# **Что такое алгоритмы и структуры данных**

Определение кратчайшего пути, выбор необходимого подмножества объектов, поиск наилучшего совпадения строк — каждую из этих задач можно решить несколькими разными способами. Все они дадут одинаковый результат, но один вариант окажется самым простым в реализации; другой — более эффективным; а третий — максимально быстрым, но при этом будет использовать много оперативной памяти.

Так происходит потому, что в каждом из этих решений будут применяться разные алгоритмы. **Алгоритм** — это инструкция, которая описывает порядок выполнения действий. Например, один из вариантов алгоритма поиска минимального элемента в массиве можно записать с помощью такого псевдокода:

функция get\_minimal\_element(input\_list):

minimal\_element = None

для element из input\_list:

если minimal\_element is None ИЛИ element < minimal\_element тогда:

minimal\_element = element

вернуть minimal\_element

Алгоритмы описывают то, как мы преобразуем данные, чтобы получить требуемый результат. С ними тесно связаны **структуры данных**, в которых эти данные хранятся. Разные структуры хранят данные по-разному и из-за этого у них возникают интересные свойства.

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 вам нужно отыскать какую-нибудь книгу у себя дома, в книжном шкафу. Если книги в нём расположены не по порядку, то, вероятно, вы потратите на поиски много времени.

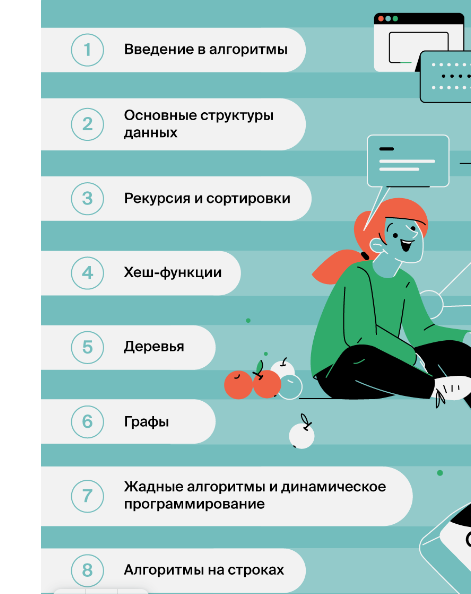
Но вот в библиотеку поступило собрание сочинений А. П. Чехова в восемнадцати томах. Свободного пространства в нужном месте может не оказаться, и библиотекарю придётся переставлять другие книги, переносить их с полки на полку. Этот процесс может занять какое-то время, зато после добавления новых книг 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.

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.

Как видите, даже в таком простом примере нет однозначно лучшего решения. Для разных задач мы будем использовать разные способы хранения данных.

Если вы когда-нибудь писали код, то вы наверняка уже работали с какими-нибудь структурами данных: они называются list, vector, set, dict, map и так далее. Чтобы эффективно ими пользоваться, разработчик должен понимать, как они устроены и какие операции предоставляют. Поэтому мы будем подробно рассказывать не только про алгоритмы, но про основные структуры данных.



Суммирование:

using System;

using System.IO;

public class Solution

{

private static TextReader reader;

private static TextWriter writer;

public static void Main(string[] args)

{

reader = new StreamReader(Console.OpenStandardInput());

writer = new StreamWriter(Console.OpenStandardOutput());

var a = ReadInt();

var b = ReadInt();

writer.WriteLine(a+b);

reader.Close();

writer.Close();

}

private static int ReadInt()

{

return int.Parse(reader.ReadLine());

}

}

## **Приёмы для работы c вводом и выводом**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

public class Solution

{

private static TextReader reader;

private static TextWriter writer;

public static void Main(string[] args)

{

reader = new StreamReader(Console.OpenStandardInput());

writer = new StreamWriter(Console.OpenStandardOutput());

var numbers = ReadList();

writer.WriteLine(numbers[0] + numbers[1]);

reader.Close();

writer.Close();

}

private static List<int> ReadList()

{

return reader.ReadLine()

.Split(new[] { ' ', '\t', }, StringSplitOptions.RemoveEmptyEntries)

.Select(int.Parse)

.ToList();

}

}

## **Считывание массивов**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

public class Solution

{

private static TextReader reader;

private static TextWriter writer;

public static void Main(string[] args)

{

reader = new StreamReader(Console.OpenStandardInput());

writer = new StreamWriter(Console.OpenStandardOutput());

var n = ReadInt();

var numbers = ReadList();

for (var i = 0; i < n; i++)

{

writer.Write("{0} ", numbers[i]);

}

reader.Close();

writer.Close();

}

private static int ReadInt()

{

return int.Parse(reader.ReadLine());

}

private static List<int> ReadList()

{

return reader.ReadLine()

.Split(new[] { ' ', '\t', }, StringSplitOptions.RemoveEmptyEntries)

.Select(int.Parse)

.ToList();

}

}

Обработка временных рядов

Временной ряд — это набор измерений некоторой величины во времени. Значения максимальной дневной температуры на Марсе по данным марсохода Curiosity — это тоже временной ряд. Можно построить график, на котором по вертикали будет указана температура в градусах Цельсия, а по горизонтали — дни миссии Curiosity.

