HDOJ 3117 – Fibonacci Numbers

——by A Code Rabbit

Description

输入 n，输出斐波那契数列 f(n)。

关键在于，如果 f(n)大于 8 位，那么就输出“前 4 位...后 4 位”这种格式。

Types

Maths :: Matrix

Analysis

这题虽然是矩阵乘法和快速幂。

但是其真正的难点在于 f(n)大于 8 位后，如何精确得算出前 4 位。

小于 8 位，我们可以递推得到。

大于 8 位的后 4 位，我们可以用矩阵乘法和快速幂得到。

而大于 8 位的前 4 位，如果我们用矩阵乘法和快速幂，然后对计算中的数只取前几位的话，当 n 接近 10^8 的时候，精度误差会越来越大，导致 WA。

那么如何去求前 4 位呢……

我们可以用通项公式：

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

对于后面一项，

$$\left(\frac{1-\sqrt{5}}{2}\right)^n$$

因为括号内的式子约等于-0.75，在 n >= 40（结果超过 8 位）的时候，已经近乎于 0。

在 n 趋近正无穷大的时候，更是趋近无穷小，所以可以忽略。

之后我们把这个式子两边对 10 取对数，为的是把指数 n 化为乘数 n，免去了求高阶幂的过程。

现在可以计算出 $\log_{10}$ f(n)了，但是我们仍然会得到一个很大的数字，直接作为指数求 10 的幂，结果依然不精确。

我们知道 $\log_{10}$ f(n)的整数部分，相当于我们的结果最后大概有几个 0，或者说是个几位数。

而我们所求的答案并不需要知道所求的斐波那契数有几位。

因此，我们可以把整数部分去掉，只保留小数部分。

换句话说，我们所求的前四位数是什么，跟 $\log_{10} f(n)$的整数部分一点**关系都木有。

这时候我们可以得到我们的结果了，把 $\log_{10} f(n)$的小数部分作为指数求 10 的幂，然后前四位即是我们要的答案。

可以求幂后乘以 1000 再取整，即得到前四位的数字。

这样做，在求幂的时候甚至都不用快速幂，直接利用 pow( )函数搞定。

而且充分利用了通项公式在计算机计算中，n 越大，越准确，数位越高，越准确的特点。

又避免了通项公式在 n 较小或者数位较低时不准确的缺点(因为这一部分我们是用矩阵乘法和快速幂做的)。

而这个牛逼方法是网上各种解题报告的答案，不知道源自何处，我就不提出处了，反正不是我想的。

Solution

```
// HDOJ 3117
// Fibonacci Numbers
// by A Code Rabbit

#include <cstdio>
#include <cmath>

const int ORDER = 2;
const int DIVISOR_SHOW = 10000;

enum Need {
    ALL,
    FIRST,
    LAST,
};

struct Matrix {
    long long element[ORDER][ORDER];
};

int n;

Matrix mat_unit;
Matrix mat_one;

int Fibonacci(int x, Need need);
```

```
void INIT();
Matrix QuickPower(Matrix mat_unit, Matrix mat_one, int index, Need need);
Matrix Multiply(Matrix mat_a, Matrix mat_b, Need need);
/* if bo is true, it means that the program is competing the first four numbers of
Fibonacci numbers. */

int main() {
    while (scanf("%d", &n) != EOF) {
        if (n < 40) {
            printf("%d\n", Fibonacci(n, ALL));
        } else {
            printf("%d...%04d\n", Fibonacci(n, FIRST), Fibonacci(n, LAST));
        }
    }

    return 0;
}

int Fibonacci(int x, Need need) {
    if (need == FIRST) {
        double ans = x * log10((1.0 + sqrt(5.0)) / 2.0) - 0.5 * log10(5.0);
        ans = ans - (int)ans;
        ans = pow(10, ans);
        return (int)(ans * 1000);
    }
    INIT();
    Matrix mat_ans = QuickPower(mat_unit, mat_one, x, need);
    int ans = mat_ans.element[1][0];
    return ans;
}

void INIT() {
    for (int i = 0; i < ORDER; ++i) {
        for (int j = 0; j < ORDER; ++j) {
            mat_unit.element[i][j] = i == j ? 1 : 0;
        }
    }
    mat_one.element[0][0] = 1;
    mat_one.element[0][1] = 1;
    mat_one.element[1][0] = 1;
    mat_one.element[1][1] = 0;
```

```
}

Matrix QuickPower(Matrix mat_unit, Matrix mat_one, int index, Need need) {
    Matrix mat_result = mat_unit;
    while (index) {
        if (index & 1) {
            mat_result = Multiply(mat_result, mat_one, need);
        }
        mat_one = Multiply(mat_one, mat_one, need);
        index >>= 1;
    }
    return mat_result;
}

Matrix Multiply(Matrix mat_a, Matrix mat_b, Need need) {
    Matrix mat_result;
    for (int i = 0; i < ORDER; ++i) {
        for (int j = 0; j < ORDER; ++j) {
            mat_result.element[i][j] = 0;
            for (int k = 0; k < ORDER; ++k) {
                mat_result.element[i][j] += mat_a.element[i][k] * mat_b.element[k][j];
            }
            if (need == ALL) {
            } else
            if (need == LAST){
                mat_result.element[i][j] %= DIVISOR_SHOW;
            }
        }
    }
    return mat_result;
}
```