Оглавление

Тема 1. Алгоритм. Оценка сложности алгоритма

I. Алгоритм: Определение и основные понятия

Алгоритм – это четкая последовательность действий, которые необходимо совершить для решения определенной задачи.

Алгоритмы используются повсеместно, от математики и информатики до географии и экономики. Они являются основой программирования и обработки данных. Все компьютерные программы, от простейших утилит до сложных систем искусственного интеллекта, базируются на алгоритмах.

Примеры алгоритмов из повседневной жизни могут включать рецепты приготовления блюд, инструкции по сборке мебели и т.д.

Алгоритмы имеют несколько основных свойств, которые помогают нам понять их функционирование и эффективность.

Выделяют следующие свойства алгоритма:

1. **Точность:** Каждый шаг в алгоритме должен быть четко определен и пониматься однозначно.
2. **Дискретность:** Алгоритм должен быть разбит на последовательность выполняемых шагов.
3. **Детерминированность:** При одинаковых входных данных алгоритм должен всегда давать один и тот же результат.
4. **Конечность (завершаемость):** Алгоритм должен завершиться после конечного числа шагов.
5. **Массовость:** Алгоритм должен быть применим к широкому классу аналогичных задач. Это означает, что алгоритм должен иметь возможность обрабатывать различные входные данные.
6. **Результативность:** Алгоритм должен решать поставленную задачу. То есть, после выполнения алгоритма должен быть получен результат, удовлетворяющий условиям задачи.
7. **Понятность:** Каждый шаг в алгоритме должен быть понятен исполнителю.

II. Алгоритм: Сложность алгоритма

Сложность алгоритма – это количественная мера того, как быстро работает алгоритм или сколько ресурсов он использует (например, время и память). Сложность алгоритма обычно выражается как функция от размера входных данных.

Временная сложность показывает количество времени, необходимое алгоритму для выполнения, как функцию, зависящую от размера входных данных. Временная сложность позволяет оценить, насколько быстро или медленно будет работать алгоритм на практике. Например, при удвоении размера входных данных алгоритм с временной сложностью $O(n)$ будет выполняться в 2 раза медленнее.

Пространственная сложность показывает количество памяти, которое требуется алгоритму для выполнения, как функцию, зависящую от размера входных данных.

Существуют также другие виды оценки сложности алгоритмов, такие как сложность по пропускной способности, сложность по параллелизму и другие, но они меньше

используются и обычно более специфичны для конкретных приложений или типов оборудования.

Оценка сложности алгоритма важна при проектировании и анализе алгоритмов, т.к. позволяет сравнивать эффективность различных алгоритмов и выбирать наиболее подходящий алгоритм для конкретной задачи.

Временная сложность обычно измеряется в терминах **элементарных операций**, которые должен выполнить алгоритм.

Элементарными операциями в алгоритмах и программировании называются простейшие действия, которые выполняются за фиксированное количество времени и не зависят от размера входных данных.

Подсчет элементарных операций выполняется с использованием вычислительной модели Random Access Machine (RAM).

Модель Random Access Machine (RAM), или модель с произвольным доступом к памяти, является теоретической моделью вычислений, которая используется при анализе сложности алгоритмов.

В модели RAM каждая элементарная операция требует фиксированное и одинаковое количество времени. Это предположение помогает упростить анализ сложности алгоритмов.

Вот некоторые примеры элементарных операций:

1. **Арифметические операции:** Это включает в себя базовые арифметические операции, такие как сложение, вычитание, умножение, деление и взятие остатка от деления.
2. **Операции сравнения:** Сюда входят операции, которые сравнивают два значения, такие как "меньше", "больше", "равно", "не равно".
3. **Операции присваивания:** Это операции, которые присваивают значение переменной.
4. **Операции доступа к данным:** Включают в себя чтение и запись в память, в частности доступ к элементам массива или чтение и изменение полей объекта.
5. **Базовые логические операции:** Логические операции, такие как AND, OR, NOT.

Важные особенности модели RAM:

1. **Память модели RAM не ограничена**. Это значит, что алгоритм может использовать столько памяти, сколько ему нужно, и время доступа к любой ячейке памяти является постоянным.
2. **В модели RAM** каждая **инструкция** выполняется **последовательно**, одна за другой. Это отличает модель RAM от более сложных моделей, которые могут учитывать параллельное и конкурентное выполнение инструкций.
3. В модели RAM предполагается, что все элементарные операции выполняются **за фиксированное количество времени**.

Предположение, что каждая элементарная операция занимает постоянное (и равное) время позволяет нам фокусироваться на общей структуре алгоритма и его поведении при увеличении размера входных данных, а не на деталях конкретной реализации.

Пример. Посчитать число элементарных операций во приведенном кода.

```
import sys
min_value = sys.maxsize #максимально возможное значение
max_value = -sys.maxsize - 1 #минимально возможное значение

for x in array:
    if x < min_value:
        min_value = x
    if x > max_value:
        max_value = x
```

Данный код выполняет поиск минимального и максимального значения в списке `array`. Давайте посчитаем количество элементарных операций.

1. Присваивание `min_value` и `max_value` на первых двух строках: 2 операции.
2. Также на первой строке `sys.maxsize` происходит обращение к памяти: 1 операции
3. На второй строке `sys.maxsize` происходит обращение к памяти, отрицание числа и вычитание: 3 операции
4. Цикл `for`: этот цикл выполняется n раз, где n – количество элементов в `array`.

   Внутри цикла:

   – Происходит операция сравнения `x < min_value`: 1 операция.
   – Если условие `x < min_value` выполняется, происходит присваивание `min_value = x`: 1 операция.
   – Операция сравнения `x > max_value`: 1 операция.
   – Если условие `x > max_value` выполняется, происходит присваивание `max_value = x`: 1 операция.

   Итого внутри цикла, максимально может выполниться 3 операции для каждого элемента (две операции сравнение и одна операция присваивания).

