АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Оглавление

1 Введение в алгоритмы и структуры данных

Определение кратчайшего пути, выбор объектов по определённым признакам, поиск наилучшего совпадения строк — каждую из этих задач можно решить несколькими способами. Но один подход окажется самым простым в реализации, другой — наименее ресурсоёмким, а третий — максимально быстрым, но при этом потребует много оперативной памяти.

Это зависит от алгоритма, который будет применяться при каждом из решений.

Алгоритм — это инструкция, которая описывает порядок выполнения действий. Алгоритмы описывают, как следует преобразовать данные, чтобы получить требуемый результат.

Допустим, вы хотите взять книгу в библиотеке. Среди нескольких тысяч изданий библиотекарь быстро найдёт нужный вам экземпляр, потому что книги на полках расставлены в алфавитном порядке. А теперь представьте, что нужно отыскать какую-нибудь книгу дома, в обычном книжном шкафу. Если книги расположены не по порядку, то на поиски может уйти много времени.

Но вот в библиотеку поступило собрание сочинений Чехова в восемнадцати томах. Свободного пространства в нужном месте может не оказаться, и библиотекарю придётся переносить книги с полки на полку. Этот процесс займёт какое-то время, зато после добавления новых книг алфавитный порядок в библиотеке сохранится. А в домашний книжный шкаф, в котором нет определённой системы, положить новые книги можно быстро и легко — на любое свободное место, но потом придётся тратить время на поиск нужной книги.

Итак, вот что мы можем сказать про эти «структуры данных»:

- Библиотека, в которой книги расставлены в алфавитном порядке: быстрый поиск, но добавление новых книг в некоторых случаях может быть медленным.
- Шкаф, в котором книги стоят без определённого порядка: медленный поиск, зато добавлять новые книги можно быстро.

Какая система выгоднее? Однозначного ответа нет.

С темой алгоритмов тесно связаны **структуры данных**: разные структуры хранят данные по-разному. Вы уже работали со списками, словарями, сетами и кортежами; для эффективной работы с ними разработчик должен понимать, как они устроены и какие операции доступны для работы с ними. Поэтому в этой теме пойдёт подробный разговор не только об алгоритмах, но и об основных структурах данных.

Зачем изучать алгоритмы и структуры данных:

- Вы узнаете, как оценивать эффективность кода и сможете применять этот навык в работе, в том числе при проведении код-ревью.
- Знание специфики разных структур данных позволит вам правильно их использовать — и ваш код будет эффективнее.
- На задачах на алгоритмы удобно тренировать навык написания чистого кода.
- Алгоритмы не устаревают. Они не привязаны к конкретным технологиям или техническому стеку, поэтому полученные знания и навыки будут актуальны до тех пор, пока люди не перестанут писать код.
- Вы не изучите все существующие алгоритмы, да это и не нужно. Столкнувшись с новой задачей, вы уже будете знать, что и где нужно искать, и сможете самостоятельно разобраться с решением.

2 Линейный и бинарные поиски

Одна из основных задач при работе с алгоритмами — оценка эффективности программы и поиск наиболее экономичного подхода. Допустим, нам нужно найти какой-нибудь элемент в массиве. Давайте рассмотрим один из возможных алгоритмов решения этой задачи и оценим время его работы.

<u>Постановка задачи:</u>

Дан массив целых чисел длины **N**. Нужно найти в нём заданное число **x** и вернуть его индекс. Если **x** в массиве не встречается — вернуть **-1**.

Рассмотрим одно из решений этой задачи:

```
def find_element(numbers, x):
    for i in range(len(numbers)): # проходим по всем элементам массива
        if numbers[i] == x: # сравниваем их с иксом
            return i  # если нашли - возвращаем индекс
    return -1  # если не нашли - возвращаем -1
```

Алгоритм решает поставленную задачу. Но насколько быстро он это делает?

Попробуем понять, сколько элементарных операций совершается в процессе его работы. Под элементарной операцией понимают любую арифметическую операцию или операцию сравнения.

В реализации алгоритма видно, что для каждого элемента массива будет выполнена одна элементарная операция — сравнение с **x**, увеличение счётчика **i** и т. д. Осталось понять, сколько элементов будет рассмотрено.

В зависимости от значения **x** будет рассмотрено разное количество элементов. Однако при оценке скорости работы алгоритма обычно оценивают сложность в **худшем случае** (англ. *worst-case complexity*).

В нашем примере можем сказать, что скорость работы алгоритма в худшем случае пропорциональна размеру массива. На математическом языке ещё говорят: «**Вычислительная сложность алгоритма линейно зависит от размера входных данных**».

Разберём эту фразу:

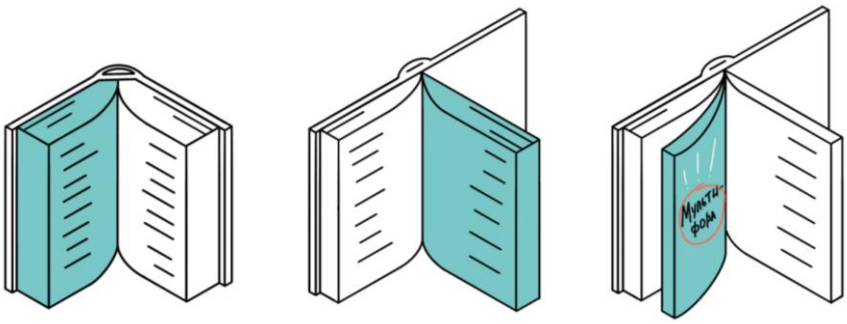
Вычислительная сложность алгоритма. Обычно под этой фразой программисты понимают количество элементарных операций, которые будут совершены.

Размер входных данных. Входные данные — то, что алгоритм получает на вход. В нашей задаче это массив **numbers** и переменная **x**. Размер входных данных примерно равен **N**.

Из-за того, что у описанного алгоритма поиска время работы линейно зависит от размера входных данных, он называется *линейным поиском*.

Есть и другой способ решить задачу поиска элемента в массиве. Если элементы в массиве упорядочены по возрастанию, то найти нужный можно гораздо быстрее.

Представьте, что ищете слово «*мультифора*» в словаре. Вряд ли вы станете листать страницу за страницей, начиная с первой, — ведь слова отсортированы по алфавиту. Лучше открыть книгу примерно посередине. Если вы попали на страницу с нужным словом, алгоритм завершён. Если же слово находится раньше открытой страницы, вы откроете словарь примерно в середине первой половины, а если после — в середине второй. И так до окончания поиска.



