

Nhân ma trận (Matrix multiplication)

Nhân ma trận (Matrix multiplication)

Nguồn: Biên soạn lại từ bài viết của Nguyễn RR Thành Trung, Nguyễn Mạnh Quân.

Tác giả:

- Nguyễn Châu Khanh - VNU University of Engineering and Technology (VNU-UET)
- Bùi Minh Hoạt - VNU University of Engineering and Technology (VNU-UET)

Reviewer:

- Trần Quang Lộc - ITMO University
- Hồ Ngọc Vĩnh Phát - VNUHCM-University of Science
- Trần Xuân Bách - HUS High School for Gifted Students
- Nguyễn Phú Bình - Hung Vuong High School for the Gifted, Binh Duong Province

Mở đầu

Thông thường, để đạt được độ phức tạp thuật toán như mong muốn, cách làm thường là tìm ra một thuật toán ban đầu làm cơ sở, rồi từ đó dùng các kỹ năng để giảm độ phức tạp của thuật toán. Trong bài viết này, tôi xin giới thiệu với bạn đọc một kỹ năng khá thông dụng: **Nhân ma trận**.

Định nghĩa

Tham khảo: [Ma trận_wikipedia](#) [🔗](#)

Ma trận

Ma trận là một mảng chữ nhật gồm các số, ký hiệu, hoặc biểu thức, sắp xếp theo hàng và cột mà mỗi ma trận tuân theo những quy tắc định trước.

Các ô trong ma trận được gọi là các phần tử của ma trận. Các phần tử được xác định bằng 2 địa chỉ hàng i và cột j tương ứng (Kí hiệu là a_{ij}).

Ma trận thường được viết trong dấu ngoặc vuông:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

Độ lớn hay kích thước của ma trận được định nghĩa bằng số lượng hàng và cột. Một ma trận m hàng và n cột được gọi là ma trận $(m \times n)$, trong khi m và n được gọi là **chiều** của nó.

- **Ví dụ:** Ma trận A là ma trận (3×2)

$$A = \begin{bmatrix} 1 & 2 \\ 5 & 7 \\ 6 & 3 \end{bmatrix}$$

Ma trận vuông

Ma trận vuông là ma trận có số hàng và số cột bằng nhau. Ma trận $(n \times n)$ còn gọi là ma trận vuông cấp n . Các phần tử a_{ii} tạo thành **đường chéo chính** của ma trận vuông.

- **Ví dụ:** Ma trận vuông cấp 3 (số hàng và số cột bằng 3)

$$\begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 1 \\ 2 & 3 & 1 \end{bmatrix}$$

Ma trận đơn vị (Identity Matrix)

Ma trận đơn vị I_n cấp n là một ma trận $(n \times n)$ trong đó mọi phần tử trên **đường chéo chính** \square bằng 1 và tất cả những phần tử khác đều bằng 0. Ma trận đơn vị cấp n cũng chính là ma trận vuông cấp n .

- **Ví dụ**

$$I_1 = [1] \quad I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \dots \quad I_n = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

Vector hàng và vector cột

Vector hàng hay **ma trận hàng** là một ma trận $(1 \times n)$, tức là ma trận chỉ gồm một hàng đơn gồm n phần tử.

$$\mathbf{a} = [a_1 \quad a_2 \quad \dots \quad a_n]$$

Vector cột hay **ma trận cột** là một ma trận $(m \times 1)$, tức là ma trận chỉ gồm một cột đơn gồm m phần tử.

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Ta định nghĩa tích của vector hàng \mathbf{a} $(1 \times n)$ với vector cột \mathbf{b} $(n \times 1)$ tương đương với **tích vô hướng** \square của hai vector \mathbf{a} và \mathbf{b} .

$$\mathbf{a} \cdot \mathbf{b} = [a_1 \quad a_2 \quad \dots \quad a_n] \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$$

Tham khảo: [Vector hàng và cột](#) 

Phép nhân ma trận

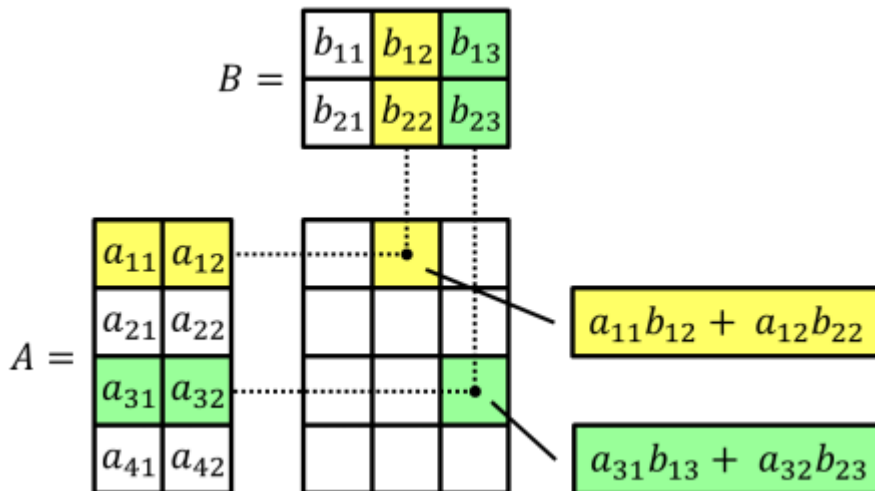
Phép nhân hai ma trận chỉ thực hiện được khi số lượng cột trong ma trận thứ nhất phải bằng số lượng hàng trong ma trận thứ hai. Ma trận kết quả, được gọi là **tích ma trận**, có số lượng hàng của ma trận đầu tiên và số cột của ma trận thứ hai.

Nếu ma trận A có kích thước $(m \times n)$ và ma trận B có kích thước $(n \times p)$, thì ma trận tích $C = A \times B$ có kích thước $(m \times p)$, phần tử đứng ở hàng thứ i , cột thứ j xác định bởi công thức:

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + \dots + A_{in}B_{nj} = \sum_{k=1}^n A_{ik}B_{kj} \text{ (Với } 1 \leq i \leq m; 1 \leq j \leq p)$$

Hay viết $C_{ij} = [a_{i1} \quad a_{i2} \quad \dots \quad a_{in}] \begin{bmatrix} b_{1j} \\ b_{2j} \\ \vdots \\ b_{nj} \end{bmatrix}$, tức là phần tử ở hàng thứ i , cột thứ j của C là tích của vector hàng thứ i của ma trận A với vector cột thứ j của ma trận B .

- Minh họa tích ma trận AB của hai ma trận A và B :



- **Ví dụ:** Cho 2 ma trận

$$A = \begin{bmatrix} 2 & 3 & 4 \\ 1 & 0 & 0 \end{bmatrix} \text{ và } B = \begin{bmatrix} 0 & 1000 \\ 1 & 100 \\ 0 & 10 \end{bmatrix}$$

Phần tử C_{12} của ma trận tích AB là tích của vector hàng thứ nhất của A và vector cột thứ hai của B , ta có:

$$C_{12} = \sum_{k=1}^3 A_{1k}B_{k2} = [2 \quad 3 \quad 4] \begin{bmatrix} 1000 \\ 100 \\ 10 \end{bmatrix} = 2 \times 1000 + 3 \times 100 + 4 \times 10 = 2340$$

Tính tương tự với tất cả phần tử còn lại của ma trận tích C . Ta được ma trận tích AB có dạng:

$$C = A \times B = \begin{bmatrix} 2 & 3 & 4 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1000 \\ 1 & 100 \\ 0 & 10 \end{bmatrix} = \begin{bmatrix} 3 & 2340 \\ 0 & 1000 \end{bmatrix} \quad \text{Mô tả quá trình nhân ma trận:}$$

Tính chất của phép nhân ma trận

- ▶ Tính chất kết hợp: $(AB)C = A(BC)$.
- ▶ Tính chất phân phối: $(A + B)C = AC + BC$, cũng như $C(A + B) = CA + CB$.
- ▶ Phép nhân ma trận **không** có tính chất giao hoán: Tích AB có thể xác định trong khi BA không nhất thiết phải xác định, tức là nếu A và B lần lượt có số chiều $(m \times n)$ và $(n \times p)$, và $m \neq p$. Thậm chí khi cả hai tích này đều tồn tại thì chúng không nhất thiết phải bằng nhau, tức là $AB \neq BA$.

