我们为什么要考斐波那契数列?

一道上机笔试题的解析，兼谈技术面试与编程基本功

凤凰木

May 29, 2020

大纲

- 题目
- 失败的案例
- 参考解法
  - 第 1 问
  - 第 2 问
  - 第 3 问
  - 第 4 问
- 思考题

题目

笔试规则

- 时间不限，语言不限，使用的工具不限;
- 可以 Google 可以百度，但不可向他人求助;
- 每个问题的代码运行时间应该不超过 1 分钟;
- 从 6 个大问题 14 个小问题中任选 N 个小问题解决;
- 如果能成功解决了 2 个或 2 个以上小问题，则进入下一轮技术终面，否则面试流程结束。

题目

在数学上，著名的斐波那契数列以递归的方法来定义：

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

1. 试算出 F_{100} 的准确整数值。
2. 请问斐波那契数列中，第一个有 1000 位数字的是第几项?
3. 试算出第 10 亿项除以 1000000007 的余数，即 $F_{10^9} \; mod \; (10^9 + 7)$.
4. 试算出第 10^{1000} 项除以 1000000007 的余数，即 $F_{10^{1000}} \; mod \; (10^9 + 7)$.

质疑

这道题目以后，收到很多质疑：

- 题目太简单了，一个循环就出来了。
- 题目太简单了，一个递归就出来了。
- 题目太简单了，一个矩阵快速幂就全秒了。
- 题目太难了，搬转工程师不应该考算法。
- 题目太难了，不应该考数学。
- 题目太难了，不应该考递归。

如果考算法

经典小学奥数题:

某人上台阶，可以一步上一级，也可以上两级。请问上 100 级的台阶，一共有多少种走法?

如果考算法

经典递归算法题:

用一些 1×2 的砖，铺满 $2 \times n$ 的矩形区域，一共有多少种铺法?

失败的案例

直接递归，卒

典型：

```python
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

直接递归，卒

简洁型，一行：

```python
def fib(n): return fib(n-1) + fib(n-2) if (n > 2) else n
```

直接递归，卒

- 直接接递归，无论机器多猛，一般在 $n = 45$ 左右就是极限了；
- $n = 100$ 递归到太阳熄灭都不会有结果；
- 一般对算法的复杂度没有概念，仅仅会用简单的递归函数；
- 我们是重度使用 Scala 函数式编程的团队，最怕的就是这种写法；
- 把第一个小问题设计为 $n = 100$, 就是为了暴露这个问题。

不能正确处理大数运算，卒

Java:

```java
public class Fib {

    public static long fib(int n) {
        long a = 0L, b = 1L, c = 1L;
        for (int i = 1; i <= n; i++) {
            c = b;
            b = a + b;
            a = c;
        }
        return c;
    }

    public static void main(String args[]) {
        System.out.println(fib(100)); // 3736710778780434371
    }
}
```

不能正确处理大数运算，卒

JavaScript:

```
function fib(n) {
    for (i = 0, x = 0, y = 1; i < n; i++) {
        var z = y
        y = x + y
        x = z
    }
    return x
}

console.log(fib(100)) // 354224848179262000000
```

不能正确处理大数运算，卒

- 有人抱怨这里故意埋大整数溢出的坑考他们。
- 能够正确地处理比较明显的数值溢出，是每一位程序员应该具备的基本素质。
- 就算考虑实际的项目，很多时候仍然需要注意数值的溢出。
  - 曾经把某些业务数据的统计值类型错误地设计成 32 位整数，导致溢出。
  - 广告业务，一张月报表的广告请求数，很容易超过 21 亿的。
  - 直接把所有统计值的类型设计成 64 位整数行不行?

不能正确处理大数运算，卒

- 有不少人选意识到了大整数溢出的问题，会直接用数组之类的实现大数加法。
- 严格来说，也可以归结为一种“错误”。
- 不熟悉自己使用的编程语言，重复制造无意义的轮子。

不能正确处理大数运算，卒

- 第一个小问题设计成 $n = 100$ 而不成 $n = 50$，
- 制造大于 64 位无符号整数的结果。
- 用 Python 做题的人可能在毫无意识的情况下避开了这个坑。

直接用通项公式计算，卒

$$F_n = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]$$

直接用通项公式计算，卒

- 一般是算不出正确答案，为什么?
- 犯这个错的人，一般数学比较好，但对工程意义上的“计算”没有概念的;
- 我们不考查数学能力，即使知道了这个公式也没什么作用;

两个数加来加去加错，卒

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        (a, b) = (1, 1)
        for i in range(n):
            b = a + b
            a = b
        return b

fib(10) # 1024 WTF!!!!
```

