* [ОБЗОР КУРСА](https://lyceum.yandex.ru/courses/123/groups/568)

[Урок Рекурсия](https://lyceum.yandex.ru/courses/123/groups/568/lessons/668)

**Рекурсия**

**План урока**

1

[Задачи на сочетания](https://lyceum.yandex.ru/courses/123/groups/568/lessons/668/materials/1348#1)

2

[Определение рекурсии, принцип работы](https://lyceum.yandex.ru/courses/123/groups/568/lessons/668/materials/1348#2)

3

[Рекурсивные определения. Рекурсия как обобщение цикла](https://lyceum.yandex.ru/courses/123/groups/568/lessons/668/materials/1348#3)

4

[Опасности использования рекурсивных алгоритмов](https://lyceum.yandex.ru/courses/123/groups/568/lessons/668/materials/1348#4)

5

[Красота требует жертв?](https://lyceum.yandex.ru/courses/123/groups/568/lessons/668/materials/1348#5)

6

[Несколько рекурсивных веток. Деревья](https://lyceum.yandex.ru/courses/123/groups/568/lessons/668/materials/1348#6)

7

[Бонус. Решаем Судоку](https://lyceum.yandex.ru/courses/123/groups/568/lessons/668/materials/1348#7)

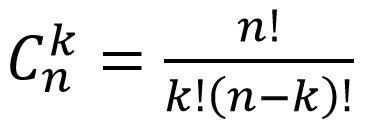
**Аннотация**

*На данном занятии мы познакомимся с понятием рекурсии, покажем ее связь с уже знакомыми нам конструкциями (циклы и функции). Также разберем наиболее часто встречаемые ошибки и «классические» примеры.*

**1. Задача на сочетания**

Задача на сочетания — это простейшая комбинаторная задача на сочетания без повторений: сколькими способами можно из данных n предметов выбрать некоторые k предметов, если порядок их выбора не важен?

Ответом на эту задачу является величина

,

называемая числом сочетаний из n элементов по k. Запись n! обозначает произведение 1 · 2 · 3 · ... · n, называемое факториалом числа n (мы уже неоднократно сталкивались с данным понятием), при этом считается, что 0! = 1.

*# Задача на сочетания*

**def** f(n):

res = 1

**for** i **in** range(2, n + 1):

res \*= i

**return** res

time=(f(10) / (f(3) \* f(10 - 3))) \* 2

**print**(time)

**2. Определение рекурсии, принцип работы**

Однако эту задачу можно решить и по-другому. В математике очень часто для упрощения вычислений, исходную задачу сводят к более простым.

**Рекурсия**

В итоге можно прийти к тому, что будет вызвана первоначальная задача, но в несколько упрощенной форме. Такой прием называется рекурсией от лат. recurcio — возвращение. Итак, в программировании рекурсия — это подпрограмма, обращающаяся сама к себе (непосредственно или через цепочку подпрограмм).

Вернёмся к нашей задаче и рассмотрим функцию вычисления факториала с несколько другой стороны, постараемся применить рекурсию. Известно, что 0! = 1, 1! = 1. А как вычислить величину n! для большого n? Если бы мы могли вычислить величину (n-1)!, то тогда мы легко вычислим n!, поскольку n! = n · (n-1)!. Но как вычислить (n-1)!? Если бы мы вычислили (n-2)!, то мы сможем вычислить и (n-1)! = (n-1) · (n-2)!. А как вычислить (n-2)!? Если бы... В конце концов, мы дойдем до величины 0!, которая равна 1. Таким образом, для вычисления факториала мы можем использовать значение факториала для меньшего числа.