▶ **Ví dụ:**

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 3 \end{bmatrix}, \text{ trong khi } \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix}.$$

- ▶ Khi thực hiện nhân một ma trận bất kì với **ma trận đơn vị** thì vẫn thu được kết quả của chính ma trận đó, tức là: $AI_n = I_m A = A$ (với ma trận A kích thước $(m \times n)$ bất kỳ).
Cũng chính vì tính chất này mà I có tên gọi là **ma trận đơn vị**.

Bạn có thể tìm hiểu thêm về **phép cộng trừ ma trận** tại [đây](#) .

Lũy thừa ma trận

Cho ma trận vuông A cấp n . Khi đó ta có phép tính ma trận A lũy thừa k (kí hiệu: A^k), với k là một số nguyên không âm.

$$A^k = \underbrace{A \times A \times A \times \dots \times A}_k$$

Trường hợp đặc biệt: Với $k = 0$, ma trận A^0 được xác định là **ma trận đơn vị** có cùng kích thước, tức là $A^0 = I_n$.

- ▶ **Ví dụ:** Cho ma trận vuông A cấp 3

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 1 \\ 2 & 3 & 1 \end{bmatrix}$$

$$A^0 = I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^2 = A \times A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 1 \\ 2 & 3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 1 \\ 2 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 7 & 2 & 2 \\ 5 & 9 & 1 \\ 13 & 7 & 4 \end{bmatrix}$$

$$A^3 = A \times A \times A = A^2 \times A = \begin{bmatrix} 7 & 2 & 2 \\ 5 & 9 & 1 \\ 13 & 7 & 4 \end{bmatrix} \begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 1 \\ 2 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 17 & 20 & 4 \\ 34 & 13 & 10 \\ 42 & 38 & 11 \end{bmatrix}$$

Nhờ **tính chất kết hợp** của phép nhân ma trận nên ta có thể tính nhanh lũy thừa của ma trận tương tự như cách tính hàm mũ thông thường bằng phương pháp **chia để trị** (tính a^k với a là số nguyên). Bạn có thể tìm hiểu về cách tính hàm mũ tại [đây](#).

Cài đặt

Lưu ý: Khác với định nghĩa bên trên, trong cách cài đặt sau, các hàng và cột của ma trận được đánh số bắt đầu từ 0 để thuận tiện cho việc xử lý.

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  using type = int; // Kiểu dữ liệu các phần tử của ma trận
6
7  struct Matrix {
8      vector <vector <type> > data;
9
10     // Số lượng hàng của ma trận
11     int row() const { return data.size(); }
12
13     // Số lượng cột của ma trận
14     int col() const { return data[0].size(); }
15
16     auto & operator [] (int i) { return data[i]; }
17
18     const auto & operator[] (int i) const { return data[i]; }
19
20     Matrix() = default;
21
22     Matrix(int r, int c): data(r, vector <type> (c)) { }
23
24     Matrix(const vector <vector <type> > &d): data(d) {
25
26         // Kiểm tra các hàng có cùng size không và size có lớn hơn 0 hay không
27         // Tuy nhiên không thực sự cần thiết, ta có thể bỏ các dòng /**/ đi
28         /**/ assert(d.size());
29         /**/ int size = d[0].size();
30         /**/ assert(size);
31         /**/ for (auto x : d) assert(x.size() == size);
32     }
33
34     // In ra ma trận.
35     friend ostream & operator << (ostream &out, const Matrix &d) {
36         for (auto x : d.data) {
37             for (auto y : x) out << y << ' ';
38             out << '\n';
39         }
40         return out;
41     }
42
43     // Ma trận đơn vị
44     static Matrix identity(long long n) {
45

```

```

46     Matrix a = Matrix(n, n);
47     while (n-- > 0) a[n][n] = 1;
48     return a;
49 }
50
51 // Nhân ma trận
52 Matrix operator * (const Matrix &b) {
53     Matrix a = *this;
54
55     // Kiểm tra điều kiện nhân ma trận
56     assert(a.col() == b.row());
57
58     Matrix c(a.row(), b.col());
59     for (int i = 0; i < a.row(); ++i)
60         for (int j = 0; j < b.col(); ++j)
61             for (int k = 0; k < a.col(); ++k)
62                 c[i][j] += a[i][k] * b[k][j];
63     return c;
64 }
65
66 // Lũy thừa ma trận
67 Matrix pow(long long exp) {
68
69     // Kiểm tra điều kiện lũy thừa ma trận (là ma trận vuông)
70     assert(row() == col());
71
72     Matrix base = *this, ans = identity(row());
73     for (; exp > 0; exp >>= 1, base = base * base)
74         if (exp & 1) ans = ans * base;
75     return ans;
76 }
77 };
78
79 int main(){
80     Matrix a({
81         {1, 2},
82         {3, 4}
83     });
84
85     Matrix b({
86         {0, 10, 100},
87         {1, 1, 10}
88     });
89
90     cout << a * b << '\n';
91     // 2 12 120
92     // 4 34 340
93
94     cout << a.pow(3) << '\n';
95     // 37 54
96     // 81 118

```

```

97
98     b = a;
99     cout << b << '\n';
100    // 1 2
101    // 3 4
102
103     b = Matrix::identity(3);
104     cout << b << '\n';
105     // 1 0 0
106     // 0 1 0
107     // 0 0 1
108
109     b = Matrix(2, 3);
110     cout << b << '\n';
111     // 0 0 0
112     // 0 0 0
113
114     Matrix c(3, 2);
115     cout << c << '\n';
116     // 0 0
117     // 0 0
118     // 0 0
    }

```

Bạn có thể tham khảo thêm cách cài đặt khác tại [đây](#) .

Đánh giá

Ngoài cách cài đặt tính **lũy thừa ma trận** như trên thì ta còn có thể cài đặt theo một cách khác bằng đệ quy như sau:

```

1  Matrix pow(long long exp) {
2      Matrix base = *this;
3      if (exp == 0) return identity(base.row());
4      if (exp == 1) return base;
5      Matrix p = pow(exp >> 1);
6      p = p * p;
7      if (exp & 1) return p * base;
8      return p;
9  }

```

Độ phức tạp

Nhân ma trận: Với ma trận A kích thước $(m \times n)$ và ma trận B kích thước $(n \times p)$. Độ phức tạp của thuật toán để tính $A \times B$ là $\mathcal{O}(m \times n \times p)$.

- **Ghi chú:** Đối với phép nhân các ma trận vuông kích thước $(n \times n)$, có thuật toán nhân ma trận [Strassen](#) với độ phức tạp $\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.807})$ theo tư tưởng chia nhỏ ma trận (tương tự cách nhân nhanh 2 số

lớn). Tuy nhiên cài đặt rất phức tạp và trên thực tế với giá trị n thường gặp, cách này không chạy nhanh hơn nhân ma trận thông thường $\mathcal{O}(n^3)$.

Lũy thừa ma trận: Với ma trận vuông A cấp n , thuật toán tính A^k có độ phức tạp $\mathcal{O}(n^3 \times \log k)$.

Ví dụ 1

Chúng ta hãy cùng xem xét một ví dụ kinh điển nhất trong ứng dụng của phép nhân ma trận.

Bài toán

[LATGACH4 - Lát gạch 4](#) 

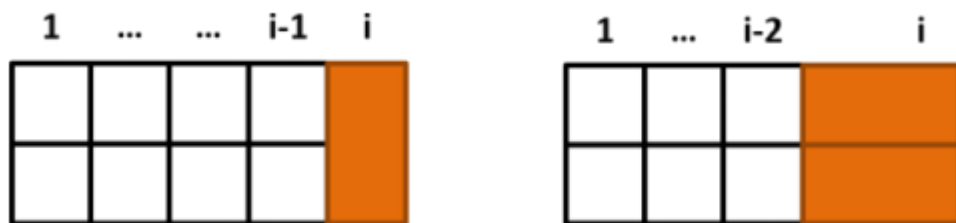
Cho một hình chữ nhật kích thước $2 \times N$ ($1 \leq N \leq 10^9$). Hãy đếm số cách lát các viên gạch nhỏ kích thước 1×2 và 2×1 vào hình trên sao cho không có phần nào của các viên gạch nhỏ thừa ra ngoài, cũng không có vùng diện tích nào của hình chữ nhật không được lát.

Phân tích

Gọi F_i là số cách lát các viên gạch nhỏ vào hình chữ nhật kích thước $2 \times i$. Ta có:

- Nếu sử dụng viên gạch kích thước 1×2 thì $F_i = F_{i-2}$.
- Nếu sử dụng viên gạch kích thước 2×1 thì $F_i = F_{i-1}$.

$$\Rightarrow F_i = F_{i-1} + F_{i-2}.$$



Do đó, bài toán quy về tìm số *Fibonacci* thứ N với dãy *Fibonacci* được định nghĩa như sau:

$$F_0 = 1$$

$$F_1 = 1$$

...

$$F_i = F_{i-1} + F_{i-2} \text{ (với } i \geq 2)$$

Hiển nhiên cách làm thông thường là tính lần lượt các F_i . Tuy nhiên, cách làm này hoàn toàn không hiệu quả với N lên đến 10^9 , và ta cần một cách tiếp cận khác.

Ta xét các lớp số:

- Lớp 1: F_1, F_0
- Lớp 2: F_2, F_1
- Lớp 3: F_3, F_2
- ...
- Lớp $i - 1$: F_{i-1}, F_{i-2}
- Lớp i : F_i, F_{i-1}

Ta hình dung mỗi lớp là một ma trận (2×1). Tiếp đó, ta sẽ biến đổi từ lớp $i - 1$ đến lớp i . Sau mỗi lần biến đổi như vậy, ta tính thêm được một giá trị F_i . Để thực hiện phép biến đổi này, chú ý là các số ở lớp sau chỉ phụ thuộc vào lớp ngay trước nó theo các phép cộng, ta tìm được cách biến đổi bằng nhân ma trận:

$$A \times \begin{bmatrix} F_{i-1} \\ F_{i-2} \end{bmatrix} = \begin{bmatrix} F_i \\ F_{i-1} \end{bmatrix}$$

Chắc hẳn đọc đến đây bạn đọc sẽ thắc mắc, làm thế nào để tìm được ma trận A ? Để tìm được ma trận này, ta làm như sau:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} F_{i-1} \\ F_{i-2} \end{bmatrix} = \begin{bmatrix} F_i \\ F_{i-1} \end{bmatrix}$$

Suy ra:

- $F_i = a_{11} \times F_{i-1} + a_{12} \times F_{i-2} = 1 \times F_{i-1} + 1 \times F_{i-2}$, do đó hàng đầu tiên của ma trận A là $[1 \quad 1]$.
- $F_{i-1} = a_{21} \times F_{i-1} + a_{22} \times F_{i-2} = 1 \times F_{i-1} + 0 \times F_{i-2}$, do đó hàng thứ hai của ma trận A là $[1 \quad 0]$.

$$\Rightarrow A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Ta có:

$$\begin{aligned} \begin{bmatrix} F_i \\ F_{i-1} \end{bmatrix} &= A \times \begin{bmatrix} F_{i-1} \\ F_{i-2} \end{bmatrix} = A^2 \times \begin{bmatrix} F_{i-2} \\ F_{i-3} \end{bmatrix} \left(\text{vì } \begin{bmatrix} F_{i-1} \\ F_{i-2} \end{bmatrix} = A \times \begin{bmatrix} F_{i-2} \\ F_{i-3} \end{bmatrix} \right) \\ &= A^3 \times \begin{bmatrix} F_{i-3} \\ F_{i-4} \end{bmatrix} = \dots = A^{i-1} \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} \\ \Rightarrow \begin{bmatrix} F_N \\ F_{N-1} \end{bmatrix} &= A^{N-1} \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{N-1} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad (\text{vì } F_0 = 1 \text{ và } F_1 = 1) \end{aligned}$$

Ma trận A còn được gọi là **ma trận hệ số** và ma trận $\begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$ được gọi là **ma trận cơ sở**.

Vậy bài toán trên được đưa về dạng **nhân ma trận**. F_N được tính dựa vào phép lũy thừa của ma trận A .

Cài đặt

Lưu ý: Khác với định nghĩa bên trên. Trong cách cài đặt sau, các hàng và cột của ma trận được đánh số bắt đầu từ 0 để thuận tiện cho việc xử lí.

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  const int mod = 111539786;
6
7  using type = int;
8
9  struct Matrix {
10     vector <vector <type> > data;
11
12     int row() const { return data.size(); }
13
14     int col() const { return data[0].size(); }
15
16     auto & operator [] (int i) { return data[i]; }
17
18     const auto & operator[] (int i) const { return data[i]; }
19
20     Matrix() = default;
21
22     Matrix(int r, int c): data(r, vector <type> (c)) { }
23
24     Matrix(const vector <vector <type> > &d): data(d) { }
25
26     friend ostream & operator << (ostream &out, const Matrix &d) {
27         for (auto x : d.data) {
28             for (auto y : x) out << y << ' ';
29             out << '\n';
30         }
31         return out;
32     }
33
34     static Matrix identity(long long n) {
35         Matrix a = Matrix(n, n);
36         while (n--) a[n][n] = 1;
37         return a;
38     }
39
40     Matrix operator * (const Matrix &b) {
41         Matrix a = *this;
42         assert(a.col() == b.row());
43         Matrix c(a.row(), b.col());
44         for (int i = 0; i < a.row(); ++i)
45             for (int j = 0; j < b.col(); ++j)
46                 for (int k = 0; k < a.col(); ++k){
47

```

```

48         c[i][j] += 111 * a[i][k] % mod * (b[k][j] % mod) % mod;
49         c[i][j] %= mod;
50     }
51     return c;
52 }
53
54 Matrix pow(long long exp) {
55     assert(row() == col());
56     Matrix base = *this, ans = identity(row());
57     for (; exp > 0; exp >= 1, base = base * base)
58         if (exp & 1) ans = ans * base;
59     return ans;
60 }
61 };
62
63 int main(){
64     Matrix a({
65         {1, 1},
66         {1, 0}
67     });
68
69     int t;
70     cin >> t;
71     while (t--) {
72         int n;
73         cin >> n;
74         Matrix tmp = a.pow(n - 1);
75         cout << (tmp[0][0] + tmp[0][1]) % mod << '\n';
76     }
}

```

Đánh giá

Độ phức tạp

Độ phức tạp của thuật toán là $\mathcal{O}(T \times 2^3 \times \log N)$. Với T là số lượng bộ test.

Ví dụ 2

Bây giờ chúng ta sẽ cùng xem xét một ví dụ tổng quát hơn của **ví dụ 1**.

Bài toán

SEQ - Recursive Sequence [✎](#)

Cho 2 dãy số nguyên độ dài k ($1 \leq k \leq 10$) là b_1, b_2, \dots, b_k và c_1, c_2, \dots, c_k ($0 \leq b_i, c_i \leq 10^9$ với $1 \leq i \leq k$). Dãy số a được xác định như sau:

- $a_i = b_i$ (với $1 \leq i \leq k$)

$$\triangleright a_i = c_1 \times a_{i-1} + c_2 \times a_{i-2} + \dots + c_k \times a_{i-k} \text{ (với } i > k)$$

Yêu cầu: Tính a_n với $n \leq 10^9$. Đáp án in ra theo modulo 10^9 .

