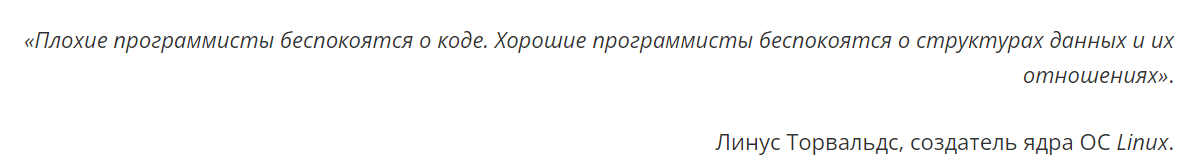
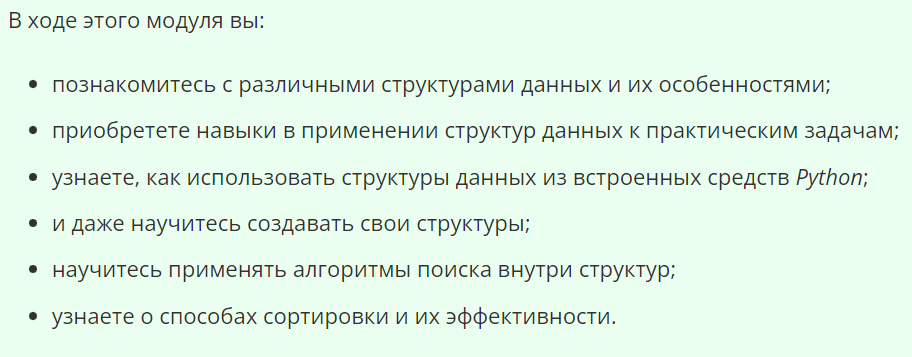
**Модуль B9. Алгоритмы и структуры данных**



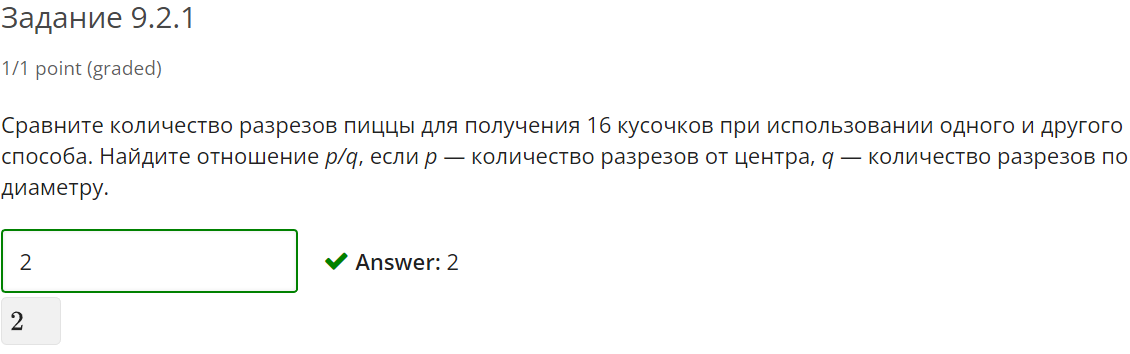
Да, да. Может быть когда-нибудь. Чую этот модуль будет ну\*(д/ж)ным.



Но к этому лучше в другой день. И снова этот день настал.

Пример с пиццей, можно разрезать ее от центра. И тогда количество разрезов будет эквивалентно количеству кусков пиццы: 0 разрезов - 1 целая, 1 разрез – 1 надрезанная, 2 разреза, 1 кусок и 1 остаток пиццы.

А можно резать ее по диаметру до конца и сразу получить четное количество кусков. Я реально это пишу? Да.. этого даже задача касалась.



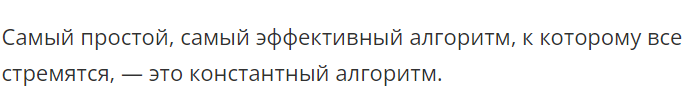


Сложность алгоритмов… да, что-то такое было.

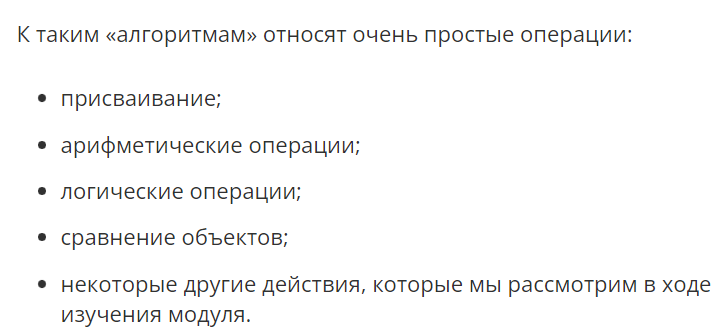
Существует **временная** и **пространственная** сложность.

