**НЕГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ ЧАСТНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«МОСКОВСКИЙ ФИНАНСОВО-ПРОМЫШЛЕННЫЙ УНИВЕРСИТЕТ «СИНЕРГИЯ»**

|  |  |  |
| --- | --- | --- |
| **Факультет/Институт** |  | Информационных технологий |
|  |  | (наименование факультета/ Института) |
| **Направление/специальность** |  | Информационные системы и программирование |
| **подготовки:** |  | (код и наименование направления /специальности подготовки) |
| **Форма обучения:** |  | Очная |
|  |  | (очная, очно-заочная, заочная) |
|  | | |

**Лабораторный практикум № *3***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **по дисциплине** |  | Разработка программных модулей | | |
|  | | |  | (наименование дисциплины) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Обучающийся** |  | **Балаганский В.О.** |  |  |
|  |  | (ФИО) |  | (подпись) |
| **Группа** |  | ***группа VДКИП-111прог*** |  |
|  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Преподаватель** |  | Сибирев И.В. |  |  |
|  |  | (ФИО) |  | (подпись) |

**Москва 2025 г.**

**1.3. Индивидуальные задания**

Нам нужно выбрать одно из 15 индивидуальных заданий, составить алгоритм в виде блок-схемы, написать программу с рекурсивной и нерекурсивной функциями, отладить её и сравнить результаты.

**Выбор задания: 4**

**Числа Фибоначчи определяются следующим образом: F(0)=0 F(0) = 0 F(0)=0, F(1)=1 F(1) = 1 F(1)=1, F(n)=F(n−1)+F(n−2) F(n) = F(n-1) + F(n-2) F(n)=F(n−1)+F(n−2). Определить F(n) F(n) F(n).**

**Решение задания 4: Числа Фибоначчи**

**1. Составление алгоритма в виде блок-схемы**

**Числа Фибоначчи можно вычислить двумя способами:**

* **Рекурсивно: F(n)=F(n−1)+F(n−2) F(n) = F(n-1) + F(n-2) F(n)=F(n−1)+F(n−2), с базовыми случаями F(0)=0 F(0) = 0 F(0)=0, F(1)=1 F(1) = 1 F(1)=1.**
* **Нерекурсивно: использовать цикл для последовательного вычисления чисел Фибоначчи.**

**Блок-схема для рекурсивной функции:**

**[Начало]**

**|**

**[Функция F(n)]**

**|**

**[Если n = 0?] --> Да --> [Вернуть 0]**

**| Нет**

**[Если n = 1?] --> Да --> [Вернуть 1]**

**| Нет**

**[Вернуть F(n-1) + F(n-2)]**

**|**

**[Конец]**

**Блок-схема для нерекурсивной функции:**

**[Начало]**

**|**

**[Функция F(n)]**

**|**

**[Если n = 0?] --> Да --> [Вернуть 0]**

**| Нет**

**[Если n = 1?] --> Да --> [Вернуть 1]**

**| Нет**

**[a = 0, b = 1, i = 2]**

**|**

**[Цикл: i <= n]**

**| Да**

**[c = a + b]**

**[a = b, b = c]**

**[i = i + 1]**

**| Нет**

**[Вернуть b]**

**|**

**[Конец]**

**2. Написание программы на C++**

**Теперь напишем программу, которая вычисляет F(n) F(n) F(n) рекурсивно и нерекурсивно, с интерфейсом, аналогичным примеру из документа.**

**#include <iostream>**

**using namespace std;**

**// Рекурсивная функция для вычисления числа Фибоначчи**

**long long Fibonacci\_R(int n) {**

**if (n == 0) return 0; // Базовый случай: F(0) = 0**

**if (n == 1) return 1; // Базовый случай: F(1) = 1**

**return Fibonacci\_R(n - 1) + Fibonacci\_R(n - 2); // Рекурсивный шаг**

**}**

**// Нерекурсивная функция для вычисления числа Фибоначчи**

**long long Fibonacci(int n) {**

**if (n == 0) return 0; // Базовый случай: F(0) = 0**

**if (n == 1) return 1; // Базовый случай: F(1) = 1**

**long long a = 0, b = 1, c;**

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

**c = a + b; // Следующее число Фибоначчи**

**a = b;**

**b = c;**

**}**

**return b;**

**}**

**int main() {**

**int n, kod;**

**while (true) {**

**cout << "\nInput n (n >= 0): ";**

**cin >> n;**

**// Проверка на корректность ввода**

**if (n < 0) {**

**cout << "Error: n must be non-negative!" << endl;**

**continue;**

**}**

**cout << "Recurs - 0\nSimple - 1\nElse - Exit" << endl;**

**cin >> kod;**

**switch (kod) {**

**case 0:**

**cout << "Recurs = " << Fibonacci\_R(n) << endl;**

**break;**

**case 1:**

**cout << "Simple = " << Fibonacci(n) << endl;**

**break;**

**default:**

**cout << "Exiting..." << endl;**

**return 0;**

**}**

**}**

**}**

**3. Отладка программы**

**Запустим программу и протестируем её на нескольких значениях n n n:**

* **n=0 n = 0 n=0:  
  Ожидаем: F(0)=0 F(0) = 0 F(0)=0.**

**Вывод:**

**Input n (n >= 0): 0**

**Recurs - 0**

**Simple - 1**

**Else - Exit**

**0**

**Recurs = 0**

**Input n (n >= 0): 0**

**Recurs - 0**

**Simple - 1**

**Else - Exit**

**1**

**Simple = 0**

**Результаты совпадают: F(0)=0 F(0) = 0 F(0)=0.**

**n=1 n = 1 n=1:  
Ожидаем: F(1)=1 F(1) = 1 F(1)=1.  
Вывод:**

**Input n (n >= 0): 1**

**Recurs - 0**

**Simple - 1**

**Else - Exit**

**0**

**Recurs = 1**

**Input n (n >= 0): 1**

**Recurs - 0**

**Simple - 1**

**Else - Exit**

**1**

**Simple = 1**

**Результаты совпадают: F(1)=1 F(1) = 1 F(1)=1.**

**n=5 n = 5 n=5:  
Ожидаем: F(5)=5 F(5) = 5 F(5)=5 (последовательность: 0, 1, 1, 2, 3, 5).  
Вывод:**

**Input n (n >= 0): 5**

**Recurs - 0**

**Simple - 1**

**Else - Exit**

**0**

**Recurs = 5**

**Input n (n >= 0): 5**

**Recurs - 0**

**Simple - 1**

**Else - Exit**

**1**

**Simple = 5**

**Результаты совпадают: F(5)=5 F(5) = 5 F(5)=5.**

**n=10 n = 10 n=10:  
Ожидаем: F(10)=55 F(10) = 55 F(10)=55 (последовательность: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55).  
Вывод:**

**Input n (n >= 0): 10**

**Recurs - 0**

**Simple - 1**

**Else - Exit**

**0**

**Recurs = 55**

**Input n (n >= 0): 10**

**Recurs - 0**

**Simple - 1**

**Else - Exit**

**1**

**Simple = 55**

**Результаты совпадают: F(10)=55 F(10) = 55 F(10)=55.**

**4. Сравнение результатов**

* **Корректность: Обе функции (рекурсивная и нерекурсивная) дают одинаковые результаты для всех протестированных значений n n n, что подтверждает правильность реализации.**
* **Производительность:** 
  + **Рекурсивная функция: Для больших n n n (например, n>40 n > 40 n>40) становится очень медленной, так как вызывает себя экспоненциально много раз (O(2n) O(2^n) O(2n)). Например, для n=45 n = 45 n=45 программа может работать несколько секунд.**
  + **Нерекурсивная функция: Работает значительно быстрее (O(n) O(n) O(n)), так как использует цикл и не создаёт дополнительных вызовов. Для n=45 n = 45 n=45 результат выводится мгновенно.**
* **Использование памяти:** 
  + **Рекурсивная функция: Использует стек вызовов, что может привести к переполнению стека при больших n n n.**
  + **Нерекурсивная функция: Использует фиксированное количество памяти (несколько переменных), независимо от n n n.**

**Вывод: Нерекурсивный метод предпочтительнее для больших n n n из-за лучшей производительности и меньшего потребления памяти.**

**1.4. Контрольные вопросы**

**Отлично, давай разберём контрольные вопросы из Лабораторной работы №1: Рекурсивные функции, которые ты указал. Я отвечу на каждый вопрос подробно, опираясь на теорию из документа и общее понимание рекурсии.**

**1. Какая функция называется рекурсивной?**

**Ответ:**

**Функция называется рекурсивной, если она в процессе своего выполнения вызывает саму себя для решения задачи. Рекурсия — это способ организации вычислительного процесса, при котором задача разбивается на меньшие подзадачи того же типа, и функция решает их, вызывая себя с новыми параметрами.**

**Для корректной работы рекурсивной функции должны быть выполнены два условия:**

* **Базовый случай (условие выхода): должен существовать случай, при котором функция возвращает результат без дальнейших рекурсивных вызовов. Это предотвращает бесконечный цикл вызовов.**
* **Рекурсивный шаг: каждый вызов функции должен приближать задачу к базовому случаю, уменьшая масштаб проблемы.**

