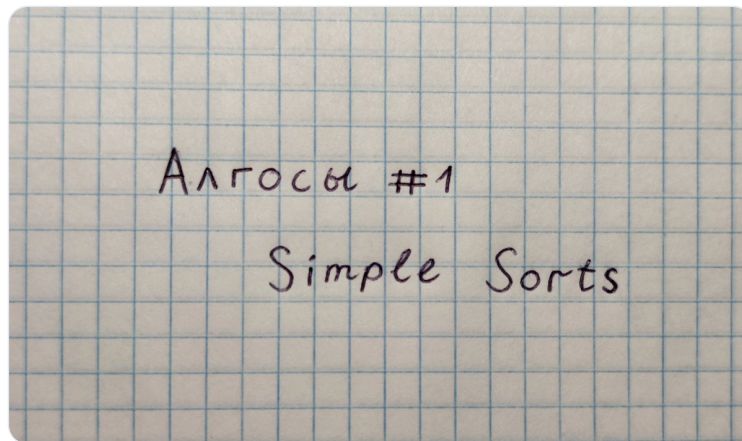




Валерий Жила @ValeriiZhyla

Jan 18, 2022 · 124 tweets · [ValeriiZhyla/status/1483346987248402435](https://twitter.com/ValeriiZhyla/status/1483346987248402435)

Сегодня я расскажу простым языком про три сортировки, с которых неизбежно начинается изучение алгоритмов. Первые шаги в создании алгоритмического мышления!



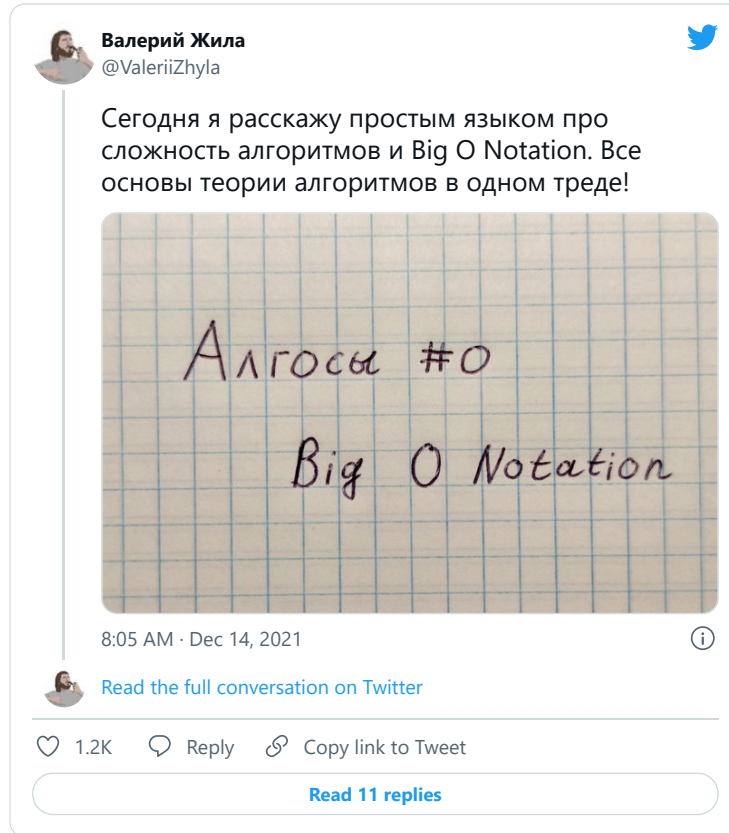
Речь пойдет о трёх самых базовых алгоритмах сортировки — о Bubble Sort, Insertion Sort и Selection Sort.

Цель всей истории — показать, какие идеи и мысли лежат в основе этой троицы, а также продемонстрировать их сильные и слабые стороны.

Мы выстроим эти алгоритмы по шагам, рассмотрим их простые версии и попробуем их немножко улучшить.

Френдли ремайндер о ретвитах моих тредов. Это простое действие делает сон крепче, здоровье стабильнее, пельмени сочнее и вызывает у ваших врагов рак рака.

Кстати, если вы новичок в теории алгоритмов, и тред с основами основ прошел мимо вас — я настойчиво, очень настойчиво советую вам ознакомиться. В нём всё про O-Нотацию и псевдокод, а так же приведена парочка простых алгоритмов.



Итак, дальше я действую и пишу исходя из того, что слова "Worst Case алгоритма по времени равен $O(n^2)$ " для вас не пустой звук. Иначе — последний настойчивый совет посетить Алгосы #0 моего авторства.

Вступление окончено. Чем именно мы займемся? Может, попробуем захватить мир?!

Лучше! Мы будем сортировать массивы целых чисел. Положительных. По возрастанию, от меньшего к большему. А ещё рисовать писать псевдокод и разглядывать примеры.

Time Complexity я буду называть "сложностью по времени", а Space Complexity — "сложностью по памяти".

Давайте разберем вопросы и загадки, который бы встретил гипотетический программист, впервые встретившийся с проблемой сортировки. Оставим его безымянным, чтоб было легче себя с ним ассоциировать.

Итак, вопрос номер раз: нужно ли отсортировать переданный массив (именно ЭТОТ массив), или лучше создать его копию, и сортировать уже её, не трогая и никак не модифицируя оригинал?

Давайте прикинем, на пальцах. В чём плюсы и минусы решения без создания копии?

Первое, и очевидное последствие: мы потеряем изначальный массив. Имел ли этот массив особую ценность? Зависит от конкретной ситуации. В основном, отсортированные данные хуже хаотичных не бывают.

Но всё же это побочный, или сторонний, эффект работы алгоритма. Их называют side effects.

Это очень важная штука, которую всегда нужно держать в голове. Лишний сторонний эффект отнесем к минусам.

Плюсы этой стратегии? Нам не придется выделять память под копию массива, что стоило бы $O(n)$ в копилку Space Complexity. Итого $O(1)$ памяти, in-place, красота.

К тому же массив не придется копировать в новое место, а значит экономится ещё $O(n)$ Time Complexity.

