请阅读以下10个计算 fibonacci 中第 i 个元素的实现函数，并对每个函数回答下列问

题：

1）、该实现版本是否正确？如果不正确又什么需要修改的。

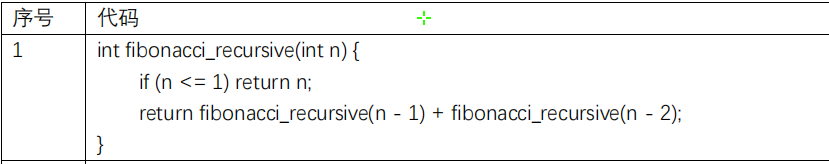
2）、请说明该实现有什么特点？这种实现⽅式的主要优点是什么？适合于什么样的场景？

3）、请说明该实现版本中有哪些 c/c++程序设计语法和算法细节是你原来不了解的，请说明这些语法。

4）、请按照 Google C++ Coding Style 的要求，为每个函数添加注释。

5）、请你在综合这 10 个版本之后，自己重新写⼀个版本。该版本是你认为最优雅，最好的计算 fibonacci 中第 i 个元素的实现。

## 一、



1）、正确。

2）、使用return语句来计算和调用函数，并进行递归。优点是非常简短，高效。适合完全了解原理的时候使用，因为初学者看不明白。

3）、无。

4）、

// 计算斐波那契数列的第n项（递归实现）。

//

// 参数:

// n - 要计算的斐波那契数列的项数（非负整数）。

//

// 返回值:

// 返回斐波那契数列的第n项的值。

//

// 注意:

// 由于递归实现，对于较大的n值，可能会导致性能低下和栈溢出。

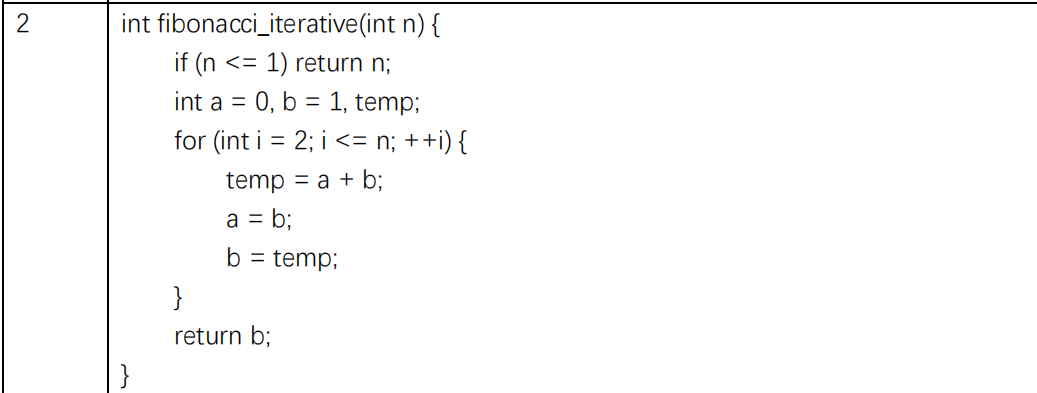
int fibonacci\_recursive(int n) {

if (n <= 1) return n;

return fibonacci\_recursive(n - 1) + fibonacci\_recursive(n - 2);

}

## 二、



1）、正确。

2）、使用了最基础的循环语法来计算。优点是容易看懂，逻辑简单。适合初学者使用。

3）、无。

4）、

// 计算斐波那契数列的第n项（迭代实现）。

//

// 参数:

// n - 要计算的斐波那契数列的项数（非负整数）。

//

// 返回值:

// 返回斐波那契数列的第n项的值。

//

// 注意事项:

// 此方法通过迭代计算斐波那契数列，相比递归实现，具有更好的性能表现，

// 能够处理较大的n值而不会导致栈溢出。

int fibonacci\_iterative(int n) {

if (n <= 1) return n; // 基本情况：如果n是0或1，直接返回n。

int a = 0, b = 1, temp; // 初始化前两个斐波那契数。

for (int i = 2; i <= n; ++i) { // 从第2项开始迭代计算。

temp = a + b; // 计算当前项的值。

a = b; // 更新a为前一项的值。

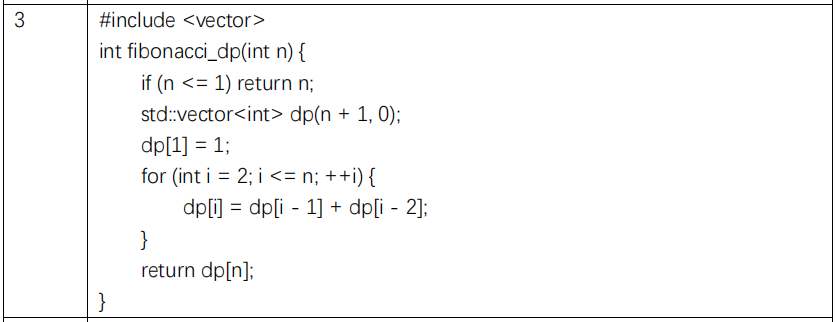
b = temp; // 更新b为当前项的值。

}

return b; // 返回第n项的值。

}

## 三、



1）、正确。

2）、使用vector容器进行累加运算。优点是直观简单，并且可以存储整个数列。在需要获得整个斐波那契数列时可以使用。

3）、无

4）、

// 计算斐波那契数列的第 n 项（n >= 0），使用动态规划（DP）方法优化性能。

//

// 参数:

// n - 要计算的斐波那契数列的项数（非负整数）。

//

// 返回值:

// 返回斐波那契数列的第 n 项的值。

//

// 方法:

// 通过动态规划数组 dp 存储中间结果，避免重复计算，从而提高效率。

// dp[i] 表示斐波那契数列的第 i 项。

//

int fibonacci\_dp(int n) {

if (n <= 1) return n;

std::vector<int> dp(n + 1, 0);

dp[1] = 1;

for (int i = 2; i <= n; ++i) {

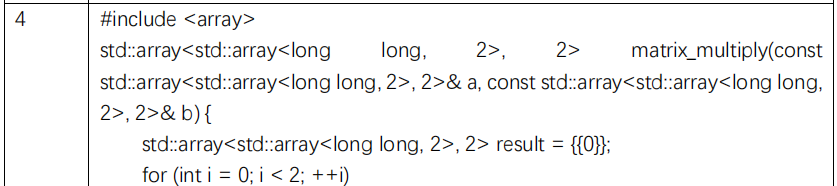
dp[i] = dp[i - 1] + dp[i - 2];

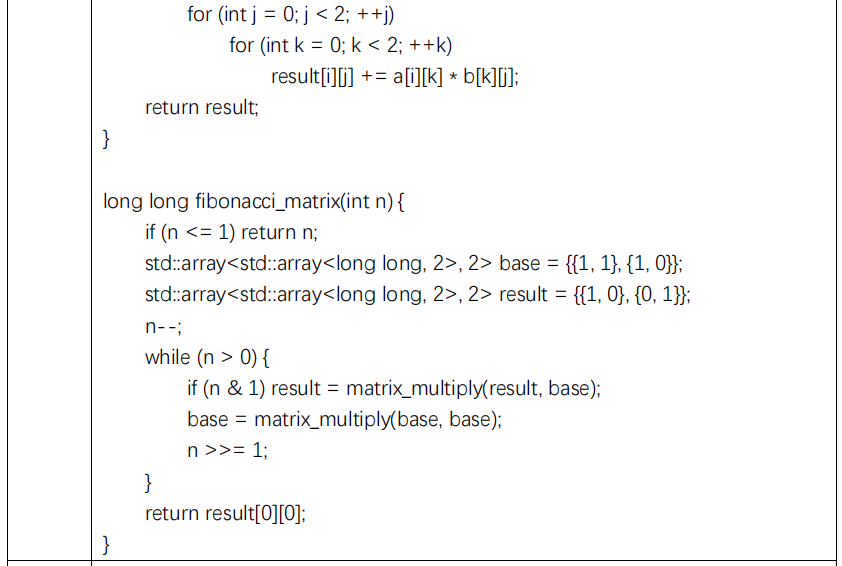
}

return dp[n];

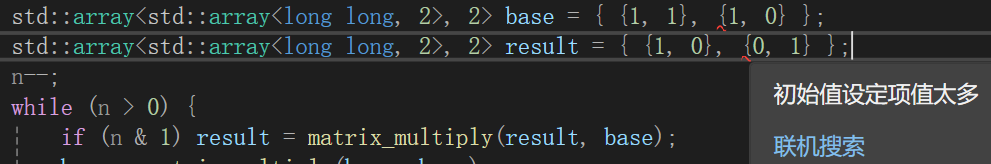
}

## 四、





1）、错误，运行时报错。



改为：

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}} };

