Рекурсия

Сегодня мы обсудим одну из самых важных концепций в программировании – рекурсию. **Рекурсивной функцией** (или рекурсией) называется функция, которая вызывает сама себя. Сегодня мы будем говорить про простые рекурсии – функция будет вызывать себя без посредников.

1 Числа Фибоначчи

Рассмотрим использование рекурсии на примере следующей задачи:

Задача

Дано число `n`. Требуется вывести `n`-ое число Фибоначчи.

Числа Фибоначчи определятся так: нулевое и первое числа Фибоначчи равны единице ($F_0 = F_1 = 1$), а каждое следующее число равно сумме двух предыдущих ($F_{n+2} = F_{n+1} + F_n$). Например, первые 10 элементов этой последовательности равны $1, 1, 2, 3, 5, 8, 13, 21, 34, 55$....

Создадим рекурсивную функцию, которая будет принимать в качестве аргумента один аргумент – номер числа Фибоначчи, которое необходимо найти. Для того, чтобы определить `n`-ое число, нам достаточно узнать `n-1`-ое и `n-2`-ое. Давайте предположим, что наша функция `fib` умеет вычислять числа Фибоначчи, и вызовем её для аргументов `n-1` и `n-2`. Сумма результатов вызовов и будет `n`-ым числом Фибоначчи:

```
def fib(n):
    return fib(n - 1) + fib(n - 2)

print(fib(3))
```

Однако если мы запустим нашу функцию для вычисления, например, третьего числа Фибоначчи, то мы получим ошибку *RecursionError: maximum recursion depth exceeded*. Дело в том, что поиск третьего числа запустит `fib(2)` и `fib(1)`, то есть поиск второго и первого чисел Фибоначчи, затем `fib(2)` запустит поиск `1`-го и `0`-го, затем поиск `1`-го запустит поиск `0`-го и `-1`-го. Как вы понимаете, поиск `-1`-го числа Фибоначчи – уже неправильное поведение программы. Более того, он влечет поиск `-2`-го и `-3`-го, и так до бесконечности. В реальности, Python сам останавливает этот бесконечный процесс, потому что в языке есть ограничение на глубину

рекурсии – количество вызовов рекурсивной функции, приведшей к соответствующему запуску функции.

Чтобы не вызывла себя рекурсивно до бесконечности, у каждой рекурсии должна быть **точка выхода** – некоторое условие, для которого рекурсивная функция не совершает новых рекурсивных вызовов, а сразу сообщает ответ. В задаче посика чисел Фибоначчи, из определения мы точно знаем, что нулевое и первое числа Фибоначчи равны единице. Давайте допишем условие, и увидим, что теперь число Фибоначчи вычисляется верно:

```
def fib(n):
    if n == 0 or n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
print(fib(3))
```

```
3
```

Посмотрим, как происходят рекурсивные вызовы. Поиск третьего числа Фибоначчи запускает поиск второго и первого. Сначала происходит запуск поиск второго числа Фибоначчи. Он, в свою очередь, вызывает `fib(1)` и `fib(0)`. Каждый из этих запусков заканчивается на этапе обработки точки выхода, поэтому не создает новых вызовов рекурсии. Значения `fib(1)` и `fib(0)` равны единице, поэтому `fib(2)` возвращает `fib(1) + fib(0) = 2`. Вернемся на уровень запуска `fib(3)`: запуск `fib(2)` завершился, запускается `fib(1)`. Он завершается сразу, поскольку соответствует условию выхода, возвращается значение `1`. Все запуски из `fib(3)` завершились, и возвращается значение `fib(2) + fib(1) = 3`.

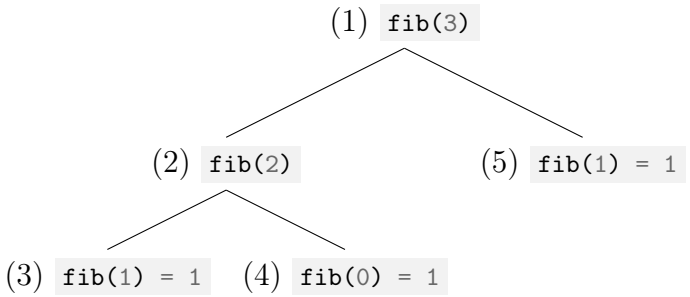


Диаграмма рекурсивных запусков при вызове `fib(3)`

2 Проверка на степень двойки

Рассмотрим еще один пример:

Задача

Дано целое положительное число `n`. Требуется определить, является ли данное число степенью двойки (напомним, что положительное число является степенью двойки, если оно равно произведению некоторого количества двоек).