Допустим, мы хотим применить этот алгоритм к словарю из 100 страниц. Объём рассматриваемой части книги будет каждый раз уменьшаться вдвое до тех пор, пока не останется всего одна страница. Делить словарь из 100 страниц пополам мы можем максимум 7 раз. Получается, чтобы найти нужное слово, нам достаточно будет просмотреть не более 7 страниц.

А теперь давайте развернём этот алгоритм в обратную сторону. Имея одну страницу, можно за 7 итераций получить 128 (что даже больше 100) страниц, на каждом шаге увеличивая объём вдвое: 2, 4, 8, 16, 32, 64, 128.

На математическом языке говорят, что число 7 является «логарифмом числа 128 по основанию 2».

Определение: $\log_b(a)$ — это то, в какую степень нужно возвести число **b**, чтобы получить число **a**.

$\log_2(100) \approx 6.64$ Но нас интересуют целые числа, так как число просмотров страниц не может быть дробным.

Сколько всего будет шагов?
Через 6 шагов будет 64 страницы.
Недостаточно!

$\log_2(64) = 6$

А уже на 7 шаге получим число,
превосходящее искомое.

$\log_2(128) = 7$

⟶ Будет максимум 7 шагов.

Рассмотренный алгоритм называется *бинарным поиском*. Ещё его называют: двоичный поиск, метод деления пополам, дихотомия. Скорость его работы имеет логарифмическую зависимость от размера входных данных.

Линейный vs Бинарный поиск

Если запустить линейный и бинарный поиски на одинаковых данных, то результат времени работы в зависимости от размера массива, в котором производится поиск, будет выглядит следующим образом:

Размер массива	Линейный поиск	Бинарный поиск
10 элементов	0.01 с.	0.01 с.
100 элементов	0.1 с.	0.02 с.
1 000	0.98 с.	0.03 с.
10 000	9.75 с.	0.06 с.
100 000	67.25 с.	0.45 с.
1 000 000	около 10 минут	2.29 с.
10 000 000	больше полутора часов	20.59 с.

Обратите внимание, что на небольших объёмах входных данных разница между линейным и бинарным поиском не такая уж большая. Но чем больше данных на входе — тем сильнее заметна разница. Именно поэтому работу алгоритмов обычно сравнивают на больших числах.

3 Сложность алгоритма

Как мы выяснили выше у разных алгоритмов может быть разное время работы. Чтобы каждый раз не повторять фразы «Вычислительная сложность алгоритма линейно зависит от размера входных данных» или «У алгоритма логарифмическая асимптотическая сложность», придумали следующие сокращения:

- **O(n)** — линейная зависимость;
- **O(log n)** — логарифмическая зависимость.

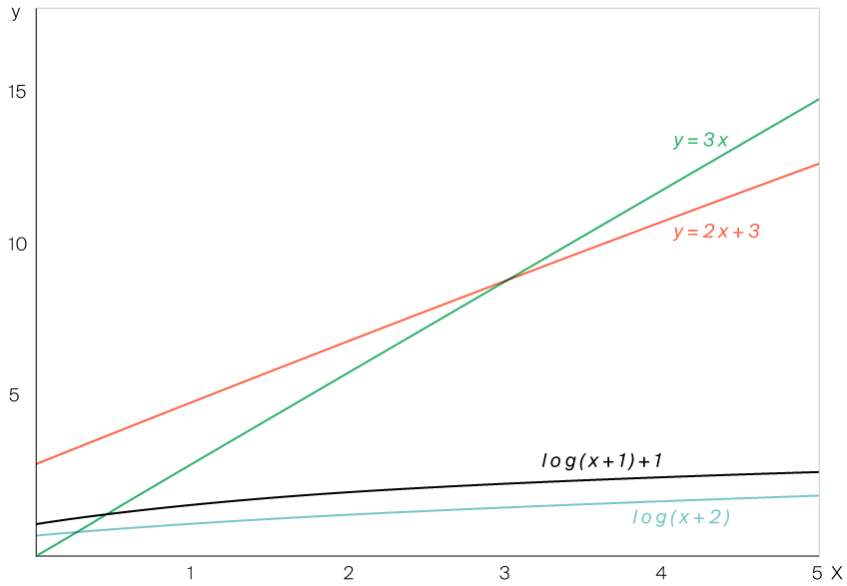
И говорят так: «Сложность алгоритма — O(n)» (читается «*о большое от эн*» или просто «*о от эн*»). Такие сокращения называются **О-нотацией** (англ. Big O *notation*).

В **О-нотации** не учитываются константы и коэффициенты. То есть если в алгоритме совершается **5n + 3** операций, его сложность будет все равно **O(n)**. В оценке не учитываются значения констант при **n**. Нельзя сказать, что константы совсем уж не важны, но они не могут принципиально изменить применимость алгоритма на практике.

Чтобы показать это, сравним время работы нескольких алгоритмов.

- Первый алгоритм делает **2n + 3** операции.
- Второй — **3n** операций.
- Третий — **log(n+2)** операций.
- Четвёртый — **log(n+1) + 1** операций.

Посмотрим на графики зависимости количества операций от **n**:



Первый и второй алгоритмы похожи между собой, так же как третий и четвёртый. Но два первых алгоритма сильно отличаются от двух последних. Так происходит потому, что при больших **n** вид зависимости имеет бóльшее значение, чем коэффициенты и дополнительные слагаемые. Поэтому в **О-нотации** пишут только вид зависимости.

Кроме линейной и логарифмической, при оценке времени работы алгоритмов часто встречаются ещё такие зависимости:

- Квадратичная зависимость — **O(n^2)**.
- Кубическая зависимость — **O(n^3)**.
- Экспоненциальная зависимость — **O(2^n)**.

- Константная зависимость — **O(1)**. Бывает и так, что время работы алгоритма не зависит от размера входных данных, и в любом случае выполняется константное количество операций.

Пример 1:

В больнице работает доктор Игнатий Петрович. Нужно определить, сколько раз в день откроется дверь в кабинет Игнатия Петровича при условии, что все пациенты, уходя, закрывают её за собой.

Каждый посетитель кабинета откроет дверь два раза. То есть количество открытий двери будет вдвое больше, чем количество вошедших. Получаем линейную зависимость, которую можно задать уравнением: **y = 2x**, где **y** обозначает, сколько раз будет открываться дверь, а **x** — количество вошедших.

Можно сказать, что количество открытий двери — это **O(x)**, где **x** — количество пациентов.

Пример 2:

Рассмотри пример с рукопожатиями коллег на работе: если каждый будет жать руку каждому. Предположим, всего встретились **n** человек. Каждый будет здороваться со всеми, кроме себя самого, то есть **n - 1** раз. Всего случится $\frac{\mathbf{n}*(\mathbf{n}-\mathbf{1})}{\mathbf{2}} = \frac{\mathbf{n^2}}{\mathbf{2}} - \frac{\mathbf{n}}{\mathbf{2}}$ рукопожатий.