С in-place решением разобрались. Теперь посмотрим на решение с созданием копии и работе с ней, out-of-place.

В чём плюсы и минусы этого решения? Диаметриально противоположные! Никаких сторонних эффектов, зато ожидаемые дополнительные $O(n)$ памяти и времени.

Забегая наперёд, скажу, что лишние $O(n)$ времени в вопросах сортировок никакой глобальной роли не играют, так как сама сортировка сожрёт намного больше. А вот $O(n)$ памяти это серьезный аргумент избегать out-of-place алгоритмы, если возможно.

Никто же не любит приложения, которые жрут память гигабайтами, или выжирают всё, что доступно, и падают с ошибками? Правда ведь?



Вопрос номер два: как менять элементы массива местами?

Я бы опустил этот момент, но давайте для полноты картины напишем функцию, которая призвана эффективно переставлять два элемента по их индексам на место друг друга. Чтоб исключить появление белых пятен и недомолвок. Вот она:

```
1 def swap(int[] arr, fst_idx, snd_idx):
2     int temp = arr[fst_idx]
3     arr[fst_idx] = arr[snd_idx]
4     arr[snd_idx] = temp
```

Чуть больше контекста? Хорошо. Функция `swap` получает массив, и два индекса. Первым делом она сохраняет копию элемента по первому индексу (какое-то число) в отдельную переменную `temp` (от слова *temporary*, временный, очень частое сокращение, озадачивающее молодые умы).

`fst` это типичное сокращение для *first*, а `snd` — для *second*. Сокращением `idx` часто обозначают слово индекс, потому что мы очень не любим трать буквы впустую (особенно я, да) и писать слово *index* целиком.

Дальше она перезаписывает содержимое по первому индексу копией содержимого по второму индексу. В этот момент у нас появляется два одинаковых элемента, а оригинальное значение из ячейки с первым индексом утрачивается.

Благо мы сохранили его в переменную `temp`, и в конце записали в ячейку со вторым индексом.

Со своим разобрался!

Вопрос номер три, ключевой: как упорядочить массив нужным нам образом?

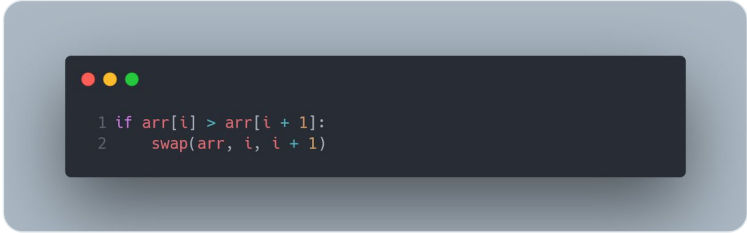
Хорошей идеей было бы начать со сравнения элементов между собой.

Сравнивать-то можно, но что делать дальше? Менять их местами! Хорошо, что в этом нам теперь сам чёрт не товарищ...

Упорядочивание с помощью сравнения и обмена местами звучит как что-то, что действительно может понизить уровень хаоса. Сразу возникает вопрос: “А какие именно элементы мы будем сравнивать?”.

Самый очевидный вариант — сравнивать элементы с соседними индексами. Логика элементарная: сравниваем таких соседей, если значение левого больше правого — меняем их местами.

Произвольный индекс массива назовём i . Следовательно, правый сосед будет иметь индекс $i + 1$. Теперь переварим последнюю мысль в маленький блок кода:


A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains two lines of Python code:

```
1 if arr[i] > arr[i + 1]:  
2     swap(arr, i, i + 1)
```

```
1 if arr[i] > arr[i + 1]:  
2     swap(arr, i, i + 1)
```

Этот код — замечательная отправная точка для сортировки массива от меньшего к большему.

Как бы его довести до ума? Для начала попробуем применить это действие на каждый индекс массива. Разумеется, кроме последнего, отсюда и $n-1$ — в противном случае наша программа натворила бы делов.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains four lines of Python code:

```
1 int n = arr.length  
2 for i in 0..n-1:  
3     if arr[i] > arr[i + 1]:  
4         swap(arr, i, i + 1)
```

```
1 int n = arr.length  
2 for i in 0..n-1:  
3     if arr[i] > arr[i + 1]:  
4         swap(arr, i, i + 1)
```

Будет ли наш массив после этого отсортирован? Нет, но станет чуть менее хаотичным! Попробуем посмотреть на каком-то простом примере:

```
arr: [4, 2, 1, 3, 7, 9, 0, 5, 8, 6]
--> loop
arr: [2, 1, 3, 4, 7, 0, 5, 8, 6, 9]
```

Что можно из него вынести? Во-первых, все элементы массива сместились. Почти все — в нужную сторону. Во-вторых, самый большой элемент массива занял своё законное место — девятка оказалась в самой правой ячейке!

Она поменялась местами с 0, потом с 5, потом с 8 и с 6, каждую итерацию “передвигаясь” или “всплывая” на своё законное место.

Прогоним наш код ещё несколько раз!


```
arr: [4, 2, 1, 3, 7, 9, 0, 5, 8, 6]
--> loop
arr: [2, 1, 3, 4, 7, 0, 5, 8, 6, 9]
--> loop
arr: [1, 2, 3, 4, 0, 5, 7, 6, 8, 9]
--> loop
arr: [1, 2, 3, 0, 4, 5, 6, 7, 8, 9]
```

Ситуация явно улучшилась, за всего три прохода. Каждый виток гарантирует нам “всплытие” одного из элементов на его позицию — после первого витка на место встало число 9, после второго — число 8, а дальше по счастливой случайности — сразу четыре элемента.

Сколько витков этих свопов нам нужно, чтоб гарантированно отсортировать массив? Если каждый виток ставит на место минимум один элемент, то через n витков мы гарантированно получим отсортированный массив!

Даже через $n-1$, потому что для самого правого элемента и так останется только одна возможность размещения.

