**ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»**

**(СПбГУТ)**

**Кафедра безопасности информационных систем**

**ОТЧЁТ**

по практической работе № 3 на тему:   
**«Понятие рекурсии. Рекурсивные функции.»**

по дисциплине «Алгоритмы и структуры данных»

Выполнил: студент группы ИСТ-312, Кандиков М.В.

«24» Сентября 2024 г. \_\_\_\_\_\_\_\_\_\_\_/М.В. Кандиков /

Принял: к.ф.-м.н., доцент, И.А. Моисеев

«24» Сентября 2024 г. \_\_\_\_\_\_\_\_\_\_\_/ И.А. Моисеев /

**Содержание**

**1. Титульный лист 1**

**2. Содержание 2**

**3. Основная часть 3**

3.1. Цель работы 3

3.1. Теоретическая часть 3

3.2. Задача №1 3

3.3. Задача №2 4

3.4. Задача №3 6

3.5. Задача №4 7

3.6. Задача №5 9

3.5. Выводы 10

**4. Приложение 9**

**Основная часть**

**Цель работы:**

Целью данной практической работы является изучение рекурсии и рекурсивных функций в программировании, а также освоение приёмов построения и применения рекурсивных методов для решения различных задач на языке C++. Основное внимание уделяется сравнению эффективности рекурсивных и итерационных методов.

**Теоретическая часть:**

**Рекурсивный алгоритм** — это такой алгоритм, в котором происходит вызов самого себя с изменённым набором параметров. Рекурсия широко применяется в программировании, так как позволяет кратко выразить алгоритмы, которые имеют свойство самоподобия. При решении задач с помощью рекурсии выделяют три основных этапа, которые называют **рекурсивной триадой**:

1. **Параметризация** — выделение параметров, которые описывают условия задачи.
2. **База рекурсии** — случай, при котором решение очевидно и не требует дальнейшего рекурсивного вызова.
3. **Декомпозиция** — разбиение задачи на более простые подзадачи с измененными параметрами, решение которых сводится к вызову той же функции.

**Задача №1:**

Задача заключается в нахождении n-го члена последовательности Фибоначчи. Последовательность Фибоначчи формируется по следующему правилу: первый и второй элементы равны единице, каждый последующий элемент равен сумме двух предыдущих. Требуется разработать рекурсивную функцию для вычисления n-го члена последовательности.

Рекурсивная триада:

1. Параметризация: n – номер элемента последовательности.
2. База рекурсии: для n=0 значение равно 0, для n=1 значение равно 1.
3. Декомпозиция: каждый элемент последовательности Фибоначчи равен сумме двух предыдущих: f(n)=f(n−1)+f(n−2)

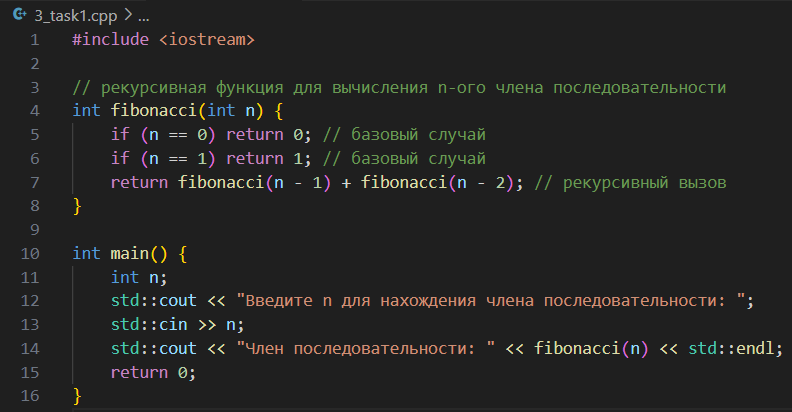


Рисунок 1. Решение для задачи №1.



Рисунок 2. Результат решения для 8 члена последовательности Фибоначчи.



Рисунок 3. Результат решения для 30 члена последовательности Фибоначчи.

Рекурсивный метод работает эффективно для малых значений n, но с увеличением n он становится крайне неэффективным из-за большого количества повторяющихся вызовов. Для больших n рекомендуется использовать итеративный или оптимизированный метод с запоминанием (мемоизацией).