两个数加来加去加错，卒

- 这类错误难以一一列举;
- 基本属于久没写代码，或者不会写代码;
- 一想好简单，一写就错;
- 一错就说题目难。

其他错误

- 一上来就用动态规划 (DP)，但是实现错误的
  - 实际上这个问题不属于典型的动态规划问题,
  - 最多只能算是动态规划的极端版,
  - 一般刷过算法题,
  - 但对算法理解过浅,
  - 不能处理实际工程项目里更加普遍更加复杂的算法问题。
- 一上来就祭出矩阵快速幂，因姿势不对而失败。这种一般是
  - 理论知识较好,
  - 但工程能力稍欠,
  - 实现已有算法能力不够强

参考解法

直接循环：简单粗暴，快速有效

需要解决的问题：

- 计算逻辑；
  - 要求用笔死算，90% 以上没问题；
  - 要求写代码算，90% 以上凉凉！
- 大数运算。
  - Scala 有 BigInt,
  - Java 有 BigInteger,
  - JavaScript 有 BigInt,
  - Python 默认的 Int 就是，
  - Haksell 有 Integer

直接循环：简单粗暴，快速有效

Scala:

```scala
object FibLoop extends App {
  def fib(n: Int) = {
    var (a, b, c) = (BigInt(0), BigInt(1), BigInt(1))
    var i = 1
    while (i <= n) {
      c = b
      b = a+b
      a = c
      i += 1
    }
    a
  }
  println(fib(100)) // 354224848179261915075
}
```

直接循环：简单粗暴，快速有效

Java:

```java
import java.math.BigInteger;

public class FibJava {

    public static BigInteger fib(int n) {
        BigInteger a = BigInteger.valueOf(0L);
        BigInteger b = BigInteger.valueOf(1L);
        BigInteger c = BigInteger.valueOf(1L);
        for (int i = 1; i <= n; i++) {
            c = b;
            b = a.add(b);
            a = c;
        }
        return a;
    }

    public static void main(String args[]) {
        System.out.println(fib(100)); // 354224848179261915075
    }
}
```

直接循环：简单粗暴，快速有效

Python:

```python
def fib(n):
    a, b = 0, 1
    i = 1
    while (i <= n):
        a, b = b, a + b
        i = i + 1
    return a

print(fib(100)) # 354224848179261915075
```

直接循环：简单粗暴，快速有效

JavaScript:

```javascript
function fib(n) {
    for (i = 0, x = 0n, y = 1n; i < n; i++) {
        y = x + y
        x = y - x
    }
    return x
}

console.log(fib(100)) // 354224848179261915075n
```

直接循环：简单粗暴，快速有效

Haskell:

```
-- 本人不会！
```

递归：函数式的解

Scala:

```scala
object FibTailRec extends App {
  def fib(n: Int) = {
    @scala.annotation.tailrec
    def f(x: BigInt, y: BigInt, i: Int): BigInt =
      if (i == 0) x else f(y, x + y, i - 1)
    f(0, 1, n)
  }
  (1 to 100).map(x => (x, fib(x))).foreach(println)
}
```

递归：函数式的解

Java:

```java
import java.math.BigInteger;

public class FibJava {

    private static BigInteger _fib(BigInteger x, BigInteger y, int i) {
        return (i == 0) ? x : _fib(y, x.add(y), i - 1);
    }

    public static BigInteger fib(int n) {
        return _fib(BigInteger.valueOf(0L), BigInteger.valueOf(1L), n);
    }

    public static void main(String args[]) {
        System.out.println(fib(100)); // 354224848179261915075
    }
}
```