Итого, количество элементарный операций составляет $f(n) = 2 + 1 + 3 + 3n = 6 + 3n$, где n – это количество элементов в `array`. Однако, стоит отметить, что реальное количество выполненных операций может быть меньше $6 + 3n$, т.к. условия `x < min_value` и `x > max_value` не обязательно выполняются для каждого элемента.

В примере выше $f(n)$ – функция трудоемкости, определяющая зависимость между размером входных данными и количеством элементарных операций (временными затратами алгоритма).

Пример. Посчитать число элементарных операций во приведенном кода.

```
import sys
min_value = sys.maxsize #максимально возможное значение
max_value = -sys.maxsize - 1 #минимально возможное значение

for x in array:
    if x < min_value:
        min_value = x
for x in array:
    if x > max_value:
        max_value = x
```

Данный код выполняет поиск минимального и максимального значения в списке `array`. Давайте рассмотрим количество элементарных операций.

1. Присваивание `min_value` и `max_value` на первых двух строках: 2 операции.
2. Также на первой строке `sys.maxsize` происходит обращение к памяти: 1 операции
3. На второй строке `sys.maxsize` происходит обращение к памяти, отрицание числа и вычитание: 3 операции
4. Первый цикл `for`: этот цикл выполняется n раз, где n – количество элементов в `array`.

   Внутри цикла:

   - Происходит операция сравнения `x < min_value`: 1 операция.
   - Если условие `x < min_value` выполняется, происходит присваивание `min_value = x`: 1 операция.

   Итого внутри первого цикла $2n$ операций.

5. Второй цикл `for`: также выполняется n раз, где n – количество элементов в `array`.

   Внутри цикла:

   - Происходит операция сравнения `x > max_value`: 1 операция.
   - Если условие `x > max_value` выполняется, происходит присваивание `max_value = x`: 1 операция.

   Итого внутри второго цикла $2n$ операций.

Итого, количество элементарный операций составляет $f(n) = 2 + 1 + 3 + 2n + 2n = 6 + 4n$, где n – это количество элементов в `array`. Однако, стоит отметить, что реальное количество выполненных операций может быть меньше $6 + 4n$, т.к. условия `x < min_value` и `x > max_value` не обязательно выполняются для каждого элемента.

Однако, подобный расчёт количества элементарных операций достаточно трудоемок и поэтому говорят об **асимптотической оценке сложности алгоритма** (асимптотика).

В контексте анализа сложности алгоритмов асимптотика помогает описать, как изменяется время выполнения алгоритма или объем занимаемой им памяти при увеличении размера входных данных.

Основные виды асимптотических нотаций включают:

1. ***O*-большое (Big O, оценка сверху)**: Описывает верхнюю границу сложности алгоритма, то есть максимальное время выполнения или объем занимаемой памяти. Например, если алгоритм имеет временную сложность $O(n)$, это означает, что его время выполнения будет расти максимум линейно с увеличением размера входных данных.
2. ***Ω*-большое (Big Omega, оценка снизу)**: Описывает нижнюю границу сложности алгоритма, то есть минимальное время выполнения или объем занимаемой памяти. Если алгоритм имеет сложность $\Omega(n)$, это означает, что его время выполнения будет расти как минимум линейно при увеличении размера входных данных.
3. ***Θ*-большое (Big Theta)**: Сочетает в себе O-большое и Ω-большое, описывая алгоритмы, у которых верхняя и нижняя границы совпадают. Если алгоритм имеет сложность $\Theta(n)$, это означает, что время его выполнения будет расти линейно при увеличении размера входных данных, и это будет являться и верхней, и нижней границей.

Для перехода от подсчета числа элементарных операций к асимптотической оценке сложности применяют следующие упрощения:

1. У функции, представляющей зависимость количества элементарный операций от размера входных данных, оценивают только самую быстрорастущую часть.
2. Игнорируют коэффициенты.

Пример. Переход от функции количество элементарный операций к асимптотике.

В примерах выше количество элементарных операций выражалось формулами $6 + 3n$ и $6 + 4n$, соответственно. Говоря об оценке худшего случая, т.е. максимального количества возможных операций и используя упрощения, определенные выше, приходим к асимптотике $O(n)$. Таким образом, с точки зрения асимптотики обе функции эквивалентны.

III. *O*-большое (Big O, оценка сверху)

«*O*-большое» (Big O) – это асимптотическая нотация, используемая для описания верхней границы сложности алгоритма. Это помогает оценить наихудшую производительность алгоритма, то есть максимальное время выполнения или объем занимаемой памяти в зависимости от размера входных данных.

Другими словами, если алгоритм имеет временную сложность $O(g(n))$, это означает, что время выполнения алгоритма не будет расти быстрее, чем функция $g(n)$, когда размер входных данных (n) стремится к бесконечности.

Пусть $f(n)$ – число элементарных операций работы некоторого алгоритма. Тогда условие $f(n) = O(g(n))$ можно записать в виде математической зависимости:

$$f(N) \in O(g(n)), \exists(C > 0), n_0 : \forall(n > n_0)\ |f(n)| \leq |Cg(n)|$$

$O(g(n))$ обозначает класс функций, таких, что все они растут не быстрее, чем функция $g(n)$ с точностью до положительного постоянного множителя. Говорят, что функция $g(n)$ мажорирует функцию $f(n)$. Зависимость можно отобрать на графике (рисунок 1).

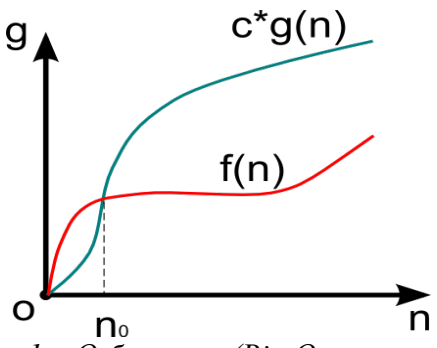

Рисунок 1 – O-большое (Big O, оценка сверху)

Пример. Оценки сложности для некоторых функций.

