Алгоритмы и структуры данных

Лекция 2. Асимптотический анализ алгоритмов

# Введение

Существует 3 причины, по которым следует изучать О-большое.

**Причина 1**. Концепцию О-большое необходимо понимать, чтобы уметь видеть и исправлять не оптимальный код. Зная концепцию О-большое, вы сможете увидеть код, который является неоптимальным, а также вы сможете оптимизировать его.

**Причина 2.** Ни один серьёзный проект, как они одно серьёзное собеседование не могут обойтись без вопросов О-большое. Допустим, вы проходите собеседование. Вы идеально написали код, полностью решив задачу. Вы уверены, что код работает правильно и максимально быстро? Проблема кроется в следующем. После того, как вы написали код, интервьювер, попросит вас, пожалуйста, оцените ваш алгоритм в рамках О-большого. В случае, если вы допустите хотя бы одну маленькую ошибку, вы не пройдёте собеседование. Не важно, как быстро работает ваш алгоритм, потому как если вы не умеете оценивать его в рамках О-большого, это значит одно: Вы не понимаете, что вы написали.

То же самое касается серьезных крупных компаний. На каждом проекте такой компании существует процедура кодревью. Это проверка вашего кода более опытным коллегой. Допустим, ваш коллега нашел не оптимальное место в вашем алгоритме. Он попросит его исправить. Он это сделает 1 – 2 раза, может быть 3 раза. На 4 раз у него возникнут вопросы, а на 5 вас могут просто уволить.

**Причина 3.** Непонимание О-большого ведет к серьёзной потере производительности в ваших алгоритмов.

С причинами изучения разобрались.

**Цель лекции** – это научиться применять основой концепции О-большого.

Прежде чем двигаться дальше, необходимо привести пример из реального мира, который касается О-большого. Дело в том, что человеку порой невозможно в голове представить математическую модель О-большого. О-большое является именно математической моделью, и для этого нужен пример из реального мира.

Представьте ситуацию, допустим, вы живёте в Минске и вам нужно передать файл другу, который живет за 10000 км от вас. Допустим, в Пекине, как это лучше сделать?

Первый вопрос, который вы должны задать это чему равен размер файла? Ясное дело, что если мы хотим передать картинку, которая весит 100 килобайт, то интернет является лучшим способом передачи информации. Но этот вариант хороших для маленьких файлов. Допустим, Google решила перекачать все свои данные в Минск. Это уже не терабайт данных, это гораздо больше. Ясное дело, что интернет в этом случае не подходит. И вот здесь возникает вопрос, как это лучше сделать?

На самом деле. Все данные Google легко можно перевести на самолете, медленнее он от этого лететь не станет. Запомните этот пример, мы к нему ещё вернёмся.

Теперь рассмотрим с вами оценку работы программы с точки зрения времени.

В прошлом веке на заре программирования возникла проблема: как решить, какой алгоритм быстрее и эффективнее работает? Дональд Кнут был первым, кто предоставил решение этой проблемы. Допустим, у нас есть алгоритм. Он сортирует массив из 10000 элементов. Согласитесь, что если выполнить этот алгоритм на современном компьютере, то он отработает за 0,000001 секунду. Если же взять старый процессор, например, пентиум 1997 года выпуска, то на нём алгоритм отработает, скажем, за 2 секунды. И вот здесь кроется проблема. Мы не можем оценивать скорость работы алгоритма с точки зрения времени, потому как один и тот же алгоритм на разных машинах выполняется за разное время. Выход из сложившегося положения был очень простым. Любой алгоритм включает себя какое-то число шагов, или говоря более точно, количество тактов, которая выполняет процессор, прежде чем алгоритм закончится. Таким образом не важно на старой и на новой машине выполняется алгоритм, так как количество шагов на обеих машинах будет одинаковым. Таким образом, идея О-большого - это показать, какое количество шагов необходимо сделать, чтобы алгоритм закончил свое выполнение?