Итак, какие условия необходимы, чтобы это определить? Конечно, наше число должно быть четным или равным единице. Если число нечетное, то это уже точно не степень двойки, кроме случая с единицей. Это и будет наша точка выхода из условия: если число `n` нечетное, то оно является степенью двойки только в случае `n == 1`, а во всех остальных случаях это уже точно не степень двойки.

Что делать, если число четное? Если число `n` является степенью двойки, то число `n//2` тоже должно являться степенью двойки! Давайте запустим проверку для числа `n//2` и вернем результат этой проверки в качестве результата проверки числа `n`:

```
def is_power_of_2(n):
    if n % 2 == 1:                  # точка выхода из рекурсии
        if n == 1:
            return True
        return False
    return is_power_of_2(n // 2)    # рекурсивный запуск

print(is_power_of_2(16))
print(is_power_of_2(24))
```

```
True
False
```

2.1 Нахождение степени

Несложно модифицировать нашу программу, чтобы она не только определяла, является ли число степенью двойки, но и определяла эту степень в случае положительного результата. Пусть функция возвращает число `k` такое, что `2**k == n`, и `-1`, если такого `k` не существует (если число не является степенью двойки). Начнем модификацию с точки выхода. Если `n` равно единице, то мы должны вернуть `0`, так как `2**0 == 1`. Если `n` нечетное и не единица, вернем `-1` (число точно не является степенью двойки). Теперь для основного случая и рекурсивного вызова нужно разобрать два случая. Если при рекурсивном запуске от `n // 2` мы получили `-1`, то нужно вернуть `-1`, ведь запуск сообщил, что число не является степенью двойки. А иначе, если вернулось `p`, то `n // 2 == 2**p`, а значит `n == 2**(p + 1)` – возвращаем число `p + 1`:

```
def is_power_of_2(n):
    if n % 2 == 1:
        if n == 1:                # если n == 1, возвращаем степень 0
            return 0
        return -1                 # если n нечетное и не 1, возвращаем -1

    p = is_power_of_2(n // 2)     # находим степень числа n // 2
    if p == -1:                   # если n // 2 не степень двойки, возвращаем -1
        return -1
    return p + 1                  # если n // 2 == 2**p, возвращаем p+1

print(is_power_of_2(16))
print(is_power_of_2(24))
```

```
4
-1
```

Задача

Дано число `n`. Требуется найти сумму цифр этого числа, не используя в решении циклы.

Эта задача также легко решается при помощи рекурсии. Создадим функцию, которая будет считать сумму цифр принимаемого в качестве аргумента числа. Сначала разберемся с точкой выхода. Если наше число уже состоит из одной цифры, то очевидно, сумма его цифр равна этой цифре – вот и точка выхода. Иначе, мы можем отделить последнюю цифру, посчитать сумму цифр числа без последней цифры при помощи рекурсии, и добавить к этой сумме последнюю цифру отдельно. Получается, нам нужно сделать рекурсивный вызов для числа `n // 10` (`n` без последней цифры), и прибавить к результату этого вызова последнюю цифру, то есть остаток от деления числа `n` на `10`:

```
def digit_sum(n):
    if n < 10:
      return n
    return digit_sum(n // 10) + n % 10

print(digit_sum(165346257))
```

```
39
```

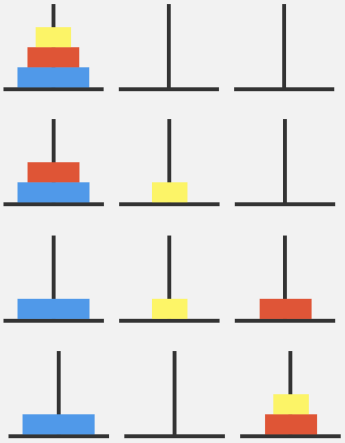
3 Ханойские башни

Предыдущие задачи несложно было решить и без использования рекурсии с помощью циклов. Теперь давайте посмотрим на задачу, решение которое с помощью рекурсии проще, чем решение без неё. Это довольно классическая задача на использование рекурсии.

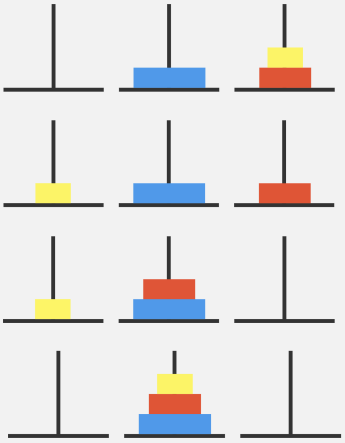
Задача

Задача о ханойской башне. Даны три стержня, на один из которых нанизаны несколько колец, причём кольца отличаются размером и лежат меньшее на большем. Задача состоит в том, чтобы перенести пирамиду колец на другой стержень. За один ход разрешается переносить только одно кольцо, причём нельзя класть большее кольцо на меньшее.