Пусть имеются следующие функции $f_1(n) = 10$, $f_2(n) = 3n + 10$, $f_3(n) = n^2 + 4$, $f_4(n) = 2^n + 10$. Для всех этих функций будет справедлива оценка $O(2^n)$. Однако, следует указывать наиболее «близкую» мажорирующую функцию, т.к. оценка $O(2^n)$ не позволяет разделить эти алгоритмы и практически она малопригодна.

Пример. Доказать, что $n^2 + 2n + 1 = O(n^2)$.

Построим график роста функций, входящих в состав выражения $n^2 + 2n + 1$ (рисунок 2).

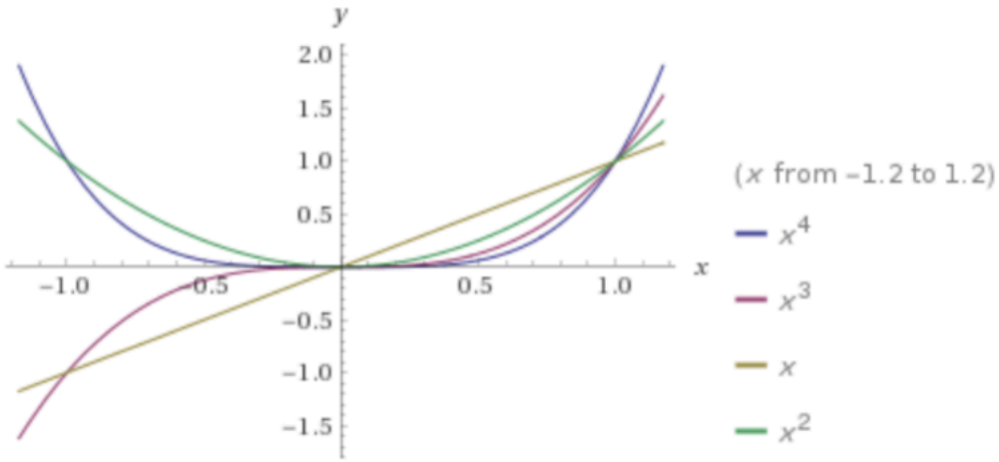

Рисунок 2 – График роста полиномиальных функций

Заметим, что при $n \geq 1$, $n^2 > n$ и $n^2 > 1$.

Преобразуем исходную функцию:

$$n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 \leq 4n^2 \leq Cn^2$$

Воспользуемся определением $f(N) \in O(g(n)), \exists(C > 0), n_0 : \forall(n > n_0)\ |f(n)| \leq |Cg(n)|$. Таким образом, при $C = 4$, начиная с $n_0 = 1$: $n^2 + 2n + 1 \leq 4n^2$. Ч.т.д.

Пример. Оценить сложность кода с точки зрения *O*-большое

```
i = 0
while i < n and A[i] != x:
    i += 1

if i < n:
    print('Yes')
else:
    print('No')
```

Этот код выполняет линейный поиск значения `x` в списке `A` размером `n`.

Рассмотрим код построчно:

1. Инициализация переменной `i`: это константное время, $O(1)$.
2. Цикл `while`: в худшем случае, когда значение `x` отсутствует в списке или находится в его конце, цикл будет выполняться `n` раз. Каждая итерация включает сравнение и, возможно, инкрементацию, каждое из которых выполняется за константное время. Таким образом, общая сложность выполнения этого цикла – $O(n)$.
3. Последнее условие `if i < n` выполняется за константное время, $O(1)$.

При сложении этих сложностей, доминирующим членом будет $O(n)$, так как остальные члены ($O(1)$) становятся незначительными при больших `n`.

Таким образом, асимптотическая сложность всего кода с точки зрения O-большое будет $O(n)$.

Пример. Метод сортировки со сложностью $\boldsymbol{O(n \log n)}$ тратит ровно 1 миллисекунду на сортировку 1 000 элементов данных. Предполагая, что время $\boldsymbol{T(n)}$ сортировки $\boldsymbol{n}$ элементов прямо пропорционально $\boldsymbol{n \log n}$, то есть $\boldsymbol{T(n)= cn \log n}$, выведите формулу для $\boldsymbol{T(n)}$. Учитывая время $\boldsymbol{T(N)}$ для сортировки $\boldsymbol{N}$ оцените, как долго этот метод будет сортировать 1 000 000 элементов.

Выразим из формулы $T(n)= cn \log n$ константу c:

$$c = \frac{T(n)}{n \log n}$$

Подставим константу в формулу для расчета $T(N)$:

$$T(N) = cN \log N = \frac{T(n)}{n \log n} N \log N$$

Теперь оценим время сортировки 1 000 000 элементов. Подставляем $n = 1\,000$ и $N = 1\,000\,000$ в полученную формулу:

$$T(N) = \frac{1}{1000 \log 1000} 1000000 \log 1000000 = \frac{6000000}{3000} = 2000 \text{ миллисекунд}$$

В силу свойств логарифма мы можем взять любое основание (для удобства расчетов взяли логарифм по основанию 10).

Таким образом, данный алгоритм сортировки должен сортировать 1 000 000 элементов примерно за 2 секунды.

Пример. Предположим, что каждое из приведенных ниже выражений дает время обработки T(n), затраченное алгоритмом на решение задачи размера n. Выберите доминирующие члены, имеющие наибольший рост n, и укажите наименьшую сложность Big-Oh для каждого алгоритма.

