## Оценка сложности

Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. В обоих случаях сложность зависит от размеров входных данных: массив из 100 элементов будет обработан быстрее, чем аналогичный из 1000. При этом точное время мало кого интересует: оно зависит от процессора, типа данных, языка программирования и множества других параметров. Важна лишь асимптотическая сложность, т. е. сложность при стремлении размера входных данных к бесконечности.

Допустим, некоторому алгоритму нужно выполнить 4n3 + 7n условных операций, чтобы обработать n элементов входных данных. При увеличении n на итоговое время работы будет значительно больше влиять возведение n в куб, чем умножение его на 4 или же прибавление 7n. Тогда говорят, что временная сложность этого алгоритма равна О(n3), т. е. зависит от размера входных данных кубически.

**Функция f(n)** имеет порядок O(g(n)) если имеется константа K и счетчик n0, такие, что f(n)<=K\*g(n), для n>=0;

Использование заглавной буквы О (или так называемая О-нотация) пришло из математики, где её применяют для сравнения асимптотического поведения функций. Формально O(f(n)) означает, что время работы алгоритма (или объём занимаемой памяти) растёт в зависимости от объёма входных данных не быстрее, чем некоторая константа, умноженная на f(n).

## Примеры

**O(n) — линейная сложность**

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

**O(log n) — логарифмическая сложность**

Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.

**O(n2) — квадратичная сложность**

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n \* n, т. е. n2.

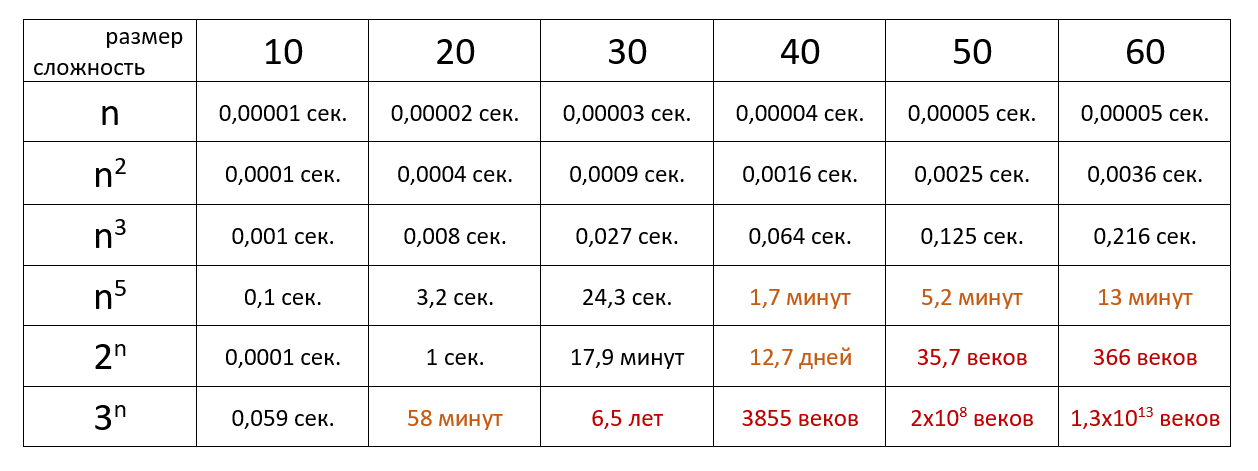
Бывают и другие оценки по сложности, но все они основаны на том же принципе.

Также случается, что время работы алгоритма вообще не зависит от размера входных данных. Тогда сложность обозначают как O(1). Например, для определения значения третьего элемента массива не нужно ни запоминать элементы, ни проходить по ним сколько-то раз. Всегда нужно просто дождаться в потоке входных данных третий элемент и это будет результатом, на вычисление которого для любого количества данных нужно одно и то же время.

Аналогично проводят оценку и по памяти, когда это важно. Однако алгоритмы могут использовать значительно больше памяти при увеличении размера входных данных, чем другие, но зато работать быстрее. И наоборот. Это помогает выбирать оптимальные пути решения задач исходя из текущих условий и требований.

