Урок 10.  
Рекурсия.

короткая линия

# План урока

1. Рекурсия
2. Рекурсивные алгоритмы
3. Эффективность рекурсии

# Рекурсия

В прошлом уроке мы говорили, что функции используются, чтобы вынести независимый кусок кода для его переиспользования в разных местах программы. Кроме того, в функцию можно помещать другие, внутренние функции, однако это применяется реже. Например:

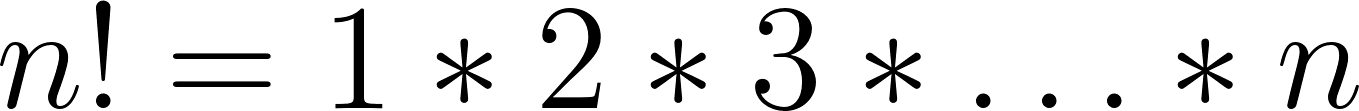
|  |
| --- |
| **def** temp\_convert(temp):  **"""  Функция перевода температур  """**   **def** get\_convert\_type():  **"""  Возвращаем шкалу, в которую переводим   """**  **return** **"F"** **if** temp[-1] == **"C"** **else** **"C"**   **def** format\_temp():  **"""  Функция, форматирующая вывод  """**  **if** result **is** **None**:  **return** **"Неверный формат"**  **return** **"Температура в {}: {}\n"** \  **"Температура в {}: {}"**.format(temp[-1], temp[:-1],  get\_convert\_type(), result)   **if** temp.endswith(**"F"**):  result = (int(temp[:-1]) - 32) / 1.8  **elif** temp.endswith(**"C"**):  result = int(temp[:-1]) \* 1.8 + 32  **else**:  result = **None**   **return** format\_temp() |

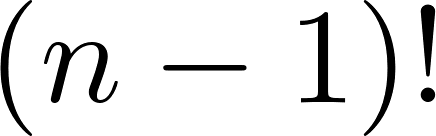
В функции перевода температур мы разбили задачу на несколько подзадач (получение символа шкалы, в которую мы переводим, а также форматирование результата) и оформили для каждой собственные функции.

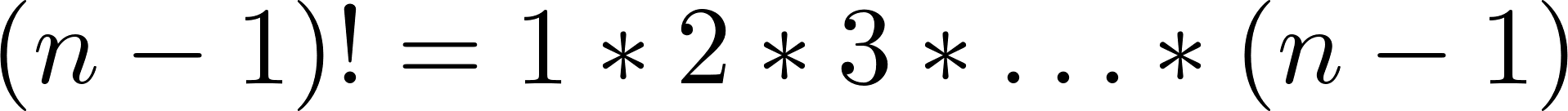
Однако это еще не все: функции могут вызывать не только свои внутренние функции, но и самих себя! Проверьте, что выведут эти примеры кода (сначала подумайте, потом проверьте свой ответ в IDE):

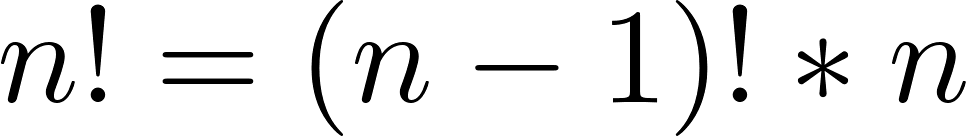
|  |
| --- |
| 1. **def** print\_number():  print(42)  print\_number()  print\_number() 2. **def** drink():  print(**'Рецепт коктейля "Рекурсивный": '**  **'20% спирта, 30% воды, 50% коктейля "Рекурсивный"...'**)  drink()  drink() |

