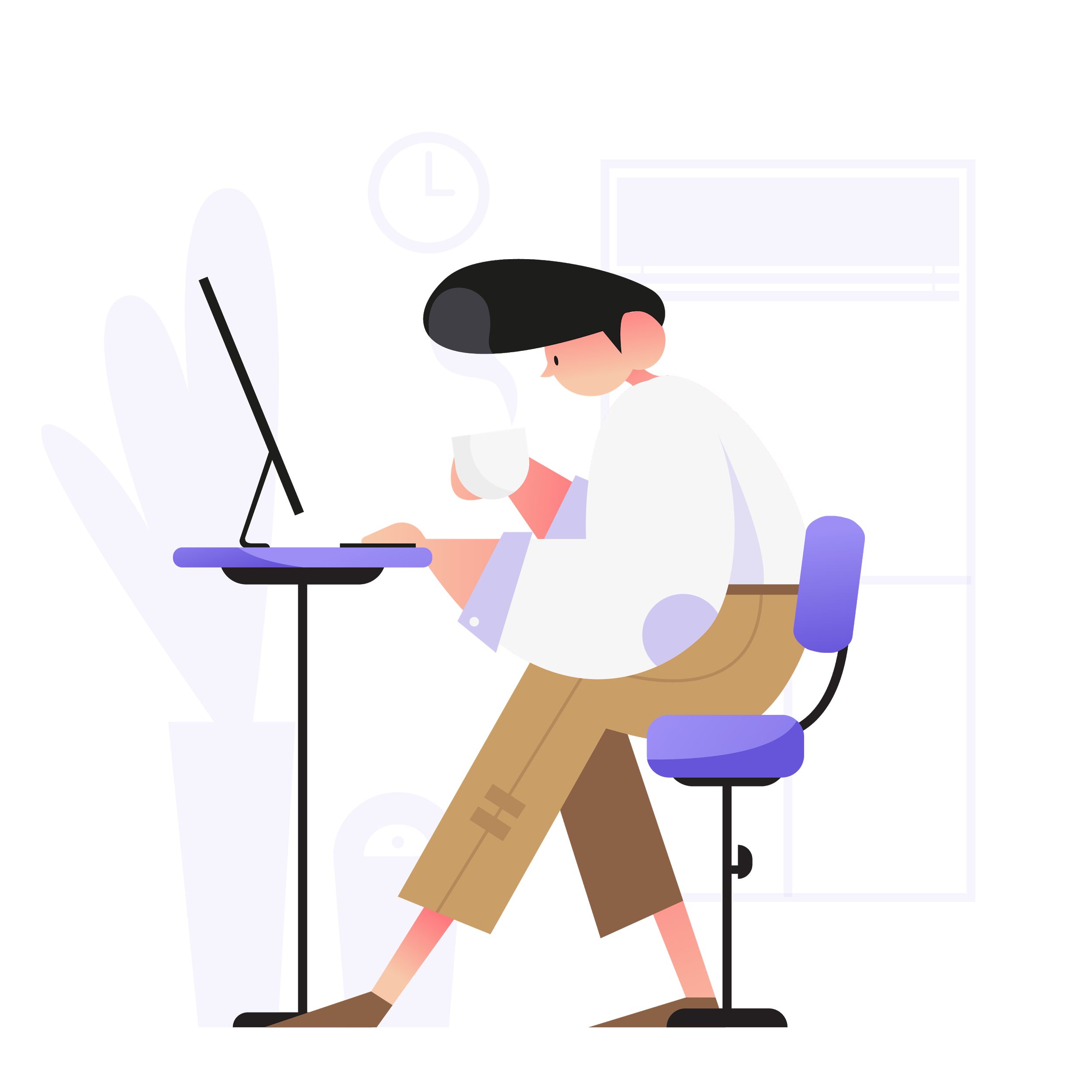
Алгоритмы и структуры данных на С#

Блок-схемы, асимптотическая сложность, рекурсия

Netcore 3.1





**На этом уроке**

1. Узнаем определение блок-схемы.
2. Рассмотрим примеры блок-схем.
3. Узнаем, что такое асимптотическая сложность.
4. Познакомимся с методом вычисления асимптотической сложности.
5. Выясним, что такое рекурсия и проблема переполнения стека.
6. Изучим метод ухода от рекурсии.