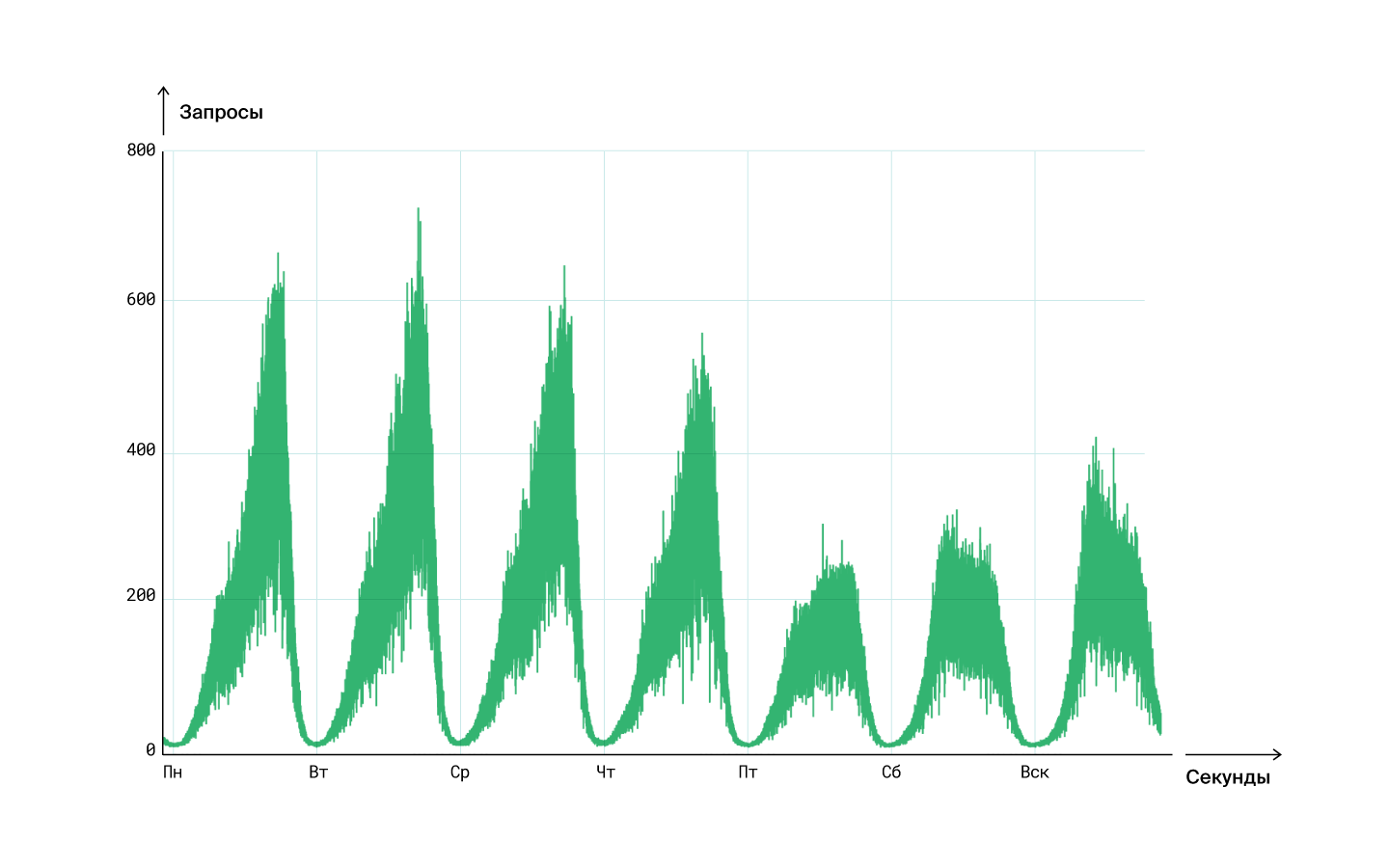


## **Трудовые будни и собачки**

Сформулируем задачу обработки временных рядов чуть иначе. Представим, что одна знаменитая компания запустила сайт, где по запросам пользователей генерируются картинки с весёлыми собачками и мотивирующими фразами.

Менеджер по развитию продукта хочет выяснить, насколько востребован сервис и как распределяется активность пользователей. Все запросы сохранены, данных достаточно, значит, можно построить график.

По горизонтальной оси будут идти секунды, по вертикальной — запросы пользователей.



Общую тенденцию можно уловить с первого взгляда: больше всего запросов приходит в рабочие часы по будням. А вот в пятницу и на выходных собачки теряют актуальность — до следующего понедельника.

Нужно построить более понятный график на основе исходных данных.

## **Метод скользящего среднего**

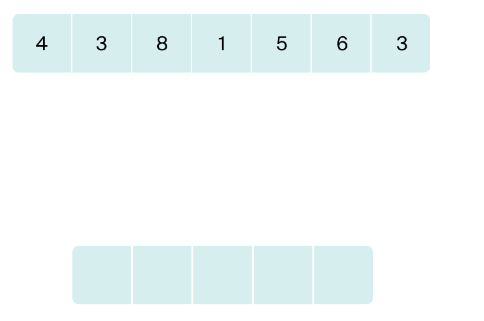
Для решения этой задачи можно применить [метод скользящего среднего](https://ru.wikipedia.org/wiki/%D0%A1%D0%BA%D0%BE%D0%BB%D1%8C%D0%B7%D1%8F%D1%89%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%BD%D1%8F%D1%8F). С его помощью мы снизим шумы в данных и сгладим график, выделив общую тенденцию изменения числа запросов.

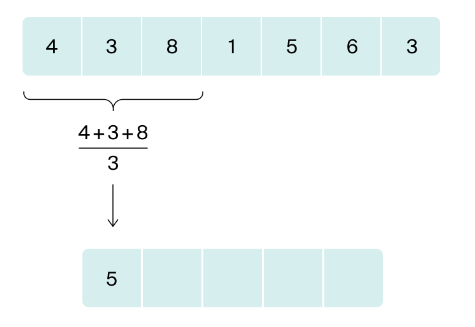
Метод заключается в том, что создаётся новый массив данных, и в нём значение каждой точки высчитывается как среднее арифметическое предыдущих *�K* значений из исходного набора.

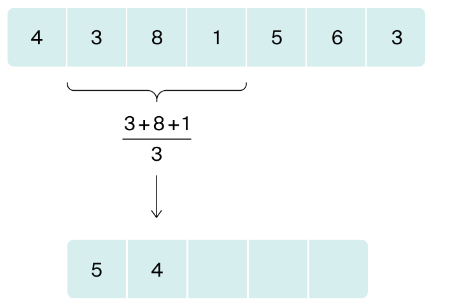
То есть для каждой секунды посчитаем среднее арифметическое от количества запросов за предыдущие *�K* секунд. Этот интервал *�K* будем называть «окном сглаживания». На каждой итерации он смещается, «скользит» — отсюда и название метода.

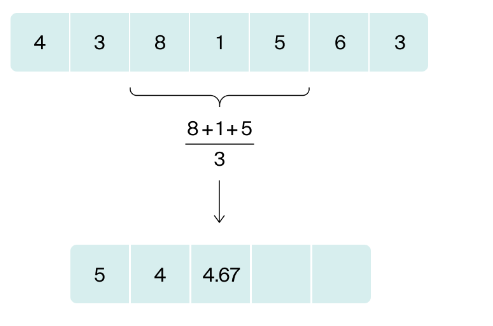
Например, если данные по нагрузке на сайт для 7 секунд будут такие: *[4,3,8,1,5,6,3]*[4,3,8,1,5,6,3], а *�=3K*=3, то сглаженные значения получатся следующими: *[5,4,4.67,4,4.67]*[5,4,4.67,4,4.67].

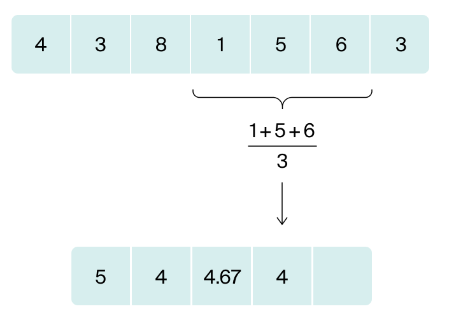
Обратите внимание, что в результирующем массиве будет на *�−1K*−1 элементов меньше: мы не считаем среднее арифметическое меньше чем для *�K* элементов.