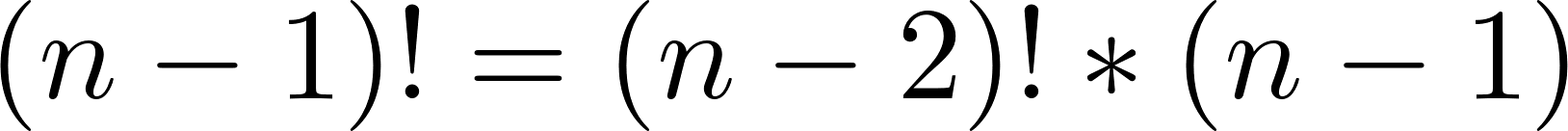
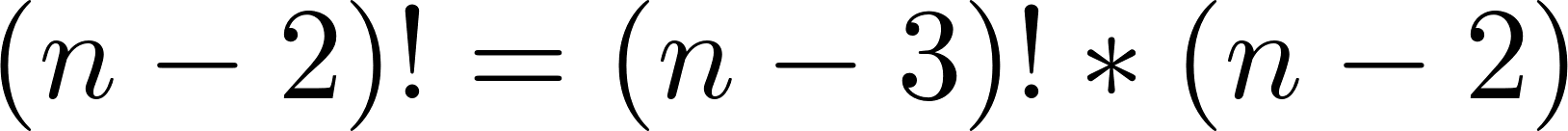
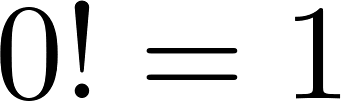
Рекурсия – это и есть вызов функцией самой себя. Рекурсию сложно понять и прочувствовать, но она позволяет записывать решение многих задач гораздо более ёмко и нередко экономит время программиста. Одним из лучших примеров применения рекурсии является задача вычисления факториала.

Известно, что [](https://www.codecogs.com/eqnedit.php?latex=n!%20%3D%201%20*%202%20*%203%20*%20%5Cldots%20*%20n%0)

Теперь посмотрим на [](https://www.codecogs.com/eqnedit.php?latex=(n%20-%201)!%0):

[](https://www.codecogs.com/eqnedit.php?latex=(n%20-%201)!%20%3D%201%20*%202%20*%203%20*%20%5Cldots%20*%20(n%20-%201)%0)

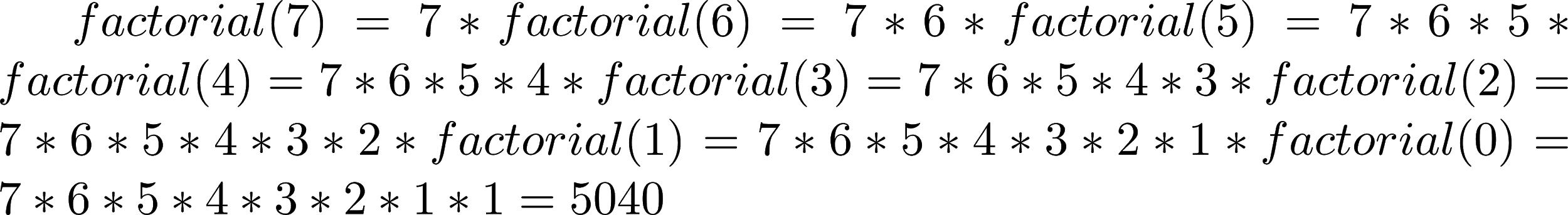
Значит, [](https://www.codecogs.com/eqnedit.php?latex=n!%20%3D%20(n%20-%201)!%20*%20n%0)

В свою очередь [](https://www.codecogs.com/eqnedit.php?latex=(n%20-%201)!%20%3D%20(n%20-%202)!%20*%20(n%20-%201)%0), а [](https://www.codecogs.com/eqnedit.php?latex=(n%20-%202)!%20%3D%20(n%20-%203)!%20*%20(n%20-%202)%0) и так далее, до тех пор, пока мы не доберемся до [](https://www.codecogs.com/eqnedit.php?latex=0!%20%3D%201%0).

