

# Сортировка

**11/02/2023**

# Проблема

Неупорядоченные данные - отстой

- У неупорядоченного массива нет никаких свойств!
- Поэтому, например, в нем сложнее удалить дубликаты
- И бинарный поиск сделать не получится :(

# Поиск элемента в массиве

Мы уже обсуждали линейный поиск, который перебирает элементы по одному в поисках заданного

- Докажите, что если у нас нет гарантий на порядок элементов в массиве, то нельзя сделать поиск элемента быстрее, чем за  $n$  сравнений
- Почему нельзя, как в бинарном поиске, за  $O(\log n)$  сравнений?
- А почему нельзя методом исключения, за  $n - 1$  сравнение?

# Отсортированность

Отсортированность дает знание, что если  $i < j$ , то  $a_i \leq a_j$ .

Можно обобщить на  $a_i \geq a_j$ ,  $a_i^2 \leq a_j^2$  и кучу других функций с помощью *компараторов*

```
from functools import cmp_to_key

def compare(x, y):
    return y - x

data = [4, 3, 1, 2]
```

```
>>> sorted(data, key=cmp_to_key(compare))
[4, 3, 2, 1]
```

<https://docs.python.org/3/library/functions.html#sorted>

# Проверка

- Правда ли, что сортировка чисел по свойству  $a_i \leq a_j$  совпадает с сортировкой по свойству  $a_i^2 \leq a_j^2$ ?
- А в каких случаях совпадает?

# Неявная отсортированность

Можем ли мы считать, что массив `profits`, сгенерированный из данных с банковского терминала, отсортирован?

```
while not terminal.has_stopped():  
    for transaction in terminal.get_new_transactions():  
        profits.append(transaction.value)
```

# Неявная отсортированность

Можем ли мы считать, что массив `profits`, сгенерированный из данных с банковского терминала, отсортирован?

```
while not terminal.has_stopped():  
    for transaction in terminal.get_new_transactions():  
        profits.append(transaction.value)
```

Данные по прибыли (в случае разумной реализации `terminal.get_new_transactions`) неявно отсортированы по времени, а это значит, что в массиве есть неявные свойства (например, в начале дня средний чек может быть меньше чем в конце дня)

Вывод для реальной жизни: иногда индекс элемента в массиве является достаточно важным для того, чтобы не менять порядок элементов в массиве.

