Урок 5

Рекурсия

Зачем функция вызывает саму себя?

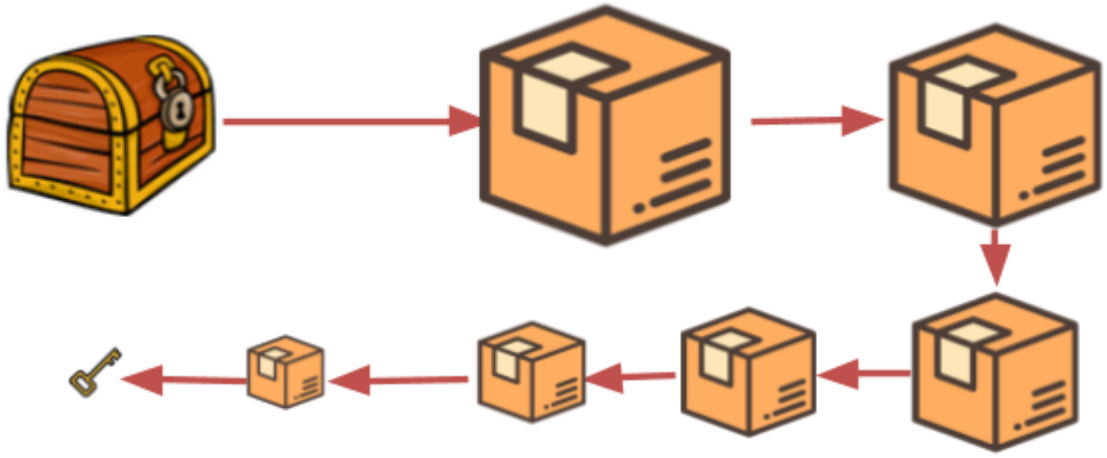
Введение

В программировании рекурсия — это понятие, которое тесно связано с методами. Точнее, это методология программирования, согласно которой метод вызывает сам себя. Начинающего программиста рекурсия может ввести в ступор. Но она применяется для решения многих задач в программировании.

На этом уроке рассмотрим примеры применения рекурсии: вычисления треугольных чисел и факториалов, а еще несколько интересных задач из книги Лафорте «Структуры данных и алгоритмы в Java».

Понятие рекурсии

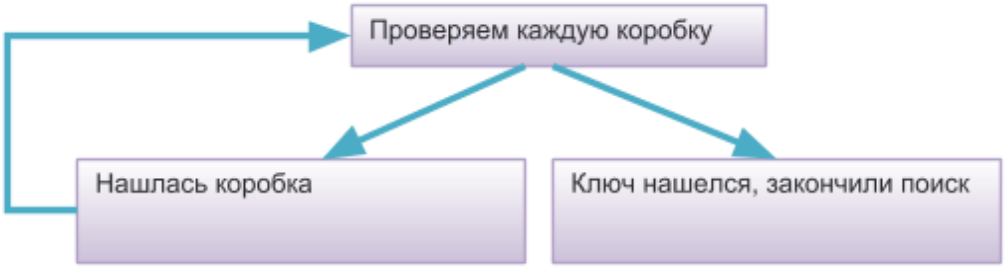
В книге “Grokking Algorithms” приводится удачное объяснение рекурсии. Представим, что есть закрытый на замок чемодан, а ключ от него лежит в коробке.



А в большой коробке лежит еще много маленьких. Как будем искать в них ключ? Знакомый способ — перебор с использованием цикла с предусловием.

Все коробки собираются в кучу. Открываем первую коробку. Если в ней ключ, заканчиваем поиск. Если в ней коробка — добавляем ее в кучу. Повторяем, пока не просмотрим все коробки. Такой способ для решения задачи использует цикл **while**.

Еще один способ решения этой задачи — замена цикла на рекурсивную функцию. Проверяем каждый предмет в коробке. Если находится коробка — возвращаемся к первому шагу и проверяем коробку. Если нашелся ключ — поиск закончен.



