# Об авторе

Всем привет!

Меня зовут Иосиф Дзеранов. Я программист, инженер-разработчик со стажем более 10 лет. Работал в крупных компаниях (Сбербанк, Mail.ru). Мною пройден длинный путь от начинающего до старшего разработчика, и я могу помочь вам добиться такого же успеха.

Коротко про мои достижения:

- Семьянин. Есть двое прекрасных детей;
- Преподаватель и основатель онлайн-школы IRON PROGRAMMER;
- Сертифицированный преподаватель IT школы Samsung;
- Четвертьфиналист чемпионата мира по олимпиадному программированию ACM ICPC;
- Создатель центра олимпиадной подготовки СОГУ;
- Победитель VK FELLOWSHIP 2020;
- Победитель в конкурс “Умник” при фонде содействия инновациям;
- Создатель более 55 онлайн курсов по программированию и информатике/

Связаться со мной можно по почте iodzeranov@mail.ru или в Telegram (t.me/JosefDzeranov).

**Telegram:** https://t.me/csharp_publics

**Вконтакте:** https://vk.com/ironprogrammer

**Сайт:** https://ironprogrammer.ru/

**Youtube:** https://www.youtube.com/@IRONPROGRAMMER

**Почта:** pro.csharp@mail.ru

Для общения с преподавателем и однокурсниками был создан данный чат. Здесь можно общаться на любые темы.

## Кому адресована эта книга

Эта книга поможет развить навыки алгоритмов, про которые постоянно спрашивают на собеседованиях при приеме на работу программистов.

Познакомимся с языком алгоритмов, научимся определять эффективность алгоритма по времени и по памяти.

Затем Вас ждет увлекательный мир алгоритмов поиска. Для каждого алгоритма разберем его преимущества и недостатки и в каких случаях он применяется.

Далее мы погрузимся в мир алгоритмов сортировки. Рассмотрим основные сортировки, отличия между ними, когда и какой алгоритм применять.

# Как читать эту книгу

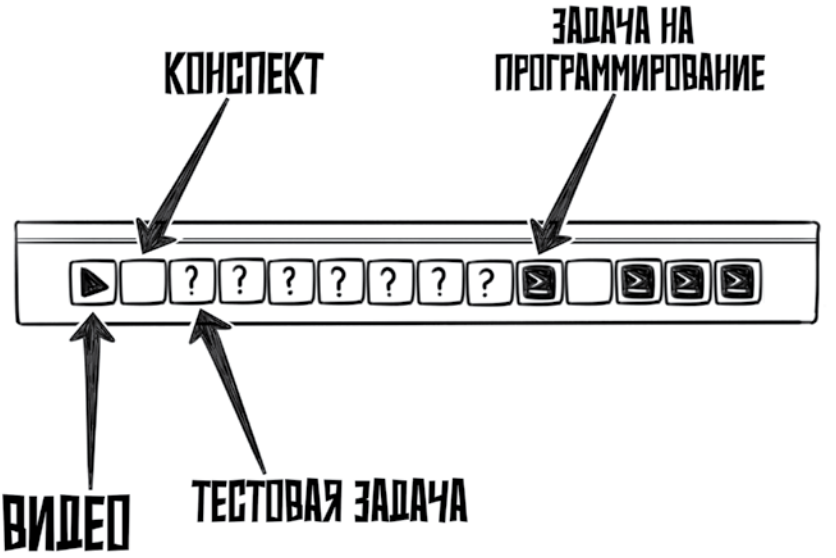
Данная книга является печатной версией онлайн курса.

Обучающие модули расположены от простого к сложному, что предполагает последовательное и вдумчивое чтение.

Программирование требует практики, поэтому для чтения этой книги понадобится компьютер или ноутбук — так вы сможете сразу отрабатывать теоретические навыки на практических упражнениях.

Книга также может быть использована в качестве справочника для того, чтобы освежить знания в определенной теме.

Для тех, кто желает закрепить теорию на практике с автоматической проверкой заданий и обратной связью от преподавателя, я оставлю ссылку на онлайн-курс.



# Анализ алгоритмов

## Асимптотический анализ

Существуют две полярные точки зрения на алгоритмы.

1. Алгоритмы — являются основой компьютерных наук. Такой подход часто встречается.
2. Алгоритмы не нужны.

Я скажу так: **важно знать не большое количество алгоритмов, а язык, на котором обсуждаются алгоритмы.**

**Алгоритм** — это последовательность элементарных действий для выполнения конкретной цели.

Элементарные действия:

1. **арифметические операции**
2. **ввод строки, числа**
3. **вывод строки, числа**
4. **операторы сравнения**
5. **операция присваивания**

Не элементарные действия:

1. **цикл**
2. **рекурсия**

С точки зрения теории алгоритмов, у каждого алгоритма есть входные данные (input), заложенный алгоритм, который манипулирует входными данными и выходные данные, собственно результат манипуляций алгоритма.