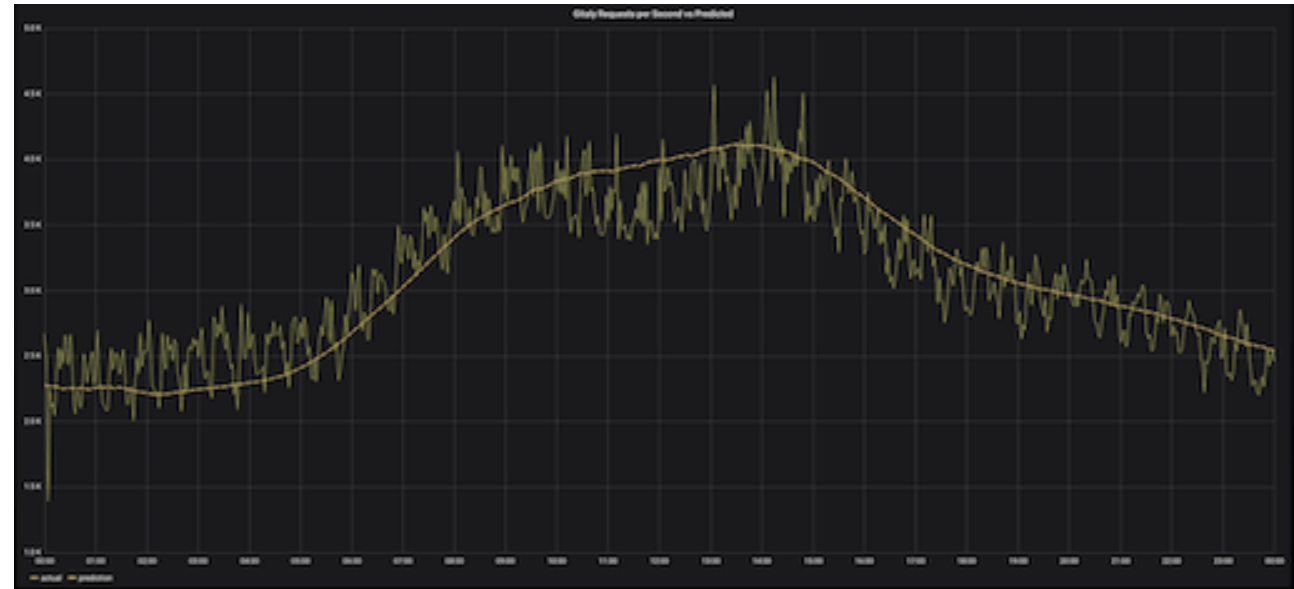
## Проверка

Придумайте еще какой-нибудь пример из жизни, где данные в массиве нельзя просто так перемешать без потери информации



# Оффтоп

Вот, например, типичный график нагрузки на веб-сервис. Если сохранять метрики последовательно в массив, но этот массив тоже будет неявно отсортирован по времени. Более того, люди специально изучают, как выглядят подобные массивы, чтобы предсказывать нагрузку и делать гибкие облачные решения.



# Сортировка по нескольким параметрам

Как отсортировать пары  $(0, 1)$ ,  $(1, 0)$ ,  $(1, 1)$  по возрастанию?

Правда ли, что  $(1, 1)$  всегда окажется последней парой?

# Сортировка по нескольким параметрам

В дискретной математике такие семейства объектов называются *частично упорядоченными множествами*, где некоторые элементы попросту несравнимы

В примере выше можно доопределить правила сравнения до покоординатного порядка:

```
def compare(a, b):  
    if a[0] != b[0]:  
        return a[0] - b[0]  
    return a[1] - b[1]
```

# Почему в компараторах $a - b$ ?

Well, *плохой* дизайн.

As any other compare function, returns a negative, or a positive value, or 0, depending on whether  $a < b$ ,  $a > b$  or  $a = b$ .

Рекомендую посмотреть примеры вот отсюда

<https://docs.python.org/3/howto/sorting.html#sortinghowto>

В некоторых случаях можно свести свой компаратор к стандартному

```
key=lambda x: -x
```

# Еще один оффтоп

Примером дизайна получше будет  
возвращать какой-нибудь `enum`

Enum `std::cmp::Ordering` 

1.0.0 · [source](#) · [-]

```
pub enum Ordering {  
    Less,  
    Equal,  
    Greater,  
}
```

[?] An `Ordering` is the result of a comparison between two values.

## Examples

```
use std::cmp::Ordering;  
  
let result = 1.cmp(&2);  
assert_eq!(Ordering::Less, result);  
  
let result = 1.cmp(&1);  
assert_eq!(Ordering::Equal, result);  
  
let result = 2.cmp(&1);  
assert_eq!(Ordering::Greater, result);
```

# Проверка

А что будет, если компаратор такой?

```
def cmp(a, b):  
    if random.randint(0, 1) == 0:  
        return -1  
    else:  
        return 1
```

**Где алгоритмы-то?**

# Алгоритмы сортировки

Сигнатура

```
def sorted(a: List[T]): -> List[T]
```

$$\forall_{i,j,i < j} \text{sorted}(a)_i \leq \text{sorted}(a)_j$$

Ну, то есть, массив отсортирован :)

Свойства:

- Стабильность - если *not*  $a_i <_* a_j$  and *not*  $a_j <_* a_i$  and  $i < j$ , то в итоговом массиве элемент  $i$  раньше элемента  $j$ .
- In-place - алгоритм не выделил  $O(n)$  дополнительной памяти, просто сделал `swap` много раз.

Стабильность, как правило, лучше нестабильности, а in-place, как правило, лучше not in-place.



# Проверка

Зачем нужно свойство стабильности?

# Квадратичные сортировки

Много разных видов, я не знаю точно ни одной, у них какие-то страшные названия:

- Пузырьком
- Выбором
- Вставками
- etc

## Квадратичная сортировка на swap

Достаточно сделать  $\frac{n(n-1)}{2}$  swap-ов соседних элементов, чтобы получить отсортированный массив. (доказать можно через тот факт, что количество инверсий  $i, j, i < j : a_i > a_j$  уменьшается).

