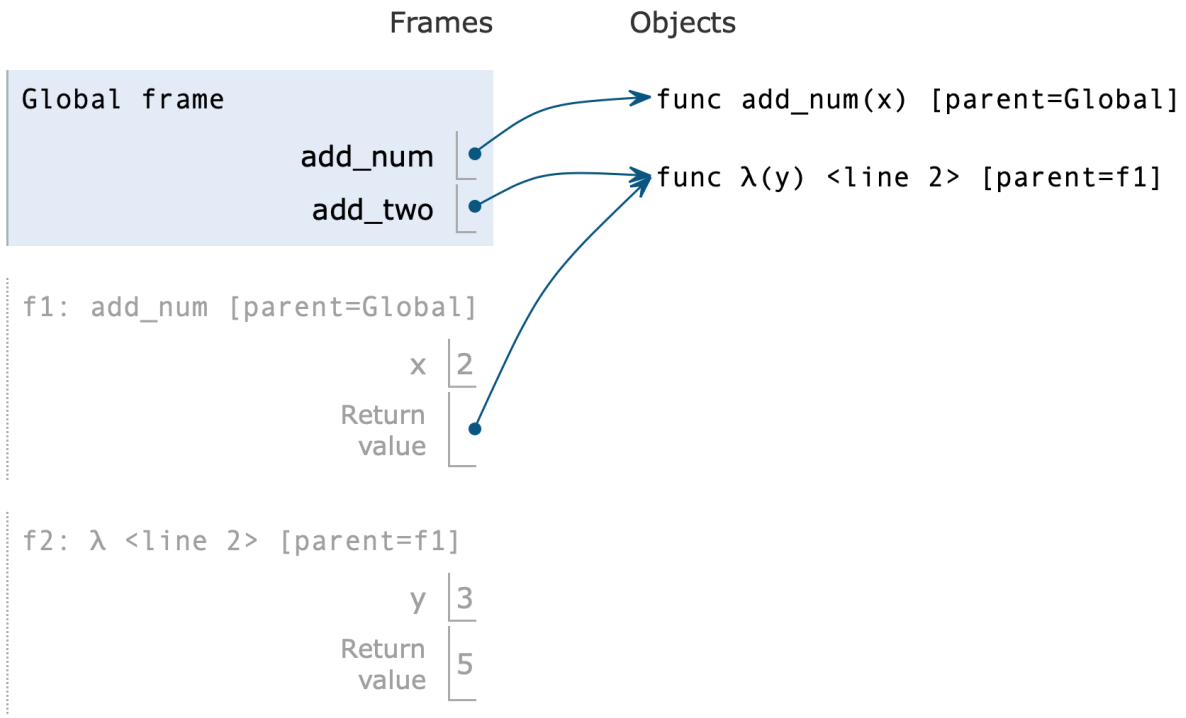
Практика 2. Функции высшего порядка и рекурсия

материалы преподавателя

Функции высшего порядка

Тебе уже известно, что на **диаграммах окружения** отображаются все переменные и их текущие значения. Ошибочно думать, что значения могут быть только числами или строками текста. Они могут быть *функциями*. Диаграммы окружения позволяют моделировать сложные программы с функциями высшего порядка.

```
def add_num(x):
    return lambda y: x + y
add_two = add_num(2)
add_two(3)
```



На диаграммах окружения функции, заданные с помощью лямбда-выражений, отображаются также, как обычные функции, за исключением того, что *внутреннее* имя лямбда-функции отсутствует, и вместо него записывается номер строки в файле с определением этой лямбда-функции.

В коде примера выражение `lambda y: x + y` может быть прочитано так: функция одного аргумента `y`, возвращающая `x+y`.

Значение лямбда-выражения — функция. Однако эта функция не связана с именем. Такие функции называют *анонимными*. Тело лямбда-функции, так же как и тело обычной функции, не выполняется до момента её вызова.

```
>>> what = lambda x : x + 5
>>> what
<function <lambda> at 0x031337>
```

В отличии от `def`-инструкций, лямбда-выражения могут непосредственно использоваться на месте оператора или операнда в вызывающем выражении.

