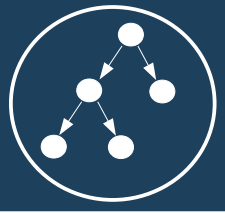


# Data Structures and Algorithms

## Алгоритмы.
## Рекурсия. Реализация в Python и Java.

# Рекурсия

Рекурсия — описание какого то объекта или процесса внутри этого же объекта или процесса. Т.е. рекурсия это описание объекта которые содержит подобный себе объект внутри, или процесса который в свою очередь состоит из таких же процессов. И хотя определение рекурсии встречается в различных областях человеческой деятельности, в дальнейшем мы будет рассматривать рекурсию в программировании.

# Рекурсия в программировании

Рекурсия — возможность функции или процедуры вызывать саму себя.

В зависимости от того, как реализуется рекурсивный вызов рассматривают следующие виды рекурсии:

- Простая — функция или процедура вызывают сами себя.
- Сложная или косвенная — вызов функции происходит посредством вспомогательных функций. Например основная функция вызывает функцию А, а уже в свою очередь функция А вызывает основную функцию.

В зависимости от того сколько раз (за один вызов базовой) производиться рекурсивных вызовов рекурсия подразделяется на:

- Одиночная — в рекурсивной части производиться ровно один рекурсивный вызов.
- Множественная (параллельная рекурсия) — в рекурсивной части производиться более одного вызова рекурсивной функции или процедуры.

Анонимная рекурсия — рекурсия с использованием неявных реализаций функций или процедур (лямбда функции и подобные им механизмы).

Глубина рекурсии — количество вложенных вызовов функции или процедуры.

# Описание базовых составляющих рекурсии

Рекурсивная функция или процедура всегда должна содержать условный оператор описывающий условие прекращение рекурсии - терминальная часть. Терминальная часть выполняется если условие прекращения рекурсии вернет true. После терминальной части идет описание рекурсивной части. Рекурсивная часть должна выполняться в случае когда условие прекращение рекурсии вернет false. Правильно реализованная рекурсивная функция или процедура должна давать гарантию завершения за конечное число вызовов.

# Как обычно реализуется рекурсия

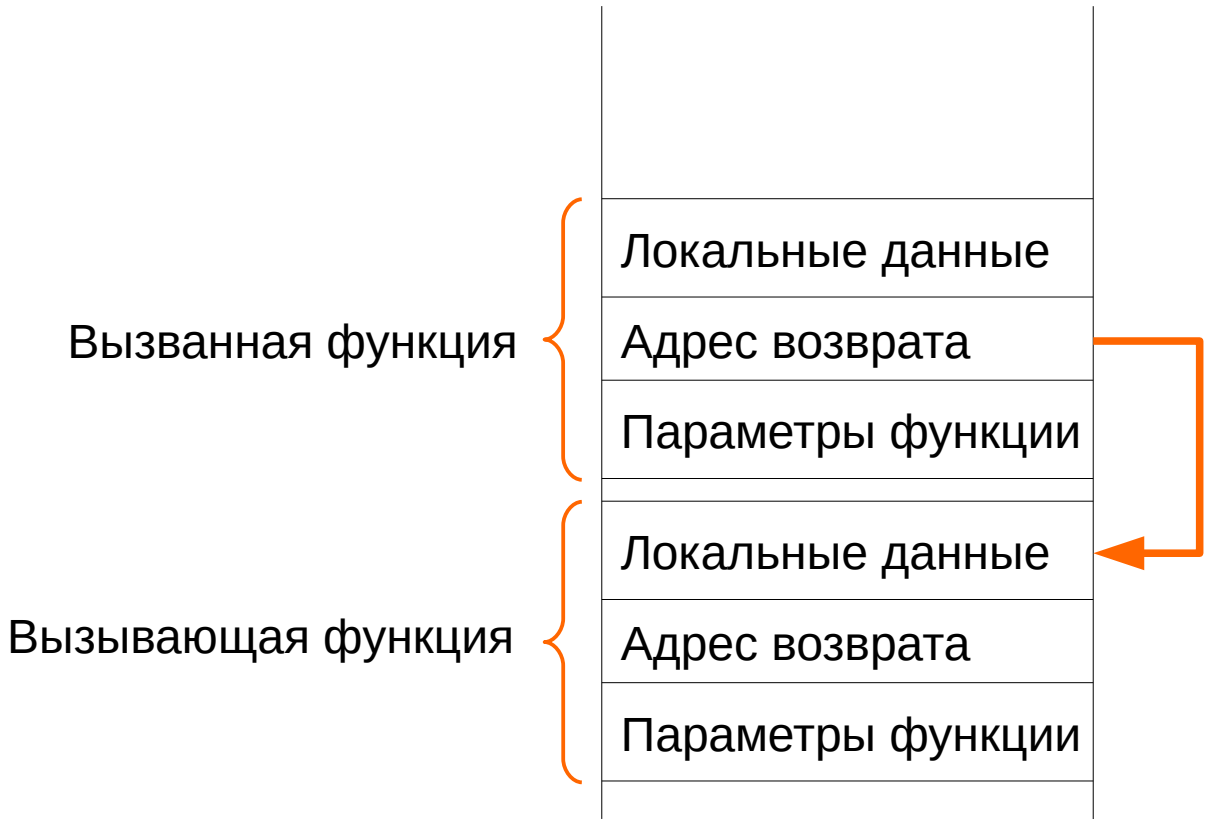
В большинстве языков программирования реализация рекурсии опирается на механизм стека вызовов. Переменные функции и адрес возврата сохраняются в стек (область оперативной памяти используемой ПО), благодаря чему каждый следующий рекурсивный вызов этой функции пользуется своим набором локальных переменных и за счёт этого работает корректно.