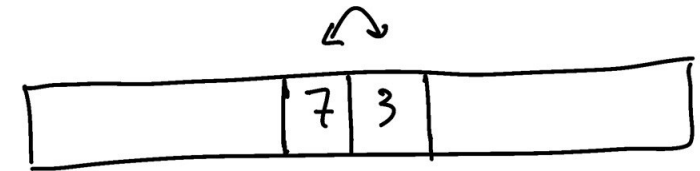
То есть, если  $a_i > a_{i+1}$ , то эти соседние элементы можно смело менять местами!

Если сделать такое  $O(n^2)$  раз, можно получить алгоритм сортировки

# Изменение инверсий

$$|(a_i > a_j)_{i < j}|$$

| $i/j$   | $< i$ | $i$ | $i + 1$ | $> i$ |
|---------|-------|-----|---------|-------|
| $< i$   | 0     | 0   | 0       | 0     |
| $i$     | x     | x   | -1      | 0     |
| $i + 1$ | x     | x   | x       | 0     |
| $> i$   | x     | x   | x       | 0     |



# Квадратичная сортировка

```
for _ in range(n):  
    for i in range(n - 1):  
        if a[i] > a[i + 1]:  
            a[i], a[i + 1] = a[i + 1], a[i]
```

Проверка:

- Где будет максимальный элемент после первой итерации внешнего цикла?
- Это in-place сортировка?
- Это стабильная сортировка?

## Ускоряем в два раза

```
for iteration in range(n):  
    for i in range(n - iteration - 1):  
        if a[i] > a[i + 1]:  
            a[i], a[i + 1] = a[i + 1], a[i]
```

## И ускорим average-case

```
for iteration in range(n):  
    had_swaps = False  
    for i in range(n - iteration - 1):  
        if a[i] > a[i + 1]:  
            a[i], a[i + 1] = a[i + 1], a[i]  
            had_swaps = True  
    if not had_swaps:  
        break
```

worst-case остался такой же

## Остальные квадратичные сортировки

Либо не in-place, либо делают что-то похожее, меняя порядок обхода в цикле.

Мне кажется, их бесполезно учить, можно названия для собеседований запомнить. В остальном это примерно одно и то же.