Например, пусть есть башня из трех колец. Мы можем переложить верхнее, желтое кольцо на второй стержень, затем красное на третий стержень, а желтое на красное:



Теперь синее кольцо положим на второй стержень, затем желтое вернем на первый, красное на синее, а желтое на красное:

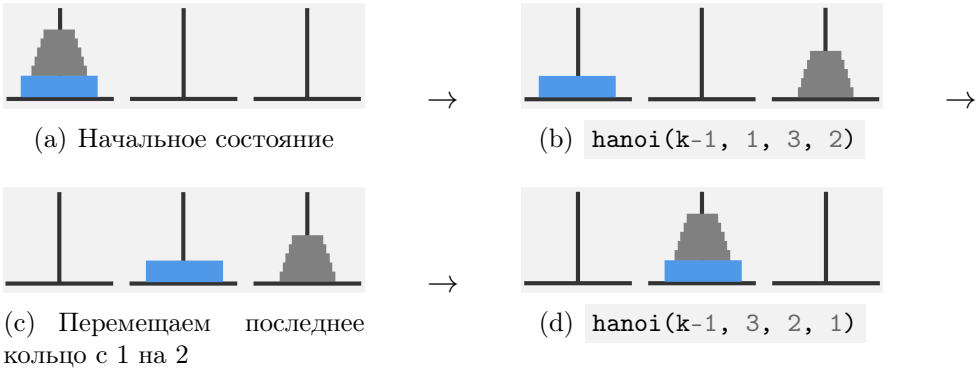


Необходимо написать функцию, решающую эту задачу для произвольного количества колец.

Для конкретики будем считать, что нам необходимо переместить башню на второй стержень. Основная сложность задачи заключается в том, что мы не можем большее кольцо класть на меньшее. Для начала, обобщим задачу: будем в общем случае брать верхние `k` колец со стержня `a` и перекладывать на стержень `b`, используя стержень `c` в качестве вспомогательного. При этом, если на стержнях `b` и `c` есть

какие-то кольца, то будем гарантировать, что все эти кольца больше каждого из `k` перекладываемых колец (то есть, любое из перекладываемых колец можно положить на низ сверху, а значит эти кольца можно не учитывать).

Чтобы понять алгоритм, посмотрим на исходную ситуацию. Необходимо переместить `k=n` колец с первого стержня на второй. Чтобы это сделать, давайте возьмем верхние `k-1` кольцо, и переместим их стопкой на третий стержень. Да, если `k>2`, то мы не можем это сделать за одно действие, однако мы можем сделать рекурсивный запуск, ведь мы гарантировали, что для всех этих перемещаемых колец всё, что ниже, имеет больший радиус. Сделаем рекурсивный запуск `hanoi` с параметрами `(k-1, 1, 3, 2)`, то есть переместим `k-1` кольцо с первого стержня на третий. Теперь переместим оставшееся, `k`-е кольцо на второй стержень одним действием. Отлично, нам осталось переместить всю башню из `k-1` кольца с третьего стержня на второй, используя первый стержень в качестве вспомогательного. У нас снова выполнено требование, что всё что внизу больше, чем каждое из перемещаемых колец. Выполним рекурсивный запуск с параметрами `(k-1, 3, 2, 1)`.



(a) Начальное состояние

(b) `hanoi(k-1, 1, 3, 2)`

(c) Перемещаем последнее кольцо с 1 на 2

(d) `hanoi(k-1, 3, 2, 1)`

Алгоритм решения задачи о ханойской башне

Теперь перейдем к коду. Напишем функцию `hanoi`, перемещающую `k` верхних колец со стержня `a` на стержень `b`, используя стержень `c`. Как и все рекурсивные функции, начнем с точки выхода. Если `k == 0`, то перемещать нечего, выйдем из функции. Первым делом, как мы выяснили ранее, нам нужно перенести башню из `k-1` кольца на вспомогательный стержень `c`. Значит, нам нужно сделать вызов функции `hanoi` с параметрами `(k-1, a, c, b)`. Теперь, когда верхние `k-1` кольцо расположены на стержне `c`, нужно переместить последнее `k`-ое кольцо на стержень `b` – сделаем вывод, который соответствует этому перемещению. И после этого, снова с помощью рекурсивного вызова функции `hanoi` с параметрами `(k-1, c, b, a)` переместим башню со вспомогательного стержня на стержень `b`:

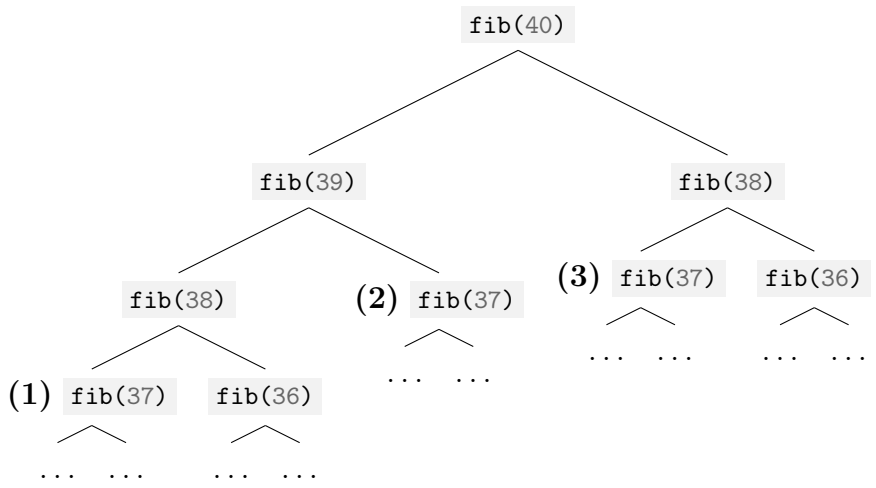
```python
def hanoi(k, a, b, c):
    if k == 0:
        return
    hanoi(k - 1, a, c, b)   # переносим верхние k-1 с a на c
    print(f'{a}->{b}')      # перемещаем верхнее кольцо с a на b
    hanoi(k - 1, c, b, a)   # переносим верхние k-1 с c на b

hanoi(3, 1, 2, 3)
```

```
1->2
1->3
2->3
1->2
3->1
3->2
1->2
```

Попробуйте вручную проверить, как работает этот код, ии убедитесь, что он выводит правильную последовательность операций. Если вы понимаете, что этот код правильно решает задачу, то вы поняли принцип рекурсии. Однако не пугайтесь если задача про ханойские башни показалась непонятной, особенно с первого раза. Попробуйте решить задачи попроще, затем вернитесь и пересмотрите это решение еще раз.

4 Скорость работы рекурсии

Давайте вернемся к первой задаче, где мы вычисляли числа Фибоначчи. Попробуем посчитать какое-нибудь относительно большое число Фибоначчи, например, сороковое. Если мы запустим функцию `fib(40)`, то она будет выполняться очень долго. Дело в том, что мы постоянно запускаем функцию `fib` с одним и тем же параметром; например когда мы считаем `fib(40)`, она три раза запускает `fib(37)`:



При этом нет смысла вычислять 37-ое число несколько раз, оно от этого не поме-

няется. В Python есть функционал, который позволяет не запускать функцию, если её уже запускали с такими параметрами - вместо этого Python просто вспомнит результат и сразу же его использует. Данная фишка использует декоратор, инструмент языка, с которым мы еще не знакомы, но в использовании он очень прост: нужно всего лишь дописать приведенную строчку перед функцией, предварительно подключив декоратор:

```
from functools import lru_cache

@lru_cache()
def fib(n):
    if n == 0 or n == 1:
        return 1
    return fib(n-1) + fib(n-2)

print(fib(10))
```

Теперь, `fib` не будет запускаться с одним и тем же аргументом несколько раз. Для демонстрации того, насколько наша программа ускорится, добавим счетчик количества запусков рекурсивной функции. Будем использовать глобальный счетчик, и запустим оба решения (с декоратором и без) для поиска десятого числа Фибоначчи:

```
def fib(n):
    global cnt
    cnt += 1
    if n == 0 or n == 1:
        return 1
    return fib(n-1) + fib(n-2)

cnt = 0
print(fib(10))
print(cnt)
```

```
177
```

```
from functools import lru_cache

@lru_cache()
def fib(n):
    global cnt
    cnt += 1
    if n == 0 or n == 1:
        return 1
  return fib(n-1) + fib(n-2)

cnt = 0
print(fib(10))
print(cnt)
```

```
11
```

Заметим, что в случае без декоратора, нам потребовалось 177 запусков функции, а для случая с декоратором – только 11 запусков (по одному для каждого `n` от `0` до `10`). С декоратором можно посчитать очень большие числа Фибоначчи за весьма разумное время. Однако, не всегда его уместно использовать. Например, когда мы решали задачу о ханойских башнях, нам часто приходилось на нижнем уровне рекурсии перекладывать одно кольцо. Это требовалось делать каждый раз, чтобы вывести соответствующую операцию на экран – нельзя было взять готовый результат.