С другой стороны каждый рекурсивный вызов требуется некоторое количество оперативной памяти компьютера, и при чрезмерно большой глубине рекурсии может наступить переполнение стека вызовов (классический Stack Overflow).

Поэтому обычно не рекомендуют использовать рекурсивные методы с большой глубиной рекурсии. В таком случае если есть возможность, то следует попробовать использовать циклический подход.

Хвостовая рекурсия - частный случай рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции. Такой вызов может быть оптимизирован компилятором языка в простую итерацию, как следствие переполнение стека не произойдет.

# Пример сохранения данных на стеке



# Когда стоит использовать рекурсию

Рекурсивный подход облегчает реализацию в нескольких случаях:

1) Задача разбивается на подзадачи. Но каждая подзадача эквивалентна базовой задаче.
2) Задача изначально сформулирована с помощью рекурсивного описания.

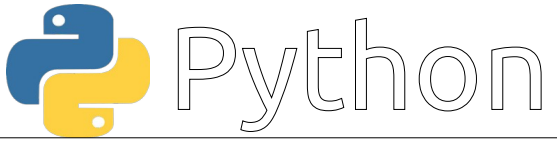
# Поддержка рекурсии


Поддерживается:

- Прямой рекурсивный вызов методов
- Косвенный рекурсивный вызов методов
- Параллельная рекурсия

Не поддерживается:

- Оптимизация хвостовой рекурсии




Поддерживается:

- Прямой рекурсивный вызов методов
- Косвенный рекурсивный вызов методов
- Параллельная рекурсия

Не поддерживается:

- Оптимизация хвостовой рекурсии

# Реализация рекурсии в Python

Прямой рекурсивный вызов функции в Python реализован довольно просто, достаточно в теле функции применить оператор вызова функции к идентификатору этой функции. Если существует еще одна ссылка которая также указывает на текущую функцию, то можно использовать и ее.

В качестве примера приведем функцию которая считает количество пробелов в строке текста. Сначала реализуем эту функцию циклически, а впоследствии рекурсивно.

# Реализация с помощью цикла

```
def counting_white_space_cyclically(text):
    n = 0
    for symbol in text:
        if symbol == " ":
            n = n+1
    return n
```

В этом примере продемонстрирована реализация  с помощью цикла. Переменная цикла проходит по всем символам строки и если текущий символ равен пробелу то увеличиваем переменную для хранения количества пробелов. После цикла возвращаем значение этой переменной.