## Наглядно

Время выполнения алгоритма с определённой сложностью в зависимости от размера входных данных при скорости 106 операций в секунду:



## Мой код

class Point

{

private float \_x1; // x1

private float \_x2; // x2

private float \_y1; // y1

private float \_y2; // y2

/// Конструктор 1 с параметрами

public Point(float value1, float value2, float value3, float value4)

{

set\_FirstPoint(value1, value2);

set\_SecondPoint(value3,value4);

}

/// Конструктор 2 без параметров

public Point()

{

}

// Сеттер x1,y1

public void set\_FirstPoint(float value1, float value2)

{

\_x1 = value1;

\_y1 = value2;

}

// get x1,y1 (кортеж)

public (float, float) get\_FirstPoint() { return (\_x1, \_y1); }

// сеттер x2,y2

public void set\_SecondPoint(float value1, float value2 )

{

\_x2 = value1;

\_y2 = value2;

}

// get x2,y2 (кортеж)

public (float, float) get\_SecondPoint() { return (\_x2, \_y2); }

// Процедура вычисления

public string calcDist()

{

// инициализация объекта таймера

Stopwatch stopWatch = new Stopwatch();

stopWatch.Start();

if ((\_x1 == 0) && (\_y1 == 0))

{

Console.WriteLine("1-я точка ближе");

stopWatch.Stop();

TimeSpan ts1 = stopWatch.Elapsed;

Console.WriteLine("Время выполнения функции: " + ts1.ToString() + '\n');

return ts1.ToString();

}

else

if ((\_x2 == 0) && (\_y2 == 0))

{

Console.WriteLine("2-я точка ближе");

stopWatch.Stop();

TimeSpan ts1 = stopWatch.Elapsed;

Console.WriteLine("Время выполнения функции: " + ts1.ToString() + '\n');

return ts1.ToString();

} else

if (((\_x1 \* \_x1) + (\_y1 \* \_y1)) < ((\_x2 \* \_x2) + (\_y2 \* \_y2)))

Console.WriteLine("1-я точка ближе");

else if (((\_x1 \* \_x1) + (\_y1 \* \_y1) == (\_x2 \* \_x2) + (\_y2 \* \_y2)))

Console.WriteLine("Точки равноудалены");

else Console.WriteLine("2-ая точка ближе");

// big O(16)

stopWatch.Stop();

TimeSpan ts = stopWatch.Elapsed;

Console.WriteLine("Время выполнения функции: "+ts.ToString()+'\n');

return ts.ToString();

}

}

class MainFunc

{

public static void Main(string [] args)

{

// инициализация переменных и обьекта класса

Point point = new Point();

float value = 0;

float value2 = 0;

int count = 0;

List<Point> listPoint = new List<Point>();

/// Создание файлового потока с созданием/открытием файла

StreamWriter fs = new StreamWriter("../../../out.txt");

fs.WriteLine("Напишите программу, определяющую какая из двух точек,\nзаданных своими координатами, расположена ближе к началу координат.\nКакой случай для алгоритма является наилучшим? Нанхудшим?\nСколько сравнений требуется этому алгоритму? Результаты тестирования вывести в текстовый файл.\n");

while (count<5) // делаем 5 кругов

{

count++;

Console.WriteLine($"Count {count} of 5\n");

// ввод точек x1,y1

Console.WriteLine("Input point x1, y1");

value = (float)Convert.ToDecimal(Console.ReadLine());

value2 = (float)Convert.ToDecimal(Console.ReadLine());

point.set\_FirstPoint(value, value2); // сеттер

// ввод точек x2,y2

Console.WriteLine("Input point x2, y2");

value = (float)Convert.ToDecimal(Console.ReadLine());

value2 = (float)Convert.ToDecimal(Console.ReadLine());

point.set\_SecondPoint(value, value2); // сеттер

listPoint.Add(point); // Добавим в коллекцию ссылку на обьект

string str = point.calcDist(); // Расчёт + получение данных о времени выполнения работы

fs.WriteLine ($"Время выполнения функции, проход {count}, точки x1, y1 {point.get\_FirstPoint()}, точки x2,y2 {point.get\_SecondPoint()}: " + str);

}

fs.Close();

}

}

Лучший случай для моей программы – когда точки находятся на координатах (0;0) (0;0).

Худший – любое отличное от этого значение. В целом **алгоритм выполнится за константное время O(16);**

**Ссылки – Структуры и алгоритмы обработки данных в С++ У. Топп**

**https://tproger.ru/articles/computational-complexity-explained/**