Fibonacci 函数的分析

罗力

2024 年 10 月 20 日

0 引言

本篇文档主要对程设作业 Assignment3 的第四题中出现的各种斐波那契函数进行分析，主要从时间复杂度角度切入，还会涉及到程序设计理念和可读性等方面。

1 Code 1

1) 这个版本是对的
2) 特点是太慢，还有可能爆栈，代码简洁易懂，适合去少儿编程班骗家长钱；
3) 没有
4)

```
/**
 * @file main.cpp
 * @brief This is the main file for calculating the ith number of
    Fibonacci sequence.
 */

 /**
 * @brief Function to calculate the ith number of Fibonacci sequence.
 *
 * This function calculates the ith number of Fibonacci sequence using
    common recursion.
 *
 * @param n The integer to be solved.
 * @return The nth number of Fibonacci sequence.
 */
// TODO: Reduce the time complexity and handle stack overflow risk
int fibonacci_recursive(int n)
{
    if (n <= 1) return n;
    // According to Fibonacci sequence
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2);
}
```

2 Code 2

1) 这个版本是对的

2) 这是正向求取斐波那契数列的循环算法，单次查询时间复杂度为 o(n)，多次查询累加，适合只查询一个斐波那契数列值的情景

3) 没有

4)

```
/**
 * @file main.cpp
 * @brief This is the main file for calculating the ith number of
     Fibonacci sequence.
 */

 /**
 * @brief Function to calculate the ith number of Fibonacci sequence.
 *
 * This function calculates the ith number of Fibonacci sequence using
     common loop.
 *
 * @param n The integer to be solved.
 * @return The nth number of Fibonacci sequence.
 */

// TODO: Reduce time complexity
int fibonacci_iterative(int n) {
    if (n <= 1) return n;
    int a = 0, b = 1, temp;
    // temp stores a new item of fibonacci sequence
    // a stores fib(n-2)
    // b stores fib(n-1)
    for (int i = 2; i <= n; ++i) {
        temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}
```

3 Code 3

1) 这个版本是对的

2) 使用了动态规划去存储斐波那契数列的各项，避免了重复运算，代码简洁易懂，并且对于多次查询，时间复杂度也是 o(n)。是一个 n 不太大时编码较快的写法并且就算是

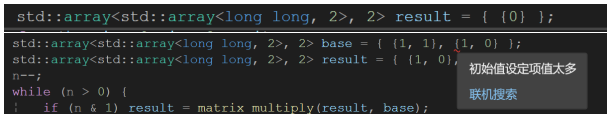
3) 没有

4)

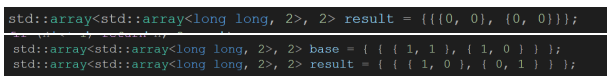
```cpp
/**
 * @file main.cpp
 * @brief This is the main file for calculating the ith number of
    Fibonacci sequence.
 */
#include <vector>

 /**
 * @brief Function to calculate the ith number of Fibonacci sequence.
 *
 * This function calculates the ith number of Fibonacci sequence using
    dynamic programing.
 *
 * @param n The integer to be solved.
 * @return The nth number of Fibonacci sequence.
 */

int fibonacci_dp(int n) {
 if (n <= 1) return n;
 // dp[i] refers to the i th item of Fibonacci sequence
 std::vector<int> dp(n + 1, 0);
 dp[1] = 1;
 for (int i = 2; i <= n; ++i) {
 dp[i] = dp[i - 1] + dp[i - 2];
 }
 return dp[n];
}
```

4 Code 4

1) 这个版本含语法错误



正确的修改方式是:



2) 使用了矩阵快速幂的算法去计算斐波那契数列（时间复杂度为 N^3logn，这里 N=2）在 n 较大的时候比 dp 有显著优势，但是在 n 较小的时候可能还是 dp 好一点

3) 没有

4)

```cpp
/**
 * @file main.cpp
 * @brief This is the main file for calculating the ith number of
     Fibonacci sequence.
 */
#include <array>

 /**
 * @brief Function to multiply two matrix.
 *
 * This function multiplys two matrix.
 *
 * @param a The first matrix.
 * @param b The second matrix.
 * @return The result of matrix multiply.
 */
/*
* matrix multiply function
*/
std::array<std::array<long long, 2>, 2> matrix_multiply(const
    std::array<std::array<long long, 2>, 2>& a, const std::array<std::
       array<long long,
    2>, 2>& b) {
    std::array<std::array<long long, 2>, 2> result = {{{0, 0}, {0,
       0}}};
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
            for (int k = 0; k < 2; ++k)
                result[i][j] += a[i][k] * b[k][j];
    return result;
}
/**
 * @brief Function to calculate the ith number of Fibonacci sequence.
 *
 * This function calculates the ith number of Fibonacci sequence using
     matrix quickPower.
 *
 * @param n The integer to be solved.
```

```
 * @return The nth number of Fibonacci sequence.
 */
long long fibonacci_matrix(int n) {
    if (n <= 1) return n;
    std::array<std::array<long long, 2>, 2> base = { { { 1, 1 }, { 1, 0 } } };
    std::array<std::array<long long, 2>, 2> result = { { { 1, 0 }, { 0, 1 } } };
    n--;
    while (n > 0) {
        if (n & 1) result = matrix_multiply(result, base);
        base = matrix_multiply(base, base);
        n >>= 1;
    }
    return result[0][0];
}
```