Применение рекурсии не ускоряет работу программы, а в некоторых случаях использовать циклы бывает намного эффективнее. Рекурсия используется во многих алгоритмах — например, в деревьях, — о которых мы будем говорить через несколько уроков.

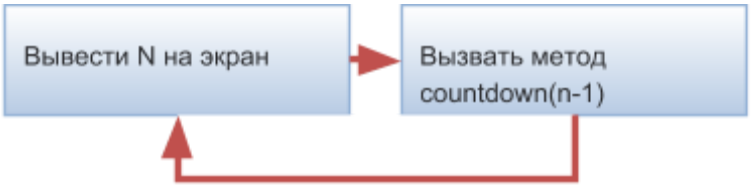
Прежде чем разбирать конкретные задачи, которые решаются при помощи рекурсии, усвоим теоретические сведения о ней.

Базовый и рекурсивный случай

В рекурсии метод вызывает сам себя. В таком коде программист может запутаться и написать программу, которая будет работать бесконечно. Например, такую, которая ведет обратный отсчет. На экран по очереди выводятся цифры 5, 4, 3, 2, 1. Попробуем написать такой метод с помощью рекурсии.

```
public static int countdown(int n){
	System.out.println(n);
	return countdown(n-1);

}
```

Метод **countdown** выводит на экран входящий аргумент **n**, после чего вызывает сам себя с аргументом **n-1**. И это будет происходить бесконечно, так как в методе отсутствуют механизмы завершения рекурсивного случая.



Когда программист пишет метод, который содержит рекурсию, он должен позаботиться о выходе из нее.

Рекурсивный случай — это вызов методом самого себя, а базовый — когда метод себя не вызывает. Доработаем наш рекурсивный метод, добавив в него выход из рекурсии.

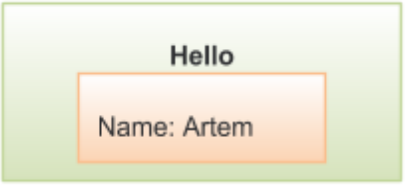
```
public static int countdown(int n){
	System.out.println(n);
	if (n == 1){
		return 1;
	}
	return countdown(n-1);

}
```

Стек вызовов

Рассмотрим, как работает рекурсия внутри компьютера. Для этого используется стек вызовов. Помним, что стек — это структура данных, в которой элемент зашел последним, а вышел первым. Так же построен и стек вызовов. Представим, что у нас есть метод **hello**, в который параметром передается аргумент с именем человека, которого следует поприветствовать. Внутри этого метода вызывается метод **print**, выводящий приветствие на экран, и метод **bye**, который прощается с человеком.

```java
public class HelloBye {

    public static void main(String[] args) {
        hello("Artem");
    }

    public static void hello(String name){
        System.out.println("Hello, "+name+"!");
        bye(name);
    }

    public static void bye(String name){
        System.out.println("Good bye, "+name+"!");
    }


}
```

Первым в программе вызывается метод **hello** и записывается в стек вызова.



Далее вызывается метод **print** и тоже записывается в стек вызова.

Теперь выводится строка с надписью “**Hello, Artem”**, и метод **print** удаляется из стека. На его место помещается метод **bye**, который вызывается следом за **println**.

В методе **bye** вызывается метод **println** и помещается в стек вызовов. Далее по очереди выводится сообщение на экран, завершаются методы **bye** и **hello**.

Стек вызовов с рекурсией

При работе с рекурсивными методами также используется стек вызовов. Рассмотрим пример расчета факториала числа. Факториал — это произведение всех чисел, входящих в искомое число. Например, факториал числа 4! = 1*2*3*4 = 24.

Напишем программу, которая будет выводить на экран факториал числа **N**.

```
public class Factorial {

    public static void main(String[] args) {
        System.out.println(factorial(3));
    }

    public static int factorial(int n){
        if (n==1)
            return 1;
        return n*factorial(n-1);
    }


}
```

В программе реализован единственный метод **factorial**, который рассчитывает факториал числа 3. В нем есть базовый случай, который описан в условии при **n=1**, и рекурсивный, когда метод вызывает сам себя. Посмотрим, как рекурсивный метод попадает в стек вызова. Попробуем сделать это пошагово.