Давайте напишем соответствующую функцию:

*# Вычисление факториала*

**def** factorial(n):

**if** n == 0:

**return** 1

**else**:

**return** n \* factorial(n - 1)

Логическая сложность рекурсивных функций заключается в изменении параметров и особенностях получения промежуточных результатов при последовательном обращении подпрограммы к себе. Выполняется две серии шагов. Первая серия — это шаги рекурсивного погружения подпрограмм в себя до тех пор, пока выбранный параметр не достигнет граничного значения (глубина рекурсии). Вторая серия — это шаги рекурсивного выхода до тех пор, пока значение выбранного параметра не достигнет начального. Она, как правило, и обеспечивает получение промежуточных и конечных результатов. Отличной иллюстрацией сказанного служит программа:

*# Принцип работы рекурсивной функции*

**def** depth(n):

**print**('Погружение ', n);

**if** (n > 1):

depth(n - 1)

**else**:

**print**('\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Выход', n)

depth(6)

На практике необходимо убеждаться, чтобы глубина рекурсий была мала.

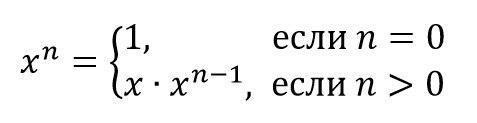
В общем случае рекурсия тяготеет к [декларативному](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) стилю программирования. Если в двух словах, то когда мы пишем императивную функцию (как делали всё время до этого), то отвечаем на вопрос «Как достигнуть необходимого результата?», а когда создаём декларативную — на вопрос «Что такое наш результат?».

**3. Рекурсивные определения. Рекурсия как обобщение цикла**

**Рекурсивное определение**

Рекурсивное определение — часто применяемый в математике способ задания функций, при котором значение искомой функции в данной точке определяется через ее значения в предшествующих точках.

Мощность рекурсивного определения заключается, очевидно, в том, что оно позволяет с помощью конечного высказывания определить бесконечное множество объектов. Например, функцию степени числа с целым неотрицательным показателем, можно представить так:



Если известно, что нечто можно описать рекурсивно, то это можно решать рекурсивно, но не значит, что это нужно делать.

Рекурсия также является обобщением цикла. Пример ниже демонстрирует простую замену цикла рекурсией. Однако гораздо чаще следует делать обратную замену, так как рекурсии требуют дополнительной памяти и замедляют работу программ.

Цикл:

**def** iter():

**global** i, s1

**while** i < 5:

i += 1

s1 += i

i, s1 = 0, 0

iter()

**print**('Циклы:', s1)

Рекурсия:

**def** rec():

**global** i, s2

**if** i < 5:

i += 1

s2 += i

rec()

i, s2 = 0, 0

rec()

**print**('Рекурсия:', s2)

**4. Опасности использования рекурсивных алгоритмов**

**Важно**

Наиболее распространённой ошибкой при использовании рекурсии является бесконечная рекурсия, когда цепочка вызовов функций никогда не завершается и продолжается, пока не кончится свободная память в компьютере.

Определим две наиболее распространенные причины для бесконечной рекурсии на примере некорректно написанной функции нахождения факториала числа (рекурсивное отношение которой мы уже рассматривали с вами сегодня).

**def** factorial(n):

**return** n \* factorial(n - 1)

**def** factorial(n):

**if** n == 0:

**return** 1

**else**:

**return** n \* factorial(n)

Итак, при разработке рекурсивной функции необходимо, прежде всего, оформлять условия завершения рекурсии и думать, почему рекурсия когда-либо завершит работу.

**Важно**

Ещё одна проблема, связанная с использованием рекурсивных функций, — это нетривиальность задачи оценки сложности и эффективности алгоритма. Сложность этих алгоритмов зависит не только от сложности внутренних циклов, но и от количества итераций рекурсии. Рекурсивная процедура может выглядеть достаточно простой, но она может серьёзно усложнить программу, многократно вызывая себя.

**5. Красота требует жертв?**

Напишем программу перевода числа из десятичной системы счисления в двоичную (а на самом деле и в любую другую позиционную систему).