5 Code 5

1) 这个版本是对的

2) 使用了记忆化搜索的方法去求斐波那契数列，代码简洁易懂，这个时间复杂度也和 dp 一样的是 o(n)，但是常数会大一点（主要来自于系统栈的入栈和出栈操作），也有可能爆栈（没有暴力递归容易），总的来说什么方面都不如 dp。

3) 没有

4)

```
/**
 * @file main.cpp
 * @brief This is the main file for calculating the ith number of Fibonacci sequence.
 */
#include <unordered_map>

 /**
 * @brief Function to calculate the ith number of Fibonacci sequence.
 *
 * This function calculates the ith number of Fibonacci sequence using memorization search.
 *
 * @param n The integer to be solved.
 * @return The nth number of Fibonacci sequence.
 */
// TODO: Handle stack overflow risk
```

```cpp
long long fibonacci_memoization(int n, std::unordered_map<int, long
    long>&
     memo) {
    if (n <= 1) return n;
    if (memo.find(n) != memo.end()) return memo[n];
    memo[n] = fibonacci_memoization(n - 1, memo) +
        fibonacci_memoization(n - 2, memo);
    return memo[n];
}
```

6 Code 6

1) 这个版本是对的

2) 这和 Code 2 的版本其实是一样的，只不过用了一个很新的特性 c++17 里的 optional，它可以返回一个指定类型的值，也可以就返回空值。这里的用法增加了安全性（n<1 时斐波那契数列未定义，于是返回 nullopt）。由于使用了 optional，安全性得以保障，也许可以用于军工，车工代码?（乱说的，反正安全就对了）

3)optional 是啥，去查了一下

4)

```cpp
/**
 * @file main.cpp
 * @brief This is the main file for calculating the ith number of
     Fibonacci sequence.
 */
#include <optional>

 /**
 * @brief Function to calculate the ith number of Fibonacci sequence.
 *
 * This function calculates the ith number of Fibonacci sequence using
     common loop and uses optional container to keep return value safe.
 *
 * @param n The integer to be solved.
 * @return The nth number of Fibonacci sequence or nullopt if n < 1.
 */

// TODO: Reduce time complexity
std::optional<long long> fibonacci_safe(int n) {
    if (n < 0) return std::nullopt;
    if (n <= 1) return n;
    long long a = 0, b = 1, temp;
    for (int i = 2; i <= n; ++i) {
```

```
        temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}
```

7 Code 7

1) 这个版本是对的

2) 这和 Code 2 的版本其实是一样的，但是用到了 constexpr 这个关键字去修饰函数，表示这个函数是常量表达式，可以在编译期就得到表达式的值，节省了运行期的运行时间，当然也对使用场景进行了限制，通俗来说，就是调用这个函数时传入的参数必须是常熟（字面值类型，比如 5，一个 const int 或者 define n 5 这些都算）

3) 无

4)

```
/**
 * @file main.cpp
 * @brief This is the main file for calculating the ith number of
    Fibonacci sequence.
 */

 /**
 * @brief A constexpression function to calculate the ith number of
    Fibonacci sequence.
 *
 * This function calculates the ith number of Fibonacci sequence using
    common loop and uses optional container to keep return value safe.
 *
 * @param n The integer to be solved(must be literal type).
 * @return The nth number of Fibonacci sequence or nullopt if n < 1.
 */

// TODO: Reduce time complexity
constexpr long long fibonacci_constexpr(int n) {
    if (n <= 1) return n;
    long long a = 0, b = 1;
    for (int i = 2; i <= n; ++i) {
        long long temp = a + b;
        a = b;
        b = temp;
    }
```

```
    return b;
}
```

8 Code 8

1) 这个版本是对的

2) 使用到了多线程技术（这里得点个赞了，之前一直好奇 c++ 的多线程怎么实现的，但是一直没有去学，这个作业突然让我想起来这么一回事了  ），多线程的优点不用多说，肯定是提高运行效率嘛，不然玩个游戏真没法玩了，使用场景也是非常广泛啊（比如原神）

3) 去了解了一下啥是 async 和 future，结果大吃一惊啊

4)

```
/**
 * @file main.cpp
 * @brief This is the main file for calculating the ith number of
    Fibonacci sequence.
 */

#include <future>

 /**
 * @brief Function to calculate the ith number of Fibonacci sequence
    using multi-threading technology.
 *
 * This function calculates the ith number of Fibonacci sequence using
    multi-threading recursion.
 *
 * @param n The integer to be solved.
 * @return The nth number of Fibonacci sequence.
 */
// TODO: Reduce the time complexity and handle stack overflow risk
long long fibonacci_parallel(int n) {
 if (n <= 1) return n;
 auto future = std::async(std::launch::async, fibonacci_parallel, n -
    2);
 long long result = fibonacci_parallel(n - 1);
 return result + future.get();
}
```