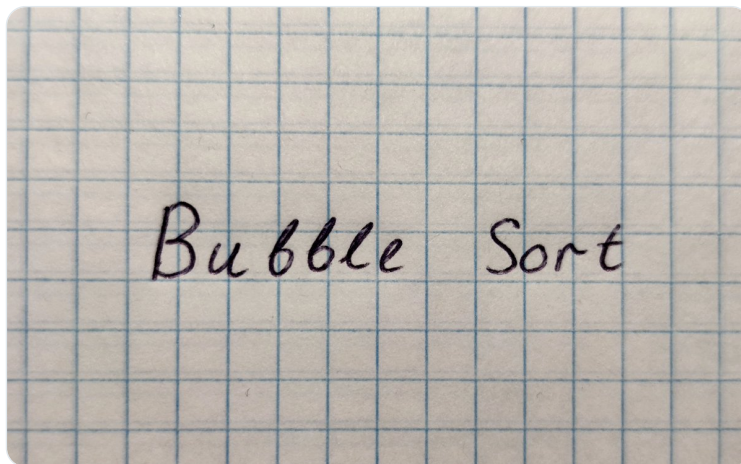
Взглянем на этот код:



```
1 int n = arr.length
2 for j in 0..n-1:
3     for i in 0..n-1:
4         if arr[i] > arr[i + 1]:
5             swap(arr, i, i + 1)
```

Великолепно! Мы только что собрали из ничего первый алгоритм сортировки!

Итерацию за итерацией большие числа “всплывают” наверх, а маленькие “тонут” вниз. Кто-то давно, до нашего рождения, увидел в этом аналогию с пузырьками. Поэтому такая сортировка называется пузырьковой, или же Bubble Sort.



Наша версия “пузырька” вышла максимально наивной — $O(n^2)$ по времени и в худшем, и в лучшем случае. Звучит как план, надёжный, как швейцарские часы, но его можно улучшить!

Во-первых, мы знаем, что каждое выполнение внутреннего цикла гарантированно ставит на место один правый элемент. Значит, с каждым новым j можно сдвигать эту “границу передвигания” на единичку, что и происходит на строке 3. Зачем лезть туда, где точно ничего не произойдет?

```
1 int n = arr.length
2 for j in 0..n-1:
3     for i in 0..n-1-j:
4         if arr[i] > arr[i + 1]:
5             swap(arr, i, i + 1)
```

Уже лучше, но всё ещё много дурной работы. Как бы нам добиться того, чтоб программа могла остановиться, как только массив будет отсортирован? Раньше срока, так сказать, если работа выполнена.

Как это распознать? Например, если массив отсортирован, то следующий внутренний цикл не сделает ни единого свопа! За это определённо стоит зацепиться.

```
1 int n = arr.length
2 bool was_swapped = false
3 for j in 0..n-1:
4     was_swapped = false
5     for i in 0..n-1-j:
6         if arr[i] > arr[i + 1]:
7             swap(arr, i, i + 1)
8             was_swapped = true
9     if was_swapped == false:
10        break
```

Что мы сделали? Ввели новую переменную, которая в начале каждого внешнего цикла устанавливается на `false`, и после каждого свопа меняется на `true`.

Если внутренний цикл не сделает ни одного свопа, переменная останется со значением `false` и прервет внешний цикл.

Чувствуете это? Это наш Best Case стал $O(n)$!

Программе с изначально отсортированным массивом потребуется лишь провести один проход внутреннего цикла, распознать отсутствие изменений и прервать внешний цикл.

Вот это уже смахивает на Bubble Sort, который и на собесе не стыдно нарисовать!

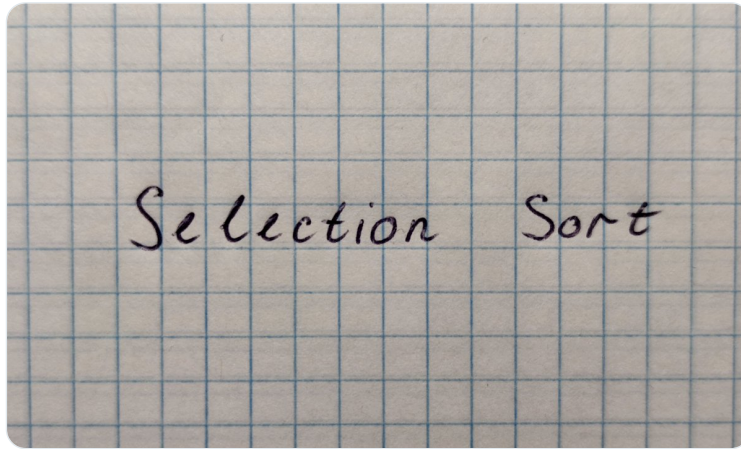
Облачим этот код в функцию и продолжим наш полёт фантазии:

```
1 def bubble_sort(int[] arr):
2     int n = arr.length
3     bool was_swapped = false
4     for j in 0..n-1:
5         was_swapped = false
6         for i in 0..n-1-j:
7             if arr[i] > arr[i + 1]:
8                 swap(arr, i, i + 1)
9                 was_swapped = true
10        if was_swapped == false:
11            break
12    return arr
```

Для протокола:

Name	Bubble
Worst	$O(n^2)$
Average	$O(n^2)$
Best	$O(n)$
Space	$O(1)$

Двигаемся дальше! Наш новый подопытный — сортировка выбором, она же Selection Sort!



Ранее мы увидели алгоритм, основанный на сравнении соседних элементов. Какой ещё способ можно выбрать для сортировки? Способ этот называется Selection Sort, он же “Сортировка Выбором”. И сейчас станет ясно, почему именно выбором, и что мы собираемся “выбирать”.

Начнём с небольшой воображаемой сцены.

Стоит мальчик (можете заменить на дедушку, бабушку, котика или герань, на свой вкус), а перед ним на столике хаотично лежат бумажки с разными числами. Ему нужно (по зловещей задумке автора) эти числа сложить в рядок, по возрастанию.

Он находит из этого набора бумажек одну с самым маленьким числом, и откладывает его на соседний столик. Возвращается к хаосу из бумажек с числами, находит очередное наименьшее число, забирает его и кладет на второй столик, справа от предыдущего.