Теперь реализуем это в виде рекурсивного алгоритма:

|  |
| --- |
| **def** factorial(n):  **if** n == 0:  **return** 1  **else**:  **return** n \* factorial(n - 1)  print(factorial(7)) |

Покажем цепочку вызовов функции factorial():

[](https://www.codecogs.com/eqnedit.php?latex=factorial(7)%20%3D%207%20*%20factorial(6)%20%3D%207%20*%206%20*%20factorial(5)%20%3D%207%20*%206%20*%205%20*%20factorial(4)%20%3D%207%20*%206%20*%205%20*%204%20*%20factorial(3)%20%3D%207%20*%206%20*%205%20*%204%20*%203%20*%20factorial(2)%20%3D%207%20*%206%20*%205%20*%204%20*%203%20*%202%20*%20factorial(1)%20%3D%207%20*%206%20*%205%20*%204%20*%203%20*%202%20*%201%20*%20factorial(0)%20%3D%207%20*%206%20*%205%20*%204%20*%203%20*%202%20*%201%20*%201%20%3D%205040%0)

Функция выведет верный ответ **5040**. Вы можете протестировать рекурсивный алгоритм и сравнить его вывод с итеративным (с использованием циклов) алгоритмом, который Вы писали в предыдущих уроках.

Чтобы функция не вызывала сама себя бесконечно, как это случилось с print\_number(), каждый рекурсивный алгоритм должен содержать две части: базовый случай (иногда говорят просто “база рекурсии”) и рекурсивный случай. Дойдя до базового случая, функция должна остановиться, так как в дальнейших вызовах нет смысла. Например, функция факториал определена только на множестве неотрицательных чисел, поэтому не нужно вызывать ее на числах, меньших нуля. По этой причине, прежде чем реализовывать рекурсивный алгоритм, необходимо продумать, каким будет базовый случай.

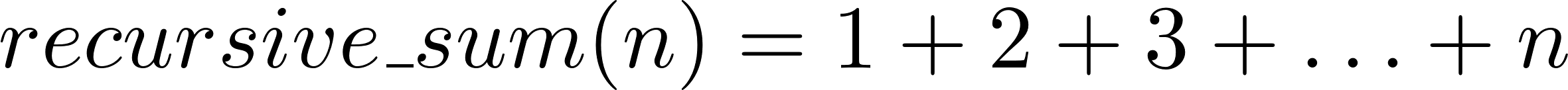
**Задача “Функция Аккермана”**

# Рекурсивные алгоритмы

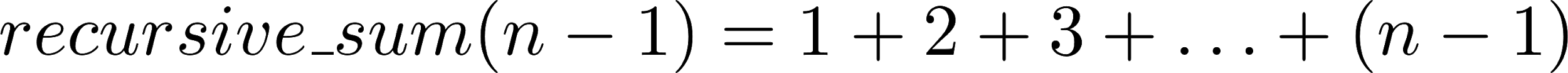
Чтобы показать, что рекурсии находится достаточно широкое применение, реализуем некоторые простенькие алгоритмы рекурсивно. При этом стоит помнить, что использовать рекурсию следует лишь в том случае, если она заметно упрощает модель решения и экономит время. Для каждого алгоритма мы рассмотрим базовый и рекурсивный случаи.

**Сумма натуральных чисел до n**

Необходимо вывести сумму всех чисел от 1 до n. Назовем нашу функцию recursive\_sum(n). Тогда

[](https://www.codecogs.com/eqnedit.php?latex=recursive%5C_sum(n)%20%3D%201%20%2B%202%20%2B%203%20%2B%20%5Cldots%20%2B%20n%0)

В то же время

[](https://www.codecogs.com/eqnedit.php?latex=recursive%5C_sum(n%20-%201)%20%3D%201%20%2B%202%20%2B%203%20%2B%20%5Cldots%20%2B%20(n%20-%201)%0)

