|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования |
| **«МИРЭА – Российский технологический университет»** |
| **РТУ МИРЭА** |
|  |

| **Отчет по выполнению практического задания № 7** | |
| --- | --- |
| **Тема:** | |
| **«Рекурсивные алгоритмы и их реализация»** | |
| Дисциплина: «Структуры и алгоритмы обработки данных» | |
|  | Выполнил студент: Гендриксон А.А. |
|  | Группа: ИКБО-74-23 |

Москва – 2024

СОДЕРЖАНИЕ

[1 ЦЕЛЬ 3](#_gjdgxs)

[2 ЗАДАНИЕ №1 4](#_30j0zll)

[2.1 Формулировка задачи (Вариант 6, в списке 6) 4](#_1fob9te)

[2.2 Итерационный алгоритм 4](#_2et92p0)

[2.3 Рекуррентная зависимость 6](#)

[2.4 Рекурсивная функция 6](#_17dp8vu)

[2.5 Программа с итерационным алгоритмом и рекурсивной функцией 7](#_1ci93xb)

[3 ЗАДАНИЕ №2 9](#_2bn6wsx)

[3.1 Формулировка задачи 9](#_qsh70q)

[3.2 Рекурсивная функция 9](#_u290pxmzmczz)

[4 ВЫВОДЫ 12](#_ihv636)

[5 ЛИТЕРАТУРА 13](#_32hioqz)

# **1 ЦЕЛЬ**

Получить знания и практические навыки по разработке и реализации рекурсивных процессов.

# **2 ЗАДАНИЕ №1**

## **2.1 Формулировка задачи (Вариант 6, в списке 6)**

Разработать и протестировать рекурсивные функции в соответствии с задачами варианта

Требования к выполнению первой задачи варианта:

• приведите итерационный алгоритм решения задачи

• реализуйте алгоритм в виде функции и отладьте его

• определите теоретическую сложность алгоритма

• опишите рекуррентную зависимость в решении задачи

• реализуйте и отладьте рекурсивную функцию решения задачи

• определите глубину рекурсии, изменяя исходные данные

• определите сложность рекурсивного алгоритма, используя метод подстановки и дерево рекурсии

• приведите для одного из значений схему рекурсивных вызовов

• разработайте программу, демонстрирующую выполнение обеих функций и покажите результаты тестирования.

Задание: Сколько квадратов можно отрезать от прямоугольника со сторонами a и b.

## **2.2 Итерационный алгоритм**

Начинаем с получения минимальной стороны прямоугольника. Это делается с помощью функции min(a, b), которая возвращает наименьшее из двух чисел. Инициализируем переменную squares нулем. Эта переменная будет использоваться для накопления количества квадратов. Запускаем цикл for, который будет выполняться от 1 до минимальной стороны прямоугольника включительно. В каждой итерации цикла вычисляем количество квадратов, которые можно отрезать с текущей длиной стороны. Для этого мы умножаем количество квадратов, которые можно отрезать по горизонтали (a - i + 1) на количество квадратов, которые можно отрезать по вертикали (b - i + 1). Это значение добавляется к переменной squares. После завершения цикла возвращаем значение переменной squares, которое представляет общее количество квадратов, которые можно отрезать от прямоугольника.

Реализуем алгоритм в блок кода 1.

| int countSquaresIterative(int a, int b) {  int minSide = min(a, b);  int squares = 0;   for (int i = 1; i <= minSide; ++i) {  squares += (a - i + 1) \* (b - i + 1);  }   return squares; } |
| --- |

Блок код 1 - Реализация итерационного алгоритма для задачи 1

Проведем отладку данной функции при a=4 и b = 3 (рис. 1).



Рисунок 1 - Тестирование алгоритма

Теоретическая сложность алгоритма определяется количеством итераций цикла for, который выполняется от 1 до минимальной стороны прямоугольника. Пусть минимальная сторона прямоугольника равна minSide. Тогда количество итераций равно minSide.

Внутри каждой итерации выполняется постоянное количество операций: вычисление (a - i + 1) \* (b - i + 1) и сложение этого значения с переменной squares. Таким образом, общее количество операций, выполняемых в цикле, будет пропорционально minSide.

Следовательно, теоретическая сложность алгоритма равна O(n).

## **2.3 Рекуррентная зависимость**

Рекуррентная зависимость:

T(a, b) = a \* b + T(a-1, b-1)

В этом случае:

При a = 0 или b = 0 достигается базовый случай, и рекурсия завершается, возвращая 0.