Шаг 1. В стек попадает функция с аргументом n = 3.

Шаг 2. Проверяется условие n == 1. Так как n равна 3, то это рекурсивный случай.

Шаг 3. В стек попадает функция с аргументом n = n - 1, т.е. 2.

Шаг 4. Проверяется условие n == 1. Так как n равна 2, то это рекурсивный случай.

Шаг 5. В стек попадает функция с аргументом n=n-1, т.е. 1.

Шаг 5. Проверяется условие n == 1. Так как n равна 1, то это базовый случай. Теперь рекурсия разворачивается обратно, и возвращается 1.

Шаг 6. factorial(1) возвращает 1, n = 1*1.

Шаг 7. Рекурсия разворачивается дальше, и функция **factorial(2)** возвращает 2, n = 1* 2.

Шаг 8. Из стека выбирается последняя функция, которая была в нем: **factorial(3)**, n=2*3.

Шаг 9. Функция **factorial(3)** возвращает значение 6.

Получается, что методы сначала помещаются в стек вызова, а когда рекурсивный вызов заканчивается, они начинают выполняться с конца.

Переполнение стека вызовов

Попробуйте через рекурсивный метод написать обратный отсчет большого числа. Программа завершится с ошибкой **Exception in thread "main" java.lang.StackOverflowError**. Причина в том, что сначала рекурсивные методы попадают в стек вызовов, и только когда наступает базовый случай, начинается их выполнение. Стек вызовов тоже имеет свой размер и может переполниться. Поэтому необходимо следить за количеством методов, которые могут попасть в стек, или увеличивать его размер. Сделать это можно в конфигурации проекта или через командную строку.

Рассмотрим примеры использования рекурсии.

Анаграммы

Анаграмма — это слово, составленное путем перестановки букв в другом:

> «*Но и в РЕАЛИЗМЕ при желании*
>
> *обнаружат сговор с ИЗРАИЛЕМ*».

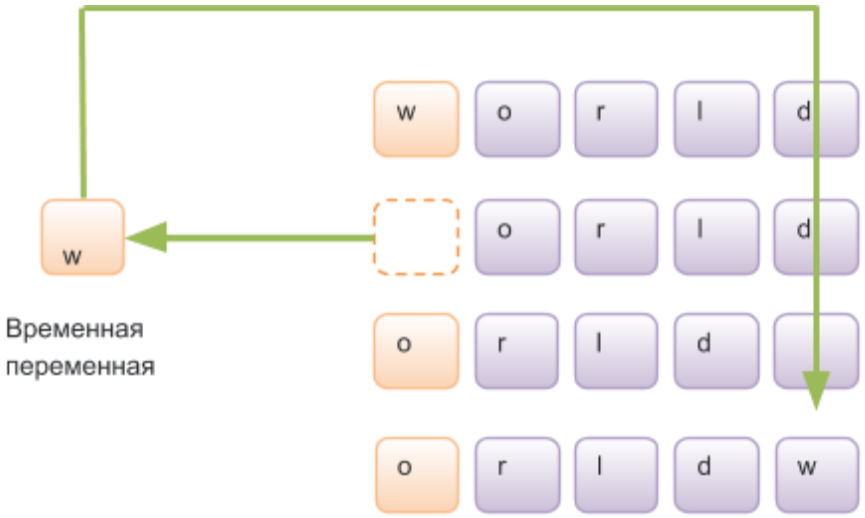
Часто анаграммы используют для создания псевдонимов.

В программировании бывают задачи на перечисление всех анаграмм заданного слова. Например, для слова **cat** программа выводит:

- cat;
- cta;
- atc;
- act;
- tca;
- tac.

Чем больше букв в слове, тем многочисленнее возможные варианты анаграмм. Если быть точнее, их количество равно факториалу числа букв в слове, если они не повторяются. Слово cat состоит из трех букв, факториал **3! = 6**.