**Задача №2:**

Смысл этой задачи в вычисление числа сочетаний из n элементов по m. Требуется разработать программу для вычисления числа сочетаний из n по m. Необходимо решить задачу как рекурсивным, так и итеративным методом, а затем сравнить результаты.

Рекурсивная триада:

1. Параметризация: n и m – параметры для расчёта числа сочетаний.
2. База рекурсии: если m=0 или n=m, то результат равен 1.
3. Декомпозиция**:** число сочетаний можно выразить через формулу: C(n по m)=C(n−1 по m−1)+C(n−1 по m).

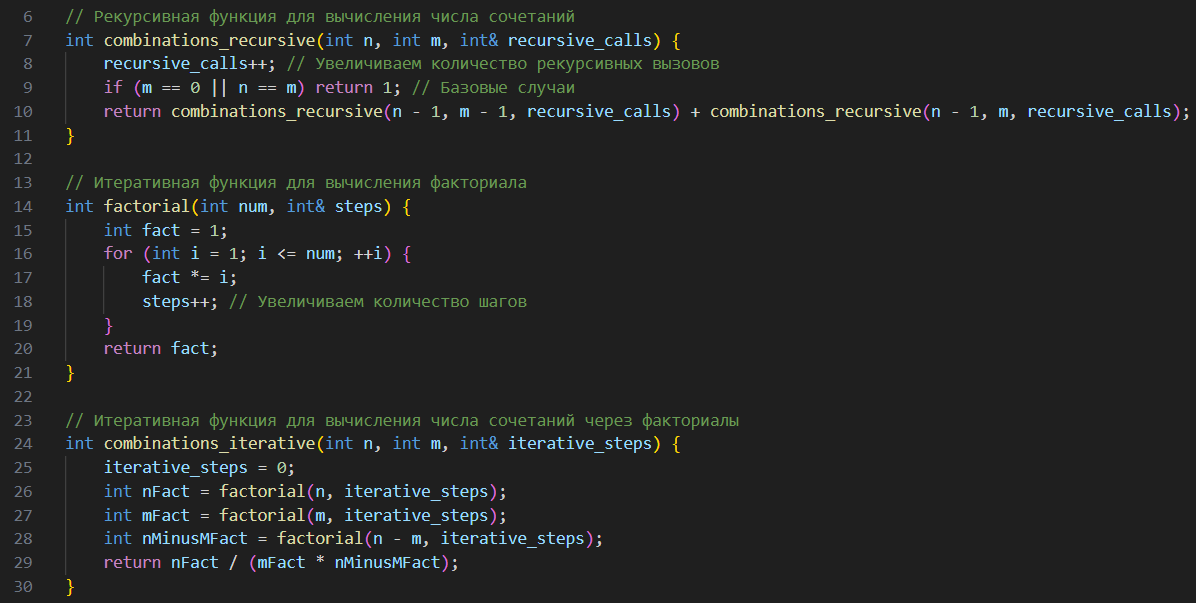


Рисунок 4. Код рекурсивной функции и итеративной функции для задачи №2.

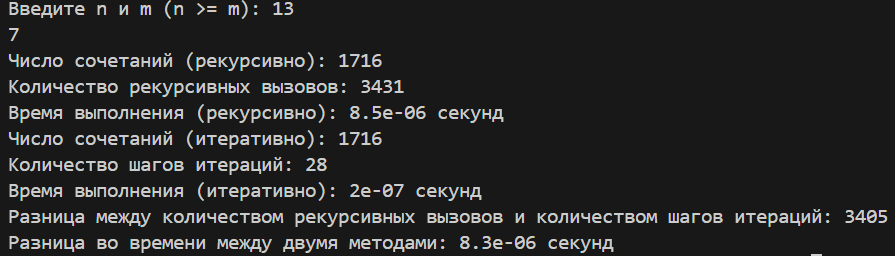


Рисунок 5. Результат для n = 13, m = 7.

Рекурсивный метод вычисления числа сочетаний оказывается медленнее, чем итеративный, особенно при больших значениях n и m. Итеративный метод не требует вложенных вызовов, что делает его предпочтительным для вычисления на больших объёмах данных.