Так, шаг за шагом, мальчик выстроит стройную шеренгу из чисел. Несложно догадаться, что получившийся аналог массива будет отсортирован по возрастанию — слева лежит самое маленькое число, справа — самое большое.

Примерно так же работает "наивный" Selection Sort. Давайте попробуем изобразить его!

Во-первых, нам нужна функция для поиска минимума в массиве. Точнее, для поиска индекса наименьшего элемента. Например, такая:

```
1 def find_min(int[] arr):
2     int current_min_idx = 0
3     int current_min_value = arr[0]
4     for i in 0..arr.length:
5         if (arr[i] < current_min_value):
6             current_min_value = arr[i]
7             current_min_idx = i
8     return current_min_idx
```

Можно было бы обойтись без `current_min_value`, но я всё-таки оставлю эту переменную, исключительно для наглядности.

Суть `find_min()` максимально проста: за отправную точку берём первый элемент массива, называем это текущим минимумом. Далее поочерёдно сравниваем элементы массива с текущим минимумом, обновляя его значение, если обнаруживается элемент с меньшим значением.

При этом мы принимаем за данность, что все ячейки массива `arr` заполнены числами, без пустых ячеек, и в массиве есть минимум один элемент. В противном случае `find_min()` станет ещё более уродливым.

Ещё нам нужна функция удаления элемента из массива. Исключительно красоты ради, максимально простая функция:

```
1 def delete_element(int[] arr, int idx):
2     arr[idx] = null
```

Итак, приготовления завершены, можно думать о самой сортировке. Оригинальный массив мы будем урезать шаг за шагом, а сам ответ будет выстраиваться во втором массиве. Значит, начать следует с создания пустого массива, с тем же размером, что и у оригинала.

```
1 int n = arr.length
2 int[] sorted_arr = new int[n] # empty int array with n cells
```

А дальше мы засунем шаги из недавнего мысленного эксперимента в наш алгоритм!

```
1 def naive_selection_sort(int[] arr):
2     int n = arr.length
3     int[] sorted_arr = new int[n] # empty int array with n cells
4     for i in 0..n:
5         int idx_of_min_element = find_min(arr)
6         sorted_arr[i] = arr[idx_of_min_element]
7         delete_element(arr)
8     return sorted_arr
```

Какие к этому товарищу могут возникнуть замечания? Во-первых, он жрёт память аки чорт — из-за необходимости создать отдельный массив под результаты, получаем с порога $O(n)$ по памяти прямо в кадык.

Во-вторых, код необходимо немного переделать, так как `delete_element()` создает пустоты в арт, о которые споткнется `find_min()`.

В-третьих, по времени исполнения алгоритм не блещет — $O(n^2)$ как в худшем, так и в лучшем случае.

Возникает резонный вопрос — что делать?

Великолепным лайфхаком в этой ситуации является “переиспользование” места в массиве под остальные нужды. Что значит “переиспользование”? Дед, ты экологом заделался?

Сейчас будет чуть-чуть замешательства, а дальше всё станет на свои места, обещаю!

Итак, вот код оптимального Selection Sort, который переиспользует массив арт, наращивая в нём же отсортированный массив:

```
1 def selection_sort(int[] arr):
2     int n = arr.length
3     for i in 0..n:
4         min_idx = find_min_between(arr, i + 1, n)
5         swap(arr, i, min_idx)
6     return arr
```

Компактно, правда? Я добавил новую функцию, `find_min_between(array, start_idx, end_idx)`. Работает точно так же, как наш старый знакомый `find_min(array)`, только проходит не по всему массиву, а только по кусочку между `start_idx` и `end_idx`.

Думаю, что наш второй старый знакомый, `swar(arr, i, j)`, в представлении не нуждается.

Итак, теперь нужно объяснить, как, и главное, почему это работает. Для начала, возьмём массив `arr` с десятью числами:

```
arr = [94, 12, 35, 88, 22, 39, 15, 76, 81, 98]
```

Для затравки посмотрим на состояние `arr` после нескольких проходов цикла в `selection_sort()`, то есть первых, например, пяти итераций. Просто чтобы попытаться заметить, что же такого делает наш алгоритм. Итак, состояния:

```
arr: [12, 94, 35, 88, 22, 39, 15, 76, 81, 98]    i: 0
arr: [12, 15, 35, 88, 22, 39, 94, 76, 81, 98]    i: 1
arr: [12, 15, 22, 88, 35, 39, 94, 76, 81, 98]    i: 2
arr: [12, 15, 22, 35, 88, 39, 94, 76, 81, 98]    i: 3
arr: [12, 15, 22, 35, 39, 88, 94, 76, 81, 98]    i: 4
```

А теперь давайте разбираться. Для переиспользования левой части массива `arr` под сохранение отсортированного массива наш алгоритм использует “барьер” между отсортированной и неотсортированной частью.

“Барьер” этот находится ровноёхонько между соседними элементами по индексам `i` и `(i + 1)`.

При `i=1` в отсортированной части массива будет только один элемент. При `i=2` их будет уже два. В каждой итерации наш алгоритм будет искать наименьшее значение в правой части (от `i + 1` и до конца массива), и менять его местами с элементом по индексу `i`.

Сделаем весь процесс чуть более наглядным, добавив разделитель между элементами, по которым проходит граница между отсортированной и неотсортированной частями массива:

```
arr: | [94, 12, 35, 88, 22, 39, 15, 86, 81, 98]    i: 0
arr: [12] | [94, 35, 88, 22, 39, 15, 86, 81, 98]    i: 1
arr: [12, 15] | [35, 88, 22, 39, 94, 86, 81, 98]    i: 2
arr: [12, 15, 22] | [88, 35, 39, 94, 86, 81, 98]    i: 3
arr: [12, 15, 22, 35] | [88, 39, 94, 86, 81, 98]    i: 4
arr: [12, 15, 22, 35, 39] | [88, 94, 86, 81, 98]    i: 5
arr: [12, 15, 22, 35, 39, 81] | [94, 86, 88, 98]    i: 6
arr: [12, 15, 22, 35, 39, 81, 86] | [94, 88, 98]    i: 7
arr: [12, 15, 22, 35, 39, 81, 86, 88] | [94, 98]    i: 8
arr: [12, 15, 22, 35, 39, 81, 86, 88, 94] | [98]    i: 9
arr: [12, 15, 22, 35, 39, 81, 86, 88, 94, 98] |    i: 10
```

Уточню, что состояние *arr* показывается каждый раз в конце соответствующей итерации. Первая строчка после первой итерации (при *i* равным нулю), и дальше по списку.