递归：函数式的解

Python:

```python
def fib(n):
    def f(x, y, i): return x if not i else f(y, x + y, i-1)
    return f(0, 1, n)

print(fib(100)) # 354224848179261915075
```

递归：函数式的解

JavaScript

```javascript
'use strict';

function fib(n) {
    let f = (x, y, i) => (i === 0) ? x : f(y, x+y, i-1)
    return f(0n, 1n, n)
}

console.log(fib(100)) // 354224848179261915075n
```

递归：函数式的解

Haskell:

```haskell
fib n = fib' 0 1 n where
    fib' x y 0 = x
    fib' x y i = fib' y (x + y) (i-1)

main = do print $ fib 100 -- 354224848179261915075
```

其他解法

Haskell 著名的斐波那契数列生成式:

```haskell
fibs = 1 : 1 : (zipWith (+) fibs (tail fibs))
```

```
~ $ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
λ> fibs = 1 : 1 : (zipWith (+) fibs (tail fibs))
λ> take 100 $ fibs
   [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
    2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
    317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465,
    14930352, 24157817, 39088169, 63245986, 102334155, 165580141, 267914296,
    433494437, 701408733, 1134903170, 1836311903, 2971215073, 4807526976,
    7778742049, 12586269025, 20365011074, 32951280099, 53316291173,
    86267571272, 139583862445, 225851433717, 365435296162, 591286729879,
    956722026041, 1548008755920, 2504730781961, 4052739537881, 6557470319842,
    10610209857723, 17167680177565, 27777890035288, 44945570212853,
    72723460248141, 117669030460994, 190392490709135, 308061521170129,
    498454011879264, 806515533049393, 1304969544928657, 2111485077978050,
    3416454622906707, 5527939700884757, 8944394323791464, 14472334024676221,
    23416728348467685, 37889062373143906, 61305790721611591,
    99194853094755497, 160500643816367088, 259695496911122585,
    420196140727489673, 679891637638612258, 1100087778366101931,
    1779979416004714189, 2880067194370816120, 4660046610375530309,
    7540113804746346429, 12200160415121876738, 19740274219868223167,
    31940434634990099905, 51680708854858323072, 83621143489848422977,
    135301852344706746049, 218922995834555169026, 354224848179261915075]
λ> head $ drop 99 fibs
   354224848179261915075
```

其他解法

```
λ> print $ take 140 $ zip [1..] fibs
[(1,1),(2,1),(3,2),(4,3),(5,5),(6,8),(7,13),(8,21),(9,34),(10,55),(11,89),(12,144),
(13,233),(14,377),(15,610),(16,987),(17,1597),(18,2584),(19,4181),(20,6765),(21,10946),
(22,17711),(23,28657),(24,46368),(25,75025),(26,121393),(27,196418),(28,317811),(29,514229),
(30,832040),(31,1346269),(32,2178309),(33,3524578),(34,5702887),(35,9227465),(36,14930352),
(37,24157817),(38,39088169),(39,63245986),(40,102334155),(41,165580141),(42,267914296),(43,433494437),
(44,701408733),(45,1134903170),(46,1836311903),(47,2971215073),(48,4807526976),(49,7778742049),
(50,12586269025),(51,20365011074),(52,32951280099),(53,53316291173),(54,86267571272),(55,139583862445),
(56,225851433717),(57,365435296162),(58,591286729879),(59,956722026041),(60,1548008755920),
(61,2504730781961),(62,4052739537881),(63,6557470319842),(64,10610209857723),(65,17167680177565),
(66,27777890035288),(67,44945570212853),(68,72723460248141),(69,117669030460994),(70,190392490709135),
(71,308061521170129),(72,498454011879264),(73,806515533049393),(74,1304969544928657),(75,2111485077978050),
(76,3416454622906707),(77,5527939700884757),(78,8944394323791464),(79,14472334024676221),(80,23416728348467685),
(81,37889062373143906),(82,61305790721611591),(83,99194853094755497),(84,160500643816367088),
(85,259695496911122585),(86,420196140727489673),(87,679891637638612258),(88,1100087778366101931),
(89,1779979416004714189),(90,2880067194370816120),(91,4660046610375530309),(92,7540113804746346429),
(93,12200160415121876738),(94,19740274219868223167),(95,31940434634990099905),(96,51680708854858323072),
(97,83621143489848422977),(98,135301852344706746049),(99,218922995834555169026),(100,354224848179261915075),
,(101,573147844013817084101),(102,927372692193078999176),(103,1500520536206896083277),(104,2427893228399975082453),
(105,3928413764606871165730),(106,6356306993006846248183),(107,10284720757613717413913),(108,16641027750620563662096),
(109,26925748508234281076009),(110,43566776258854844738105),(111,70492524767089125814114),(112,114059301025943970552219),
(113,184551825793033096366333),(114,298611126818977066918552),(115,483162952612010163284885),(116,781774079430987230203437),
(117,1264937032042997393488322),(118,2046711111473984623691759),(119,3311648143516982017180081),
(120,5358359254990966640871840),(121,8670007398507948658051921),(122,14028366653498915298923761),
(123,22698374052006863956975682),(124,36726740705505779255899443),(125,59425114757512643212875125),
(126,96151855463018422468774568),(127,155576970220531065681649693),(128,251728825683549488150424261),
(129,407305795904080553832073954),(130,659034621587630041982498215),(131,1066340417491710595814572169),
(132,1725375039079340637797070384),(133,2791715456571051233611642553),(134,4517090495650391871408712937),
(135,7308805952221443105020355490),(136,11825896447871834976429068427),(137,19134702400093278081449423917),
(138,30960598847965113057878492344),(139,50095301248058391139327916261),(140,81055900096023504197206408605)]
```

