Модуль 4. Алгоритмы и структуры данных

Тема 4.5. Рекурсия

2 часа

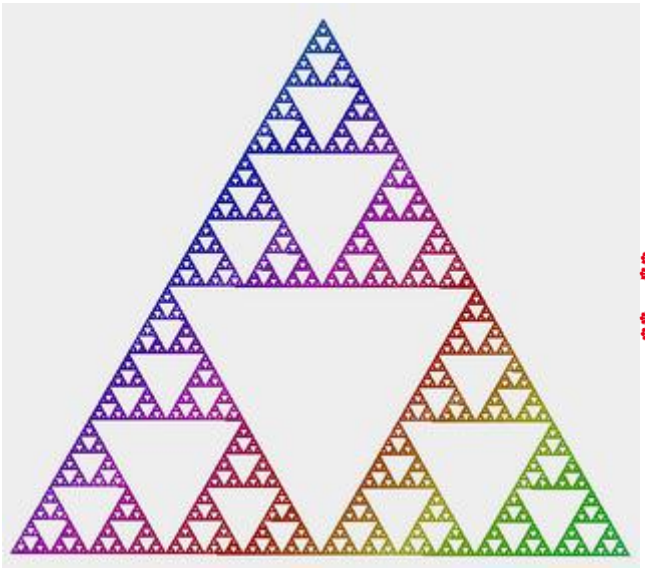
Оглавление

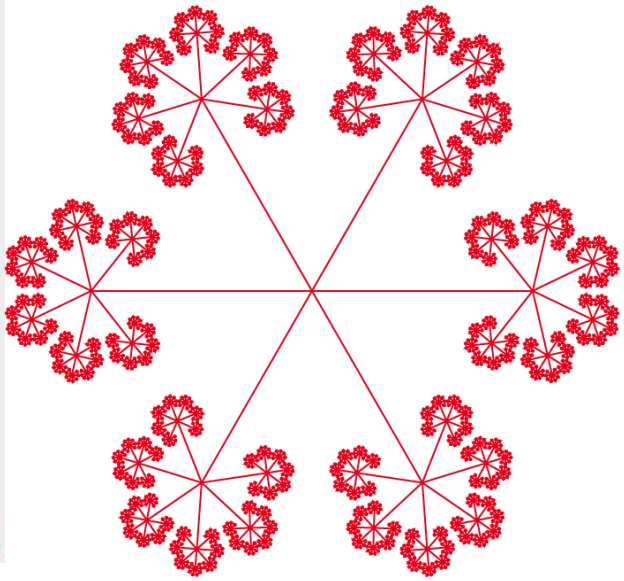
4.5. Рекурсия

4.5.1. Рекурсия в программировании и не только

Реку́рсия — определение, описание, изображение какого-либо объекта или процесса внутри самого этого объекта или процесса, то есть ситуация, когда объект является частью самого себя. Термин «рекурсия» используется в различных специальных областях знаний — от лингвистики до логики, но наиболее широкое применение находит в математике и информатике [1].

Примеры рекурсии в изображениях:





Возьмите в руки зеркало и встаньте перед другим зеркалом - вы увидите бесконечное рекурсивное повторение себя.

Известное стихотворение Маршака “Дом, который построил Джек” также использует этот прием.

Классический пример из математики: рекурсивно-определённый факториал целого неотрицательного числа:

$$n! = \begin{cases} n \cdot (n-1)!, & n > 0 \\ 1, & n = 0 \end{cases}$$

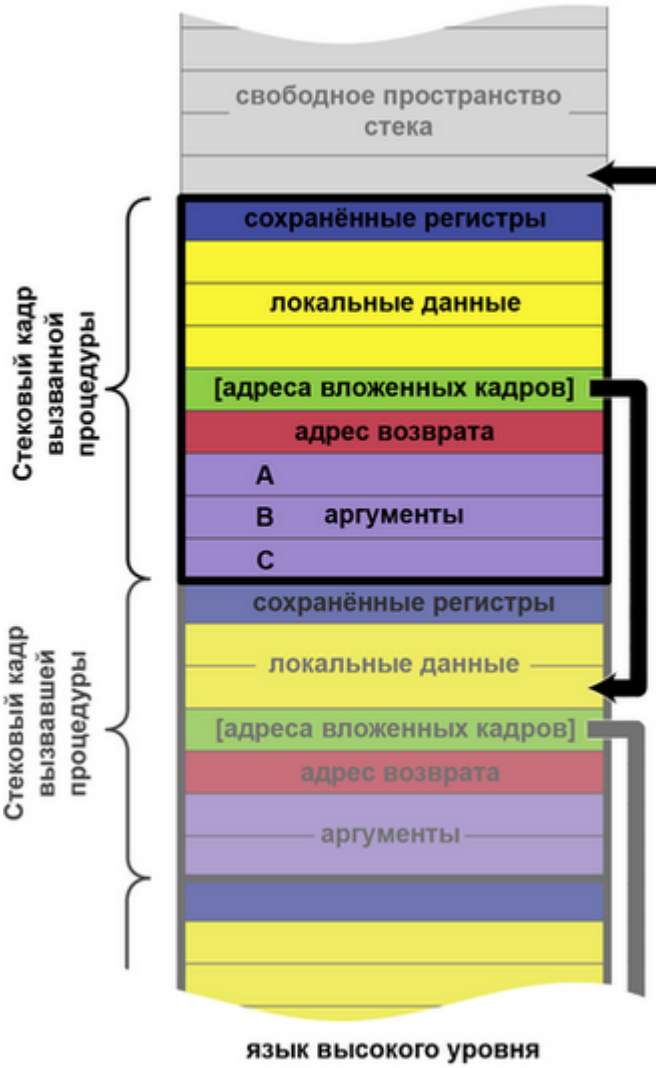
И наконец, в программировании есть рекурсивные функции (методы). Рекурсивные методы внутри своего тела вызывают сами себя.

Рекурсия не очень широко применяется в программировании, хотя любой циклический алгоритм можно реализовать с помощью рекурсии. При этом обратное правило не работает: не каждый рекурсивный алгоритм можно реализовать с помощью цикла. Например, при обработке структур типа “дерево” без рекурсии не обойтись. Рекурсия имеет ряд недостатков, о которых мы расскажем позже, и поэтому, если есть выбор, то грамотный программист выберет цикл.

Поклонники рекурсии любят ее за выразительность и лаконичность.

4.5.2. Стек вызовов

Рекурсию невозможно освоить без понимания такого важного понятия, как стек вызовов. В одном потоке в текущий момент времени может исполняться только один единственный метод из всей программы. Какой метод запускать понятно по ходу выполнения программы, но как потом вернуться назад в точку программы после выполненного метода? Для этого используют стек вызовов.



Мы уже изучили структуру данных “стек” и помним, что он работает на основе списка с дисциплиной обслуживания LIFO “последним пришел-первым ушел” (вспомните колбочку и шайбы!).

Что в нем хранится? **Стек вызовов** хранит информацию для возврата управления из методов в метод его вызвавший.

Пусть метод А вызывает метод Б. При вызове метода Б в стек заносится адрес точки возврата — адрес в памяти инструкции в методе А, которая должна выполниться после выполнения метода Б. Если метод Б вызовет еще один метод С, то в стек вызовов будет занесен очередной адрес возврата - инструкция из метода Б и т.д.

При возврате из метода С адрес возврата снимается со стека и управление передается на следующую инструкцию в приостановленном методе Б. По завершении метода Б из стека будет снят адрес возврата в метод А и т.д.

В реальности для языков высокого уровня, такого, как Java в стеке хранится не только адрес вовзрата, но и другая необходимая информация, например аргументы (параметры) и локальные переменные метода (см. рисунок) [2].

Становится понятным, что каждый вызов метода занимает объем памяти в стеке и тем самым уменьшает доступную оперативную память.

4.5.3. Линейная рекурсия

Разберем шаги построения рекурсивных программ на примере вычисления целой степени числа. Сразу заметим, что мы выбрали этот пример для простоты иллюстрации, но в реальности таким образом решать эту задачу не рекомендуется.

Упражнение 4.5.1.

Вычисление целой степени n числа x: n-я степень числа X получается путем домножения самого X на (n-1)-ю степень этого числа. Нулевая степень числа равна 1, число в 1-й степени равно самому числу (это и есть условия завершения рекурсии).

$$X^n = \begin{cases} 1, n = 0 \\ X * X^{n-1}, n > 0 \\ \frac{1}{X^{|n|}}, n < 0 \end{cases}$$

Т.е. мы описали факты:

- При возведении числа в степень 0 результат = 1.
- n-я положительная степень числа X получается путем домножения самого X на (n-1)-ю степень этого числа.
- n-я отрицательная степень числа X получается путем деления 1 на X умноженное на (n-1)-ю степень этого числа.