**Оглавление**

[Введение](#_fbh22easandh)

[Блок-схемы для описания алгоритмов](#_5ku3mq4nc00e)

[Блок-схема](#_wz371bvslc14)

[Определение наибольшего общего делителя (алгоритм Евклида)](#_1lcx6m4vwf1f)

[Переворот числа](#_8z8fs05ugk97)

[Асимптотическая сложность](#_21tgs5mwgy4h)

[Асимптотическая сложность алгоритма](#_3znysh7)

[Правило 1](#_bb1imdoa1qau)

[Правило 2](#_5px949dbjtvd)

[Правило 3](#_3medy59641o9)

[Правило 4](#_1t3h5sf)

[Правило 5](#_to0hrengymp8)

[Рекурсия](#_tvrdznbujqrh)

[Пример. Цикл с помощью рекурсии](#_il13akxamvl0)

[Пример. Вычисление факториала](#_hjn6ebfqexg2)

[Рекурсивный вариант](#_xov7lq7vral3)

[Вариант с циклом](#_159qyfavvush)

[Вариант со структурой](#_xas4zujh4hkb)

[Заключение](#_pgjch5xha956)

[Практическое задание](#_u9zjs5ygcqvh)

[1. Напишите на C# функцию согласно блок-схеме](#_c9dtpfeb0upc)

[2. Посчитайте сложность функции](#_1k9x2fitps9y)

[3. Реализуйте функцию вычисления числа Фибоначчи](#_l3iozxn0vkbh)

[Дополнительные материалы](#_gl5mhdcsp6on)

[Используемые источники](#_7c9jrnh1eqkx)

# 

# Введение

Любое современное программное обеспечение явно или неявно использует и реализует алгоритмы и структуры данных, на которых построена логика его работы. Программисту важно их знать и понимать.

На этом курсе вы узнаете о структурах данных и алгоритмах, которые необходимы начинающему программисту для старта карьеры. Мы рассмотрим массивы, списки, деревья, графы, алгоритмы поиска и сортировки. Узнаем, в каких задачах эти алгоритмы и структуры могут применяться и какие структуры уже есть в C#.

На первом уроке мы рассмотрим блок-схемы, асимптотическую сложность и рекурсию.