Рассмотрим алгоритм построения списка анаграмм для слова из **N** букв. Построим анаграммы для **n-1** правых букв, выполним циклический сдвиг всех **n** букв и повторим эти действия **n** раз.

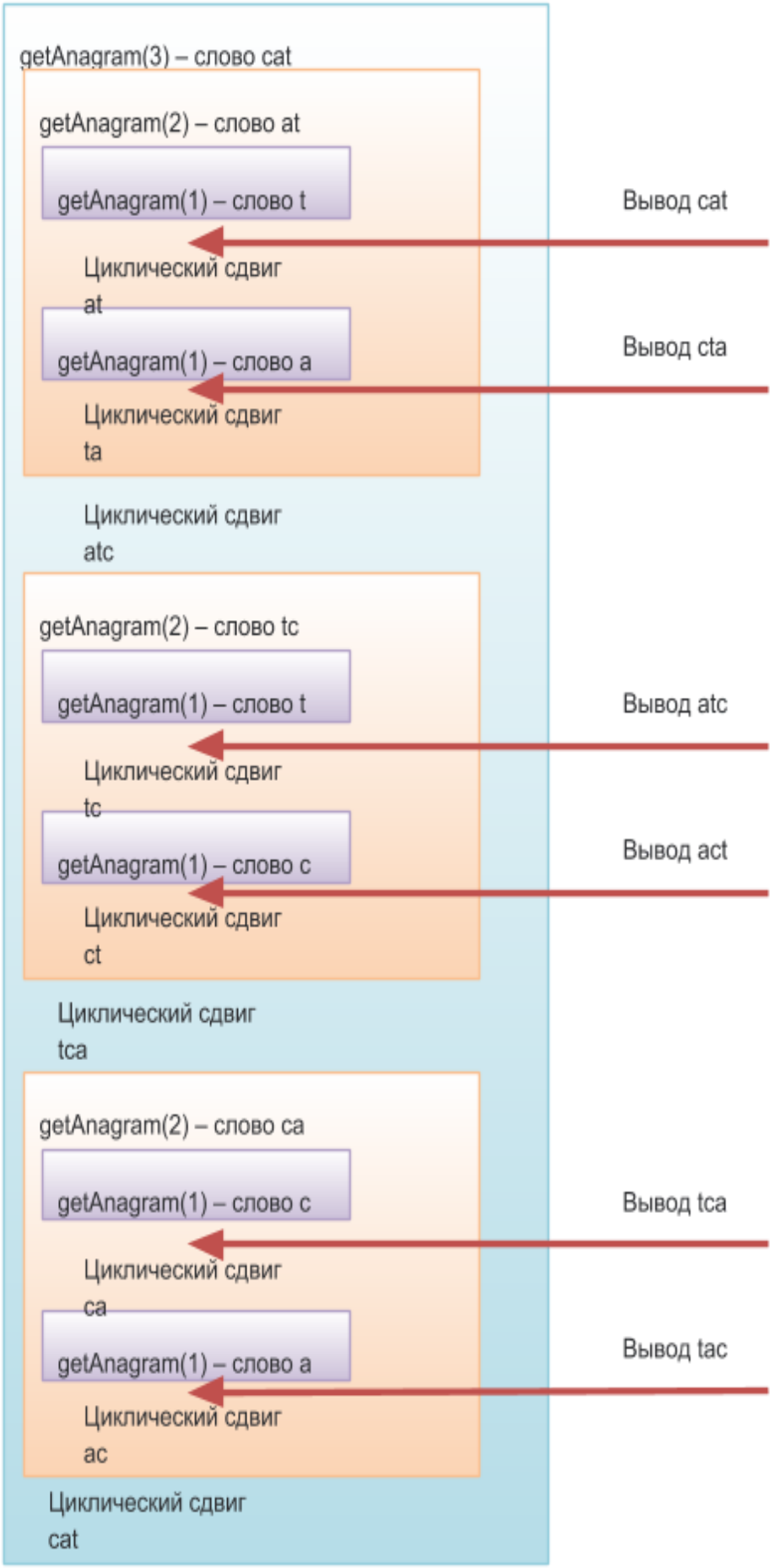


При циклическом сдвиге **n** раз каждая из букв побывает на первом месте. Пока буква стоит там, все остальные переставляются во всех возможных сочетаниях. Список анаграмм правых **n-1** букв строится через рекурсивный вызов. Рекурсивный метод **getAnagramm** получает единственный параметр с размером слова, в котором требуется выполнить перестановки букв. При каждом рекурсивном вызове метода **getAnagramm** количество букв в слове уменьшается на 1. Базовый случай наступает, когда количество букв становится равным единице.

Метод **getAnagramm**:

```
public static void getAnagramm(int newSize){
      if (newSize == 1)
            return;
      for (int i=0;i<newSize;i++){
            getAnagramm(newSize-1);
            if (newSize == 2)
                  display();
            rotate(newSize);
      }
}
```

Для наглядности пройдемся по рекурсивному вызову метода **getAnagram** для слова **cat**.


getAnagram(3) – слово cat
getAnagram(2) – слово at
getAnagram(1) – слово t
Вывод cat
Циклический сдвиг
at
getAnagram(1) – слово a
Вывод cta
Циклический сдвиг
ta
Циклический сдвиг
atc
getAnagram(2) – слово tc
getAnagram(1) – слово t
Вывод atc
Циклический сдвиг
tc
getAnagram(1) – слово c
Вывод act
Циклический сдвиг