Отсюда получаем программу:

```
public class MyMath {

        /* Рекурсия: вычисление целой степени числа */
        static double nPow(double x, int n)

        {
                double y;
                if (n == 0) /* условие завершения рекурсии */
                        return 1;
                if (n < 0)
                        y = 1. / nPow(x, -n);
                else
                        y = x * nPow(x, n - 1);
                return y;
        }

        public static void main(String[] args) {
                double x = 2.;  //основание
                int n = 3;      //степень
                System.out.println(x + " ^ " + n + " = " + nPow(x, n));
        }

}
```

Зная, как работает стек вызовов, мы можем представить работу рекурсивных методов. Здесь нужно четко понимать, что каждый рекурсивный вызов - это не вызов одного и того же метода. Для компьютера это разные методы со своими аргументами и локальными переменными.

Ниже на рисунке приведена иллюстрация работы нашего метода на примере вычисления 3-ей степени числа 2. Черные стрелки – рекурсивные вызовы, зеленые –обратный ход рекурсии и возвращаемые значения. В данном примере произошло 3 рекурсивных вызова. Иллюстрация наглядно показывает, что цепочка вызовов линейная, такие алгоритмы называют **линейной рекурсией**.

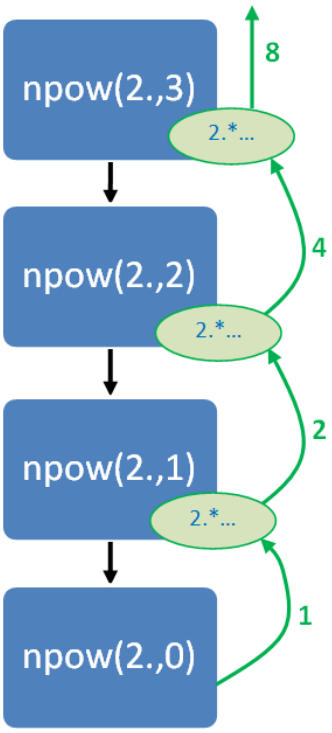


Рисунок 1. Иллюстрация рекурсивных вызовов для nPow(2.,3)

Мы вычисляем степень, спускаясь от n вниз.

Условие n==0 - это признак завершения (“заглушка”) рекурсии. Начиная с этого значения цепочка рекурсивных вызовов будет свертываться (а стек освобождаться) в обратном направлении. В любом рекурсивном алгоритме должно присутствовать такое условие - **условие завершения рекурсии**, в обратном случае алгоритм будет бесконечным.

Каждый раз при рекурсивном вызове метода nPow стек пополняется. Очевидно что при определенной глубине вызовов стек переполнится и возникнет ошибка. К тому же на вызов очередного метода требуется гораздо больше времени чем на шаг цикла. В этом и состоят основные **недостатки** рекурсивных алгоритмов.

Поэксперементируйте с программой:

- при каком значении n время выполнения программы значительно замедлиться (больше 1 мин)?
- реализуйте возведение в степень через цикл, при каком значении n время выполнения программы значительно замедлиться (больше 1 мин)?

4.5.4. Ветвящаяся рекурсия

Как уже понятно из названия раздела кроме линейной рекурсии выделяют ветвящуюся. Такая рекурсия возникает тогда, когда из метода происходит более чем 1 рекурсивный вызов. Разберем ее на примере вычисления числа Фибоначчи.

Упражнение 4.5.2.

Вычисление n-го числа Фибоначчи. Числа Фибона́ччи - элементы числовой последовательности, в которой каждое последующее число равно сумме двух предыдущих чисел:

Число Фибоначчи	1	1	2	3	5	8	13	21	34	55	...
n	0	1	2	3	4	5	6	7	8	9	...

Какие тут можно выделить факты?

- n-е число Фибоначчи - это сумма двух предыдущих чисел Фибоначчи (n-1) и (n-2)
- выделяются из общего правила 2 первых числа Фибоначчи: нулевое и первое числа равны 1 (это и будет условием выхода из рекурсии)

```
public class MyMath {

	/* Рекурсия: вычисление чисел Фибоначи */
	static long nFib(int n) {
		if (n == 0 || n == 1)
			return 1;// Условие завершения рекурсии – «заглушка»
		return nFib(n - 1) + nFib(n - 2);//возвращаем сумму предыдущих чисел
	}

	public static void main(String[] args) {
		// вывод первых 7 чисел Фибоначчи
		for (int i = 0; i < 7; i++)
			System.out.println("Fibonachi(" + i + ")=" + nFib(i));
	}

}
```

На рисунке приведена иллюстрация работы рекурсивной функции на примере вычисления пятого числа Фибоначчи nFib(5). Черные стрелки – рекурсивные вызовы, зеленые – обратный ход рекурсии и возвращаемые значения. В данном примере произошло 14 рекурсивных вызовов функции nFib(). Посмотрите, схема выглядит как дерево!