**Быстрая сортировка!**

**~~Быстрая сортировка!~~**

**Сортировка слиянием**

Ну реально проще объяснить, че поделать

# Сортировка слиянием

```
def sorted(a):  
    return merge(sorted(a[0:n/2]), sorted(a[n/2:n]))
```

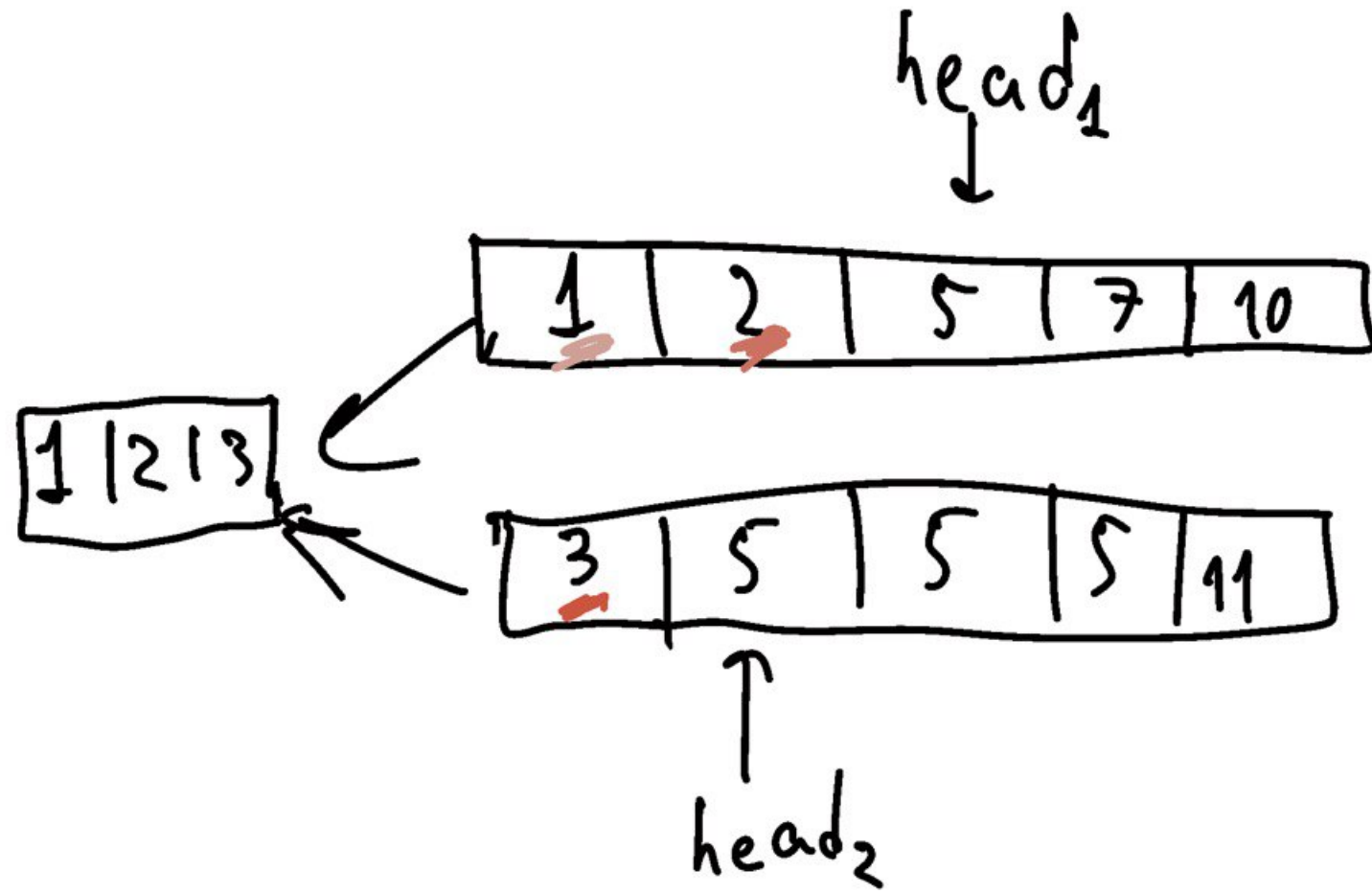
Это называется методом "Разделяй-и-властвуй" - разобьем массив пополам, решим задачу рекурсивно, а потом объединим результат, чтобы решить задачу на целом массиве

# Слияние

Как собрать из двух сортированных массивов один отсортированный массив?

Ну, первым элементом точно будет минимум из первых элементов, а дальше определим рекурсивно

```
def merge(a, b):  
    if a[0] < b[0]:  
        return [a[0]] + merge(a[1:], b)  
    else:  
        return [b[0]] + merge(a, b[1:])
```








# Избавляемся от рекурсии

```
def merge(a, b):  
    ptr_a, ptr_b = 0, 0  
    res = []  
    while ptr_a + ptr_b < len(a) + len(b):  
        if a[ptr_a] < b[ptr_b]:  
            res += a[ptr_a]  
            ptr_a += 1  
        else:  
            res += b[ptr_b]  
            ptr_b += 1  
    return res
```

# Асимптотика

$$O(n \log n)$$

Кстати, быстрее нельзя!

|   | len      | $\Sigma \text{len}$ | depth    |
|---|----------|---------------------|----------|
|  | $n$      | $n$                 | 0        |
|  | $n/2$    | $n$                 | 1        |
|  | $n/4$    | $n$                 | 2        |
|  | $\vdots$ | $\vdots$            | $\vdots$ |
|  | 1        | $n$                 | $\log n$ |

# Асимптотика

$$O(n \log n)$$

~~Кстати, быстрее нельзя!~~

Ну, почти нельзя.



# Почему нельзя сортировать быстрее, чем $O(n \log n)$ ?

Ну, потому что  $\log(n!) \sim n \log n$

<https://stackoverflow.com/a/2095472>

# Так почему иногда можно сортировать быстрее?

Потому что сортировка не всегда должна использовать попарные сравнения - иногда мы можем находить информацию неявно.

# Сортировка подсчетом

```
cnt = [0] * C
for elem in a:
    cnt[elem] += 1
res = []
for i in range(C):
    res.extend([i] * cnt[i])
```

Работает за  $O(n + C)$ , если  $\forall_i a_i < C$

Например: вы упорядочиваете школьников по классу (или возрасту! или росту!). Все случаи, где  $C < n \log n$  валидны. В таком случае их проще сгруппировать.