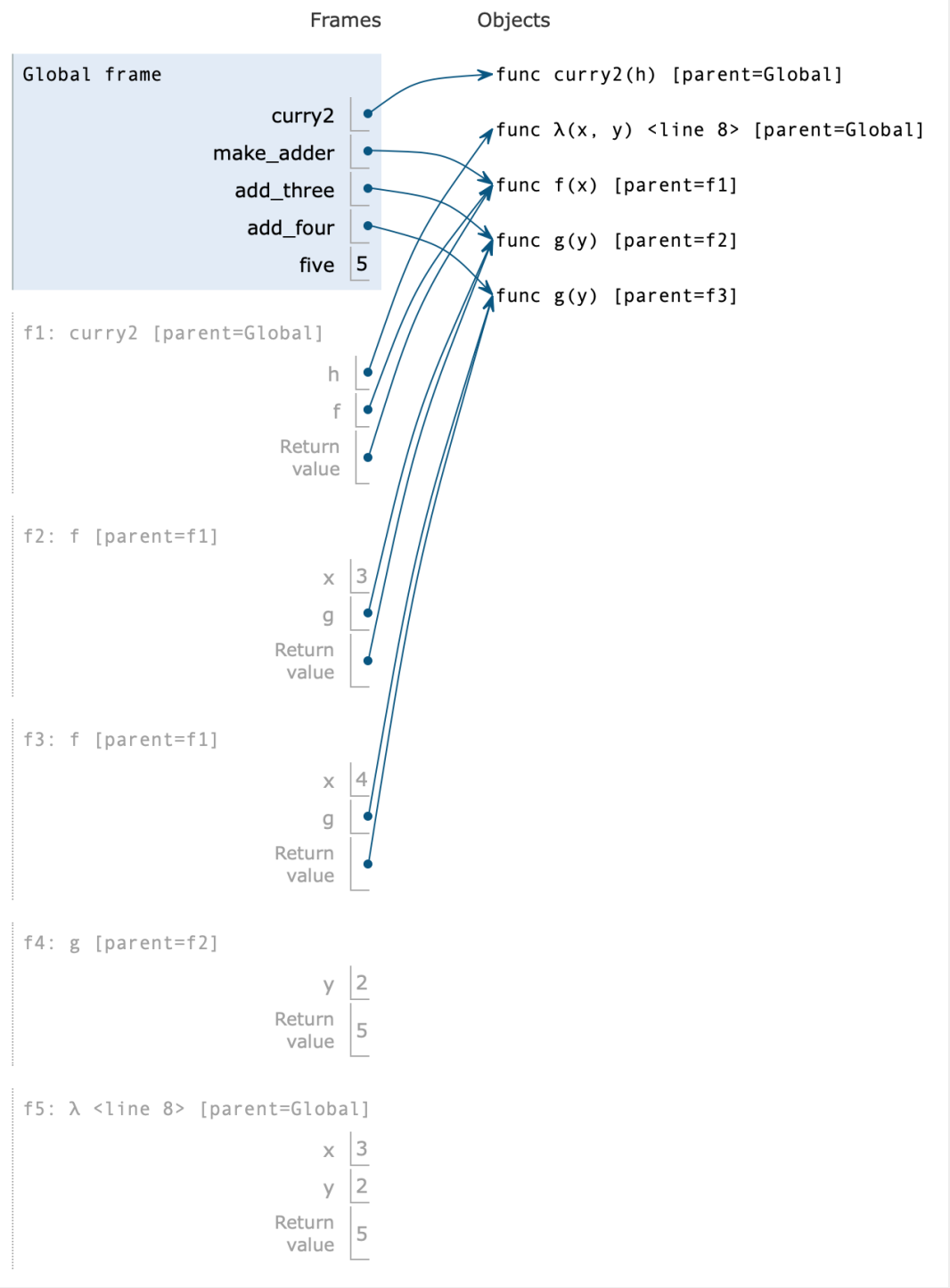
```
>>> (lambda y: y + 5)(4)
9
>>> (lambda f, x: f(x))(lambda y: y + 1, 10)
11
```

Вопрос 1

Нарисуй диаграмму окружения, которая получится после выполнения следующего кода:

```
def curry2(h):
    def f(x):
        def g(y):
            return h(x,y)
        return g
    return f

make_adder = curry2(lambda x, y: x + y)
add_three = make_adder(3)
add_four = make_adder(4)
five = add_three(2)
```

Ответ



Вопрос 2

Запиши функцию `curry2` из предыдущего вопроса в виде лямбда-выражения.