# Реализация с помощью рекурсии

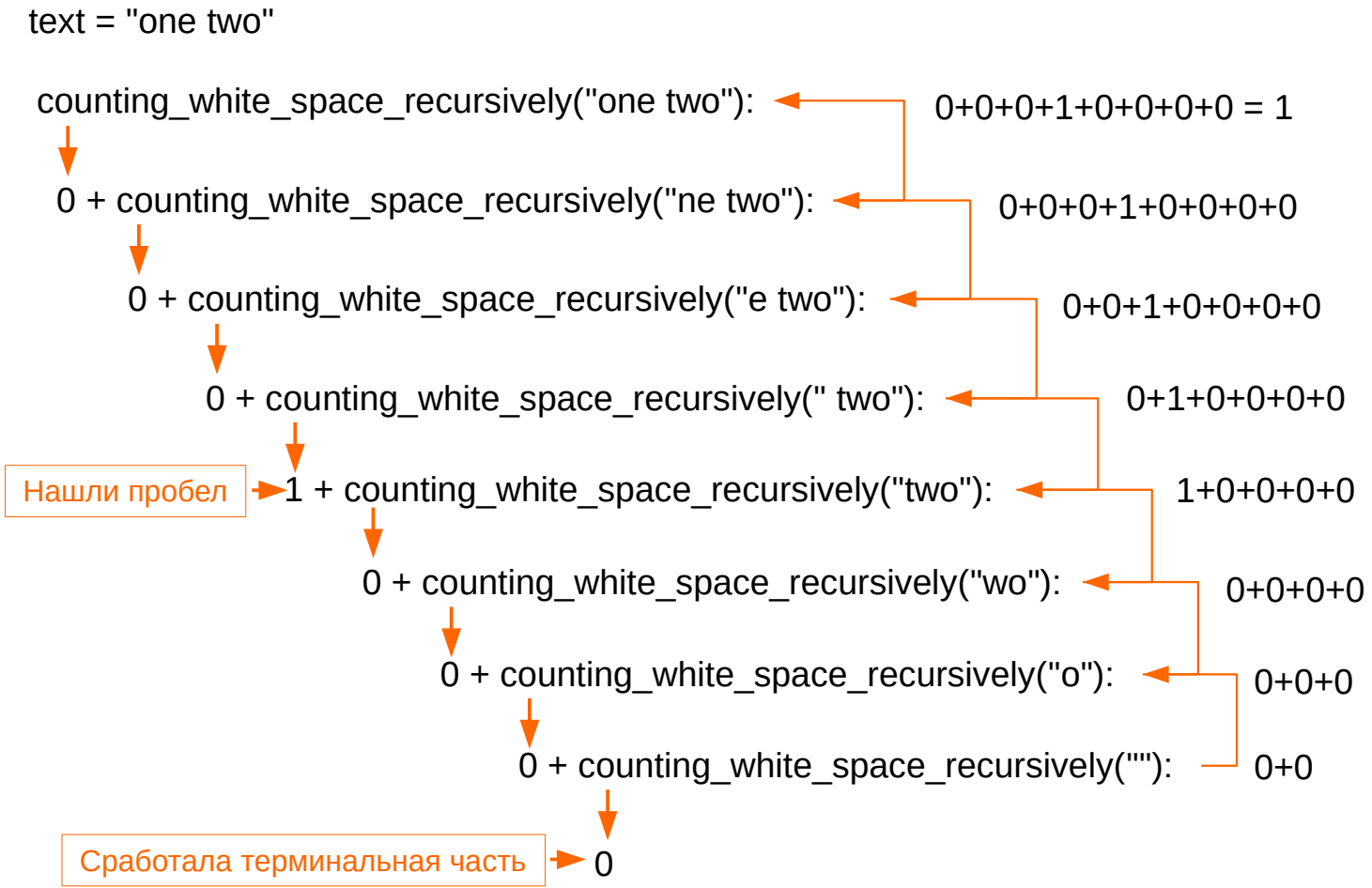
```
def counting_white_space_recursively(text):
    print(text)
    if len(text) == 0:
        return 0
    n = 0
    if text[0] == " ":
        n = 1
    return n + counting_white_space_recursively(text[1:])
```

Терминальная часть

Прямой рекурсивный вызов

В этом примере продемонстрирована реализация  с помощью рекурсии. Терминальная стадия выполняет проверку длинны строки и если длинна строки равна 0, то возвращаем 0. Потом проверяем какой символ стоит на первом месте в этой строке. Если это пробел то вернуть число 1 плюс рекурсивный вызов этой же функции. Но теперь в качестве параметра используется строка без первого символа (вы уменьшили ее длину на единицу).