# Быстрая сортировка: ликбез

Сортировка слиянием делала работу на обратном ходе рекурсии - сливала два уже отсортированных массива.

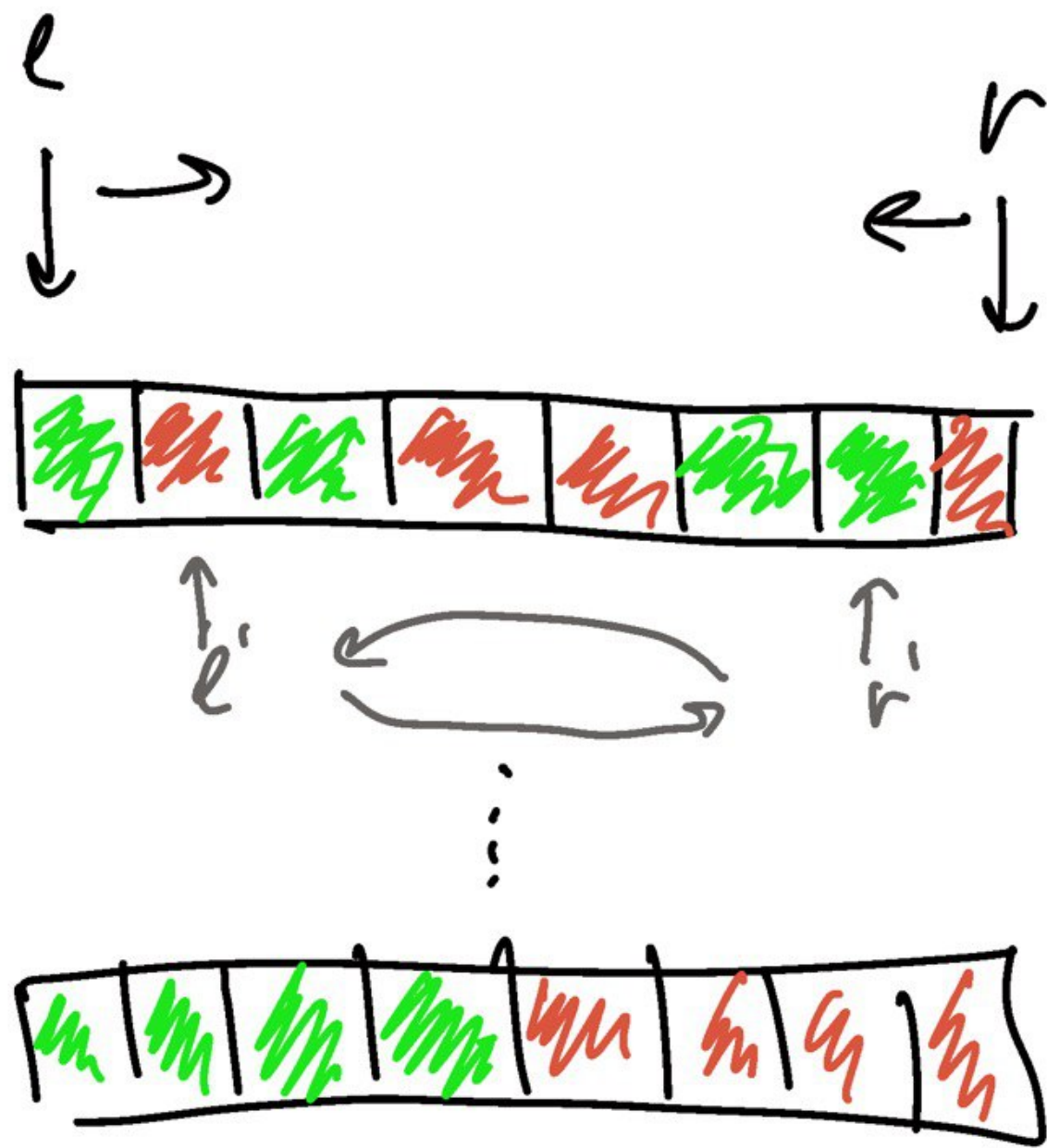
Сейчас мы хотим сделать наоборот - сначала разбить массив на две равные части так, чтобы слева все элементы были меньше, чем справа. Это действие называется *partition*: мы выбираем элемент и делаем partition относительно него

