) for (int j = 0; j < y; j++)   
 for (int k = 0; k < z; k++)   
 res.d[i][k] = (res.d[i][k] + 1LL \* d[i][j] \* a.d[j][k]) % MOD;  
 return res;  
 }  
  
 Matrix operator ^ (long long k) const { // phép luỹ thừa ma trận  
 Matrix res(n, n);  
 for (int i = 0; i < n; i++) res.d[i][i] = 1;  
 Matrix mul = \*this;  
 while (k > 0) {  
 if (k & 1) res = res \* mul;  
 mul = mul \* mul;  
 k >>= 1;  
 }  
 return res;  
 }

};

# Cài đặt nâng cao

Vẫn sử dụng cho trường hợp cần tính toán kết quả theo modulo, nhưng có thêm một số tối ưu để thay đổi cấu trúc cũng như tăng thời gian chạy.

+ Trong thao tác nhân ma trận, số phép % cần thực hiện giảm từ O(n^3) xuống còn O(n^2) -> tiết kiệm đáng kể thời gian. Tuy nhiên, để thực hiện được điều này thì mảng d phải có kiểu long long

+

const int MOD = (int)1e9 + 22071997;

#define MATRIX\_SIZE 55

struct Matrix {

int m, n;

long long d[MATRIX\_SIZE][MATRIX\_SIZE];

Matrix(int \_m = 0, int \_n = 0) {

m = \_m; n = \_n;

memset(d, 0, sizeof d);

}

Matrix operator + (const Matrix &a) const {

Matrix res(m, n);

for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) {

res.d[i][j] = d[i][j] + a.d[i][j];

if (res.d[i][j] >= MOD) res.d[i][j] -= MOD;

}

return res;

}

Matrix operator \* (const Matrix &a) const {

int x = m, y = n, z = a.n;

Matrix res(x, z);

for (int i = 0; i < x; i++) for (int j = 0; j < y; j++)   
 for (int k = 0; k < z; k++) {

res.d[i][k] += 1LL \* d[i][j] \* a.d[j][k];

if (res.d[i][k] >= 1LL \* MOD \* MOD) res.d[i][k] -= 1LL \* MOD \* MOD;

}

for (int i = 0; i < x; i++) for (int k = 0; k < z; k++) res.d[i][k] %= MOD;

return res;

}

Matrix operator ^ (long long k) const {

Matrix res(n, n);

for (int i = 0; i < n; i++) res.d[i][i] = 1;

Matrix mul = \*this;

while (k > 0) {

if (k & 1) res = res \* mul;

mul = mul \* mul;

k >>= 1;

}

return res;

}

long long\* operator[] (int i) {

return d[i];

}

const long long\* operator[] (int i) const {

return d[i];

}

};

Nếu không có 2 hàm màu tím, để truy cập (lấy giá trị) của ô (i, j) của ma trận m, ta phải dùng cú pháp m.d[i][j]

Matrix m(5, 5); // khởi tạo ma trận kích thước 5 \* 5, các phần tử đã gán = 0

m.d[0][1] = 2; // gán ô (0, 1) của ma trận m = 2

cout << m.d[3][4] << endl; // in ra giá trị của ô (3, 4) của ma trận m

Tuy nhiên, khi có khai báo thêm hai hàm màu tím, tahy vì viết m.d[i][j] ta chỉ cần viết m[i][j] là đủ

Matrix m(5, 5); // khởi tạo ma trận kích thước 5 \* 5, các phần tử đã gán = 0

m[0][1] = 2; // gán ô (0, 1) của ma trận m = 2

cout << m[3][4] << endl; // in ra giá trị của ô (3, 4) của ma trận m

Cách sử dụng thuật toán nhân ma trận

Các bài toán sử dụng nhân ma trận thường có dạng như sau:   
 Cho các hàm số F(i), G(i), H(i) có công thức truy hồi i phụ thuộc i - 1, chẳng hạn