Это квадратичная зависимость, то есть можно сказать, что будет **O(n^2)** рукопожатий.

Обратите внимание, что в оценке мы отбрасываем слагаемое $\frac{\mathbf{n}}{\mathbf{2}}$ и коэффициент $\frac{\mathbf{1}}{\mathbf{2}}$ при слагаемом $\mathbf{n^2}$.

Поскольку при большом **n** эти числа не будут играть значимой роли. Главное, что зависимость квадратичная.

4 Оперативная память и представление данных

Эффективность программы можно оценить по тому, как она расходует разные типы ресурсов компьютера. Основные из них — процессорное время и оперативная память. А ещё есть пропускная способность сетевых подключений, время работы жёсткого диска и некоторые другие ограниченные ресурсы.

Выше мы говорили о том, как алгоритм использует процессорное время. Теперь поговорим о потреблении оперативной памяти — это называется **пространственной сложностью алгоритма**.

Как устроена оперативная память

Оперативная память, или **ОЗУ** — это своего рода «черновик», который программа использует для вычислений. В него можно записывать и перезаписывать информацию, и оттуда же её можно считывать.

Современные компьютеры располагают всего несколькими гигабайтами оперативной памяти, и данные исчезают из неё сразу после завершения работы программы. Поэтому эта память не подходит для длительного хранения информации, зато она работает в сотни раз быстрее SSD и во многие тысячи раз быстрее обыкновенного жёсткого диска.

Оперативная память разбита на ячейки размером в 1 байт. У каждой ячейки есть свой порядковый номер, который называется «адрес» (отдельные биты внутри ячейки адресов не имеют). Процессор взаимодействует с оперативной памятью напрямую и использует адреса, чтобы обращаться к данным, — почти как программа обращается к переменным по имени.

Чтобы понять, сколько памяти потребляет программа, нужно выяснить, сколько места занимают отдельные объекты. В разных языках программирования этот объём памяти может отличаться, но основные принципы будут общими. Для начала мы разберём, как используется память в языке C++, так как он позволяет сделать это очень эффективно, гибко и предсказуемо.

Представление базовых типов данных в ОП

Что можно записать в одну ячейку памяти? Как мы уже говорили, размер наименьшей ячейки — **1 байт**, состоящий из **8 бит**. Каждый бит может принимать одно из двух значений: либо **0**, либо **1**. Восемь бит

позволяют закодировать всего **2^8=256** вариантов значений. Получается, что в такую ячейку можно записать, например, целое число в промежутке от 0 до 255.

Если считать эти числа номерами символов, то в **1 байт** можно уместить **1 символ** текста (тип **char**): все буквы латиницы и кириллицы, цифры и основные символы вполне умещаются в 256 вариантов.

Самый распространённый тип целых чисел **int** занимает **4 байта** и позволяет закодировать числа от *–2 147 483 648* до *2 147 483 647*. Эти границы можно вычислить: в **4 байтах** содержится **32 бита**. Один бит тратится на то, чтобы закодировать знак. Оставшийся **31 бит** позволяет закодировать **2^{31}** = *2147483648* чисел. Число «**ноль**» тоже нужно закодировать, поэтому положительных чисел остаётся на одно меньше, чем отрицательных. Если ваши числа не превышают по модулю два миллиарда, можно пользоваться этим типом чисел. Обратите внимание, что всего чисел около четырёх миллиардов, но половина из них отрицательные. Если точно известно, что переменная не может быть отрицательной, можно воспользоваться типом *беззнаковых* целых **unsigned int**, он позволяет хранить числа от *0* до *4 294 967 295*.

Будьте внимательны: если в ходе вычислений получается число за пределами диапазона допустимых значений, то происходит переполнение (англ. *overflow*) типа, и результат вычисления может быть некорректным. Но это не относится к языкам с неограниченным размером целых чисел — таких как, например, Python.

Иногда программе требуется запомнить адрес, то есть местоположение данных в оперативной памяти. Ещё несколько лет назад для адресации использовали **4 байта** (т. е. **32 бита**) — этого достаточно, чтобы адресовать **2^{32} байт = 4 ГБ** памяти. Но с тех пор мощности компьютеров и аппетиты программ выросли, четыре гигабайта памяти стали тратиться слишком быстро. Программистам пришлось перейти на **8-байтовую** адресацию памяти, которой должно хватить надолго.

Составные типы данных

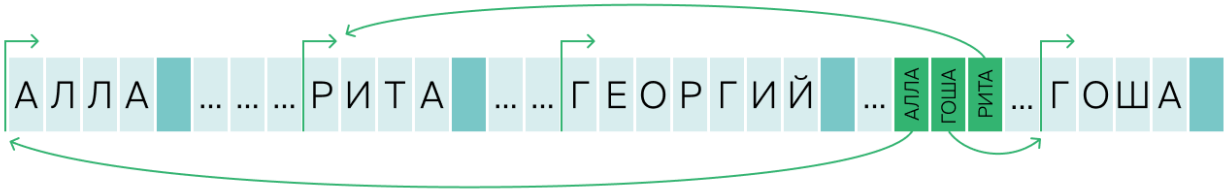
Теперь посчитаем, сколько памяти потребуется для хранения массива из **10 строк** по **20 символов** каждая.

Во-первых, потребуется хранить сам текст. Это займёт:

$$10 \text{ строк} * \left(20 \frac{\text{символов}}{\text{строка}}\right) * \left(1 \frac{\text{байт}}{\text{символ}}\right) = 200 \text{ байт}$$

Этого было бы достаточно, если бы строки располагались в памяти друг за другом. Но производить с ними какие-либо операции будет сложно. Например, чтобы добавить символ в первую строку, нам пришлось бы сначала сдвинуть все остальные строки — как на картинке:

Поэтому в массивы (и в другие составные структуры) зачастую помещаются не сами объекты, а только указатели на них — адреса, по которым они хранятся в ячейках оперативной памяти:



В нашем примере строки располагаются в произвольных местах памяти, а массив из десяти строк содержит лишь десять адресов. Получается, что массив строк занимает 200 байт на само содержимое строк + 10 адресов * 8 байт/адрес = **280 байт**.

Во многих других языках каждый объект в программе записан в специальную «обёртку», которая, помимо данных, содержит вспомогательную информацию. Из-за этого, например, в языке Python даже короткие **целые числа** могут занимать не 4 байта, а почти **30 байт**. Для хранения **строки** требуется около **40 байт** вспомогательной информации, для хранения массива — ещё больше. Обычно объекты в таких языках ведут себя подобно строкам из предыдущего примера: они могут находиться в произвольном месте памяти, и любое обращение к ним происходит по сохранённому адресу.