2）、使用矩阵乘法和快速幂算法来计算斐波那契数列。优点是时间复杂度低，运算快。在需要节约时间时可以使用。

3）、学到了如何使用array容器和如何初始化array数组。

4）、

#include <array>

// 矩阵乘法函数，用于计算两个2x2矩阵的乘积

//

// 参数:

// a: 第一个输入的2x2矩阵（常量引用）

// b: 第二个输入的2x2矩阵（常量引用）

//

// 返回值:

// 返回两个矩阵的乘积，结果也是一个2x2矩阵

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}} }; // 初始化结果矩阵，所有元素为0

for (int i = 0; i < 2; ++i) { // 遍历第一个矩阵的行

for (int j = 0; j < 2; ++j) { // 遍历第二个矩阵的列

for (int k = 0; k < 2; ++k) { // 遍历乘积的求和项

result[i][j] += a[i][k] \* b[k][j]; // 计算乘积并累加到结果矩阵中

}

}

}

return result; // 返回结果矩阵

}

#include <array>

// 使用矩阵快速幂计算斐波那契数列的第 n 项（n 从 0 开始计数）

//

// 参数:

// n: 斐波那契数列中的位置（非负整数）

//

// 返回值:

// 斐波那契数列的第 n 项（long long 类型）

long long fibonacci\_matrix(int n) {

if (n <= 1) return n; // 如果 n 小于等于 1，直接返回 n

// 初始化斐波那契矩阵（用于表示斐波那契数列的递推关系）

std::array<std::array<long long, 2>, 2> base = { {{1, 1}, {1, 0}} };

// 初始化单位矩阵（矩阵乘法的恒等元素）

std::array<std::array<long long, 2>, 2> result = { {{1, 0}, {0, 1}} };

n--; // 将 n 减 1，因为我们要计算的是第 n 项，而矩阵幂次是从 0 次方开始的

// 使用矩阵快速幂算法计算 base 的 n 次方

while (n > 0) {

if (n & 1) { // 如果 n 是奇数，将 result 乘以 base

result = matrix\_multiply(result, base);

}

base = matrix\_multiply(base, base); // 将 base 平方

n >>= 1; // 将 n 右移一位，相当于整除 2

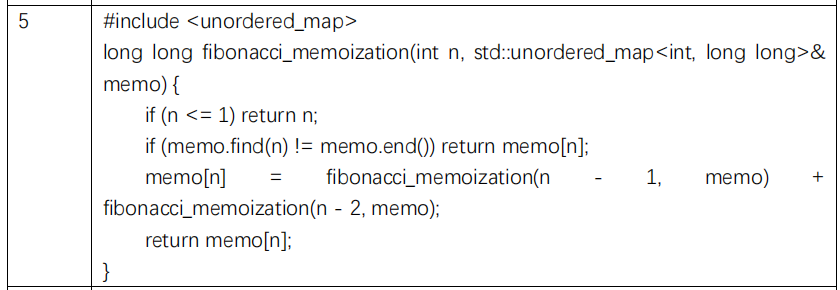
}

// 返回结果矩阵的左上角元素，即斐波那契数列的第 n 项

return result[0][0];

}

## 五、



1）、正确。

2）、使用了memo哈希表来储存，递归来计算数列。优点是效率高，错误率低。适用于需要提升算法效率的情景。

3）、学到了使用unordered\_map存储键对并初始化和使用它来计算。

4）、

#include <unordered\_map>

// 使用备忘录法计算斐波那契数列的第 n 项（n 从 0 开始计数）

//

// 参数:

// n: 斐波那契数列中的位置（非负整数）

// memo: 用于存储已计算斐波那契数的哈希表（引用传递，以节省内存和避免复制）

//

// 返回值:

// 斐波那契数列的第 n 项（long long 类型）

long long fibonacci\_memoization(int n, std::unordered\_map<int, long long>& memo) {

// 基本情况：n 小于等于 1 时，直接返回 n

if (n <= 1) {

return n;

}

// 检查 memo 中是否已存储 n 对应的斐波那契数

// 如果已存储，则直接返回该值，避免重复计算

if (memo.find(n) != memo.end()) {

return memo[n];