Иначе, происходит шаг в рекурсию, вычисляется количество квадратов для текущего значения сторон a и b, а затем вызывается рекурсивно для уменьшенных значений a-1 и b-1.

Это описывает, как текущее значение функции зависит от более мелких значений при каждом рекурсивном вызове, пока не достигнет базового случая.

## **2.4 Рекурсивная функция**

Реализуем рекурсивную функцию для данной задачи (блок кода 2).

| int countSquaresRecursive(int a, int b) {  if (a == 0 || b == 0) *// базовый случай*  return 0;  else  return a \* b + countSquaresRecursive(a - 1, b - 1); } |
| --- |

Блок кода 2 - Рекурсивная функция для задачи 1

Проведем отладку данной функции на тех же значениях, что и при отладке итерационного алгоритма (рис. 2).



Рисунок 2 - Тестирование рекурсивной программы

Глубина рекурсии для данной функции будет равна минимальному значению из a и b. Это происходит потому, что на каждом шаге одно из значений a или b уменьшается на 1, пока одно из них не станет равным 0.

Метод подстановки:

Давайте рассмотрим алгоритм для a = 3 и b = 2.

T(3, 2) = 3 \* 2 + T(2, 1)

= 6 + (2 \* 1 + T(1, 0))

= 6 + 2 + (1 \* 0 + T(0, 0))

= 8 + 0

= 8

Таким образом, T(3, 2) = 8.

Схема рекурсивных вызовов:

Для значений a = 3 и b = 2:

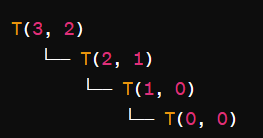


Рисунок 3 - Вызов рекурсии для массива значений

Это дерево рекурсии демонстрирует последовательные вызовы рекурсивной функции.

## **2.5 Программа с итерационным алгоритмом и рекурсивной функцией**

Реализуем программу, где будут присутствовать программы итерационного алгоритма и рекурсивной функции в блок коде 3 с выполнением обоих алгоритмов. Продемонстрируем результаты работы программы на рисунке 4.

| #include <iostream> using namespace std; *// Итерационный алгоритм* int countSquaresIterative(int a, int b) {  int minSide = min(a, b);  int squares = 0;   for (int i = 1; i <= minSide; ++i) {  squares += (a - i + 1) \* (b - i + 1);  }   return squares; }  *// Рекурсивная функция* int countSquaresRecursive(int a, int b) {  if (a == 0 || b == 0) *// базовый случай*  return 0;  else  return a \* b + countSquaresRecursive(a - 1, b - 1); }  int main() {  *// пример значений сторон прямоугольника*  int a = 4;  int b = 3;   *// Проверка итерационного алгоритма*  cout << "Итерационный подход: " << countSquaresIterative(a, b) << " квадратов\n";   *// Проверка рекурсивного алгоритма*  cout << "Рекурсивный подход: " << countSquaresRecursive(a, b) << " квадратов\n";   return 0; } |
| --- |

Блок кода 3 - Объединение программ



Рисунок 4 - Тестирование программы

# **3 ЗАДАНИЕ №2**

## **3.1 Формулировка задачи**

Требования к выполнению второй задачи варианта:

• рекурсивную функцию для обработки списковой структуры согласно варианту. Информационная часть узла – простого типа – целого;

• для создания списка может быть разработана простая или рекурсивная функция по желанию (в тех вариантах, где не требуется рекурсивное создание списка);

• определите глубину рекурсии

• определите теоретическую сложность алгоритма

• разработайте программу, демонстрирующую работу функций и покажите результаты тестов.

Задание: Удаление связанного стека

## **3.2 Рекурсивная функция**

Реализуем задачу с помощью рекурсивной функции(блок кода 4).