其他解法

Scala, 忠诚的 Haskell 追随者:

```scala
val fibs: LazyList[BigInt] = BigInt(1) #:: BigInt(1) #:: fibs.zip(fibs.tail).map(x => x._1 + x._2)
```

```
~ amm
Loading...
Welcome to the Ammonite Repl 2.0.4 (Scala 2.13.1 Java 1.8.0_242)
@ val fibs: LazyList[BigInt] = BigInt(1) #:: BigInt(1) #:: fibs.zip(fibs.tail).map(x => x._1 + x._2)
fibs: LazyList[BigInt] = LazyList(...)
@ fibs(99)
res1: BigInt = 354224848179261915075

@ fibs.take(10)
res2: LazyList[BigInt] = LazyList(1, 1, 2, 3, 5, 8, 13, 21, 34, 55)
```

其他解法

$$
\begin{aligned}
\mathit{fibs} &= 1, \quad 1, \quad 2, \quad 3, \quad 5, \quad 8 \ldots \\
(\mathit{tail} \quad \mathit{fibs}) &= 1, \quad 2, \quad 3, \quad 5, \quad 8, \quad 13 \ldots \\
\mathit{fibs}' &= 1+1, \quad 1+2, \quad 2+3, \quad 3+5, \quad 5+8 \ldots
\end{aligned}
$$

第 2 问，易如反掌

```scala
fibs
.zipWithIndex
.dropWhile(_._1.toString.length < 1000)
.head
._2 + 1
```

```
~ amm
Loading...
Welcome to the Ammonite Repl 2.0.4 (Scala 2.13.1 Java 1.8.0_242)
@ val fibs: LazyList[BigInt] = BigInt(1) #:: BigInt(1) #:: fibs.zip(fibs.tail).map(x => x._1 + x._2)
fibs: LazyList[BigInt] = LazyList(...)
@ fibs.zipWithIndex.dropWhile(_._1.toString.length < 1000).head._2 + 1
res1: Int = 4782
```

第 3 问，快速矩阵幂?