# 

# Блок-схемы для описания алгоритмов

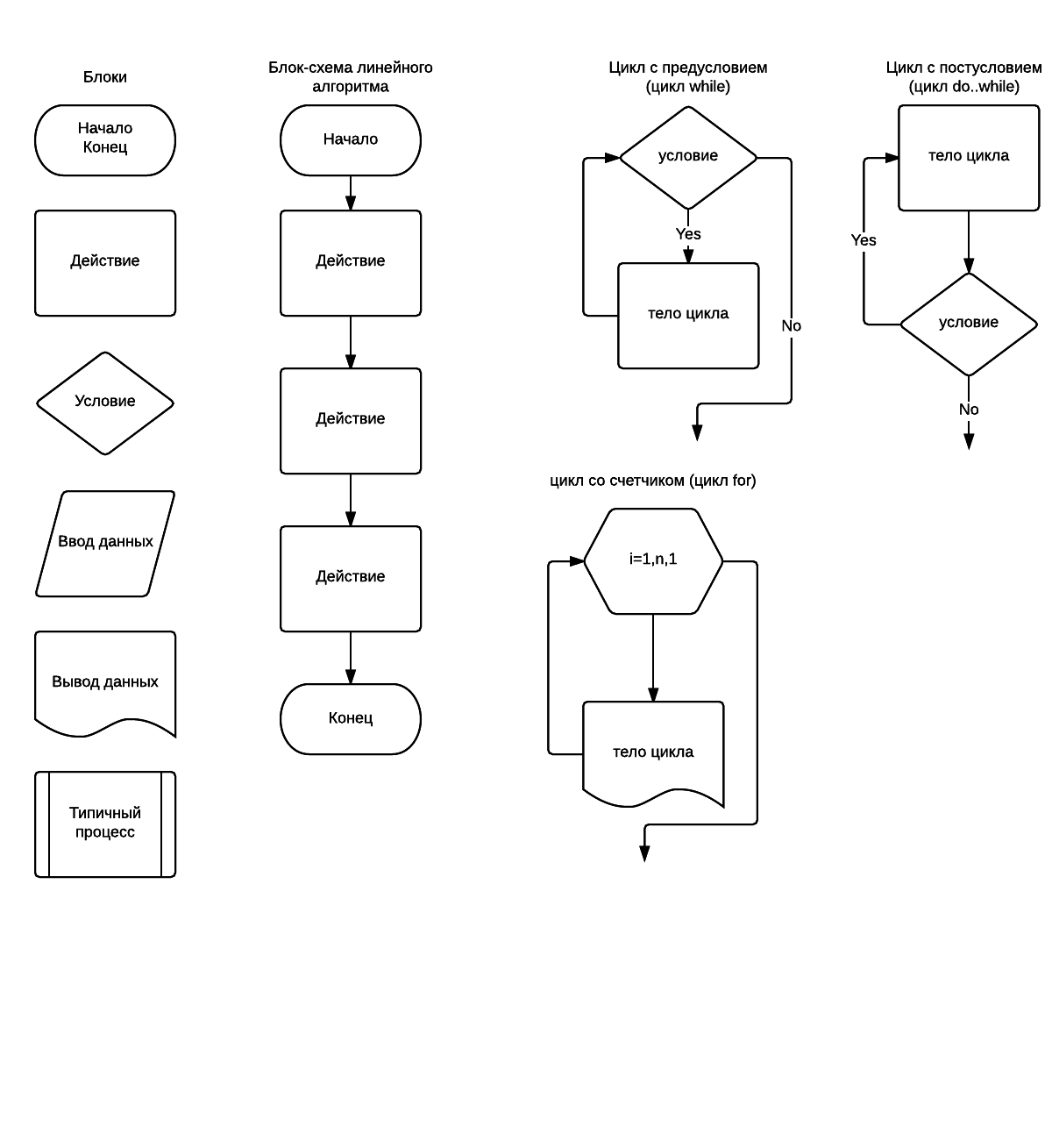
## Блок-схема

Любой алгоритм — это последовательность действий или инструкций, направленных на достижение какой-либо цели. Такую последовательность можно описать с помощью блок-схемы.

Ниже представлены структурные компоненты для описания блок-схем.

## 

## 



Если проводить аналогию с C#, то цикл — это операторы while и for, условие — это операторы if-else и switch-case, ввод-вывод — класс Stream и его наследники.

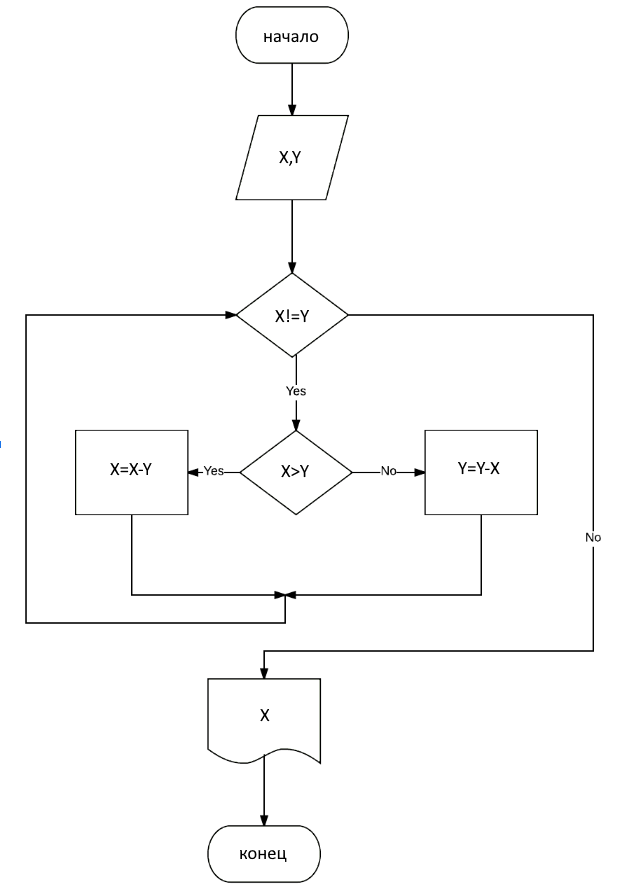
Мы познакомились с компонентами для составления блок-схем. Далее мы рассмотрим примеры блок-схем (алгоритм Евклида и переворот числа) и их реализацию на C#.

## Определение наибольшего общего делителя (алгоритм Евклида)

Алгоритм поиска наибольшего общего делителя (алгоритм Евклида) заключается в следующем: пока **a** не равно **b**, вычитаем из большего числа меньшее, где **a** и **b** —входные числа. Рассмотрим его на примере кода с комментариями:

|  |
| --- |
| using System;  namespace GeekBrainsAlgos {  class Program  {  static void Main(string[] args)  {  int a;  int b;   *//НАЧАЛО БЛОКА ВВОДА*  Console.WriteLine("Input a:");  a = int.Parse(Console.ReadLine()); *// правильнее будет написать int.TryParse, так как это позволит избежать exception в случае, если в строке будет не число. Для данного примера этим можно пренебречь.*  Console.WriteLine("Input b:");  b = int.Parse(Console.ReadLine());  *//КОНЕЦ БЛОКА ВВОДА*    while (a != b) *//УСЛОВИЕ ЦИКЛА*  { *//НАЧАЛО ТЕЛА ЦИКЛА*   if (a > b) *//УСЛОВИЕ*  {  a = a - b; *//ДЕЙСТВИЕ, ЕСЛИ УСЛОВИЕ ВЫПОЛНИЛОСЬ*  }  else  {  b = b - a; *//ДЕЙСТВИЕ, ЕСЛИ УСЛОВИЕ НЕ ВЫПОЛНИЛОСЬ*  }   } *//КОНЕЦ ТЕЛА ЦИКЛА*   Console.WriteLine($"GDC: {a}"); *//ВЫВОД*  }  } } |