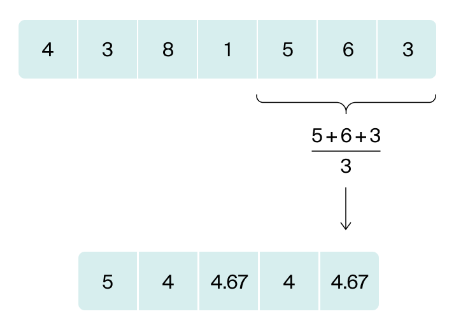










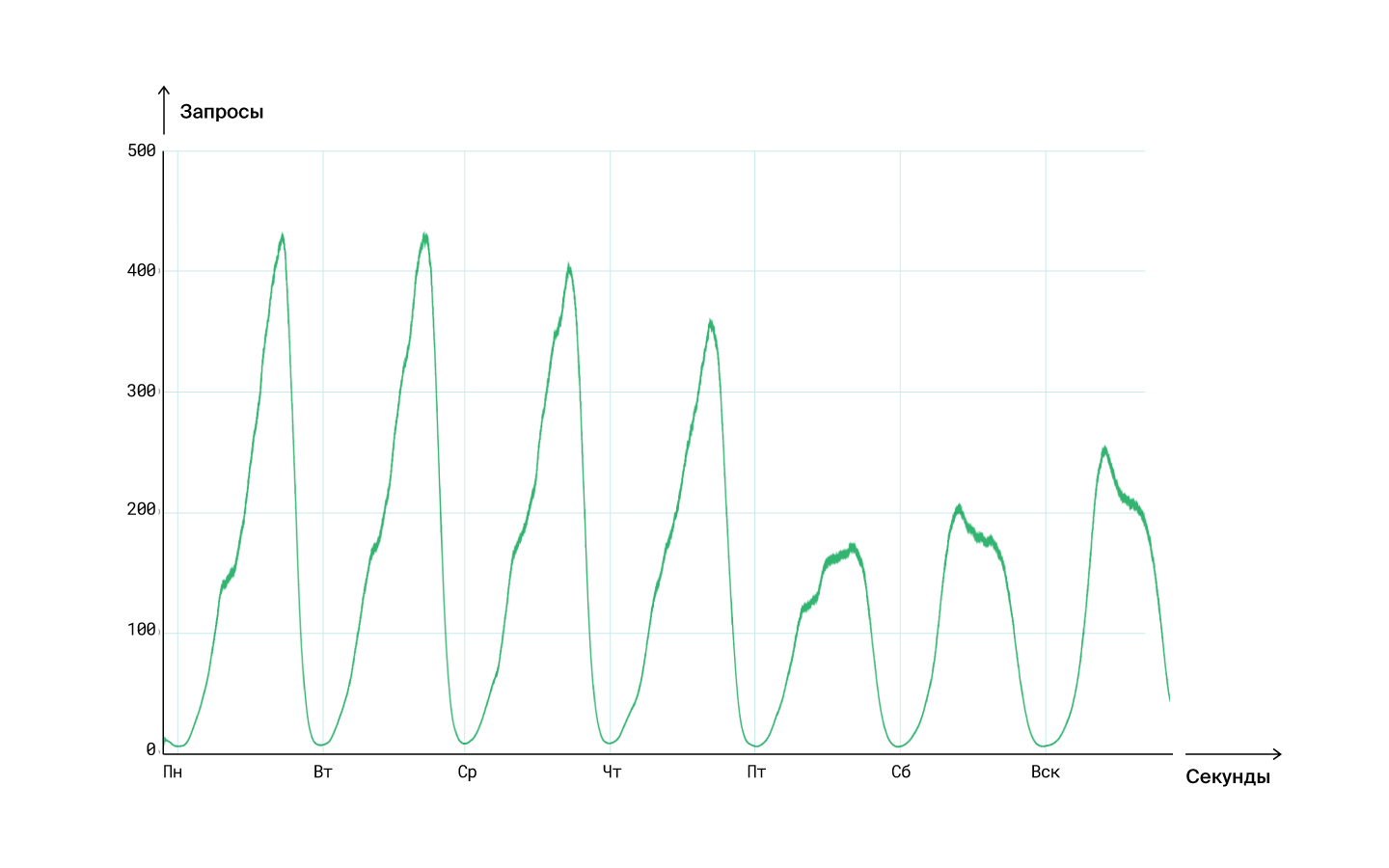


Предположим, что данные упорядочены по времени, записаны с шагом в секунду и не содержат пропусков. Напишем самый простой псевдокод функции, реализующей метод скользящего среднего.

На вход функция будет принимать массив с данными timeseries и размер окна сглаживания K.

функция скользящее\_среднее(timeseries, K): result = [] *# Пустой массив.* для begin\_index из [0 .. len(timeseries) - K]: end\_index = begin\_index + K *# Просматриваем окно шириной K.* current\_sum = 0 для каждого v из timeseries[begin\_index .. end\_index): current\_sum += v current\_avg = current\_sum / K result.добавить(current\_avg) вернуть result

Передадим в функцию скользящее\_среднее() данные о собачках и окно сглаживания со значением K = 3600. Функция вернёт массив result, на основе которого построим сглаженный график:



Выглядит намного лучше! Теперь на графике можно разглядеть среднее количество сгенерированных собачек в любой момент времени в течение недели.

## **О пользе полуинтервалов**

В псевдокоде применено обозначение [a..b) для аналогии с полуинтервалами.

Полуинтервалом *[�;�)*[*a*;*b*) в математике обозначают такое множество чисел *�x*, где *�≤�<�a*≤*x*<*b*. Программисты обычно оперируют полуинтервалами при работе с индексами массивов. В коде удобно исключать правую границу *�b*, потому что:

* Элементы в массиве длины *�N* нумеруются индексами от *0*0 до *�−1N*−1: *0,1,2,…,�−1.*0,1,2,…,*N*−1. Для каждого элемента *��xi* выполняется условие *0≤�<�*0≤*i*<*N*. Индексы этого массива красиво записываются полуинтервалом *[0;�)*[0;*N*).
* Если из массива длины *�N* нам необходимо взять первые *�k* элементов, то этот массив можно естественно разбить на два подмассива — с индексами *[0;�)*[0;*k*) и *[�;�)*[*k*;*N*) соответственно.
* Легко вычисляется длина полуинтервала (то есть размер подмассива) *[�;�)*[*a*;*b*) — это просто разность *�−�b*−*a*.
* Запись *[�;�)*[*a*;*a*) валидна и обозначает пустой полуинтервал (то есть пустой подмассив).

Концепция полуинтервалов применяется в большинстве языков программирования:

var N = values.Length;

for (var i = 0; i < N; ++i)

// Пробежим по числам от 0 до N-1.

## **Скорость работы наивного алгоритма**

Давайте посмотрим, насколько эффективно работает функция скользящее\_среднее().

Пусть длина входного массива будет равна *�N*, а окно сглаживания — *�K*. В коде есть два вложенных цикла:

* внешний выполняет ровно *�−�+1N*−*K*+1 итераций;
* внутренний выполняет *�K* итераций на каждой из *�−�+1N*−*K*+1 итераций внешнего цикла.

Получается, общее число операций в циклах будет равно *�×�−�2+�N*×*K*−*K*2+*K*. Поскольку в нашем случае *�N* сильно больше *�K*, частью *(−�2+�)*(−*K*2+*K*) в этом выражении можно пренебречь и считать число операций примерно равным *�×�N*×*K*.

Если величина *�X* сильно больше величины *�,Y*, то это можно обозначить так: *�>>�.X*>>*Y*. Можно также сказать, что *�Y* сильно меньше *�X* с помощью обозначения *�<<�.Y*<<*X*. Это значит, что *�Y* настолько мало по сравнению с *�,X*, что в задаче можно считать величину *��XY* равной нулю. Из этого, в частности, следует, что *�−�X*−*Y* приблизительно равно *�.X*.

*�×�N*×*K* операций в функции — это много или мало?

Логи могут храниться довольно долгое время, и это не просто так: часто возникает необходимость посмотреть динамику данных за месяц, квартал или даже год. Допустим, нужно обработать данные за 30 дней с окном сглаживания в один час:

*�=30×24×60×60=2592000N*=30×24×60×60=2592000 — это длина входного массива.

*�=60×60=3600K*=60×60=3600 — окно сглаживания.

Количество операций можно посчитать так:

*�×�=9331200000N*×*K*=9331200000

Примерно столько операций придётся совершить, чтобы обработать данные за месяц. В зависимости от языка программирования такая программа будет выполняться от нескольких десятков секунд до нескольких десятков минут. А если увеличить размер окна, то ещё дольше.

# **Ускорение скользящего среднего**

## **Оптимизация алгоритма**

Наивный алгоритм из прошлого урока будет неплохо работать на небольших объёмах данных, но уже при обработке информации за месяц ему придётся выполнить около 9 миллиардов операций.

Напомним псевдокод функции:

функция скользящее\_среднее(timeseries, K):

result = [] # Пустой массив.

для begin\_index из [0 .. len(timeseries) - K]:

end\_index = begin\_index + K

# Просматриваем окно шириной K.

current\_sum = 0

для каждого v из timeseries[begin\_index .. end\_index):

current\_sum += v

current\_avg = current\_sum / K

result.добавить(current\_avg)

вернуть result

Обратите внимание, на соседних итерациях внешнего цикла суммируются практически одни и те же элементы за небольшим исключением.