Phân tích

Cũng như trong **ví dụ 1**, ta xét các lớp số:

- ▶ Lớp 1: a_1, a_2, \dots, a_k
- ▶ Lớp 2: a_2, a_3, \dots, a_{k+1}
- ▶ ...
- ▶ Lớp i : $a_i, a_{i+1}, \dots, a_{i+k-1}$

Mỗi lớp là một ma trận ($k \times 1$). Ta cũng sẽ áp dụng phép nhân ma trận để biến đổi từ lớp i sang lớp $i + 1$ như sau:

$$D \times \begin{bmatrix} a_i \\ a_{i+1} \\ \vdots \\ a_{i+k-2} \\ a_{i+k-1} \end{bmatrix} = \begin{bmatrix} a_{i+1} \\ a_{i+2} \\ \vdots \\ a_{i+k-1} \\ a_{i+k} \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} d_{1,1} & d_{1,2} & \dots & d_{1,k-1} & d_{1,k} \\ d_{2,1} & d_{2,2} & \dots & d_{2,k-1} & d_{2,k} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ d_{k-1,1} & d_{k-1,2} & \dots & d_{k-1,k-1} & d_{k-1,k} \\ d_{k,1} & d_{k,2} & \dots & d_{k,k-1} & d_{k,k} \end{bmatrix} \begin{bmatrix} a_i \\ a_{i+1} \\ \vdots \\ a_{i+k-2} \\ a_{i+k-1} \end{bmatrix} = \begin{bmatrix} a_{i+1} \\ a_{i+2} \\ \vdots \\ a_{i+k-1} \\ a_{i+k} \end{bmatrix}$$

Để xây dựng ma trận vuông D như trên, ta thực hiện tương tự như trong ví dụ trước: Phân tích a_{i+1} đến a_{i+k} dưới dạng a_i, \dots, a_{i+k-1} :

- ▶ $a_{i+1} = 0 \times a_i + 1 \times a_{i+1} + 0 \times a_{i+2} + \dots + 0 \times a_{i+k-1}$ nên hàng đầu tiên của ma trận D là $[0 \ 1 \ 0 \ \dots \ 0]$.
- ▶ $a_{i+2} = 0 \times a_i + 0 \times a_{i+1} + 1 \times a_{i+2} + \dots + 0 \times a_{i+k-1}$ nên hàng thứ hai của ma trận D là $[0 \ 0 \ 1 \ \dots \ 0]$.
- ▶ ...
- ▶ $a_{i+k-1} = 0 \times a_i + 0 \times a_{i+1} + 0 \times a_{i+2} + \dots + 1 \times a_{i+k-1}$ nên hàng thứ $k - 1$ của ma trận D là $[0 \ 0 \ 0 \ \dots \ 1]$.
- ▶ $a_{i+k} = c_k \times a_i + c_{k-1} \times a_{i+1} + c_{k-2} \times a_{i+2} + \dots + c_1 \times a_{i+k-1}$ nên hàng thứ k của ma trận D là $[c_k \ c_{k-1} \ c_{k-2} \ \dots \ c_1]$.

$$\Rightarrow D = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ c_k & c_{k-1} & c_{k-2} & \dots & c_1 \end{bmatrix}$$

Từ đó, ta thu được cách làm như trong **ví dụ 1**. Vì ta cần tính a_n nên chỉ cần xác định đến lớp $n - k + 1$.

$$\Rightarrow \begin{bmatrix} a_{n-k+1} \\ a_{n-k+2} \\ \vdots \\ a_{n-1} \\ a_n \end{bmatrix} = D^{n-k} \times \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{k-1} \\ a_k \end{bmatrix} = D^{n-k} \times \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{k-1} \\ b_k \end{bmatrix} \quad (\text{vì } a_i = b_i \text{ với } 1 \leq i \leq k)$$

Cài đặt

Lưu ý: Khác với định nghĩa bên trên. Trong cách cài đặt sau, các hàng và cột của ma trận được đánh số bắt đầu từ 0 để thuận tiện cho việc xử lý.

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  const int mod = 1e9;
6
7  using type = int;
8
9  struct Matrix {
10     vector <vector <type> > data;
11
12     int row() const { return data.size(); }
13
14     int col() const { return data[0].size(); }
15
16     auto & operator [] (int i) { return data[i]; }
17
18     const auto & operator[] (int i) const { return data[i]; }
19
20     Matrix() = default;
21
22     Matrix(int r, int c): data(r, vector <type> (c)) { }
23
24     Matrix(const vector <vector <type> > &d): data(d) { }
25
26     friend ostream & operator << (ostream &out, const Matrix &d) {
27         for (auto x : d.data) {
28             for (auto y : x) out << y << ' ';
29             out << '\n';
30         }
31         return out;
32     }
33
34     static Matrix identity(long long n) {
35         Matrix a = Matrix(n, n);
36         while (n--) a[n][n] = 1;

```

```

37         return a;
38     }
39
40     Matrix operator * (const Matrix &b) {
41         Matrix a = *this;
42         assert(a.col() == b.row());
43         Matrix c(a.row(), b.col());
44         for (int i = 0; i < a.row(); ++i)
45             for (int j = 0; j < b.col(); ++j)
46                 for (int k = 0; k < a.col(); ++k){
47                     c[i][j] += 1ll * a[i][k] % mod * (b[k][j] % mod) % mod;
48                     c[i][j] %= mod;
49                 }
50         return c;
51     }
52
53     Matrix pow(long long exp) {
54         assert(row() == col());
55         Matrix base = *this, ans = identity(row());
56         for (; exp > 0; exp >>= 1, base = base * base)
57             if (exp & 1) ans = ans * base;
58         return ans;
59     }
60 };
61
62 int b[15], c[15];
63
64 int main(){
65     int t;
66     cin >> t;
67     while (t--) {
68         int n, k;
69         cin >> k;
70         for (int i = 1; i <= k; ++i) cin >> b[i];
71         for (int i = 1; i <= k; ++i) cin >> c[i];
72         cin >> n;
73
74         if (n <= k) { cout << b[n] << '\n'; continue; }
75
76         // Xây dựng ma trận cơ sở
77         Matrix base(k, 1);
78         for (int i = 1; i <= k; ++i) base[i - 1][0] = b[i];
79
80         // Xây dựng ma trận hệ số D
81         Matrix d(k, k);
82         for (int i = 0; i < k - 1; ++i) d[i][i + 1] = 1;
83         for (int i = 0; i < k; ++i) d[k - 1][i] = c[k - i];
84
85         Matrix ans = d.pow(n - k) * base;
86         cout << ans[k - 1][0] << '\n';
87     }
88 }

```

Đánh giá

Độ phức tạp

Độ phức tạp của thuật toán là $\mathcal{O}(t \times k^3 \times \log(n - k))$. Với t là số lượng bộ test.

Ví dụ 3

Bài toán

VNOJ - THBAC 

Người ta mới tìm ra một loại vi khuẩn mới. Chúng sống thành N bầy ($N \leq 100$), đánh số từ 0 đến $N - 1$. Ban đầu, mỗi bầy này chỉ có một con vi khuẩn. Tuy nhiên, mỗi giây, số lượng vi khuẩn trong các bầy lại có sự thay đổi: Một số vi khuẩn có thể chết đi, sinh sản thêm, hoặc di chuyển sang bầy khác. Những thay đổi này luôn tuân theo một quy luật có sẵn. Tại mỗi giây chỉ xảy ra đúng một quy luật. Các quy luật này được thực hiện nối tiếp nhau và theo chu kỳ. Có nghĩa là, nếu đánh số các quy luật từ 0 đến $M - 1$, tại giây thứ S thì quy luật được áp dụng sẽ là $(S - 1) \bmod M$ ($M \leq 1000$).

Nhiệm vụ của bạn là tìm xem, với một bộ các quy luật cho trước, sau T đơn vị thời gian ($T \leq 10^{18}$), mỗi bầy có bao nhiêu vi khuẩn.

Các loại quy luật có thể có:

- **A i 0** : Tất cả các vi khuẩn thuộc bầy i chết.
- **B i k** : Số vi khuẩn trong bầy i tăng lên gấp k lần ($1 \leq k \leq 100$).
- **C i j** : số vi khuẩn trong bầy i tăng lên một số lượng bằng với số vi khuẩn trong bầy j .
- **D i j** : Các vi khuẩn thuộc bầy j di chuyển toàn bộ sang bầy i .
- **E i j** : Các vi khuẩn thuộc bầy i và bầy j đổi vị trí cho nhau.
- **F 0 0** : Vị trí các vi khuẩn di chuyển trên vòng tròn. Nghĩa là các vi khuẩn ở bầy i di chuyển sang bầy $(i + 1) \bmod N$. Các di chuyển xảy ra đồng thời.

Phân tích

Cách làm đơn giản nhất là chúng ta mô phỏng lại số lượng vi khuẩn trong mỗi bầy qua từng đơn vị thời gian. Cách làm này có độ phức tạp $\mathcal{O}(T \times N \times k)$ với $\mathcal{O}(k)$ là độ phức tạp cho xử lý số lớn. Cách này không thể chạy được với T lớn.

Ta hình dung số lượng vi khuẩn trong mỗi bầy trong một đơn vị thời gian là một dãy số. Như vậy, mỗi quy luật cho trước thực chất là một phép biến đổi từ một dãy số thành một dãy số mới, và ta hoàn toàn có thể thực hiện biến đổi này bằng một phép nhân ma trận.

Cụ thể hơn, ta coi số lượng vi khuẩn trong N bầy tại một thời điểm xác định là một ma trận $1 \times N$, và mỗi phép biến đổi là một ma trận $N \times N$. Khi áp dụng mỗi phép biến đổi, ta nhân hai ma trận nói trên với nhau.

Bây giờ, xét trường hợp $N = 4$, các ma trận tương ứng với các phép biến đổi lần lượt được mô tả như sau:

► Biến đổi: **A 2 0**

$$\begin{array}{c|cccc} 1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 1 \end{array}$$

► Biến đổi: **B 2 6**

$$\begin{array}{c|cccc} 1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 3 & 0 & 0 & 6 & 0 \\ 4 & 0 & 0 & 0 & 1 \end{array}$$

► Biến đổi: **C 1 3**

$$\begin{array}{c|cccc} 1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 \\ 4 & 0 & 1 & 0 & 1 \end{array}$$

► Biến đổi: **D 1 3**

$$\begin{array}{c|cccc} 1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 \\ 4 & 0 & 1 & 0 & 0 \end{array}$$

► Biến đổi: **E 1 3**

$$\begin{array}{c|cccc} 1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 1 & 0 \\ 4 & 0 & 1 & 0 & 0 \end{array}$$

► Biến đổi: **F 0 0**

$$\begin{array}{c|cccc} 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 1 & 0 & 0 & 0 \end{array}$$

Cũng như các bài toán trước, ta sẽ cố gắng áp dụng việc tính toán lũy thừa, kết hợp với phép nhân ma trận để giảm độ phức tạp từ T xuống $\log T$. Tuy nhiên, có thể thấy việc sử dụng phép lũy thừa trong bài toán này phần nào phức tạp hơn bởi các ma trận được cho không giống nhau. Để giải quyết vấn đề này, ta làm như sau:

Gọi X_1, X_2, \dots, X_m là các ma trận tương ứng với các phép biến đổi được cho.

Đặt $X = X_1 \times X_2 \times \dots \times X_m$.

Đặt $S = [1, 1, \dots, 1]$ (dãy số lượng vi khuẩn tại thời điểm đầu tiên).

Như vậy, $Y = S \times X^t \times X_1 \times X_2 \times \dots \times X_r$ là ma trận thể hiện số lượng vi khuẩn tại thời điểm $M \times t + r$.

Như vậy, thuật toán đến đây đã rõ. Ta phân tích $T = M \times t + r$, nhờ đó, ta có thể giải quyết bài toán trong $\mathcal{O}(N^3 \times M)$ cho bước tính ma trận X và $\mathcal{O}(N^3 \times (\log T/M + M))$ cho bước tính Y . Bài toán được giải quyết.

Ví dụ 4

Bài toán

[beautynumbers](#) - Số đẹp 

Số đẹp là một số nguyên dương với bất kỳ chữ số lẻ nào (1, 3, 5, 7, 9) đều xuất hiện lẻ lần nếu nó xuất hiện và bất kỳ chữ số chẵn nào (0, 2, 4, 6, 8) cũng xuất hiện chẵn lần nếu nó xuất hiện. Ví dụ số 141222124 là một số đẹp. Gọi f_n là số lượng số đẹp có không quá n chữ số. Yêu cầu với một số n ($1 \leq n \leq 10^{18}$) tính $f_n \bmod 1000000123$.

Phân tích

Cách làm đơn giản nhất là ta sử dụng [quy hoạch động](#) với 4 trạng thái:

- ▶ **added** : số lượng chữ số đã thêm vào.
- ▶ **ewoc** (*even_with_odd_cnt*) : số chữ số chẵn đã xuất hiện lẻ lần.
- ▶ **owoc** (*odd_with_odd_cnt*) : số chữ số lẻ đã xuất hiện lẻ lần.
- ▶ **added_odd** : số chữ số lẻ đã xuất hiện.

Ta không phải lưu **số chữ số chẵn đã xuất hiện** vì nếu chữ số chẵn đó không xuất hiện thì cũng đã được tính vào trường hợp nó đã xuất hiện chẵn lần rồi.

Do đó, ta có công thức quy hoạch động theo như code sau:

```

#include <bits/stdc++.h>

using namespace std;

const int mod = 1e9 + 123;

int n;
int dp[10005][6][6][6];

long long digit_dp(int added, int ewoc, int owoc, int added_odd) {

    // Khi đã chọn đủ n chữ số
    if (added == n) return (!ewoc && owoc == added_odd);

    if (dp[added][ewoc][owoc][added_odd] != -1) return dp[added][ewoc][owoc][added_odd];
    long long cur = 0;

    // Thêm vào 1 số chẵn đã xuất hiện lẻ lần
    if (ewoc)
        cur += digit_dp(added + 1, ewoc - 1, owoc, added_odd) * ewoc;

    // Thêm vào 1 số chẵn đã xuất hiện chẵn lần
    if (ewoc < 5)
        cur += digit_dp(added + 1, ewoc + 1, owoc, added_odd) * (5 - ewoc);

    // Thêm vào 1 số lẻ chưa xuất hiện
    if (added_odd < 5)
        cur += digit_dp(added + 1, ewoc, owoc + 1, added_odd + 1) * (5 - added_odd);

    // Thêm vào 1 số lẻ đã xuất hiện lẻ lần
    if (owoc)
        cur += digit_dp(added + 1, ewoc, owoc - 1, added_odd) * owoc;

    // Thêm vào 1 số lẻ đã xuất hiện chẵn lần
    if (owoc < added_odd)
        cur += digit_dp(added + 1, ewoc, owoc + 1, added_odd) * (added_odd - owoc);

    // Không nhất thiết phải chọn đủ n chữ số
    if (!ewoc && owoc == added_odd) ++cur;

    return dp[added][ewoc][owoc][added_odd] = cur % mod;
}

int solve(int n1) {
    n = n1;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < 6; ++j)
            for (int k = 0; k < 6; ++k)
                for (int l = 0; l < 6; ++l) dp[i][j][k][l] = -1;
}

```

```

51 // Loại trường hợp chọn phải số 0 vô nghĩa bằng cách đặt trước chữ số đầu
52 long long tmp1 = digit_dp(1, 1, 0, 0) * 4;
53
54 long long tmp2 = digit_dp(1, 0, 1, 1) * 5;
55 return (tmp1 + tmp2) % mod;
56 }
57
58 int main(){
59     int n1;
60     while (cin >> n1) cout << solve(n1) << '\n';
61 }

```

Bạn có thể tìm hiểu thêm về **kỹ thuật quy hoạch động sử dụng đệ quy có nhớ (Top-Down)** tại [đây](#) .

Vì số chữ số lẻ đã xuất hiện lẻ lần không được vượt quá số chữ số lẻ đã xuất hiện, nên ta tối ưu được số trạng thái xuống còn khoảng $6 \times 6 \times 6 / 2$. Khi đó, độ phức tạp của thuật toán trên sẽ là $\mathcal{O}(n \times (6 \times 6 \times 6 / 2))$.

Tuy nhiên, với $n \leq 10^{18}$ thì hiển nhiên thuật toán trên sẽ bị quá cả thời gian lẫn bộ nhớ.

Do đó, ta cần phải cải tiến thuật toán bằng **lũy thừa ma trận** với số trạng thái cho 1 lớp dp là $6 \times 6 \times 6 / 2 = 108$.

Tối ưu hóa thuật toán bằng cách tách n thành các lũy thừa của 2 sau đó sử dụng các ma trận hệ số tương ứng đã tính toán trước để tính nhanh kết quả.

Cài đặt

Lưu ý: Trong cách cài đặt sau, các hàng và cột của ma trận được đánh số bắt đầu từ 0 để thuận tiện cho việc xử lý.