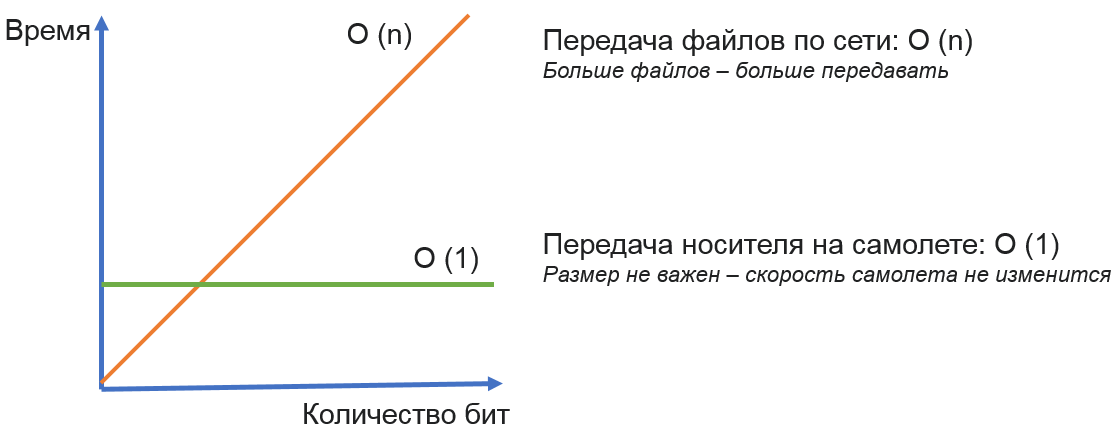
Если мы посмотрим определение О-большое в Wikipedia, то мы увидим следующее, О-большое - это математическое значение для сравнения асимптотического поведения функций. К сожалению, это определение фактически ничего не даёт нам с точки зрения понимания, как мы можем оптимизировать или оценивать алгоритмы, но мы можем переписать данные определение, используя суть, которую в него вкладывал автор.

**О-большое показывает верхнюю границу зависимости между входными параметрами функций и количество операций, которая выполнит процессор.**

Поэтому определении появилось новое словосочетание верхняя граница.

Главное, что мы здесь должны понять эта фраза *зависимость между входными параметрами и количеством операций*. Эта фраза означает, что О-большое показывает зависимость между допустим массивом на тысячу элементов, которые подаются на вход функций и количеством тактов процессора, которая необходимо выполнить, чтобы алгоритм обработал эти тысячу элементов.

Вернемся к нашему примеру самолетом.



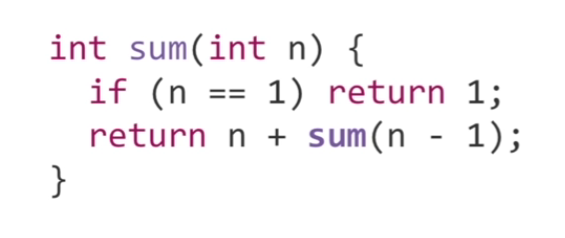
Если представить, что ось х - это количество бит, а ось у - это количество времени, которое необходимо на передачу данных, что мы можем сделать. Вывод, что для передачи файлов по сети мы видим, что чем больше бит, тем дольше их передавать, то есть зависимость прямо пропорционально или просто n при передаче данных на самолёте нам не важен размер файла, скорость самолета от этого не изменится или просто мы можем написать это как о от одного?

Здесь может возникнуть вопрос, что означают буквы **О** и **N**? Это принятое обозначение концепции О-большого. В скобках указывается формула, которая описывает поведение алгоритма для первого случая. Сложность является прямо пропорционально и зависит от количества передаваемых бит. Больше бит - дольше передавать. Зависимость является линейной в рамках О-большого. Зависимость принято описывать используя букву.

Если же у нас есть другие переменные, то это будет **М**, **Л**, **К** и другие буквы. Второй случай от единицы представляет собой постоянную величину? Может возникнуть вопрос, почему от одного? Ведь нет зависимости между размером файла и скоростью самолета.

То же самое касается алгоритмов, если мы не исполняем никакой код, значит ничего не происходит, а значит, сложность равно О(0). Если же хотя бы мы вызвали функцию, которая ничего не выполняет, то мы уже совершили одно действие вызов функции, следовательно, сложности рано О(1).

На самом деле подскобками может стоять совершенно любое выражение. Наша задача на сегодня, глядя на код, научиться выводить формулы, описывающие сложность алгоритма. Давайте рассмотрим пример, у нас есть функция.