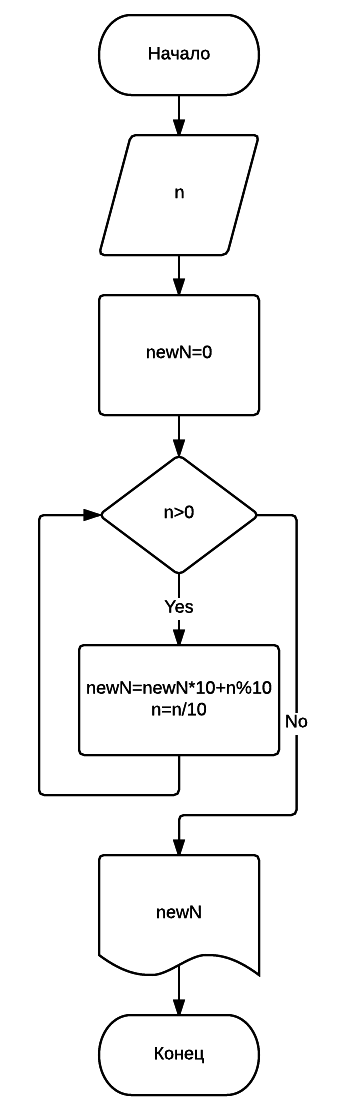
Тело функции main соответствует блок-схеме ниже:



## Переворот числа

Алгоритм переворота числа состоит в следующем: на каждом шаге от него отбрасывается последняя цифра и прибавляется к новому числу, умноженному на 10.

Алгоритм визуализирует следующая блок-схема:



Рассмотрим переворот числа на примере кода:

|  |
| --- |
| using System;  namespace GeekBrainsAlgos {  class Program  {   public static long Reverse(long n) *//В данном случае ввод — это просто аргумент функции. Просто входной параметр.*  {  long newN = 0;  while (n > 0)  {  newN = newN \* 10 + n % 10;  n /= 10;  }  return newN;  }    static void Main(string[] args)  {  int n = 123;  Console.WriteLine($"Output: {Reverse(n)}");  }  } } |

# 

# 

# Асимптотическая сложность

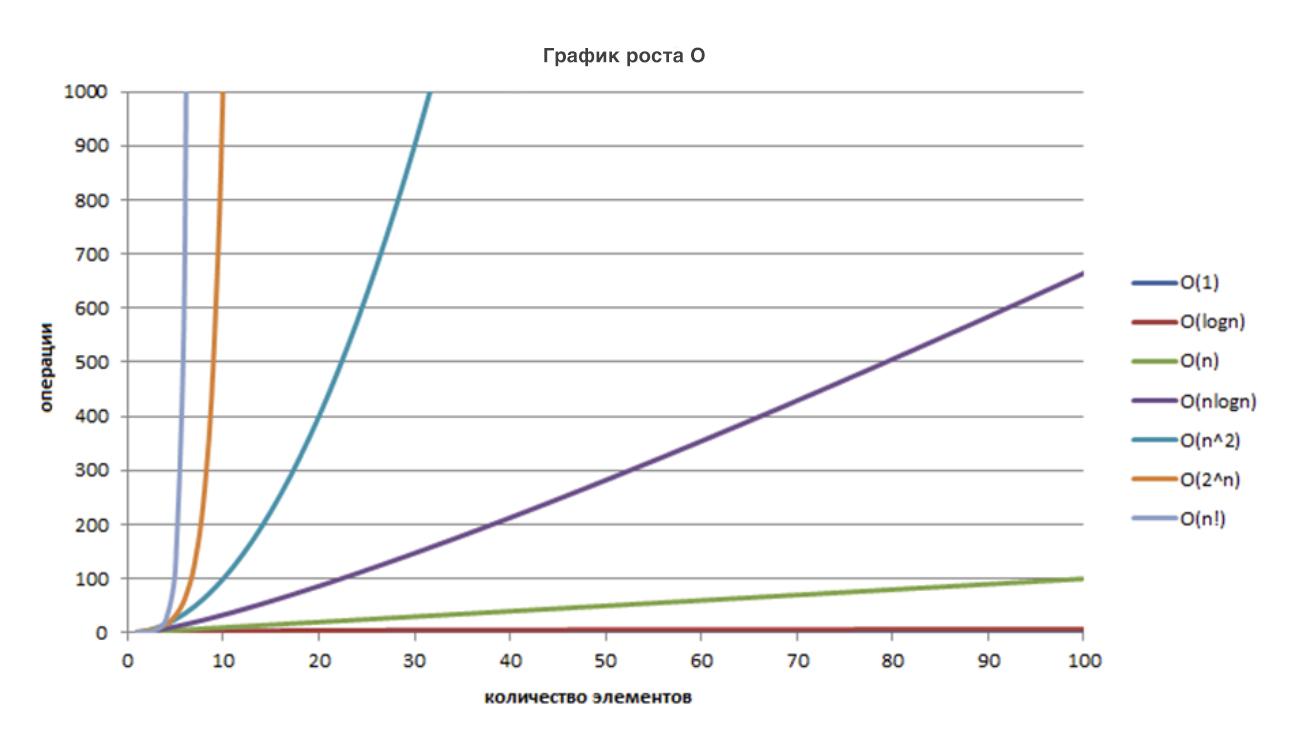
В серьёзных проектах часто необходимо доказывать эффективность или неэффективность кода. Для этого обязательно требуется его формальный анализ, который невозможно провести без теоретических знаний об алгоритмах и их асимптотической сложности.