Из этого следует, что

[](https://www.codecogs.com/eqnedit.php?latex=recursive%5C_sum(n)%20%3D%20recursive%5C_sum(n%20-%201)%20%2B%20n%0)

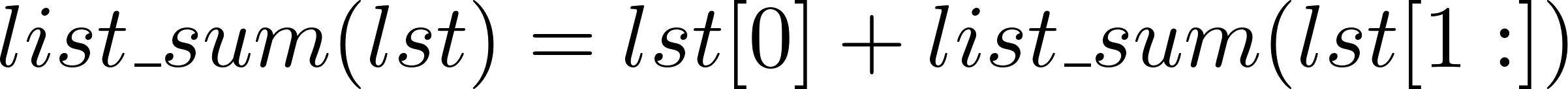
Таким образом, мы рассмотрели рекурсивный случай. Очевидно, что для recursive\_sum(n-1) выполняется тот же переход и вызов этой функции раскладывается на сумму recursive\_sum(n - 2) + (n - 1). Так как сумма всегда начинается с единицы, базовым случаем будет вызов функции от единицы. Напишем этот алгоритм на Python:

|  |
| --- |
| **def** recursive\_sum(n):  **if** n == 1:  **return** 1  **else**:  **return** recursive\_sum(n - 1) + n  print(recursive\_sum(4)) |

Для полного понимания Вы можете расписать цепочку вызовов в том же виде, в каком мы это делали для функции factorial(n).

**Сумма всех чисел списка**

Задача похожа на предыдущую, но здесь мы будем оперировать списками. Рекурсивный случай можно описать так: сумма чисел списка равна сумме первого элемента и суммы остальных.

[](https://www.codecogs.com/eqnedit.php?latex=list%5C_sum(lst)%20%3D%20lst%5B0%5D%20%2B%20list%5C_sum(lst%5B1%3A%5D)%0)

Останавливать вызовы функции мы будем тогда, когда очередной lst[1:] окажется пустым списком, то есть когда элементы закончатся. Это будет нашим базовым случаем.

|  |
| --- |
| **def** list\_sum(lst):  **if** len(lst) == 0:  **return** 0  **else**:  **return** lst[0] + list\_sum(lst[1:])  print(list\_sum([1, 2, 3, 4])) |

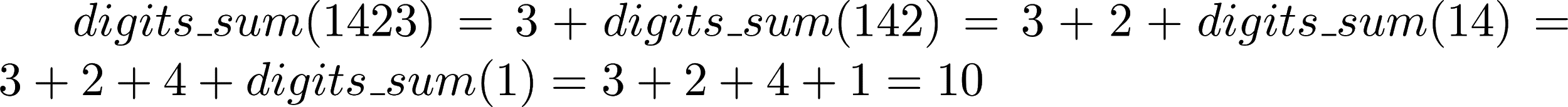
**Проверка на палиндром**

Строка s является палиндромом, если s[0] = s[-1], а также если s[1:-1] – палиндром. Строка s[1:-1] является палиндромом, если s[1] = s[-2] и если s[2:-2] – палиндром и т. д. Базовый случай – строка, состоящая из одного символа, либо пустая строка. Такая строка в любом случае является палиндромом.

|  |
| --- |
| **def** palindrome(s):  **if** len(s) < 2:  **return** **True**  **else**:  **return** s[0] == s[-1] **and** palindrome(s[1:-1])  print(palindrome(**"радар"**)) |

**Сумма цифр числа**

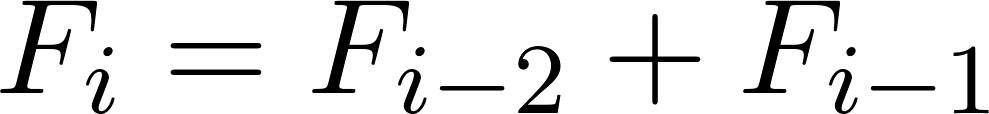
Рассмотрим произвольное натуральное число n. Для этого числа мы можем сделать такой вывод: сумма его цифр равна сумме последней цифры числа и суммы цифр числа n // 10. Например,

