**Анализ сложности алгоритмов: введение**

В предыдущем спринте вы [тренировались решать задачи для собеседований](https://praktikum.yandex.ru/learn/web/courses/134735fe-bc3e-4772-b04d-206235915714/sprints/1704/topics/801fc3d9-e081-4184-a44f-b44fd8c4af1e/lessons/d0260867-750d-4aac-b700-d036e59b47d1/). Может, даже зарегистрировались на [Codewars](https://www.codewars.com/) и испытали себя там. По формату это упражнения «из задачника», но ведь любой код — тоже в какой-то степени задача.

К решению можно подойти по-разному. Чтобы понять, какой путь эффективнее, алгоритм решения анализируют: оценивают его вычислительную сложность или, по-научному, проводят асимптотический анализ сложности. О том, как именно это сделать, расскажем в следующих уроках.

Это непростая тема, и мы не включили задания по ней в практическую работу. Но знание алгоритмов — одна из тех вещей, по которым можно распознать профессионального веб-разработчика. Не страшно, если какой-то из уроков будет даваться с трудом: к ним всегда можно вернуться и перечитать. Главное — начать разбираться.

# Оценка сложности алгоритма: асимптотический анализ

Для производительных программ важно подбирать эффективный алгоритм. Но что он должен делать, чтобы мы могли оценить его эффективность: вставать в 5:30, готовить себе на завтрак свежепророщенную зелёную гречку и отправляться бежать полумарафон с подкастом «Запуск завтра» в эйрподсах? Не совсем. Давайте разбираться вместе.

По двум параметрам:

* время выполнения (временна́я сложность алгоритма);
* расход памяти (пространственная сложность алгоритмов).

Чем дольше работает алгоритм и больше памяти потребляет, тем он хуже. Но параметры алгоритма зависят от устройства, на котором он выполняется, и количества входных данных, поэтому неверно утверждать, что «алгоритм работает 1 секунду и задействует 1 мегабайт памяти». Один и тот же алгоритм может работать на старом компьютере 10 секунд, а на современном — 1 секунду.

Из-за таких погрешностей применяют другой метод оценки производительности алгоритма — асимптотический анализ. Он не зависит от устройства, но учитывает размер данных, которые загрузили в программу на вход.

Когда говорят «сложность алгоритма», обычно имеют в виду временнýю сложность. На её примере рассмотрим, как работает асимптотический анализ.

Возьмём простую функцию и рассчитаем временную сложность:

Скопировать кодJAVASCRIPT

function sum(numbers) {

let sum = 0;

for (let i = 0; i < numbers.length; i += 1) {

sum += numbers[i]

}

return sum;

}

Она оценивает время работы программы. Чем больше операций нужно выполнить — тем дольше программа будет работать.

Выпишем все операции в функции sum и посчитаем их общее число:

* let sum = 0 — создание переменной,
* let i = 0 — создание переменной,
* i < numbers.length — проверка условия,
* i += 1 — увеличение переменной на единицу,
* numbers[i] — доступ к элементу массива,
* sum += numbers[i] — увеличение переменной sum.

Последние четыре операции находятся в цикле и выполняются конкретное число раз — numbers.length. Посчитаем сумму всех операций. 4 операции в цикле нужно умножить на количество итераций цикла (numbers.length) и прибавить 2 первые операции в функции: 2 + 4 \* numbers.length. При длине массива n получаем выражение 2 + 4n.

А так изменяется выражение при разных значениях n:

| **N** | **2 + 4N** | **Δ** |
| --- | --- | --- |
| 1 | 6 |  |
| 10 | 42 | 7 |
| 100 | 402 | 9.5714 |
| 1000 | 4002 | 9.9552 |
| 10000 | 40002 | 9.9955 |
| 100000 | 400002 | 9.9995 |
| 1000000 | 4000002 | 9.9999 |

В первом столбце таблицы записаны разные значения n. Во втором –– как меняется количество операций в абсолютном выражении. А в третьем столбце указано, как меняется количество операций в относительном выражении по отношению к предыдущему значению.