快速矩阵幂？杀鸡用牛刀。

暴力尾递归，你值得拥有：

```scala
def fib(n: Int) = {
  @scala.annotation.tailrec
  def f(x: Int, y: Int, i: Int): Int =
    if (i == 0) x else f(y, (x + y) % 1000000007, i - 1)
  f(0, 1, n)
}
```

```
Scala >> val t = System.currentTimeMillis ;  fib(1000000000) ; println(s”time escape: ${System.currentTimeMillis-t}ms”)
time escape: 2519ms
t: Long = 1589459096621L
res10_1: Int = 21
```

不到 3 秒！

第 3 问，C++

C++ 暴力循环：

```cpp
#include <iostream>

int fib(int n) {
    int a = 0, b = 1, c = 0;
    for (int i = 0 ; i < n ; i++) {
        c = b;
        b = (a + b) % 1000000007;
        a = c;
    }
    return a;
}

int main() {
    std::cout << fib(1000000000) << "\n";
    return 0;
}
```

```
~ $ g++ -O2 fibonacci.cc -o fibonacci-cpp
~ $ time ./fibonacci-cpp
21
./fibonacci-cpp  2.47s user 0.00s system 99% cpu 2.473 total
```

2.47 秒！

第 3 问，Rust

Rust 尾递归：

```rust
fn main() {
    fn fib_m(n: i32) -> i32 {
        fn _fib(x: i32, y: i32, i: i32, n: i32) -> i32 {
            if i == n { x } else { _fib(y, (x+y) % 1000000007, i+1, n) }
        }
        _fib(0, 1, 0, n)
    }
    println!("fibM(10^9) = {}", fib_m(1_000_000_000));
}
```

```
~ $ time ./fibonacci_rs
fibM(10^9) = 21
./fibonacci_rs  2.53s user 0.00s system 99% cpu 2.500 total
```

2.50 秒！

第 3 问，JavaScript

JavaScript 循环：

```
function fibM(n) {
    for (i = 0, x = 0, y = 1; i < n; i++) {
        var z = y
        y = (x + y) % 1000000007
        x = z
    }
    return x
}

console.log(fibM(1000000000))
```

```
~ $ time node fibonacci-loop.js
21
node fibonacci-loop.js  4.08s user 0.01s system 99% cpu 4.088 total
```

4.1 秒！

第 3 问，JavaScript

JavaScript 尾递归：

```javascript
'use strict';

function fibM(n) {
    let f = (x, y, i) => (i === 0) ? x : f(y, (x+y) % 1000000007, i-1)
    return f(0, 1, n)
}

console.log(fibM(1000000000)) // RangeError: Maximum call stack size exceeded
```

栈溢出！

第 3 问，Python

Python 循环：

```python
def fib(n):
    a, b = 0, 1
    i = 1
    while (i <= n):
        a, b = b, (a + b) % 1000000007
        i = i + 1
    return a

print(fib(1000000000))
```

第 3 问，Python

Python 循环测试

```
~ $ time python2.7 fibonacci-loop.py
21
python2.7 fibonacci-loop.py  57.04s user 0.01s system 99% cpu 57.103 total
~ $ time python fibonacci-loop.py
21
python3.7 fibonacci-loop.py  108.94s user 0.02s system 99% cpu 1:49.00 total
~ $ time python3.8 fibonacci-loop.py
21
python3.8 fibonacci-loop.py  112.31s user 0.01s system 99% cpu 1:52.33 total
~ $ time python3.9 fibonacci-loop.py
21
python3.9 fibonacci-loop.py  120.74s user 0.01s system 99% cpu 2:00.81 total
~ $ time pypy fibonacci-loop.py
21
pypy fibonacci-loop.py  3.11s user 0.01s system 99% cpu 3.134 total
~ $ time pypy3 fibonacci-loop.py
21
pypy3 fibonacci-loop.py  3.13s user 0.01s system 99% cpu 3.150 total
```

发生了什么?

第 3 问，Python

Python 尾递归：

```python
def fib(n):
    def f(x, y, i): return x if not i else f(y, (x + y) % 1000000007, i-1)
    return f(0, 1, n)

print(fib(1000000000)) # RecursionError: maximum recursion depth exceeded
```

栈溢出！

快速矩阵幂

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{21}b_{22} & a_{21}b_{12}+a_{22}b_{22} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11}+a_{12}b_{21} \\ a_{21}b_{11}+a_{21}b_{22} \end{bmatrix}$$

快速矩阵幂

$$
\begin{aligned}
\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \times \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^3 \times \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \times \begin{bmatrix} F_2 \\ F_1 \end{bmatrix}
\end{aligned}
$$