ct
Циклический сдвиг
tca
getAnagram(2) – слово ca
getAnagram(1) – слово c
Вывод tca
Циклический сдвиг
ca
getAnagram(1) – слово a
Вывод tac
Циклический сдвиг
ac
Циклический сдвиг
cat

Листинг программы по построению анаграмм:

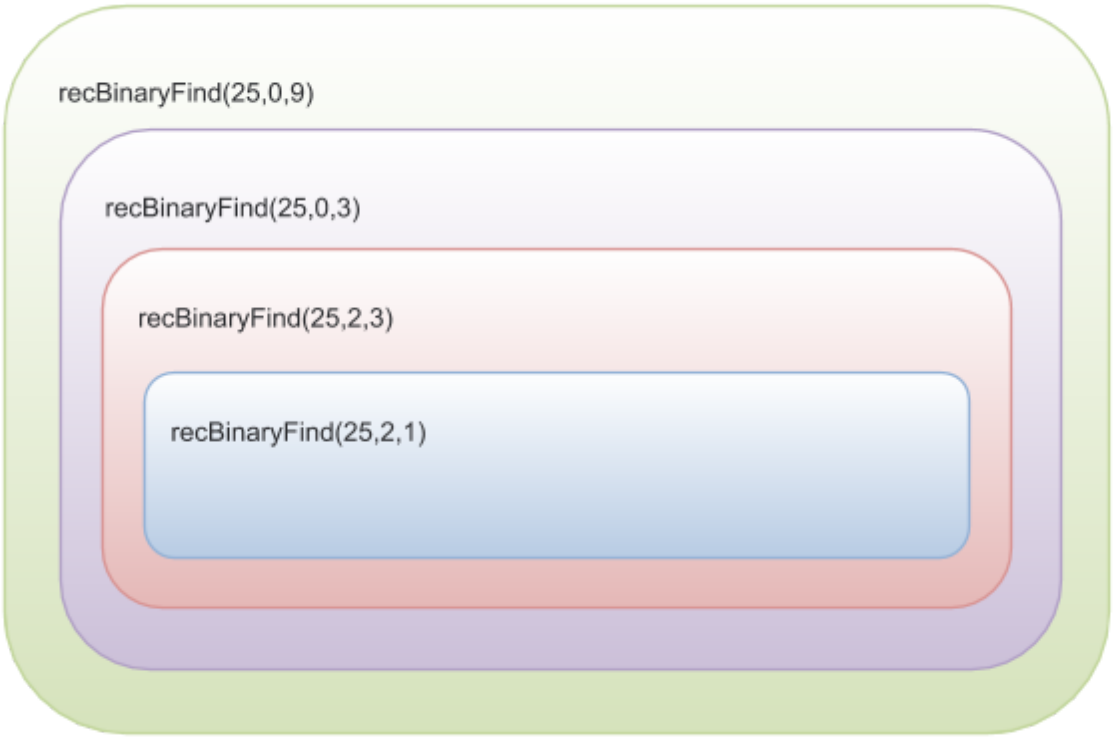
```
public class AnagrammApp {
    static int size;
    static int count;
    static char[] arr = new char[3];

    public static void main(String[] args) throws IOException{
        String input = getString();
        size = input.length();
        count = 0;
        for(int i=0;i<size;i++){
            arr[i] = input.charAt(i);
        }
        getAnagramm(size);
    }

    public static void getAnagramm(int newSize){
        if (newSize == 1)
            return;
        for (int i=0;i<newSize;i++){
            getAnagramm(newSize-1);
            if (newSize == 2)
                display();
            rotate(newSize);
        }
    }

    public static void rotate(int newSize){
        int i;
        int pos = size - newSize;
        char temp = arr[pos];
        for (i=pos+1;i<size;i++){
            arr[i-1] = arr[i];
        }
        arr[i-1] = temp;
    }

    public static void display(){


        for(int i=0; i<size; i++){
            System.out.print(arr[i]);
        }
        System.out.println("");

    }

    public static String getString() throws IOException{
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        return br.readLine();

    }

}
```

