МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждениевысшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №4

Рекурсивные алгоритмы и их реализации
по дисциплине
«СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ»

Выполнил студент Иолович Е.А.

группа ИНБО-03-22

Москва 2023

СОДЕРЖАНИЕ

0 ОТВЕТЫ НА ВОПРОСЫ 3

1 ОТЧЕТ ПО ЗАДАЧЕ 1 6

1.1 Условие задачи 6

1.2 Постановка задачи 6

1.3 Описание алгоритма – рекуррентная зависимость 6

1.4 Код используемой функции (SumofDigits) 7

1.5 Код программы и скриншоты результатов тестирования 8

2 ОТЧЕТ ПО ЗАДАНИЮ 2 9

2.1 Условие задачи 9

2.2 Постановка задачи 9

2.3 Описание алгоритма – рекуррентная зависимость 9

2.4 Код используемой функции (countEven) 9

2.5 Код программы и скриншоты результатов тестирования 10

3 ВЫВОДЫ 12

4 СПИСОК ЛИТЕРАТУРЫ 13

0 ОТВЕТЫ НА ВОПРОСЫ

1) Определение рекурсивной функции – рекурсивной функцией называется функция, частично состоящей из самой себя или определяемой с помощью себя.

2) Шаг рекурсии – это активизация очередного рекурсивного выполнения алгоритма при других исходных данных.

3) Глубина рекурсии – наибольшее одновременное количество

рекурсивных вызовов функции, определяющее максимальное количество слоев рекурсивного стека, в котором осуществляется хранение отложенных вычислений.

4) Условие завершения рекурсии – это условие, которое, при его выполнении, остановит вызов рекурсивной функции самой себя.

5) Виды рекурсии:

Линейная – при которой рекурсивные вызовы на любом рекурсивном срезе, инициируют не более одного последующего рекурсивного вызова, называется линейной. Это наиболее простой и часто встречающийся тип рекурсии. Большая часть ранее разобранных примеров относилась именно к этому типу рекурсии.

Каскадная – рекурсивные обращения, как правило, приводят к необходимости многократно решать одни и те же подзадачи

6) Прямая и косвенная рекурсия:

Косвенная рекурсия имеет место, если алгоритм А вызывает алгоритм В, и алгоритм В вновь вызывает алгоритм А.

Прямая рекурсия имеет место, если решение задачи сводится к разделению ее на меньшие подзадачи, выполняемые с помощью одного и того же алгоритма.

7) Организация стека рекурсивных вызовов – это механизм работы с рекурсией в компьютерных программах, который основан на использовании стека – структуры данных, в которой элементы добавляются и удаляются

только в определенном порядке. При вызове рекурсивной функции, компьютер создает новый экземпляр функции в памяти и помещает его в вершину стека. После того, как рекурсивный вызов завершается, компьютер удаляет его из вершины стека и продолжает работу с предыдущим вызовом. Таким образом, каждый последующий вызов рекурсивной функции добавляется в вершину стека, а предыдущий вызов продолжает работу только после того, как завершится текущий вызов. Организация стека рекурсивных вызовов влияет на использование памяти и скорость выполнения программы. Если вызовы функций глубоко вложены, то может произойти переполнение стека, и программа завершится с ошибкой.

8) Стек рекурсивных вызовов. Модель формирования для алгоритма вычисления n!.

При каждом новом рекурсивном вызове функции в стеке создается новое множество локальных переменных, их имена одинаковы, но они имеют различные значения.

9) Дерево рекурсии вычисления 5! (рис. 2) и пятого числа Фибоначчи (рис. 1).

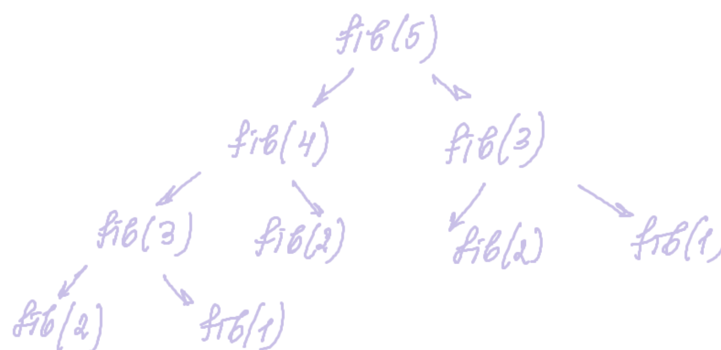

Рисунок 1 – Дерево рекурсии вычисления пятого числа Фибоначчи

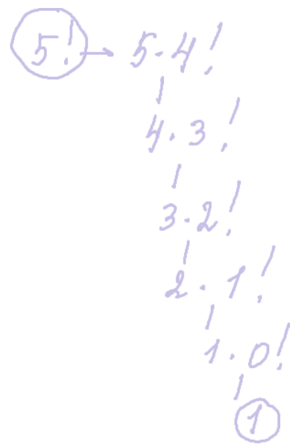

Рисунок 2 – Дерево рекурсии вычисления факториала 5 числа

1 ОТЧЕТ ПО ЗАДАЧЕ 1

1.1 Условие задачи

Вычислить значение цифрового корня для некоторого целого числа N.