Посчитаем, например, сколько памяти расходует массив из 10 чисел в Python и на что она уходит:

```
import sys
sys.getsizeof(42)  # => 28 байт занимает короткое целое число
sys.getsizeof([])  # => 56 байт занимает пустой массив
sys.getsizeof([42])  # => 64 = (56 + 8) байт занимает массив с одним элементом.
sys.getsizeof([1,2,3,4,5,6,7,8,9,10])  # => 136 = (56 + 8*10) байт занимает массив
                                       #          с десятью элементами.
                                       # сами данные хранятся отдельно
                                       # и добавляют 280 = (28 * 10) байт
```

Получается, мы тратим **56 байт** на то, чтобы создать **массив**, а дальше по **8 байт** на каждый новый объект в массиве — это **адреса** элементов. Но это ещё не всё. Ведь каждому **числу** нужно ещё по **28 байт**. Итого **массив** из **десяти** чисел в Python занимает суммарно 56 + (8 + 28) * 10 байт = 416 байт. Сравните это с 40 байтами в языке C++.

5 Массивы постоянного размера

Начнем с того, что такое структура данных. **Структура данных** — это способ организации информации в памяти, который позволяет проводить с ней определённый набор операций. Например, быстро искать или изменять данные.

Массив — одна из базовых структур данных. Самый простой тип массивов имеет **фиксированный размер** и может хранить элементы одного и того же типа. Например, если мы создадим массив из десяти целых чисел, в него нельзя будет добавить ещё один элемент или записать объект неподходящего типа. Такие массивы вы можете встретить, например, в языке C — *int numbers[10]*.

С массивами фиксированного размера можно выполнить только две операции:

- узнать значение элемента по индексу,
- перезаписать значение по индексу.

Да, они не слишком функциональные, но зато очень эффективные. Каждая из операций выполняется за O(1).

Чтобы понять, как массиву удаётся настолько быстро находить нужный элемент, — разберёмся в его механике работы и посмотрим, как он хранит данные.

Как устроен массив

Массив — это набор однотипных данных, расположенных в памяти друг за другом. Поэтому он всегда занимает один непрерывный участок памяти. Точное местоположение этого участка операционная система сообщает программе в момент создания массива.

В начале выделенного участка памяти находится нулевой элемент, сразу за ним — первый и далее по порядку. Они идут друг за другом, без пропусков. Зная положение элемента в памяти — его адрес, мы можем этот элемент прочитать или записать. Теперь посмотрим, как узнать адрес элемента.

Пусть у нас есть массив **numbers** из 10 *беззнаковых* целых чисел. Адрес нулевого элемента нам известен — это **1000**. Чтобы узнать положение следующего элемента, нам нужно прибавить к адресу начала размер элемента в байтах. Каждое число занимает по 4 байта, так что первый (нулевой) элемент займёт байты 1000, 1001, 1002, 1003. Значит, элемент с индексом 1 будет записан по адресу 1004.

$20 + 2 \times 256 = 532$				192				$4 + 1 \times 256^2 = 65540$				...	Значения итоговых чисел
20	2	0	0	192	0	0	0	4	0	1	0	...	Значения отдельных байтов
1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011	...	Адреса

Для многих операций, выполняемых с массивом, необходимо знать его длину. Например, чтобы пройти по всем элементам массива с помощью цикла. В большинстве языков программирования эта информация хранится в специальном поле, привязанном к объекту массива. Благодаря этому не нужно следить за этой переменной самостоятельно.

6 Динамические массивы

В Python динамические массивы реализуются классом **list**, но пусть вас не обманывает название — это не список, а именно массив.

Сложность вставки в динамический массив

Бывают ситуации, когда невозможно сказать точно, какая длина массива может понадобиться. Поэтому так важно иметь возможность добавлять элементы в массив прямо во время выполнения программы. Рассмотрим пару примеров.

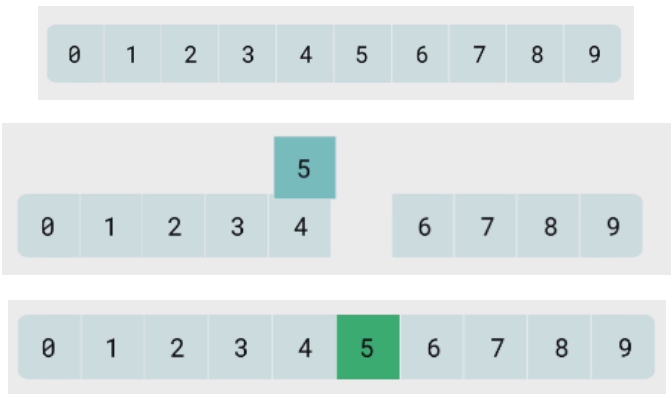
Есть перечень фильмов, в который мы хотим добавить еще один в конец списка:

```
films_wish_list = ["Джон Уик 3", "Аватар 2", "Форсаж 9", "Индиана Джонс 5", "Бэтмен"]
films_wish_list.append('Чёрная Вдова')
```

Чтобы это сделать, достаточно узнать номер последней использованной ячейки массива и записать элемент в следующую за ней ячейку. На это уйдёт **O(1)** времени.

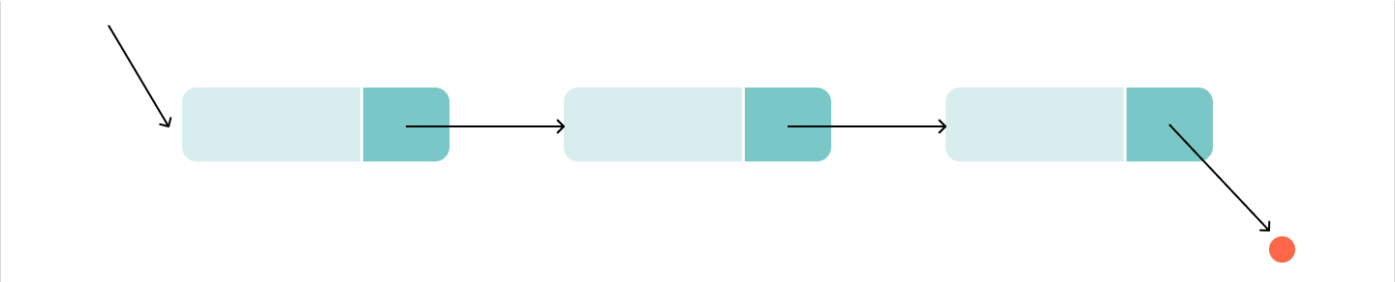
Но что, если необходимо добавить фильм в начало списка? Представьте, что вы записываете дела на листе бумаги. В конец легко добавить новый пункт, а вот в начале места уже нет. Значит, чтобы создать перечень с новым первым пунктом, придётся всё переписывать. Так же и компьютеру нужно переместить каждый из уже существующих элементов на одну ячейку вправо. Поэтому сложность вставки элемента в начало массива составляет **O(n)**, где **n** — количество элементов в массиве.