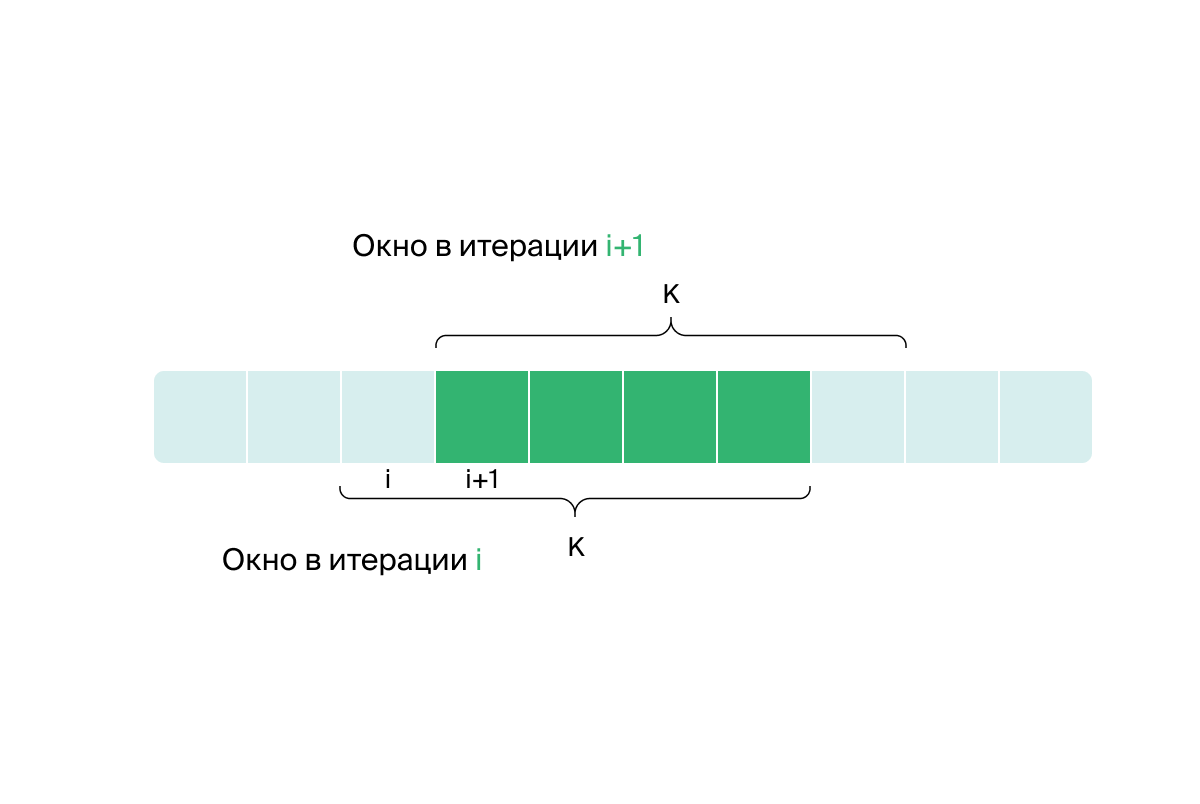
Рассмотрим итерацию *�+1i*+1. На предыдущей итерации *�i* мы вычислили значение переменной current\_sum как сумму элементов от *�i* до *�+�−1i*+*K*−1 включительно:

current\_sum = timeseries[i] + timeseries[i+1] + ... + timeseries[i+K-1]

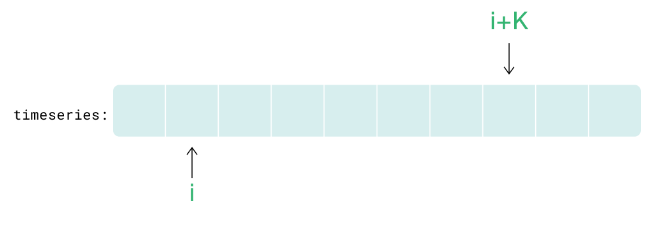
На текущей итерации *�+1i*+1 значение current\_sum считается как сумма элементов от *�+1i*+1 до *�+�i*+*K* включительно:

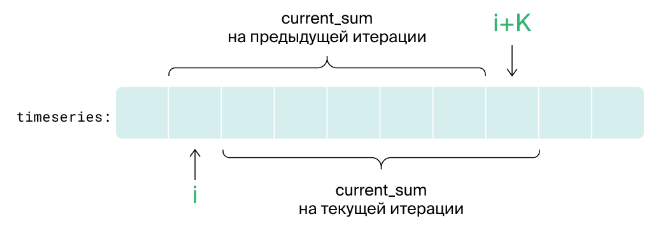
current\_sum = timeseries[i+1] + ... + timeseries[i+K-1] + timeseries[i+K]

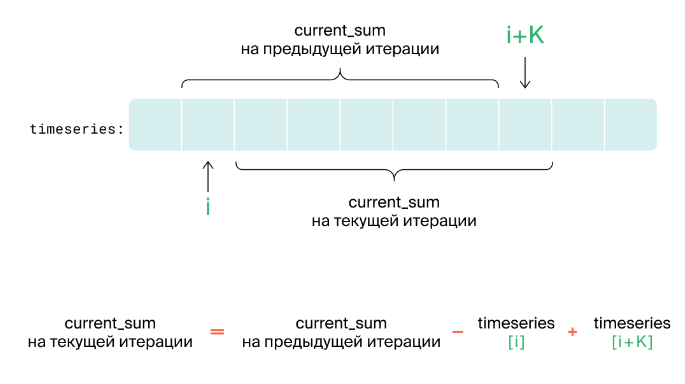
Сумма элементов с *�+1i*+1 по *�+�−1i*+*K*−1 пересчиталась повторно: значит, на последовательных итерациях значения current\_sum отличаются только за счёт элементов *�+�i*+*K* и *�i*.



Получается, что на каждой итерации (кроме первой) можно использовать результат предыдущей и, убрав вложенный цикл, оптимизировать наш алгоритм.







# Пусть в current\_sum записана сумма на прошлой итерации i.

# Вычислим её значение для текущей итерации i+1.

current\_sum -= timeseries[i]

current\_sum += timeseries[i+K]

Такой пересчёт возможен на любой итерации, кроме первой: на ней значение current\_sum придётся посчитать честно, просуммировав все элементы с *0*0 по *�−1K*−1.

Запишем оптимизированный алгоритм:

функция скользящее\_среднее(timeseries, K):

result = [] # Пустой массив.

# Первый раз вычисляем значение честно и сохраняем результат.

current\_sum = сумма элементов timeseries[0..K)

result.добавить(current\_sum / K)

для i из [0 .. len(timeseries) - K):

current\_sum -= timeseries[i]

current\_sum += timeseries[i+K]