**Задача №3:**

Задача описывает нахождение НОД с помощью алгоритма Евклида. Требуется найти наибольший общий делитель (НОД) двух чисел с помощью алгоритма Евклида, реализованного как в рекурсивной, так и в итерационной форме.

Рекурсивная триада:

1. Параметризация: a и b – два целых числа.
2. База рекурсии: если одно из чисел равно 0, результат равен другому числу.
3. Декомпозиция: НОД можно выразить как НОД(a,b)=НОД(b,a%b

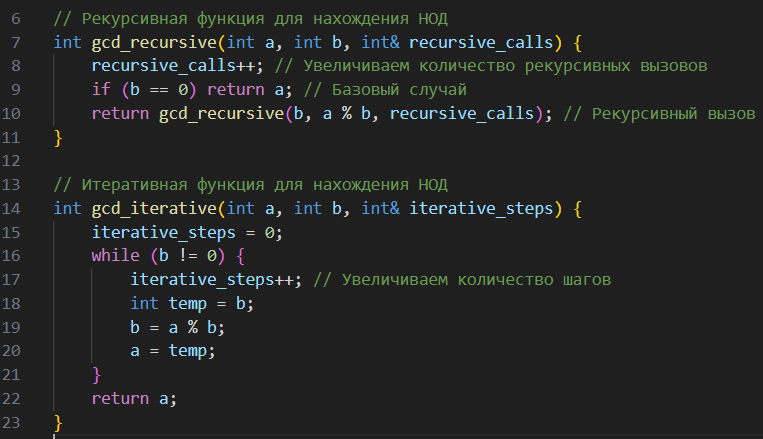


Рисунок 6. Код рекурсивной функции и итеративной функции для задачи №3.

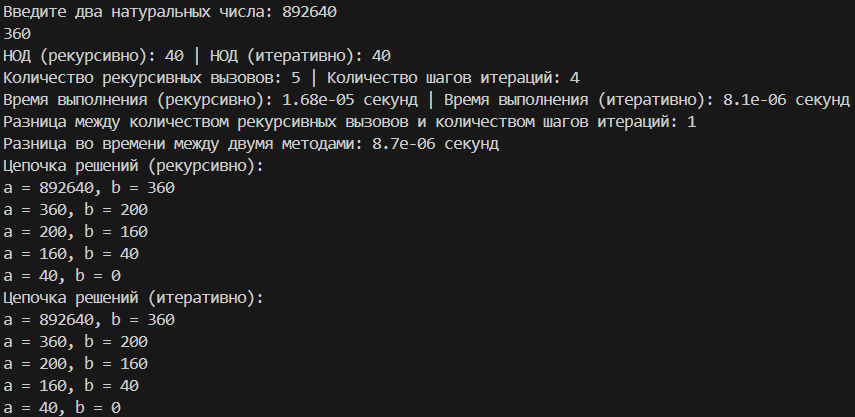


Рисунок 7. Результат для чисел 892640 и 360.

Рекурсивный и итеративный методы для нахождения НОД работают эффективно и практически одинаково по времени. Однако, итеративный метод не вызывает вложенных вызовов, что делает его более предпочтительным с точки зрения использования памяти.

**Задача №4**

Это задание подразумевает под собой нахождение суммы кубов цифр числа. Требуется вычислить сумму кубов цифр числа и повторять этот процесс, пока результат не станет равен 153. Задачу нужно решить как рекурсивным, так и итеративным методом.

Рекурсивная триада:

1. Параметризация: n – натуральное число, кратное 3.
2. База рекурсии: если n равно 153, функция завершает работу.
3. Декомпозиция: каждый новый результат равен сумме кубов цифр предыдущего числа.