Ответ

```
curry2 = lambda h: lambda x: lambda y: h(x, y)
```

Вопрос 3

Напиши функцию `and_add`, которая принимает функцию одного аргумента `f` и число `n`. Результатом должна быть функцию одного аргумента, которая будет делать то же самое, что и функция `f`, но с поправкой на `n` (прибавляется к результату).

```
def and_add(f, n):
    """Создаёт функцию, принимающую аргумент x и возвращающую f(x) + n.
   >>> def square(x):
   ...     return x * x
   >>> new_square = and_add(square, 3)
   >>> new_square(4)   # 4 * 4 + 3
   19
   """
```

Ответ

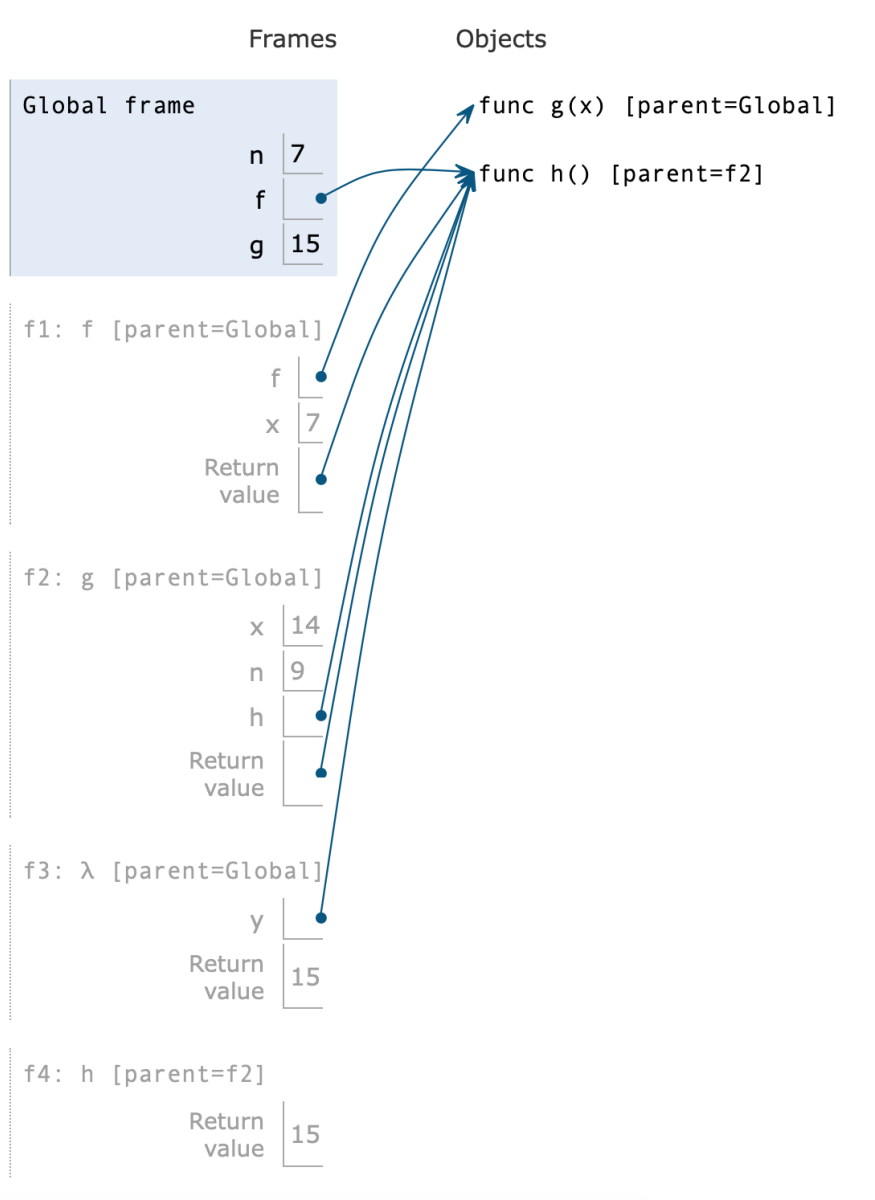
```
    def g(x):
        return f(x) + n
    return g
```

Вопрос 4

Нарисуй диаграмму окружения, которая получится после выполнения следующего кода.

```
n = 7

def f(x):
    n = 8
    return x + 1

def g(x):
    n = 9
    def h():
        return x + 1
    return h

def f(f, x):
    return f(x + n)

f = f(g,n)
g = (lambda y: y())(f)
```