```

#include <bits/stdc++.h>

using namespace std;

const int mod = 1e9 + 123;

using type = int;

struct Matrix {
    vector <vector <type> > data;

    int row() const { return data.size(); }

    int col() const { return data[0].size(); }

    auto & operator [] (int i) { return data[i]; }

    const auto & operator [] (int i) const { return data[i]; }
}

```

```

Matrix() = default;

Matrix(int r, int c): data(r, vector<type>(c)) { }

Matrix(const vector<vector<type>> &d): data(d) { }

friend ostream & operator << (ostream &out, const Matrix &d) {
    for (auto x : d.data) {
        for (auto y : x) out << y << ' ';
        out << '\n';
    }
    return out;
}

Matrix operator * (const Matrix &b) {
    Matrix a = *this;
    assert(a.col() == b.row());
    Matrix c(a.row(), b.col());
    for (int i = 0; i < a.row(); ++i)
        for (int j = 0; j < b.col(); ++j)
            for (int k = 0; k < a.col(); ++k){
                c[i][j] += 1ll * a[i][k] % mod * (b[k][j] % mod) % mod;
                c[i][j] %= mod;
            }
    return c;
}

};

int last;
int odd_id[6][6];
Matrix coef, base;
vector<Matrix> coef_pow;

int id(int ewoc, int owoc, int added_odd) {
    assert(owoc <= added_odd);
    return ewoc * last + odd_id[added_odd][owoc];
};

// Xây dựng ma trận hệ số
void build_coef() {
    int ans_id = id(5, 5, 5) + 1;
    coef = Matrix(ans_id + 1, ans_id + 1);

    for (int added_odd = 0; added_odd <= 5; ++added_odd)
        for (int ewoc = 0; ewoc <= 5; ++ewoc)
            for (int owoc = 0; owoc <= added_odd; ++owoc) {

                int cur_id = id(ewoc, owoc, added_odd);
                if (ewoc)
                    coef[id(ewoc - 1, owoc, added_odd)][cur_id] += ewoc;
            }
}

```

```

71         if (ewoc < 5)
72             coef[id(ewoc + 1, owoc, added_odd)][cur_id] += 5 - ewoc;
73
74         if (added_odd < 5)
75             coef[id(ewoc, owoc + 1, added_odd + 1)][cur_id] += 5 - added_odd;
76
77         if (owoc)
78             coef[id(ewoc, owoc - 1, added_odd)][cur_id] += owoc;
79
80         if (owoc < added_odd)
81             coef[id(ewoc, owoc + 1, added_odd)][cur_id] += added_odd - owoc;
82
83         if (!ewoc && owoc == added_odd) ++coef[ans_id][cur_id];
84     }
85
86     coef[ans_id][ans_id] = 1;
87 }
88
89 // Xây dựng ma trận cơ sở
90 void build_base() {
91     int ans_id = id(5, 5, 5) + 1;
92     base = Matrix(ans_id + 1, 1);
93     base[id(1, 0, 0)][0] = 4;
94     base[id(0, 1, 1)][0] = 5;
95 }
96
97 int main() {
98     last = 0;
99     for (int added_odd = 0; added_odd <= 5; ++added_odd)
100         for (int owoc = 0; owoc <= added_odd; ++owoc)
101             odd_id[added_odd][owoc] = ++last;
102
103     build_base();
104     build_coef();
105
106     // Tính lũy thừa ma trận hệ số tương ứng với lũy thừa của 2
107     coef_pow.push_back(coef);
108     for (int i = 1; i <= 60; ++i)
109         coef_pow.push_back(coef_pow[i - 1] * coef_pow[i - 1]);
110
111     long long n;
112     while (cin >> n) {
113         Matrix ans = base;
114         for (int i = 0; n > 0; n >= 1, ++i)
115             if (n & 1) ans = coef_pow[i] * ans;
116         cout << ans[ans.row() - 1][0] << '\n';
117     }
118 }

```

Đánh giá

Độ phức tạp

Ta mất độ phức tạp $\mathcal{O}(6 \times 6 \times 6 / 2)$ cho việc xây dựng ma trận hệ số.

Vì ma trận kết quả có kích thước là $((\text{số trạng thái}) \times 1)$ chứ không phải là $((\text{số trạng thái}) \times (\text{số trạng thái}))$, nên độ phức tạp nhân ma trận trong lúc tính kết quả là $(\text{số trạng thái})^2$ chứ không phải $(\text{số trạng thái})^3$. Nên độ phức tạp của thuật toán là $\mathcal{O}(\log 10^{18} \times 108^2)$.

Ngoài ra, kể cả khi ta không giảm số trạng thái xuống còn khoảng $6 \times 6 \times 6 / 2$ thì thuật toán này vẫn đủ tốt.

Ví dụ 5

Bài toán

[Codeforces - 446C DZY Loves Fibonacci Numbers](#) 

Dãy *Fibonacci* được định nghĩa như sau:

- $F_1 = 1$
- $F_2 = 1$
- \dots
- $F_n = F_{n-1} + F_{n-2} \ (n > 2)$

Cho một dãy gồm n số nguyên: a_1, a_2, \dots, a_n . Có m truy vấn, mỗi truy vấn thuộc một trong hai dạng:

- Dạng 1: **1 l r** : Tăng mỗi phần tử a_i thêm F_{i-l+1} , trong đó $l \leq i \leq r$.
- Dạng 2: **2 l r** : In ra giá trị của $\sum_{i=l}^r a_i$ theo modulo $10^9 + 9$.

Hãy thực hiện tất cả các truy vấn.

Phân tích

Bằng phương pháp quy nạp, ta có thể dễ dàng chứng minh 2 định lý sau:

- **Định lý 1:** Cho dãy $f_1 = a, f_2 = b, \dots, f_n = f_{n-1} + f_{n-2} \ (n > 2)$ thì $f_n = b \times F_{n-1} + a \times F_{n-2} \ (n > 2)$, trong đó F_i là số hạng thứ i của dãy *Fibonacci*.
- **Định lý 2:** Cho dãy $f_1 = a, f_2 = b, \dots, f_n = f_{n-1} + f_{n-2} \ (n > 2)$ thì $f_1 + f_2 + \dots + f_n = f_{n+2} - b$.

Ta còn có tính chất của dãy *Fibonacci* như sau:

- Ta có thể chuyển đổi hai số hạng đầu tiên của dãy *Fibonacci* để nhận được một dãy mới.
- Gọi $f1, f2$ là hai dãy mới được tạo thành từ việc chuyển đổi hai số hạng đầu tiên của dãy *Fibonacci*, và dãy $f3$ được xác định như sau $f3_i = f1_i + f2_i \ (i \geq 1)$ thì dãy $f3$ vẫn tuân theo công thức truy hồi $f_n = f_{n-1} + f_{n-2}$.

Sau khi sử dụng các tính chất trên, bài toán trở thành một hoạt động rất cơ bản của [cây phân đoạn](#) (Cây *IT - Interval Tree / Segment Tree*). Với mỗi nút của cây phân đoạn lưu lại hai giá trị đầu tiên của dãy. Bạn có thể tham khảo code **không** sử dụng phương pháp nhân ma trận tại [đây](#) để hiểu rõ hơn về cách cập nhật cây phân đoạn.

Ở bài viết này, tôi sẽ sử dụng phương pháp **nhân ma trận** kết hợp với cây phân đoạn để giải quyết bài toán. Với mỗi nút của cây sẽ lưu lại ma trận hệ số của dãy *Fibonacci*.

Cài đặt

Lưu ý: Trong cách cài đặt sau, các hàng và cột của ma trận được đánh số bắt đầu từ 0 để thuận tiện cho việc xử lí.