Метод **rotate** осуществляет циклический сдвиг на одну позицию влево для каждой буквы слова. Метод **display** выводит на экран полученную анаграмму.

Рекурсивный двоичный поиск

Во втором уроке мы рассматривали бинарный поиск в отсортированном массиве. Изменим метод **binaryFind**. Чтобы не менять пользовательский класс **MyArrApp**, который вызывает метод **binaryFind**, создадим метод **recBinaryFind** и будем вызывать его из **binaryFind**.

Допустим, наш отсортированный массив содержит 10 элементов [-10, 20, 25, 26, 40, 45, 75, 80, 82, 91]. Ищем элемент, равный 25. Он находится в третьей позиции.



```
class MyArr{
    private int[] arr;
    private int size;

    public MyArr(int size){
        this.size = 0;
        this.arr = new int[size];
    }

    public int binaryFind(int search){
         return recBinaryFind(search, 0, size-1);
    }


    private int recBinaryFind(int searchKey, int low, int high){
        int curIn;
        curIn = (low + high ) / 2;
        if(arr[curIn]==searchKey)
            return curIn;
        else
            if(low > high)
                return size;
            else{
                if(arr[curIn] < searchKey)
                    return recBinaryFind(searchKey, curIn+1, high);
                else
                    return recBinaryFind(searchKey, low, curIn-1);
            }
}


    public void insert(int value){
      int i;
      for(i=0;i<this.size;i++){
            if (this.arr[i]>value)
                  break;
      }
      for(int j=this.size;j>i;j--){
            this.arr[j] = this.arr[j-1];
      }
      this.arr[i] = value;
      this.size++;
    }

}

public class MyArrApp {
    public static void main(String[] args) {

        MyArr arr = new MyArr(10);
        arr.insert(-10);
        arr.insert(45);
        arr.insert(26);
        arr.insert(20);
        arr.insert(25);
        arr.insert(40);
        arr.insert(75);
        arr.insert(80);
```

```
        arr.insert(82);
        arr.insert(91);

        int search = -10;


        System.out.println(arr.binaryFind(search));
    }

}
```

Домашнее задание

1. Написать программу по возведению числа в степень с помощью рекурсии.
2. Написать программу «Задача о рюкзаке» с помощью рекурсии.

Дополнительная литература

1. Grokking Algorithms: An Illustrated Guide for Programmers and Other. Aditya Y. Bhargava. Recursion.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Лафорте Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. — СПб.: Питер, 2013. — 121-178 сс.