# 2 Рекурсия

> Чтобы понять рекурсию, необходимо для начала понять рекурсию.
>
> *Аноним*

**Итеративный алгоритм** решает задачи повтором шагов снова и снова, используя, как правило, цикл. Большинство алгоритмов, которые вы до сих пор писали, наверняка были итеративными. **Рекурсия** — метод решения задачи, когда вы решаете более мелкие экземпляры задачи, пока не придете к решению. Рекурсивные алгоритмы опираются на функции, вызывающие сами себя. Любую задачу, которую можно решить с помощью итеративного алгоритма, можно решить и с помощью рекурсивного. Однако иногда рекурсивный алгоритм является более элегантным решением.

Рекурсивный алгоритм пишется внутри функции или метода, которые вызывают сами себя. Код внутри функции меняет входные данные и передает новые, отличные от предыдущих, когда в следующий раз функция вызывает саму себя. По этой причине у функции должна присутствовать **терминальная ветвь** — условие, которое завершает рекурсивный алгоритм, останавливая его навсегда. Каждый раз, когда функция вызывает саму себя, она приближается к терминальной ветви. В конце концов условие терминальной ветви удовлетворяется, то есть задача решена, и функция перестает вызывать саму себя. Алгоритм, следующий этим правилам, удовлетворяет трем законам рекурсии.

- У рекурсивного алгоритма должны быть как минимум одна рекурсивная и одна терминальная ветвь.

- Рекурсивный алгоритм должен менять свое состояние и стремиться к терминальной ветви.
- Рекурсивный алгоритм должен вызывать себя рекурсивно.

Чтобы помочь вам понять принцип работы рекурсивного алгоритма, рассмотрим задачу нахождения факториала числа с помощью рекурсивного и итеративного алгоритмов. **Факториал** числа — это произведение всех положительных целых чисел, меньших или равных числу. Например, факториал числа 5 равен $5 \times 4 \times 3 \times 2 \times 1$.

```
5! = 5 * 4 * 3 * 2 * 1
```

Ниже представлен итеративный алгоритм, который вычисляет факториал числа `n`:

```
def factorial(n):
    the_product = 1
    while n > 0:
        the_product *= n
        n = n - 1
    return the_product
```

Функция `factorial` принимает число `n`, которое вы используете в своих вычислениях.

```
def factorial(n):
```

Внутри функции вы объявляете переменную `the_product` и устанавливаете ей значение `1`. Вы используете `the_product` для отслеживания результата при умножении `n` на предшествующие ему числа, например 5 * 4 * 3 * 2 * 1.

Далее вы используете цикл `while` для итерации в обратном направлении от `n` до 1, отслеживая результат.

```
while n > 0:
    the_product *= n
    n = n - 1
```

В конце цикла `while` вы возвращаете `the_product`, которая содержит факториал `n`.

```
return the_product
```

А вот как написать этот же алгоритм рекурсивно:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Сначала вы объявляете функцию `factorial`, которая принимает число `n` в качестве параметра. Далее следует ваша терминальная ветвь. Функция будет возвращать себя до тех пор, пока `n` не станет равным `0`, после чего она вернет число `1` и перестанет вызывать себя.

```
if n == 0:
    return 1
```

Каждый раз, когда не удовлетворяются условия терминальной ветви, выполняется рекурсивная ветвь, то есть эта строка кода:

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

Как видите, ваш код вызывает функцию `factorial`, которая является самой собой. Если вы впервые видите рекурсивный алгоритм, он может показаться вам странным. Вы можете даже подумать, что такой код не будет работать. Но поверьте, он работает. В данном случае функция `factorial` вызывает саму себя и возвращает результат. Однако она не вызывает саму себя со значением `n`; точнее, она вызывает себя со значением `n` – 1. В конечном итоге `n` будет меньше 1, что удовлетворит условие терминальной ветви:

```
if n == 0:
    return 1
```

Вот весь код, который вам необходимо написать для рекурсивного алгоритма, состоящего всего лишь из четырех строк:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Как же это работает? Каждый раз, когда функция встречает оператор возврата, она помещает его в стек[1]. Стек — тип структуры данных, о котором вы узнаете больше во второй части книги. Это как список в Python, но вы удаляете элементы в том же порядке, в котором их добавляли. Допустим, вы вызываете свою рекурсивную функцию `factorial` следующим образом:

```
factorial(3)
```

Переменная `n` начинается с 3. Функция проверяет условие терминальной ветви. Возвращается значение `False`, поэтому Python выполняет строку кода рекурсивной ветви:

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

---

[1] Так называемый стек вызовов. — *Примеч. науч. ред.*

Python пока не знает результат `n * factorial(n-1)`, поэтому помещает выражение в стек.

```
# Внутренний стек (не выполняйте этот код)

# n = 3
[return n * factorial(n - 1)]
```

Затем функция вызывает саму себя после уменьшения `n` на 1:

```
factorial(2)
```

Функция опять проверяет условие терминальной ветви, и вновь возвращается значение `False`, поэтому Python снова идет по рекурсивной ветви:

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

Python все еще не знает результат `n * factorial(n-1)`, поэтому помещает выражение в стек.

```
# Внутренний стек

# n = 3                       # n = 2
[return n * factorial(n - 1), return n * factorial(n - 1),]
```

Снова ваша функция вызывает саму себя после уменьшения `n` на 1:

```
factorial(1)
```

Python опять не знает результат `n * factorial(n-1)`, поэтому помещает выражение в стек.

```
# Внутренний стек

# n = 3
[return n * factorial(n - 1),
# n = 2
return n * factorial(n - 1),
# n = 1
return n * factorial(n - 1),]
```

Вновь ваша функция вызывает саму себя после уменьшения `n` на 1, но на этот раз `n` равно `0`, а это значит, что условие терминальной ветви удовлетворено и вы возвращаете число `1`.

```
if n == 0:
    return 1
```

Python снова помещает возвращаемое значение в стек, но на этот раз он знает, что возвращает: число `1`. Теперь внутренний стек Python выглядит следующим образом:

```
# Внутренний стек

# n = 3
[return n * factorial(n - 1),
# n = 2
return n * factorial(n - 1),
# n = 1
return n * factorial(n - 1), 1]
```

Поскольку Python знает результат последнего возврата, он может вычислить результат предыдущего возврата и удалить его из стека. Другими словами, Python умножает 1 * `n`, а `n` равно `1`.

```
1 * 1 = 1
```

Теперь внутренний стек Python выглядит так:

```
# Внутренний стек

# n = 3                       # n = 2
[return n * factorial(n - 1), return n * factorial(n - 1), 1]
```

И снова, поскольку Python знает результат последнего возврата, он может вычислить результат предыдущего возврата и удалить его из стека:

```
2 * 1 = 2
```

В этот момент внутренний стек Python выглядит так:

```
# Внутренний стек

# n = 3
[return n * factorial(n - 1), 2]
```

В конечном счете, поскольку Python знает результат последнего возврата, он может вычислить результат предыдущего возврата, удалить этот результат из стека и вернуть ответ.

```
3 * 2 = 6

# Внутренний стек

[return 6]
```

Теперь вы видите, что вычисление факториала числа — прекрасный пример задачи, которую можно решить, находя решения для меньших экземпляров одной и той же задачи. Осознав это и написав рекурсивный алгоритм, вы создадите элегантное решение для вычисления факториала числа.

## Когда использовать рекурсию

Как часто использовать рекурсию в алгоритме, зависит только от вас. Любой алгоритм, написанный с помощью рекурсии, можно также написать с помощью итерации. Основное преимущество рекурсии — ее изящество. Как вы видели ранее, итеративное решение для вычисления факториалов заняло шесть строк кода, в то время как для рекурсивного решения потребовалось лишь четыре строки. Недостаток рекурсивных алгоритмов в том, что обычно они занимают больше памяти, так как им нужно хранить данные во внутреннем стеке Python. Рекурсивные функции по сравнению с итеративными может быть труднее считывать и отлаживать, поскольку отследить, что происходит в рекурсивном алгоритме, сложнее.

Будете ли вы использовать рекурсию для решения задачи, зависит от специфики ситуации — например, от того, насколько важен задействованный объем памяти по сравнению с изяществом применения рекурсивного алгоритма, а не итерационного. Далее в книге вы встретите больше примеров, когда рекурсия предлагает более красивое решение, как в случае обхода двоичного дерева.

## Словарь терминов

**Итеративный алгоритм** — алгоритм, решающий задачи повтором шагов снова и снова, обычно с помощью цикла.

**Рекурсия** — метод решения задачи, при котором результат зависит от решений меньших экземпляров одной и той же задачи.

**Терминальная ветвь** — условие, при котором рекурсивный алгоритм завершается, чтобы он не продолжался вечно.

**Факториал** — произведение всех положительных целых чисел, меньших или равных числу.

## Практикум

1. Выведите числа от 1 до 10 рекурсивно.