# Принцип работы рекурсивной функции



# Принцип работы рекурсивной функции

На предыдущем слайде показан принцип работы рекурсивной функции для подсчета количества пробелов в строке. Терминальная часть содержит проверку длинны строки и если длинна строки равна 0, то нужно вернуть ноль и все (строка в которой ничего нет, не содержит пробелов). В рекурсивной части проверяем первый символ этой строки и если это пробел, то стоит вернуть 1 плюс вызов этой же функции где в качестве параметра используется строка но без первого символа. Так, как строка параметр на каждом вызове уменьшается на один символ то как только ее длинна станет равно 0, то сработает терминальная часть и рекурсивные вызовы будут окончены. Начнется процесс возвратов результатов, и начиная от терминальной части происходит обратный подъем по стеку вызовов до базового вызова.

# Хвостовая рекурсия

Хвостовая рекурсия — особый вид рекурсии при котором вызов рекурсивной функции является единственным выражением в операторе возврата. Предыдущий пример **не является хвостовой рекурсией** потому что оператор возврата выглядит как:

```
return n + counting_white_space_recursively(text[1:])
```

В этом примере после оператора возврата идет еще сложение с результатом рекурсивного вызова.

Для демонстрации принципа хвостовой рекурсии, функцию придется изменить так что бы после оператора возврата был только рекурсивный вызов.

**Внимание!** Python (версия 3.9) не производит оптимизацию хвостовой рекурсии, так что особого смысла в ее применении в данном языке **нет**.

# Пример хвостовой рекурсии

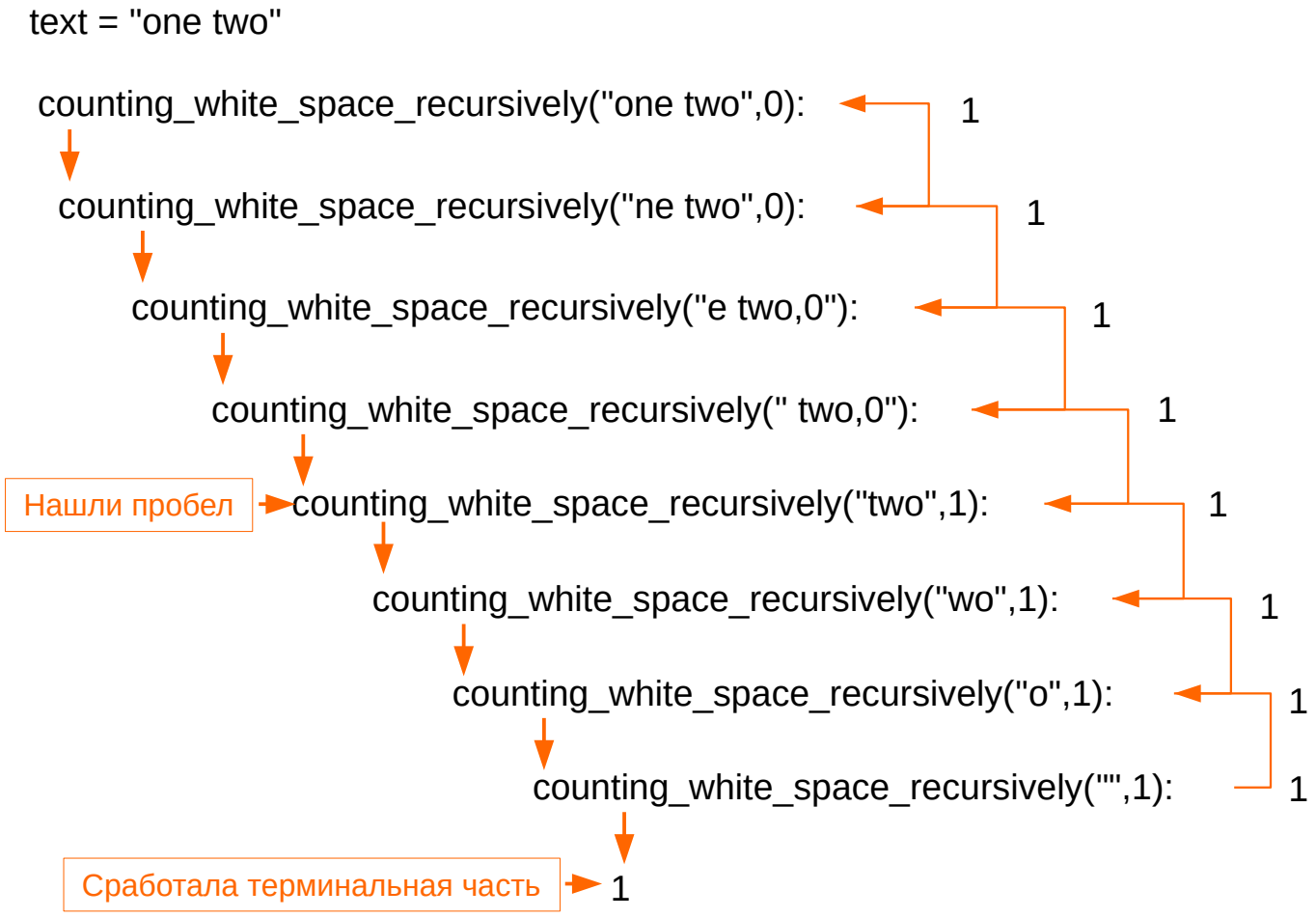
```
def counting_white_space_tail_recursion(text, count=0):
    if len(text) == 0:
        return count
    n = 0
    if text[0] == " ":
        n = 1
    return counting_white_space_tail_recursion(text[1:], count+n)
```