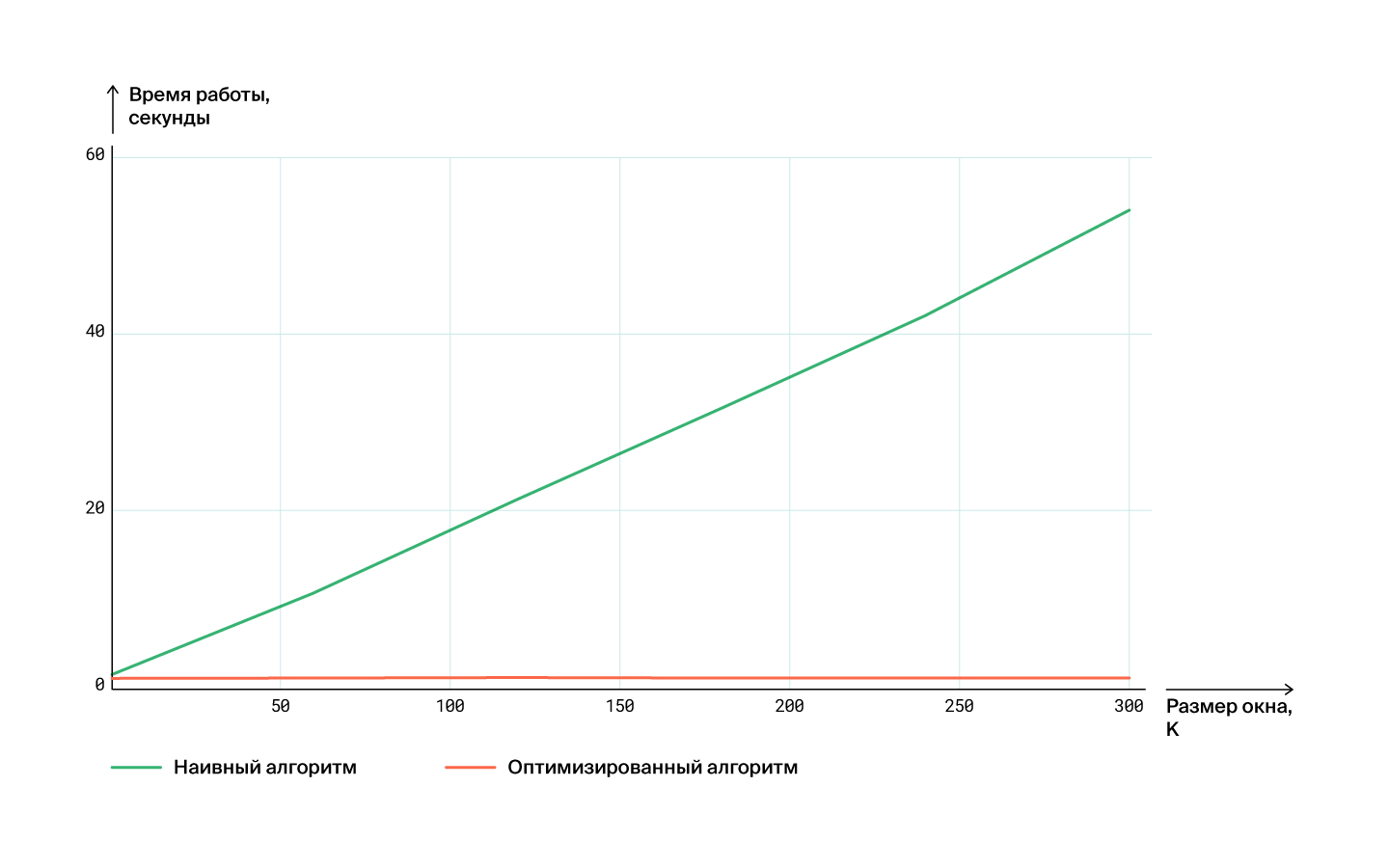
current\_avg = current\_sum / K

result.добавить(current\_avg)

вернуть result

## **Метод двух указателей**

Теперь, чтобы обработать данные за месяц, нашему алгоритму понадобится всего несколько секунд — при любом окне сглаживания. Вот сравнительный график, показывающий время работы обеих функций на данных за месяц в зависимости от размера окна *�K*.



При фиксированном N наивный алгоритм замедляется с увеличением K, а время работы второго не изменяется

Идея, которая помогла нам оптимизировать функцию, называется «метод двух указателей». В алгоритме действительно есть два указателя: они задают начало и конец рассматриваемого интервала. Чтобы пересчитать среднее арифметическое, мы:

* сначала изменяем его исходя из тех значений, на которые ссылаются указатели,
* а потом эти указатели смещаем.

Эта простая и эффективная техника. Она часто применяется для решения задач на алгоритмических собеседованиях.

## **2-SUM**

Задача, которую мы разберём в этом уроке, часто встречается на собеседованиях и называется 2-SUM, или TwoSum. Звучит она так: дан массив целых чисел numbers и целое число X; нужно найти в массиве два элемента, сумма которых равняется X. Гарантируется, что такие элементы в массиве есть.

Простой пример: Катя пришла в спортзал и хочет позаниматься на силовом тренажёре с нагрузкой в 16 килограммов. Рядом лежит груда стальных блинов. Девушка видит, что пудовых грузов в наборе нет, значит, одним блином не обойтись, а больше двух на тренажёр не установить.

Кате нужно найти два блина, которые в сумме дают 16 килограммов, и при этом не потратить на поиск всё отведённое на тренировку время. Гарантируется, что это выполнимая задача и такие два блина точно есть: об этом Кате рассказал её друг, который занимался на этом же тренажёре с той же нагрузкой.

## **Наивный алгоритм**

Начнём с наивного решения. Самое простое, что можно придумать, — перебрать все возможные пары блинов и для каждой из них проверить, равен ли её вес искомому.

Именно с наивного алгоритма рекомендуется начинать решение любой задачи, в том числе и на собеседованиях. Если от вас захотят получить более эффективное решение — вам скажут об этом явно.

Катя возьмёт первый блин и по очереди примерит его к остальным. Если ничего не подойдёт, вернёт первый блин на место, возьмёт следующий и повторит те же действия. Поиск будет продолжаться до тех пор, пока девушка не найдёт нужную пару блинов.

Однако даже в таком простом варианте есть нюансы. Например, пусть исходный массив будет [2,1,3,5,5], а X=4. В этом случае, если итерация будет происходить по индексам, алгоритм вернёт 1,3. Если же итерироваться по значениям, то легко допустить ошибку, при которой алгоритм вернёт 2,2, хотя второй двойки в массиве нет.

функция twosum(numbers, X):

для i из [0..len(numbers)):

для j из [i+1 .. len(numbers)):

если numbers[i] + numbers[j] == X, то:

вернуть numbers[i], numbers[j]

# По условию задачи пара обязательно должна найтись.

# Но предусмотрительность не помешает:

# если пара не найдена - вернём None, None (или можно выкинуть exception).

вернуть None, None

Внутренний цикл лучше начинать не с 0, а с i+1: если Катя выберет два блина, и они не подойдут, то нет смысла сравнивать те же блины в другом порядке. А значит, случаи j ≤ i точно не содержат решений, которых мы бы не видели на ранних итерациях. Кроме того, так проще писать код, потому что не придётся отдельно обрабатывать случай, когда i == j.

При удачном стечении обстоятельств Катя найдёт нужные блины с первой попытки. Если же искомая пара будет находиться ближе к концу, придётся сделать значительно больше операций.