## Асимптотическая сложность алгоритма

Асимптотическая сложность (производительность) определяется функцией, которая указывает, насколько ухудшается работа алгоритма с усложнением поставленной задачи. Такую функцию записывают в круглых скобках, предваряя прописной буквой О.

Например, доступ к ячейке массива описывается как O(1), так как для него требуется всего одна элементарная операция и сложность здесь не возрастает независимо от размера массива.

O(N2) означает, что по мере увеличения количества входных данных время работы алгоритма — использование памяти либо другой измеряемый параметр — возрастает квадратично. Если данных станет вдвое больше, производительность алгоритма замедлится приблизительно в четыре раза. При увеличении количества входных данных в три раза она станет меньше в девять раз.



**Пять основных правил для расчёта асимптотической сложности алгоритма:**

1. Если для математической функции f алгоритму необходимо выполнить определённые действия f(N) раз, то ему понадобится сделать O(f(N)) шагов.
2. Если алгоритм выполняет одну операцию, состоящую из O(f(N)) шагов, а затем вторую, включающую O(g(N)) шагов, то общая производительность алгоритма для функций f и g составит O(f(N) + g(N)).
3. Если алгоритму необходимо сделать O(f(N) + g(N)) шагов и область значений N функции f(N) больше, чем у g(N), то асимптотическую сложность можно упростить до выражения O(f(N)).
4. Если алгоритму внутри каждого шага O(f(N)) одной операции приходится выполнять ещё O(g(N)) шагов другой операции, то общая производительность алгоритма составит O(f(N) × g(N)).
5. Постоянными множителями (константами) можно пренебречь. Если C — константа, то O(C × f(N)) или O(f(C × N)) можно записать как O(f(N)).

Приведённые правила кажутся формальными из-за абстрактных функций f(N) и g(N), но ими очень легко пользоваться на практике. Вот несколько примеров, которые сделают их понятнее.

### Правило 1

Если для математической функции f алгоритму необходимо выполнить определённые действия f(N) раз, то ему понадобится сделать O(f(N)) шагов.

|  |
| --- |
| static int FindMax(int[] array)  {  int result = array[0]; *// O(1)*  for (int i = 1; i < array.Length; i++) *// O(N)*  if (array[i] > result)  result = array[i];  return result; *// O(1)*  } |

В качестве входного параметра программа использует массив целых чисел, результат возвращается в виде одного целого числа. В самом начале переменной max присваивается значение первого элемента массива. Затем алгоритм перебирает оставшиеся элементы и сравнивает значение каждого из них с max. Если он находит большую величину, то приравнивает max к ней и по окончании цикла возвращает наибольшее найденное значение. Алгоритм проверяет каждый из N элементов массива всего один раз, поэтому его производительность составляет O(N).

### Правило 2

Если алгоритм выполняет одну операцию, состоящую из O(f(N)) шагов, а затем вторую операцию, включающую O(g(N)) шагов, то общая производительность алгоритма для функций f и g составит O(f(N) + g(N)).

Вернёмся к алгоритму FindMax. На этот раз обратите внимание, что несколько строк в действительности не включены в цикл. В следующей программе в комментариях справа приведён порядок времени выполнения всё тех же шагов.

|  |
| --- |
| static int FindMax(int[] array)  {  int result = array[0]; *// O(1)*  for (int i = 1; i < array.Length; i++) *// O(N)*  if (array[i] > result)  result = array[i];  return result; *// O(1)*  } |

Приведённый алгоритм выполняет один шаг отладки перед циклом и ещё один после него. Каждый из них имеет производительность O(1) (это однократное действие), поэтому общее время работы алгоритма составит O(1 + N + 1), то есть O(2 + N).