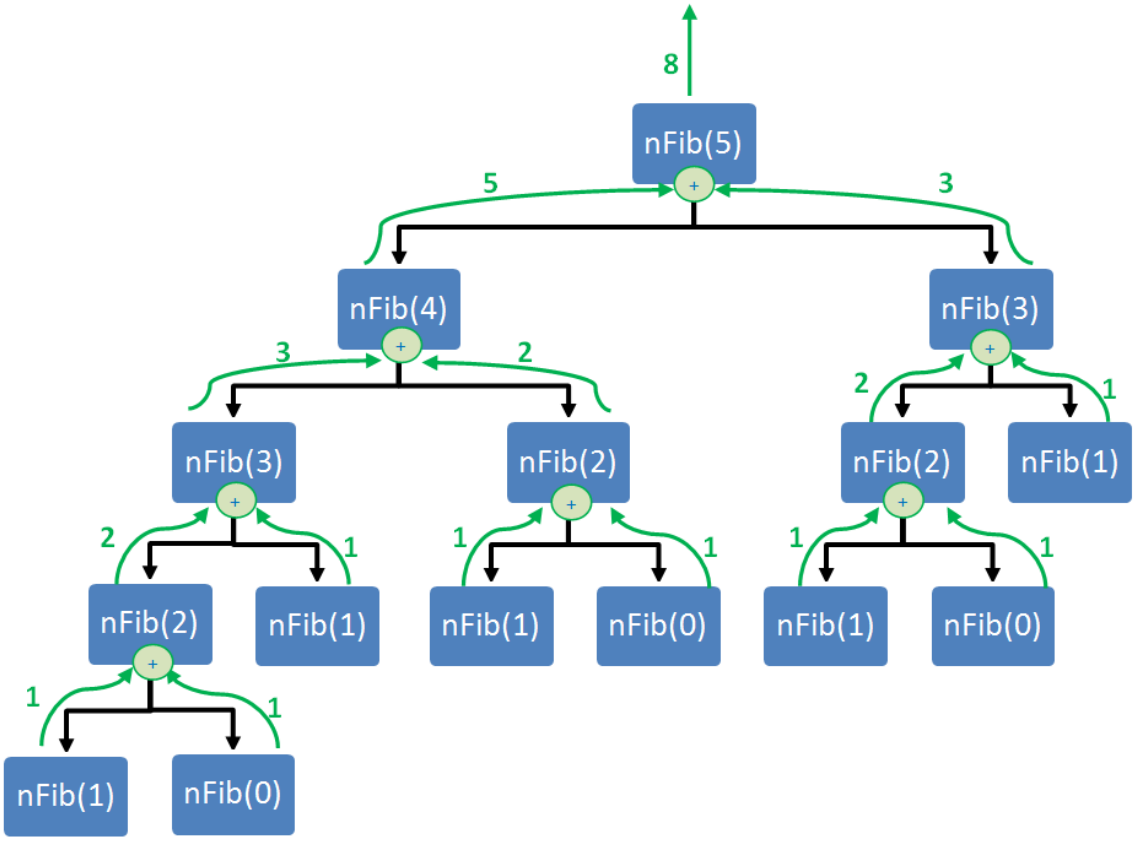


Рисунок 2. Иллюстрация рекурсивных вызовов для nFib(5)

Задание 4.5.1.

Внимательно изучите дерево рекурсивных вызовов на рисунке 2. Задумайтесь, чем плох этот алгоритм? Очевидны повторения: целые ветки совпадают в различных частях общего дерева! А это означает, что мы вычисляли уже известные числа Фибоначчи.
Напишите метод, который сохранял бы уже вычисленные значения Фибоначчи (например, в ArrayList<Long>) и использовал их для дальнейших вычислений.

Упражнение 4.5.3.

Запустите приведенную Android программу, реализующую вывод дерева каталогов, начиная с заданной и вывод на экран списка обнаруженных папок.

Определим разметку:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.samsung.itschool.recursionexample.MainActivity">
    <TextView
        android:id="@+id/tvLog"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</ScrollView>
```

И код с комментариями:

```
package com.samsung.itschool.recursionexample;

import android.os.AsyncTask;
import android.os.Bundle;
import android.os.Environment;
import android.support.v7.app.AppCompatActivity;
import android.widget.TextView;

import java.io.File;

public class MainActivity extends AppCompatActivity {

    //строка для хранения списка директорий
    private StringBuilder strTree = new StringBuilder();

    //поле для вывода реузультатов работы программы на экран
    private TextView logTview;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        logTview = (TextView) findViewById(R.id.tvLog);
        //Создаем AsyncTask для работы с рекурсией
        assyncRecursion task = new assyncRecursion();
        //Запускаем AsyncTask
        task.execute();
    }