## **Скорость работы наивного алгоритма**

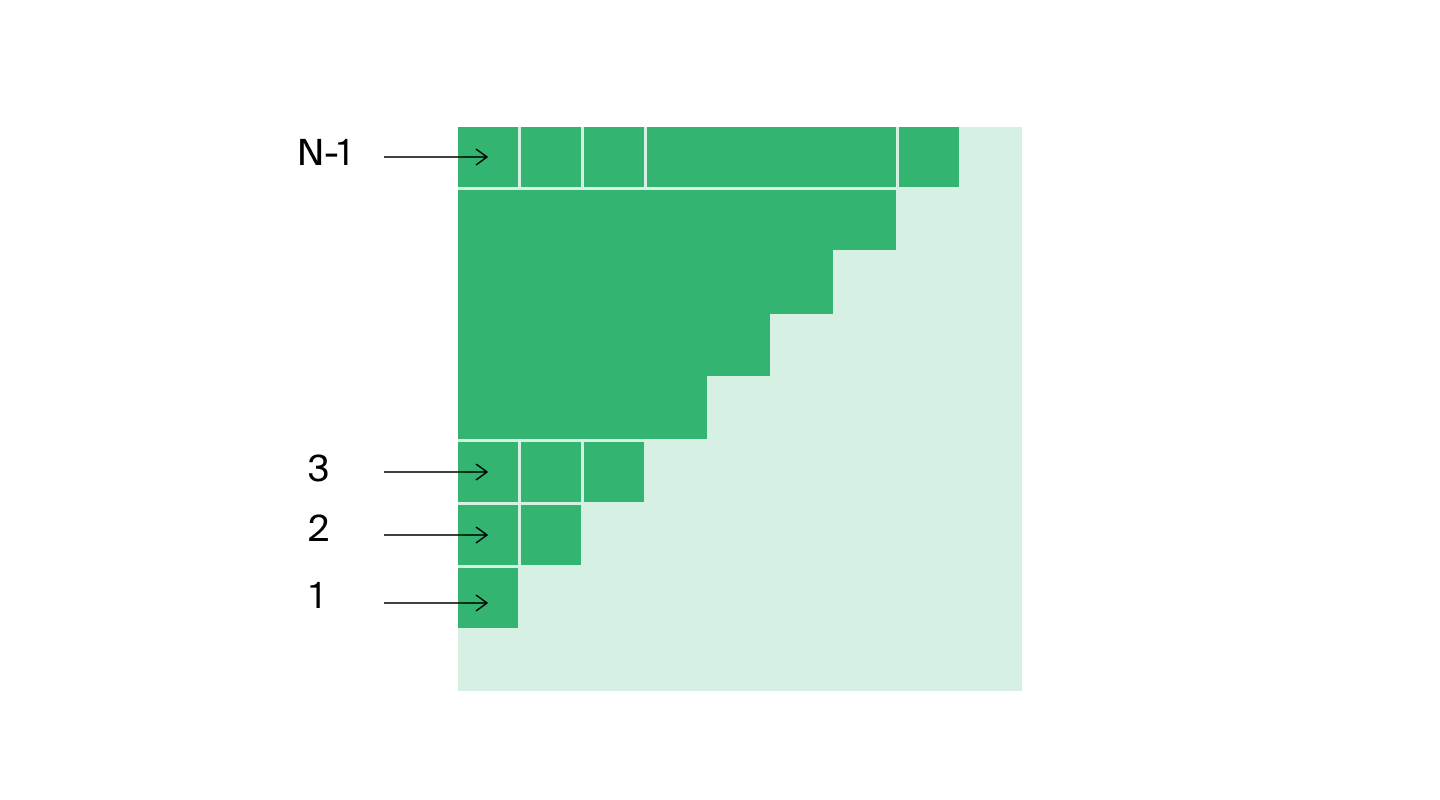
Сначала посчитаем максимально возможное количество операций.

На первой итерации Катя возьмёт первый блин и проверит, подходит ли к нему в пару какой-то из остальных. Их количество равно N - 1; в коде это соответствует первой итерации внешнего цикла: при i == 0 индекс j проходит значения от 1 до N - 1 включительно, в общей сложности N - 1.

На каждой следующей итерации i увеличивается на единицу, а значит, индекс j проходит на одну итерацию меньше, чем на предыдущей.

Получается, что общее число операций будет *(�−1)+(�−2)+(�−3)+…+2+1.*(*N*−1)+(*N*−2)+(*N*−3)+…+2+1. Давайте для наглядности представим эту информацию в виде схемы.

Каждая закрашенная клетка на иллюстрации — это одна операция сравнения. Количество таких полей составляет примерно половину площади квадрата со стороной *�.N*. Значит, в худшем случае Кате придётся сделать примерно *�22*2*N*2 попыток.

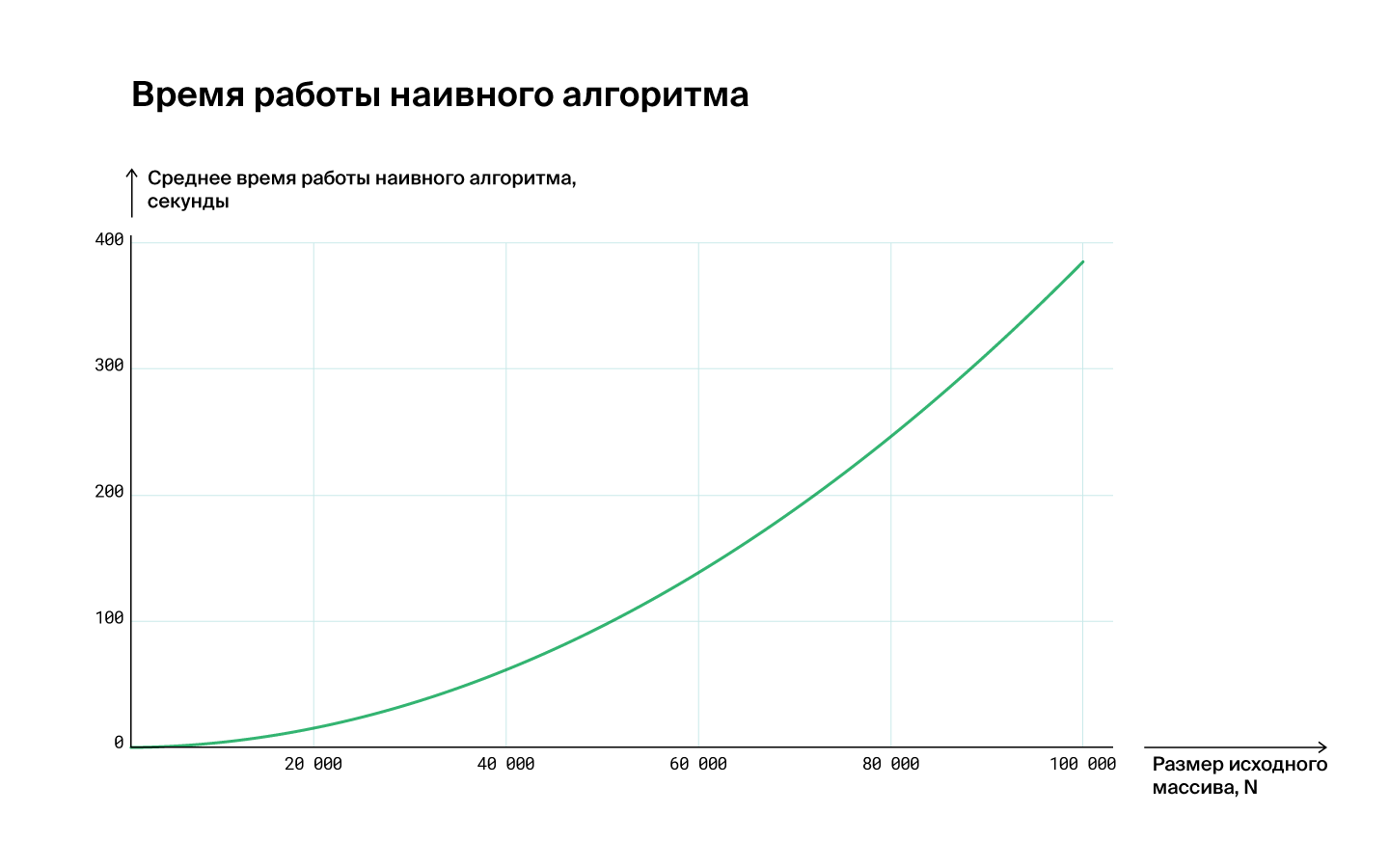


Если блины будут лежать как попало (говорят «в среднем случае» или просто «в среднем»), то Кате нужно будет сделать вдвое меньше сравнений, примерно *�24*4*N*2 . И в среднем, и в худшем случаях зависимость числа операций от числа элементов в массиве описывается квадратичной функцией.