При изменении n в 10 раз, изменение функции 2+4n также стремится к 10. Например, при n=10, 2+4n=42, а при n=100, 2+4n=402. В относительном выражении –– 402 / 42 = 9.571. Темп роста функции 2+4n очень близок к функции n, особенно на больших значениях. Так происходит потому, что основной вклад в рост функции вносит самый значимый её член. Здесь это n.

С помощью асимптотического анализа мы оцениваем, как сложность алгоритма растёт с ростом входных данных. Важен только характер изменения функции, поэтому мы опускаем все константные коэффициенты и члены, кроме значимого.

Функция 2+4n даёт сложность О(n). Читается как «О большое от n». Записать можно так: 2 + 4n = О(n), или О(2 + 4n)= О(n). Значит, что эта функция растёт не быстрее, чем функция n, умноженная на константу. Что правда, ведь обе функции линейные.

Здесь мы применили методику асимптотического анализа. Результат записали с помощью О-нотации.

Чтобы потренироваться находить О большое, рассмотрим ещё три выражения. Вы можете самостоятельно определить характер их роста, подставив разные значения n.

10 можно представить как 10\*n^0. Сложность не меняется от размера входных данных, поэтому она будет О(n^0) или О(1).

Функция n^2 растёт быстрее, чем n, поэтому n^2 — самый значимый член. Сложность равна O(n^2).

Какая сложность функции 10n+50?



О(10n)

Нет. Нужно определить самый значимый член в функции и опустить коэффициент. Это и есть сложность.



О(n)

Верно. 10n+50=O(n).



О(n^2)



О(50)

Какая сложность функции n + n^2 / 1000?



О(log(n))



О(n)



О(n^2)

Верно. n^2 растёт быстрее, чем n.



О(n\*log(n))

Какая сложность функции 2^n + n^3?



О(2^n)

Верно. C ростом n 2^n растёт быстрее, чем n^3.



О(n)



О(n^3)

Нет. Нужно определить самый значимый член в функции. Это 2^n, так как с ростом n 2^n растём быстрее, чем n^3.



О(1^n)

Ещё раз запишем функцию sum и посчитаем пространственную сложность:

Скопировать кодJAVASCRIPT

function sum(numbers) {

let sum = 0;

for (let i = 0; i < numbers.length; i += 1) {

sum += numbers[i]

}

return sum;

}

Пространственная сложность оценивает объём памяти, который занимает алгоритм. Считать его в мегабайтах не нужно, просто оценим, как изменяется количество хранимой информации с изменением входных данных.

Посчитаем все переменные, созданные в функции sum:

* let sum = 0 –– переменная для хранения суммы;
* let i = 0 –– переменная для хранения индекса.

Мы создали всего две переменные. Пространственная сложность этого алгоритма –– О(1). Она не зависит от входных данных –– массив любой длины будет расходовать одинаковый объём памяти.

В этом уроке при анализе сложности мы подсчитывали операции внутри функции sum. Но иногда за одной операцией скрывается гораздо больше. В следующем уроке научимся выявлять и анализировать такие операции.

# Непростые операции

Элементарные операции не зависят от структуры и объёма данных, с которыми работают. К ним относят присвоение значений переменным (a = 5), математические операции (2\*2), обращение к полям объекта (obj.a), проверка логических условий (a < 5). Все они работают достаточно быстро, поэтому при расчёте сложность таких операций принимают за единицу и просто считают их количество.