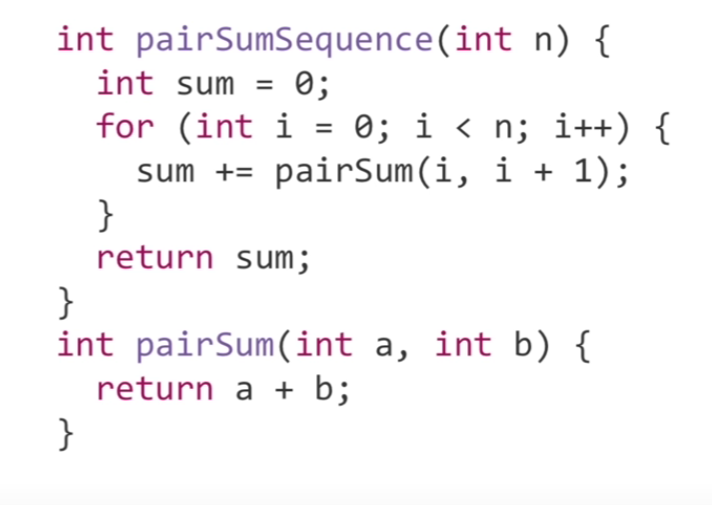


Она является рекурсивной и считает сумму чисел. Если мы вызовем её с N = 3, то ответ будет 1 + 2 + 3 = 6. Если мы передадим N = 4, то ответ будет 1 + 2 + 3 + 4 = 10 и так далее. Возникает вопрос, какая зависимость между значением входного параметра N и количеством раз, сколько функция вызовет сама себя?

Если N = 3, то функция вызывает себя 3 раза, потому что мы начинаем выполнять функцию от трёх и каждый раз отнимаем единицу, пока не дойдём до единицы. То есть мы вызываем функцию от 3, вызываем функцию от 2 и затем от одного. Итого функция вызвала сама себя 3 раза. Если N = 10, то функция выполняется 10 раз, если N = 100, функция вызывает сама себя 100 раз.

Другими словами, быстродействия этой функции прямо пропорционально входному значению параметра N, чем больше N, тем больше раз функция вызовет сама себя.

Посмотрим на вторую функцию.



Она складывает какие-то пары чисел нам не суть важно, что она делает. Функция выполняет сложения, А + В. В ней нет ни циклов, не рекурсии. Она всегда выполняет константное количество операции, следовательно, сложность функции равна О(1).

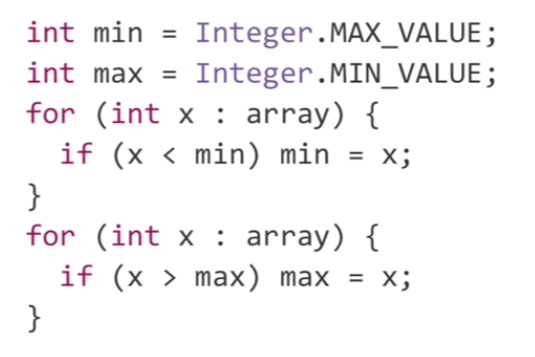
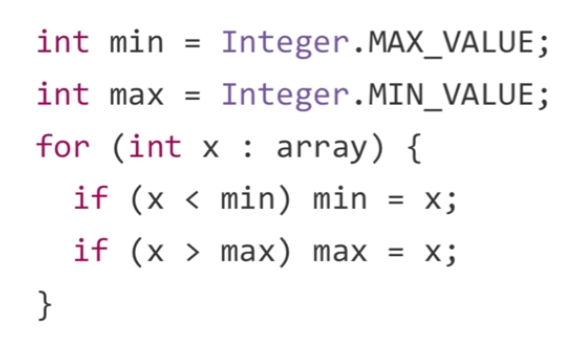
Главная функция pairSumSequence в цикле складывает какие-то пары чисел, вызывая функцию pairSum. Цикл выполняется от 0 до N. Другими словами, чем больше N, тем больше раз выполняется цикл. Если N = 3, то цикл выполнится от 0 до 2. Если N = 10, то цикл выполнится 10 раз, если N = 100, то цикл выполнится 100 раз. Другими словами, данная сложность выполнения данной функции тоже О(N).

Этим примером мы получили первое впечатление о том, как оценивать скорость выполнения функции. Прежде чем двигаться дальше, мы должны понять некоторые математические особенности оценки сложности выполнения функций в рамках О-большого.

На самом деле фраза верхняя граница и отбрасывание констант это одно и то же. Что это значит? При оценке скорости выполнения алгоритма мы не говорим о каких-то конечных входных значениях. Допустим, за сколько выполняется алгоритм, если входной массив имеет 3 элемента? Такая оценка не имеет смысла, поскольку ответ будет мгновенно 3. Это слишком мало, чтобы оценить алгоритм. Тогда возникает вопрос, какое число это много? 10000 элементов? На самом деле мы говорим о том, как будет себя вести алгоритму на бесконечности.