### Правило 3

Если алгоритму необходимо сделать O(f(N) + g(N)) шагов и область значений N функции f(N) больше, чем у g(N), то асимптотическую сложность можно упростить до выражения O(f(N)).

В предыдущем примере мы выяснили, что время работы алгоритма FindMax определяется выражением O(2 + N). Если параметр N начнёт возрастать, его значение превысит постоянную величину 2 и предыдущее выражение можно будет упростить до O(N).

Игнорирование меньших функций позволяет пренебречь небольшими задачами отладки и очистки, чтобы сосредоточить внимание на асимптотическом поведении алгоритма при усложнении задачи. Другими словами, время, затраченное алгоритмом на построение простых структур данных перед выполнением объёмного вычисления, несущественно по сравнению с длительностью основных расчётов.

### Правило 4

Если алгоритму внутри каждого шага O(f(N)) одной операции приходится выполнять ещё O(g(N)) шагов другой операции, то общая производительность алгоритма составит O(f(N) × g(N)).

Рассмотрим алгоритм, который определяет, содержатся ли в массиве повторяющиеся элементы. Это не самый эффективный способ обнаружения дубликатов.

|  |
| --- |
| static bool FindDublicates(int[] array)  {  for (int i = 0; i < array.Length; i++)  for (int j = 0; j < array.Length; j++)  if (i != j)  if (array[i] == array[j])  return true;  *// Если мы дошли до этого места, значит, дубликатов нет*  return false;  } |

Алгоритм содержит два цикла, один из которых вложенный. Внешний цикл перебирает все элементы массива N, выполняя O(N) шагов. Внутри каждого такого шага внутренний цикл повторно пересматривает все N элементов массива, совершая те же O(N) шагов. Следовательно, общая производительность алгоритма составит O(N × N) = O(N2).

### Правило 5

Постоянными множителями (константами) можно пренебречь. Если C — константа, то O(C × f(N)) или O(f(C × N)) можно записать как O(f(N)).

Снова посмотрите на алгоритм FindDuplicates из предыдущего примера и обратите внимание на внутренний цикл, который представлен условием if. В рамках этого условия определяется, равны ли друг другу индексы i и j. Если нет, тогда сравниваются величины array[i] и array[j]. В случае их совпадения возвращается значение true.

Пренебрегая дополнительным шагом в выражении return (как правило, он выполняется один раз), предположим, что срабатывают оба оператора if — так и происходит в большинстве случаев. Тогда внутренний цикл будет пройден за O(2N) шагов. Следовательно, общая производительность алгоритма составит O(N x 2N) = O(2N2). Последнее правило позволяет пренебречь коэффициентом 2 и записать производительность алгоритма в виде O(N2).

На самом деле мы возвращаемся к сути асимптотической сложности: нужно выяснить, как поведёт себя алгоритм, если N начнёт возрастать. Предположим, вы увеличите N в два раза, то есть будете оперировать значением 2N. Теперь, если подставить его в выражение 2N2, получится следующее: 2 × (2N)2 = 2 × 4N2 = 8N2. Это и есть наша величина 2N2, только умноженная на 4. Таким образом, время работы алгоритма увеличится в четыре раза.

Теперь оценим производительность алгоритма, используя упрощённое по правилу выражение O(N2). При подстановке в него 2N получим следующее: (2N)2 = 4N2. То есть наша изначальная величина N2 возросла в четыре раза, как и время работы алгоритма.

Из всего вышесказанного следует, что независимо от того, будете ли вы использовать развёрнутую формулу 2N2 или ограничитесь просто N2, результат останется прежним: увеличение сложности задачи в два раза замедлит работу алгоритма в четыре раза. Таким образом, здесь важна не константа 2, а тот факт, что время работы возрастает вместе с увеличением количества вводов N2.

Важно понимать, что асимптотическая сложность даёт представление о теоретическом поведении алгоритма. Практические результаты могут отличаться.

# 

# Рекурсия

Рекурсией называется механизм работы программы, в котором для решения задачи из подпрограммы вызывается та же самая подпрограмма. Рекурсию удобно использовать как альтернативу циклу, когда заранее известна глубина рекурсии. Код с рекурсией в некоторых случаях может выглядеть более читаемым, чем версия кода с циклом (например, функция вычисления факториала). В случае неконтролируемой глубины рекомендуется использовать цикл.

Следует понимать, что любой рекурсивный метод можно преобразовать в обычный. И практически любой метод можно преобразовать в рекурсивный, если выявить рекуррентное соотношение между вычисляемыми в методе значениями.

**Внимание!** *Рекурсия — удобный, но опасный инструмент, потому что неконтролируемая или слишком глубокая рекурсия может привести к переполнению стека. Каждый вызов функции или метода приводит к увеличению стека, так как во время вызова функции её аргументы и адрес возврата кладутся в стек, а при завершении функции эти данные из него убираются. Рекомендуется заменять рекурсию на цикл.*

## 

## Пример. Цикл с помощью рекурсии

Рассмотрим пример реализации цикла с помощью рекурсии. Тут вместо for мы используем рекурсию. В функции Loop **a** — номер текущей итерации, **b** — номер конечной итерации.

|  |
| --- |
| using System;   namespace GeekBrainsAlgos {  class Program  {    static void Loop(int a, int b)  {  Console.WriteLine(a);  if (a < b)  Loop(a + 1, b);  }    static void Main(string[] args)  {  Loop(0, 10);  }  } } |

## 

## Пример. Вычисление факториала

Факториал вычисляется по формуле n! = n × (n-1)!. Если n = 0, то n! = 1 . Идеальный кандидат, чтобы написать рекурсивную функцию.

### Рекурсивный вариант

Рекурсия из примера ниже также называется хвостовой, так как вызов функции находится в конце тела вызывающей функции:

|  |
| --- |
| static int Factorial(int n)  {  if (n == 0)  return 1;    return n \* Factorial(n - 1);  } |

### 

### Вариант с циклом

Как видно из примера, рекурсию можно заменить циклом. Мы начали использовать переменную **factorial** для хранения состояния и переменную **i** дляпрохода по всем значениям вместо выражения **n - 1** в конце тела функции рекурсивной версии.

|  |
| --- |
| static int Factorial(int n)  {  int factorial = 1;  for(int i = 1; i <= n; i++)  {  factorial \*= i;  }  return factorial;  } |

### Вариант со структурой

Для более сложных случаев, где функция может вызывать сама себя несколько раз или где решение с циклом трудно применить, есть другой способ ухода от рекурсии. Можно использовать стек (имеется в виду не стек вызова) или очередь, куда будут помещаться элементы, которые нужно обработать, и обработчик, который будет вытаскивать для вычислений элементы из стека или очереди, пока они не иссякнут. Код ниже демонстрирует этот способ снова на примере вычисления факториала. В этом случае такое решение считается излишним и можно использовать вариант с циклом, так как он более простой. Как работает стек и очередь, мы обсудим на следующих уроках.

|  |
| --- |
| using System; using System.Collections.Generic;   namespace GeekBrainsAlgos {  class Program  {    static int Factorial(int n)  {  if(n == 0)  return 1;    int factorial = 1;    Stack<int> stack = new Stack<int>(); *//структура для хранения элементов для обработки*   stack.Push(n); *//положить в стек первое значение*    while(stack.Count > 0) *//цикл будет выполняться, пока есть что-нибудь в стеке*  {  int currentN = stack.Pop(); *//взять последнее значение из стека*  factorial \*= currentN;    int nextN = currentN - 1;    if(nextN > 0) *//условие, при котором мы перестанем класть в стек*  stack.Push(currentN - 1); *//положить в следующее значение в стек*  }  return factorial;  }    static void Main(string[] args)  {  Console.WriteLine(Factorial(5));  }  } } |

# Заключение

Мы рассмотрели, как блок-схема может быть отображена в C#-коде и из каких компонентов она может состоять. Узнали, как рассчитывать асимптотическую сложность с помощью пяти правил. Поняли, что рекурсия удобна, когда мы можем контролировать её глубину.