[](https://www.codecogs.com/eqnedit.php?latex=digits%5C_sum(1423)%20%3D%203%20%2B%20digits%5C_sum(142)%20%3D%203%20%2B%202%20%2B%20digits%5C_sum(14)%20%3D%203%20%2B%202%20%2B%204%20%2B%20digits%5C_sum(1)%20%3D%203%20%2B%202%20%2B%204%20%2B%201%20%3D%2010%0)

Из этого же примера понятно, что считать базовым случаем: для числа, состоящего из одной цифры (то есть меньшего 10), дальнейшие вызовы функции не нужны. Оформим алгоритм:

|  |
| --- |
| **def** digits\_sum(n):  **if** n < 10:  **return** n  **else**:  **return** n % 10 + digits\_sum(n // 10)  print(digits\_sum(1423)) |

**Числа Фибоначчи**

Преимущество рекурсии еще и в том, что мы можем реализовывать вещи согласно их определению. Зная, что каждое число Фибоначчи [](https://www.codecogs.com/eqnedit.php?latex=F_i%20%3D%20F_%7Bi%20-%202%7D%20%2B%20F_%7Bi%20-%201%7D%0), с помощью рекурсии мы можем это записать в том же виде. Базовым случаем нашего алгоритма будет n <= 2, так как для этих n числа Фибоначчи равны 1.

**def** fib(n):  
 **if** n <= 2:  
 **return** 1  
 **else**:  
 **return** fib(n - 2) + fib(n - 1)  
  
print(fib(8))

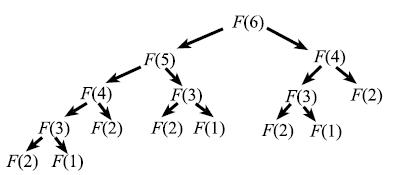
**Задача “Цифры числа”**

**Задача “Длина объекта”**

**Задача “Степень двойки”**

# Эффективность рекурсии

У рекурсии есть существенный недостаток: она требует намного больше времени и памяти, чем итеративные алгоритмы. Рассмотрим дерево вызовов функции fib(n) при n = 6:



В отличие от предыдущих алгоритмов, этот разделяет каждый вызов функции не на один рекурсивный вызов, а на два. Так, fib(6) распадается на fib(5) и fib(4). Именно поэтому цепочка вызовов выглядит как дерево. Это дерево позволяет нам увидеть очевидную проблему: рекурсия делает слишком много лишней работы: например, fib(3) вычисляется три раза, fib(4) – два раза. При больших n лишней работы становится настолько много, что функция не может получить результат за приемлемое время. Попробуйте вызвать fib(42), и, скорее всего, через несколько минут Вы устанете ждать. В то же время решение через цикл for выведет результат мгновенно.

В этом случае может возникнуть вопрос: зачем же использовать рекурсию, если она настолько неэффективна? Во-первых, на рекурсии основано огромное количество алгоритмов категории “разделяй и властвуй”, которые очень трудно представить в итеративном виде (быстрая сортировка, обход графов). Во-вторых, существуют способы ускорить рекурсивные алгоритмы: например, можно запоминать уже вычисленные числа Фибоначчи и прекращать вызов функции, если результат для этого числа уже известен. Попробуйте реализовать такой механизм “кэширования” самостоятельно.

**Задача “Быстрое возведение в степень”**

**Задача “Кэширование”**

**Задача “Перебор подмножеств”**

**Задача “Перебор перестановок”**

**Задача “Ханойская башня”**

**Материал для самостоятельного изучения:** если Вы хотите лучше разобраться в рекурсии и алгоритмах в целом, рекомендуем книгу “Грокаем алгоритмы” (Бхаргава А.). Она написана простым языком, снабжена иллюстрациями и рассчитана на начинающих программистов.