Таким образом, О-большое описывает только скорость роста. Поэтому, мы отбрасываем константы при оценке сложности, и поэтому алгоритм, описываемый как О(2N), должен описываться как O(N).

Разберем пример, у нас есть 2 функции, левая и правая. Обе функции находит минимум и максимум массиве.



Какая из этих функций работает быстрее? Логично ответить, что левая, потому что там одна инициализация цикла и один проход по циклу. И на самом деле, если посмотреть команды процессора, то правый пример медленнее, но это неверный подход к решению. О-большое показывает, как ведут себя эти алгоритмы, и для обоих случаев это О(N).

Следующее, о чем необходимо поговорить, чтобы понять О-большое это неважная сложность. Допустим, у нас есть выражение О(N2 + N). N не является константой. Как с этим быть на самом деле? Мы знаем, что О(N2 + N) = О(N2), так как мы бросаем константы. И мы точно знаем, что N2 значительно больше N. Здесь стоит уточнить, что значит слово значительно с точки зрения О-большого, значительно – это значит в 2 раза. И по сути, раз N значительно меньше, чем N2, от она не влияет на темп роста функции, то есть её сложность является неважной, и ей можно пренебречь. Следовательно, О(N2 + N) = О(N2).

Давайте разберем несколько примеров для лучшего понимания отбрасывания неважной сложности.

О(N + log N) = О(N) – это просто, потому что log намного меньше, чем N.

О(5 \* 2N + 10 \* N100) = О(2N).

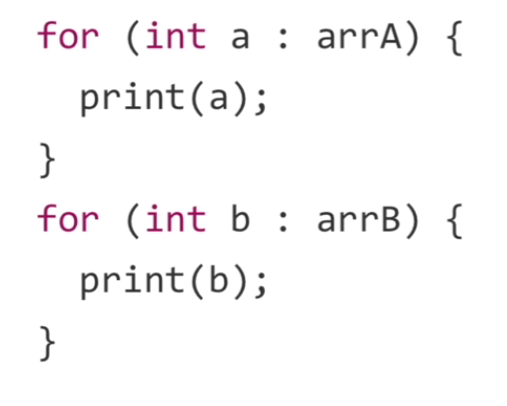
Вот здесь прежде, чем решать такие сложные выражения, необходимо подумать, что можно выбросить сразу. Пятёрка и десятка являются константами, поэтому от них можно избавиться. Остаются 2N и N100. На самом деле степенная функция растёт гораздо быстрее, чем на N степени 100. Таким образом, ответ будет О(2N).

И последний пример,

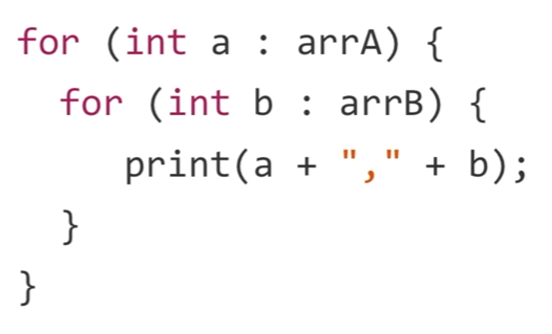
О(N2 + B) = О(N2 + B).

Так как мы ничего не знаем о В, то В может быть равным N, может быть просто числом 3, а может быть какой-нибудь другой буквой? А пока мы ничего не знаем о нашей переменной, мы не можем выбросить её из нашей формулы.

На предыдущих формулах мы видели знаки сложения и умножения. Давайте попробуем научиться их расставлять, правильно. На самом деле практически у всех новичков возникает главный вопрос. Почему-то они не могут понять, когда нужно складывать, а когда нужно умножать ответ очень простой? Давайте рассмотрим пример.