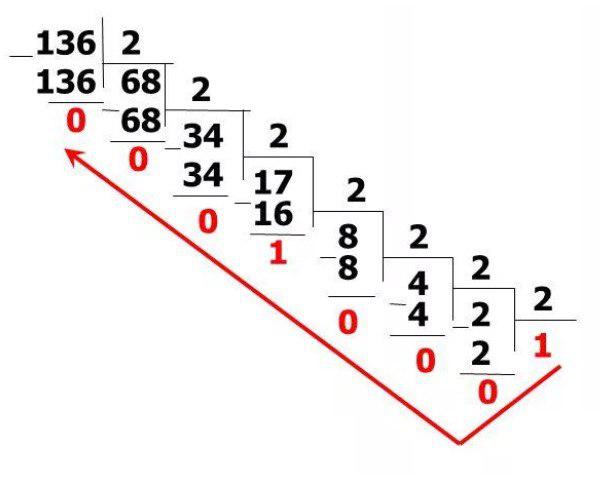
Для начала вспомним базовый алгоритм перевода:

**Шаг 1.** Разделить число на основание системы счисления в которую осуществляется перевод (в нашем случае два). Запишите остаток

**Шаг 2.** Если результат деления больше или равен 2, продолжать делить его на 2 до тех пор, пока результат деления не станет равен 1.

**Шаг 3.** Выписать результат последнего деления и все остатки от деления в обратном порядке в одну строку.

Рассмотрим на примере перевода числа 136 в двоичную систему счисления



13610 = 100010002

С помощью рекурсии реализовать этот алгоритм можно очень красивым и лаконичным кодом.

**def** bin(a):

**if** a > 1:

bin(a // 2)

**print**(a % 2)

**6. Несколько рекурсивных веток. Деревья**

Рассмотрим в качестве примера функцию, вычисляющую числа Фибоначчи.

Числа Фибоначчи — это ряд чисел: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89..., в котором два первых элемента равны 1, а каждый следующий — сумме двух предыдущих. Удивительно, что отношение двух соседних чисел Фибоначчи стремится к числу золотого сечения: 1,6180339887.

Составим рекурсивное определение этих чисел и сразу запишем его в виде функции:

**def** fib(n):

**if** n <= 1:

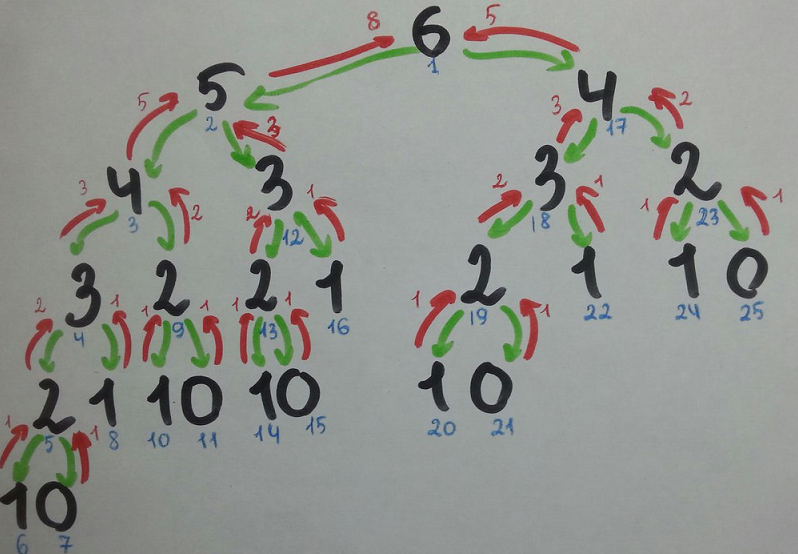
**return** 1

**else**:

**return** fib(n - 1) + fib(n - 2)

Это удивительно, но программа почти слово в слово совпадает с определением чисел Фибоначчи!

Однако в этом примере мы столкнулись с новым типом рекурсии, в котором функция порождает целых две рекурсивные ветки. Неявно во время выполнения программы мы обходим дерево в глубину. Проиллюстрируем это на примере **fib(6)**:



Важно понимать, что экземпляры функции выполняются не параллельно, а в детерминированной (то есть определённой) последовательности: сначала левое поддерево, а потом всё правое поддерево из любой вершины. Синим показан порядок прохождения, красным — возвращаемые значения, зелёным — структура дерева.

Интересным является тот факт, что дерево очень быстро разрастается при росте номера числа Фибоначчи, что влечет замедление программы и трату памяти.

Кстати, следующее число Фибоначчи вычисляется ровно в золотое сечение раз медленнее, чем предыдущее. Таким образом, **fib(500)** вычислится уже после того, как исчезнет Солнечная система.

