МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического занятия 9

Тема: Рекурсивные алгоритмы и их реализация

Дисциплина: Структуры и алгоритмы обработки данных

Выполнил студент Хвостов В. В.

Группа ИКБО-01-20

Москва 2021

Содержание

Ответы на вопросы

Рекурсивная функция

Рекурсивная функция - числовая функция числового аргумента, которая в своём определении содержит себя же (математическое определение).

Рекурсивная функция -это функция, которая вызывает саму себя (в языках программирования).

Шаг рекурсии

Шаг рекурсии - это условие продолжения рекуррентного вызова.

Глубина рекурсии

Глубина рекурсии - это количество одновременно выполняемых процедур.

Условие завершения рекурсии

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

Линейная рекурсия

Линейная рекурсия - вид рекурсии, при которой рекурсивные вызовы на любом рекурсивном срезе, инициируют не более одного последующего рекурсивного вызова.

Каскадная рекурсия

Каскадная рекурсия - рекурсия, не являющаяся линейной. При каскадной рекурсии, рекурсивные обращения, как правило, приводят к необходимости многократно решать одни и те же подзадачи.

Прямая рекурсия

Прямая рекурсия вызывает алгоритм (функцию, процедуру) P из текста самого алгоритма P.

Косвенная рекурсия

Косвенная рекурсия - циклическая последовательность вызовов нескольких алгоритмов $P_1, P_2, \ldots, P_n$ (функций, процедур) друг друга: P_1 вызывает P_2, P_2 вызывает P_3, …, P_n вызывает P_1 ($n > 1$).

Организация стека рекурсивных вызовов

Стек рекурсии - область памяти, в которую заносятся значения всех локальных переменных алгоритма (программы) в момент рекурсивного обращения. Каждое такое обращение формирует один слой данных стека. При завершении вычислений по конкретному обращению a из стека считывается соответствующий ему слой данных, и локальные переменные восстанавливаются, снова принимая значения, которые они имели в момент обращения a.

Главное задание

Вариант 10

2.1. Задача 1

2.1.1. Условие задачи

Вычислить значение цифрового корня для некоторого целого числа N.

2.1.2. Постановка задачи

Реализовать функцию вычисления произвольного корня для некоторого целого числа N.

2.1.3. Описание алгоритма – рекуррентная зависимость

Для реализации данной функции воспользуемся методом Ньютона из математического анализа, который позволяет найти приблизительное значение функции в конкретной точке (в нашем случае $f(x) = x^{\frac{1}{n}}$, где $n \in \mathbf{N}$). Рекуррентным соотношение в данном случае является формула

$$x_{k+1} = \frac{1}{n}\left((n-1)x_k + \frac{N}{x_k^{n-1}}\right).$$

где x_k - значение x на k шаге рекурсии. Чем больше количество вызовов рекурсии, тем точнее наш результат. Но т.к. компьютер ограничен в точности, то мы зададим условие выхода из рекурсии такое, чтобы разница между x_n и x_{n+1} была меньше определенной константы h, определенной ниже.

2.1.4. Коды используемых функций

```
#include <cmath>
#include <iostream>

constexpr long double h = 0.00000001; // 10^-8

double newtonsMethodRoot(double N, double x, double n) {
  double xn = 1/n * ((n-1)*x+N/std::pow(x, n-1));
  if (std::fabs(xn - x) <= h) {
```

```
    return xn;
  } else {
    return newtonsMethodRoot(N, xn, n);
  }
}
```

2.1.5. Ответы на задания по задаче 1

Итерационный алгоритм решения задачи

Для решения данной задачи можно использовать алгоритм, который сравнивает разность текущего и предыдущего значения x с некоторой погрешностью в цикле и каждый раз считает новое значение корня n-ой степени, сохраняя предыдущее для следующей итерации.

Реализация в виде функции для итерационного алгоритма

```
double newtonsMethodRootIter (double N, double x, double n) {
  double xn = 0;
  double xprev = x;
  for (;;) {
    xn = 1/n * ((n-1)*xprev+N/std :: pow(xprev, n-1));
    if (std :: fabs(xprev - xn) <= h) {
      break;
    }
    xprev = xn;
  }
  return xn;
}
```

Теоретическая сложность алгоритма

Теоретическая сложность алгоритма равна $\Theta(d^p \lg d)$ т.к. каждый раз в цикле точность увеличивается примерное в p раз, а также $\Theta(d^p)$ - сложность выполнения возведения в степень(где p - степень корня, d - необходимая точность результата т.е. количество цифр после запятой).

Рекуррентная зависимость

Рекуррентная зависимость в данной задаче - значение корня k-ой степени, вычисленное на предыдущем шаге.

Реализация в виде функции для рекуррентного алгоритма

```
#include <cmath>
#include <iostream>

constexpr long double h = 0.00000001; // 10^-8

double newtonsMethodRoot(double N, double x, double n) {
  double xn = 1/n * ((n-1)*x+N/std::pow(x, n-1));
  if (std::fabs(xn - x) <= h) {
    return xn;
  } else {
    return newtonsMethodRoot(N, xn, n);
  }
}
```

Глубина рекурсии

Условием выхода из рекурсии является вычисление корня с необходимой точностью т.е. из теоретической сложности получаем, что глубина алгоритма = $\lg d$ (т.к. на каждом уровне происходит умножением со сложностью $\Theta(d^p)$).

