# Лекция\_1 Рекурсия: Базовые алгоритмы.

## Лабораторные

### Лабораторная: proc.recursion.hanoi\_tower

Проверьте себя, сделав лабораторную по рекурсивному решению задачи о Ханойской башне.

### proc.recursion.hanoi\_tower

Проставьте правильные аргументы методов в рекурсивных вызовах exchange вместо вопросительных знаков. Итоговый код должен корректно решать задачу о Ханойской башне.  
Решение должно иметь вид

public static void exchange(

Stack from, Stack help,

Stack to, int count) {

if (count > 0) {

exchange(?, ?, ?, ?);

int biggest = from.pop();

to.push(biggest);

exchange(?, ?, ?, ?);

}

}

import java.util.Stack;

public class HanoiSolver {

public static void exchange(Stack from, Stack help, Stack to, int count) {



}

}

### Лабораторная: proc.recursion.permutation

Проверьте себя, сделав лабораторную по рекурсивной генерации всех перестановок (объяснение на видео – ???).

### proc.recursion.permutation

Задача о генерации всех возможных перестановок элементов заданного списка.  
Из списка длины N можно построить N! возможных перестановок (N! = 1 \* 2 \* … \* (N – 1) \* N, 1!=1, 2!=1\*2=2, 3!=1\*2\*3=6, 4!=1\*2\*3\*4=24, …).  
Пример:  
Из элементов списка [10, 20, 30] можно построить 6 различных перестановок:  
[20, 30, 20]  
[10, 20, 30]  
[30, 20, 10]  
[20, 10, 30]  
[30, 10, 20]  
[10, 30, 20]

Схема рекурсивного решения:  
Все возможные перестановки N элементов ([a(1), a(2), a(3), ..., a(N)]) распадаются на N непересекающихся множеств:  
1) те, что начинаются с a(1)  
2) те, что начинаются с a(2)  
3) те, что начинаются с a(3)  
…  
N) те, что начинаются с a(N)  
При этом множество перестановок начинающихся с a(K) (для К = 1,2,3,…N) можно построить следующим образом:  
1) первым элементов поставим a(K)  
2) все остальные позиции занимают всевозможные перестановки оставшихся элементов [a(1), a(2), ..., a(K-1), a(K+1), ..., a(N-1), a(N)]

Вот графическая схема решения для списка [10, 20, 30]

| -> [10, 20, ??] -> | -> [10, 20, 30]

| -> [10, ??, ??] -> |

| | -> [10, 30, ??] -> | -> [10, 30, 20]

|

| | -> [20, 10, ??] -> | -> [20, 10, 30]

[??, ??, ??] -> | -> [20, ??, ??] -> |

| | -> [20, 30, ??] -> | -> [20, 30, 10]

|

| | -> [30, 10, ??] -> | -> [30, 10, 20]

| -> [30, ??, ??] -> |

| -> [30, 20, ??] -> | -> [30, 20, 10]

Которая приводит к множеству перестановок:  
[10, 20, 30]  
[10, 30, 20]  
[20, 10, 30]  
[20, 30, 20]  
[30, 10, 20]  
[30, 20, 10]

Как это работает.  
1. В начале стоит ИЛИ 10 ИЛИ 20 ИЛИ 30;  
1.1 Если вначале стоит 10, то дальше стоит ИЛИ 20 ИЛИ 30  
1.1.1 Если вначале стоит 10, и дальше стоит 20, то осталось 30  
1.1.2 Если вначале стоит 10, и дальше стоит 30, то осталось 20  
1.2 Если вначале стоит 20, то дальше стоит ИЛИ 10 ИЛИ 30  
1.2.1 Если вначале стоит 20, и дальше стоит 10, то осталось 30  
1.2.2 Если вначале стоит 20, и дальше стоит 30, то осталось 10  
1.3 Если вначале стоит 30, то дальше стоит ИЛИ 10 ИЛИ 20  
1.3.1 Если вначале стоит 30, и дальше стоит 10, то осталось 20  
1.3.2 Если вначале стоит 30, и дальше стоит 20, то осталось 10

Это некорректный код, который должен рекурсивно строить все возможные перестановки.  
Исправьте его.

import java.util.Arrays;

public class Permutator {

public static void main(String[] args) {

int[] arr = {1, 2, 3};

permutation(arr, arr.length);

}

private static void permutation(int[] arr, int size) {

if (size < 2) {

System.out.println(Arrays.toString(arr));

} else {

for (int k = 0; k < size; k++) {

swap(arr, k, size - 1);

permutation(arr, size - 1);

}

}

}

private static void swap(int[] arr, int index0, int index1) {

int tmp = arr[index0];

arr[index0] = arr[index1];

arr[index1] = tmp;

}

}

>> [2, 3, 1]

>> [2, 3, 1]

>> [1, 2, 3]

>> [1, 2, 3]

>> [2, 1, 3]

>> [2, 1, 3]

Этот код “недоперемешивает” элементы. Количество вариантов верное, но по какой-то причине имеются дубликаты.

import java.util.ArrayList;

import java.util.List;

public class Permutator {

public static List<List<Integer>> permute(List<Integer> list, int size) {



}

private static void swap(List<Integer> list, int index0, int index1) {

Integer tmp = list.get(index0);

list.set(index0, list.get(index1));

list.set(index1, tmp);

}

}

### Лабораторная: proc.recursion.parser

Проверьте себя, сделав лабораторную по рекурсивному разбору арифметического выражения (объяснение на видео – ???).