Терминальная часть

Прямой рекурсивный вызов

Это пример хвостовой рекурсии. После оператора возврата описан только рекурсивный вызов. Для ее реализации этого приема бы добавлен дополнительный параметр функции в котором будет храниться количество найденных пробелов.

# Принцип работы хвостовой рекурсии



# Превышение допустимой глубины рекурсии

Для демонстрации превышения допустимой глубины рекурсии напишем рекурсивную функцию вычисления факториала.

```
def factorial(number):
    if number <= 1:
        return 1
    else:
        return number * factorial(number-1)
```

Терминальная часть

Рекурсивная часть

Данная функция прекрасно работает для небольших значений. Однако если в качестве параметра передать значение 1000. То программа завершиться с исключением:

RecursionError: maximum recursion depth exceeded in comparison

Это связанно с тем, что по умолчанию в Python установлено ограничение на глубину рекурсивных вызовов равное 1000 вызовов.

# Ограничение максимальной глубины рекурсивных вызовов в Python

В Python установлено ограничение на глубину рекурсивных вызовов. По умолчанию это значение равно 1000. Узнать значение этого ограничения можно используя функцию getrecursionlimit() модуля sys. Ниже приведен пример кода который выведет на экран это ограничение:

```
import sys
print(sys.getrecursionlimit())
```

При желании вы можете установить новое ограничение. Используя функцию setrecursionlimit(limit) из этого же модуля. Например вот таким способом вы можете поднять ограничение до 2000 вызовов:

```
import sys

sys.setrecursionlimit(2000)
```

**Внимание!** Не стоит увлекаться установкой слишком больших значений так как это может привести в повышенному потреблению оперативной памяти.

# Пример когда рекурсия действительно упрощает решение

Предположим у нас есть список разных элементов. Это могут быть как обычные элементы (например числа) так и другие списки, и даже списки списков и так далее. Задача создать список в котором будут элементы хранящиеся во всех списках в базовом списке. Например:

base_list = [1,[2,3],[4,5,[6,7]],[8,[9]],10]  => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

И хотя эту задачу можно решить циклически, но гораздо проще она решается рекурсивно.

# Пример рекурсивного решения

```
def get_element(base_list, result_list=None):
    if result_list is None:
        result_list = []
    for element in base_list:
        if type(element) == list:
            get_element(element, result_list)
        else:
            result_list.append(element)
    return result_list
```

Решение этого задания в рекурсивном стиле довольно простое. Перебираем элементы базового списка и если текущий элемент список, то вызываем этот же метод еще раз передавая текущий элемент в качестве параметра. Если же текущий элемент не список, то просто добавляем его в список результат.

# Реализация рекурсии в Java

Рекурсия в Java реализована также довольно просто. Для рекурсивного вызова метода достаточно просто вызвать его в теле этого метода (прямой рекурсивный вызов), или в другом методе вызванном из базового (косвенная рекурсия).

Так же как и в Python, в Java нет оптимизации хвостовой рекурсии.