Ответ



Вопрос 5

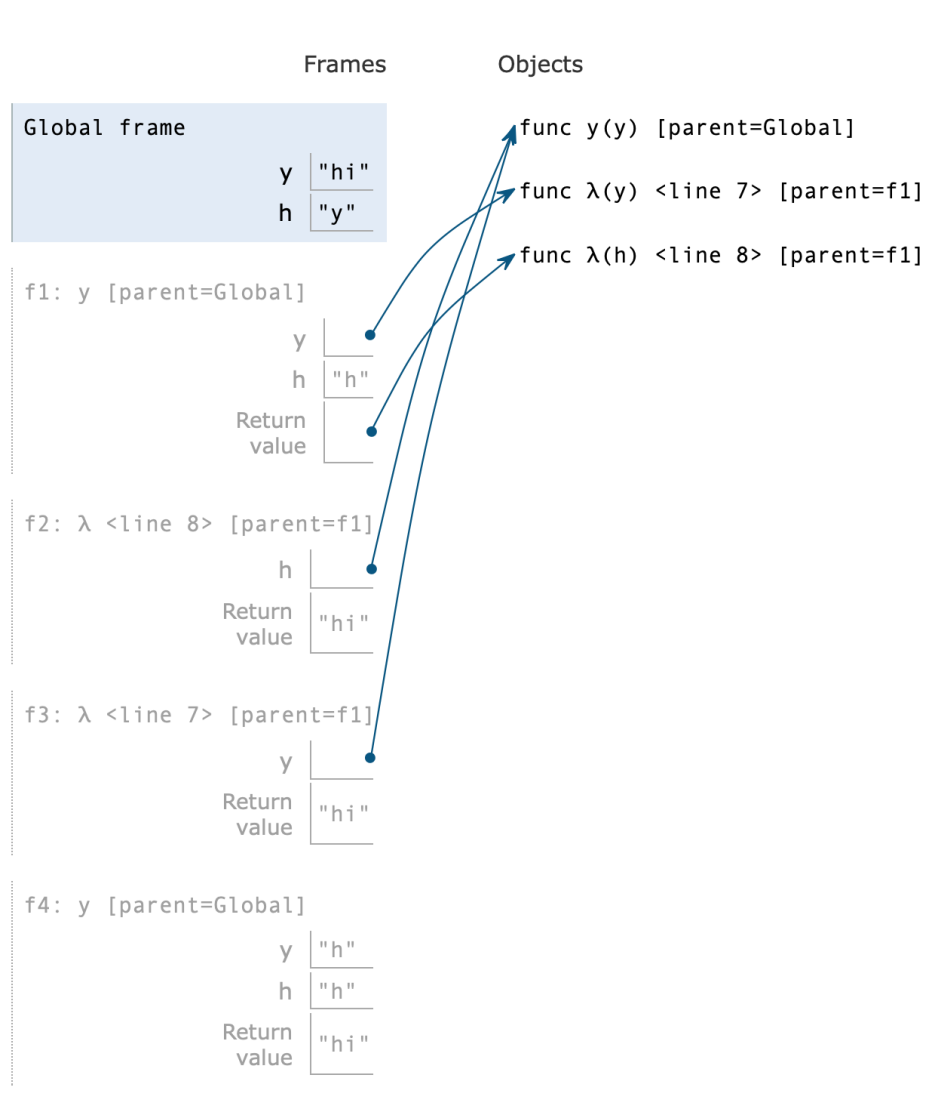
Задание экстремальной сложности, превосходящее даже вопросы из грядущей контрольной работы. Попробуй решить, будет весело.

Нарисуй диаграмму окружения, которая получится после выполнения следующего кода.

Со строками оператор `+` работает как оператор *конкатенации* (присоединения) — к первому аргументу будет присоединён второй. Например, результатом выражения `'Riff' + 'Raff'` станет новая объединённая строка `RiffRaff`.

```
y = "y"
h = y
def y(y):
    h = "h"
    if y == h:
        return y + "i"
    y = lambda y: y(h)
    return lambda h: y(h)
y = y(y)(y)
```

Ответ



Рекурсия

Рекурсивная функция — это функция, которая определена в терминах собственных вызовов. Хорошим примером рекурсивной функции является функция `factorial(n)`. Когда `n=0` (или `n=1`), функция просто возвращает `1`, — это *базовый случай* - механизм не позволяющий рекурсивной функции бесконечно вызывать себя. Теперь можно вычислить `factorial(2)` в терминах `factorial(1)`, `factorial(3)` в терминах `factorial(2)`, `factorial(4)` в терминах `factorial(3)` и так далее.