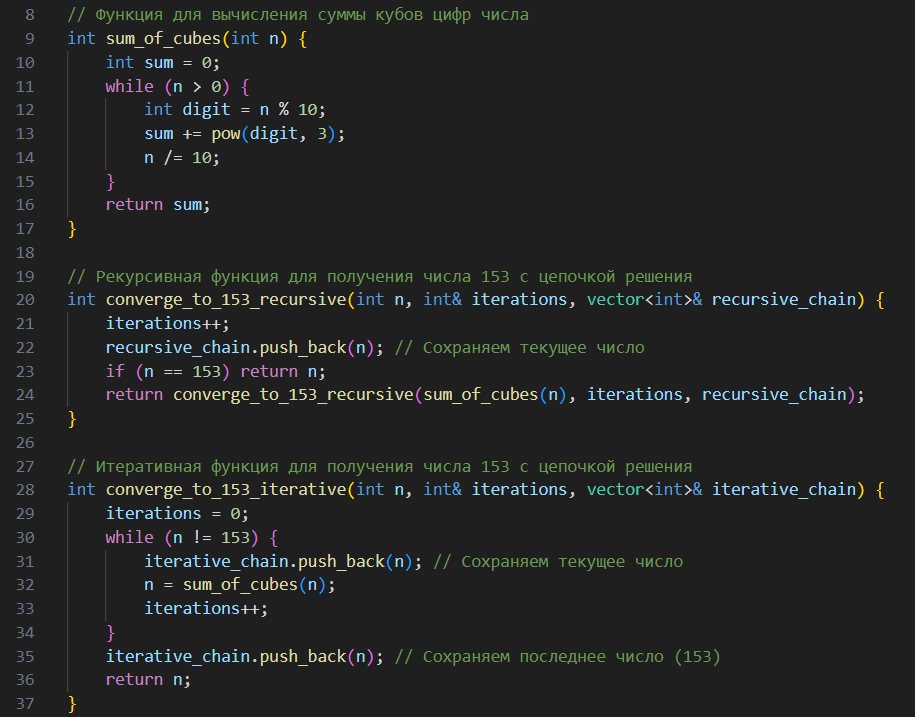


Рисунок 8. Главные функции для задачи №4.

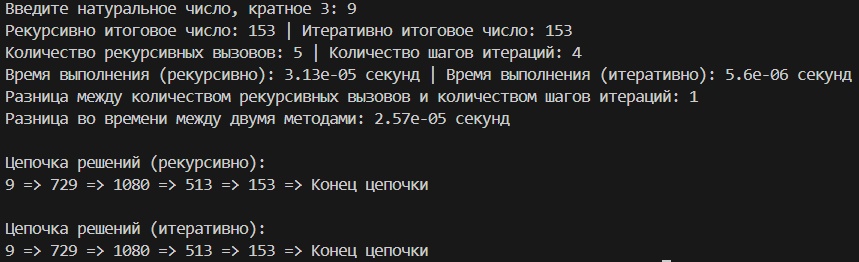


Рисунок 9. Результат для числа 9.

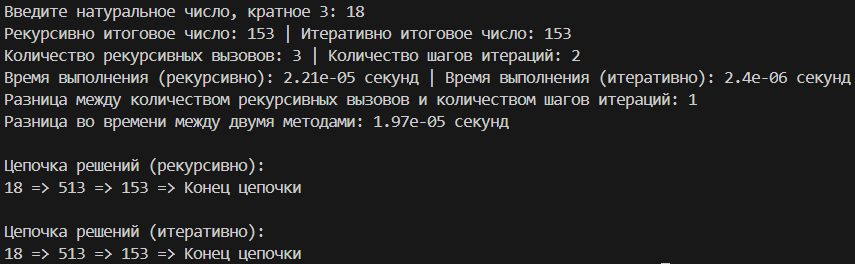


Рисунок 10. Результат для числа 18.

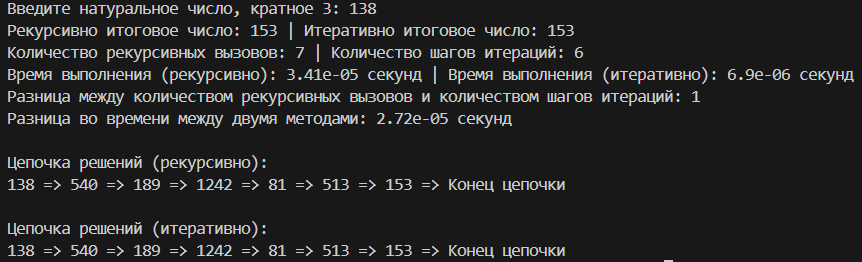


Рисунок 11. Результат для числа 138.