Следующий пример демонстрирует этот факт. Запустите его и убедитесь:

**from** time **import** time

**for** i **in** range(20, 30):

s = time()

**print**(i, fib(i), "%.03f" % (time() - s))

Как видим, никакого **кэширования** (запоминания предыдущих вычислений) не происходит. Кстати, такие оптимизации присутствуют в функциональных языках (LISP, Haskell). К сожалению, в Python по умолчанию такие оптимизации недоступны, но можно искуственно сделать что-то подобное.

В следующем примере запоминаются последние 1000 вызовов функции **fib**.

**from** functools **import** lru\_cache

@lru\_cache(maxsize=1000)

**def** fib(n):

**if** n <= 1:

**return** 1

**else**:

**return** fib(n - 1) + fib(n - 2)

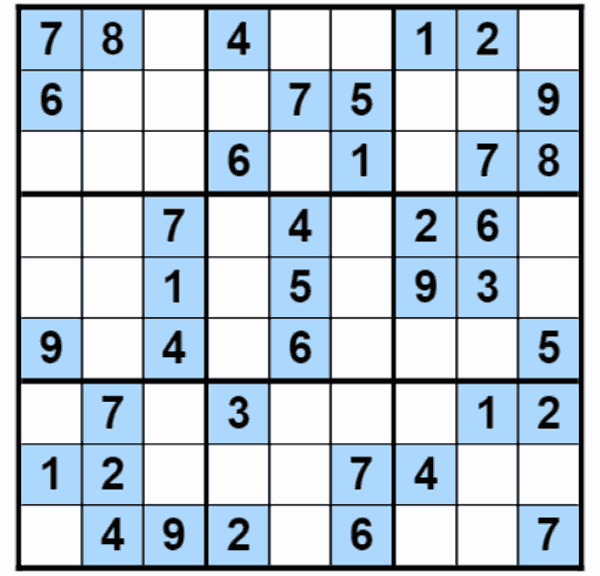
**print**(fib(100))

Сделаем главный вывод:

**Важно**

Рекурсивный метод обеспечивает удобный обход списка, дерева или графа, при этом контролируя перемещения по элементам и возвращение к предыдущим состояниям. Этим можно пользоваться во многих математических и прикладных задачах.

**7. Бонус. Решаем Судоку**



Вы уже могли сталкиваться с этой задачей раньше. Вспомните, как мы её решали? Попробуем теперь предложить иной способ её решения.

Предположим, что нам нужно сделать программу, которая разгадывает Судоку. Пусть поле моделируется списком списков с целыми числами:

field = [

[0,0,0,0,0,0,0,0,0],

[0,1,0,0,2,0,0,3,0],

[0,0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0,0],

[0,4,0,0,5,0,0,6,0],

[0,0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0,0],

[0,7,0,0,8,0,0,9,0],

[0,0,0,0,0,0,0,0,0],

]

Сформулируем рекурсивный алгоритм решения судоку.

* Если на поле судоку нет пустых клеток, то оно уже решено и надо просто вернуть поле в качестве решения.
* Если есть пустые клетки, надо вычислить какую-либо пустую клетку, для которой количество возможных вариантов минимально. Попробовать по очереди проверять эти варианты, и, если будет найдено решение, вернуть его.

Сама функция будет не сильно больше данного описания:

**from** pprint **import** pprint

**from** copy **import** deepcopy

**from** random **import** shuffle

**from** time **import** clock

"""

Для всех клеток на основе ограничений возвращает список возможных чисел,  
например:

(0, 0, {2, 3, 4, 5, 6, 7, 8, 9})

(0, 1, {2, 3, 5, 6, 8, 9})

(0, 2, {2, 3, 4, 5, 6, 7, 8, 9})

(0, 3, {1, 3, 4, 5, 6, 7, 8, 9})

(0, 4, {1, 3, 4, 6, 7, 9})

(0, 5, {1, 3, 4, 5, 6, 7, 8, 9})

(0, 6, {1, 2, 4, 5, 6, 7, 8, 9})