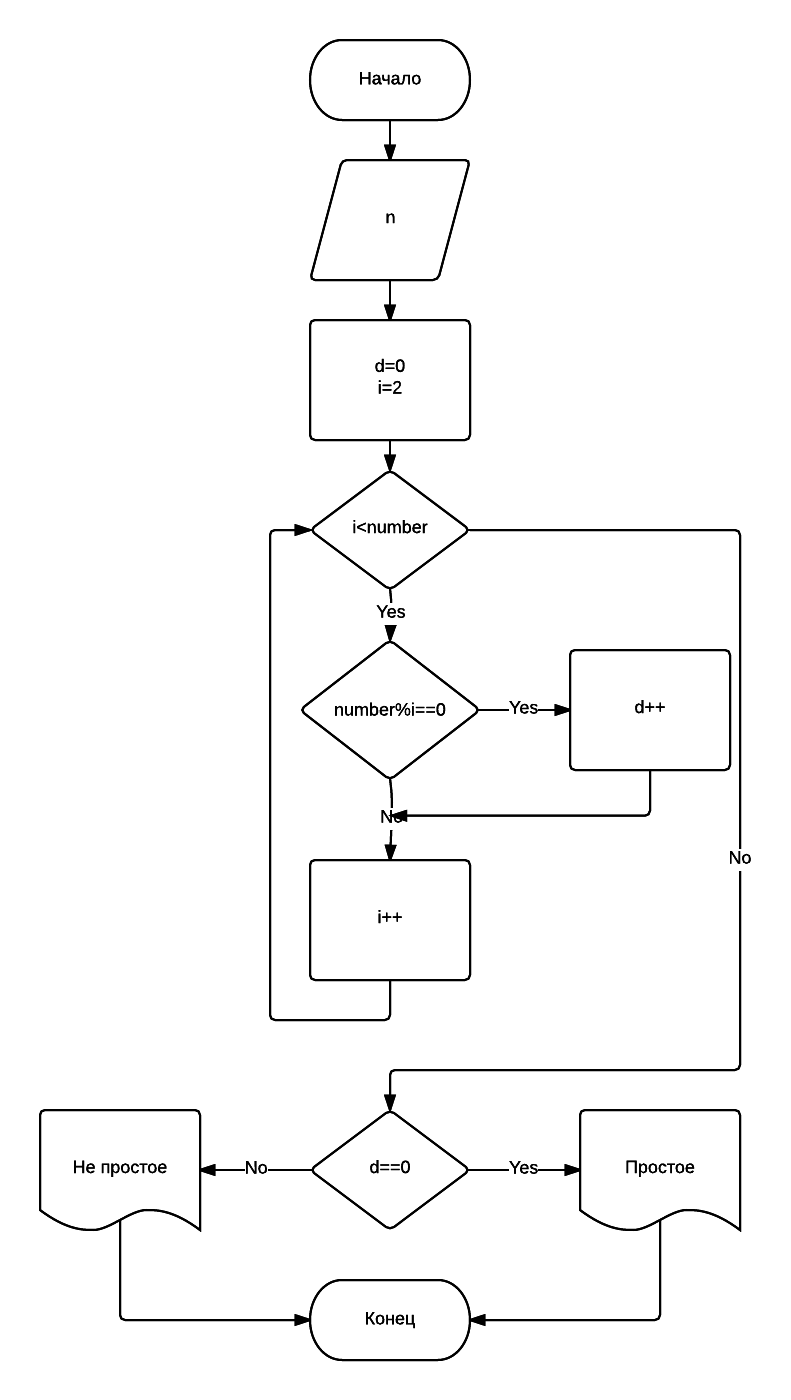
# 

# Практическое задание

## 1. Напишите на C# функцию согласно блок-схеме

Требуется реализовать на C# функцию согласно блок-схеме. Блок-схема описывает алгоритм проверки, простое число или нет.

1. Написать консольное приложение.
2. Алгоритм реализовать отдельно в функции согласно блок-схеме.
3. Написать проверочный код в main функции .
4. Код выложить на GitHub.



## 2. Посчитайте сложность функции

Вычислите асимптотическую сложность функции из примера ниже.

|  |
| --- |
| public static int StrangeSum(int[] inputArray) {  int sum = 0;  for (int i = 0; i < inputArray.Length; i++)  {  for (int j = 0; j < inputArray.Length; j++)  {  for (int k = 0; k < inputArray.Length; k++)  {  int y = 0;   if (j != 0)  {  y = k / j;  }   sum += inputArray[i] + i + k + j + y;  }  }  }   return sum; } |

## 3. Реализуйте функцию вычисления числа Фибоначчи

Требуется реализовать рекурсивную версию и версию без рекурсии (через цикл).

Пример чисел Фибоначчи:

F(0) = 0,

F(1) = 1.

Для остальных чисел:

F(N) = F(N-2) + F(N-1).

То есть для F(2) будет F(2) = F(0) + F(1) = 0 + 1 = 1.

F(3) будет F(3) = F(1) + F(2) = 1 + 1 = 2.

# 

# Дополнительные материалы

1. [Переполнение стека](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81%D1%82%D0%B5%D0%BA%D0%B0).
2. [Вычислительная сложность](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C).
3. [Хвостовая рекурсия](https://ru.wikipedia.org/wiki/%D0%A5%D0%B2%D0%BE%D1%81%D1%82%D0%BE%D0%B2%D0%B0%D1%8F_%D1%80%D0%B5%D0%BA%D1%83%D1%80%D1%81%D0%B8%D1%8F).

# Используемые источники

1. Р. Стивенс. Алгоритмы. Теория и практическое применение. М.: Издательство «Э», 2016.
2. Н. Вирт. Алгоритмы и структуры данных. Новая версия для Оберона. М.: ДМК-пресс, 2010.
3. П. Дейтел, Х. Дейтел. C для программистов. С введением в C11. М.: ДМК-пресс, 2014.