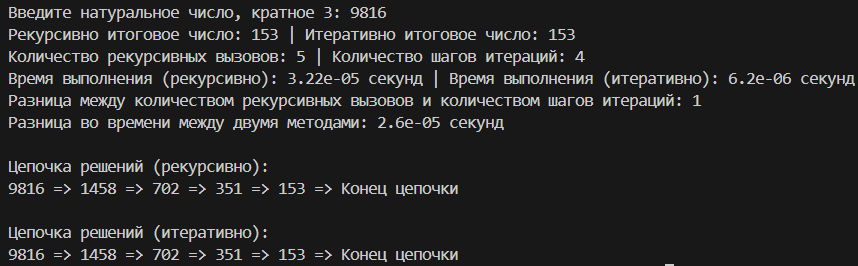


Рисунок 12. Результат для числа 9816.

Оба метода сходятся к числу 153, но итеративный подход более предпочтителен по времени и памяти. Рекурсия в данной задаче избыточна и увеличивает нагрузку на стек вызовов при больших числах.

**Задача №5:**

Задача сводится к получению числа 100 минимальным количеством операций. Исполнитель умеет выполнять два действия: прибавить 1 или умножить на 2. Необходимо составить рекурсивную функцию, которая минимальным количеством операций из числа 1 получит число 100.

Рекурсивная триада:

1. Параметризация: текущее число и целевое число.
2. База рекурсии: если текущее число равно целевому (100), возвращаем 0.
3. Декомпозиция: минимальное количество шагов равно минимуму между прибавлением 1 и умножением на 2: min\_operations(n) = 1 + min(min\_operations(n+1), min\_operations(n\*2))

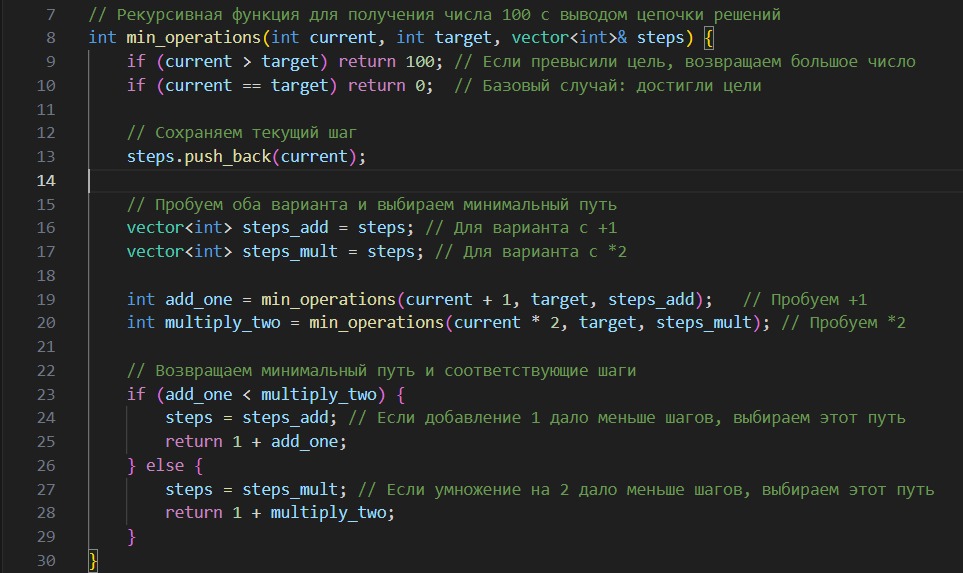


Рисунок 10. Полный код программы.



Рисунок 11. Результат задачи №5.

Рекурсивный метод эффективно решает задачу, но для больших целей может потребоваться оптимизация за счёт использования динамического программирования для запоминания уже рассчитанных шагов.

**Выводы:**

Рекурсия — это мощный инструмент для решения задач, но её использование должно быть обоснованным. Итерационные методы часто оказываются более эффективными по скорости и потреблению памяти, особенно для больших объёмов данных. В задачах, где результат может быть представлен с помощью рекурсивного подхода, можно использовать рекурсию для упрощения кода, но для критически важных вычислений лучше применять оптимизированные или итерационные методы.