(0, 7, {1, 2, 4, 5, 7, 8})

(0, 8, {1, 2, 4, 5, 6, 7, 8, 9})

(1, 0, {4, 5, 6, 7, 8, 9})

"""

**def** get\_variants(sudoku):

variants = []

**for** i, row **in** enumerate(sudoku):

**for** j, value **in** enumerate(row):

**if** **not** value:

*# значения в строке*

row\_values = set(row)

*# значения в столбце*

column\_values = set([sudoku[k][j] **for** k **in** range(9)])

*# в каком квадрате 3x3 находится клетка?  
 # Координаты этого квадрата*

sq\_y = i // 3

sq\_x = j // 3

square3x3\_values = set([

sudoku[m][n]

**for** m **in** range(sq\_y \* 3, sq\_y \* 3 + 3)

**for** n **in** range(sq\_x \* 3, sq\_x \* 3 + 3)

])

exists = row\_values | column\_values | square3x3\_values

*# какие значения остались?*

values = set(range(1, 10)) - exists

variants.append((i, j, values))

**return** variants

**def** solve(sudoku):

*# Если судоку заполнено, это ответ*

**if** all([k **for** row **in** sudoku **for** k **in** row]):

**return** sudoku

*# Иначе посмотрим все варианты*

variants = get\_variants(sudoku)

*# Выберем тот, у которого меньше всего возможностей.*

x, y, values = min(variants, key=**lambda** x: len(x[2]))

*# Попробуем все по очереди*

**for** v **in** values:

*# deepcopy создает полную копию списка с учетом всех вложенностей*

new\_sudoku = deepcopy(sudoku)

new\_sudoku[x][y] = v

*# Если оно решилось, возвратим ответ.*

s = solve(new\_sudoku)

**if** s:

**return** s

s = clock()

pprint(

solve([

[0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 1, 0, 0, 2, 0, 0, 3, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 4, 0, 0, 5, 0, 0, 6, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 7, 0, 0, 8, 0, 0, 9, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0],

]))

**print**('Затраченное время:', clock() - s, 'сек')

А вот ещё один пример рекурсивного решения задачи. Он ещё короче, но в нём есть **питоновские** штуки.

**from** random **import** shuffle

**from** copy **import** deepcopy

**from** pprint **import** pprint

**def** make\_assumptions(sudoku):

**for** i, row **in** enumerate(sudoku):

**for** j, value **in** enumerate(row):

**if** **not** value:

values = set(row) \

| set([sudoku[k][j] **for** k **in** range(9)]) \

| set([sudoku[m][n]   
 **for** m **in** range((i // 3) \* 3,

(i // 3) \* 3 + 3)

**for** n **in** range((j // 3) \* 3,

(j // 3) \* 3 + 3)])

**yield** i, j, list(set(range(1, 10)) - values)

**def** solve(sudoku):

**if** all([k **for** row **in** sudoku **for** k **in** row]):

**return** sudoku

assumptions = list(make\_assumptions(sudoku))

shuffle(assumptions)

x, y, values = min(assumptions, key=**lambda** x: len(x[2]))

**for** v **in** values:

new\_sudoku = deepcopy(sudoku)

new\_sudoku[x][y] = v

s = solve(new\_sudoku)

**if** s:

**return** s

pprint(solve(field))

Два предыдущих примера демонстрируют преимущество рекурсии — написание коротких и легко читаемых программ.

Попробуйте сравнить эти программы с императивным вариантом (без использования рекурсии).

С рекурсией вы ещё встретитесь много раз. Помните, что она — **не панацея**, но позволяет элегантно и эффективно решать широкий круг задач.

А пока — всё, переходите к задачам!

[Справка](https://yandex.ru/support/lyceum-students)

Исключительное право на учебную программу и все сопутствующие ей учебные материалы, доступные в рамках проекта «Яндекс.Лицей», принадлежат АНО ДПО «ШАД». Воспроизведение, копирование, распространение и иное использование программы и материалов допустимо только с предварительного письменного согласия АНО ДПО «ШАД».

© 2018 – 2020  ООО «[Яндекс](https://yandex.ru/)»

Чаты