    //Класс для выполнения тяжелых задач в отдельном потоке и передача в UI-
поток результатов работы.
    private class assyncRecursion extends AsyncTask<Void, Void, Void> {

        /*Рекурсивный метод формирует текст со списком директорий
        String path - путь к директории
        String indent - строка, иллюстрирующая глубину вложения директории
(длина ветки дерева директорий),
        за каждый уровень добавлем --
        */
        private void recursiveCall(String path, String indent) {
            //получаем список фалов и директорий внутри текущей директории
            File[] fileList = new File(path).listFiles();
            for (File file : fileList) {
                if (file.isDirectory()) { //для каждой директории из списка
                    //сохраняем глубину вложения и имя директории в строку
strTree

strTree.append(indent).append(file.getName()).append("\n");
                    //рекурсивный вызов для каждой директории из списка
                    recursiveCall(file.getAbsolutePath(), indent + "-- ");
                }
            }
        }

        //Метод выполняется в отдельном потоке, нет доступа к UI
        @Override
        protected Void doInBackground(Void... params) {
            //вызов метода с корневой папкой внешней памяти, глубина 0

recursiveCall(Environment.getExternalStorageDirectory().getPath(), "");
            return null;
        }
```

```
        //Метод выполняется после doInBackground, есть доступ к UI
        @Override
        protected void onPostExecute(Void result) {
            super.onPostExecute(result);
            //отображаем сформированный текст со списком директорий
            logTview.setText(strTree);
        }

    }
}
```

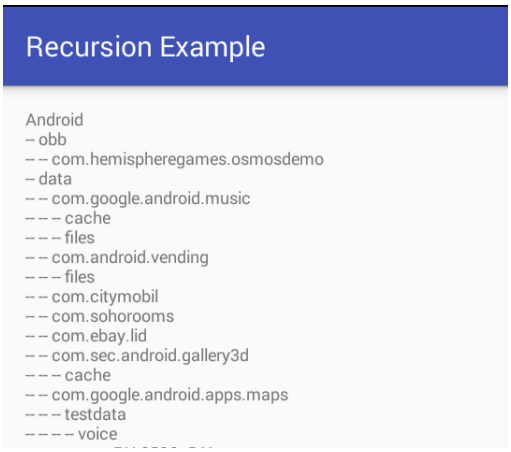
Заметьте, что используем AsyncTask для отделения тяжелой рекурсивной программы в отдельный поток.

При первом вызове рекурсивного метода передается путь до внешней памяти устройства с помощью библиотечного метода Environment.getExternalStorageDirectory().getPath().

В Android устройстве выделяют следующие виды памяти:

1. *Внутренняя (internal) память — это часть встроенной в телефон карты памяти. При ее использовании по умолчанию папка приложения защищена от доступа других приложений (Using the Internal Storage).*
2. *Внешняя (external) память — это общее «внешнее хранилище» для ваших файлов, которое может быть как на встроенной, так и SD карте. Но, как правило, в современных версиях Android - это часть встроенной памяти. (Using the External Storage).*
3. *Удаляемая (removable) память — все хранилища, которые могут быть удалены из устройства пользователем, например SD карта.*

Результат работы программы:



Задумайтесь, можно ли реализовать ту же самую функциональность без помощи рекурсии? Какой вид рекурсии в данной программе?

Задание 4.5.2

Для приведенных примеров: определить вид рекурсии (линейная или ветвящаяся), нарисовать иллюстрацию рекурсивных вызовов, подсчитать количество рекурсивных вызовов.

1.

```
// Вычисление факториала
static int fact(int n) {
         if (n == 1)
              return 1;
         return n * fact(n - 1);
}
public static void main(String[] args) {
         int a= fact(6);
}
```

2.

```
/* Перевод десятичного числа в двоичное */
static void DecToBin(int num) {
        if (num < 0) {
                num *= -1;
                System.out.print("-");
        }
        if (num / 2 > 0)
                DecToBin(num / 2);
        System.out.print(num % 2);
}
public static void main(String[] args) {
        DecToBin(21);
}
```

Задание 4.5.3

Используя рекурсию решить задачи на Informatics №№ 157, 199, 3050 (Ханойская башня).

Источники

[1] Рекурсия // Википедия. [2015—2015]. Дата обновления: 28.10.2015. URL: http://ru.wikipedia.org/?oldid=74165967.

[2] Стек вызовов // Википедия. [2015—2015]. Дата обновления: 19.11.2015. URL: http://ru.wikipedia.org/?oldid=74613896.