斐波那契数列

Josh-Cena

2020 年 10 月 10 日

斐波那契数列：

$$F_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F_{n-2} + F_{n-1}, & n > 1 \end{cases}$$

给定 n，求 $F_n \bmod 10^9 + 7$。

数据规模	内存限制	运行时间
$0 \le n \le 10^{19}$	64 MB	1.0 s

题解. 10^{19} 显然灭掉了所有用循环解决的想法。有没有比简单的 $\mathcal{O}(n)$ 更好一点的方法？用**矩阵快速幂**，可以达到 $\mathcal{O}(\log n)$。观察到：

$$\begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n + F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

这一步对于所有递推数列都是适用的，因此在有经验之后应该非常容易得到。一般地，对于 $F_{n+2} = aF_n + bF_{n+1}$，有

$$\begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ aF_n + bF_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ a & b \end{pmatrix} \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

从递推式中有

$$\begin{pmatrix} F_{n+m} \\ F_{n+m+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^m \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

取 $n=0$，得到

$$\begin{pmatrix} F_m \\ F_{m+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^m \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

因此把问题转化成了如何求矩阵 m 次方的问题。如果设 $m = 2^0 a_0 + 2^1 a_1 + 2^2 a_2 + \dots$（也就是把 m 用二进制表示），那么有

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^m = \left(\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^1\right)^{a_0} \times \left(\left(\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2\right)^{a_1} \times \left(\left(\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^4\right)^{a_2} \dots\right.\right.$$

而这些矩阵的 2^k 次方，完全可以预处理。当 m 的数量级为 10^{19} 时，$k < \log_2 10^{19} < 64$，最多只需要存储 63 个矩阵。并且

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{2^k} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{2^{k-1}} \times \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{2^{k-1}}$$

这些乘方可以在 $\mathcal{O}(\log m)$ 时间内得到。这便是快速幂的思想：计算所有的 2^k 次方，然后把其中需要的那些组合起来即可。

下面是 C++ 代码，其中最繁琐的部分是实现矩阵乘法：

```cpp
#include <iostream>
#include <cmath>

using namespace std;

struct mat {
   unsigned long long a[4];
   mat operator *(mat o) {
      mat t;
      t.a[0] = (this->a[0] * o.a[0] + this->a[1] * o.a[2]) % 1000000007;
      t.a[1] = (this->a[0] * o.a[1] + this->a[1] * o.a[3]) % 1000000007;
      t.a[2] = (this->a[2] * o.a[0] + this->a[3] * o.a[2]) % 1000000007;
      t.a[3] = (this->a[2] * o.a[1] + this->a[3] * o.a[3]) % 1000000007;
      return t;
   }
};

// 预处理的矩阵 2^k 次幂
mat mat_pow[64];
```

```cpp
int fib(unsigned long long k){
   // 临时矩阵，每次在此上面乘以 mat_pow 中的某项
   mat tmp;
   tmp.a[0] = 1;
   tmp.a[1] = 0;
   tmp.a[2] = 0;
   tmp.a[3] = 1;
   for (int i = 0; i < 64; i++) {
      // 如果 a_i 为 1
      if (k & (1ull << i)) {
         tmp = tmp * mat_pow[i];
      }
   }
   return tmp.a[1];
}

int main(){
   mat_pow[0].a[0] = 0;
   mat_pow[0].a[1] = 1;
   mat_pow[0].a[2] = 1;
   mat_pow[0].a[3] = 1;
   for (int i = 1; i < 64; i++) {
      mat_pow[i] = mat_pow[i-1] * mat_pow[i-1];
   }
   unsigned long long n;
   cin >> n;
   cout << fib(n) << endl;
   return 0;
}
```

补充一下矩阵的乘法公式：

$$\begin{pmatrix} a_0 & a_1 \\ a_2 & a_3 \end{pmatrix} \times \begin{pmatrix} b_0 & b_1 \\ b_2 & b_3 \end{pmatrix} = \begin{pmatrix} a_0b_0 + a_1b_2 & a_0b_1 + a_1b_3 \\ a_2b_0 + a_3b_2 & a_2b_1 + a_3b_3 \end{pmatrix}$$

$$\begin{pmatrix} a_0 & a_1 \\ a_2 & a_3 \end{pmatrix} \times \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = \begin{pmatrix} a_0b_0 + a_1b_1 \\ a_2b_0 + a_3b_1 \end{pmatrix}$$