```
#include <bits/stdc++.h>

using namespace std;

const int mod = 1e9 + 9;

struct Matrix {
    static const int size = 2;
    int row, col;
    int data[size][size];

    Matrix(){
        row = col = size;
        for (int i = 0; i < row; ++i) fill_n(data[i], col, 0);
    };

    auto & operator [] (int i) { return data[i]; }

    const auto & operator[] (int i) const { return data[i]; }

    // Phép cộng ma trận
    Matrix operator + (const Matrix &b) {
        Matrix a = *this;
        for (int i = 0; i < a.row; ++i)
            for (int j = 0; j < a.col; ++j)
                a[i][j] = (a[i][j] + b[i][j]) % mod;
        return a;
    }

    Matrix operator * (const Matrix &b) {
        Matrix a = *this, c;
        for (int i = 0; i < a.row; ++i)
            for (int j = 0; j < b.col; ++j)
                for (int k = 0; k < a.col; ++k) {
                    c[i][j] += 1ll * a[i][k] * (b[k][j] % mod) % mod;
                    c[i][j] %= mod;
                }
    }
};
```

```

    }
    return c;
}

// Kiểm tra xem tất cả phần tử của ma trận có bằng 0 hay không
bool iszero() {
    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)
            if (data[i][j]) return false;

    return true;
}
};

const int maxN = 3e5 + 10;

int n, m;
int a[maxN];
int st[4 * maxN];
Matrix lazy[4 * maxN], base_pow[4 * maxN];

void build(int id, int l, int r) {
    if (l == r) {
        st[id] = a[l];
        return;
    }
    int mid = (l + r) >> 1;
    build(id << 1, l, mid);
    build(id << 1 | 1, mid + 1, r);
    st[id] = (st[id << 1] + st[id << 1 | 1]) % mod;
}

void fix(int id, int l, int r) {
    if (lazy[id].iszero()) return;

    long long a = lazy[id][0][1];
    long long b = lazy[id][0][0];
    int tmp = (r - l + 1) + 2;
    st[id] += (b * base_pow[tmp - 1][0][1] + a * base_pow[tmp - 2][0][1] - b) % mod;
    st[id] %= mod;

    if (l != r) {
        int mid = (r - l) >> 1;
        lazy[id << 1] = lazy[id << 1] + lazy[id];
        lazy[id << 1 | 1] = lazy[id << 1 | 1] + lazy[id] * base_pow[mid + 1];
    }
    lazy[id] = Matrix();
}

void update(int id, int l, int r, int u, int v) {
    fix(id, l, r);

```



```

89     if (l > v || r < u) return;
90     if (l >= u && r <= v) {
91         lazy[id] = lazy[id] + base_pow[l - u + 1];
92         fix(id, l, r);
93         return;
94     }
95     int mid = (l + r) >> 1;
96     update(id << 1, l, mid, u, v);
97     update(id << 1 | 1, mid + 1, r, u, v);
98     st[id] = (st[id << 1] + st[id << 1 | 1]) % mod;
99 }
100
101 int get(int id, int l, int r, int u, int v) {
102     fix(id, l, r);
103     if (l > v || r < u) return 0;
104     if (l >= u && r <= v) return st[id];
105
106     int mid = (l + r) >> 1;
107     int g1 = get(id << 1, l, mid, u, v);
108     int g2 = get(id << 1 | 1, mid + 1, r, u, v);
109     return (g1 + g2) % mod;
110 }
111
112 main() {
113     cin >> n >> m;
114     for (int i = 1; i <= n; ++i) cin >> a[i];
115     build(1, 1, n);
116
117     // Xây dựng lũy thừa ma trận hệ số của dãy Fibonacci
118     base_pow[1][0][0] = base_pow[1][0][1] = base_pow[1][1][0] = 1;
119     for (int i = 2; i <= n + 2; ++i)
120         base_pow[i] = base_pow[i - 1] * base_pow[1];
121
122     while (m--) {
123         int t, l, r;
124         cin >> t >> l >> r;
125         if (t == 1) update(1, 1, n, l, r);
126         else cout << get(1, 1, n, l, r) << '\n';
127     }
128 }

```

Đánh giá

Ở thuật toán này, ta sử dụng mảng tĩnh để lưu ma trận thay vì sử dụng mảng động (Vector) như những bài toán trước. Vì số lượng ma trận phải lưu lên đến $4 \times n$ nên việc khai báo mảng động sẽ khiến thuật toán bị quá thời gian.

Độ phức tạp

Với mỗi truy vấn, ta sẽ mất độ phức tạp $\mathcal{O}(\log N)$ cho các thao tác trên cây phân đoạn. Và ta cũng mất thêm $\mathcal{O}(2^2)$ và $\mathcal{O}(2^3)$ cho các phép cộng và phép nhân ma trận. Nhìn chung, độ phức tạp của thuật toán là $\mathcal{O}(m \times \log N \times 2^3)$.

Ví dụ 6

Phép nhân ma trận cộng tối thiểu (Min-plus matrix multiplication)

Tham khảo: [Min-plus matrix multiplication](#) 

Nhận thấy rằng, ta hoàn toàn có thể thay thế phép nhân và phép cộng trong định nghĩa phép nhân ma trận, chỉ cần đảm bảo giữ nguyên tính chất kết hợp. Cụ thể hơn, với A và B là hai ma trận vuông cấp n , thay vì $C_{ij} = \sum_{k=1}^n A_{ik}B_{kj}$, ta có thể định nghĩa phép "nhân ma trận" mới như sau: $C_{ij} = \min_{k=1}^n (A_{ik} + B_{kj})$. Nó còn được gọi là **phép nhân ma trận cộng tối thiểu** hay **tích ma trận khoảng cách**.

Từ đó, ta có thể thu được một lớp các bài toán khác. Sau đây là một ví dụ minh họa cho nhóm các bài toán này.

Bài toán

[CSES - Graph Paths II](#) 

Cho đồ thị có hướng có trọng số gồm N đỉnh và M cạnh. Hãy tìm đường đi ngắn nhất xuất phát từ đỉnh 1 và kết thúc tại đỉnh N đi qua chính xác k cạnh.

- $N \leq 100$
- $1 \leq M \leq N(N - 1)$
- $1 \leq k \leq 10^9$

Phân tích

Gọi ma trận $C(k)$ kích thước $N \times N$, với $C(k)[i, j]$ là độ dài đường đi ngắn nhất từ i đến j đi qua đúng k cạnh.

Xét ma trận A là ma trận kề của đồ thị đã cho. Ta có:

- $C(1) = A$
- $C(2)[i, j] = \min(A[i, u] + A[u, j])$ với $1 \leq u \leq N$
- $C(k)[i, j] = \min(C(k-1)[i, u] + A[u, j])$ với $1 \leq u \leq N$

Như vậy, nếu ta thay phép nhân và phép cộng trong nhân ma trận thông thường lần lượt bởi phép cộng và phép lấy \min , ta thu được một phép "nhân ma trận" mới, ký hiệu là \star , thì:

$$C(1) = A$$

$$C(2) = C(1) \star C(1) = A \star C(1)$$



$$C(3) = C(1) \star C(2) = A \star C(2)$$

$$C(4) = C(1) \star C(3) = A \star C(3)$$

...

$$C(k) = C(1) \star C(k-1) = A \star C(k-1)$$

Do đó, $C(k) = A^k$

Như vậy, bài toán được đưa về bài toán tính lũy thừa của một ma trận, ta hoàn toàn có thể giải tương tự các ví dụ trước. Cài đặt phép nhân ma trận mới này hoàn toàn không phức tạp hơn cài đặt phép nhân ma trận thông thường. Việc cài đặt xin nhường lại cho bạn đọc.

Phép toán kết hợp và độ phức tạp tính toán

Nhân tổ hợp dãy ma trận

Trong phần [Cài đặt](#), ta đã có thuật toán nhân hai ma trận A kích cỡ $(m \times n)$ và B kích cỡ $(n \times p)$ cần độ phức tạp $\mathcal{O}(m \times n \times p)$. Giả sử ta có thêm ma trận C có kích cỡ $(p \times q)$ và ta cần tính tích $A \times B \times C$. Xét hai cách thực hiện phép nhân này:

- **Cách 1:** $(A \times B) \times C$ thực hiện nhân A và B rồi nhân với C cần độ phức tạp $\mathcal{O}(m \times n \times p) + \mathcal{O}(m \times p \times q) = \mathcal{O}(m \times p \times (n + q))$.
- **Cách 2:** $A \times (B \times C)$ thực hiện nhân B và C rồi nhân với A cần độ phức tạp $\mathcal{O}(n \times p \times q) + \mathcal{O}(m \times n \times q) = \mathcal{O}(n \times q \times (m + p))$.

Như vậy là hai cách thực hiện khác nhau cần hai độ phức tạp khác nhau. Ví dụ:

- Cho $m = n = 500, p = 1000, q = 2$. *Cách 1* sẽ cần tới $500 \times 1000 \times (500 + 2) = 251 \times 10^6$ phép tính, trong khi *cách 2* chỉ cần $500 \times 2 \times (500 + 1000) = 1.5 \times 10^6$ phép tính, nghĩa là *cách 1* chậm hơn *cách 2* tới gần 200 lần.

Khi độ dài của dãy ma trận tăng lên, sự khác biệt có thể còn lớn hơn nữa. Ví dụ trên đã cho thấy rằng trong một số trường hợp thứ tự thực hiện phép nhân ma trận có ý nghĩa rất lớn đối với việc tìm lời giải của các bài toán.

Trong thực tế, bài toán xác định thứ tự nhân ma trận hiệu quả nhất là một bài toán rất phổ biến, bạn có thể tìm đọc chi tiết thêm tại [đây](#) hoặc ở [Phần 3 mục 3.5 Phép Nhân Tổ Hợp dãy Ma Trận trong sách Giải thuật và lập trình của thầy Lê Minh Hoàng](#).

Giải thuật Freivalds kiểm tra tích hai ma trận

[Giải thuật Freivalds](#) là một ví dụ điển hình về việc áp dụng thứ tự thực hiện phép nhân ma trận để giảm độ phức tạp tính toán của phép nhân một dãy ma trận. Bài toán đặt ra là cho ba ma trận vuông A, B, C có kích cỡ $N \times N$ với $N < 1000$. Ta cần kiểm tra xem C có phải là tích của A và B , nói cách khác ta cần kiểm tra $A \times B = C$ có phải là mệnh đề đúng hay không (đây chính là bài [VMATRIX - VNOI Marathon 2014](#)).

Phân tích

Cách làm thông thường là nhân trực tiếp hai ma trận A, B rồi so sánh kết quả với C . Như đánh giá trong phần [Cài đặt](#), độ phức tạp của cách làm này là $\mathcal{O}(N^3)$, với $N = 1000$ thì cách làm này không đủ nhanh. Giải thuật Freivalds thực hiện việc kiểm tra thông qua thuật toán xác suất kiểu [Monte Carlo](#) [☑](#) với k lần thử cho xác suất kết luận sai là xấp xỉ 2^{-k} , mỗi lần thử có độ phức tạp $\mathcal{O}(N^2)$. Các bước cơ bản của một phép thử Freivalds như sau:

1. Sinh ngẫu nhiên một ma trận v kích cỡ $(N \times 1)$ với các phần tử chỉ nhận giá trị 0 hoặc 1.
2. Tính hiệu $P = A \times B \times v - C \times v$. Dễ thấy rằng P là ma trận kích cỡ $N \times 1$.
3. Trả về [True](#) nếu P chỉ gồm phần tử 0 (bằng với vector 0) và [False](#) nếu ngược lại.

Bạn có thể tìm hiểu thêm về **phép cộng trừ ma trận** tại [đây](#) [☑](#).

Ta thực hiện k lần thử, nếu gặp phép thử trả về [False](#) thì ta kết luận là $A \times B \neq C$. Ngược lại nếu sau k phép thử mà luôn thấy [True](#) thì ta kết luận $A \times B = C$. Vì xác suất lỗi giảm theo hàm mũ của k nên thông thường chỉ cần chọn k vừa đủ là sẽ thu được xác suất đúng rất cao ($k = 5$ với bài **VMATRIX** ở trên). Một nhận xét quan trọng khác là cận trên của đánh giá xác suất kiểm tra lỗi không phụ thuộc vào kích cỡ N của ma trận được cho mà chỉ phụ thuộc vào số lần thực hiện phép thử.

Xét bước thứ 2, ta thấy rằng phép thử Freivalds chỉ có ý nghĩa nếu như ta có thể thực hiện phép nhân $A \times B \times v$ trong thời gian $\mathcal{O}(N^2)$ (vì phép nhân $C \times v$ đã đạt sẵn $\mathcal{O}(N^2)$ rồi). Thay vì thực hiện tuần tự từ trái qua phải sẽ cần $\mathcal{O}(N^3)$, ta thực hiện theo thứ tự $A \times (B \times v)$. Vì kết quả của phép nhân B và v là một ma trận $(N \times 1)$ nên độ phức tạp tổng cộng sẽ là $\mathcal{O}(N^2)$. Trên tất cả các phép thử, độ phức tạp là $\mathcal{O}(k \times N^2)$.

Cài đặt

Bài toán [VMATRIX - VNOI Marathon 2014](#) [☑](#)

Lưu ý: Trong cách cài đặt sau, các hàng và cột của ma trận được đánh số bắt đầu từ 0 để thuận tiện cho việc xử lí.

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  const int mod = 10;
6
7  using type = int;
8
9  struct Matrix {
10     vector <vector <type> > data;
11
12     int row() const { return data.size(); }
13
14     int col() const { return data[0].size(); }
15
16     auto & operator [] (int i) { return data[i]; }
17
18     const auto & operator[] (int i) const { return data[i]; }
19

