|  |
| --- |
| 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: Харченко А.А. |
|  | Группа: ИКБО-74-23 |

Москва – 2024

СОДЕРЖАНИЕ

[1 ЦЕЛЬ 3](#_gjdgxs)

[2 ЗАДАНИЕ №1 3](#_30j0zll)

[2.1 Формулировка задачи 3](#_1fob9te)

[2.2 Итерационный алгоритм 3](#_2et92p0)

[2.3 Рекуррентная зависимость 6](#)

[2.4 Рекурсивная функция 6](#_17dp8vu)

[2.5 Объединение программ 9](#_1ci93xb)

[3 ЗАДАНИЕ №2 12](#_2bn6wsx)

[3.1 Формулировка задачи 12](#_qsh70q)

[3.2 Рекурсивная функция 12](#_u290pxmzmczz)

[4 ВЫВОДЫ 14](#_ihv636)

[5 ЛИТЕРАТУРА 15](#_32hioqz)

# **1 ЦЕЛЬ**

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

# **2 ЗАДАНИЕ №1**

## **2.1 Формулировка задачи**

Вариант 11, в списке 27.

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: Вычислить x1(x2+x3)(x4+x5+x6)....(x46+x47+...+x55)

## **2.2 Итерационный алгоритм**

Инициализируем переменную result значением 1. Эта переменная будет содержать результат вычисления всего выражения. Устанавливаем начальное значение переменной x равным 1, так как первый элемент 𝑥1  равен 1.

Проходим по последовательности групп по 9 элементов. Начиная с 𝑖=2 (первая группа), идем до i=46 (последняя группа) с шагом 9. Для каждой группы элементов (𝑥𝑖, 𝑥𝑖 + 1,…, 𝑥𝑖+8) вычисляем сумму элементов группы и сохраняем ее в переменной sum. Умножаем текущее значение result на sum, чтобы обновить результат вычисления. Переходим к следующей группе. Возвращаем итоговый результат.

Алгоритм завершается после прохождения всех групп элементов и корректно вычисляет значение заданного выражения.

Реализация данного алгоритма представленная в функции calculateExpressionIterative(блок кода 1)

| // Итерационная функция для вычисления выражения **long** **long** **calculateExpressionIterative**() {  **long** **long** result = 1; // Инициализация результата  **long** **long** x = 1; // Начальное значение x    **for** (**int** i = 2; i <= 46; i += 9) { // Проходим по группам элементов с шагом 9  **long** **long** sum = 0; // Инициализация суммы группы    // Суммируем элементы в текущей группе  **for** (**int** j = i; j <= i + 8; ++j) {  sum += x + j;  }    result \*= sum; // Обновляем результат, умножая на сумму текущей группы  }    **return** result; // Возвращаем итоговый результат } |
| --- |

Блок схема 1 - Реализация итерационного алгоритма для задачи 1

Проведем отладку данной функции (рис. 1).



Рисунок 1 - Тестирование алгоритма

Итерационный алгоритм имеет теоретическую сложность O(N).

## **2.3 Рекуррентная зависимость**

f(i) представляет собой результат выражения для подвыражения, начинающегося с элемента xi. Если i превышает 46, мы достигли конца последовательности, и возвращаем 1. В противном случае мы складываем элемент 𝑥𝑖 с результатом выражения для подвыражения, начинающегося с элемента 𝑥i+1. Это означает, что мы решаем задачу рекурсивно, пошагово обрабатывая каждый элемент последовательности.

## **2.4 Рекурсивная функция**

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

| // Рекурсивная функция для вычисления выражения **long** **long** **calculateExpressionRecursive**(**int** i, **long** **long** x) {  **if** (i > 46) {  **return** 1;  }    **long** **long** sum = 0;  **for** (**int** j = i; j <= i + 8; ++j) {  sum += x + j;  }    **return** sum \* calculateExpressionRecursive(i + 9, x); } |
| --- |

Блок кода 2 - Рекурсивная функция для задачи 1

Проведем отладку данной функции(рис. 2).