}

// 递归计算斐波那契数，并将结果存储在 memo 中

// 注意：这里会进行两次递归调用，但由于 memo 的存在，重复计算会被避免

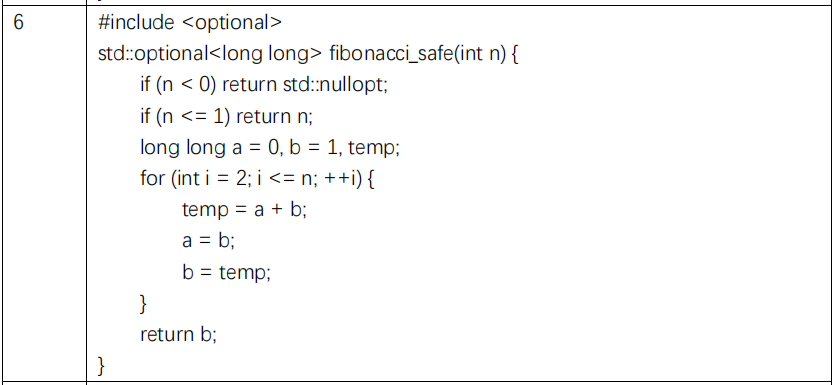
memo[n] = fibonacci\_memoization(n - 1, memo) + fibonacci\_memoization(n - 2, memo);

// 返回计算得到的斐波那契数

return memo[n];

}

## 六、



1）、正确。

2）、使用了optional返回值，使得当输入不合法时可以返回nullopt，计算仍使用了循环。优点是有更多样化的输出。可以替代之前的第二个程序，不过需要C++17及以上的语言标准才可以。

3）、学到了optional来多样化输出。

4）、

#include <optional>

// 计算斐波那契数列中第 n 项的值（n 从 0 开始计数）。

// 如果 n 为负数，则返回 std::nullopt 表示无效输入。

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;

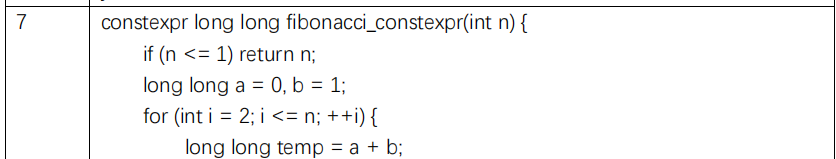
b = temp;

}

return b;

}

## 七、





1）、正确。

2）、利用 constexpr 关键字允许在可能的情况下在编译时进行计算。优点是可以优化性能。在需要节约性能时使用。

3）、学到了一个新的关键字constexpr

4）、

// 计算斐波那契数列中的第 n 个数字（n 从 0 开始计数）。

//

// 使用编译时常量表达式函数（如果可能的话），通过迭代方式计算斐波那契数列。

//

// 参数:

// n - 一个非负整数，表示要计算的斐波那契数列中的位置（从 0 开始）。

//

// 返回值:

// 返回斐波那契数列中的第 n 个数字，类型为 long long。

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;

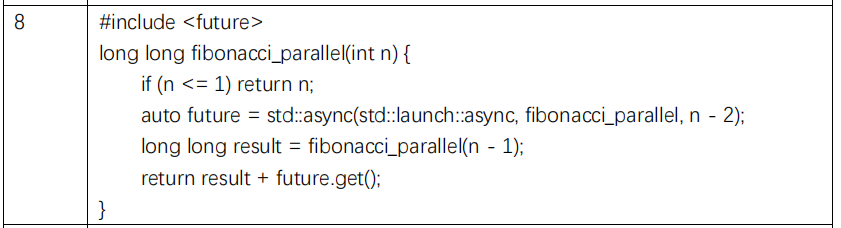
b = temp;

}

return b;

}

## 八、



1）、正确。

2）、启动了异步任务来计算数列。可以通过并行计算来加快计算速度。在需要节约时间的情况下可以使用。

3）、了解了异步任务，但是了解的还很少，需要更加深入。

4）、

#include <future>

// 并行计算斐波那契数列中的第 n 个数字（n 从 0 开始计数）。

//

// 该函数使用异步任务来并行计算斐波那契数列中的两个子问题，从而加速整体计算过程。

// 它通过递归调用自身来计算斐波那契数列，并利用 C++11 的 std::async 和 std::future

// 来实现异步并行计算。

//

// 参数:

// n - 一个非负整数，表示要计算的斐波那契数列中的位置（从 0 开始）。

//

// 返回值:

// 返回斐波那契数列中的第 n 个数字，类型为 long long。

long long fibonacci\_parallel(int n) {

if (n <= 1) return n; // 基础情况：直接返回 n（0 或 1）。

// 异步计算斐波那契数列中的第 n-2 个数字。

auto future = std::async(std::launch::async, fibonacci\_parallel, n - 2);

// 同步计算斐波那契数列中的第 n-1 个数字。

long long result = fibonacci\_parallel(n - 1);