**Приложение**

Листинг программы:

Задача №1

#include <iostream>

// рекурсивная функция для вычисления n-ого члена последовательности

int fibonacci(int n) {

    if (n == 0) return 0; // базовый случай

    if (n == 1) return 1; // базовый случай

    return fibonacci(n - 1) + fibonacci(n - 2); // рекурсивный вызов

}

int main() {

    int n;

    std::cout << "Введите n для нахождения члена последовательности: ";

    std::cin >> n;

    std::cout << "Член последовательности: " << fibonacci(n) << std::endl;

    return 0;

}

Задача №2

#include <iostream>

#include <chrono> // Для замеров времени

using namespace std;

// Рекурсивная функция для вычисления числа сочетаний

int combinations\_recursive(int n, int m, int& recursive\_calls) {

    recursive\_calls++; // Увеличиваем количество рекурсивных вызовов

    if (m == 0 || n == m) return 1; // Базовые случаи

    return combinations\_recursive(n - 1, m - 1, recursive\_calls) + combinations\_recursive(n - 1, m, recursive\_calls);

}

// Итеративная функция для вычисления факториала

long long factorial(int num, int& steps) {

    long long fact = 1;

    for (int i = 1; i <= num; ++i) {

        fact \*= i;

        steps++; // Увеличиваем количество шагов

    }

    return fact;

}

// Итеративная функция для вычисления числа сочетаний через факториалы

long long combinations\_iterative(int n, int m, int& iterative\_steps) {

    iterative\_steps = 0;

    long long nFact = factorial(n, iterative\_steps);

    long long mFact = factorial(m, iterative\_steps);

    long long nMinusMFact = factorial(n - m, iterative\_steps);

    return nFact / (mFact \* nMinusMFact);

}

int main() {

    int n, m;

    cout << "Введите n и m (n >= m): ";

    cin >> n >> m;

    int recursive\_calls = 0;

    auto start\_recursive = chrono::high\_resolution\_clock::now(); // Начало замера времени

    int result\_recursive = combinations\_recursive(n, m, recursive\_calls);

    auto end\_recursive = chrono::high\_resolution\_clock::now(); // Конец замера времени

    chrono::duration<double> time\_recursive = end\_recursive - start\_recursive;

    int iterative\_steps = 0;

    auto start\_iterative = chrono::high\_resolution\_clock::now(); // Начало замера времени

    long long result\_iterative = combinations\_iterative(n, m, iterative\_steps);

    auto end\_iterative = chrono::high\_resolution\_clock::now(); // Конец замера времени

    chrono::duration<double> time\_iterative = end\_iterative - start\_iterative;

    cout << "Число сочетаний (рекурсивно): " << result\_recursive << endl;

    cout << "Количество рекурсивных вызовов: " << recursive\_calls << endl;

    cout << "Время выполнения (рекурсивно): " << time\_recursive.count() << " секунд" << endl;

    cout << "Число сочетаний (итеративно): " << result\_iterative << endl;

    cout << "Количество шагов итераций: " << iterative\_steps + 2 << endl;

    cout << "Время выполнения (итеративно): " << time\_iterative.count() << " секунд" << endl;

    cout << "Разница между количеством рекурсивных вызовов и количеством шагов итераций: " << abs(recursive\_calls - iterative\_steps) << endl;

    cout << "Разница во времени между двумя методами: " << abs(time\_recursive.count() - time\_iterative.count()) << " секунд" << endl;

    return 0;

}

Задача №3

#include <iostream>

#include <chrono> // Для замеров времени

#include <vector> // Для хранения цепочек решений

using namespace std;

// Рекурсивная функция для нахождения НОД с цепочкой решений

int gcd\_recursive(int a, int b, int& recursive\_calls, vector<pair<int, int>>& recursive\_chain) {

    recursive\_calls++; // Увеличиваем количество рекурсивных вызовов

    recursive\_chain.push\_back({a, b}); // Добавляем текущие значения в цепочку решений

    if (b == 0) return a; // Базовый случай

    return gcd\_recursive(b, a % b, recursive\_calls, recursive\_chain); // Рекурсивный вызов

}

// Итеративная функция для нахождения НОД с цепочкой решений

int gcd\_iterative(int a, int b, int& iterative\_steps, vector<pair<int, int>>& iterative\_chain) {

    iterative\_steps = 0;

    while (b != 0) {

        iterative\_steps++; // Увеличиваем количество шагов

        iterative\_chain.push\_back({a, b}); // Добавляем текущие значения в цепочку решений

        int temp = b;

        b = a % b;

        a = temp;

    }

    iterative\_chain.push\_back({a, b}); // Добавляем финальное значение

    return a;

}

int main() {

    int a, b;

    cout << "Введите два натуральных числа: ";

    cin >> a >> b;

    // Для рекурсивного метода

    int recursive\_calls = 0;

    vector<pair<int, int>> recursive\_chain; // Хранение цепочки решений для рекурсии

    auto start\_recursive = chrono::high\_resolution\_clock::now(); // Начало замера времени

    int result\_recursive = gcd\_recursive(a, b, recursive\_calls, recursive\_chain);

    auto end\_recursive = chrono::high\_resolution\_clock::now(); // Конец замера времени

    chrono::duration<double> time\_recursive = end\_recursive - start\_recursive;

    // Для итеративного метода

    int iterative\_steps = 0;

    vector<pair<int, int>> iterative\_chain; // Хранение цепочки решений для итераций

    auto start\_iterative = chrono::high\_resolution\_clock::now(); // Начало замера времени

    int result\_iterative = gcd\_iterative(a, b, iterative\_steps, iterative\_chain);

    auto end\_iterative = chrono::high\_resolution\_clock::now(); // Конец замера времени

    chrono::duration<double> time\_iterative = end\_iterative - start\_iterative;

    // Вывод результатов

    cout << "НОД (рекурсивно): " << result\_recursive << " | НОД (итеративно): " << result\_iterative << endl;

    cout << "Количество рекурсивных вызовов: " << recursive\_calls << " | Количество шагов итераций: " << iterative\_steps << endl;

    cout << "Время выполнения (рекурсивно): " << time\_recursive.count() << " секунд" << " | Время выполнения (итеративно): " << time\_iterative.count() << " секунд" << endl;

    cout << "Разница между количеством рекурсивных вызовов и количеством шагов итераций: " << abs(recursive\_calls - iterative\_steps) << endl;

    cout << "Разница во времени между двумя методами: " << abs(time\_recursive.count() - time\_iterative.count()) << " секунд" << endl;

    // Вывод цепочки решений для рекурсивного метода

    cout << "Цепочка решений (рекурсивно):" << endl;

    for (const auto& step : recursive\_chain) {

        cout << "a = " << step.first << ", b = " << step.second << endl;

    }

    // Вывод цепочки решений для итеративного метода

    cout << "Цепочка решений (итеративно):" << endl;

    for (const auto& step : iterative\_chain) {

        cout << "a = " << step.first << ", b = " << step.second << endl;

    }

    return 0;

}

Задача №4

#include <iostream>

#include <cmath> // Для функции pow

#include <chrono> // Для измерения времени

#include <vector> // Для хранения цепочек решений

using namespace std;

// Функция для вычисления суммы кубов цифр числа

int sum\_of\_cubes(int n) {

    int sum = 0;

    while (n > 0) {

        int digit = n % 10;

        sum += pow(digit, 3);

        n /= 10;

    }

    return sum;

}

// Рекурсивная функция для получения числа 153 с цепочкой решения

int converge\_to\_153\_recursive(int n, int& iterations, vector<int>& recursive\_chain) {

    iterations++;

    recursive\_chain.push\_back(n); // Сохраняем текущее число

    if (n == 153) return n;

    return converge\_to\_153\_recursive(sum\_of\_cubes(n), iterations, recursive\_chain);

}

// Итеративная функция для получения числа 153 с цепочкой решения

int converge\_to\_153\_iterative(int n, int& iterations, vector<int>& iterative\_chain) {

    iterations = 0;

    while (n != 153) {

        iterative\_chain.push\_back(n); // Сохраняем текущее число

        n = sum\_of\_cubes(n);

        iterations++;