### proc.recursion.parser

Это пример небольшого и ограниченного по функциональности рекурсивного парсера арифметических выражений.  
Он умеет:  
1) встретив скобки с двух сторон – свести задачу к анализу содержимого скобок (пример:”(1+1)” -> “1+1″, пример “(1+(2/3))” -> “1+(2/3)”);  
2) встретив число-знак-XXX – свести задачу к анализу XXX  
(пример:”1\*(10-3)” -> “(10-3)”, пример “3+(2/(9+1))” -> “(2/(9+1)”);  
Задание: переписать его, что бы он ВМЕСТО пункта 2 дела пункт 2′  
2′) встретив XXX-знак-число – сведил задачу к анализу XXX.  
(пример:”(10-3)\*1″ -> “(10-3)”, пример “(2/(9+1))+3″ -> “(2/(9+1)”);  
Предупреждение #1: это не полноценный парсер арифметических выражений, есть множество корректных  
выражений на которых он “падает” или вычисляет некорректно (“(1+1)+(1+1)”, “-1″, …).  
Предупреждение #2: в условиях лабораторной ожидается, что старые выражения вида число-знак-XXX  
больше на разбираются.

public class Parser {

public static void main(String[] args) {

System.out.println(eval("123"));

System.out.println(eval("2\*3"));

System.out.println(eval("2\*(1+3)"));

System.out.println(eval("1+(5-2\*(13/6))"));

}

private static int eval(String expr) {

return eval(expr, 0, expr.length());

}

private static int eval(String expr, int from, int to) {

if (expr.charAt(from) == '(') {

return eval(expr, from + 1, to - 1);

} else {

int pos = from;

while (pos < to) {

if (Character.isDigit(expr.charAt(pos))) {

pos++;

} else {

int leftOperand = Integer.valueOf(expr.substring(from, pos));

char operation = expr.charAt(pos);

int rightOperand = eval(expr, pos + 1, to);

switch (operation) {

case '+':

return leftOperand + rightOperand;

case '-':

return leftOperand - rightOperand;

case '\*':

return leftOperand \* rightOperand;

case '/':

return leftOperand / rightOperand;

}

}

}

return Integer.valueOf(expr.substring(from, to));

}

}

}

Тестирующий код (для моего примера)

public class ParserTest {

public static void main(String[] args) {

System.out.println(">> 123 = " + Parser.eval("123"));

System.out.println(">> 2\*3 = " + Parser.eval("2\*3"));

System.out.println(">> 2\*(1+3) = " + Parser.eval("2\*(1+3)"));

System.out.println(">> 1+(5-2\*(13/6)) = " + Parser.eval("1+(5-2\*(13/6))"));

}

}

Тестирующий код (для вашего решения)

public class ParserTest {

public static void main(String[] args) {

System.out.println(">> 123 = " + Parser.eval("123"));

System.out.println(">> 2\*3 = " + Parser.eval("2\*3"));

System.out.println(">> (1+3)\*2 = " + Parser.eval("(1+3)\*2"));

System.out.println(">> ((13/6)\*2-5)+1 = " + Parser.eval("((13/6)\*2-5)+1"));

}

}

Демонстрация работы метода Character.isDidit(...)

public class IsDigitDemo {

public static void main(String[] args) {

System.out.println(Character.isDigit('0'));

System.out.println(Character.isDigit('2'));

System.out.println(Character.isDigit('4'));

System.out.println(Character.isDigit('5'));

System.out.println(Character.isDigit('7'));

System.out.println(Character.isDigit('9'));

System.out.println(Character.isDigit(' '));

System.out.println(Character.isDigit('A'));

System.out.println(Character.isDigit('!'));

}

}

>> true

>> true

>> true

>> true

>> true

>> true

>> false

>> false

>> false

Демонстрация работы метода Integer.valueOf(...)

public class ValueOfDemo {

public static void main(String[] args) {

int value0 = Integer.valueOf("123");

System.out.println(value0);

int value1 = Integer.valueOf("abc");

System.out.println(value1);

}

}

>> 123

>> ... java.lang.NumberFormatException: For input string: "abc" ...

Демонстрация работы метода String.substring(...)

public class SubstringDemo {

public static void main(String[] args) {

System.out.println("Hello!".substring(0, 6));

System.out.println("Hello!".substring(0, 5));

System.out.println("Hello!".substring(0, 4));

System.out.println("Hello!".substring(0, 3));

System.out.println();

System.out.println("Hello!".substring(0, 6));

System.out.println("Hello!".substring(1, 6));

System.out.println("Hello!".substring(2, 6));

System.out.println("Hello!".substring(3, 6));

System.out.println();

System.out.println("Hello!".substring(0, 6));

System.out.println("Hello!".substring(1, 5));

System.out.println("Hello!".substring(2, 3));

}

}

>> Hello!

>> Hello

>> Hell

>> Hel

>>

>> Hello!

>> ello!

>> llo!

>> lo!

>>

>> Hello!

>> ello

>> ll

public class Parser {

public static int eval(String expr) {



}

}