Рисунок 2 - Тестирование адаптированной программы

Глубина рекурсии в данном случае определяется количеством вызовов функции до достижения базового случая. В данной функции базовый случай находится в условии if (i > 46), когда переменная i превышает 46.

Так как функция вызывается с аргументом i, начиная с 2 и увеличиваясь на 9 при каждом рекурсивном вызове до достижения 46, мы можем оценить количество рекурсивных вызовов.

Начнем с i = 2:

Первый вызов: i = 2

Второй вызов: i = 11

Третий вызов: i = 20

...

Последний вызов: i = 47

Количество рекурсивных вызовов можно вычислить как количество шагов от 2 до 46 с шагом 9:

44/9+1=5

Таким образом, глубина рекурсии составляет примерно 5 вызовов.

Определим сложность рекурсивной функции.

Метод подстановки:

Предположение: Пусть T(n) - сложность рекурсивной функции для вычисления выражения, где n - количество элементов в последовательности (в нашем случае n=55).

Базовый случай: Если n≤0, то выполнение функции завершается, и сложность составляет O(1).

Рекурсивный случай: При каждом рекурсивном вызове функция делает O(9) операций на суммирование элементов в текущей группе и еще один рекурсивный вызов. Таким образом, сложность рекурсивного случая может быть записана как T(n)=O(9)+T(n−9).

Раскрытие рекурсии: Продолжая раскрывать рекурсию, мы обнаружим, что функция будет вызвана 𝑛/9 раз, прежде чем достигнет базового случая.

Объединение: Общая сложность рекурсивного алгоритма будет составлять сумму операций во всех вызовах, то есть:

T(n)=O(9)+O(9)+⋯+O(9)= n/9 ×O(9)=O(n)

Таким образом, метод подстановки показывает, что сложность рекурсивного алгоритма составляет O(n).

Дерево рекурсии:

Построим дерево рекурсии для вычисления сложности алгоритма.

| T(n)  / \  T(n-9) T(n-9)  / \ / \  T(n-18) ... T(n-18)  / \ / \  T(n-27) ... T(n-27)  / \ / \  . . . . |
| --- |

Как видно из дерева, каждый уровень дает O(9) операций, и количество уровней равно 𝑛/9 . Таким образом, общее количество операций составляет O(n).

Таким образом, как метод подстановки, так и дерево рекурсии показывают, что сложность рекурсивного алгоритма составляет O(n).

## **2.5 Объединение программ**

Объединим программы итерационного алгоритма и рекурсивной функции в блок коде 3 с выполнением обоих алгоритмов. Продемонстрируем результаты работы программы на рисунке 3.

| #include <iostream> **using** **namespace** std;  // Итерационная функция для вычисления выражения **long** **long** **calculateExpressionIterative**() {  **long** **long** result = 1;  **long** **long** x = 1;    **for** (**int** i = 2; i <= 46; i += 9) {  **long** **long** sum = 0;  **for** (**int** j = i; j <= i + 8; ++j) {  sum += x + j;  }  result \*= sum;  }    **return** result; }  // Рекурсивная функция для вычисления выражения **long** **long** **calculateExpressionRecursive**(**int** i, **long** **long** x) {  **if** (i > 46) {  **return** 1;  }    **long** **long** sum = 0;  **for** (**int** j = i; j <= i + 8; ++j) {  sum += x + j;  }    **return** sum \* calculateExpressionRecursive(i + 9, x); }  **int** **main**() {  **long** **long** x = 1;    // Вычисление с использованием итеративного алгоритма  cout << "Результат итерационного вычисления: " << calculateExpressionIterative() << endl;    // Вычисление с использованием рекурсивного алгоритма  cout << "Результат рекурсивного вычисления: " << calculateExpressionRecursive(2, x) << endl;    **return** 0; } |
| --- |