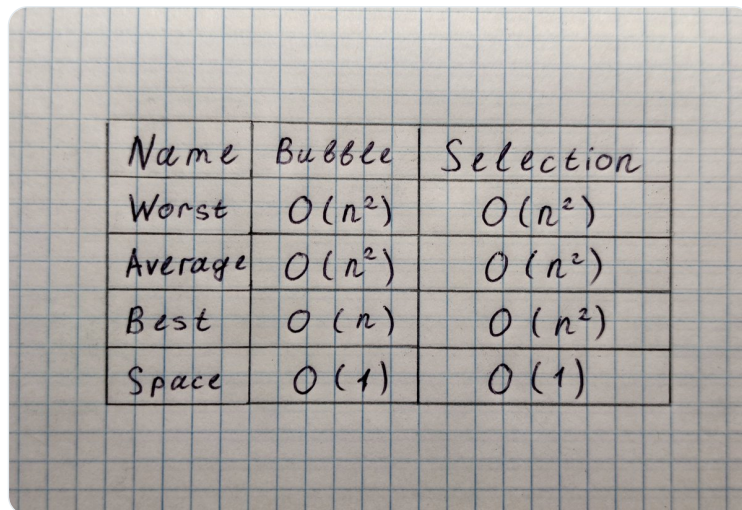
Возможно, поставить самый большой элемент массива в последнее место было не самой полезной идеей...

Надеюсь, что идея Selection Sort теперь кажется простой и естественной. Давайте взглянем на его характеристики. Во-первых, у этой реализации $O(1)$ по памяти, за счёт использования левой части массива. Это хорошо.

Во-вторых, за счёт поиска минимального элемента (есть $O(n)$ времени) в каждом цикле (n раз), мы получаем как Worst Case, так и Best Case $O(n^2)$. Это не хорошо.

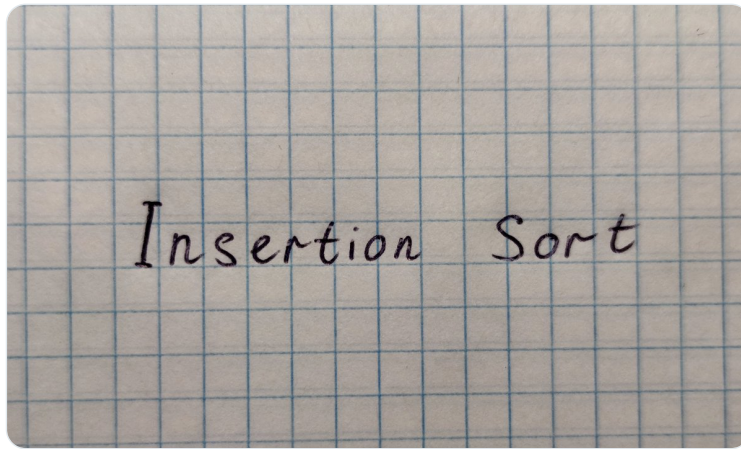
Выходит, что Bubble Sort всегда лучше Selection Sort по скорости, из-за Best Case $O(n)$? Тут, как в одном известном анекдоте, есть один нюанс. В конце треда мы поговорим про представленные алгоритмы и сравним те их аспекты, которые не учитываются в O-Нотации.

Для протокола:



Name	Bubble	Selection
Worst	$O(n^2)$	$O(n^2)$
Average	$O(n^2)$	$O(n^2)$
Best	$O(n)$	$O(n^2)$
Space	$O(1)$	$O(1)$

Последний герой на сегодня, Insertion Sort!



Insertion Sort, он же Сортировка Вставками. В отличие от предыдущих сортировок, тут мы обойдемся без хождений вокруг да около. Почему? Дед, халтуришь?

Дело в том, что эта сортировка очень похожа на предыдущий Selection Sort. Я даже за глаза его называю “Selection Sort наоборот”.

"Наоборот" не из-за другого направления сортировки, а из-за “обратности” самого принципа.

В Selection Sort мы искали наименьший элемент в правой части, присоединяли его на кончик левой и шли дальше. Самая дорогая операция — поиск наименьшего элемента.

В Insertion Sort мы так же используем левую часть массива под отсортированную последовательность. Вот только логика немного другая: мы берём первый попавшийся элемент из правой части, и ищем, на какое место его вставить в левую.

Вставка в левую часть возможно оттого, что левая часть по своей конструкции всегда отсортирована. Значит, всё, что нужно сделать — найти подходящее место, сдвинуть всех правых соседей на одну позицию правее и вставить новый элемент на освободившееся место.

Как работает эта вставка? Немножко напоминает толкучку в общественном транспорте. Вошедший боксёр-тяжеловес в забитый трамвай, вероятно, спровоцирует движение всех мирно стоящих и уставших людей на пол шажочка от выхода.

Вот маленький пример этой вставки, только вместо трамвая у нас отсортированный массив.