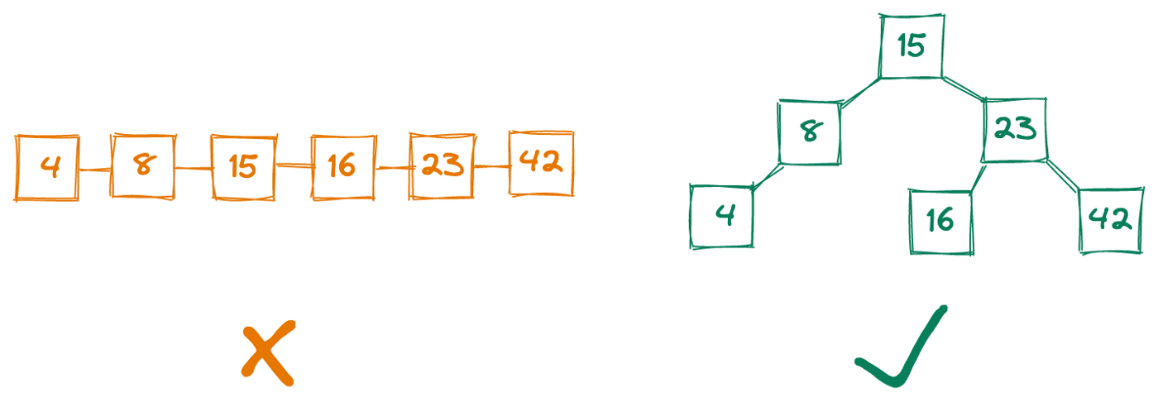
У нас есть код, он проходит массив длиной А, затем проходит массив длиной В. Сложность выполнения данного кода будет равна О(А + В). То есть выполнение первого цикла совершенно не зависит от выполнения второго цикла.



Есть другой код. Сначала мы проходим массив длиной А, а внутри этого цикла проходим массивные В. И вот здесь мы уже будем выполнять умножение О(А \* В), потому что функция print выполняется В раз и из этих В раз она выполнится ещё А раз. Другими словами, цикл по В зависит от цикла по А.

Если вы откроете в Википедии любой алгоритм сортировки или поиска, вы наверняка увидите, что оценка в сложности данного алгоритма содержит выражение O(logN). На самом деле вопрос появления данного логарифма в формулах сводит с ума многих людей, и люди не понимают, откуда он берётся.

Прежде чем начать разбирать, откуда берется O(logN), давайте вспомним примерный алгоритм бинарного поиска в отсортированном массиве.



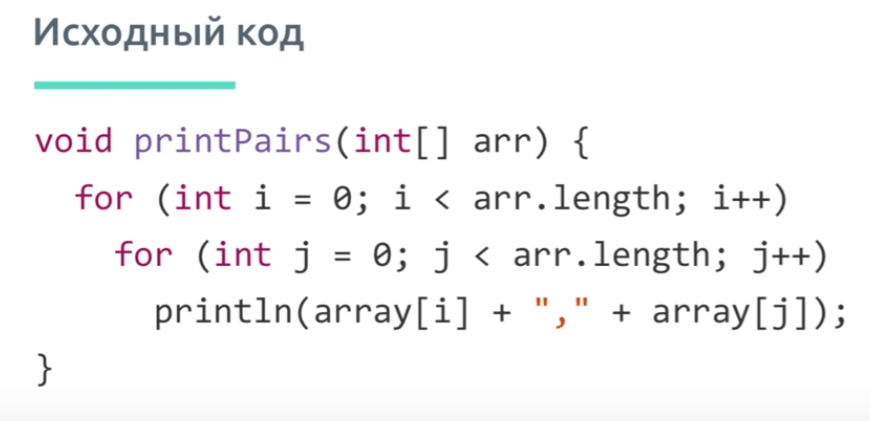
Шаг 1

Мы берём середину массива. Если искомое число меньше середины массива, идем в левый под массив больше вправо. Повторяем то же самое для подмассива. Алгоритм заканчивается, если мы нашли искомое значение. Допустим, у нас есть массив из 16 элементов, то есть N=16.

Если мы хотим найти число, то мы должны сначала искать в 16 и элементах, потом 8, потом в 4, потом в 2, потом в одном. Если этот один оставшийся элемент неравен искомому, то элементы не существует. Давайте посмотрим на цифры, они отображают, как массив уменьшается в своем размере при поиске элемента.