快速矩阵幂

$$\begin{bmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = A^n$$

快速矩阵幂

```scala
val Q = 1000000007L

final case class Matrix(a11: Long, a12: Long, a21: Long, a22: Long) {
    /**
      * 取模矩阵乘法
      * @param x 被乘数
      * @param m 模
      * @return
      */
    def *(x: Matrix, m: Long = Q): Matrix =
      Matrix(a11 = (a11 * x.a11 + a12 * x.a21) % m,
             a12 = (a11 * x.a12 + a12 * x.a22) % m,
             a21 = (a21 * x.a11 + a22 * x.a21) % m,
             a22 = (a21 * x.a12 + a22 * x.a22) % m)
    /**
      * 矩阵幂 (取模), m = Q
      * @param p 幂
      * @return
      */
    def ^(p: Int): Matrix = {
      require(p > 0, ”parameter `p` must be greater than 0”)
      Iterator.continually(this).take(p).reduceLeft(_ * _)
    }
}
```

快速矩阵幂

```scala
val Q = 1000000007L

final case class Matrix(a11: Long, a12: Long, a21: Long, a22: Long) {
    /**
      * 取模矩阵乘法
      * @param x 被乘数
      * @param m 模
      * @return
      */
    def *(x: Matrix, m: Long = Q): Matrix =
      Matrix(a11 = (a11 * x.a11 + a12 * x.a21) % m,
             a12 = (a11 * x.a12 + a12 * x.a22) % m,
             a21 = (a21 * x.a11 + a22 * x.a21) % m,
             a22 = (a21 * x.a12 + a22 * x.a22) % m)
    /**
      * 矩阵幂（取模），m = Q
      * @param p 幂
      * @return
      */
    def ^(p: Int): Matrix = {
      require(p > 0, "parameter `p` must be greater than 0")
      Iterator.continually(this).take(p).reduceLeft(_ * _)
    }
}
```

快速矩阵幂

```scala
val Q = 1000000007L

final case class Matrix(a11: Long, a12: Long, a21: Long, a22: Long) {
    def fastPow(n: Int): Matrix = {
      @scala.annotation.tailrec
      def f(z: Matrix, x: Matrix, ys: List[Char]): Matrix =
        ys match {
          case '0' :: xs => f(z * z, x, xs)
          case '1' :: xs => f(z * z, x * z, xs)
          case _         => x
        }
      val ZERO = Matrix(1, 0, 0, 1)
      f(A, ZERO, n.toBinaryString.reverse.toList)
    }

    def ^(p: Int): Matrix = fastPow(p)
}

def fib(n: Int) = (A^(n)).a12
```

快速矩阵幂

```
Scala >> def fib(n: Int) = A.fastPow(n).a12
defined function fib

Scala >> val t = System.currentTimeMillis ;  fib(1000000000) ; println(s”time escape: ${System.currentTimeMillis-t}ms”)
time escape: 0ms
t: Long = 1589518862649L
res11_1: Long = 21L

Scala >>
```

不到 1ms！

快速矩阵幂

```scala
object CatsMonoidFib extends App {

  import cats._

  val Q = 1000000007

  final case class Matrix(a11: Long, a12: Long, a21: Long, a22: Long)

  class MatrixMultiplicationMonoid(m: Int) extends Monoid[Matrix] {

    override def empty: Matrix = Matrix(1, 0, 0, 1)

    override def combine(x: Matrix, y: Matrix): Matrix =
      Matrix(a11 = (x.a11 * y.a11 + x.a12 * y.a21) % m,
             a12 = (x.a11 * y.a12 + x.a12 * y.a22) % m,
             a21 = (x.a21 * y.a11 + x.a22 * y.a21) % m,
             a22 = (x.a21 * y.a12 + x.a22 * y.a22) % m)
  }

  implicit val monoid: MatrixMultiplicationMonoid = new MatrixMultiplicationMonoid(Q)

  val A: Matrix = Matrix(0, 1, 1, 1)

  def fib(n: Int): Long = monoid.combineN(A, n).a12

  val logger: Logger = Logger(getClass)

  logger.info("fib(10^9) = {}", fib(1000000000))
}
```