# Пример реализации рекурсии в Java

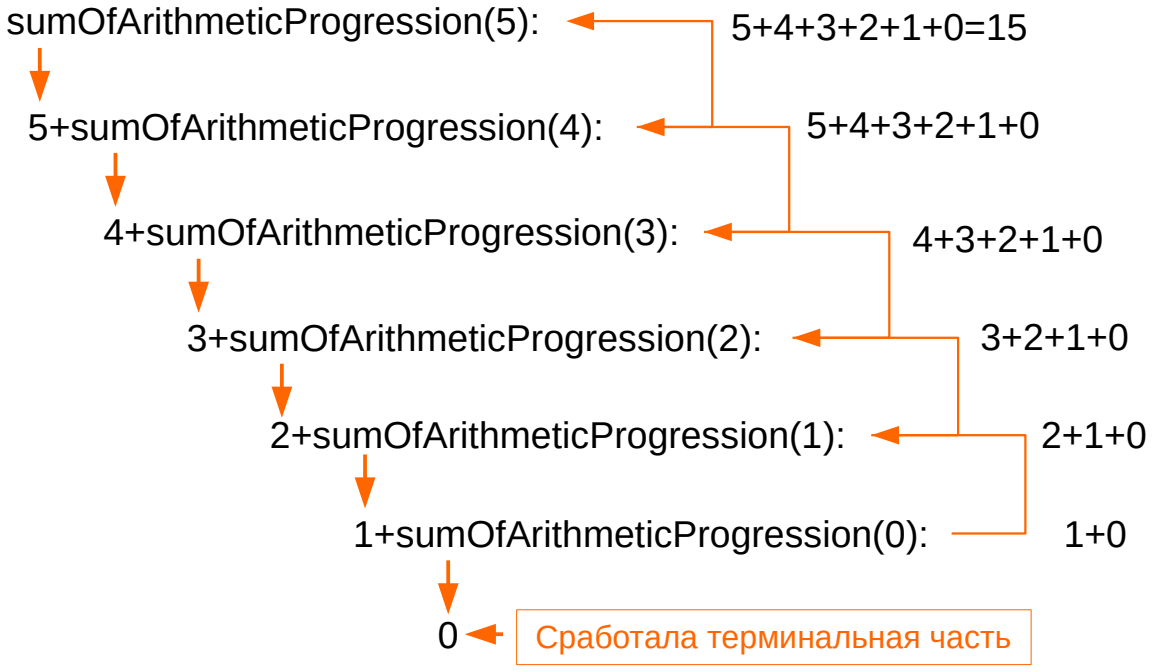
```
public static int sumOfArithmeticProgression(int number) {

    if (number <= 0) {
        return 0;
    } else {
        return number + sumOfArithmeticProgression(number - 1);
    }

}
```

Терминальная часть

Рекурсивная часть

В примере показан метод для вычисления суммы арифметической прогрессии.

# Пример реализации рекурсии в Java



# Превышение глубины рекурсивных вызовов

Для демонстрации превышения допустимой глубины рекурсии напишем рекурсивную функцию вычисления факториала.

```
public static BigInteger factorial(int number) {
    if (number <= 1) {
        return new BigInteger("1");
    }
    return BigInteger.valueOf(number).multiply(factorial(number - 1));
}
```

В Java ограничение идет не на количество рекурсивных вызовов, а на объем оперативной памяти занимаемой стеком вызовов методов. Поэтому чем больше локальных переменных в методе и чем больше у метода параметров тем быстрее переполняется стек и генерируется исключение

java.lang.StackOverflowError

Этот метод генерирует данное исключение при приблизительно 12000 вызовов.

# Увеличение объема стека в Java

За выполнение программ на Java отвечает JVM(Java Virtual Machine) - основная часть исполняющей системы Java. Данная платформа поддерживает возможности тонкой настройки почти всех аспектов выполнения программ, в том числе и управление памятью отводимой под стек. Для этого используются ключи запуска JVM:

- -Xms size - Устанавливает начальный размер (в байтах) кучи. Это значение должно быть кратным 1024 и превышать 1 МБ.
- -Xmx size - Задает максимальный размер (в байтах) пула распределения памяти в байтах. Это значение должно быть кратным 1024 и превышать 2 МБ.
- -Xss size - Устанавливает размер стека потока (в байтах).

Добавьте букву k или K для обозначения KB, m или M для обозначения MB и g или G для обозначения GB.

Размер стека по умолчанию зависит от типа ОС. Обычно это 1 МБ.

Если использовать такой ключ запуска: java -Xss100M — то будет установлен размер стека в 100 Мбайт, чего с избытком хватит для вычисления данной задачи.

# Список литературы

1) Головешкин В.А. Ульянов М.В. « Теория рекурсии для программистов ». М.: Физматлит, 2006. — 296 с.
2) https://docs.python.org/3/library/sys.html
3) https://docs.oracle.com/en/java/javase/11/tools/java.html#GUID-3B1CE181-CD30-4178-9602-230B800D4FAE