Но бывают операции, сложность которых не равна единице, например, вызов функции. Методы map и join перебирают элементы массива, поэтому имеют сложность О(n). Если мы используем их в коде, то должны учитывать это при расчётах. Оценим сложность функции:

Скопировать кодJAVASCRIPT

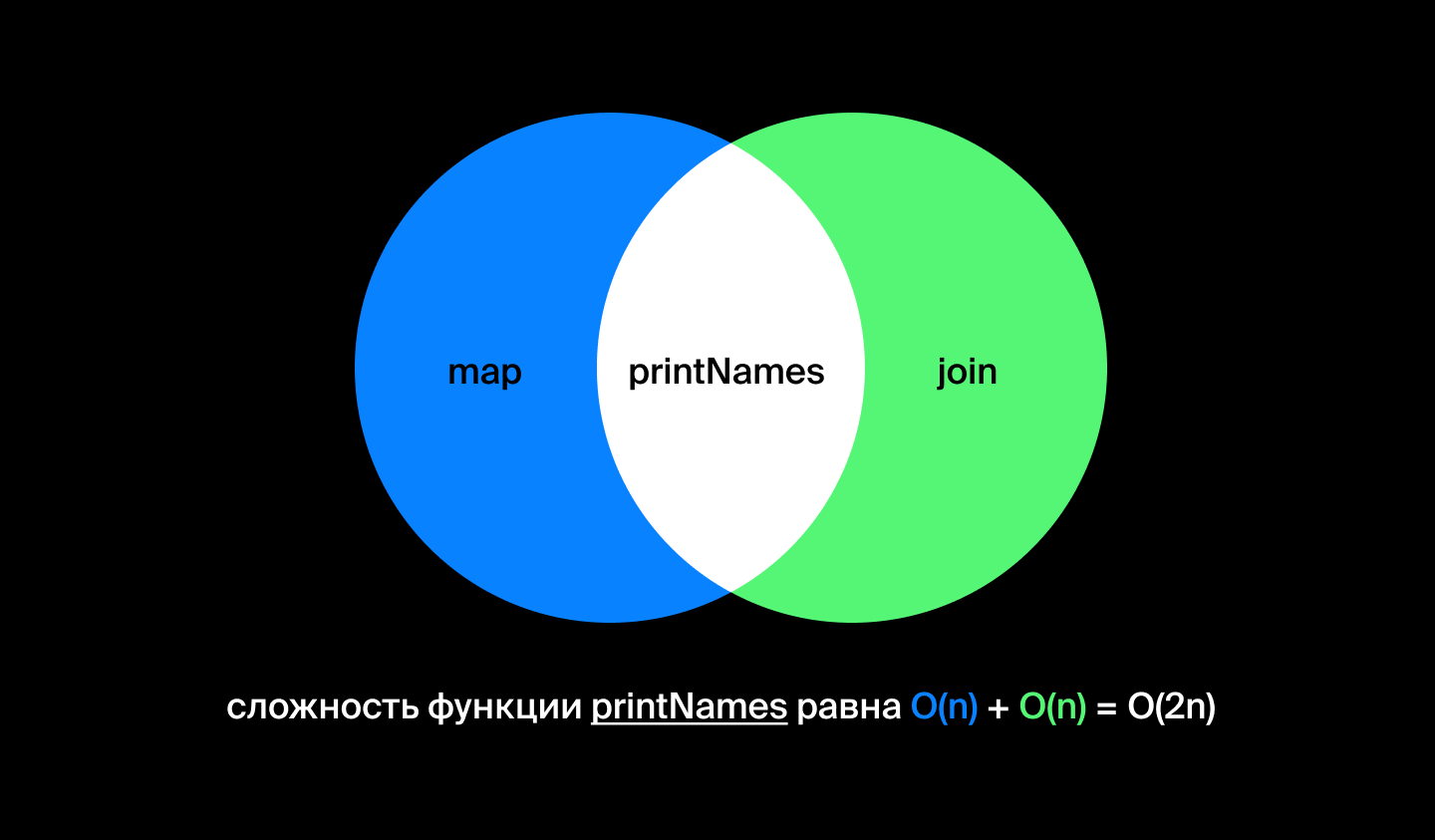
function printNames(people) {

const names = people.map(p => p.name); *// О(n), считаем как n операций*

return names.join(', '); *// О(n), считаем как n операций*

}

Сложность функции printNames равна сумме сложностей двух вызываемых внутри неё функций: map и join.