```

```

20
21 Matrix() = default;
22
23 Matrix(int r, int c): data(r, vector <type> (c)) { }
24
25 Matrix(const vector <vector <type> > &d): data(d) { }
26
27 friend ostream & operator << (ostream &out, const Matrix &d) {
28     for (auto x : d.data) {
29         for (auto y : x) out << y << ' ';
30         out << '\n';
31     }
32     return out;
33 }
34
35 // Phép trừ ma trận
36 Matrix operator - (const Matrix &b) {
37     Matrix a = *this;
38
39     // Kiểm tra điều kiện phép trừ ma trận
40     assert(a.row() == b.row() && a.col() == b.col());
41
42     for (int i = 0; i < a.row(); ++i)
43         for (int j = 0; j < a.col(); ++j)
44             a[i][j] = (a[i][j] - b[i][j] + mod) % mod;
45     return a;
46 }
47
48 Matrix operator * (const Matrix &b) {
49     Matrix a = *this;
50     assert(a.col() == b.row());
51     Matrix c(a.row(), b.col());
52     for (int i = 0; i < a.row(); ++i)
53         for (int j = 0; j < b.col(); ++j)
54             for (int k = 0; k < a.col(); ++k){
55                 c[i][j] += a[i][k] % mod * (b[k][j] % mod) % mod;
56                 c[i][j] %= mod;
57             }
58     return c;
59 }
60 };
61
62 mt19937 rd(chrono::steady_clock::now().time_since_epoch().count());
63
64 int random(int l, int r) { return l + rd() % (r - l + 1); }
65
66 bool check(Matrix a, Matrix b, Matrix c, int n) {
67     int k = 5;
68     while (k--) {
69         Matrix v(n, 1);
70         for (int i = 0; i < n; ++i) v[i][0] = random(0, 1);

```

```










71         Matrix p = (a * (b * v)) - (c * v);
72         for (int i = 0; i < n; ++i)
73             if (p[i][0]) return false;
74     }
75     return true;
76 }
77
78 int main(){
79     int t;
80     cin >> t;
81     while (t--) {
82         int n;
83         cin >> n;
84         Matrix a(n, n), b(n, n), c(n, n);
85         for (int i = 0; i < 3 * n; ++i) {
86             string s;
87             cin >> s;
88             for (int j = 0; j < s.size(); ++j) {
89                 if (i / n == 0) a[i][j] = s[j] - '0';
90                 if (i / n == 1) b[i - n][j] = s[j] - '0';
91                 if (i / n == 2) c[i - n - n][j] = s[j] - '0';
92             }
93         }
94
95         if (check(a, b, c, n)) cout << "YES\n";
96         else cout << "NO\n";
97     }
98 }




```

Đánh giá

Ngoài thuật toán trên, ta vẫn có thể nhân trực tiếp 2 ma trận A, B rồi so sánh với C bằng cách sử dụng thuật toán nhân ma trận Strassen với độ phức tạp $\mathcal{O}(N^{\log_2 7}) = \mathcal{O}(1000^{\log_2 7}) \approx \mathcal{O}(2.6 \times 10^8)$. Tuy nhiên, cách cài đặt này phức tạp hơn.

Bài tập áp dụng

- [Codeforces - 1182E Product Oriented Recurrence](#) 
- [HackerEarth - PK and interesting language](#) 
- [HackerEarth - Long walks from Office to Home Sweet Home](#) 
- [HackerEarth - Tiles](#) 
- [HackerEarth - ABCD Strings](#) 
- [HackerEarth - Mehta and the difficult task](#) 
- [HackerEarth - Mehta and the Evil Strings](#) 
- [VNOJ - PA06ANT](#) 
- [VNOJ - ONE4EVER](#) 

- [VNOJ - CONNECTE](#) 
- [VNOJ - DHLOCO](#) 
- [VNOJ - FBRICK](#) 

Được cung cấp bởi [Wiki.js](#)