1.2 Постановка задачи

Разработать и протестировать рекурсивные функции в соответствии с задачей варианта.

1.3 Описание алгоритма – рекуррентная зависимость

Данная программа имеет рекурсивную зависимость в функции SumofDigits(). Если сумма цифр введенного числа N больше или равна 10, то функция вызывает саму себя с аргументом sum, где sum - это сумма цифр N. Рекурсивные вызовы функции SumofDigits() продолжаются до тех пор, пока не будет получено число с суммой цифр меньше 10. Каждый рекурсивный вызов использует новое значение sum, вычисленное в предыдущем вызове функции.

Глубина рекурсии в данной программе зависит от количества цифр в исходном числе. Если число состоит из N цифр, то глубина рекурсии будет равна N-1, т.к. на каждом уровне вызывается функция с числом, сумма цифр которого меньше, чем в исходном числе, и так до тех пор, пока не останется одна цифра. Например, для числа 12345 глубина рекурсии будет равна 4, т.к. на каждом уровне будет вызываться функция с числом, сумма цифр которого меньше, чем в предыдущем: 12345 -> 15 -> 6 -> 6.

Метод подстановки:

Каждый раз, когда происходит вызов функции SumofDigits, аргумент N уменьшается в 10 раз, то есть размер входных данных уменьшается в 10 раз. Кроме того, выполняется постоянное количество операций над каждым N внутри функции. Таким образом, сложность функции можно оценить как O(log N), где N - входной аргумент функции (Здесь O обозначает "асимптотическую сложность" алгоритма. Она показывает, как быстро растет

время выполнения алгоритма при увеличении размера входных данных).

Дерево рекурсии:

Каждый вызов функции SumofDigits порождает два вызова: SumofDigits(sum) и один вызов внутри while. В худшем случае, когда сумма цифр в числе равна 99 (например, для числа 999), дерево рекурсии будет иметь высоту 2, так как на каждом уровне аргумент N будет уменьшаться в 10 раз. Таким образом, количество вызовов функции будет равно 2log N, что также дает оценку сложности O(log N).

Итак, оба метода подтверждают, что сложность рекурсивной функции SumofDigits составляет O(log N).

Для значения N = 12345 схема рекурсивных вызовов будет выглядеть следующим образом:

```
SumofDigits(12345)
  d = 5
  sum = 5
  N = 1234
  SumofDigits(1234)
    d = 4
    sum = 9
    N = 123
    SumofDigits(123)
      d = 3
      sum = 12
      N = 12
      SumofDigits(12)
        d = 2
        sum = 14
        N = 1
        SumofDigits(1)
          return 1
        return 1
      return 5
    return 5
  return 6
```

1.4 Код используемой функции (SumofDigits)

```
int SumofDigits(int N)
{
    int d = 0;
    int sum = 0;
    while (N != 0)
    {
        d = N % 10;
        sum = sum + d;
        N = N / 10;
    }
    if (sum < 10)
```

```
    {
        return sum;
    }
    else
        SumofDigits(sum);
}
```

1.5 Код программы и скриншоты результатов тестирования

```
#include <iostream> //1_задание
using namespace std;


int SumofDigits(int N)
{
    int d = 0;
    int sum = 0;
    while (N != 0)
    {
        d = N % 10;
        sum = sum + d;
        N = N / 10;
    }
    if (sum < 10)
    {
        return sum;
    }
    else
        SumofDigits(sum);
}

int main()
{
    setlocale(LC_ALL, "Ru");
    int expression;
    cout << "Введите цифровое значение для вычисления цифрового корня: ";
    cin >> expression;
    cout << SumofDigits(expression);
}
```

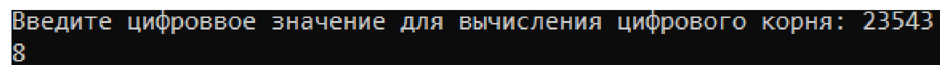

Рисунок 3 – Тестирование программы, вычисляющей цифровой корень

2 ОТЧЕТ ПО ЗАДАНИЮ 2

2.1 Условие задачи

Найти в двунаправленном списке количество четных элементов.