Выражение	Доминирующий член	$O(\dots)$
$5 + 0.001n^3 + 0.025n$		
$500n + 100n^{1.5} + 50n\log_{10} n$		
$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$		
$n^2 \log_2 n + n\,(\log_2 n)^2$		
$n\log_3 n + n\log_2 n$		
$3\log_3 n + \log_2\log_2\log_2 n$		
$100n + 0.01n^2$		
$0.01n + 100n^2$		
$2n + n^{0.5} + 0.5n^{1.25}$		
$0.01n\log_2 n + n\,(\log_2 n)^2$		
$100n\log_3 n + n^3 + 200n$		
$0.003\log_4 n + \log_2\log_2 n$		

В выражении $5 + 0.001n^3 + 0.025n$ доминирующим членом будет $0.001n^3$, так как это член с самой высокой степенью n.

Сложность Big-Oh для этого алгоритма будет $O(n^3)$, так как мы выбираем наибольший рост n (n^3) в выражении в качестве доминирующего члена и игнорируем остальные, более слабые слагаемые (5 и $0.025n$), так как они малы по сравнению с $0.001n^3$ при достаточно больших значениях n.

Аналогично для других выражений. Заполненная таблица будет выглядеть следующим образом.

Выражение	Доминирующий член	$O(\dots)$
$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
$500n + 100n^{1.5} + 50n\log_{10} n$	$100n^{1.5}$	$O(n^{1.5})$
$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$	$2.5n^{1.75}$	$O(n^{1.75})$
$n^2 \log_2 n + n\,(\log_2 n)^2$	$n^2 \log_2 n$	$O(n^2 \log n)$
$n\log_3 n + n\log_2 n$	$n\log_3 n\,, n\log_2 n$	$O(n\log n)$
$3\log_3 n + \log_2\log_2\log_2 n$	$3\log_3 n$	$O(\log n)$
$100n + 0.01n^2$	$0.01n^2$	$O(n^2)$
$0.01n + 100n^2$	$100n^2$	$O(n^2)$
$2n + n^{0.5} + 0.5n^{1.25}$	$0.5n^{1.25}$	$O(n^{1.25})$
$0.01n\log_2 n + n\,(\log_2 n)^2$	$n\,(\log_2 n)^2$	$O(n(\log n)^2)$
$100n\log_3 n + n^3 + 200n$	n^3	$O(n^3)$
$0.003\log_4 n + \log_2\log_2 n$	$0.003\log_4 n$	$O(\log n)$

Пример. Алгоритмы A и B тратят ровно $T_A(n) = c_A n log_2 n$ и $T_B(n) = c_B n^2$ микросекунды соответственно для задачи размера n. Найдите лучший алгоритм для обработки n=2^{20} элементов данных, если алгоритм A тратит 10 микросекунд на обработку 1024 элементов, а алгоритм B тратит всего 1 микросекунду на обработку 1024 элементов.

Найдём коэффициенты c_A, c_B. Выразим их через известные нам формулы времени:

$$c_A = \frac{T_A(n)}{nlog_2 n};\ c_B = \frac{T_B}{n^2}$$

Теперь подставим числа:

$$c_A = \frac{10}{1024log_2 1024} = \frac{1}{1024};\ c_B = \frac{1}{1024^2}$$

Теперь мы можем вычислить время выполнения алгоритмов A и B для n=2^{20}:

$$T_A(2^{20}) = \frac{1}{1024}2^{20}\log_2(2^{20}) = 20280ms$$

$$T_B(2^{20}) = \frac{1}{1024^2}2^{40} = 2^{20}ms = 1048576ms$$

Таким образом, для обработки 2^{20} элементов данных алгоритм A значительно лучше, так как он затрачивает всего 20280 микросекунды на выполнение, в то время как алгоритм B требует 1048576 микросекунды.

Пример. Один из двух пакетов программного обеспечения, A или B, должен быть выбран для обработки очень больших баз данных, каждая из которых содержит до 10^{12} записей. Среднее время обработки пакетом A составляет $T_A(n) = 0,1nlog_2 n$ микросекунд, а среднее время обработки пакетом B составляет $T_B(n)$=$5n$ микросекунд. Какой алгоритм имеет лучшую производительность в смысле «Big-O»? Определите точные условия когда эти пакеты превосходят друг друга.

Сравнивая их асимптотические сложности, получаем:

Пакет A: $O(n\ log\ n)$
Пакет B: $O(n)$
Следовательно, с точки зрения "Big-O", пакет A имеет более высокую производительность в сравнении с пакетом B при обработке очень больших баз данных.

Что касается точных условий, при которых один пакет превосходит другой:

Пакет A будет более производительным, если "$n\ log\ n$" растет быстрее, чем "n", то есть когда n увеличивается значительно. Это происходит, например, когда n достаточно большое.

Пакет B будет более производительным, если "n" растет быстрее, чем "$n\ log\ n$". Это может быть случай, когда n мало.

Таким образом, для маленьких значений n, пакет B может быть более производительным. Но по мере роста n, пакет A начнет проявлять свою преимущественную производительность, так как асимптотически его сложность выше, чем у пакета B.

Пример. Пусть время обработки алгоритма сложности Big-O $(f(n))$ прямо пропорционально $f(n)$. Пусть три таких алгоритма A, B и C имеют временную сложность $O\ (n^2)$, $O\ (n^{1.5})$ и $O\ (n\ log\ n)$ соответственно. Во время теста

каждый алгоритм тратит 10 секунд на обработку 100 элементов данных. Найдите время, которое каждый алгоритм потратит на обработку 10000 элементов.

Рассмотрим алгоритм A с временной сложностью $O(n^2)$. Если время обработки 100 элементов данных составляет 10 секунд, то для 10000 элементов данные алгоритм A потребует времени, пропорционального квадрату отношения количества элементов:
$(\frac{10000}{100})^2 * 10 = 100000$ секунд
Для алгоритма B с временной сложностью $O(n$^1.5), аналогично:
$(\frac{10000}{100})^{1.5} * 10 = 10000$
И наконец, для алгоритма C с временной сложностью $O(n\log n)$:
$\left(\frac{10000}{100}\right) * log_2\left(\frac{10000}{100}\right) * 10 = 2000$ секунд.
Итак, время, которое каждый алгоритм потратит на обработку 10000 элементов данных, будет примерно:
• Алгоритм A: 100000 секунд
• Алгоритм B: 10000 секунды

• Алгоритм C: 2000 секунд.