А что если нужно вставить элемент в произвольное место массива? Конечно, вставка в начало и в конец — частные случаи этой задачи. Вставка элемента в начало — худший случай. Сложность этой операции **O(n)**. Добавление элемента в конец — лучший случай. И его сложность **O(1)**.



7 Связанные списки

В связном списке у каждого элемента, помимо его значения, есть ссылка на следующий элемент списка. За исключением последнего — он ссылается в никуда. В зависимости от языка программирования ссылка в никуда может представлять собой объект **None**, нулевой указатель или аналогичную сущность. Кроме того, у связного списка принято определять точку старта — её называют «*головой списка*» (англ. **head**).



Например, вы решили устроить квест своему другу. Вы положили подарок в тайное место и сделали несколько капсул с подсказками, которые тоже потом спрятали. То есть бумажка из первой капсулы давала подсказку, где найти подарок, вторая указывала, где искать первую капсулу, третья отсылала ко второй, четвёртая — к третьей и так далее. А уже другу вы вручаете последнюю написанную вами капсулу.

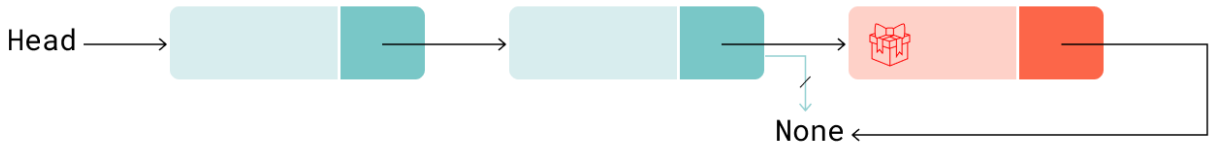
В этом примере подсказки являются указателями на капсулы. Каждой записке соответствует строго одна капсула. Такой же принцип и у связанных списков.

Операции со связанным списком

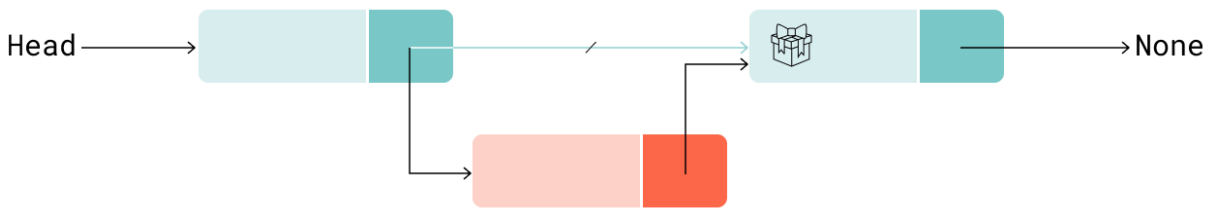
Вернемся к нашему квесту. Если вы вдруг решили добавить еще одну записку, то есть три принципиально разных варианта - в самое начало, в самый конец, либо куда-то в середину.

1. Если добавить новую локацию в самое начало квеста, то в этом месте прячут капсулу с запиской, которую раньше хотели вручить другу, а в новой бумажке указывают на это место и именно её вручают другу.

2. Если добавить ещё одну капсулу в самый конец квеста, то на место подарка кладут новую записку, которая укажет на новую локацию, где и будет спрятан подарок.



3. Если добавить новое место в середину квеста, то надо: во-первых, написать новую записку X1, которая будет лежать в новой локации и указывать на место с капсулой X2; во-вторых, исправить записку X, раньше отсылавшей к записке X2, чтобы теперь она указывала на капсулу в локации X1. При этом записку X2 исправлять не надо.



8 Структура данных стек

Структура данных **стек** (англ. stack, «стопка») часто встречается в программировании и во многих вещах, связанных с компьютерами. Стек работает по принципу **LIFO** (англ. Last In First Out, «*последним вошёл — первым вышел*»).

Вы наверняка пользуетесь возможностью браузеров, реализованной с применением стека. Речь идёт о кнопке «назад». Нажмёте на неё один раз — вернётесь на предыдущую страницу, нажмёте снова — попадёте на страницу, где были до неё. Во многих текстовых редакторах можно отменять последние операции. Эта функция тоже реализована при помощи стека. Если одна функция вызывает другую, в памяти компьютера эти операции также помещаются в стек.

Интерфейс стека

В отличие от массивов и связных списков, стек — это прежде всего удобный интерфейс для взаимодействия с данными, а не физическая особенность их хранения. Когда мы говорим про стек, мы подразумеваем структуру данных, которая реализует следующие методы:

- **push(item)** — добавляет элемент на вершину стека;
- **pop()** — возвращает элемент с вершины стека и удаляет его;
- **size()** — возвращает размер стека (количество лежащих в нём элементов).

Иногда стек реализует дополнительные операции:

- **peek()** или **top()** — возвращает элемент с вершины стека, не удаляя его;
- **isEmpty()** — определяет, пуст ли стек.

В большинстве случаев стек удобно реализовывать на массиве. Но бывают ситуации, когда стек должен хранить очень много элементов, а выделить единый блок памяти для этого невозможно. Тогда для реализации лучше воспользоваться связным списком.

Реализация стека

Реализуем стек на основе массива. В конструкторе для экземпляра класса создаётся пустой массив. Далее при вызове методов **push()** и **pop()** этот массив будет меняться:

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[-1]

    def size(self):
        return len(self.items)
```

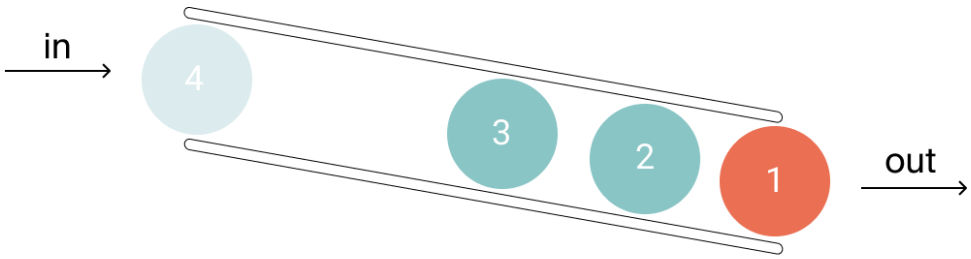
Создадим объект класса **Stack**, чтобы посмотреть, как работают его методы:

```
stack = Stack()
stack.push('apple')
stack.push('banana')
```

```
stack.push('orange')

print(stack.items)  # Вывод: ['apple', 'banana', 'orange']
# возьмем последний элемент из массива, удалив его
print(stack.pop())  # Вывод: orange
print(stack.items)  # Вывод: ['apple', 'banana']
```

8 Структура данных очередь

Ещё одна важная структура данных — «**очередь**». В отличие от стека, который работает по принципу LIFO, очередь основана на концепции **FIFO**. То есть первым извлекают элемент, который добавили раньше всех.



Как и стек, **очередь** — интерфейс общения с данными. Он гарантирует наличие определённых методов, но не даёт никаких обещаний по способу хранения этих данных. То есть «под капотом» может находиться всё что угодно.

При том, что интерфейс не накладывает ограничений на внутреннее устройство, очередь принято реализовывать таким образом, чтобы операции вставки и удаления элемента выполнялись за O(1).

Когда мы говорим про очередь, мы ожидаем, что у структуры будут следующие методы:

- **push(item)** — добавляет элемент в конец очереди;
- **pop()** — берёт элемент из начала очереди и удаляет его;
- **peek()** — берёт элемент из начала очереди без удаления;
- **size()** — возвращает количество элементов в очереди.

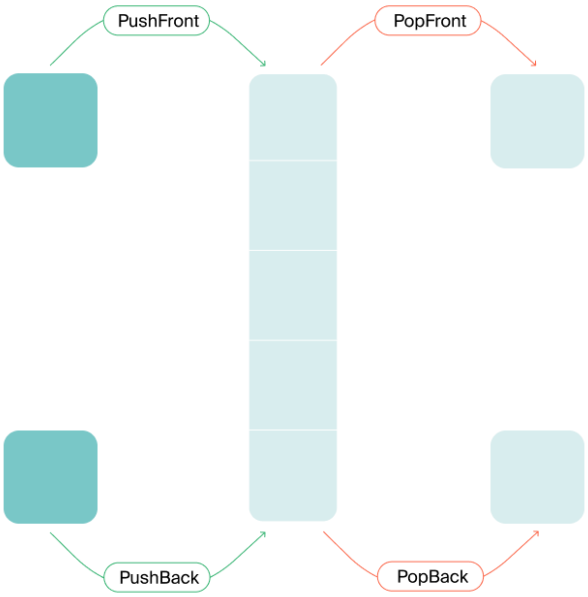
Реализуем очередь на основе массива. В конструкторе для экземпляра класса создаётся пустой массив. Далее при вызове методов **push()** и **pop()** этот массив будет меняться:

```
class Queue:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop(0)

    def peek(self):
        return self.items[0]

    def size(self):
        return len(self.items)


# Создадим объект класса Queue, чтобы посмотреть, как работают его методы:
```

```
queue = Queue()
queue.push('apple')
queue.push('banana')
queue.push('orange')

print(queue.items)  # Вывод: ['apple', 'banana', 'orange']
# возьмем первый элемент из массива, удалив его
print(queue.pop())  # Вывод: apple
print(queue.items)  # Вывод: ['banana', 'orange']
```

9 Структура данных дек

Множество задач можно решить, используя либо стек, либо очередь. Но иногда требуется интерфейс, который позволяет и добавлять, и извлекать элементы с обоих концов. Такой интерфейс называют «**дек**» (англ. **deque** — double ended queue, «*очередь с двумя концами*»).



Интерфейс дека подразумевает, что в нём будут реализованы следующие методы:

- **push_back(item)** — вставка нового элемента в конец;
- **pop_back()** — удаление последнего элемента;
- **push_front(item)** — вставка нового элемента в начало;
- **pop_front()** — удаление первого элемента;
- **size()** — количество элементов в очереди.

Реализуем дек на основе массива. В конструкторе для экземпляра класса создаётся пустой массив. Далее при вызове методов **push()** и **pop()** этот массив будет меняться:

```
class Deque:
    def __init__(self):
        self.items = []

    def push_back(self, item):
        self.items.append(item)

    def pop_back(self):
        return self.items.pop()
```

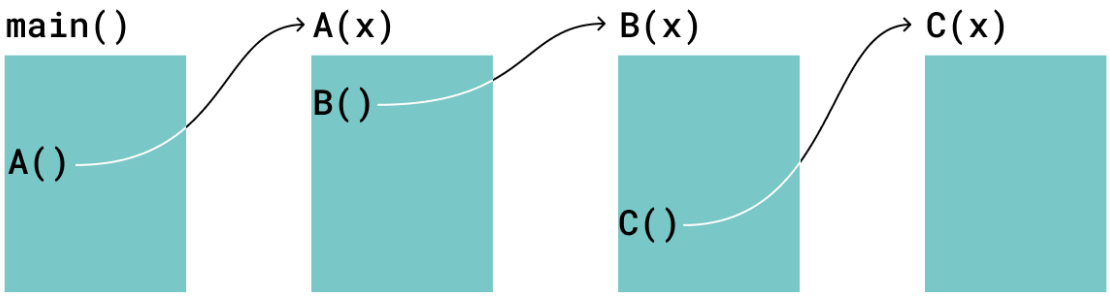
```
    def push_front(self, item):
        self.items.insert(0, item)

    def pop_front(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)


# Создадим объект класса Queue, чтобы посмотреть, как работают его методы:
deque = Deque()
deque.push_front('apple')
deque.push_front('banana')
deque.push_back('orange')

print(deque.items)  # Вывод: ['banana', 'apple', 'orange']
# возьмем первый элемент из массива, удалив его
print(deque.pop_front())  # Вывод: banana
print(deque.items)  # Вывод: ['apple', 'orange']
# возьмем последний элемент из массива, удалив его
print(deque.pop_back())  # Вывод: orange
print(deque.items)  # Вывод: ['apple']
```

10 Стек вызовов

Код любой сложной программы разделён на множество функций. Поэтому работа всей системы описывается как взаимодействие небольших блоков кода. Базовые функции объединяются в сложные, а те — в ещё более сложные. В результате вся программа может быть представлена как одна сложносоставная функция, которая вызывает множество простых.

Допустим, мы написали программу. Первой вызывается функция **main()**. В ней вызывается функция **A**. Она вызывает функцию **B**, которая, в свою очередь, вызывает **C**.

Так выглядит стек вызовов этой программы:



После возврата из функции управление должно вернуться туда, откуда эту функцию вызвали. Для этого при вызове функции в стеке вызовов сохраняется адрес той инструкции программы, которая будет выполнена после возврата из функции.

Для примера рассмотрим функцию, которая приветствует пользователя по имени и выдаёт ему гороскоп на сегодняшний день. Для простоты предположим, что пользователь пришёл на сайт, где всем выдают один и тот же гороскоп.