Рекурсивные *partition* отсортируют массив.

```
def quicksort(arr, l, r):  
    if l + 1 == r:  
        return  
    l1, r1, l2, r2 = partition(arr, l, r)  
    quicksort(arr, l1, r1)  
    quicksort(arr, l2, r2)
```

# Partition

```
def partition(arr, l, r):  
    index = l + r / 2 # 🤔  
    val = arr[index]  
    remember = (l, r)  
    while l <= r:  
        if arr[l] < val:  
            l += 1  
            continue  
        if arr[r] >= val:  
            r -= 1  
            continue  
        # arr[l] >= val, arr[r] < val  
        arr[l], arr[r] = arr[r], arr[l]  
        l += 1  
        r -= 1  
  
    return remember[0], r, l, remember[1]
```



# Выбор элемента для partition

partition не дает гарантий на размеры после разделения

А значит, мы можем не угадать с разделением и разделить в пропорции  $1 : n - 1$

Чтобы матожидание времени работы было  $O(n \log n)$ , в качестве разделяющего значения берут случайный элемент.

## Проверка

Если quick sort имеет worst-time  $O(n^2)$ , а merge sort  $O(n \log n)$ , то почему же быстрая сортировка быстрая?

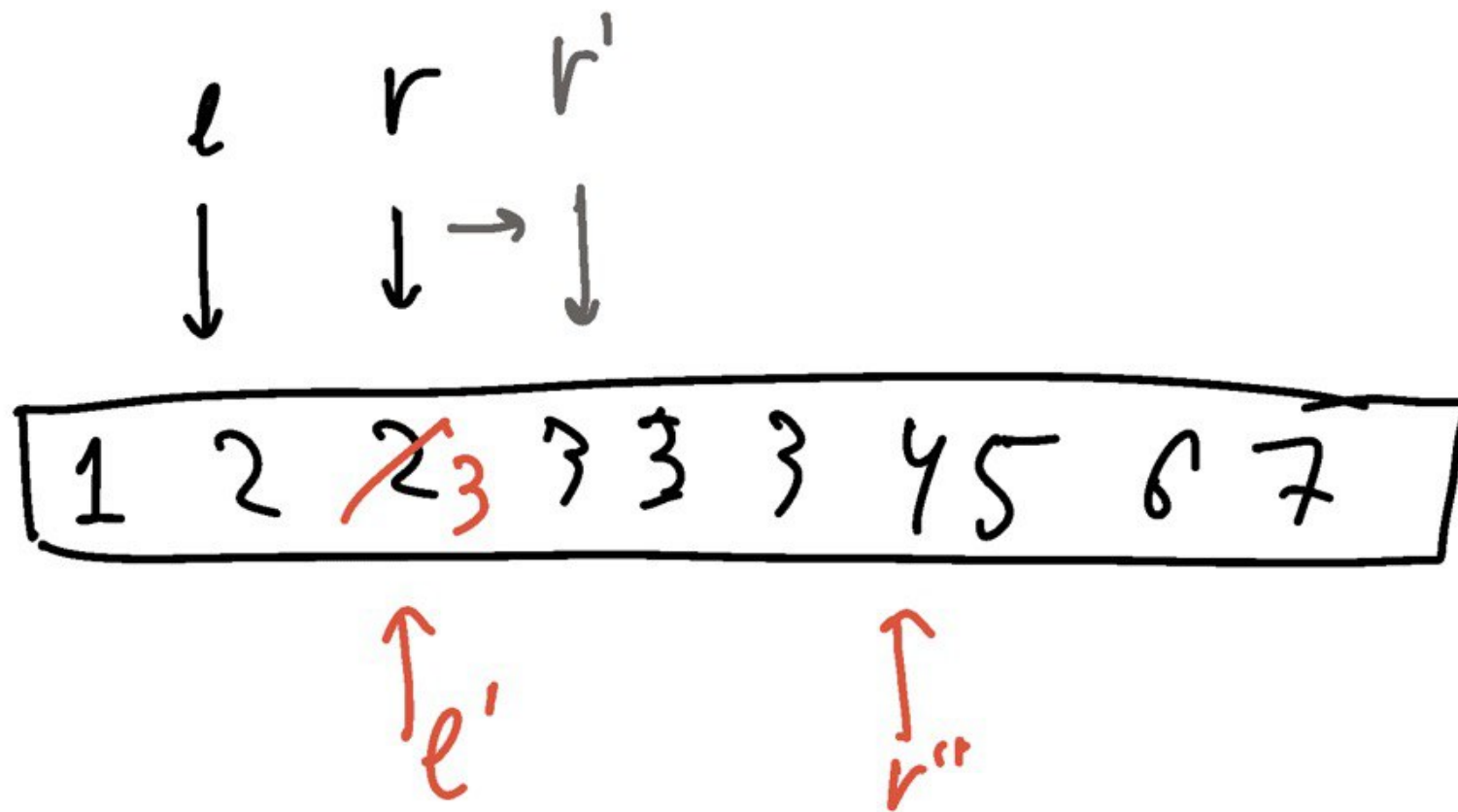


# Удаление дубликатов за $O(n)$

Если массив отсортирован, то в нем можно удалить все дубликаты

```
def unique(arr):  
    l = 0  
    r = 0  
    while r < len(arr):  
        while r < len(arr) and arr[r] == arr[l]:  
            r += 1  
        l += 1  
        arr[l] = arr[r]  
        r += 1  
    return arr[:l]
```

Почему это  $O(n)$ ?



# Мемы

xkcd

## INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):  
  IF LENGTH(LIST) < 2:  
    RETURN LIST  
  PIVOT = INT(LENGTH(LIST) / 2)  
  A = HALFHEARTEDMERGESORT(LIST[:PIVOT])  
  B = HALFHEARTEDMERGESORT(LIST[PIVOT:])  
  // UMMMMM  
  RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):  
  // AN OPTIMIZED BOGOSORT  