超级快速矩阵幂

$$
\begin{aligned}
A^{p+q} &= A^p \cdot A^q \\
A^{p \cdot q} &= (A^p)^q \\
A^{p^q} &= A^{p^{q-1} \cdot p} \\
&= \left(A^{p^{q-1}}\right)^p \\
&= \left(\left(A^{p^{q-2}}\right)^p\right)^p \\
&= \left(\left(\left(A^{p^{q-3}}\right)^p\right)^p\right)^p \\
&= \underbrace{(((A^p)^p \cdots)^p)^p}_{\text{q}}
\end{aligned}
$$

超级快速矩阵幂

```scala
val Q = 1000000007L

final case class Matrix(a11: Long, a12: Long, a21: Long, a22: Long) {
  /**
    * 超级矩阵幂 (取模), M^(p^q), m = Q
    * @param p 幂
    * @param q 超幂
    * @return
    */
  def ^^(p: Int, q: Int): Matrix = {
    require(p > 0, "parameter `p` must be greater than 0")
    require(q > 0, "parameter `q` must be greater than 0")
    (1 until q).foldLeft(this ^ p)((z, _) => z ^ p)
  }
}

def fibPQ(p: Int, q: Int) = (A^^(p, q)).a12
```

超级快速矩阵幂

```scala
def fibPQ(p: Int, q: Int) = (A^^(p, q)).a12

val t = System.currentTimeMillis

(1 to 1000).foreach { _ ⇒
  val _ = fibPQ(10, 1000)
}

println(s”””F(10^1000) = fibPQ(10, 1000) = ${fibPQ(10, 1000)}”””)
println(s”””F(10^9)    = fibPQ(10, 9)    = ${fibPQ(10, 9)}”””)
println(s”time escape: ${System.currentTimeMillis - t}ms”)
```

```
F(10^1000) = fibPQ(10, 1000) = 552179166
F(10^9)    = fibPQ(10, 9)    = 21
time escape: 527ms
```

思考题

快速幂能做到多快

问题 1：Euler Project 122(https://projecteuler.net/problem=122)

The most naive way of computing n^{15} requires fourteen multiplications:

$$n \cdot n \cdot n \cdot n \cdot n \cdot n \cdot n \cdot n \cdot n \cdot n \cdot n \cdot n \cdot n \cdot n \cdot n = n^{15}$$

But using a “binary” method you can compute it in 6 multiplications:

$$\begin{aligned}
n \cdot n &= n^2 \\
n^2 \cdot n^2 &= n^4 \\
n^4 \cdot n^4 &= n^8 \\
n^8 \cdot n^4 &= n^{12} \\
n^{12} \cdot n^2 &= n^{14} \\
n^{14} \cdot n &= n^{15}
\end{aligned}$$

快速幂能做到多快

问题 1: Euler Project 122(https://projecteuler.net/problem=122)

However it is yet possible to compute it in only 5 multiplications:

$$
\begin{aligned}
n \cdot n &= n^2 \\
n^2 \cdot n &= n^3 \\
n^3 \cdot n^3 &= n^6 \\
n^6 \cdot n^6 &= n^{12} \\
n^{12} \cdot n^3 &= n^{15}
\end{aligned}
$$

We shall define $m(k)$ to be the minimum number of multiplications to compute n^k; for example $m(15) = 5$.

For $1 \leq k \leq 200$, find $\sum m(k)$.

问题 2:

我们用 G_n 来表示那菲波契数列每一项截取最后 n 位数字组成的新数列。比如,

$$
\begin{aligned}
G_1 &= 0,1,1,2,3,5,8,3,1,4,5,9,4,3,7,0,7,7,4,1,5,6,1,7,8\ldots \\
G_2 &= 0,1,1,2,3,5,8,13,21,34,55,89,44,33,77,10,87,97,84\ldots \\
G_3 &= 0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987\ldots
\end{aligned}
$$

已知 G_1 中一共有 10 个不重复的元素,G_2 中一共有 100 个不重复的元素,而 G_3 中一共只有 750 个不重复的元素。

记 G_n 中所有不重复元素的个数为 $g(n)$. 试计算 $g(4), g(5), g(6)$ 和 $g(7)$.

完。

谢谢大家！