```
def say_hello(name):
    print(f"Привет, {name}")
    print_horoscope(name.upper())
    print(f"Пока, {name}, хорошего дня!")

def print_horoscope(name):
    print(f"{name}! Сегодня подходящий день для изучения рекурсии")

say_hello('Гоша')

# Получим на выходе:
# Привет, Гоша
# ГОША! Сегодня подходящий день для изучения рекурсии
# Пока, Гоша, хорошего дня!
```

Обратите внимание, что в разных частях программы переменная **name** может хранить разные значения. Например, при входе в функцию **say_hello()** в переменной **name** будет записано 'Гоша'. А если мы вызовем **print_horoscope()**, там появится *'ГОША'*. Но, когда мы перейдём обратно в **say_hello()**, значение вновь вернётся к первоначальному *'Гоша'*.

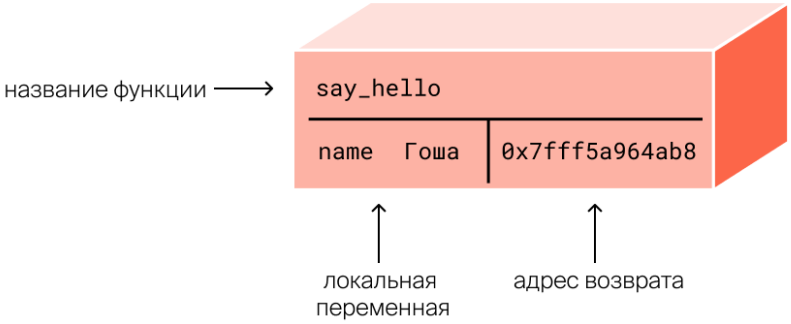
Чтобы добиться такого поведения, программа хранит все локальные переменные на том же стеке. Можно представить это так: на вершине стека находится структура, в которой записаны локальные переменные и аргументы функции, а также адрес возврата. Проследим, как меняется содержимое стека в нашем примере.

Работа стека: шаг за шагом

Первая команда, которая будет выполнена при старте программы, — вызов функции **say_hello()**. Под этот вызов на стеке выделяется блок памяти.

В выделенный блок памяти записывается адрес возврата. Он нужен для того, чтобы определить, куда передать управление после завершения работы функции. Этот адрес соответствует инструкции, следующей сразу за местом вызова функции.

Также в стек помещаются переданные в функцию аргументы. С этих параметров начинается формирование набора локальных переменных функции. В нашем случае есть один аргумент *name = 'Гоша'*, а другие локальные переменные отсутствуют.

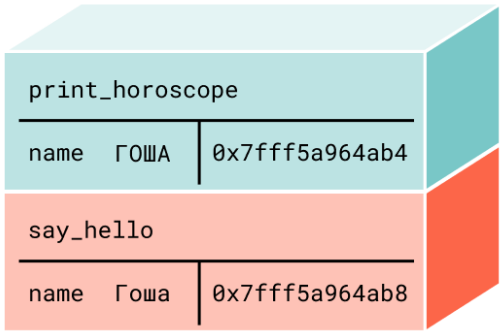


Затем выполняется первая инструкция в функции **say_hello()**, которая выводит сообщение: «*Привет, Гоша*».

Далее происходит вызов метода **upper()**, который вернёт вызвавшей его функции строку *'ГОША'*, написанную большими буквами. Эта строка станет параметром функции **print_horoscope()**.

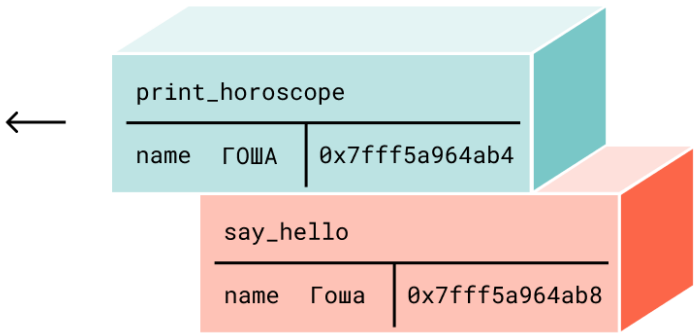
Вызовы функции **print()** и метода **upper()** мы для краткости на картинке не показываем, хотя они, конечно же, тоже кладутся на стек при вызове и снимаются с него после завершения выполнения.

Когда вызывается функция **print_horoscope()** с параметром *'ГОША'*, под этот вызов выделяется новый блок памяти, куда записываются значения переданных функции параметров и новый адрес возврата, а также выделяется место для других локальных переменных. Блок памяти для функции **print_horoscope()** помещается поверх блока, отведённого под **say_hello()**.



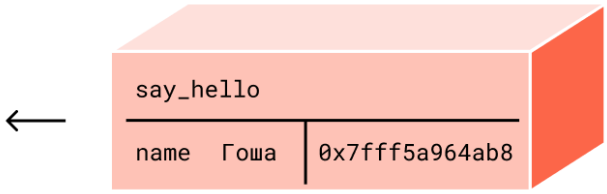
Затем исполнится первая инструкция в функции **print_horoscope()**. Выведется сообщение: «*ГОША! Сегодня подходящий день для изучения рекурсии*».

В **print_horoscope()** больше нет инструкций. Настал момент возвращения управления. В блоке памяти функции **print_horoscope()**, лежащем на стеке, записан адрес возврата, поэтому программе известно, какая инструкция должна исполниться следующей. Блок, отведённый под **print_horoscope()**, больше не нужен, и теперь извлекается из стека. Значение переменной **name** восстанавливается.



Затем исполняется инструкция из **say_hello()**, следующая за вызовом функции **print_horoscope()**. Эта инструкция — вывод сообщения: «*Пока, Гоша, хорошего дня!*»