Для определения рекурсивной функции нужно совершить **три** шага:

1. **Отыскать базовый случай**. Обычно наипростейший аргумент для разрабатываемой функции и есть базовый случай. Например, `factorial(0)` равен `1` по определению. Можно мыслить базовый случай как условие останова рекурсии.
2. **Сделай рекурсивный вызов с более простым аргументом**. Нужно сформулировать решение в более простых терминах задачи, считая, что более простой случай задачи уже решён. Например, считать, что при вычислении `factorial(n)` уже известен результат `factorial(n-1)`.
3. **Использовать рекурсивный вызов для решения задачи**. `factorial(n) = n*factorial(n-1)`

для понимания рекурсии нужно отделить *внутреннюю корректность* от *конечности исполнения*. Рекурсивная функция внутренне корректна, если каждый из рекурсивных вызовов сам по себе корректен и возвращает правильные результаты. Например, если из функции `factorial(n)` убрать базовый случай, то она будет внутренне корректна, но при этом никогда не остановится.

Для понимания рекурсии нужно отделить *внутреннюю корректность* от *конечности исполнения*.

Рекурсивная функция внутренне корректна, если каждый из рекурсивных вызовов сам по себе корректен и возвращает правильные результаты. Например, если из функции `factorial(n)` убрать базовый случай, то она будет внутренне корректна, но при этом никогда не остановится.

Рекурсивная функция корректна тогда и только тогда, когда она и внутренне корректна, и остановима. Хотя эти свойства можно рассматривать отдельно.

Вопрос 6

Напиши функцию `multiply(m, n)`. Её поведение — перемножение аргументов `m` и `n`. Считай, что `m` и `n` положительные целые. **Используй рекурсию**, а не `mul` или `*`!

```
5*3 = 5 + 5*2 = 5 + 5 + 5*1
```

В чём состоит базовый случай этой рекурсивной функции?

Ответ

```
Если один из аргументов равен единице, то нужно вернуть второй.
```

К чему приведёт вызов `multiply(m - 1, n)` в рекурсивной части? А вызов `multiply(m, n - 1)`? Какой лучше?

Ответ

```
Такие вызовы приведут к вычислению результата меньше необходимого на n или m
соответственно. Они равнозначны. Достаточно использовать один из них.
```

```
def multiply(m, n):
    """
    >>> multiply(5, 3)
    15
    """
```

