Khử nhân ma trận

Khử nhân ma trận

Tác giả: Nguyễn Tiến Trung Kiên

Đôi lời về tác giả:

Nguyễn Tiến Trung Kiên là cựu học sinh Chuyên Tổng Hợp, với 1 HCĐ IOI năm 2014 và 1 HCB IOI năm 2015. Kiên còn nổi tiếng với blog chứa code nhiều thuật toán và series Free contest .

Giới thiệu

Nhân ma trận thật sự hữu dụng. Có nhiều bài toán khi n nhỏ, ta dùng **DP** (**Dynamic Programming - Quy Hoạch Động**) để giải. Nhưng khi n lớn (khoảng 10^9), ta phải dùng nhân ma trận để giảm **độ phức tạp**. Trong quá trình code nhân ma trận, việc sinh ra ma trận gốc không phải lúc nào cũng đơn giản. Tôi đã tìm ra một phương pháp tốt để giải những bài toán này mà không cần nhân ma trận.

Khi dùng phương pháp này, ta không cần phải sinh ma trận gốc và không cần cài phép toán nhân hai ma trận $A*B$ và luỹ thừa ma trận A^k. Tuy nhiên, phương pháp này chỉ dùng được trong các bài toán đếm, nghĩa là nó không thể hoàn toàn thay thế nhân ma trận.

Bắt đầu bằng ví dụ đơn giản nhất

Để ví dụ, tôi sẽ dùng bài toán sau:

Đếm xem có bao nhiêu dãy ngoặc đúng độ dài n mà độ sâu không quá L. $(n \leq 10^9, L \leq 10)$.

Ví dụ, khi $n = 4$ và $L = 1$, thì `()()` là dãy ngoặc đúng duy nhất thoả mãn, còn `(())` , `(((` , và `))((` thì không thoả mãn.

Bài toán này có thể giải bằng phương pháp **Quy hoạch động** như sau:

- Nhận xét: Nếu ta đi qua lần lượt từng ký tự của dãy ngoặc và duy trì một biến `sum` : Khi gặp `(` ta tăng `sum` lên 1 đơn vị. Khi gặp `)` ta giảm `sum` đi 1 đơn vị. 1 dãy ngoặc là dãy ngoặc đúng nếu thỏa mãn 2 điều kiện sau:
  - Không có thời điểm nào `sum` nhỏ hơn 0
  - Đến cuối cùng, `sum` bằng 0.
- Đồng thời, nếu làm như trên, độ sâu của dãy ngoặc chính là giá trị tối đã của `sum` trong quá trình trên.

Từ nhận xét trên, ta tìm ra công thức $f(n, h) = f(n-1, h-1) + f(n-1, h+1)$ trong đó $f(n, h)$ là số dãy mà phần còn lại cần xây dựng có độ dài n và tổng hiện tại (sum) là h. Mục tiêu của chúng ta là tính $f(n, 0)$. Tất nhiên độ phức tạp của hàm f là quá lớn.

Bây giờ, gọi $f(n, h, h_0)$ là số dãy độ dài n bắt đầu từ tổng h và kết thúc tại tổng h_0.

Xét các trường hợp:

- Nếu $n = 0$: trả về 1 nếu $h = h_0$, trả về 0 nếu ngược lại.
- Nếu $n = 2*k$: $f(2*k, h, h_0) = \sum f(k, h, i)*f(k, i, h_0)$ với mọi i trong khoảng $[0, L]$.
- Nếu $n = 2*k + 1$: $f(2*k+1, h, h_0) = f(2*k, h-1, h_0) + f(2*k, h+1, h_0)$.

Ngoài ra, chú ý đến trường hợp sau: nếu $h < 0$ hoặc $h > L$ thì trả về 0.

Mục tiêu của ta là tính $f(n, 0, 0)$.

Độ phức tạp của phương pháp này là $\mathcal{O}(L^3 logn)$, nhanh bằng với nhân ma trận. Chú ý rằng ta chỉ có $\mathcal{O}(L^2 logn)$ trạng thái, không phải là $\mathcal{O}(L^2 n)$. Chẳng hạn khi $n = 100$, các giá trị của n sẽ nằm trong tập sau: $100, 50, 25, 24, 12, 6, 3, 2, 1, 0$. Thế nên n chỉ nhận khoảng $2 * logn$ giá trị trong tập hợp đó. Ta có thể dùng độ sâu của hàm f để đại diện cho n.

```
function f(n, h, h_0, Depth):
    if h < 0 or h > L:
      return 0
    if n == 0:
      return (h==h_0 ? 1 : 0)

    if Saved[h][h_0][Depth]:
      return Value[h][h_0][Depth]

    if n is even:
        Result = 0
        for i in 0..L:
          Result += f(n/2, h, i, Depth+1) * f(n/2, i, h_0, Depth+1)
    else:
        Result = f(n-1, h-1, h_0, Depth+1) + f(n-1, h+1, h_0, Depth+1)

    Saved[h][h_0][Depth] = true
    Value[h][h_0][Depth] = Result

input n, L
output f(n, 0, 0, 0)
```