Так как это последняя из инструкций в **say_hello()**, происходит возврат из функции. Блок памяти, выделенный под неё, освобождается.



Теперь вы знаете, как интерпретатор (или компилятор) языка программирования вызывает функции.

11 Рекурсия. Переполнение стека вызовов

В этом уроке мы поговорим о функциях, которые вызывают сами себя, но с изменёнными значениями аргументов. Такие функции называются «**рекурсивными**». И они тоже используют стек вызовов.

Разберем следующий пример:

Необходимо вычислить количество вариантов упорядочить 3 книги. Сначала ставится первая книга. Есть три способа сделать это. На втором месте может стоять одна из двух оставшихся книг, а для последнего места остаётся всего 1 вариант. Чтобы получить общее число вариантов, нужно перемножить эти числа: **3·2·1=6**. Математики говорят в таком случае, что число перестановок из 3 элементов равно 6.

Чтобы вычислять число перестановок из **n** элементов, ввели специальную функцию, которая называется «**факториал**» и обозначается **n!**

Факториал натурального числа можно вычислить как произведение всех натуральных чисел от 1 до **n**:

$$n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$$

Теперь, когда у нас есть специальная функция для вычисления количества перестановок, можно посмотреть на эту задачу с другой стороны.

Чтобы посчитать число способов расставить **n** книг, мы можем **n** способами выбрать первую книгу, а затем умножить это на число способов расставить оставшиеся **(n−1)** книг. Мы можем записать это в виде формулы:

$$n! = n \cdot (n-1)!$$

Выходит, что мы можем вычислить факториал числа, зная факториал предшествующего ему числа. Это и называется «**рекурсией**».

Распишем рекурсивные вычисления в нашем примере:

$$3! = 3 \cdot 2! = 3 \cdot (2 \cdot 1!) = 3 \cdot (2 \cdot (1 \cdot 0!))$$

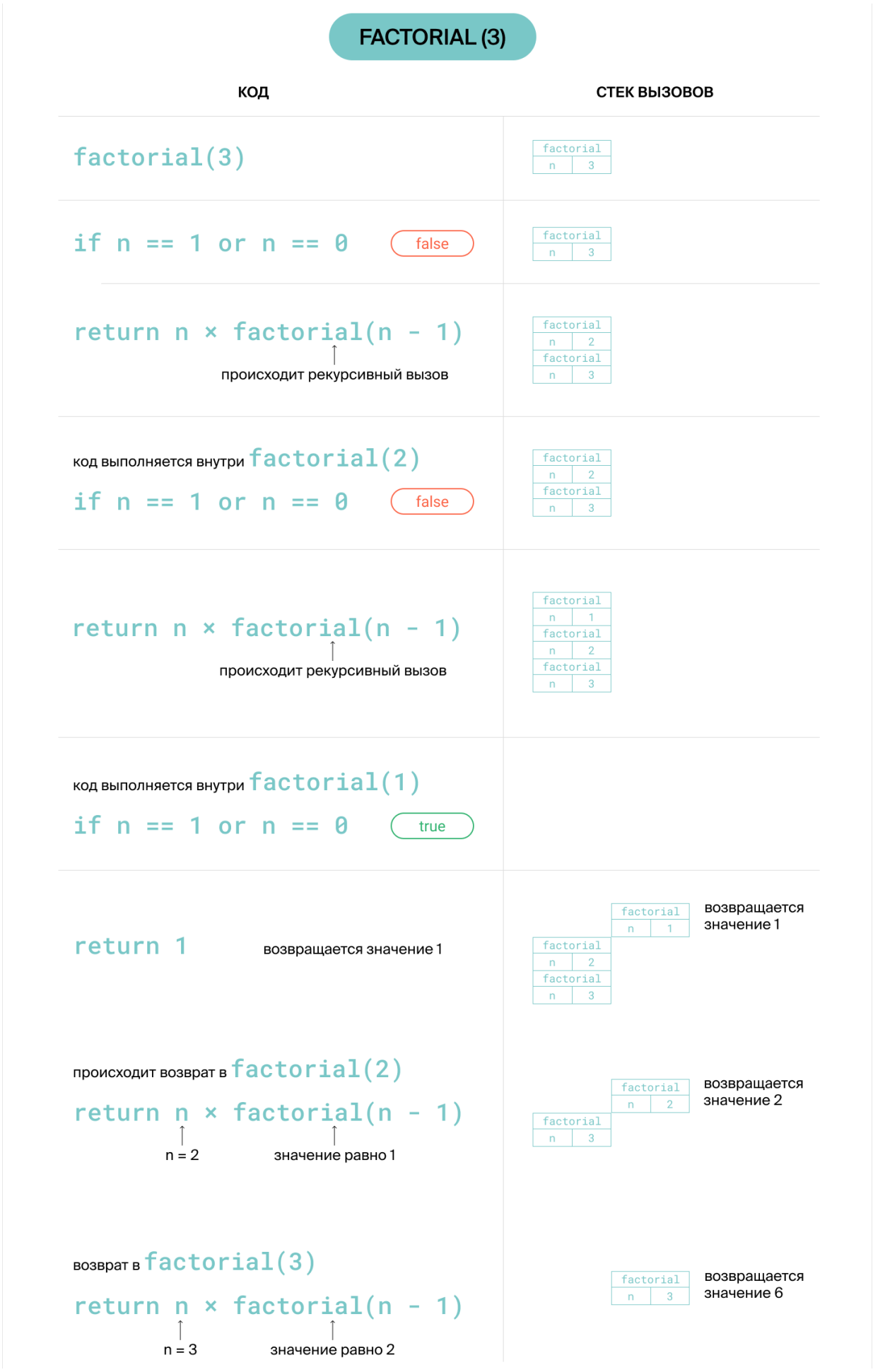
Дальше продолжать мы не можем сразу по двум причинам. Во-первых, факториал не определён для отрицательных чисел. А во-вторых, такую цепочку можно было бы продолжать вечно, и наше вычисление никогда бы не остановилось. Поэтому нам нужно в какой-то точке прервать нашу цепочку вычислений, и в данном случае число **0** — удачное место, чтобы выйти из рекурсии.

Значение **0!** можно определить явным образом: есть ровно один способ расставить на полке ноль книг, поэтому **0! = 1**. Число способов расположить одну книгу — тоже равно 1, поэтому **1! = 1**. Это значение также можно записать как особый случай, хотя делать это и необязательно.

Теперь напишем код рекурсивной функции вычисления факториала:

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

print(factorial(3))  # 6
```

Проанализируем, что происходит при вызове функции **factorial(3)**.



При каждом вызове функции на стеке создаётся новый набор локальных переменных. Таким образом, переменная **n** в каждом вызове оказывается новая. Благодаря этому переменные с одним и тем же именем даже в одной и той же функции не конфликтуют между собой. Невозможно обратиться к переменной **n**, которая относится к другому уровню рекурсии. Ни узнать, ни поменять её значение не получится.

Однако, чтобы хранить локальные переменные и адрес возврата на каждом уровне рекурсии, потребуется **O(d)** памяти на стеке, где **d** — **глубина рекурсии**. При вычислении факториала **n!** глубина рекурсии равна **n**, поэтому этот метод расходует **O(n)** памяти на стеке.