**Временная сложность** связана напрямую с тем, сколько алгоритм будет выполняться. Может зависеть от количество обрабатываемых данных.

**Пространственную сложность** затрагиваютреже. Она касается напрямую затрагиваемой памяти. Чем больше переменных, мб лишних действий и тд будет в алгоритме, тем больше памяти он потребует.



Впервые слышу. **Константный** или **постоянный по времени** это алгоритм который выполняет необходимое действие за всегда одинаковое количество времени.



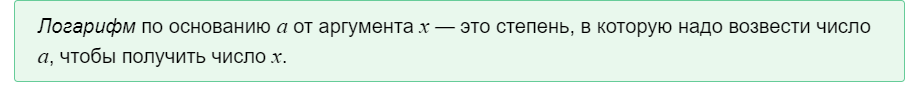
Более сложные, например найти наибольший элемент в списке уже сложнее. Будет зависеть от количества данных, 3 штуки или 3 миллиона? Самым простым вариантом будет пройтись по всем элементам списка. Время вычисления будет расти прямо-пропорционально данным. Такой алгоритм будет считаться **линейным**.

Бывают ситуации, когда алгоритмы могут работать быстрее чем **линейные**. Самые распространенные – это **логарифмические** **алгоритмы**. Блин, уже не помню почти ничего о логарифмах. Придется вспомнить.

Их название напрямую связано с тем, как возрастает временная сложность алгоритма с ростом размера входных данных.

Принято считать основанием логарифма двойку если не указано иное.

Вспоминаем логарифмы!!

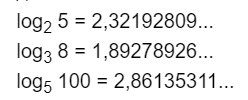






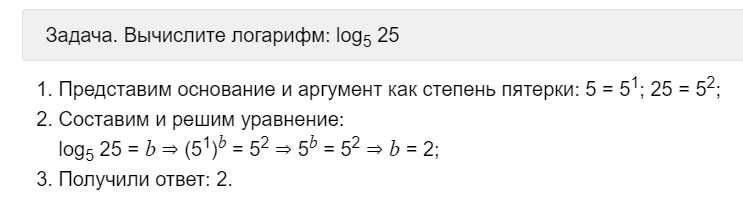
Угу, ага. А зачем?

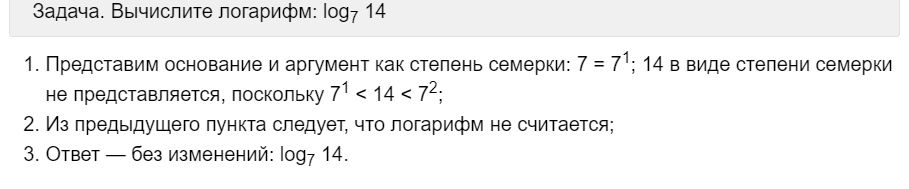
Но это простые логарифмы. А вот те, которые не получаются точно.



**Эти цифры напомню называются иррациональными.** Те, у которых можно писать цифры после запятой и они никогда не повторяться. Ясно. Значит делители числа 7 другие. Они повторяются.

Аргумент и основание. Всегда должны быть больше 0.





Ну да, рационального ответа нет, но что насчет 7^1.356 = Это почти 14.

Также существует десятичные логорифмы. Это логорифмы по основанию 10. Пишутся просто lg.

Мб потому что **g** это отсылка к числу **g** скорости свободного падения или уже не помню на нашей планете, которая почти равна 10.

Кстати. Как вычислять логорифм на калькуляторе. Да, я не школьник мне уже можно.

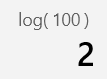


Надо отдельно логарифм верхнего числа, поделить на логарифм нижнего числа.



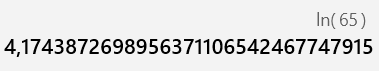
А теперь объясните мне, а как самому получить логарифм 8? Я имею ввиду какое основание используется 10?

Похоже, что да?

 10^x == 100 == 10^2. x = 2

Ну и на последок проверю ln. Насколько я знаю это логарифм числа по основанию 2.7

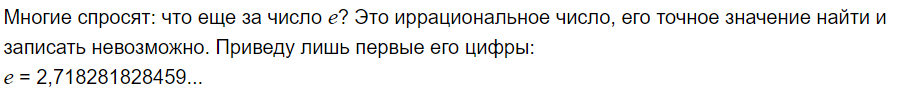
ln 65 = 4.20274?



LN это не 2.7 может 2.07. Ща…

**2.7 это e**





Как и число Пи. А это случайно не цифры числа 7? Не… У 7 такие: **142857**

**Так…**

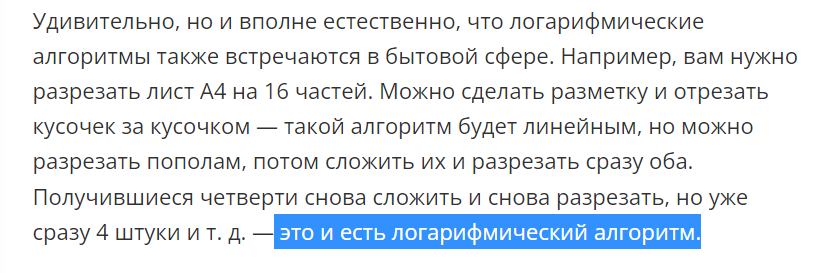
**Но это все же ln был.**

**Понятно. Я ошибся потому что использовал только часть основания 2.7, а калькулятор использовал более точное 2.718… и тд. Для моего числа я подобрал максимально близкое.**

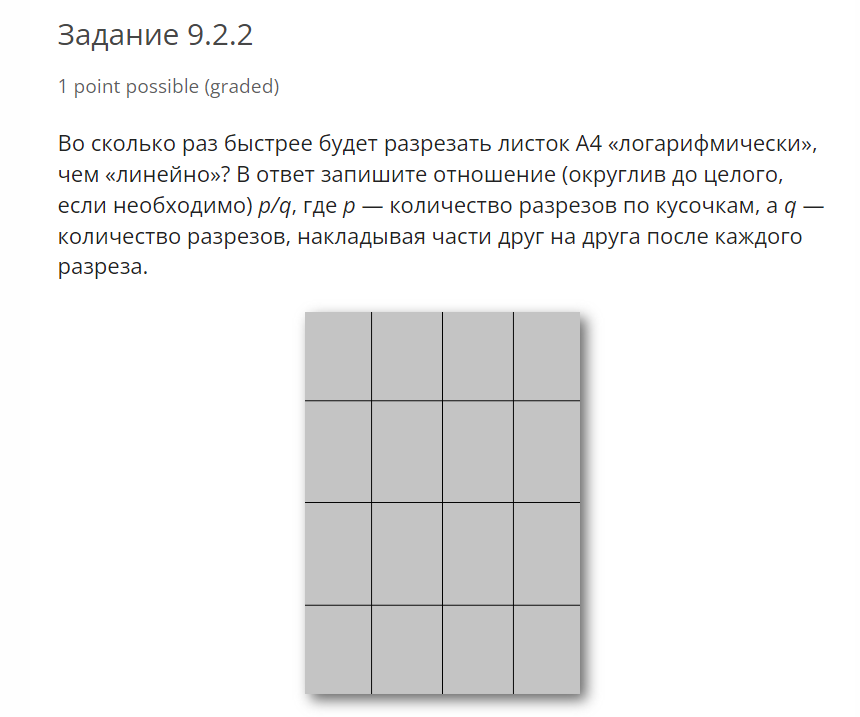
Если внутри ln используется число e. То ответом логарифма будет степень числа e. Потому что в основании и так e.



Ладно. Вернемся к программированию.



**Это.. довольно понятный пример.**





Линейно 24 разреза. А логарифмически 4 разреза. Ответ 24/4 = 6

Чтобы раставить 10 книг на полке в алфавитном порядке нужно провести такое количество сравнений:



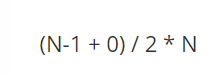
45. Если бы в этой арифметической прогрессии было бы миллион чисел, было бы сложнее выяснить результат. Для этого есть формула:

**найти среднее арифметическое первого и последнего членов прогрессии и умножить на количество всех членов.**

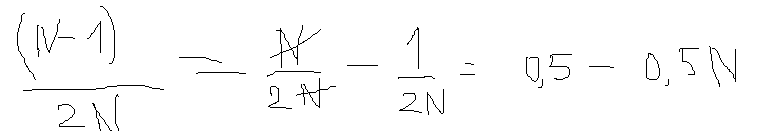
Попробую для такой же прогрессии но из миллиона чисел.

(999999 + 0) / 2 \* 1000000 = 499 999 500 000

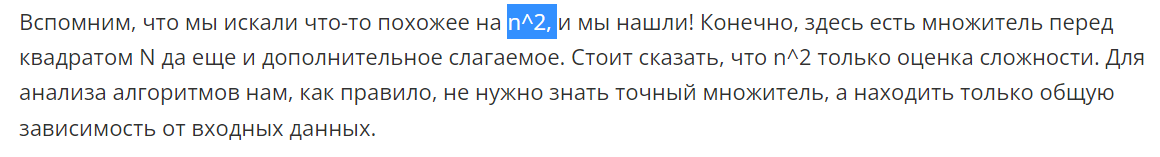
Таким алгоритмом можно посчитать такую прогрессию.



Сразу как-то не понял как произошло это упрощение. Нука…



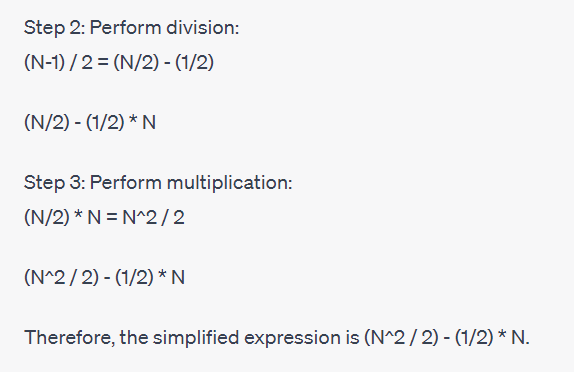
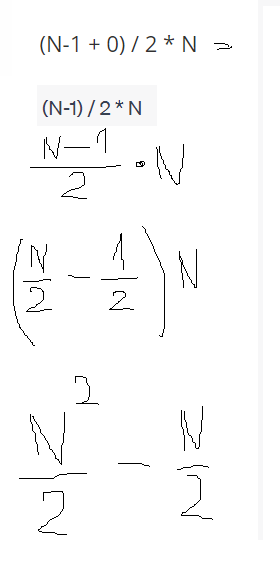
Да! Нахер там квадрат. Чтобы было похоже на линейное уравнение? x^2 + ax + b



Ясно, все для этого? Эм.. честно не понял, мб я считать разучился, возможно. Они просто выражение умножили еще раз на **N! Это не упрощение, а уложнение. Как они получили это число!!!**



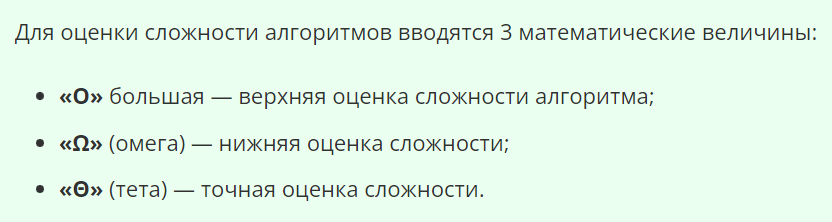
**Долбоебы!!!** Но как и я полагаю. GPT такой же ответ выдал, посмотрю как.

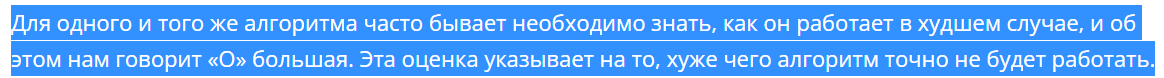
****

**Это я тупой(((**

**простите skillFactory.**

Случайно объединил операцию деления на 2 и N. Получилось будто полученное выражение я лишил одной степени N.

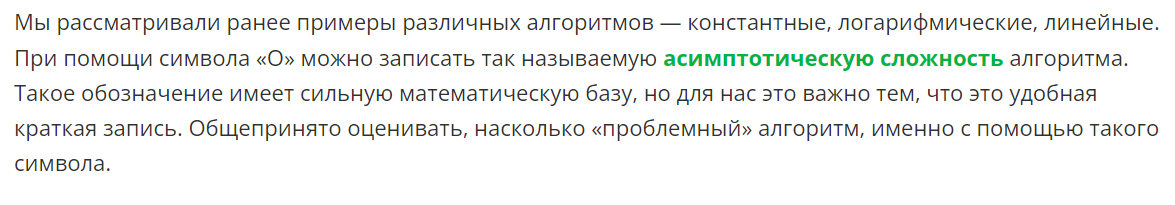


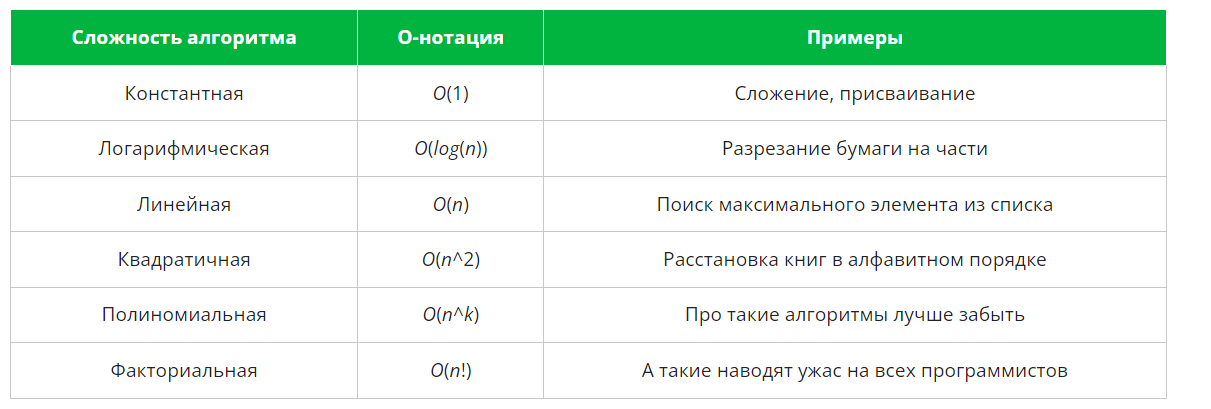


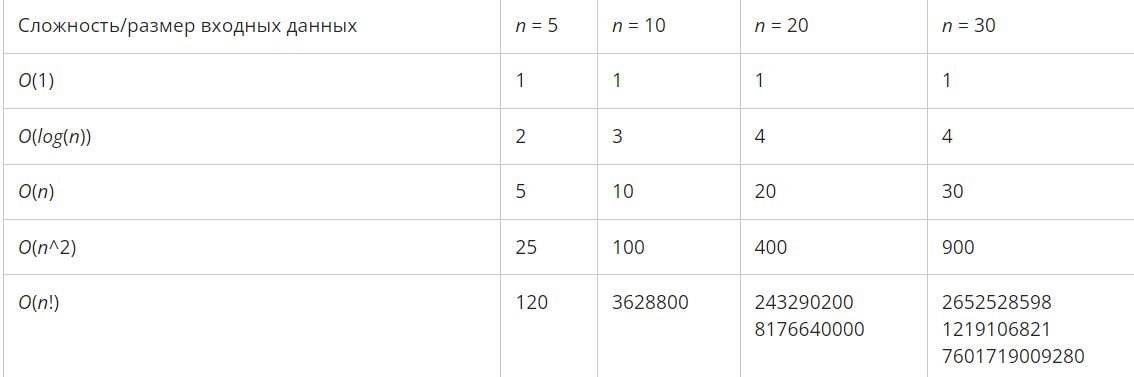
Не понял.

Это типо максимально сложная работа алгоритма сложнее которой он уже не отработает?

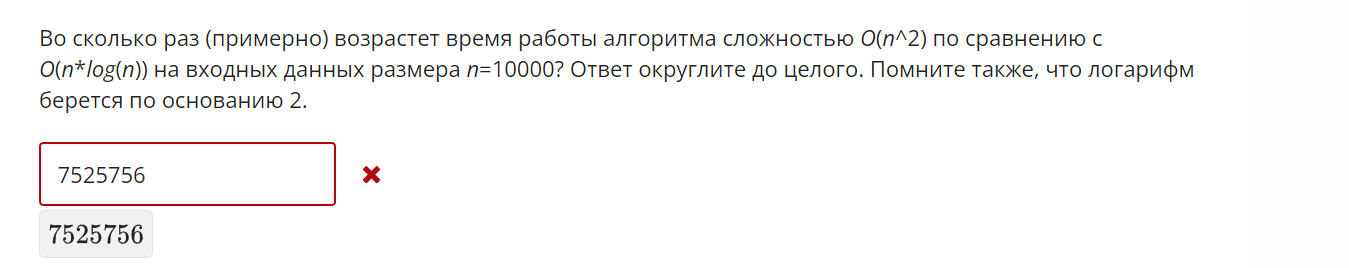
Тут видимо да речь он входных данных. Максимально плохих. Там.. террабайты данных или типо того…   
Нижняя оценка сложности, это работа алгоритма в идеальных условиях. Ну например при игре в морской бой компьютер внезапно с первого раза угадает все корабли.







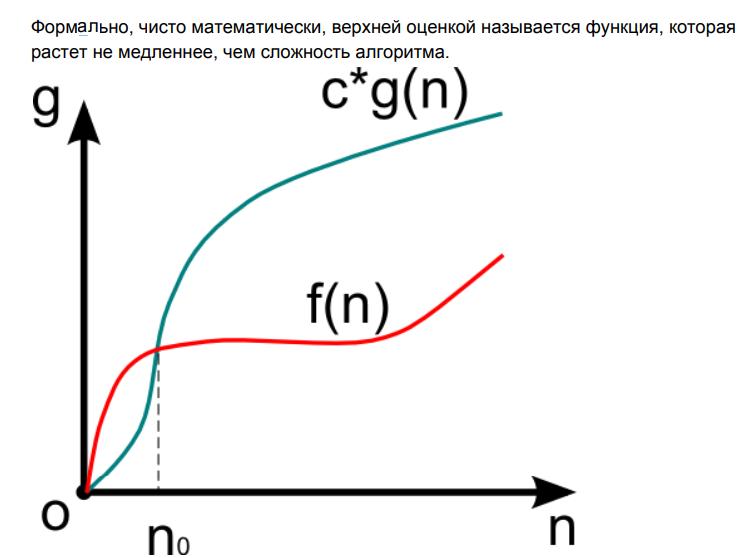
Для логарифма log2 30 = не 4, а 4.9

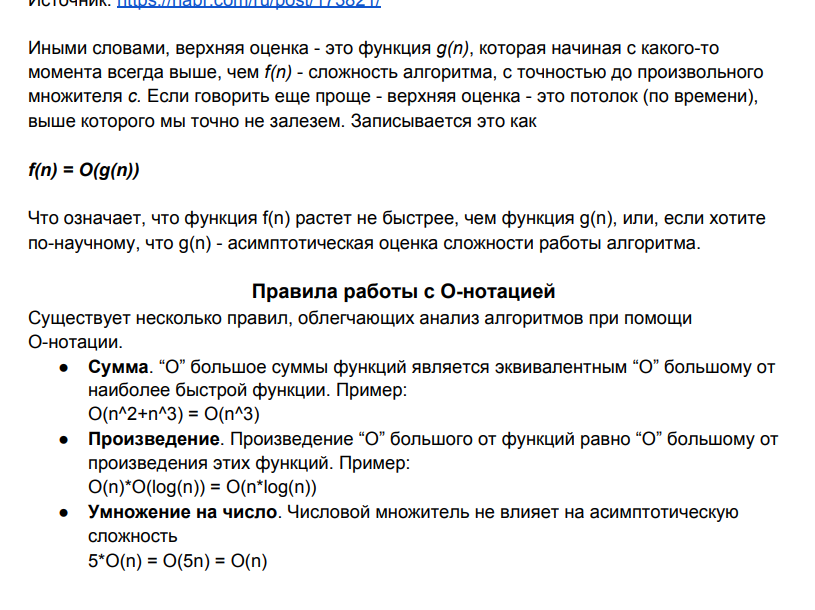


Ладно. В чем ошибка? Округлить вверх? Нет. А забыл на 10000 логорифм умножить.

132877.123 и 100000000. Ответ просто 752? **\*753**

Еее.. еще могу!!! (с калькулятором… ну и ладно, проги собственно для этого и пишут, чтобы самим не решать)



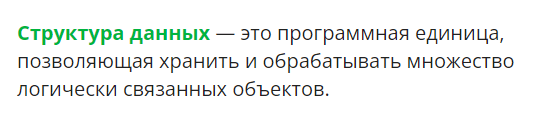


Ну правила понятны. Хотя их стоит повторять.

**B9.3. Основные структуры данных**

Нужно сделать перерыв, а то я сегодня совсем не работал. А сегодня я чуть поработал, а еще читал мангу. Еще чуть почитаю, и продолжу… Или нет. Ладно. Сегодня я и поработал и почитал и еще не поздно. Позанимаюсь.

Бла, бла, бла. Структуры данных. Свои классы, которые внутри себя могут содержать различные формы данных.

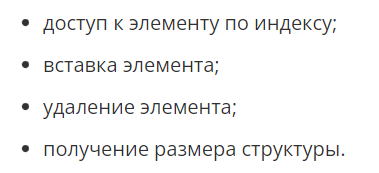


Сначала рассмотрим такие структуры данных как **массивы** и **Хэштаблицы.**

**Массивы** понятное дело, бывают статические и динамические. К структуре данных массива в **Python** относят списки. **Хэштаблицы** не уверен, но полагаю имеются ввиду что-то вроде словарей и тд.

Свои мысли закончил, а теперь что на деле…

Рассматривать структуры данных будем с точки зрения ее хранения в памяти, и способов взаимодействия с этой структурой. Например:



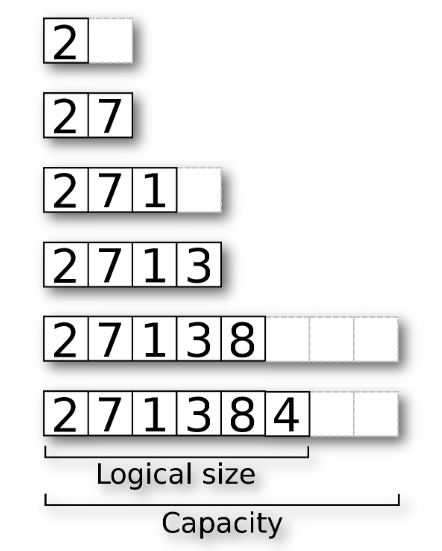
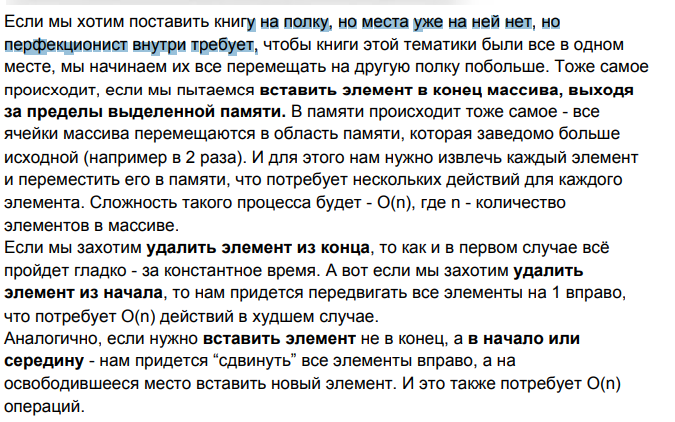
Если бы меня спросили что такое массив, я бы объяснил. Но термин не помню. А так,

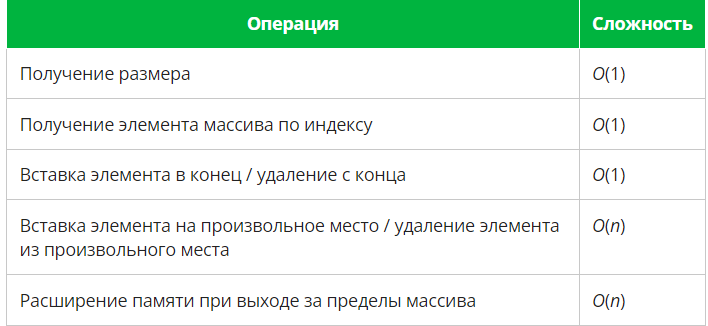
**массив –** это упорядоченный набор данных. Важно не путать упорядоченные структуры и отсортированные, о них тоже будет. Массивы бывают одномерных, двумерные и тд…

Особенность в том, что они хранят элементы в последовательных ячейках памяти. При его создании всегда нужно указывать какое количество памяти нужно занять.

Такие массивы называются **статическими**, помню в плюсах были.

Помимо них существуют **динамические массивы**. Они с помощью буферного механизма могут изменять свой изначальный размер. Важно не путать логический размер. То, насколько массив сейчас чем-то заполнен и **Capacity,** размер массива в памяти.



**List** в **python** не массив, а **динамический массив.** Да вы что…



На этом документе пока остановлюсь.

Какой интересный код. Есть модуль, который замеряет Сколько будет исполнятся тот или иной код.

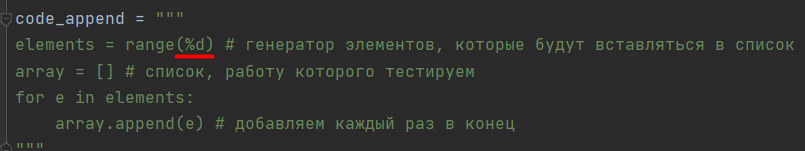
Модуль **timeit,** а функция:



Тоже **timeit.** В качестве первого аргумента она принимает функцию. Но сразу вопрос что это за процент **size**?

Сейчас объясню. Через него мы будем передавать в функцию **range**.

Сама функция должна быть передана в форме строки с корректными отступами. Потому что этот код будет действительно отработан во время проверки. Вот пример такой строки:



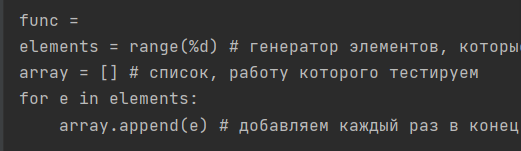
Элементарная код, которая создает список из **%d** чисел. И затем добавляет из этого списка эти числа в другой список через **append.** Именно для того, чтобы этот код можно было проверять этот код с разными аргументами и используется процент. Это схоже с **f**строками. Сначала идет строка внутри которой есть разные аргументы с процентами, а затем через % эти аргументы записываются по порядку. Вот простой пример из этой же программы, как такие проценты пишутся.



Только скажу честно не сильно практиковал. Но число **ratio,** оставляет только 2 знака после запятой. Но, что значит 5 с точкой я не помню.

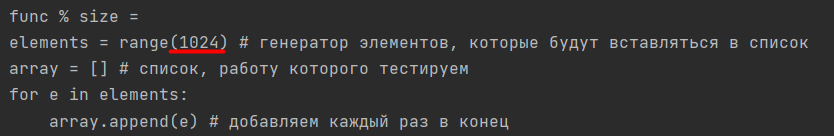
Так вот, если переменную func, попытаться вывести без size будет так:



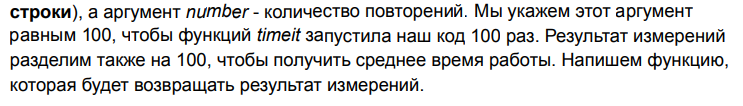


А если передать:





А второй аргумент в функции **timeit.timeit –** это **number.**



В итоге запусти эту функцию с добавляением:

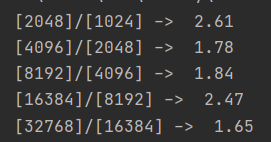
Получились такие примеры, так как время крайне мало, проще будет смотреть с числом больше 1. То произвели деление выполнения этой функции степени n + 1 / степени n



5 циклов от числа s 10, до 15.

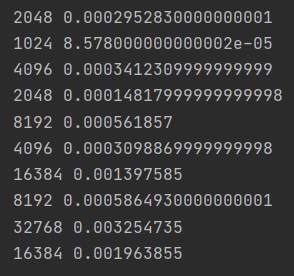


Результаты по времени:



Сейчас, прочитаю объяснение. Просто не знаю это время или память или хз что. Но чем больше код выполнялся, тем меньше результат. Если проводить обычное деление, то ответ будет 2. Но тут показатели range, в функции добавления.

Если рассматривать их отдельно без деления, то вот:

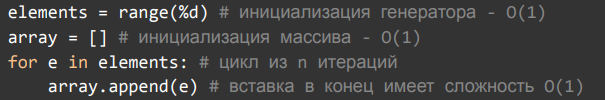


Не обманитесь числом 1024. Там есть е-05, Что значит чисто меньше в 10000 раз.

Так что говорят результаты:

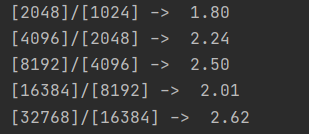
Результаты говорят, что с ростом размера массива, время выполнения увеличивается примерно в 2 раза. В таком случае сложность равна:





Посмотрим на ситуацию вставки элемента в начало массива:

Для метода **insert(0, e).** Результаты по мере роста длины массива от 2, до 2.5



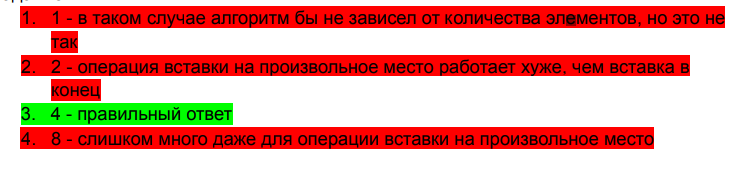
Ну в среднем все же 2.

Сложность такая же. ведь. 2n + константа наверно. Лан вот ответы:

**В среднем увеличилось в 4 раз. Сложность вот:**



**Но я не понял, как получить ответ 4. Вот типо объяснение:**

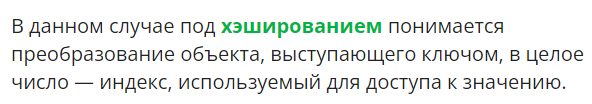


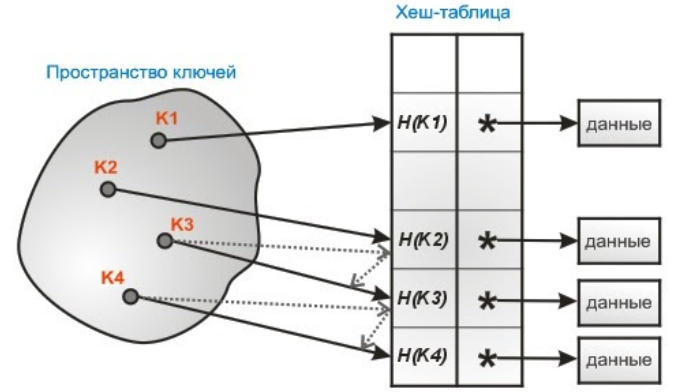
**В остальном познавательно.**

**Хэш-таблицы**

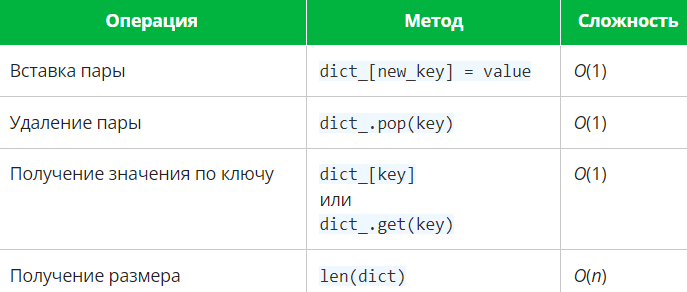
Массив нумерация которой не имеет смысла. Как я уже говорил, объекты, словари и тд.

Словарных тип данных использует такую структуру как Хэш таблицы. Думаю именно так она называется в общем, хотя на отдельных языках может называться по своему.





Из-за условий использования ключ должен быть неизменяемым и уникальным. В таком случае не будет никаких проблем с тем, чтобы получить по нему данных.



Может показаться, что раз уж практически у всего сложность константная, то почему хэш таблицы не используют всегда вместо массивов? Дело в том, что хоть сложность и константна. Но операции более тяжеловесны как по памяти, так и по времени, нежели другие структуры данных.

Тем не менее они очень удобны, особенно удобны при работе с базами данных. Надеюсь доживу до них.

**B9.4. Основные структуры данных: список, стек, очередь**

Стек, да помню. Всегда сравнивал его с патронами в магазине. Так, перерыв на мангу. Не видимо сегодня все.