



# Алгоритмы



Артём Трофимушкин

# Что такое алгоритм?

---

Алгоритм — это **последовательность шагов**, которая решает определенную задачу. Иными словами алгоритм — это способ решения этой задачи.

В программировании алгоритм, как правило, имеет **входные данные**, над которыми производятся **вычисления**, и выходной **результат**. По сути задача алгоритма состоит в преобразовании входных значений в выходные.



# Эффективность алгоритма

---

Важным критерием алгоритма выступает **эффективность**. Алгоритм может прекрасно решать поставленную задачу, но при этом быть не эффективным. Как правило, под эффективностью алгоритма подразумевается **время работы**, т.е. время преобразований данных.

Но время работы, например в секундах, всегда относительно – оно может быть разным на разных компьютерах, разных ОС, оно может зависеть от количества оперативной памяти, частоты и разрядности процессора.

В связи с этим, эффективность алгоритма часто измеряют **функцией, зависящей от количества элементарных операций процессора**. В таком виде алгоритмы можно сравнивать даже не запуская их на компьютере.



# Асимптотический анализ алгоритмов

Асимптотическое поведение — это производительность алгоритма **при росте размера задачи**. Часто размер задачи обозначается как  **$N$** . Чтобы описать асимптотическое поведение, нужно ответить на вопрос — **что случится** с производительностью алгоритма, **если  $N$  сильно вырастет?**

Для представления временной сложности алгоритмов в основном используют три асимптотических нотации:

- **$O$**  (нотация **о большое**) - представляет наихудший порядок сложности,
- **$\Omega$**  (нотация **омега большое**) - представляет наилучший порядок сложности,
- **$\Theta$**  (нотация **тета большое**) - описывает порядок сложности, когда наихудший и наилучший случаи пересекаются.



# Подсчет количества операций

```
int x = 0;  
x = x + 1;
```

В приведенном коде на первый взгляд 2 команды, но количество операций будет бóльшим:

- **int x = 0** : инициализация переменной состоит из **2 операций**:
  - создать локальную переменную
  - записать в нее значение 0
- **x = x + 1** : присвоение значения переменной состоит также из **2 операций**:
  - вычислить значение по формуле  $x + 1$
  - записать его в переменную  $x$



# Магия IL DASM

## Сколько же операций на самом деле?

IL DASM: "C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.7.1 Tools\ildasm.exe"

```
// int x = 0;
IL_0001: ldc.i4.0          // кладем 0 как int на вершину стека.
IL_0002: stloc.0          // кладем значение вершины стека
                          // в локальную переменную под номером 0.

// x = x + 1;
IL_0003: ldloc.0          // достаем значение локальной переменной
                          // под номером 0 и кладем его на вершину стека.
IL_0004: ldc.i4.1          // кладем 1 как int на вершину стека
IL_0005: add              // сейчас в стеке лежит 2 значения: 0, который
                          // мы достали из переменной x и 1, взятая из
                          // выражения x + 1; над ними выполняется
                          // операция сложения; результат кладётся на
                          // вершину стека.
IL_0006: stloc.0          // берем с вершины стека результат сложения
                          // и записываем его в локальную переменную 0.
```

// Детали по командам IL можно найти здесь: [https://en.wikipedia.org/wiki/List\\_of\\_CIL\\_instructions](https://en.wikipedia.org/wiki/List_of_CIL_instructions)



# Расчет **сложности** алгоритма

---

При расчете сложности алгоритмов в виде асимптотических нотаций  $O$  и  $\Omega$  делаются следующие 2 упрощения:

1. Все операции, не зависящие от переменных факторов сводятся к 1,
2. В расчет идут только функции высшего порядка.



# Расчет $O$ большого

$O$ -нотация описывает **наихудший порядок сложности** нашего кода.

Когда мы говорим “худший”, мы имеем в виду “худший при разных значениях переменного фактора или переменных величин”.

Поскольку в нашем примере нет переменных величин (4 — это константа), то это просто 4.

Применяем правило асимптотических нотаций “Все операции, не зависящие от переменных факторов сводятся к 1”.