Блок кода 3 - Объединение программ



Рисунок 3 - 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 двунаправленного списка

## **3.2 Рекурсивная функция**

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

| #include <iostream> **using** **namespace** std; **struct** **Node** {  **int** data;  Node\* prev;  Node\* next;    Node(**int** value) : data(value), prev(**nullptr**), next(**nullptr**) {} };  // Рекурсивная функция удаления списка **void** **deleteLinkedList**(Node\* head) {  **if** (head == **nullptr**) {  **return**;  }    // Рекурсивно вызываем функцию для удаления оставшихся узлов  deleteLinkedList(head->next);    // Удаляем текущий узел  **delete** head; }  // Функция для добавления нового узла в конец списка **void** **append**(Node\*\* head\_ref, **int** new\_data) {  Node\* new\_node = **new** Node(new\_data);  Node\* last = \*head\_ref;  new\_node->next = **nullptr**;    **if** (\*head\_ref == **nullptr**) {  new\_node->prev = **nullptr**;  \*head\_ref = new\_node;  **return**;  }    **while** (last->next != **nullptr**) {  last = last->next;  }    last->next = new\_node;  new\_node->prev = last; }  // Функция для отображения списка **void** **displayList**(Node\* node) {  **while** (node != **nullptr**) {  cout << node->data << " ";  node = node->next;  }  cout << endl; }  **int** **main**() {  // Создаем двунаправленный список  Node\* head = **nullptr**;  append(&head, 1);  append(&head, 2);  append(&head, 3);  append(&head, 4);  append(&head, 5);    // Выводим исходный список  cout << "Исходный список: ";  displayList(head);    // Удаляем список  deleteLinkedList(head);  head = **nullptr**;    cout << "Список удален" << endl;    **return** 0; } |
| --- |

Блок кода 4 - Программа для задания 2 с рекурсивной функцией

В данной рекурсивной функции глубина рекурсии определяется количеством вызовов функции до достижения базового случая. В функции deleteLinkedList, базовый случай проверяется наличием указателя head, который равен nullptr. Каждый раз, когда функция вызывается рекурсивно, она передает указатель на следующий узел списка. Таким образом, глубина рекурсии определяется количеством узлов в списке.

Теоретическая сложность алгоритма удаления двунаправленного списка с использованием рекурсии зависит от количества узлов в списке. Поскольку каждый узел должен быть удален, сложность будет линейной по числу узлов. Для каждого узла выполняется константное количество операций (освобождение памяти и вызов рекурсивной функции для следующего узла). Поскольку у нас есть n узлов, где n - количество элементов в списке, общая сложность будет O(n). Таким образом, теоретическая сложность алгоритма удаления двунаправленного списка с использованием рекурсии составляет O(n), где n - количество узлов в списке.

Продемонстрируем результаты работы программы на рисунке 4.



Рисунок 4 - Тестирование программы

# 

# **4 ВЫВОДЫ**

В процессе выполнения данной практической работы были разработаны программы на языке C++, демонстрирующие работу рекурсивных функций.

В задании "Вычислить x1(x2+x3)(x4+x5+x6)....(x46+x47+...+x55)" было предложено вычислить значение выражения x1(x2+x3)(x4+x5+x6)....(x46+x47+...+x55). Этот алгоритм был реализован с использованием рекурсии. Теоретическая сложность этого алгоритма составляет O(n), где n - количество элементов в последовательности.

В задании "Удаление двунаправленного списка" была рассмотрена рекурсивная функция удаления элементов из списка. Рекурсивный подход позволяет эффективно удалить все узлы списка, начиная с головы и перемещаясь к концу. Теоретическая сложность алгоритма удаления составляет O(n), где n - количество узлов в списке.

Оба рекурсивных подхода эффективно решают свои задачи, а их сложность линейно зависит от количества элементов во входных данных.

# 

# **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).