Сложность операции join зависит только от длины массива, поэтому равна О(n). Сложность операции map зависит от размера массива и от количества операций внутри функции-колбэка. Здесь сложность равна О(n \* 1), ведь в теле колбэка метода map мы совершаем всего одну операцию.

Итоговая сложность функции printNames равна О(n) + О(n) = О(2n). Опустим константу 2 и получим сложность О(n).

В работе не нужно всегда считать все операции. Достаточно суммировать сложность операций, которые зависят от размера входных данных, а элементарные операции опустить.

Проверим. Усложним код из предыдущего примера:

Скопировать кодJAVASCRIPT

function printNames(people) {

const names = people.map(p => {

if (p.fullName) {

return p.fullName;

} else {

return [p.firstName, p.middleName, p.lastName].join(' ');

}

});

return names.join(', ');

}

Операций внутри функции-колбэка map стало больше, но они не зависят от размера входных данных. Какой бы длинный массив people мы не передали, количество операций в колбэке метода map не изменится, поэтому не нужно считать их точное число. Примем их за константу k и получим сложность функции map: k \* О(n)=О(n). В итоге вышло О(n), как и в алгоритме с меньшим количеством операций.

Рассчитайте сложность функции swap.

Скопировать кодJAVASCRIPT

function swap(arr, i, j) {

const tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

}



О(n^3)



О(n)

Неверно. Как думаете, зависит количество операций от размера массива и переменных? Нет. Функция swap выполняет 3 операции, независимо от объёма входного массива и переменных i и j. Поэтому её сложность константная –– О(1).



О(n^2)



О(1)

Верно! Функция swap выполняет 3 операции, независимо от размера массива arr и индексов i и j. Поэтому её сложность константная –– O(1).

Рассчитайте сложность функции findMinIndex.

Скопировать кодJAVASCRIPT

function findMinIndex(arr, start) {

let min = arr[start];

let minIndex = start;

for (let i = start + 1; i < arr.length; i++) {

if (arr[i] < min) {

min = arr[i];

minIndex = i;

}

}

return minIndex;

}



О(n^3)



О(n)

Верно! Функция findMinIndex выполняет цикл по элементам массива arr и поэтому линейно зависит от его размера.



О(n^2)



О(1)

Неверно. Обратите внимание на цикл по элементам массива. Какая это сложность?

# Некоторые функции сложности

В этом уроке рассмотрим несколько основных функций сложности. К каждой из них мы подобрали пример кода — так станет понятнее.

## Константная сложность О(1)

Сложность алгоритма не зависит от входных данных. Для примера возьмём арифметическое действие:

Скопировать кодJAVASCRIPT

function div(a, b) {

return (a - (a % b)) / b;

}

Объём работы здесь не зависит от значений a и b .

## Логарифмическая сложность О(log(n))

Сложность растёт логарифмически –– на каждом шаге мы в несколько раз уменьшаем количество обрабатываемых данных. В информатике часто работают с логарифмами по основанию два, то есть уменьшают объём данных вдвое. Но на сложность алгоритма основание не влияет.

Рассмотрим бинарный поиск индекса элемента по отсортированному массиву. На каждом шаге мы берём элемент из середины отсортированного массива и проверяем, равен ли он искомому. Если элемент больше, продолжаем искать в левой части. Если меньше –– в правой. И так до тех пор, пока не найдём нужный элемент.