Пример. Посчитать асимптотическую сложность с точки зрения О-большое

```
def sum(n):

    if n == 1:

        return 1

        return n + sum(n - 1)
```

Сложность этой функции можно оценить как O(n), где n – количество рекурсивных вызовов (или точнее, в данном случае, количество итераций). Это означает, что время выполнения функции будет пропорционально размеру входных данных n.

Пример. Посчитать асимптотическую сложность с точки зрения О-большое

```
def pairSumSequence(n):
    sum = 0
    for i in range(n):
        sum += pairSum(i, i + 1)
    return sum

def pairSum(a, b):
    return a + b
```

Здесь аналогично. Цикл выполняется n раз. Следовательно, общая сложность выполнения данного кода – $O(n)$

Пример. Посчитать асимптотическую сложность с точки зрения О-большое

```
for a in arrA:
    print(a)

for b in arrB:
    print(b)


for a in arrA:
    for b in arrB:
        print(str(a) + "," + str(b))
```

Первый фрагмент:

Просто два цикла, первый для `arrA`, второй для `arrB`.
Каждый цикл выполняется по размеру соответствующего массива.
Сложность второго фрагмента: $O(n + m)$, где n – размер `arrA`, m – размер `arrB`.
Обратите внимание, что при анализе сложности для двух вложенных циклов, важно учитывать размерности массивов и количество итераций вложенных циклов.

Второй фрагмент:

Вложенный цикл (`for b in arrB`) выполняется для каждого элемента a из `arrA`, и внутри этого цикла выполняется операция вывода.
Если размер `arrA` равен n, а размер `arrB` равен m, то операция вывода будет выполняться $n * m$ раз.

Сложность первого фрагмента: $O(n * m)$, где n – размер `arrA`, m – размер `arrB`.

Итак, общая сложность второго фрагмента кода – $O(n * m)$, а первого фрагмента – $O(n + m)$.

Пример. Посчитать асимптотическую сложность с точки зрения О-большое

```
def foo(array):
    sum = 0
    product = 1
    for i in range(len(array)):
        sum += array[i]
    for i in range(len(array)):
        product *= array[i]
    print(str(sum) + "," + str(product))
```

Общая сложность:

Первый цикл: $O(n)$
Второй цикл: $O(n)$
Операция вывода: $O(1)$

Так как все операции выполняются последовательно, общая сложность кода будет максимальной сложностью операций, то есть $O(n) + O(n) + O(1)$, что просто упрощается до $O(n)$.

Итак, общая сложность выполнения данного кода - $O(n)$.

Пример. Посчитать асимптотическую сложность с точки зрения О-большое

```
def printPairs(arr):
    for i in range(len(arr)):
        for j in range(len(arr)):
            print(str(arr[i]) + "," + str(arr[j]))
```

Данный код содержит два вложенных цикла. Первый цикл `(for i in range(len(arr)))` выполняется n раз, где n – длина массива `arr`. Внутри первого цикла есть второй цикл `(for j in range(len(arr)))`, который также выполняется n раз. Внутри второго цикла выполняется операция вывода.

Таким образом, операция вывода будет выполнена $n * n = n^2$ раз.

Следовательно, сложность выполнения данного кода составляет $O(n^2)$, где n - длина массива `arr`.

Пример. Посчитать асимптотическую сложность с точки зрения О-большое

```
for i in range(N):
    for j in range(N):
        foo()

for i in range(N):
    for j in range(i, N):
        foo()
```

Первый фрагмент:

У вас есть два вложенных цикла. Оба цикла выполняются N раз, где N – заданное значение. Внутри второго цикла вызывается функция `foo()`, но для анализа сложности нам нужно учесть только количество итераций.

Общая сложность первого фрагмента: $O(N^2)$.

Второй фрагмент:

Также есть два вложенных цикла. Внешний цикл выполняется N раз, и внутри него второй цикл начинается с i и выполняется до N. Внутри второго цикла также вызывается функция `foo()`.

Этот второй фрагмент несколько эффективнее первого, так как во втором цикле сокращается количество итераций во внутреннем цикле. Однако для анализа сложности мы все равно должны учесть весь потенциальный диапазон итераций.

Общая сложность второго фрагмента: $O(N^2)$.

Итак, оба фрагмента имеют сложность $O(N^2)$.

Рисунок 3. Сложность выполнения кода для первого и второго фрагментов.

Пример. Посчитать асимптотическую сложность с точки зрения О-большое

```
def printPairs(arrA, arrB):
    for i in range(len(arrA)):
        for j in range(len(arrB)):
            if arrA[i] < arrB[j]:
                print(str(arrA[i]) + "," + str(arrB[j]))
```

```
def printPairs(arrA, arrB):
    for i in range(len(arrA)):
        for j in range(len(arrB)):
            for k in range(100000):
                print(str(arrA[i]) + "," + str(arrB[j]))
```

Оба эти кода выполняют операцию вывода на экран всех возможных пар элементов из двух массивов `arrA` и `arrB`. Однако они имеют различия в том, как происходит итерация и как это влияет на их временную сложность.

В первом коде два вложенных цикла `for` используются для перебора всех пар элементов из массивов `arrA` и `arrB`. Условие `if arrA[i] < arrB[j]` гарантирует, что пара будет выведена только в том случае, если элемент из `arrA` меньше элемента из `arrB`. Временная сложность данного кода составляет $O(n^2)$, где n – количество элементов в массиве. Это потому, что для каждого элемента из `arrA` выполняется внутренний цикл `for`, который проходит по всем элементам `arrB`.

Второй код также имеет два вложенных цикла `for`, но также добавляется дополнительный внутренний цикл `for k in range(100000)`. Это означает, что для каждой пары элементов из `arrA` и `arrB`, цикл `for k` будет выполняться 100000 раз. Временная сложность данного кода составляет $O(n^2 * 100000)$, что эквивалентно $O(n^2)$, так как константный множитель 100000 играет небольшую роль в анализе сложности большого размера данных.