Таким образом, наша 4 превращается в 1

Запись сложности в нотации “о большое” будет выглядеть вот так:  $O(1)$ .

Можно сказать, что **ЭТОТ КОД В ВЫПОЛНИТСЯ “за  $O(1)$ ”**.





# Расчет Омега большого

---

$\Omega$ -нотация описывает **наилучший порядок сложности** нашего кода.

Поскольку мы только что выяснили, что переменных факторов у нас нет, наихудший порядок также сводится к единице.

В лучшем случае этот код в выполнится “**за омега 1**”, т.е.,  $\Omega(1)$ .



# Расчет Тета большого

В случае, когда нотации  $O$  и  $\Omega$  одинаковы, удобно использовать  $\Theta$ -нотацию (тета большое). Ее нельзя посчитать самостоятельно (отдельно), она требует предварительного расчета  $O$  и  $\Omega$ .

$\Theta$ -нотация дает больше информации о сложности алгоритма, чем просто  $O$ - или  $\Omega$ -нотация, так как в ней сразу заключено знание о том, что это и  $O$  и  $\Omega$  одновременно.

Для нашего примера, действительно,  $O$ - и  $\Omega$ -нотации посчитаны и равны, следовательно запись  $\Theta$ -нотации будет выглядеть так:  $\Theta(1)$ .

Ещё раз про  $\Theta$ : можно обойтись и без  $\Theta$ -нотации, можно просто написать  $O$ - и  $\Omega$ -нотации – при первом взгляде будет и так видно, что они равны, однако использование  $\Theta$ -нотации даёт более лаконичную и понятную запись этой информации.



# Важно понимать

---

Обратите внимание,  $X$  в данном случае **не является переменным фактором** алгоритма (хотя и является переменной с точки зрения программиста), так как при **при любых значениях  $X$**  в нашем коде из двух строк **будет одно и то же количество операций**.

То, что мы записали, не означает, что наш алгоритм работает за 1 операцию!

Если бы в нашем алгоритме было 100 операций или 1 000 000 операций – он занимал бы ощутимое время!

Асимптотическая запись  $\Theta(1)$  говорит лишь о том, что сложность нашего алгоритма – **это постоянная величина**. Т.е. количество операций не меняется в зависимости от исходных данных.

**С практической точки зрения**, это означает, что если наш алгоритм и время его работы устраивает всех в текущем виде, можно не бояться, что мы в будущем получим проблемы производительности. Наш код всегда будет выполняться примерно за это время.



# Когда уместно $O$ , а когда $\Omega$ ?

---

Чаще всего пользуются  $O$ -нотацией – она практически более полезна, так как отвечает на вопрос “Как в худшем случае поведет себя алгоритм?”

$\Omega$ -нотация отвечает на вопрос “Насколько сложен алгоритм при самом благоприятном стечении обстоятельств?” и может быть полезна, например, в случае, когда необходимо доказать, что алгоритм имеет слишком высокий порядок сложности и мог бы быть оптимизирован.



# Реализация алгоритма на C#

```
const int length = 5;           // длина массива
const int maxValue = 100;       // максимальное значение элемента

var arr = new int[length];      // создаем массив с размером равным length
var rnd = new Random();          // создаем объект генератора случайных чисел

for (var i = 0; i < arr.Length; i++) // перебираем каждый элемент массива
{
    arr[i] = rnd.Next(maxValue);    // заполняем его произвольным значением
                                    // меньше maxValue

// Выводим на экран результат
for (var i = 0; i < arr.Length; i++)
{
    Console.WriteLine(arr[i]);
}
```



# Реализация алгоритма на C#

```
// перебираем массив по j, не доходя до последнего элемента
// до него мы доберемся через выражение j + 1
int limit = arr.Length - 1;

for (int j = 0; j < limit; j++)
{
    // сравниваем текущий и последующий элементы
    // если текущий больше последующего, меняем их местами
    if (arr[j] > arr[j + 1])
    {
        int temp = arr[j + 1]; // обмен значений
        arr[j + 1] = arr[j];   // двух переменных
        arr[j] = temp;         // через третью
    }
}
```