Квадратичная функция — это функция вида *�(�)=��2+��+�f*(*x*)=*ax*2+*bx*+*c*, где *�,�,сa*,*b*,с — это числа. Название функции связано с тем, что главный элемент такой суммы — это *�2x*2 с некоторым коэффициентом. Главный он потому, что при больших *�x* остальные элементы окажутся сильно меньше и не сыграют заметной роли; в итоге значение функции будет приблизительно равно *��2ax*2.

В рассуждении выше *�24*4*N*2 — это квадратичная функция с коэффициентами *�=14,�=0,�=0a*=41 ,*b*=0,*c*=0.

График зависимости времени работы функции от числа элементов — это парабола, график квадратичной функции.



Кривая изгибается вверх — это показатель того, что время работы растёт быстрее, чем увеличиваются входные данные

Это значит, что если в массиве 1000 элементов, то результат посчитается мгновенно, 10 000 элементов обработаются за пару секунд, 100 000 элементов — за несколько минут, а результат для массива размера 1 000 000 мы будем ждать 10 часов.

## **Первый вариант оптимизации**

Предположим, что аккуратный тренер разложил все имеющиеся блины в порядке увеличения веса слева направо.

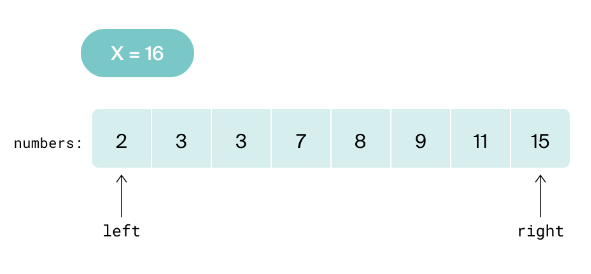
Катя посмотрела на ряд блинов, подумала — и взяла блины с двух краёв: самый лёгкий и самый тяжёлый.

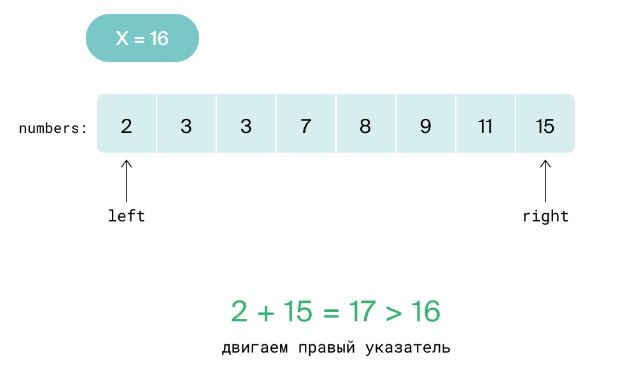
При сравнении блинов возможны три варианта:

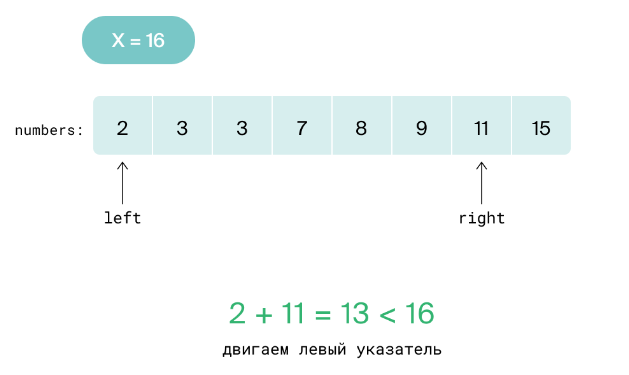
* Суммарный вес блинов в точности равен *�X*. Ура, ответ найден!
* Суммарный вес блинов больше необходимого. Заменяем один блин: тяжёлый откладываем, вместо него берём соседний с ним, меньшего веса.
* Суммарный вес блинов меньше, чем нужно. Заменяем один блин: лёгкий откладываем, вместо него берём соседний с ним, большего веса.

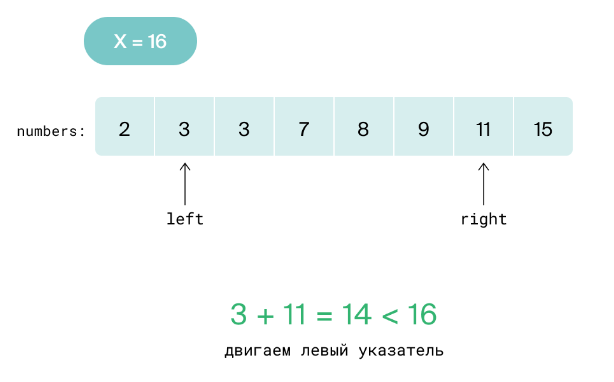
Катя будет выполнять эти действия на каждом последующем шаге до тех пор, пока не найдёт нужную пару.

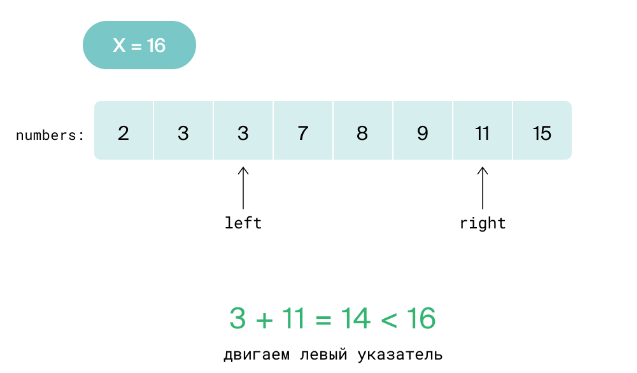
Это снова метод двух указателей: один из них указывает на левый блин, а второй — на правый. На каждом шаге либо левый указатель сдвигается на один элемент вправо, либо правый — на один элемент влево. Указатели двигаются не в одном направлении, как в истории с окном сглаживания, а сходятся, поочерёдно двигаясь навстречу друг другу. Если в какой-то момент указатели встретятся — значит, искомая пара элементов в массиве отсутствует.

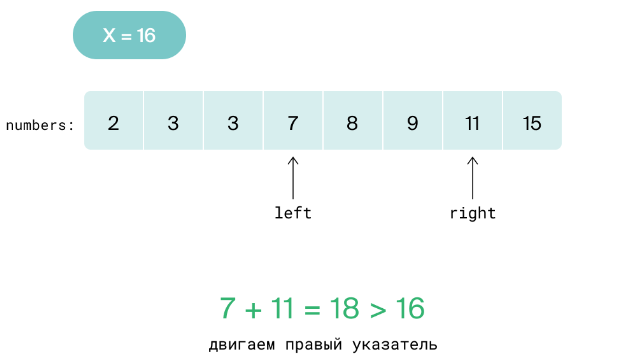


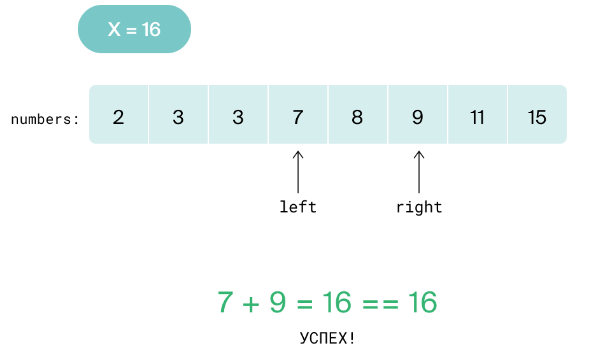












Исключённые при сравнении блины не могут образовать нужную пару, поэтому возвращаться к ним нет смысла.