В заключение, оба кода имеют временную сложность $O(n^2)$, но второй код будет немного медленнее из-за дополнительного внутреннего цикла `for k`.

Пример. Посчитать асимптотическую сложность с точки зрения О-большое

```
def reverse(arr):
    for i in range(len(arr) // 2):
        other = len(arr) - i - 1
        temp = arr[i]
        arr[i] = arr[other]
        arr[other] = temp
```

Функция reverse выполняет обращение (инвертирование) массива `arr`. Она проходит через массив до его середины и меняет местами элементы симметрично относительно середины.

Посмотрим на этапы выполнения функции:

Операция `temp = arr[i]` и `arr[i] = arr[other]` занимают постоянное время $O(1)$.

Функция выполняет цикл по половине длины массива, то есть `len(arr) // 2` раз.

Все операции внутри цикла выполняются за постоянное время.

Следовательно, временная сложность этой функции можно считать $O(n)$, где n – длина массива. Это потому, что функция выполняет фиксированное количество операций для каждого элемента массива, и это количество операций линейно зависит от размера массива.

IV. Алгоритм: Двоичный поиск

.

	0	1	2	3	4	5	6	7	8	9
Ищем 23. Так как 23>16 берем вторую половину.	2	5	8	12	16	23	38	56	72	91
23<56 значит берем первую половину из оставшегося массива						23	38	56	72	91
23<38, и так как остался всего один вариант всех ячеек массива – выбираем его. Искомое число было под номером 5.						23	38			
Готово!						23				

В верхнем ряду на изображении мы видим отсортированный массив. В нем нам необходимо найти число 23. Вместо того, чтобы перебирать числа, мы просто делим массив на 2 части и проверяем среднее число в массиве. Находим число, которое располагается в ячейке 4 и проверяем его (второй ряд на картинке). Это число равно 16, а мы ищем 23. Текущее число меньше. Что это означает? Что все предыдущие числа (которые расположены до числа 16) можно не проверять: они точно будут меньше того, которое мы ищем, ведь наш массив отсортирован! Продолжим поиск среди оставшихся 5 элементов.

Обрати внимание: мы сделали всего одну проверку, но уже отмели половину возможных вариантов. У нас осталось всего 5 элементов. Мы повторим наш шаг — снова разделим оставшийся массив на 2 и снова возьмем средний элемент (строка 3 на рисунке). Это число 56, и оно больше того, которое мы ищем. Что это означает? Что мы отметаем еще 3 варианта — само число 56, и два числа после него (они точно больше 23, ведь массив отсортирован). У нас осталось всего 2 числа для проверки (последний ряд на рисунке) — числа с индексами массива 5 и 6. Проверяем первое из них, и это то что мы искали — число 23! Его индекс = 5!

Давай рассмотрим результаты работы нашего алгоритма, а потом разберемся с его сложностью. (Кстати, теперь ты понимаешь, почему его называют двоичным: его суть заключается в постоянном делении данных на 2). Результат впечатляет! Если бы мы искали нужное число линейным поиском, нам понадобилось бы 10 проверок, а с двоичным поиском мы уложились в 3! В худшем случае их было бы 4, если бы на последнем шаге нужным нам числом оказалось второе, а не первое. А что с его сложностью? Это очень интересный момент 😊

Алгоритм двоичного поиска гораздо меньше зависит от числа элементов в массиве, чем алгоритм линейного поиска (то есть, простого перебора). При **10** элементах в массиве линейному поиску понадобится максимум 10 проверок, а двоичному — максимум 4 проверки. Разница в 2,5 раза. Но для массива в **1000 элементов** линейному поиску понадобится 1000 проверок, а двоичному — **всего 10**! Разница уже в 100 раз! Обрати внимание: число элементов в массиве увеличилось в 100 раз (с 10 до 1000), а количество необходимых проверок для двоичного поиска увеличилось всего в 2,5 раза — с 4 до 10. Если мы дойдем до **10000 элементов**, разница будет еще более впечатляющей: 10000 проверок для линейного поиска, и **всего 14 проверок** для двоичного. И снова: число элементов увеличилось в 1000 раз (с 10 до 10000), а число проверок увеличилось всего в 3,5 раза (с 4 до 14). **Сложность алгоритма двоичного поиска логарифмическая**, или,если использовать обозначения Big-O, — **O(log n)**.

Array Size	Linear — N	Binary — Log_2N
10	10	4
50	50	6
100	100	7
500	500	9
1000	1000	10
2000	2000	11
3000	3000	12
4000	4000	12
5000	5000	13
6000	6000	13
7000	7000	13
8000	8000	13
9000	9000	14
10000	10000	14

Рисунок 5. Количество проверок в зависимости от размера массива при помощи линейного поиска и алгоритма двоичного поиска

V. Оценки сложности

1. *O(1)* – постоянное время
2. *O(log(n))* – логарифмическое время
3. *O(n)* – линейное время
4. *O(n log(n))* – «n-log-n» время
5. *O(n^2)* – квадратичное время
6. *O(n^3)* – кубическое время
7. *O(c^n)* – экспоненциальное время
8. *O(n!)* – факториальное время

A. *Сложение двух чисел*
B. *Двоичный поиск*
C. *Просмотр элементов массива*
D. *Некоторые сортировки*
E. Просмотр пар элементов (сортировка вставками)
F. Динамическое программирование
G. Перечисление всех подмножеств множества
H. Все перестановки n элементов

Перечисленные термины и обозначения являются оценками сложности алгоритмов и операций с использованием большой "O" нотации. Они позволяют оценить, как быстро растет время выполнения алгоритма при увеличении размера входных данных.