    }

    iterative\_chain.push\_back(n); // Сохраняем последнее число (153)

    return n;

}

int main() {

    int n;

    cout << "Введите натуральное число, кратное 3: ";

    cin >> n;

    // Рекурсивный подход

    int recursive\_iterations = 0;

    vector<int> recursive\_chain; // Для хранения цепочки решений (рекурсивно)

    auto start\_recursive = chrono::high\_resolution\_clock::now();

    int result\_recursive = converge\_to\_153\_recursive(n, recursive\_iterations, recursive\_chain);

    auto end\_recursive = chrono::high\_resolution\_clock::now();

    chrono::duration<double> time\_recursive = end\_recursive - start\_recursive;

    // Итеративный подход

    int iterative\_iterations = 0;

    vector<int> iterative\_chain; // Для хранения цепочки решений (итеративно)

    auto start\_iterative = chrono::high\_resolution\_clock::now();

    int result\_iterative = converge\_to\_153\_iterative(n, iterative\_iterations, iterative\_chain);

    auto end\_iterative = chrono::high\_resolution\_clock::now();

    chrono::duration<double> time\_iterative = end\_iterative - start\_iterative;

    // Вывод результатов

    cout << "Рекурсивно итоговое число: " << result\_recursive << " | Итеративно итоговое число: " << result\_iterative << endl;

    cout << "Количество рекурсивных вызовов: " << recursive\_iterations << " | Количество шагов итераций: " << iterative\_iterations << endl;

    cout << "Время выполнения (рекурсивно): " << time\_recursive.count() << " секунд" << " | Время выполнения (итеративно): " << time\_iterative.count() << " секунд" << endl;

    cout << "Разница между количеством рекурсивных вызовов и количеством шагов итераций: " << abs(recursive\_iterations - iterative\_iterations) << endl;

    cout << "Разница во времени между двумя методами: " << abs(time\_recursive.count() - time\_iterative.count()) << " секунд" << endl;

    // Вывод цепочки решений для рекурсивного метода

    cout << "\nЦепочка решений (рекурсивно):" << endl;

    for (const auto& step : recursive\_chain) {

        cout << step << " => ";

    }

    cout << "Конец цепочки" << endl;

    // Вывод цепочки решений для итеративного метода

    cout << "\nЦепочка решений (итеративно):" << endl;

    for (const auto& step : iterative\_chain) {

        cout << step << " => ";

    }

    cout << "Конец цепочки" << endl;

    return 0;

}

Задача №5

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Рекурсивная функция для получения числа 100 с выводом цепочки решений

int min\_operations(int current, int target, vector<int>& steps) {

    if (current > target) return 100; // Если превысили цель, возвращаем большое число

    if (current == target) return 0;  // Базовый случай: достигли цели

    // Сохраняем текущий шаг

    steps.push\_back(current);

    // Пробуем оба варианта и выбираем минимальный путь

    vector<int> steps\_add = steps; // Для варианта с +1

    vector<int> steps\_mult = steps; // Для варианта с \*2

    int add\_one = min\_operations(current + 1, target, steps\_add);   // Пробуем +1

    int multiply\_two = min\_operations(current \* 2, target, steps\_mult); // Пробуем \*2

    // Возвращаем минимальный путь и соответствующие шаги

    if (add\_one < multiply\_two) {

        steps = steps\_add; // Если добавление 1 дало меньше шагов, выбираем этот путь

        return 1 + add\_one;

    } else {

        steps = steps\_mult; // Если умножение на 2 дало меньше шагов, выбираем этот путь

        return 1 + multiply\_two;

    }

}

int main() {

    vector<int> steps; // Для хранения цепочки шагов

    int operations = min\_operations(1, 100, steps); // Старт с 1 и цель — 100

    // Вывод результата

    cout << "Минимальное количество операций: " << operations << endl;

    // Вывод цепочки решений

    cout << "Цепочка решений: ";

    for (int step : steps) {

        cout << step << " -> ";

    }

    cout << 100 << endl; // Финальное число 100

    return 0;

}