F(i) = 3 \* F(i - 1) + 2 \* G(i - 1) + H(i - 1)

G(i) = 2 \* F(i - 1) - H(i - 1)

H(i) = -2 \* G(i - 1) + 3 \* H(i - 1)

Cho biết các giá trị F(0) G(0) H(0), cần tình F(n) G(n) H(n) với n rất lớn (n ~ 1e18)

Cách giải: Gọi V(i) là một vector hàng 3 phần tử [F(i) G(i) H(i)]. Dựa vào công thức truy hồi của F(i) G(i) H(i), ta có thể tính công thức V(i) dựa trên V(i - 1) như sau: V(i) = V(i - 1) \* A trong đó A là ma trận 3 \* 3 như sau:

( 3 2 0 )

A = ( 2 0 -2 ) V(i) = (F(i) G(i) H(i))

( 1 -1 3 )

Do đã biết F(0) G(0) H(0), ta biết V(0). V(i) = V(i - 1) \* A -> V(n) = V(0) \* A^n

Ta tính được A^n trong đpt O(log(n)) (phép luỹ thừa ma trận). Do đó, sau khi tìm được V(n), ta cũng có luôn F(n) G(n) H(n).

Để có thể áp dụng nhân ma trận vào việc tính các hàm F G H, công thức truy hồi cần thoả mãn các điều kiện sau đây:

+ F(i) G(i) H(i) phải là các tổ hợp tuyến tính của F(i - 1) G(i - 1) H(i - 1): Chỉ chứa phép cộng và phép nhân với hệ số

+ F(i) G(i) H(i) chỉ tính theo bước i - 1 mà không tính theo bất kỳ bước nào khác.

+ Các hệ số phải là hằng số không phụ thuộc vào i

+ Công thức truy hồi của i theo i - 1 phải như nhau với mọi i

Các công thức sau không thể dùng nhân ma trận để tính được:  
F(i) = G(i - 1) \* H(i - 1)

F(i) = F(i - 1)^2

F(i) = i \* G(i - 1) + H(i - 1)

F(i) = F(i - 1) + F(i - 100)

F(i) = G(i / 2)

# Ví dụ: Tính số fibonacci thứ n ~ 1e18

Dãy fibonacci:

fib(0) = 0

fib(1) = 1

fib(i) = fib(i - 1) + fib(i - 2) với mọi i

Nhận xét: Công thức truy hồi fib(i) = fib(i - 1) + fib(i - 2) chưa thoả mãn điều kiện của một công thức có thể tính bằng nhân ma trận (i truy hồi theo cả i - 1 và i - 2). Để có thể nhân ma trận được, ta phải biến đổi sao cho công thức truy hồi của i chỉ tính theo i - 1.

Do đó, ta cần biến đổi công thức như sau:

Đặt F(i) = fib(i) và G(i) = fib(i + 1). Ta có:  
 F(0) = 0, G(0) = 1

**F(i) = G(i - 1)** **G(i) = F(i - 1) + G(i - 1)** [ G(i) = fib(i + 1) = fib(i) + fib(i - 1); fib(i) = G(i - 1); fib(i - 1) = F(i - 1) ]

Các công thức truy hồi của F(i) và G(i) đã thoả mãn điều kiện có thể tính bằng nhân ma trận. Do đó, ta thiết lập nhân ma trận như sau:

Gọi V(i) = vector hàng 2 phần tử F(i), G(i). Dựa trên công thức tính F(i) G(i) theo F(i - 1) và G(i - 1), ta có

V(i) = V(i - 1) \* A, trong đó A là ma trận 2 \* 2

A = ( 0 1 ) V(i) = (F(i) G(i))

( 1 1 )

Kết hợp với V(0) = (0 1); V(n) = V(0) \* A^n. Từ đó ta tính được F(n) trong O(log(n))

Code: <https://ideone.com/1314Ch>