# Реализация алгоритма на C#

Затем усложняем этот код, накручивая сверху второй массив, чтобы все элементы заняли нужные места. При каждой итерации лимит будет уменьшаться на 1:

```
// i нам нужна уже не для доступа к массиву, а всего лишь
// для уменьшения лимита внутреннего цикла
for (int i = 0; i < arr.Length - 1; i++)
{
    // перебираем массив по j, не доходя до последнего элемента
    // до него мы доберемся через выражение j + 1
    int limit = arr.Length - 1;

    for (int j = 0; j < limit; j++)
    {
        // сравниваем текущий и последующий элементы
        // если текущий больше последующего, меняем их местами
        if (arr[j] > arr[j + 1])
        {
            int temp = arr[j + 1]; // обмен значений
            arr[j + 1] = arr[j];   // двух переменных
            arr[j] = temp;         // через третью
        }
    }
}
```



# Рассчитываем количество операций

Можно упрощенно считать, пользуясь следующей подсказкой:

- $x = y$                       1 операция присвоения значения
- $x < y$                       1 операция сравнения
- $x++$                       1 операция инкремента
- `int x = y`                2 операции (создание переменной и присвоение значения)
- `int x = y + z`          3 операции (создание переменной, сложение, присвоение значения)
- `for (int x = 0; x < N; x++)`    2 + 2N операций:
  - создание переменной и присвоение значения происходит один раз
  - сравнение и инкремент происходит каждую итерацию
  - все тело цикла будет выполняться N раз (все операции внутри умножаются на N)

Надо понимать, что это не точное определение количества команд, однако, при асимптотическом подсчете сложности это не так принципиально. Важен порядок функции  $f(N)$



# Рассчитываем количество операций

```
for (int i = 0; i < arr.Length - 1; i++)           // 2 + 2 × (N - 1) × (
{
    int limit = arr.Length - 1 - i;                // 3
    for (int j = 0; j < limit; j++)                 // 2 + 2 × (N - 1) / 2 × (
    {
        if (arr[j] > arr[j + 1])                   // 2
        {
            int temp = arr[j + 1];                 // 3 : эти три строчки
            arr[j + 1] = arr[j];                   // 3 : кода могут не
            arr[j] = temp;                          // 1 : выполниться ни разу
        }                                           // )
    }                                               // )
}
```

Худший случай:  $2 \times (N-1) \times (3+2+2 \times (N-1) / 2 \times (9)) = (2N-2) \times (9N-4) = 18N^2-26N+8$

Лучший случай:  $2 \times (N-1) \times (3+2+2 \times (N-1) / 2 \times (2)) = (2N-2) \times (2N+3) = 4N^2+2N-6$



# Рассчитываем сложность алгоритма

Худший случай:  $18 \times N^2 - 26 \times N + 8$

Лучший случай:  $4 \times N^2 + 2 \times N - 6$

Применяем упрощения:

1. Все операции, не зависящие от переменных факторов сводятся к 1
2. В расчет идут только функции высшего порядка.

Получается, что

- О-нотация:  $O(N^2)$
- $\Omega$ -нотация:  $\Omega(N^2)$

Поскольку мы вычислили и О- и  $\Omega$ -нотацию и они оказались равны, можно записать сложность в более короткой и точной  $\Theta$ -нотации:

- $\Theta$ -нотация:  $\Theta(N^2)$

# Рассчитываем сложность алгоритма

---

Вот **некоторые функции**, которые чаще всего используются для вычисления сложности. Функции перечислены в **порядке возрастания сложности**. Чем выше в этом списке находится функция, тем быстрее будет выполняться алгоритм с такой оценкой.

- $O(1)$  – константная сложность
- $O(\log(N))$  – логарифмическая сложность
- $O(N)$  – линейная сложность
- $O(N \times \log(N))$  – квазилинейная сложность
- $O(N^C)$ , где  $C > 1$  – экспоненциальная сложность
- $O(C^N)$ , где  $C > 1$  – “гладкая” функция, растет ещё быстрее, чем  $N^C$
- $O(N!)$  – факториальная сложность

# Считаем время работы алгоритма

Для подсчета времени работы кода в .NET есть класс Stopwatch, расположенный в области видимости System.Diagnostics.