Ответ

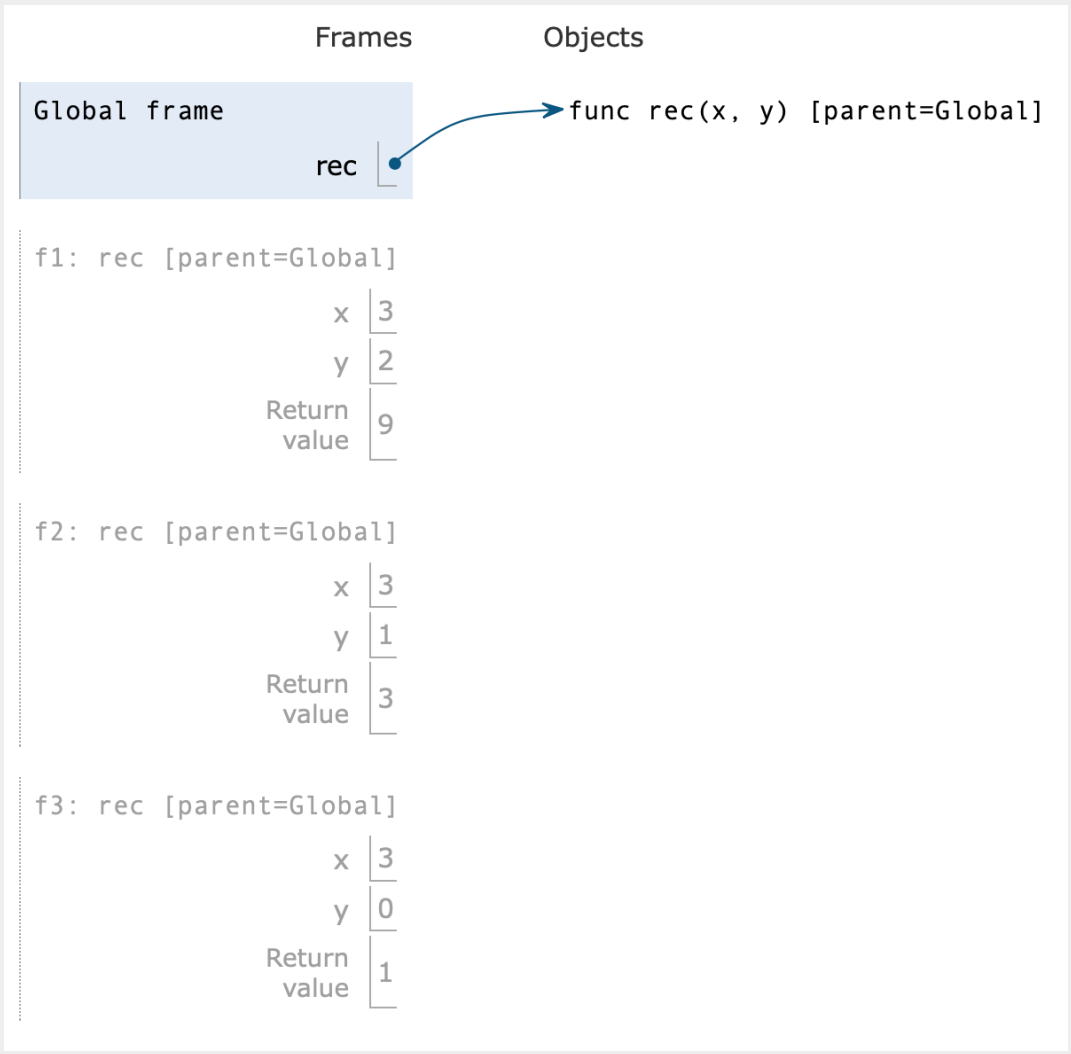
```
    if n == 1:
        return m
    else:
        return m + multiply(m, n - 1)
```

Вопрос 7

Нарисуй диаграмму окружения для следующего кода:

```
def rec(x, y):
    if y > 0:
        return x * rec(x, y - 1)
    return 1
rec(3, 2)
```

Ответ



Кстати, а что вычисляет эта функция?

Ответ

```
x ** y — возведение x в степень y
```

Вопрос 8

Перед тобой лестничный пролёт из `n` ступеней. За один шаг ты можешь преодолеть одну или две ступени. Сколькими способами можно пройти этот пролёт? Напиши функцию `count_stair_ways`, которая находит ответ. Считай `n` положительным.

Но сперва подумай о базовом случае. Каков простейший аргумент `count_stair_ways`?

Ответ

```
Если ступень в пролёте одна, то пройти её можно единственным способом. Если ступеней
две, то и способа два — две-за-раз, два-раза-по-одной.
```

Что означает `count_stair_ways(n - 1)` и `count_stair_ways(n - 2)`?

Ответ

```
- count_stair_ways(n - 1) означает количество способов пройти первые n-1 ступени;
- count_stair_ways(n - 2) означает количество способов пройти первые n-2 ступени;
- базовый случай должен учитывать варианты: осталась одна ступень, осталось две
ступени.
```

Итак, приступим-с.

```
def count_stair_ways(n):
```

Ответ

```
    if n == 1:
        return 1
    elif n == 2:
        return 2
    return count_stair_ways(n-1) + count_stair_ways(n-2)
```

Вопрос 9

Теперь представь, что ты мутант: ноги стали длинными-длинными. Короче можно проходить до `k` ступеней за раз.

Напиши функцию `count_k`, которая подсчитывает количество способов пройти пролёт в этих условиях. Считай `n` и `k` положительными целыми.

```
def count_k(n, k):
    """
    >>> count_k(3, 3) # 3, 2 + 1, 1 + 2, 1 + 1 + 1
    4
    >>> count_k(4, 4)
    8
    >>> count_k(10, 3)
    274
    >>> count_k(300, 1) # по ступеньке за шаг
    1
    """
```

Ответ

```
    if n == 0:
        return 1
    elif n < 0:
        return 0
    else:
        total = 0
        i=1
        while i <= k:
            total += count_k(n - i, k)
            i += 1
        return total
```