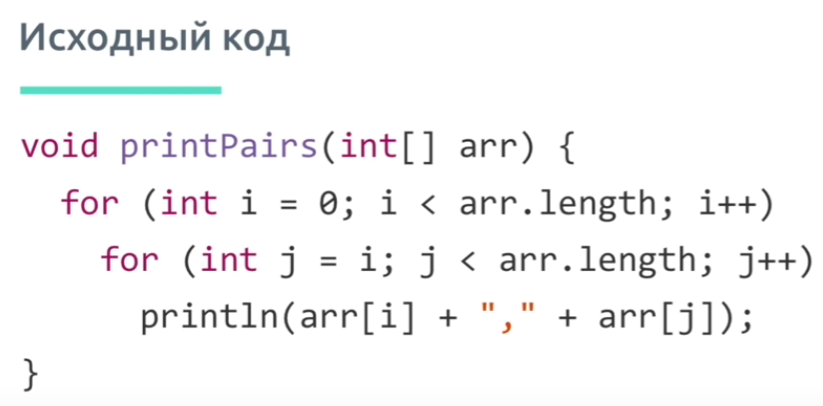
Вывод. Для алгоритма, где **на каждой итерации берется половина элементов** - сложность будет **включать** O(log N).

Пример 2.



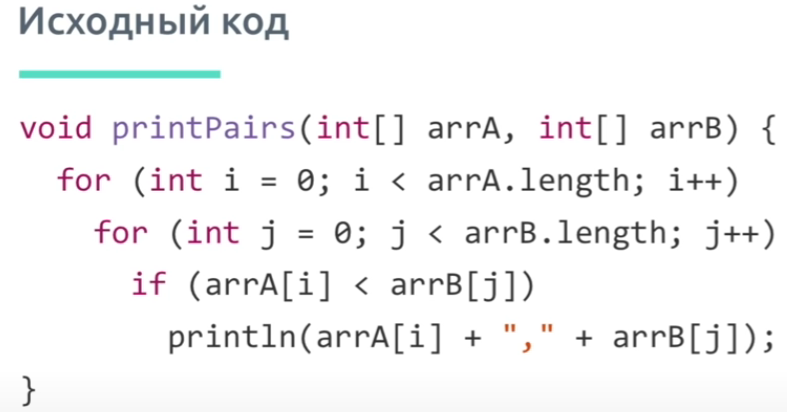
У нас во вложенных цикла, каждый из которых работает от 0 до N. Таким образом, сложность будет O(N \* N) = O(N2).

Пример 3.

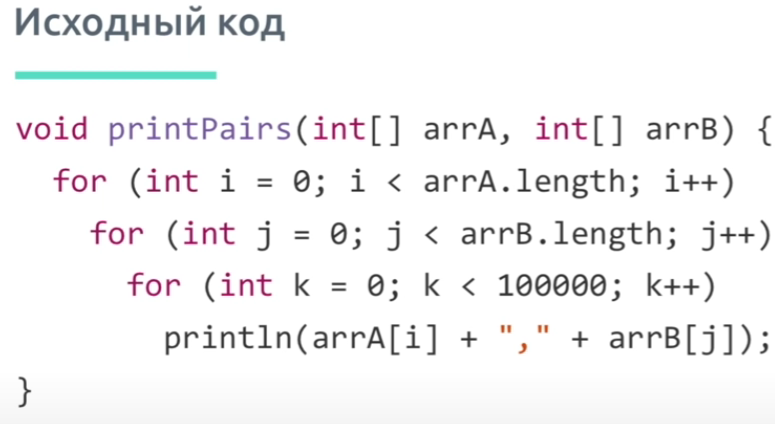


Вот здесь у нас первый цикл работает от 0 до N. А второй от i до N раз. Сначала N, потом N-1, N-2, …, 2, 1. 21. Каждый раз N уменьшается на 1, потому что каждый раз i увеличивается на 1. И таким образом, мы можем сказать, что на самом то деле сложность будет равна О(N + (N-1) + (N-2) + … + 2 + 1) = О(N2).

Пример 4.