2.2 Постановка задачи

Разработать и протестировать рекурсивные функции в соответствии с задачей варианта.

2.3 Описание алгоритма – рекуррентная зависимость

Рекуррентная зависимость данной программы заключается в том, что функция countEven вызывает саму себя с параметром node->next, то есть с указателем на следующий элемент списка. Таким образом, рекурсивный вызов функции происходит на каждом шаге по списку до тех пор, пока не будет достигнут конец списка (указатель на последний элемент списка будет равен nullptr). Каждый раз при вызове функции снова проверяется, является ли текущий элемент четным, и в зависимости от этого либо увеличивается счетчик четных элементов, либо продолжается рекурсивный обход списка. Таким образом, рекуррентная зависимость заключается в том, что количество четных элементов в списке равно сумме количества четных элементов в оставшейся части списка, сложенной с единицей, если текущий элемент является четным. Если текущий элемент нечетный, то количество четных элементов в оставшейся части списка равно количеству четных элементов в этой части списка.

2.4 Код используемой функции (countEven)

```
int countEven(Node* node)
{
    if (node == nullptr)
    {
        return 0;
    }
    if (node->data % 2 == 0)
    {
        return 1 + countEven(node->next);
    }
    return countEven(node->next);
}
```

2.5 Код программы и скриншоты результатов тестирования

```
#include <iostream> //2_задание
using namespace std;

ы
struct Node
{
    int data;
    Node* prev;
    Node* next;
};

int countEven(Node* node)
{
    if (node == nullptr)
    {
        return 0;
    }
    if (node->data % 2 == 0)
    {
        return 1 + countEven(node->next);
    }
    return countEven(node->next);
}

int main()
{
    setlocale(LC_ALL, "Ru");

    // Переменная, в которую будем сохранять вводимые значения
    int value;

    // Голова списка, изначально установлена в nullptr
    Node* head = nullptr;

    // Текущий узел списка
    Node* current = nullptr;

    // Запросим у пользователя количество элементов списка
    int n;
    cout << "Введите количество элементов списка: ";
    cin >> n;

    // Заполним список элементами, вводимыми пользователем
    for (int i = 0; i < n; i++)
    {
        cout << "Введите элемент №" << i + 1 << ": ";
        cin >> value;
        Node* newNode = new Node{ value, current, nullptr };
        if (current != nullptr)
        {
            current->next = newNode;
        }
        else
        {
            head = newNode;
        }
        current = newNode;
    }

    int evenCount = countEven(head);
```

```
    cout << "Количество четных элементов: " << evenCount << endl;
}
```

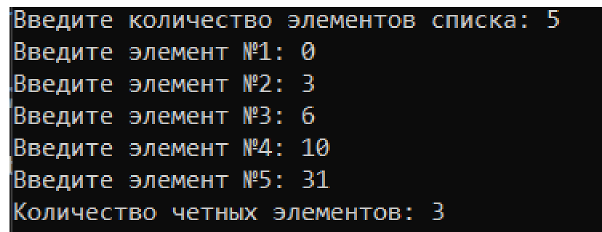
Введите количество элементов списка: 5
Введите элемент №1: 0
Введите элемент №2: 3
Введите элемент №3: 6
Введите элемент №4: 10
Введите элемент №5: 31
Количество четных элементов: 3

Рисунок 4 – Тестирование программы, вычисляющей количество четных элементов в списке

3 ВЫВОДЫ

Работа с рекурсивными алгоритмами и их реализация могут привести к следующим выводам:

Рекурсивные алгоритмы могут быть более простыми в понимании, чем итеративные алгоритмы. Они могут быть легче читаемыми и менее запутанными, поскольку они не требуют множества переменных состояния, но рекурсивные вызовы могут занимать много времени и памяти, особенно если они используются внутри цикла.

Рекурсивные алгоритмы часто используются для работы с древовидными структурами данных, такими как деревья поиска, бинарные деревья и графы. Они могут быть особенно полезными для обхода и поиска элементов в этих структурах.

Рекурсивные алгоритмы могут быть легко переполнены, если они вызывают сами себя слишком много раз. Это может привести к выходу из стека вызовов и ошибки программы.

4 СПИСОК ЛИТЕРАТУРЫ

1. Лекции по дисциплине «Структуры и алгоритмы обработки данных».

2. Habr: сайт. – URL: https://habr.com/ru/all/ (дата обращения: 26.03.2023).