Определение сложности рекурсивного алгоритма

Определим сложность рекурсивного алгоритма с помощью метода постановки и дерева рекурсии. Пусть у нас на вход подается какое-то число с количеством знаков в результате d. Построим для него дерево (рис. 1). Суммируя все

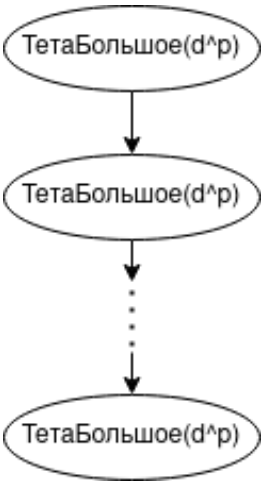


Рис. 1 - Дерево рекурсии для задания 1

узлы дерева, получаем сложность $\Theta(d^p \lg d)$ (т.к. глубина дерева = $\lg d$).

Пример схемы рекурсивных вызовов

Рис. 2 - Пример рекурсии задания 1

2.1.6. Код программы

```
int main() {
  std :: cout << "Enter number which will be calculated : ";
  int N;
  std :: cin >> N;
  std :: cout << "Enter power of root function : ";
  int power;
  std :: cin >> power;
  double res = newtonsMethodRoot(N, N, power);
  std :: cout << power << " power root = " << res
    << '\n';
  return 0;
}
```

2.1.7. Результаты тестирования

Таблица 1 - Тестирование задачи 1

Номер теста	Входные данные	Ожидаемый результат	Результат выполнения программы
1	Число: 24, Степень: 2	4.89898	2 power root = 4.89898
2	Число: 1324, Степень: 3	10.9807	3 power root = 10.9807
3	Число: 256, Порядок: 4	4	4 power root = 4

2.2. Задача 2

2.2.1. Условие задачи

Найти в двунаправленном списке количество четных элементов.

2.2.2. Постановка задачи

Реализовать функцию поиска количества четных элементов в двунаправленном списке.

2.2.3. Описание алгоритма – рекуррентная зависимость

Для решения задачи с помощи рекурсии каждый раз мы будем рассматривать элемент в списке, на который наш указатель хранит адрес, проверять является ли он четным, и затем вызывать рекурсивно эту же функцию, но для следующего элемента. Алгоритм заканчивает свою работу при достижении конца списка.

2.2.4. Коды используемых функций

```
#include <iostream>
#include <list>
#include <random>

constexpr int n = 10;

template<typename BidirIter>
void countEvenElementsListRec(BidirIter begin, BidirIter end, int& result) {
  if (begin != end) {
    if (*begin %2 == 0) {
      ++result;
    }
    return countEvenElementsListRec(++begin, end, result);
  }
}
```

2.2.5. Ответы на вопросы задания 2

Глубина рекурсии

Глубина рекурсии равна n т.е. размеру списка т.к. нам требуется пройти по всему списку для подсчета количества четных элементов.

Теоретическая сложность алгоритма

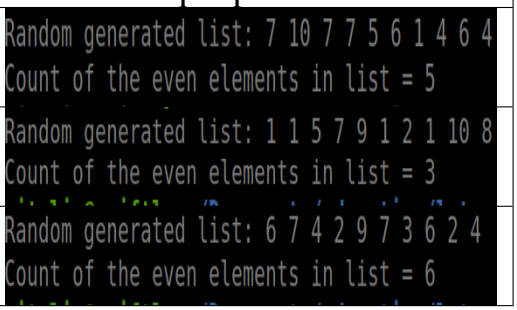
Теоретическая сложность алгоритма равна $\Theta(n)$ т.к. мы полностью проходим список 1 раз.

2.2.6. Код программы

```
template<typename Number>
void generateRandomList(std::list<Number>& list) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::    uniform_int_distribution    <Number> dist(1, n);
    for (int i = 0; i < n; ++i) {
        list.push_back(dist(gen));
    }

}

template<typename T>
std::ostream& operator<<(std::ostream& stream, const std::list<T>& list) {
    for (const auto& elem: list) {
        stream << elem << " ";
    }
    return stream;
}

int main() {
    int x = 0;
    std::list<int> list;
    generateRandomList(list);
    std::cout << "Random generated list: " << list << '\n';
    countEvenElementsListRec(list.cbegin(), list.cend(), x);
    std::cout << "Count of the even elements in list = " << x << '\n'; // the list, I know
    return 0;
}
```

2.2.7. Результаты тестирования

Таблица 2 - Тестирование задачи 2

Номер теста	Входные данные	Ожидаемый результат	Результат выполнения программы
1	Список: 7 10 7 7 5 6 1 4 6 4	5	
2	Список: 1 1 5 7 9 1 2 1 10 8	3	
3	Список: 6 7 4 2 9 7 3 6 2 4	6	

Выводы

В ходе выполнения работы были получены практические навыки по разработке рекурсивных алгоритмов. В соответствии с персональным вариантом, были созданы и проанализированы следующие рекурсивные (и итеративные) алгоритмы: функция подсчета корня n-ой степени, подсчет количества четных чисел в двунаправленном списке. Каждая рекурсивная функция прошла тестирование успешно, что подтверждает правильную работу алгоритмов.

Список информационных источников

1. Thomas H. Cormen, Clifford Stein и другие: Introduction to Algorithms, 3rd Edition. Сентябрь 2009. The MIT Press.

2. Олег Смирнов: Анализ рекуррентных соотношений . Октябрь 2011. https://nord.org.ua/static/course/algo_2011/lecture2.pdf.

3. Nth root // Wikipedia [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Nth_root (Дата обращения: 07.05.2021)

4. Курс Algorithms, part 2 // Coursera [Электронный ресурс]. URL: https://www.coursera.org/learn/algorithms-part2 (Дата обращения: 07.05.2021)