Tổng quát

Với trường hợp $f(n, [a, b, c, \dots])$ được tính từ $f(n-1, [a, b, c, \dots])$

Có t loại hoa $(t \geq 4)$. 4 trong t loại hoa này là g (gerbera), o (orchid), a (azalea) và h (hydrangea). Ta dùng các loại hoa này để tạo một dãy n chậu hoa $(n \leq 10^9)$. Có vài điều kiện được đặt ra như sau:

- Một chậu h phải được đặt giữa một a và một o
- Giữa hai chậu g bất kì, phải có ít nhất p chậu hoa loại khác $(p \leq 20)$.

Giả sử có 5 loại hoa ($t = 5$): a , h , o , g , và b (begonias).

Với $n = 6$, có 2906 dãy chậu đúng, 5 trong số đó là aoaaoo , ahohag , gbbbgo , gbbbog , bbbbbb .

Những dãy sau đây không hợp lệ: ohoaha (đoạn aha không hợp lệ vì bên cạnh h phải có một o và một a), gogbao (giữa hai g phải có ít nhất 3 hoa khác), ahoaha (chậu h cuối cùng không kề với một a và một o).

Không khó lắm để tìm ra công thức quy hoạch động: $f(n, x, Just)$ trả về số dãy chậu đúng. Trạng thái n, x, $Just$ được mô tả như sau:

- n là độ dài còn lại phải xây dựng của dãy đang xây dựng.
- x là số chậu hoa ta vừa đặt mà khác g , nói cách khác tất cả các chậu hoa trong khoảng $n + 1$ đến $n + x$ không phải là g .
- $Just$ đại diện cho chậu hoa vừa đặt (tức là chậu $n + 1$). $Just = 1$ nghĩa là a hoặc o , $Just = 2$ nghĩa là h , $Just = 0$ nghĩa là các loại hoa còn lại (bao gồm g và $t - 4$ loại hoa khác).

Hàm quy hoạch động trên có thể chạy với $n \leq 10,000$.

Bây giờ tôi sẽ nói cách giải đúng. Gọi $f(n, p, Just, p_0, Just_0)$ nghĩa là: ta xuất phát từ trạng thái $(n, p, Just)$, có bao nhiêu cách đi đến trạng thái $(0, p_0, Just_0)$.

```
long f(int n, int x, int Just) {
    if (x>=p) x=p;
    if (Just==2) {
        if (n==0) return 0;
        return f(n-1, x+1, 1);
    } else {
        if (n==0) return 1;
        if (F[x][Just].count(n)) return F[x][Just][n];
        long Sum = f(n-1, x+1, 1) * 2;
        if (Just==1) Sum += f(n-1, x+1, 2);
        if (x>=p) Sum += f(n-1, 0, 0);
        Sum += f(n-1, x+1, 0) * (t-4);
        return F[x][Just][n] = Sum % M;
    }
}

cout << f(n, ::p, 0) << endl;
```

Ta có các trường hợp:

- Nếu $n = 0$ hoặc $n = 2*k + 1$, ta viết như hàm f cũ. Nếu $n \neq 0$, nó sẽ gọi đến một trạng thái khác mà lúc này n chẵn.
- Ngược lại, $n = 2*k$, $f(2*k, p, Just, p_0, Just_0) = \sum f(k, p, Just, i, j)*f(k, i, j, p_0, Just_0)$ với tất cả bộ i, j hợp lệ (tức là i nằm trong khoảng [0, p], j nằm trong khoảng $[0, 2]$).

