Материал на 1-2 лекции

Асимптотический анализ

Когда мы говорим об измерении сложности алгоритмов, мы подразумеваем анализ времени, которое потребуется для обработки очень большого набора данных. Такой анализ называют асимптотическим. Сколько времени потребуется на обработку массива из десяти элементов? Тысячи? Десяти миллионов? Если алгоритм обрабатывает тысячу элементов за пять миллисекунд, что случится, если мы передадим в него миллион? Будет ли он выполняться пять минут или пять лет?

Порядок роста описывает то, как сложность алгоритма растет с увеличением размера входных данных. Чаще всего он представлен в виде O-нотации *(от нем. «Ordnung» — порядок)* : *O(f(x))*, где *f(x)* — формула, выражающая сложность алгоритма. В формуле может присутствовать переменная n, представляющая размер входных данных. Ниже приводится список наиболее часто встречающихся порядков роста, но он ни в коем случае не полный.

**Абстрактная сложность на практике**

Говоря об абстрактной сложности, чаще всего рассматривают три разных варианта, особенности каждого из которых следует четко понимать.

**Средняя сложность**, ожидаемое среднее время или требуемый в среднем объем памяти алгоритма. Как отмечалось ранее, говорить о среднем можно только в предположении о существовании случайного распределения для входов программы. Обычно *среднюю сложность* считают в предположении, что все входы равновероятны (например, при сортировке массива из n элементов можно считать, что все n!возможных упорядочений элементов могут появляться с равной вероятностью).

**Максимальная сложность**, также называемая *сложностью в худшем случае*, дающая время или память для случая, на котором алгоритм работает дольше всего (требует максимальной памяти).

**Минимальная сложность**, также называемая *сложностью в лучшем случае*, дающая время или память для случая, на котором алгоритм работает быстрее всего (требует минимальной памяти). Этот критерий используется редко – его любят те, кто верит в удачу.

#### **Константный — O(1)**

Порядок роста *O(1)* означает, что вычислительная сложность алгоритма не зависит от размера входных данных. Следует помнить, однако, что единица в формуле не значит, что алгоритм выполняется за одну операцию или требует очень мало времени. Он может потребовать и микросекунду, и год. Важно то, что это время не зависит от входных данных.

(например, вернуть длину массива, или изменение элемента в массиве. Пусть сами приведут примеры).

#### **Линейный — O(n)**

Порядок роста *O(n)* означает, что сложность алгоритма линейно растет с увеличением входного массива. Если линейный алгоритм обрабатывает один элемент пять миллисекунд, то мы можем ожидать, что тысячу элементов он обработает за пять секунд.

Такие алгоритмы легко узнать по наличию цикла по каждому элементу входного массива.

(например – сумма всех элементов в массиве)

public long GetSum(int[] items)

{

    long sum = 0;

    foreach (int i in items)

    {

        sum += i;

    }

    return sum;

}

копирование массива – O(n)

#### **Логарифмический — O( log n)**

Порядок роста *O(* log*n)* означает, что время выполнения алгоритма растет логарифмически с увеличением размера входного массива. (*Прим. пер.:* в анализе алгоритмов по умолчанию используется логарифм по основанию 2). Большинство алгоритмов, работающих по принципу «деления пополам», имеют логарифмическую сложность. Метод Contains бинарного дерева поиска *(binary search tree)* также имеет порядок роста *O(*log*n)*.

#### **Линеарифметический — O(n·log n)**

Линеарифметический (или линейно-логарифмический) алгоритм имеет порядок роста *O(n·*log*n)*. Некоторые алгоритмы типа «разделяй и властвуй» попадают в эту категорию. В следующих частях мы увидим два таких примера — сортировка слиянием и быстрая сортировка.

#### **Квадратичный — O(n 2)**

Время работы алгоритма с порядком роста *O(n 2)* зависит от квадрата размера входного массива. Несмотря на то, что такой ситуации иногда не избежать, квадратичная сложность — повод пересмотреть используемые алгоритмы или структуры данных. Проблема в том, что они плохо масштабируются. Например, если массив из тысячи элементов потребует  
1 000 000 операций, массив из миллиона элементов потребует 1 000 000 000 000 операций. Если одна операция требует миллисекунду для выполнения, квадратичный алгоритм будет обрабатывать миллион элементов 32 года. Даже если он будет в сто раз быстрее, работа займет 84 дня.

Показать картинку - Comparison\_computational\_complexity.svg

Здесь можно рассказать пример про молекулярную динамику и деление на сетку (но не обязательно)

Сортировки:

Пузырьком O(n2)

Глупая O(n3)

Гномья O(n2)

Вставками O(n2)

(пузырьковая сортировка, гномья сортировка)

взаимодействие частиц каждая с каждой – O(n2)

для случайной сортировки СРЕДНЯЯ СЛОЖНОСТЬ составляет:

*Для O*((*n*+1)!)[[1]](https://en.wikipedia.org/wiki/Bogosort#cite_note-Fun07-1)

Самое важное здесь то, что *O(n)* означает, что алгоритм потребует **не более** *n* шагов.

При измерении сложности алгоритмов и структур данных мы обычно говорим о двух вещах: количество операций, требуемых для завершения работы (вычислительная сложность), и объем ресурсов, в частности, памяти, который необходим алгоритму (пространственная сложность).

Алгоритм, который выполняется в десять раз быстрее, но использует в десять раз больше места, может вполне подходить для серверной машины с большим объемом памяти. Но на встроенных системах, где количество памяти ограничено, такой алгоритм использовать нельзя.

Когда мы говорим, что время поиска элемента в списке из count элементов составляет O(count), это означает, что с ростом count оно возрастает, в худшем случае, **пропорционально** count. Другая операция может иметь время O(count^2), означающее, что время возрастает самое большее пропорционально квадрату числа элементов. Те же соглашения действуют при оценке требуемой памяти.

Для такой меры:

* константные мультипликативные множители не играют роли: O(100* count^2) означает то же самое, что и O(count^2). Объяснение этого соглашения состоит в том, что не следует придавать большого значения умножению на константу, так как реализация того же алгоритма может быть в сто раз быстрее или медленнее, будучи перенесенной на другую машину. Но рост времени вычислений при изменении count не зависит от таких технических деталей;
* понятно, что не имеет значения и константная аддитивная составляющая: O(count^2 + 1000) означает то же самое, что и O(count^2). Конечно, константа может оказывать влияние при небольших значениях count, но с его ростом влияние становится ничтожным;
* аналогично: любая аддитивная составляющая с меньшим значением экспоненты также не оказывает влияния на порядок величины – O(count^3 + count^2) означает то же самое, что и O(count^3).

Как следствие, чтобы выразить тот факт, что алгоритм работает константное время, более точно – что на любой платформе время выполнения ограничено константой, будем говорить, что время работы O(1). Конечно с тем же успехом можно писать O(37) или O(1000), но принято писать O(1).

Дальше рассказать про бинарный поиск и ориентироваться по программам:

Прога 1 - бинарный поиск

Прога 2 - поиск фамилии

Задание – дописать бинарный поиск и попробовать подавать туда массив с текстом