Методы класса:

- Start() : запустить секундомер
- Stop() : остановить секундомер
- Restart() : сбросить предыдущий замер и запустить секундомер

Свойства класса:

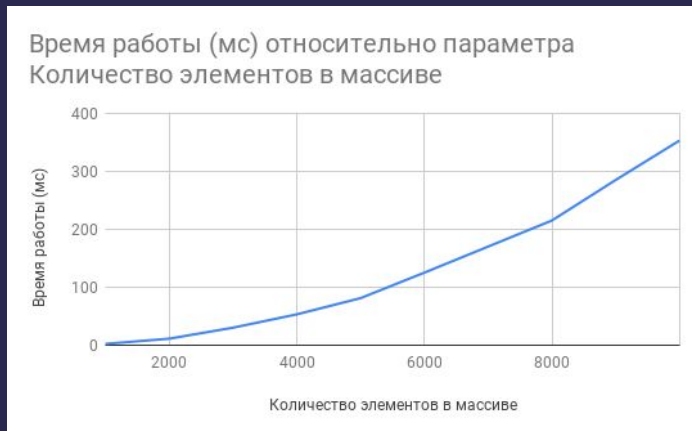
- ElapsedMilliseconds : количество миллисекунд между вызовами Start и Stop.

```
var stopwatch = new Stopwatch(); // init Stopwatch instance
stopwatch.Start();               // start time calculation
target code for execution time calculations
stopwatch.Stop();                // start time calculation
Console.WriteLine($"Code done in {stopwatch.ElapsedMilliseconds} ms");
stopwatch.Restart();             // reset last calculation and start again
other code needs in time calculations
```

# Считаем время выполнения BubbleSort

Если посчитать время выполнения нашей функции BubbleSort, мы увидим экспоненциальный рост времени работы нашего метода при линейном увеличении количества элементов массива.

- 1000 : ~2 мс
- 2000 : ~11 мс
- 3000 : ~30 мс
- 4000 : ~53 мс
- 5000 : ~81 мс
- 6000 : ~125 мс
- 7000 : ~170 мс
- 8000 : ~215 мс
- 9000 : ~285 мс
- 10000 : ~353 мс



# Встроенная сортировка .NET: `Array.Sort`

.NET имеет встроенные алгоритмы работы с данными и для сортировки, конечно же, имеется своя реализация.

Если заглянуть “под капот”, можно узнать, `Array.Sort()` динамически выбирает один из трех алгоритмов сортировки в зависимости от размера массива:

- Если размер меньше 16 элементов, используется "сортировка вставками" (Insertion Sort algorithm),  
 $\Omega(N)$ ,  $O(N^2)$
- Если размер превышает  $2 \times \log^2 N$ , где  $N$  - диапазон значений входного массива, используется алгоритм пирамидальной сортировки (Heap Sort algorithm),  
 $O(N \times \log(N))$ .
- В остальных случаях используется "быстрая сортировка" (Quicksort algorithm),  
 $\Omega(N \times \log(N))$ ,  $O(N^2)$

На частично отсортированных данных все эти методы будут работать лучше, так как наша “пузырьковая” реализация даже в лучшем случае – это  $\Omega(N^2)$ .

# Самостоятельная работа

---

Дописываем программу таким образом, чтобы можно было сравнить наш алгоритм сортировки “пузырьком” с встроенной сортировкой .NET по времени.

По завершению сортировки каждым из методов, должно выводиться 2 значения - время работы соответствующего метода в миллисекундах.



# Домашнее задание

---

Посчитать асимптотическую сложность алгоритма вашего решения задачи прошлого урока (на валидацию скобок):

- наилучший случай –  $\Omega$ -нотация,
- наихудший случай –  $O$ -нотация,

Если это имеет смысл, описать сложность в виде  $\Theta$ -нотации.





# Спасибо за внимание.