Скопировать кодJAVASCRIPT

function binarySearch(sortedNumbers, n) {

*// Определяем точки начала и конца поиска*

let start = 0;

let end = sortedNumbers.length;

while (start < end) {

*// Находим элемент в середине массива*

const middle = Math.floor((start + end) / 2);

const value = sortedNumbers[middle];

*// Сравниваем аргумент со значением в середине массива*

if (n == value) {

return middle;

}

*// Если аргумент меньше, чем серединное значение, разделяем массив пополам*

*// Теперь конец массива — это его бывшая середина*

if (n < value) {

end = middle;

*// Иначе началом массива становится элемент, идущий сразу за «серединой»*

} else {

start = middle + 1;

}

}

*// если искомое число не найдено, возвращаем -1*

return -1;

}

При каждой итерации массив делится пополам. Если изначально длина массива — 1024, то на второй итерации элементов останется 512, затем — 256, потом 128 и так далее.

То есть, для массива из 1024 элементов понадобится максимум log2(1024) = 10 шагов. И число шагов будет расти очень медленно по сравнению с размером массива. Например, чтобы найти число в отсортированном массиве размером в 1000 раз больше, понадобится всего в 2 раза больше шагов log2(1024000) = 19.96.

## Линейная сложность О(n)

Эту сложность мы получаем, когда создаём цикл по массиву элементов или вызываем метод, который перебирает массив. Она растёт прямо пропорционально количеству данных.

Линейную сложность применяют, например, когда надо найти минимум (или максимум) в несортированном массиве. Для начала идём по массиву и сравниваем числа. Если встречаем больше или меньше записанного минимума и максимума, перезаписываем их на новые.

Скопировать кодJAVASCRIPT

function minMax(numbers) {

*// Присваиваем переменным min и max первый элемент массива*

let min = numbers[0];

let max = numbers[0];

for (let i = 1; i < numbers.length; i++) {

const n = numbers[i];

*// Сравниваем элемент с min*

if (n < min) {

min = n;

}

*// Сравниваем элемент с max*

if (n > max) {

max = n;

}

}

*// Возвращаем найденную пару значений*

return { min, max };

}

Количество операций линейно зависит от длины массива: если массив увеличится вдвое, то и количество операций тоже вырастет в два раза.

## Квадратичная сложность O(n^2)

Квадратичная сложность растёт быстро: при увеличении данных в 100 раз, объём вычислений вырастет в 10000 раз. На больших данных алгоритм с такой сложностью работает медленно.

Пример –– поиск всех комбинаций элементов из двух массивов. Для каждого элемента из первого массива мы пробегаем по всему второму массиву и складываем все пары в результирующий массив.

Скопировать кодJAVASCRIPT

function combinations(arr1, arr2) {

*// Создаём массив для результата*

const result = [];

*// Запускаем вложенные циклы и формируем все возможные пары*

for (let i = 0; i < arr1.length; i++) {

for (let j = 0; j < arr2.length; j++) {

result.push([arr1[i], arr2[j]]);

}

}

return result;

}

Квадратичная сложность — частный случай полиномиальной — O(n^k). Если цикл вложен в другой цикл, а тот — в третий, зависимость будет кубической — O(n^3. Если 4 цикла вложены друг в друга — O(n^4), и так далее.

Бывают и другие функции сложности, но здесь мы привели только основные. Если не терпится познакомиться с остальными функциями сейчас, — ищите дополнительные материалы в следующем уроке.

**Где узнать больше про алгоритмы?**

Вы уже поняли, что оценка сложности алгоритма — это непросто. Многим работодателям не нужно, чтобы вы в ней разбирались. Но если когда-то решите пойти на собеседование в Яндекс или другую крупную компанию, рекомендуем повторить эту тему и пройтись по дополнительным материалам:

* [лекция по алгоритмам и структурам данных от Яндекса](https://www.youtube.com/watch?v=ijwbVxLMp58&feature=youtu.be),
* [статья про оценку сложности в блоге Hackernoon](https://hackernoon.com/big-o-for-beginners-622a64760e2),
* [ещё одна статья на ту же тему от Freecodecamp](https://www.freecodecamp.org/news/my-first-foray-into-technology-c5b6e83fe8f1/).

Надеемся, теперь оценка сложности алгоритмов станет чуть менее сложной.