**Пример из документа — вычисление факториала n! n! n!:**

**double Faktorial\_R(int n) {**

**if (n < 2) return 1; // Базовый случай: 0! = 1, 1! = 1**

**else return n \* Faktorial\_R(n - 1); // Рекурсивный вызов**

**}**

* **Здесь базовый случай: n<2 n < 2 n<2, возвращается 1.**
* **Рекурсивный шаг: n!=n⋅(n−1)! n! = n \cdot (n-1)! n!=n⋅(n−1)!, где (n−1)! (n-1)! (n−1)! вычисляется через вызов той же функции с параметром n−1 n-1 n−1.**

**Таким образом, рекурсивная функция — это функция, которая решает задачу, вызывая себя для решения подзадач, с чётко определённым условием выхода.**

**2. Может ли в реализации рекурсивной функции существовать несколько операторов передачи управления return?**

**Ответ:**

**Да, в реализации рекурсивной функции может существовать несколько операторов return, и это довольно распространённая практика. Наличие нескольких return позволяет функции возвращать результат в разных ситуациях, например, для обработки различных базовых случаев или промежуточных условий.**

**Пояснение**

* **Рекурсивная функция должна возвращать значение на каждом уровне рекурсии, чтобы передать результат обратно через стек вызовов.**
* **Разные return могут быть использованы для:** 
  + **Возврата результата в базовом случае.**
  + **Возврата результата после рекурсивного вызова.**
  + **Обработки исключительных ситуаций или ошибок (например, некорректных входных данных).**

**Пример из документа**

**В примере вычисления факториала из документа используется только один return в теле функции:**

**double Faktorial\_R(int n) {**

**if (n < 2) return 1; // Один return для базового случая**

**else return n \* Faktorial\_R(n - 1); // Второй return для рекурсивного случая**

**}**

**Здесь два оператора return:**

* **Первый возвращает 1, если n<2 n < 2 n<2 (базовый случай).**
* **Второй возвращает результат n⋅Faktorial\_R(n−1) n \cdot \text{Faktorial\\_R}(n-1) n⋅Faktorial\_R(n−1) для рекурсивного случая.**

**Более сложный пример**

**Рассмотрим задачу из индивидуальных заданий, например, задание 5 (функция Аккермана):**

**A(m,n)={n+1,если m=0,A(m−1,1),если m>0 и n=0,A(m−1,A(m,n−1)),если m>0 и n>0.A(m, n) = \begin{cases} n + 1, & \text{если } m = 0, \\ A(m-1, 1), & \text{если } m > 0 \text{ и } n = 0, \\ A(m-1, A(m, n-1)), & \text{если } m > 0 \text{ и } n > 0. \end{cases}A(m,n)=⎩⎨⎧​n+1,A(m−1,1),A(m−1,A(m,n−1)),​если m=0,если m>0 и n=0,если m>0 и n>0.​**

**Реализация на C++:**

**int Ackermann(int m, int n) {**

**if (m == 0) return n + 1; // Первый return: базовый случай**

**if (m > 0 && n == 0) return Ackermann(m - 1, 1); // Второй return: рекурсивный случай**

**if (m > 0 && n > 0) return Ackermann(m - 1, Ackermann(m, n - 1)); // Третий return: рекурсивный случай**

**return -1; // Четвёртый return: на случай некорректных входных данных (например, m < 0)**

**}**

**Здесь четыре оператора return:**

* **Первый для базового случая m=0 m = 0 m=0.**
* **Второй для случая m>0,n=0 m > 0, n = 0 m>0,n=0.**
* **Третий для случая m>0,n>0 m > 0, n > 0 m>0,n>0.**
* **Четвёртый для обработки ошибок (например, если m<0 m < 0 m<0).**

**Важное замечание**

* **Хотя несколько return допустимы, важно, чтобы каждый возможный путь выполнения функции завершался оператором return. Если этого не сделать, поведение программы будет неопределённым (undefined behavior).**
* **В некоторых случаях (например, для улучшения читаемости кода) можно минимизировать количество return, используя временные переменные и один общий return в конце функции. Например, функцию Аккермана можно переписать так:**

**int Ackermann(int m, int n) {**

**int result;**

**if (m == 0) {**

**result = n + 1;**

**} else if (m > 0 && n == 0) {**

**result = Ackermann(m - 1, 1);**

**} else if (m > 0 && n > 0) {**

**result = Ackermann(m - 1, Ackermann(m, n - 1));**

**} else {**

**result = -1; // Ошибка**

**}**

**return result; // Один return в конце**

**}**

**Однако это не обязательно, и использование нескольких return часто делает код более лаконичным и понятным.**

**Вывод**

**Да, в рекурсивной функции может быть несколько операторов return, и это полезно для обработки различных случаев (базовых, рекурсивных, ошибочных). Главное — обеспечить, чтобы каждый путь выполнения функции завершался возвратом значения.**