  // RUNS IN O(N LOG N)  
  FOR N FROM 1 TO LOG(LENGTH(LIST)):  
    SHUFFLE(LIST):  
    IF ISSORTED(LIST):  
      RETURN LIST  
  RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):  
  OK SO YOU CHOOSE A PIVOT  
  THEN DIVIDE THE LIST IN HALF  
  FOR EACH HALF:  
    CHECK TO SEE IF IT'S SORTED  
    NO, WAIT, IT DOESN'T MATTER  
    COMPARE EACH ELEMENT TO THE PIVOT  
    THE BIGGER ONES GO IN A NEW LIST  
    THE EQUAL ONES GO INTO, UH  
    THE SECOND LIST FROM BEFORE  
  HANG ON, LET ME NAME THE LISTS  
  THIS IS LIST A  
  THE NEW ONE IS LIST B  
  PUT THE BIG ONES INTO LIST B  
  NOW TAKE THE SECOND LIST  
  CALL IT LIST, UH, A2  
  WHICH ONE WAS THE PIVOT IN?  
  SCRATCH ALL THAT  
  IT JUST RECURSIVELY CALLS ITSELF  
  UNTIL BOTH LISTS ARE EMPTY  
  RIGHT?  
  NOT EMPTY, BUT YOU KNOW WHAT I MEAN  
  AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):  
  IF ISSORTED(LIST):  
    RETURN LIST  
  FOR N FROM 1 TO 10000:  
    PIVOT = RANDOM(0, LENGTH(LIST))  
    LIST = LIST[PIVOT:] + LIST[:PIVOT]  
    IF ISSORTED(LIST):  
      RETURN LIST  
  IF ISSORTED(LIST):  
    RETURN LIST  
  IF ISSORTED(LIST): // THIS CAN'T BE HAPPENING  
    RETURN LIST  
  IF ISSORTED(LIST): // COME ON COME ON  
    RETURN LIST  
  // OH JEEZ  
  // I'M GONNA BE IN SO MUCH TROUBLE  
  LIST = [ ]  
  SYSTEM("SHUTDOWN -H +5")  
  SYSTEM("RM -RF ./")  
  SYSTEM("RM -RF ~/*")  
  SYSTEM("RM -RF /")  
  SYSTEM("RD /S /Q C:\*") // PORTABILITY  
  RETURN [1, 2, 3, 4, 5]
```

## Идея для пет-проекта

Ничего лучше не придумано, чем визуализации сортировок

<https://www.sortvisualizer.com/mergesort/>

## Для дальнейшего изучения

- Сортировка кучей
- Частично упорядоченные множества
- Random shuffle
- <https://www.geeksforgeeks.org/internal-details-of-stdsort-in-c/>
- <https://www.geeksforgeeks.org/iterative-quick-sort/>