Каждая из этих оценок сложности помогает понять, как алгоритм будет вести себя при работе с разными объемами данных и как он будет масштабироваться. Более низкая сложность обычно означает более эффективный алгоритм, который будет более быстро работать на больших данных.

Пример. Что лучше $\boldsymbol{n^2}$ или $\boldsymbol{nlog(n)}$?

С точки зрения теории (асимптотика):

С точки зрения асимптотической сложности (теории), $n\ log(n)$ является более эффективной сложностью по сравнению с n^2. Это означает, что при увеличении размера входных данных n, функция $n\ log(n)$ будет расти медленнее, чем функция n^2. В данном случае, асимптотическая сложность $n\ log(n)$ означает, что алгоритм будет более эффективным при работе с большими объемами данных.

С точки зрения практики:

Однако, на практике, константы и скрытые факторы также играют важную роль. Для небольших размеров данных или в контексте конкретных реализаций, алгоритмы с более высокой асимптотической сложностью (например, n^2) могут оказаться быстрее из-за меньшей скрытой константы. Это может быть вызвано оптимизациями, структурами данных, кэшированием и другими факторами, которые влияют на реальное время выполнения.

Итак, в практических случаях, особенно при малых размерах данных или в контексте конкретных задач, n^2алгоритм может показать лучшую производительность из-за более низких констант, несмотря на более низкую асимптотическую сложность n log(n).

В итоге, выбор между n^2 и $n\ log(n)$ зависит от конкретных условий задачи, объема данных и реальных реализаций алгоритмов.

Пример. Что лучше n^{100} или 2^n?

С точки зрения асимптотической сложности (теории), 2^n считается гораздо более сложной функцией роста по сравнению с n^{100}. Это означает, что по мере роста n, функция 2^n растет экспоненциально быстрее, чем функция n^{100}. Асимптотическая сложность 2^n значительно выше и, в теоретическом плане, стоит ожидать, что она будет намного менее эффективной.

Однако на практике, как вы уже отметили, на выбор алгоритма также влияют константы и скрытые факторы. При малых значениях n, константы перед n^{100} могут быть такими большими, что функция 2^n становится более эффективной в плане времени выполнения. Это связано с тем, что возведение в степень может очень быстро увеличивать значение, и даже когда степень невелика, большие константы могут доминировать.

Однако с ростом n, функция 2^nначинает расти очень быстро, и даже небольшое увеличение n может сделать эту функцию практически невыполнимой. Напротив, n^{100} растет гораздо более умеренно, и с ростом n он всё равно будет оставаться более эффективным по времени выполнения.

Итак, в практическом контексте, при достаточно больших значениях n, n^{100} будет иметь лучшую производительность. Однако, как и в предыдущем вопросе, конкретный выбор может зависеть от контекста, размера данных и других факторов.

размер / сложность	10	20	30	40	50	60
n	*0,00001 сек*	*0,00002 сек*	*0,00003 сек*	*0,00004 сек*	*0,00005 сек*	*0,00005 сек*
n^2	*0,0001 сек*	*0,0004 сек*	*0,0009 сек*	*0,0016 сек*	*0,0025 сек*	*0,0036 сек*
n^3	*0,001 сек*	*0,008 сек*	*0,027 сек*	*0,064 сек*	*0,125 сек*	*0,216 сек*
n^5	*0,1 сек*	*3,2 сек*	*24,3 сек*	*1,7 минут*	*5,2 минут*	*13 минут*
2^n	*0,0001 сек*	*1 сек*	*17,9 минут*	*12,7 дней*	*35,7 веков*	*366 веков*
3^n	*0,059 сек*	*58 минут*	*6,5 лет*	*3855 веков*	2×10^8 веков	$1{,}3 \times 10^{13}$ веков

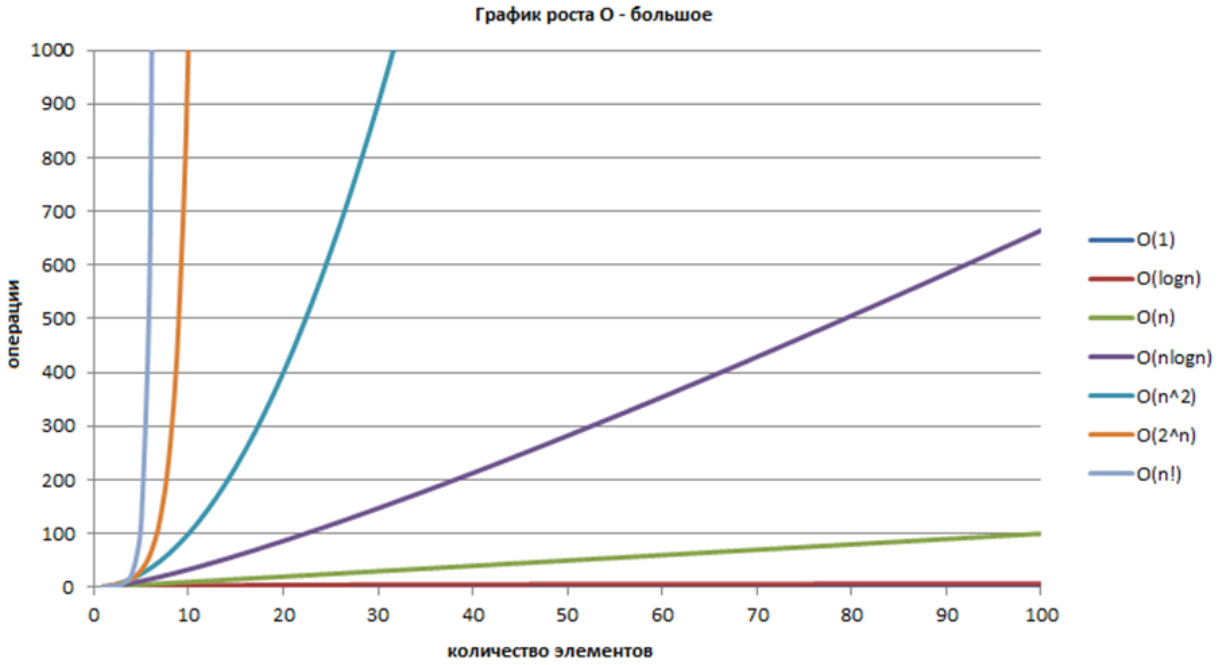

Рисунок 8. Зависимость количества элементов от количества операций при разных сложностях функций.