В качестве входных данных могут быть число или числа, строки, логические значения, комбинация разных типов данных, то есть это та информация, которая нужна алгоритму, чтобы выполнить поставленную ему задачу.

Выходные данные также могут быть разного вида. Вы должны понимать, что все зависит от конкретной задачи.

**Сложность алгоритма?**

**Сложные алгоритмы?**

**Простые алгоритмы?**

Что такое **сложность алгоритма?** Алгоритм сложен не для понимания, а для исполнения. То есть требуется много элементарных операций для выполнения данного алгоритма, и потому он сложен для машины (компьютера).

Сложность алгоритма разделяют на две категории:

1. **Сложность по времени** (временная сложность). Показывает сколько времени тратится на выполнение элементарных операций.
2. **Сложность по памяти** (емкостная сложность). Показывает сколько памяти тратится на выполнение элементарных операций.

## Расчет временной сложности

Давайте посчитаем время выполнения следующей программы:

**C#**

```
int sum = 0;
for (int i = 0; i < 100; i++)
{
   int number = i;
    if (number % 3 == 0)
    {
         sum = sum + number;
}
}
Console.WriteLine(sum);
```

Даже не представляю из каких соображений будем находить время выполнения данной программы. Следовательно, предлагаю замерить с помощью самой же программы. Для этого воспользуемся классом **Stopwatch** и посчитаем количество **тиков** — минимальная **единица измерения времени выполнения одной операции на процессоре**:

**C#**

```
Stopwatch stopwatch = new Stopwatch();
stopwatch.Start();
int sum = 0;
for (int i = 0; i < 100; i++)
{
    int number = i;
    if (number % 3 == 0)
    {
         sum = sum + number;
}
}
Console.WriteLine(sum);
Console.WriteLine($"Время выполнения {stopwatch.Elapsed.
Ticks}тиков");
```

Смотрим результаты:

*Время выполнения 96779677 тиков*

*Время выполнения 96379637 тиков*

*Время выполнения 1001210012 тиков*

*Время выполнения 1058110581 тиков*

Исходя из результатов замеров можно сказать, что **выполнение программы даже на одном и том же компьютере дает разное время выполнения**. Что уж говорить про запуск программы на разных компьютерах с разными характеристиками.

Можно задаться вопросом, **почему нельзя однозначно определить время работы алгоритма на компьютере?** Потому что программы запускаются в операционной системе (в моем случае Windows), которая устроена не так просто, как кажется обычному пользователю компьютера:

- некоторые элементарные операции могут выполняться процессором дольше, чем другие. Все зависит от того, в какой момент была запущена программа;
- в многозадачных операционных системах выполнение алгоритма может приостанавливаться на время выполнение других важных для ОС задач. Это решается на уровне операционной системы;
- кэш процессора может ускорять работу с памятью, однако алгоритмы не могут напрямую управлять работой кэша;
- компиляция одной и той же программы разными компиляторами или с разными настройками компилятора может давать разный набор элементарных машинных инструкций. Например, компиляция под **debug** и **release** версии дают разный набор машинных инструкций.

**Вывод**: **определить точное время** выполнения в тиках или в миллисекундах практически **невозможно**.

Перечислим что может влиять на время выполнения программы:

1. количество ядер на компьютере;
2. скорость чтения/записи в память;
3. 32 или 64 разрядная операционная система;

4. кэш процессора;
5. файлы подкачки;
6. **входные данные.**

В realtime системах, например таких как биржи, управление самолета, важны наносекунды, так как время, в течение которого информация актуальна, крайне мало. То есть, если мы получили значения датчиков самолета и слишком долго анализировали, и считали, то выходное воздействие может быть неактуальным. Из-за этого, в системах, где критично важно время выполнения программы, считают каждую операцию в **тиках**. В таких системах **важно** на каком компьютере данная программа выполняется, так как учитывается все окружение.

В среднем по характеристикам современном персональном компьютере **за** 1 **секунду** выполняется примерно **миллиард элементарных операций**.

## Теория алгоритмов

В теории алгоритмов не рассматривают характеристики окружения, на котором выполняется программа. Учитывают только **входные данные**! От них будет зависеть количество элементарных действий. Важна тенденция на большом количестве входных данных.