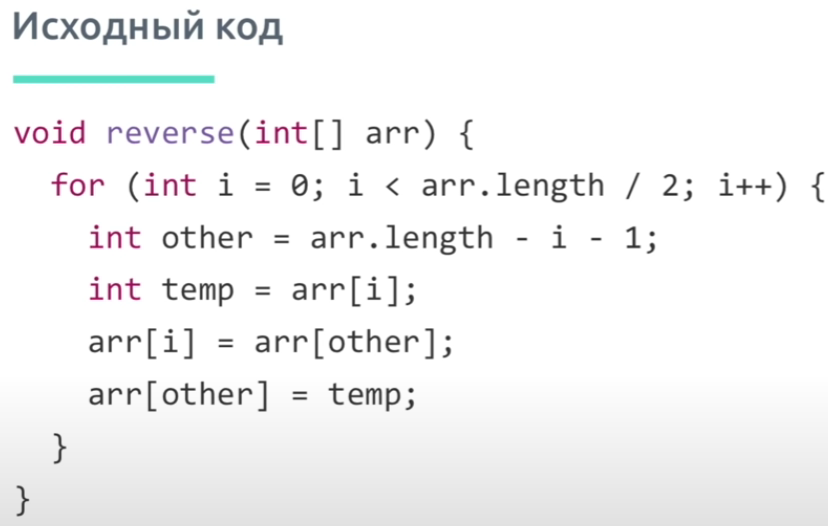


У нас есть коl, который выполняется сначала для массива длиной A, а затем для массива длиной B. Получается это 2 вложенных цикла по А и В. Итого, сложность будет равна О(А \* В).



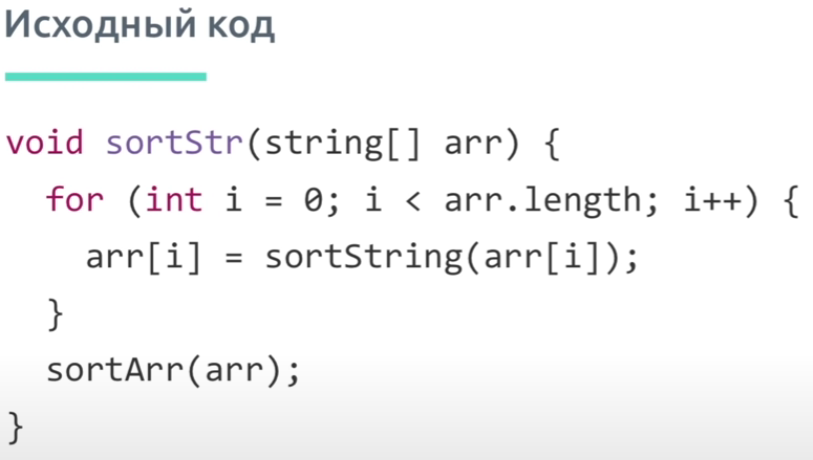
Код является точной копией предыдущего. Кроме того, еще внутри выполняется цикл 100000 раз. Конечно, 100000 это большое число. На самом деле это константа, а значит, сложность выполнения будет точно такой же, как в прошлом коде.

Пример 5.



Вот здесь мы проходим массив ровно до половины. У некоторых людей может возникнуть сомнения или вопрос. Мы берём половину элементов на операции. По идее должен быть log N. На самом деле это заблуждение, здесь мы просто идем до половины массива, а это значит, что сложность равно 1/2 или просто O(N).

Пример 6.



У нас есть функция, которая сортирует сначала строки, а потом сортирует массив строк. Вот здесь очень интересный момент. Цикл работает от 0 до N, то есть N раз далее происходит сортировка строк. Длина каждой строки будем считать равна l сортировка самая обычная l умножить на log l. А вот сортировка массива, казалось бы, тоже обычная N\*log N, но здесь мы должны учитывать ещё буквочку l, что это значит? Компьютер не может сравнивать строки за один такт процессора, он может сравнивать числа, допустим, 3 > 2, это один такт - это просто число, но компьютер не может сравнить две строки, такие как «labs» за один такт. Поэтому для сравнения этих строк ему нужно сравнить сначала а и а, б и б и т.д., Только здесь он может дать правильный ответ равны строки или нет, то есть получается, при сравнении 2 строк процессор в любом случае выполняет l количество тактов сравнения.

Количеству меньше строки, то есть он должен пройти всю строку длиной l, и поэтому при сортировке массива мы должны учитывать не только саму сложность алгоритма, но и саму сложность сравнения строк. И вот отсюда появляется буквочка l. Таким образом, это доказывает, что не зная О-большого, увы, можете в l раз просесть по производительности в вашем коде.

Если подводить итог, то сложность выполнения данного алгоритма будет равна:

О(N \* L \* log L + L \* N \* log N) = O(N \* L \* (log L + log N).

Далее мы должны сделать выводы.

Вот О-большое показывает темп роста функции, следовательно, мы не участвуем константы и неважную сложность. Определены последовательность действий сложения, умножения. Для алгоритма, где **на каждой итерации берется половина элементов** - сложность будет **включать** O(log N).