Chú ý tại trường hợp $n = 0$, việc $n = 0$ không có nghĩa đó là kết thúc của một dãy. Vì ta chia dãy thành các phần nhỏ hơn, $n = 0$ chỉ có nghĩa là kết thúc của một phần nhỏ. Vì thế ta sẽ thêm một biến $Stop$ thuộc kiểu boolean. Khi $Stop = true$, $f(n, p, Just, p_0, Just_0) = f(n, p, Just)$, ngược lại, tức là $Stop = false$, $f(n, p, Just, p_0, Just_0, Stop) = f(n, p, Just, p_0, Just_0)$.

```
map<int, int> G[21][3][21][3][2];
#define C p][Just][p0][Just0][Stop

long g(int n, int p, int Just, int p0, int Just0, bool Stop) {
    if (p>=::p) p=::p;
    if (n%2==1 || n==0) {
        if (Just==2) {
            if (n==0) return Stop ? 0 : p==p0 && Just==Just0;
            return g(n-1, p+1, 1, p0, Just0, Stop);
        } else {
            if (n==0) return Stop ? 1 : p==p0 && Just==Just0;
            if (G[C].count(n)) return G[C][n];
            long Sum = g(n-1, p+1, 1, p0, Just0, Stop) * 2;
            if (Just==1) Sum += g(n-1, p+1, 2, p0, Just0, Stop);
            if (p>=::p) Sum += g(n-1, 0, 0, p0, Just0, Stop);
            Sum += g(n-1, p+1, 0, p0, Just0, Stop) * (t-4);
            return G[C][n] = Sum % M;
        }
    } else {
        if (G[C].count(n)) return G[C][n];
        long Sum = 0;
        for (int i=0; i<=::p; i++)
        for (int k=0; k<=2; k++) {
            long G1 = g(n/2, p, Just, i, k, false);
             long G2 = g(n/2, i, k, p0, Just0, Stop);
            Sum += G1*G2;
        }
        return G[C][n] = Sum % M;
     }
}

cout << g(n, ::p, 0, rand()%21, rand()%3, true) << endl;
```

Chú ý ở code trên, `::p` và `p` là khác nhau. `::p` là biến `p` toàn cục, tức là `p` được nhập từ input. Còn `p` là tham số ở trong hàm `g`. `Rand()%21` và `rand()%3` là hai số mà ta có thể bỏ qua giá trị của

chúng (khi nào mà `Stop=true` thì `p0` và `Just0` không có ý nghĩa).

Độ phức tạp ở code trên là $\mathcal{O}(p^3log^2n)$. Thực tế, ta có thể không dùng `map`, bằng cách thêm một tham số là `Depth` đại diện cho độ sâu của hàm quy hoạch động. Khi đó, độ phức tạp mất đi một thừa số $logn$, giảm xuống còn $\mathcal{O}(p^3logn)$. Code trên tôi dùng `map` cho nó dễ hiểu.

$f(n) = f(n-1) + f(n-2)$

Bây giờ, chúng ta sẽ tính số fibonacci thứ 10^9 (trong một modulo nào đó). Chắc hẳn là bạn đã biết cách dùng nhân ma trận, nó khá dễ. Tuy nhiên, bây giờ chúng ta sẽ thử giải bằng cách không dùng nhân ma trận. Xem bài toán sau:

Bạn đang đứng ở điểm n trên trục Ox. Mỗi bước, bạn có thể di chuyển sang trái 1 hoặc 2 bước. Có bao nhiêu cách để bạn đi tới vị trí 0?

Không khó để nhận ra $f(n) = f(n-1) + f(n-2)$, trong đó $f(0) = 1$ và $f(1) = 1$. Thế nên, $f(n)$ là số fibonacci thứ $n+1$.

Có hai trường hợp:

- $n = 2*k$, ta có hai lựa chọn:
  - Lựa chọn thứ nhất là nhảy từ $2*k$ đến k rồi nhảy từ k đến 0.
  - Lựa chon thứ hai là nhảy từ $2*k$ đến $k+1$, sau đó di chuyển sang trái 2 bước, tức là từ $k+1$ đến $k-1$, rồi nhảy từ $k-1$ đến 0 (chú ý ta không hề nhảy vào ô thứ k).
    Thế nên, $f(2*k) = f(k)*f(k) + f(k-1)*f(k-1)$.
- $n = 2*k+1$, bây giờ ta chia dãy thành hai đoạn $0..k$ và $k..n$ (đoạn thứ nhất độ dài $k+1$, đoạn thứ hai dài k), ta lại có hai lựa chọn:
  - Lựa chọn thứ nhất là nhảy từ n đến k rồi nhảy từ k đến 0.
  - Lựa chọn thứ hai là nhảy từ n đến $k+1$, di chuyển sang trái 2 bước, rồi nhảy từ $k-1$ đến 0. Thế nên $f(2*k+1) = f(k)*f(k+1) + f(k-1)*f(k)$.

Lúc này độ phức tạp là $\mathcal{O}(logn)$. Bởi vì với mỗi độ sâu, chỉ có tối đa 4 giá trị n.

```
map<long, long> F;
F[0]=F[1]=1;

long f(long n) {
     if (F.count(n)) return F[n];
     long k=n/2;
     if (n%2==0) { // n=2*k
         return F[n] = (f(k) * f(k) + f(k-1) * f(k-1)) % M;
     } else { // n=2*k+1
         return F[n] = (f(k) * f(k+1) + f(k-1) * f(k)) % M;
     }
}
```