**Запомни: при анализе алгоритма следует понимать, как меняется количество действий при увеличении количества входных данных.**

Почему нам важно для больших входных данных? А все потому, что на маленьких данных большинство алгоритмов работают достаточно быстро, а вот с увеличением входных данных уже нужно смотреть и анализировать. Все в этом мире растет и приумножается: машины, железные дороги, книги, поэтому мы и исследуем то, насколько наш алгоритм сможет «выдер-

жать» потенциальный рост данных. «Выдержать» значит уложиться в ограничения по времени и по памяти из условий задачи.

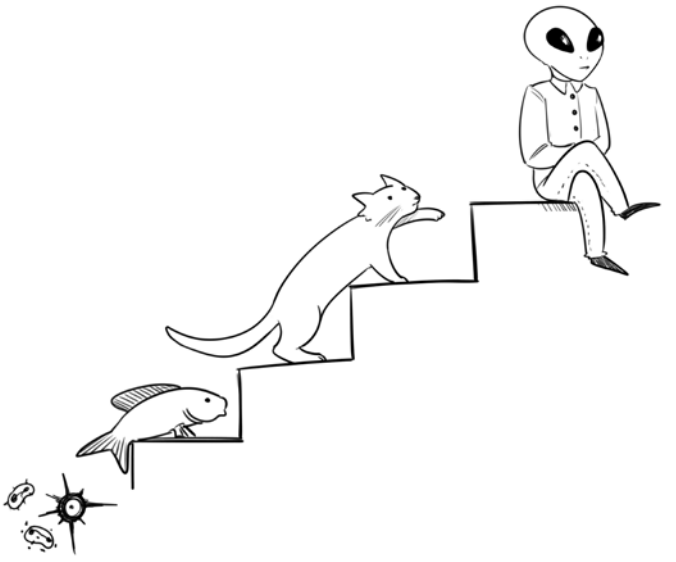
На самом деле расчет временной сложности можно рассматривать как некоторую **функцию**, которая принимает **входные данные** и выдает **количество** элементарных операций, потраченный на обработку этих данных.

Рассмотрим следующую программу:

**C#**

```
static void Sum(int a, int b)
{
    int sum = a + b;
    Console.WriteLine(sum);
}
```

Посчитаем, сколько здесь элементарных действий. В данной программе три элементарных действия: сложение, присваивание и вывод на консоль. Говорят, что данная программа занимает **константное время**, так как от увеличения входных данных **a** и **b**, количество элементарных действий не поменяется. И обозначают как **O(1)** — говорят о большое от единицы. Почему от единицы? Потому что как бы мы не увеличивали входные данные, количество действий не изменится. Эти действия **ничтожно малы и равны единице по сравнению с входными данными**.



Рассмотрим программу и посчитаем для него временную сложность:

**C#**

```
static void Print(int n)
{
    for (int i = 1; i <= n; i++)
    {
        Console.WriteLine(i);
}
}
```

При n=5 алгоритм сделает 5 действий вывода на экран. Если мы n увеличим в 10 раз, то количество действий также увеличится не более чем в 10 раз.

Так вот, сложность алгоритма называется **линейной**, если при увеличении входных данных в n раз, количество элементарных действий алгоритма увеличится не более чем в n раз. Говорят, что алгоритм имеет **линейную сложность**, либо алгоритм работает за линию. Еще говорят, что алгоритм делает **не более** чем n операций и записывают как **O(n)** — говорят O большое от n.

**Давайте все-таки считать точнее количество элементарных операций или действий:**

## Задача 1.

**C#**

```
static void Print(int n)
{
    for (int i = 1; i <= n; i++)
    {
        Console.WriteLine(i);
}
}
```

- 1 операция для **int i = 1**
- n+1 операций сравнения при **i <= n**
- 2n операций для **i++** (эквивалентно i = i + 1, а это две операции: **присваивание** и **сложение**)
- n операций для **Console.WriteLine(i)**

**Итого:** временная сложность равна $O(1 + (n + 1) + 2n + n) = O(4n + 2)$.

### Задача 2.

**C#**

```
int n = Convert.ToInt32(Console.
ReadLine());
int i = 0;
int count = 0;
while (i < n)
{
    for (int j = 0; j < n; j++)
    {
        count++;
}
    i++;
}
```

- 1 операция для **int i = 0**
- 1 операция для **int count = 0**
- n+1 операций сравнения при **i < n**
- 2n операций для **i++**
- nn итераций цикла **for** и каждый раз:
  - 1 операция для **int j = 0**
  - n+1 операций сравнения при **j < n**
  - 2n операций для **j++**
  - 2n операций для **count++**

**Итого**: временная сложность равна $O(1 + 1 + (n + 1) + 2n + n(1 + (n + 1) + 2n + 2n)) = O(5n^2 + 5n + 3)$.

Согласитесь, что **даже для простых алгоритмов сложно посчитать точное количество элементарных операций.** Из-за этого ввели такое понятие как асимптотическая сложность с помощью которого вычисляется сложность алгоритмов. Если говорить по-простому, то она получается следующим образом:

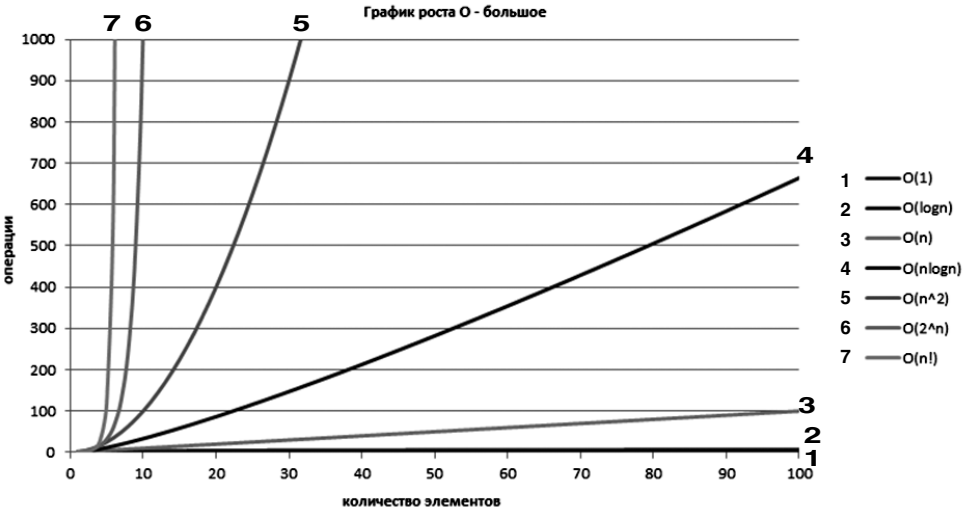
1. отбросим в функции сложности все слагаемые, **кроме одного с самой быстрой скоростью роста;**
2. отбросим все константы.

Это и будет **асимптотической оценкой сложности.**

Зная теперь, что такое асимптотическая сложность, давайте посчитаем для примеров, которые были приведены выше:

1. для первой задачи временная сложность равна **O(4n+2)**. Оставим только слагаемое 4n, так как оно является с самой быстрой скоростью роста. А потом уберем константу 4, так как при увеличении n, константа 4 большой роли не играет. То есть, асимптотическая сложность будет равна O(n).
2. для второй задачи временная сложность равна $O(5n^2 + 5n + 3)$. Оставим только слагаемое 5n2, так как оно является с самой быстрой скоростью роста. А потом уберем константу 5, так как при увеличении n, константа 5 большой роли не играет. То есть асимптотическая сложность будет равна $O(n^2)$.

Давайте посмотрим на часто используемые и встречающиеся асимптотические сложности:




Оценки сложности расположены в возрастающем порядке по асимптотической сложности. Чем выше его оценка, чем сложнее данный алгоритм, то есть алгоритм делает в разы больше операций при кратном увеличении входных данных.

Полезно иметь оценку сложности того или иного алгоритма, но еще важнее то, что с помощью таких оценок **можно сравнивать** друг с другом **разные алгоритмы** для решения одной задачи. Допустим, для некоторой задачи имеется два алгоритма, трудоемкость одного из них оценивается как $2n^2$, а другого — как 100n. Тогда **при небольшой длине входа** (n <50) предпочтительнее использовать **первый алгоритм**, а **при большой** (n >50) **— второй**. Но при малой длине входа время работы каждого из двух алгоритмов будет невелико,

так что, может быть, мы практически ничего не потеряем, если и здесь будем пользоваться вторым алгоритмом. Этот пример помогает понять, почему при анализе сложности алгоритмов **главное внимание уделяется асимптотическому поведению сложности**, то есть поведению при больших длинах входа.

## Расчет временной сложности на практике

### Задание 1.

**C#**

```
int n = Convert.ToInt32(Console.ReadLine());
int a = n / 100;
int b = n / 10% 10;
int c = n % 10;
int revert = c * 100 + b * 10 + a;
Console.WriteLine(revert);
```

**Решение**:
Заметим, что при увеличении входного параметра n, количество операций не поменяется. А количество операций будет конечное количество, следовательно асимптотическая сложность O(1).

**Ответ**: O(1).

### Задание 2.

**C#**

```
int n = Convert.ToInt32(Console.ReadLine());
int count = 0;
for (int i = 0; i < n; i++)
{
    int number = Convert.ToInt32(Console.
ReadLine());

    if (number % 10 == 0)
    {
        count = count + 1;
}
}
Console.WriteLine(count);
```