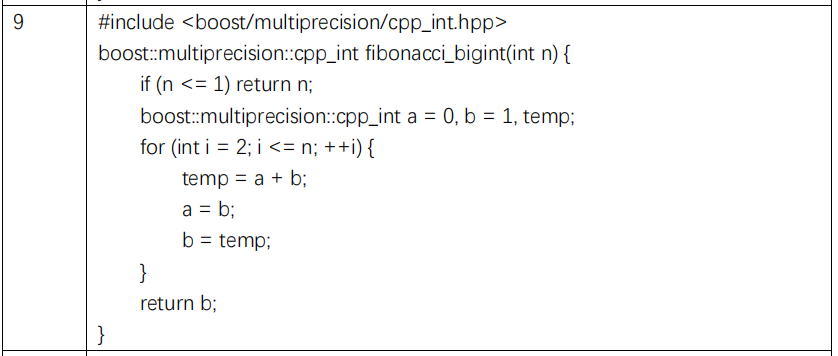
// 返回两个子问题的结果之和，即斐波那契数列中的第 n 个数字。

// 注意：future.get() 会阻塞当前线程，直到异步任务完成。

return result + future.get();

}

## 九、



1）、正确。

2）、使用了boost库来储存大数字。优点是支持大数字运算。 当安装了boost库并且需要进行大数运算时可以使用。

3）、第一次知道Boost库，并进行了简单的了解。

4）、

#include <boost/multiprecision/cpp\_int.hpp>

// 计算斐波那契数列中的第 n 个数字，返回值为 boost::multiprecision::cpp\_int 类型，

// 以支持大整数的计算。如果 n <= 1，则直接返回 n。

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;

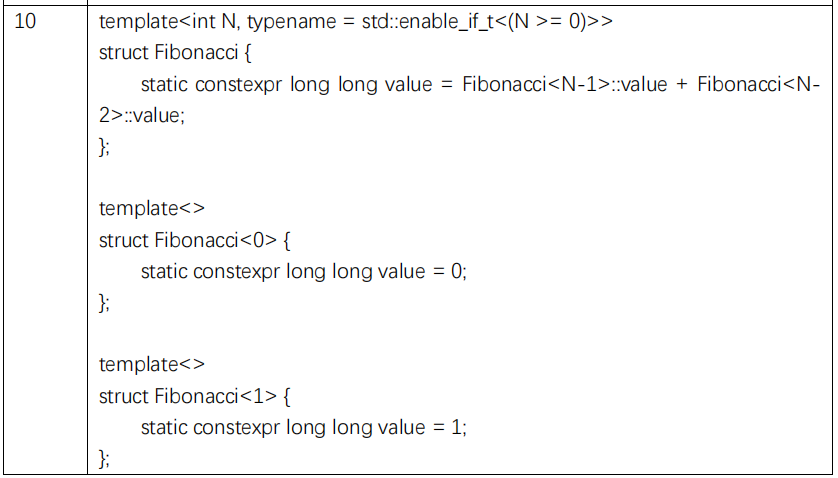
b = temp;

}

return b;

}

## 十、



1）、正确。

2）、使用了模板结构体来计算数列。优点是在编译时计算，节约时间。在N比较小且追求速度时可以使用。

3）、第一次知道了模板类，但是还需要更加深入的了解。

4）、

// 定义一个模板结构体 Fibonacci，用于在编译时计算斐波那契数列的第 N 项。

// 模板参数 N 必须是一个非负整数，这通过 std::enable\_if\_t 进行约束。

// 结构体包含一个静态常量成员 value，其值为斐波那契数列的第 N 项。

template<int N, typename = std::enable\_if\_t<(N >= 0)>>

struct Fibonacci {

// 静态常量成员 value，其值为斐波那契数列的第 N 项，通过递归模板实例化计算得出。

static constexpr long long value = Fibonacci<N - 1>::value + Fibonacci<N - 2>::value;

};

// Fibonacci 结构体的特化版本，用于处理 N 为 0 的情况。

template<>

struct Fibonacci<0> {

// 静态常量成员 value，其值为斐波那契数列的第 0 项，即 0。

static constexpr long long value = 0;

};

// Fibonacci 结构体的特化版本，用于处理 N 为 1 的情况。

template<>

struct Fibonacci<1> {

// 静态常量成员 value，其值为斐波那契数列的第 1 项，即 1。

static constexpr long long value = 1;

};

## 十一、

我认为的最好的：  
int fibonacci\_recursive(int n) {

if (n <= 1) return n;

return fibonacci\_recursive(n - 1) + fibonacci\_recursive(n - 2);

}