В этом массиве содержится восемь чисел, а так же есть одно свободное место в конце. В массив мы хотим добавить число 5. Добавить, конечно, на правильное место. Выглядеть алгоритм будет так:

```
1 int[] arr = {1, 2, 3, 4, 6, 7, 8, 9, _}  
2 int key = 5  
3 int element_to_move_idx = arr.length - 2  
4 # arr[arr.length - 2] = 9  
5  
6 while (element_to_move_idx >= 0) and (key < arr[element_to_move_idx]):  
7     arr[element_to_move_idx + 1] = arr[element_to_move_idx]  
8     element_to_move_idx--  
9 # insert new value into free cell  
10 arr[element_to_move_idx + 1] = key
```

Как будет выглядеть передвижение чисел, которые больше пятерки? Начнётся оно вот так:

```
arr: [1, 2, 3, 4, 6, 7, 8, 9, _]  
arr: [1, 2, 3, 4, 6, 7, 8, 9, 9]  
arr: [1, 2, 3, 4, 6, 7, 8, 8, 9]
```

Обратите внимание на то, что справа от передвигаемого числа возникает его копия, а на следующей итерации оригинал перезаписывается.

И дойдёт сдвигание до такого вида:

```
arr: [1, 2, 3, 4, 6, 7, 8, 9, _]  
arr: [1, 2, 3, 4, 6, 7, 8, 9, 9]  
arr: [1, 2, 3, 4, 6, 7, 8, 8, 9]  
arr: [1, 2, 3, 4, 6, 7, 7, 8, 9]  
arr: [1, 2, 3, 4, 6, 6, 7, 8, 9]
```


После чего, обнаружив четвёрку, цикл остановится, и запишет заветную пятёрку справа от четвёрки, на место “мусорной” шестёрки.

```
arr: [1, 2, 3, 4, 6, 7, 8, 9, _]  
arr: [1, 2, 3, 4, 6, 7, 8, 9, 9]  
arr: [1, 2, 3, 4, 6, 7, 8, 8, 9]  
arr: [1, 2, 3, 4, 6, 7, 7, 8, 9]  
arr: [1, 2, 3, 4, 6, 6, 7, 8, 9]  
arr: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Предлагаю теперь взглянуть на код заветной сортировки вставками!

```
1 def insertion_sort(int[] arr):  
2     int n = arr.length  
3     for i in 1..n:  
4         key = arr[i]  
5         # Move all elements in sorted part that are greater than key  
6         int element_to_move_idx = i-1  
7         while (element_to_move_idx >= 0) and (key < arr[element_to_move_idx]):  
8             arr[element_to_move_idx + 1] = arr[element_to_move_idx]  
9             element_to_move_idx--  
10        # Insert new value into free cell  
11        arr[element_to_move_idx + 1] = key  
12    return arr
```

Помните пример из Selection Sort с барьером? Мне вот он понравился. Так как алгоритм похож, и принцип использования левой части массива под отсортированный массив, предлагаю не ходить вокруг да около и просто посмотреть на его работу:

```
arr: | [35, 14, 98, 77, 23, 17, 56, 29, 81, 90]      i: 0  
arr: [35] | [14, 98, 77, 23, 17, 56, 29, 81, 90]    i: 1  
arr: [14, 35] | [98, 77, 23, 17, 56, 29, 81, 90]    i: 2  
arr: [14, 35, 98] | [77, 23, 17, 56, 29, 81, 90]    i: 3  
arr: [14, 35, 77, 98] | [23, 17, 56, 29, 81, 90]    i: 4  
arr: [14, 23, 35, 77, 98] | [17, 56, 29, 81, 90]    i: 5  
arr: [14, 17, 23, 35, 77, 98] | [56, 29, 81, 90]    i: 6  
arr: [14, 17, 23, 35, 56, 77, 98] | [29, 81, 90]    i: 7  
arr: [14, 17, 23, 29, 35, 56, 77, 98] | [81, 90]    i: 8  
arr: [14, 17, 23, 29, 35, 56, 77, 81, 98] | [90]    i: 9  
arr: [14, 17, 23, 29, 35, 56, 77, 81, 90, 98] |    i: 10
```

Вот тебе, бабушка, и Insertion Sort.

Нижняя граница производительности довольно привлекательна. Если массив уже отсортирован, то никакое сдвигание и не понадобится! Best Case этой сортировки составляет $O(n)$.

Для протокола:

Name	Bubble	Selection	Insertion
Worst	$O(n^2)$	$O(n^2)$	$O(n^2)$
Average	$O(n^2)$	$O(n^2)$	$O(n^2)$
Best	$O(n)$	$O(n^2)$	$O(n)$
Space	$O(1)$	$O(1)$	$O(1)$

Мы разобрали Великую Троицу Сортировок, с которыми неизбежно сталкивается любой человек, изучающий алгоритмы. С чем я всех и поздравляю!

Теперь пришло время немножко поговорить о стабильности сортировок. Она же устойчивость, она же stability. Что это такое?

Если мы сортируем числа, то стабильность особой роли не играет. А вот если мы сортируем объекты, то очень даже. Например, у нас есть массив с записями о клиентах, и сортировать мы их будем по возрасту.

Возьмём для наглядности максимально простой вид записей, в формате [возраст : имя]

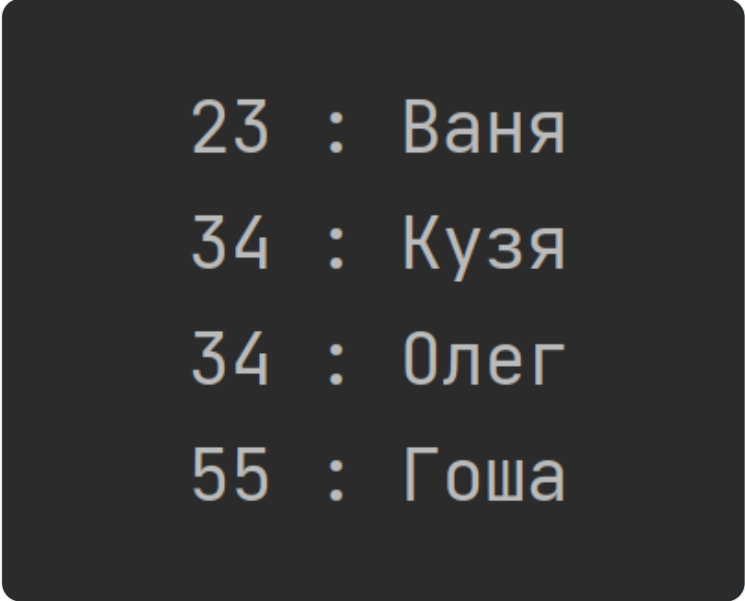
34 : Кузя

23 : Ваня

55 : Гоша

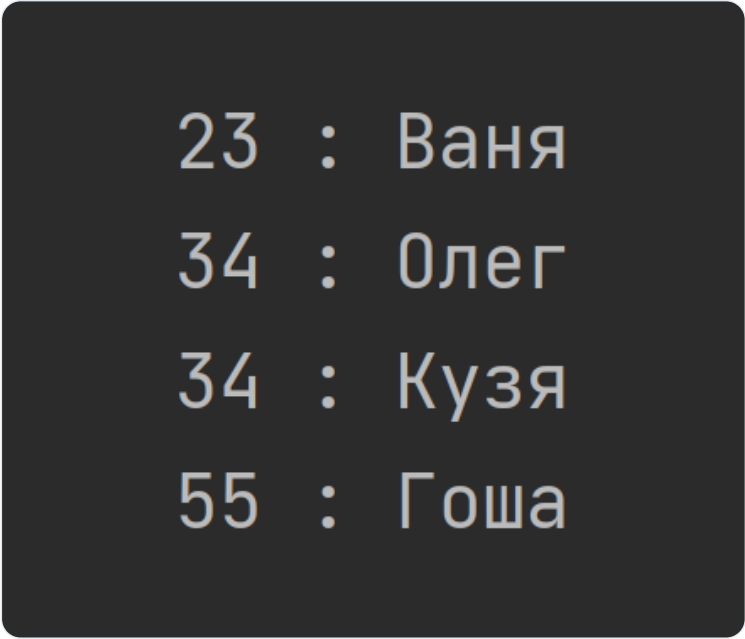
34 : Олег

Стабильным называется алгоритм сортировки, который не меняет порядок элементов с одинаковыми ключами (возраст) относительно друг друга. В стабильном варианте после сортировки Кузя гарантированно будет раньше Олега, так как в оригинальных данных было так, а ключ одинаков.



23	:	Ваня
34	:	Кузя
34	:	Олег
55	:	Гоша

В нестабильном варианте допускается и второй сценарий:



23	:	Ваня
34	:	Олег
34	:	Кузя
55	:	Гоша

Стабильность крайне важна, если мы планируем сортировать массив объектов по нескольким ключам. Например, сначала по возрасту, потом по размеру банковского счёта, и дальше по квадратным метрам жилплощади. Нестабильный алгоритм тут испортит нам всю малину.

Из нашей тройцы стабильными являются Bubble Sort и Insertion Sort, а вот Selection Sort так и норовит перемешать элементы.

Докинем эту строчку в нашу табличку, чем приведем её в финальный вид:

Name	Bubble	Selection	Insertion
Worst	$O(n^2)$	$O(n^2)$	$O(n^2)$
Average	$O(n^2)$	$O(n^2)$	$O(n^2)$
Best	$O(n)$	$O(n^2)$	$O(n)$
Space	$O(1)$	$O(1)$	$O(1)$
Stable	yes	no	yes

Помните, я обещал обсудить нюансы, выходящие за О-Нотацию и дополняющие её? Время перемывать косточки пришло!

Давайте посмотрим на массив, который почти идеально отсортирован. Одна беда — самый большой элемент в нём стоит на первом месте:

[98, 14, 17, 23, 29, 35, 56, 77, 81, 90]

Как поведут себя наши алгоритмы сортировки?

Итак, Bubble Sort сработает великолепно — за n свопов массив будет готов к употреблению.

Insertion Sort тоже хорош в этом вопросе.

А вот Selection Sort начнет чудить, и перелопатит со страшной расточительностью весь массив, получив $O(n^2)$ по времени.