| #include <iostream> using namespace std;   *// Структура для узла в связанном списке* struct Node {  int data;  Node\* next;    *// Конструктор для удобного создания узла*  Node(int val) : data(val), next(nullptr) {} };  *// Функция для добавления элемента в начало стека* void push(Node\*& top, int val) {  Node\* newNode = new Node(val);  newNode->next = top;  top = newNode; }  *// Рекурсивная функция для удаления всех элементов из стека* void deleteStack(Node\*& top) {  if (top == nullptr) *// базовый случай: стек пуст*  return;    Node\* temp = top;  top = top->next;  delete temp;    deleteStack(top); *// рекурсивный вызов для удаления следующего элемента* }  *// Функция для отображения содержимого стека (для тестирования)* void displayStack(Node\* top) {  cout << "Содержимое стека:\n";  while (top != nullptr) {  cout << top->data << " ";  top = top->next;  }  cout << endl; }  int main() {  *// Создание стека и добавление элементов*  Node\* stackTop = nullptr;  push(stackTop, 3);  push(stackTop, 7);  push(stackTop, 11);    *// Отображение начального содержимого стека*  displayStack(stackTop);    *// Удаление стека*  deleteStack(stackTop);  cout << "После удаления\n";  *// Отображение содержимого стека после удаления*  displayStack(stackTop); *// Должно вывести "Содержимое стека:"*   return 0; } |
| --- |

Блок кода 4 - Программа для задания 2 с рекурсивной функцией

Продемонстрируем результаты работы программы на рисунке 4.

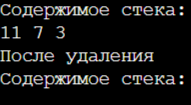


Рисунок 4 - Тестирование программы

Теоретическая сложность алгоритма удаления стека будет O(n), где n - количество элементов в стеке. Это потому, что на каждом шаге рекурсии мы удаляем один элемент из стека, и это происходит n раз, пока весь стек не будет удален.

Глубина рекурсии будет равна количеству элементов в стеке, так как на каждом рекурсивном вызове мы обрабатываем один элемент и вызываем рекурсивно функцию удаления для оставшейся части стека.

Например, если у нас есть стек из 5 элементов, то функция удаления будет вызвана 5 раз для удаления каждого элемента, что приведет к глубине рекурсии равной 5.

# **4 ВЫВОДЫ**

Задания "Удаление связанного стека" и "Сколько квадратов можно отрезать от прямоугольника со сторонами a и b, реализованные с помощью рекурсии", оба направлены на изучение и применение рекурсивных процессов.

Сколько квадратов можно отрезать от прямоугольника со сторонами a и b: Это задание требует реализации рекурсивной функции для вычисления количества квадратов, которые можно отрезать от прямоугольника. Рекурсивный алгоритм здесь связан с логикой прямоугольника и количеством квадратов, которые можно отрезать от его сторон. Понимание рекурсивной зависимости в решении этой задачи помогает развить навыки анализа и разработки рекурсивных алгоритмов.

Удаление связанного стека: Это задание позволяет понять, как реализовать рекурсивный алгоритм для удаления элементов из связанного стека. Понимание рекурсивного удаления структуры данных помогает развить навыки работы с рекурсией и понимание ее принципов. Глубина рекурсии в этом задании зависит от количества элементов в стеке, что помогает осознать связь между структурой данных и рекурсивным алгоритмом обработки этой структуры.

Оба этих задания важны для понимания рекурсии и ее применения в различных сценариях. Практическое выполнение этих задач помогает закрепить теоретические знания и развить навыки разработки рекурсивных процессов.

# 

# **5 ЛИТЕРАТУРА**

1. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб: Питер, 2017. – 288 с.

2. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. – 406 с.

3. Кнут Д.Э. Искусство программирования, том 3. Сортировка и поиск, 2-е изд. – М.: ООО «И.Д. Вильямс», 2018. – 832 с.

4. Кораблин Ю.П. Структуры и алгоритмы обработки данных: учебно-методическое пособие / Ю.П. Кораблин, В.П. Сыромятников, Л.А. Скворцова. – М.: РТУ МИРЭА, 2020. — 219 с.

5. Кормен Т.Х. и др. Алгоритмы: построение и анализ, 3-е изд. – М.: ООО «И.Д.Вильямс», 2013. – 1328 с.

6. Макконнелл Дж. Основы современных алгоритмов. Активный обучающий метод. 3-е доп. изд., - М.: Техносфера, 2018. – 416 с.

7. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. – К.: Издательство «Диасофт», 2001. – 688 с.

8. Скиена С. Алгоритмы. Руководство по разработке, - 2-е изд. – СПб: БХВ-Петербург, 2011. – 720 с.

9. Хайнеман Д. и др. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд. – СПб: ООО «Альфа-книга», 2017. – 432 с.

10. AlgoList – алгоритмы, методы, исходники [Электронный ресурс]. URL: http://algolist.manual.ru/ (дата обращения 15.03.2022).

11. Алгоритмы – всё об алгоритмах / Хабр [Электронный ресурс]. URL: https://habr.com/ru/hub/algorithms/ (дата обращения 15.03.2022).