9 Code 9

1) 这个版本是对的

2) 使用了 boost 第三方中的 cppint 类啊，也就是一个无穷限高精度整形（草了，以前我都是手写高精度的，怎么不早告诉我有第三方库），具体 boost 库还有什么东西没去看，应用场景显而易见啊，就在 n 特别特别大的时候用呗

3) 去了解了一下啥是 async 和 future，结果大吃一惊啊，居然是我梦寐以求的多线程技术

4)

```cpp
/**
 * @file main.cpp
 * @brief This is the main file for calculating the ith number of
     Fibonacci sequence.
 */

#include <future>

 /**
 * @brief Function to calculate the ith number of Fibonacci sequence.
 *
 * This function calculates the ith number of Fibonacci sequence using
     multi-threading recursion. And, what's so cool is that you don't
     have to consider return value overleap problem!
 *
 * @param n The integer to be solved.
 * @return The nth number of Fibonacci sequence.
 */
// TODO: Reduce the time complexity
#include <boost/multiprecision/cpp_int.hpp>
boost::multiprecision::cpp_int fibonacci_bigint(int n) {
 if (n <= 1) return n;
 boost::multiprecision::cpp_int a = 0, b = 1, temp;
 for (int i = 2; i <= n; ++i) {
 temp = a + b;
 a = b;
 b = temp;
 }
 return b;
}
```

10 Code 10

1) 这个版本是对的

2) 不是哥们，你来真的啊，这就是大名鼎鼎的元编程 (using template)，我看到这 Code 10 的时候是虎躯一震啊，心想写个斐波那契数列还用上模板元编程了，其实原理和 Code 7 有点像啊，都是对于字面值类型参数，可以在编译期求出斐波那契数列值，只不过这是用的递归实现（其实有一

说一，写文档写到这里了突然发现其实 Code 7 的 constexpr 也算是元编程的一种啊，之前从来没有往这方面想过，也算是有新知吧）然后补充了一下当 n=0 和 n=1 时的模板特例。对了，忘了给互评的同学说什么是元编程了（有同学认真看了我的文档吗？QAQ）元编程简单来说就是编译器用你自己写的代码生成新的代码，一个比较好理解的例子（也是我经常使用的写法）是下面的代码片段：

```
#define min(a, b) (a < b ? a : b)
```

上面的代码实现了什么功能呢？主要是使用含参宏定义，将 a 代码块和 b 代码块（注意，是代码块，不是一个值）直接填充到 a < b ? a : b 这个式子里，下面给一个实施例：

```
int func(int a)
{
    return a * a;
}
int main()
{
    int x, y; cin >> x >> y;
    cout << min(func(x), func(y)) << endl;

    return 0;
}
```

这段代码做了什么事呢？从结果来看就是输出 x^2 和 y^y 中最大的那个数。但是其中的原理其实是编译器在编译期就把你的代码给改成了下面的样子：

```
int func(int a)
{
    return a * a;
}
int main()
{
    int x, y; cin >> x >> y;
    cout << (func(x) < func(y) ? func(x) : func(y)) << endl;

    return 0;
}
```

然后再执行，可能聪明的同学已经发现了，事实上最后求值还是要到运行期用户输入 x 和 y 后才能进行，但是你不能否认，这也是一种元编程（乐）

抱歉，讲得有点多了，反正记住一点就行了，对于字面值类型参数的函数，使用元编程将会大大提高程序运行效率，而模板元编程是元编程里最强大的写法

3) 无

4)

```
/**
 * @file main.cpp
```

```
 * @brief This is the main file for calculating the ith number of
    Fibonacci sequence.
 */
/**
 * @brief metatemplate to calculate the ith number of Fibonacci
    sequence.
 *
 * @param n The integer to be solved(must be literal type).
 * @return The nth number of Fibonacci sequence.
 */
 * This template calculates the ith number of Fibonacci sequence using
    metaprogramming. And, this is almost what I can imagine the most
    beautiful and the most elegant one. But, be cautious, the input
    param should be literal type.
 *
template<int N, typename = std::enable_if_t<(N >= 0)>>
struct Fibonacci {
    static constexpr long long value = Fibonacci<N - 1>::value +
       Fibonacci<N2>::value;
};
template<>
struct Fibonacci<0> {
    static constexpr long long value = 0;
};
template<>
struct Fibonacci<1> {
    static constexpr long long value = 1;
};
```

11 后日谈

这篇文档我也算得上是呕心沥血了，其实前面几种实现我都觉得平平无奇，但是从 Code 7 开始我就越发感到兴奋。这种感觉真的很久没有了，就像是我第一次接触编程，就像是回归了那个懵懂的年纪一样，一切都是如此的新奇而又有趣，希望以后能够遇到更多这么有趣的东西，而不只是整天写一些没有太多营养的 code，也祝福看到这里的老师/助教/同学们也能收获属于自己的快乐。So, 加纳!