Проведите мысленный эксперимент, полезно для закрепления!

Если данные почти отсортированы, то Bubble Sort и Insertion Sort — наши лучшие друзья.

Теперь давайте подумаем про стоимость операций. В О-Нотации это опускается, но разные операции операциям рознь.

Прочитать ячейку массива — элементарно. Переписать ячейку массива — вроде бы тоже просто, да вот только многопоточность, обновление значений в кэше процессора, и многие другие факторы реального мира с нами не согласны.

Сойдемся сейчас на мысли, что на самом деле операция swap не так уж и проста, как кажется. Что у нас по количеству свопов?

Bubble Sort тут просто отбитый, свопает всё, свопает много, свопает, свопает, свопает. Количество операций записи, которые он выполняет, просто зашкаливает. Insertion Sort тоже не подарок — постоянное сдвигание вправо до добра не доведет.

А вот Selection Sort тут просто конфетка: всего n свопов в худшем случае! Очень экономный малый.

Итог: если поставлена задача экономить операции записи, то Selection Sort является рекордсменом!

Что до памяти, то наша тройка в этом одинаково хороша. Все жрут $O(1)$ памяти, то есть работают in-place.

Важно: алгоритмы сортировки можно улучшать. В основном эти улучшения касаются использования особых структур данных.

Маленький пример - Insertion Sort можно ускорить раза в два, а так же свести количество свопов к показателям Bubble Sort, если использовать не массив, а связный список.

A Selection Sort таки можно сделать стабильным.

О списках мы ещё поговорим в следующих тредах, так что пока не буду углубляться. Сейчас скажу только, что добавить элемент в середину такого списка — очень дешево, так как отпадает вся необходимость в сдвигании правых соседей по одному.

На этом наша долгая история про три базовые сортировки подходит к концу. Хотел сделать покороче, а вышло как всегда...

Код этих сортировок на разных языках программирования я приводить не буду, но всё можно найти по следующим ссылкам:

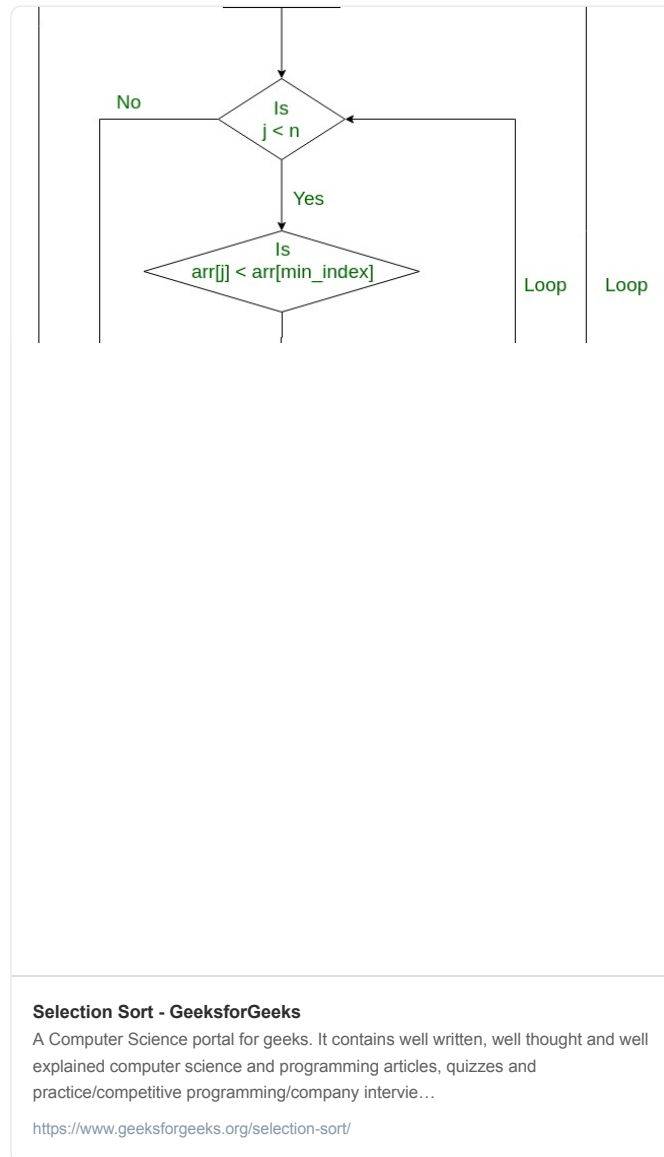
Bubble Sort:

i = 1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			

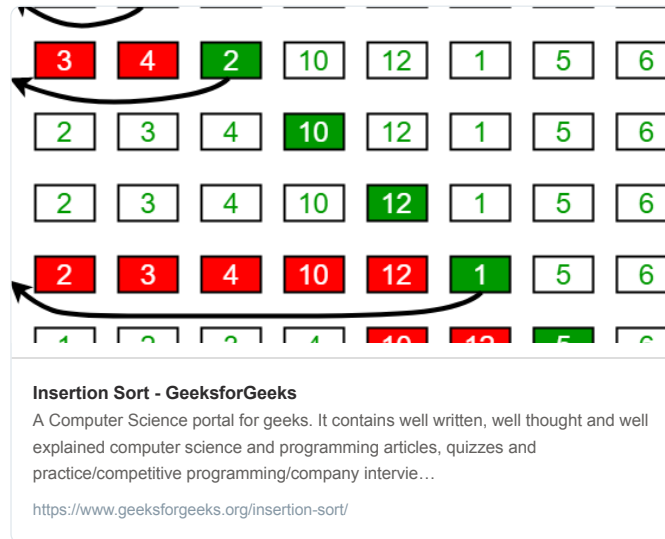
Bubble Sort - GeeksforGeeks
A Computer Science portal for geeks. It contains well written, well thought and well explained computer science and programming articles, quizzes and practice/competitive programming/company interview...

<https://www.geeksforgeeks.org/bubble-sort/>

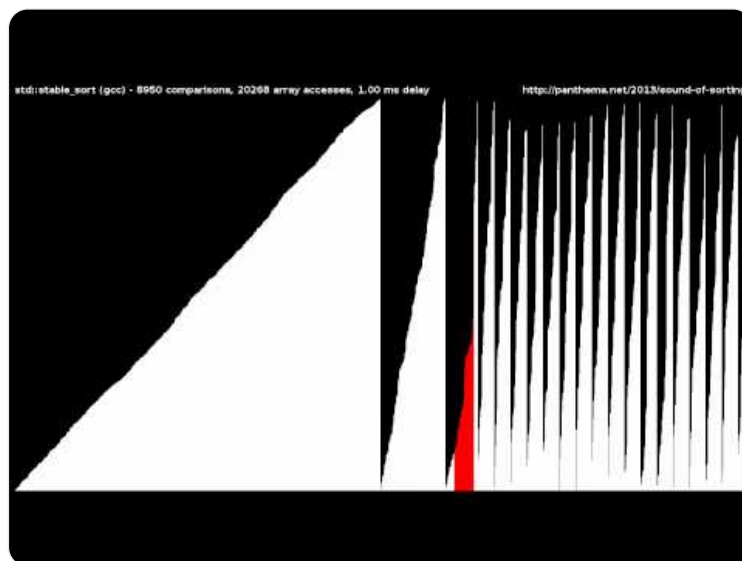
Selection Sort:



Insertion Sort:



Визуализаций этих алгоритмов на ютубе пруд пруди. Из них всех я бы выделил вот это видео:



<https://www.youtube.com/embed/kPRA0W1kECg>

Также я люблю видосы с массивами с GeeksforGeeks, на страницах по ссылкам сверху. Отдельным мемом в этом вопросе являются Венгерские Танцы. Гуглится по запросу “sorting algorithms hungarian folk dance”. Там тонны хореографического фана!

На этом, мне кажется, тема простых сортировок исчерпана. В следующем тред по алгоритмам речь пойдёт про Merge Sort, а после него будет тред про Quick Sort. На этих алгоритмах я постараюсь залезть в материю поглубже.

Надеюсь, что получилось разобрать тему и никого не запутать. Как всегда, постараюсь ответить на все возникшие вопросы!

[@threadreaderapp](#) unroll

...