VI. Худший, средний и лучший случаи.

Когда говорят о худшем, среднем и лучшем случаях в контексте временной сложности алгоритмов с использованием большой "О" нотации, это относится к различным сценариям выполнения алгоритма в зависимости от входных данных. Вот объяснение каждого случая:

Лучший случай (Best Case): Это сценарий, в котором алгоритм выполняется с наименьшим возможным количеством операций. В лучшем случае алгоритм обычно достигает наибольшей эффективности. Однако лучший случай не всегда реалистичен и может встречаться редко.

Средний случай (Average Case): Это сценарий, в котором алгоритм выполняется в среднем при рассмотрении всех возможных входных данных. Оценка среднего случая может быть сложной, так как она требует анализа вероятности входных данных и времени выполнения для каждой из них.

Худший случай (Worst Case): Это сценарий, в котором алгоритм выполняется с наибольшим количеством операций. Оценка худшего случая важна, так как она дает верхнюю границу для временной сложности алгоритма, гарантируя, что выполнение алгоритма никогда не будет хуже определенного значения.

По сути, большинство анализов алгоритмов фокусируются на худшем случае, так как это позволяет определить максимальное время выполнения алгоритма при любых входных данных. Однако для конкретных задач и алгоритмов может быть важным также рассмотреть средний и лучший случаи для более полного понимания их производительности и поведения.

Пример. Последовательный поиск.

```
def linear_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
```

Лучший случай: Лучший случай возникает, когда искомый элемент находится в начале массива. В этом случае поиск завершится на первой итерации, и функция вернет индекс 0.

Средний случай: Предположим, что искомый элемент не имеет какого-то определенного расположения, и он может быть в любом месте массива. В среднем случае, если массив имеет n элементов, вероятность того, что искомый элемент будет находиться в какой-либо позиции, одинакова для всех позиций. Поэтому, средний случай будет в среднем требовать выполнения примерно $\frac{n}{2}$ итераций.

Худший случай: Худший случай возникает, когда искомый элемент отсутствует в массиве или находится в последней позиции массива. В этом случае поиск будет проходить через все элементы массива, и количество итераций будет равно n. То что мы искали в прошлых заданиях это и есть как раз таки худший случай Big-O.

Итак, в оценке временной сложности для этой функции:

Лучший случай: $O(1)$
Средний случай: $O(n/2)$, но обычно округляется до $O(n)$
Худший случай: $O(n)$
Обратите внимание, что большая "O" нотация учитывает основную тенденцию роста, поэтому линейный поиск оценивается как $O(n)$, хотя лучший случай имеет константное время.

Алгоритм	**Структура данных**	**Временная сложность**			**Вспомогательные данные**
		Лучшее	**В среднем**	**В худшем**	**В худшем**
Быстрая сортировка	Массив	$O(nlog(n))$	$O(nlog(n))$	$O(n^2)$	$O(n)$
Сортировка слиянием	Массив	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	$O(n)$
Пирамидальная сортировка	Массив	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	$O(1)$
Пузырьковая сортировка	Массив	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка вставками	Массив	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка выбором	Массив	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Блочная сортировка	Массив	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(nk)$
Поразрядная сортировка	Массив	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$

VII. Полиномиально эквивалентные функции

Функции $f(x)$ и $g(x)$ называются *полиномиально эквивалентными*, если существуют такие полиномы $p(x)$ и $p'(x)$, что (начиная с некоторого числа)

$$f(x) \leq p(g(x))$$
$$g(x) \leq p'(f(x)).$$

Если функции $f(x)$ и $g(x)$ принадлежат разным классам, то функция с более быстрым темпом роста доминирует над менее быстро растущей функцией

VIII. Классы сложности алгоритмов

Классы P и NP сложности являются важными понятиями в теории вычислительной сложности. Они определяют группы алгоритмов, которые могут или не могут быть решены за разумное время на различных типах вычислительных устройств.

P (Polynomial Time): Класс P включает в себя алгоритмы, которые могут быть решены за полиномиальное время от размера входных данных. В других словах, алгоритм считается полиномиально разрешимым, если время его выполнения ограничено многочленом от размера входных данных. Эти алгоритмы считаются эффективными и практически применимыми.
Примеры алгоритмов класса P: линейный поиск, сортировка слиянием, быстрая сортировка.

NP (Nondeterministic Polynomial Time): Класс NP включает в себя алгоритмы, для которых можно быстро проверить правильность решения. В данном случае, решение может быть проверено за полиномиальное время, но само решение может потребовать экспоненциальное время для поиска. Однако, пока не доказано, существует ли эффективный алгоритм для нахождения решения NP-проблемы.

Примеры алгоритмов класса NP: задача о рюкзаке, задача о коммивояжере.

NP-полные (NP-Complete) проблемы: Это подкласс NP, который содержит самые трудные NP-проблемы. Если бы удавалось найти полиномиальный алгоритм для одной из NP-полных задач, это бы означало, что можно быстро решить любую NP-проблему, приведя ее к данной задаче. Однако, пока не найдено полиномиального алгоритма для ни одной NP-полной задачи.

Примеры NP-полных задач: задача о рюкзаке, задача о выполнимости булевых формул (SAT).

Существует гипотеза, известная как гипотеза P vs NP, которая предполагает, что P = NP, то есть, что каждая NP-проблема также имеет полиномиальный алгоритм решения. Однако, эта гипотеза остается нерешенной, и ее доказательство или опровержение является одной из открытых задач в компьютерной науке.