Оценить число сравнений в таком алгоритме очень просто: расстояние между указателями на старте равняется *�−1N*−1 и на каждом шаге оно сокращается ровно на *1*1. В результате или найдётся искомая пара, или указатели встретятся. Значит, в худшем случае придётся проделать не более чем *�−1N*−1 операций. Это серьёзный выигрыш в сравнении с наивным алгоритмом, который в худшем случае обречён выполнить *�22*2*N*2 операций.

Правда, чтобы оптимизированный алгоритм заработал, нужно сначала отсортировать исходный массив. А вот наивный алгоритм из прошлого урока упорядоченности не требует. Но даже с учётом сортировки оптимизированный алгоритм будет работать быстрее наивного.

функция twosum\_with\_sort(numbers, X):

# Сортируем исходный массив стандартной функцией.

sort(numbers)

left = 0

right = len(numbers) - 1

пока left < right:

current\_sum = numbers[left] + numbers[right]

если current\_sum == X, то:

вернуть numbers[left], numbers[right]

если current\_sum < X, то:

left += 1

иначе:

right -= 1

# Если ничего не нашлось в цикле, значит, нужной пары элементов в массиве нет.

вернуть None, None

## **Второй вариант оптимизации**

Если не хочется сортировать тяжёлые блины, можно использовать для решения этой задачи дополнительную память. Например, позвать тренера, который наизусть помнит вес всех блинов в спортзале. В этом случае алгоритм будет такой: можно в любом порядке перебирать блины и, вытащив очередной (допустим, с весом *�A*), спрашивать: «А есть ли у вас блин с весом *�−�X*−*A*?». Когда консультант ответит «Да, есть такой» — можно оставить себе текущий блин, взять тот, что с весом *�−�X*−*A*, и пойти заниматься.

В этой ситуации важно, чтобы тренер действительно помнил вес блинов на память, а не бегал каждый раз проверять наличие нужного блина — в таком случае скорость не будет отличаться от наивного алгоритма. Метод с «дополнительной памятью» будет работать быстро, но у него есть недостаток: тренер может попросить денег за свои услуги.

В программировании это означает, что для решения задачи можно использовать вспомогательную структуру, которая называется «структура данных поиска». Она умеет быстро наполняться содержимым и эффективно отвечать на вопросы вроде «Есть ли у тебя элемент *�Y*?». Вероятно, вы с ней уже встречались: она обычно называется set, map, dict или содержит в своём названии какое-нибудь из этих слов.

Алгоритм, применяющий структуру данных поиска, может выглядеть так:

функция twosum\_extra\_ds(numbers, X):

# Создаём вспомогательную структуру данных с быстрым поиском элемента.

previous = set()

для каждого A из numbers:

Y = X - A

если Y уже лежит в previous, то:

вернуть A, Y

иначе:

добавить A в previous

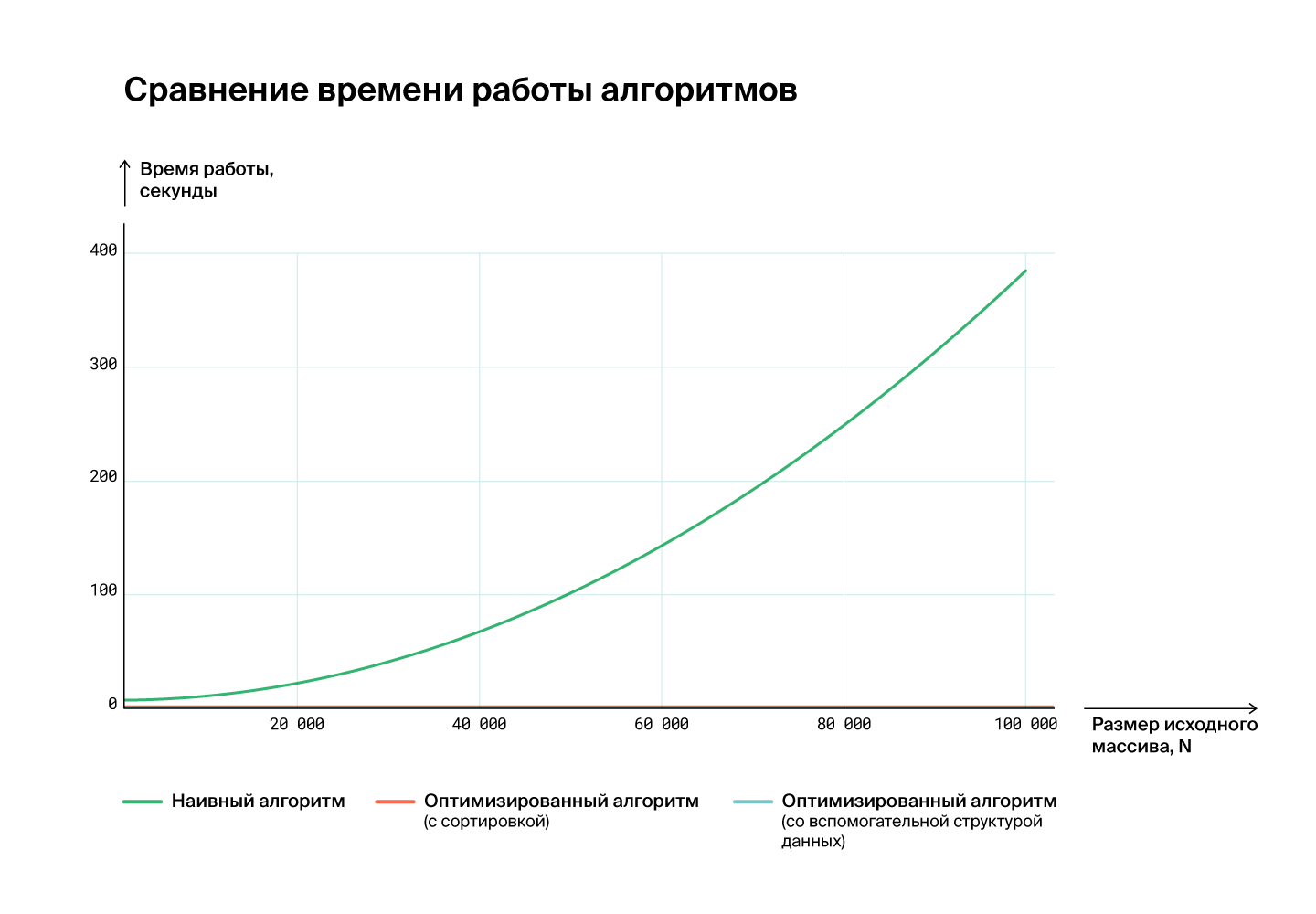
# Если ничего не нашлось в цикле, значит, нужной пары элементов в массиве нет.

вернуть None, None

Заплатить за хранение элементов в структуре данных поиска придётся, конечно, не деньгами, а оперативной памятью: в этом отличие нового алгоритма от двух предыдущих. Это решение очень быстрое, но не во всех ситуациях оно будет лучшим.

## **Все вместе**

Итак, для решения задачи 2-SUM мы нашли наивное решение и два более эффективных алгоритма. Теперь можно примерно оценить время их работы в зависимости от числа элементов *�N* и построить общий график.



Кажется, что картинка не поменялась от добавления двух алгоритмов, где же время их работы? На самом деле эффективные алгоритмы работают настолько быстрее наивного, что их графики сливаются с осью времени. Даже для *�=100000N*=100000 время их работы будет меньше *0.1*0.1 секунды. Так что для сравнения оптимизированных алгоритмов нужно убрать с графика наивный алгоритм и поменять масштаб:

