Урок 8.  
Генераторы списков. Сортировки.

короткая линия

# План урока

1. Генераторы списков
2. Асимптотика алгоритмов
3. Квадратичные сортировки
4. Сортировка подсчетом
5. sorted() в Python

# Генераторы списков

Генераторы списков — удобный и красивый способ для создания нового или фильтрации существующего списка. Такую инструкцию можно записать в одну строку и легко понять даже неподготовленному человеку. Предположим, что мы хотим создать список из квадратов чисел из диапазона [1; 10]. Наверное, мы бы сделали это так:

|  |
| --- |
| a = [] **for** i **in** range(1, 11):  a.append(i \*\* 2) |

Есть ли способ записать то же самое короче и проще? Да — с помощью генератора.

|  |
| --- |
| a = [i \*\* 2 **for** i **in** range(1, 11)] |

Все генераторы строятся одинаково, по одному шаблону:

**[<что положить в список> for <индекс/элемент> in <диапазон/итерируемый тип данных> if <условие>]**

Условие может пригодиться, если, например, мы хотим добавить только четные числа. Условие также может быть сложным.

Если нам необходимо не только условие if, но и блок else, то подойдет следующая схема:

**[<положить в список значение> if <условие> else <положить другое значение> for <индекс/элемент> in <диапазон/итерируемый тип данных>]**

C помощью этой схемы напишем программу, которая выводит нули вместо нечетных чисел и квадраты четных чисел на отрезке [1, 100]:

|  |
| --- |
| print([x \*\* 2 **if** x % 2 == 0 **else** 0 **for** x **in** range(1, 101)] |

Еще один пример использования генераторов:

|  |
| --- |
| dates = [1964, 2001, 2077, 1914, 1991] *# фильтрация дат* filtered = [date **for** date **in** dates **if** date > 1989] |

**Задача “Быстрая инициализация”**

**Задача “Вокруг света за 80 итераций”**

**Задача “Уничтожение популяции”**

# Асимптотика алгоритмов

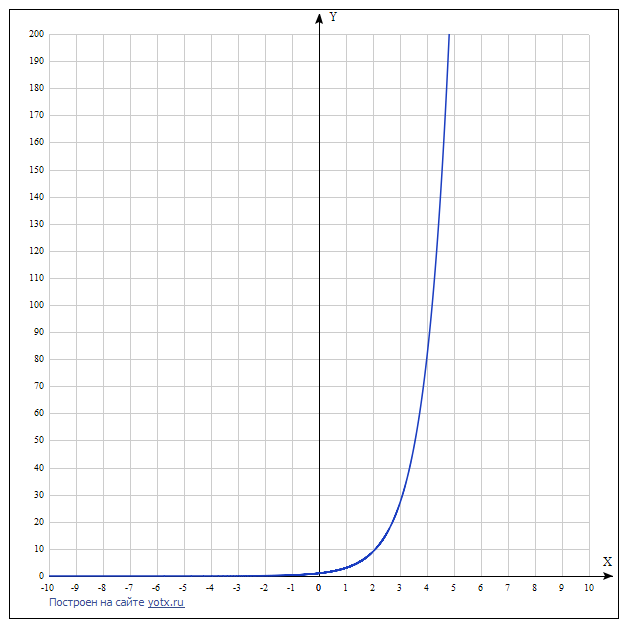
Мы уже сталкивались с простейшими алгоритмами несколько уроков назад, теперь же настало время поговорить о сортировках. Но прежде необходимо упомянуть про **сложность алгоритмов** (или, говоря по-умному, асимптотике).

Написав алгоритм, программист должен волноваться не только о правильности его работы, но и том, как алгоритм ведет себя на больших данных. Например, успешное выполнение на списках чисел из десяти элементов не гарантирует быстрое выполнение на списках из, скажем, 10 000 элементов, хотя на практике, в крупных приложениях, именно с такими данными приходиться сталкиваться чаще всего.

Оценка сложности связана прежде всего с оценкой времени работы алгоритма и количества необходимой памяти. Однако точное время работы — количество секунд или миллисекунд — не так важно. Важно то, **насколько увеличивается** время выполнения при увеличении объема входных данных. Внимательно посмотрите на данную картинку:



Обратите внимание, как незначительна разница для вычисления n \*\* 2 и насколько она **огромна** для вычисления 3 \*\* n. Это связано со сложностью функции и легко демонстрируется ее графиком (график стремительно уходит в бесконечность)



Чтобы донести информацию о сложности алгоритма, в теоретической информатике были приняты следующие обозначения:

O(1) — время выполнения не зависит от объема данных. К таким операциям можно отнести метод .append() или обращение к элементу по индексу.

O(n) — время зависит от размера списка **линейно.** Во сколько раз увеличивается список, в столько же раз увеличивается скорость выполнения. Пример: поиск элемента в списке — для этого нужно один раз пройтись по списку длины n.

O(n^2) — время зависит от квадрата (куба, четвертой степени и т. д.) размера списка. Пример: двойной цикл for, а также многие алгоритмы сортировки.

Также важны O(log n) и O(n \* log n) — это сложности продвинутых алгоритмов сортировки — однако мы на них не останавливаемся ввиду того, что логарифм проходится только в старших классах школы.

Если вам интересно, вы можете ознакомиться с темой более подробно по ссылкам:

<https://tproger.ru/articles/computational-complexity-explained/>

<https://tproger.ru/translations/algorithms-and-data-structures/>

<https://habr.com/post/196560/> (чуть сложнее)

**Задача “Таблица обозначений”**

# Квадратичные сортировки

Чаще всего изучение алгоритмов начинается именно с объяснения алгоритмов сортировки. Дональд Кнут посвятил третий том своего [фундаментального](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D0%BA%D1%83%D1%81%D1%81%D1%82%D0%B2%D0%BE_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) произведения сортировкам, сортировки пользуются популярностью на собеседованиях, а также вселяют страх в некоторых учеников. Обычно так получается, что это довольно нудный материал, и трудно представить, что он пригодится в будущем, в мире, где существуют такие замечательные вещи, как sorted(), однако именно сортировки лучше всего помогают “прочувствовать” алгоритмы.

**Сортировка выбором**

|  |
| --- |
| arr = [...] # инициализация списка for i in range(len(arr) - 1):  idx = i  for j in range(i + 1, len(arr)):  if arr[j] < arr[idx]:  idx = j  arr[i], arr[idx] = arr[idx], arr[i] |

**Сортировка вставками**

|  |
| --- |
| arr = [...] # инициализация списка for i in range(1, len(arr)):  save = arr[i]  j = i  while j != 0 and arr[j - 1] > save:  arr[j] = arr[j - 1]  j -= 1  arr[j] = save |

**Сортировка пузырьком**

|  |
| --- |
| arr = [...] # инициализация списка j = len(arr) - 1 is\_not\_ordered = True while is\_not\_ordered:  is\_not\_ordered = False  for i in range(j):  if arr[i] > arr[i + 1]:  arr[i], arr[i + 1] = arr[i + 1], arr[i]  is\_not\_ordered = True  j -= 1 |

**Задача “Наоборот”**

Эти сортировки имеют сложность O(n^2) из-за двойного цикла for.

Некоторые “интересности”:

Забавный пост о сортировках: <https://habr.com/post/323188/>

Сортировки наглядно: <https://proglib.io/p/sort-algorithms/>

Гифки :) : <https://proglib.io/p/sort-gif/>

Сортировки в танцах:

<https://www.youtube.com/playlist?list=PLuE79vNc5Wi6q34LsQcaJ7ISQ8uOyMaL_>

Если вдруг вам нечем заняться:

<https://www.youtube.com/playlist?list=PLEJyjB1oGzx2h88Tj90B5_HadLq339Cso>

# Сортировка подсчетом

|  |
| --- |
| # предположим, список состоит из положительных элементов max\_elem = max(arr) count = [0] \* (max\_elem + 1) for elem in arr:  count[elem] += 1 arr.clear() for i in range(len(count)):  arr += [i] \* count[i] |

Данная сортировка имеет сложность O(n + k), где k — количество элементов в диапазоне чисел списка. По этой причине этот алгоритм наиболее эффективен в том случае, когда список состоит из большого количества чисел, однако все они принадлежат узкому диапазону. В таком случае сложность стремится к O(n).

**Домашнее задание:**

**Задача “Контейнеры”**

**Задача “Подсчет отрицательных”**

# sorted() в Python

Функция sorted() в Python является модификацией сразу нескольких алгоритмов: сортировки вставками и сортировки слиянием. Такая сортировка носит название Timsort и признана одной из самых эффективных практически для любого набора данных:

<https://habr.com/ru/post/133996/> (не пугайтесь технических подробностей, поверхностного ознакомления будет достаточно)

В Ваших программах чаще всего понадобится писать именно sorted(), а не реализовывать свои алгоритмы сортировки, однако это не значит, что нужно пребывать в полнейшем неведении относительно реализации алгоритмов сортировки.