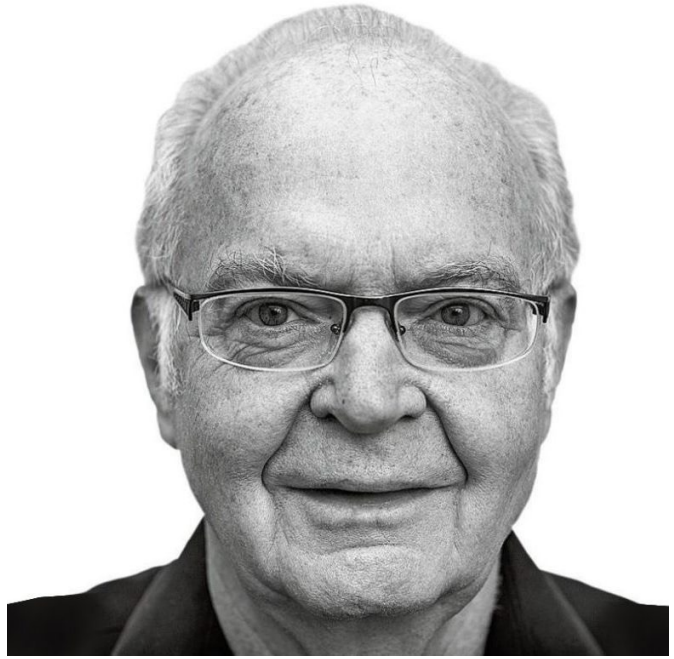
# Лекция 2. Сортировки

Горденко М.К.

# Задача сортировки

- Сортировка – это упорядочивание набора однотипных данных по возрастанию или убыванию.

# Критерии оценки алгоритмов сортировки

- Время сортировки
- Требуемая память
- Устойчивость

# Сортировка выбором (Selection sort)

- Сортировка выбором является одним из простейших алгоритмов сортировки
- Может быть как устойчивой, так и не устойчивой
- Шаги алгоритма:
  - находим минимальное значение в текущей части массива;
  - производим обмен этого значения со значением на первой неотсортированной позиции;
  - далее сортируем хвост массива, исключив из рассмотрения уже отсортированные элементы

# Сортировка выбором (Selection sort)

# Сортировка выбором (Selection sort)

```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n-1):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        if min_index != i:
            arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr
```

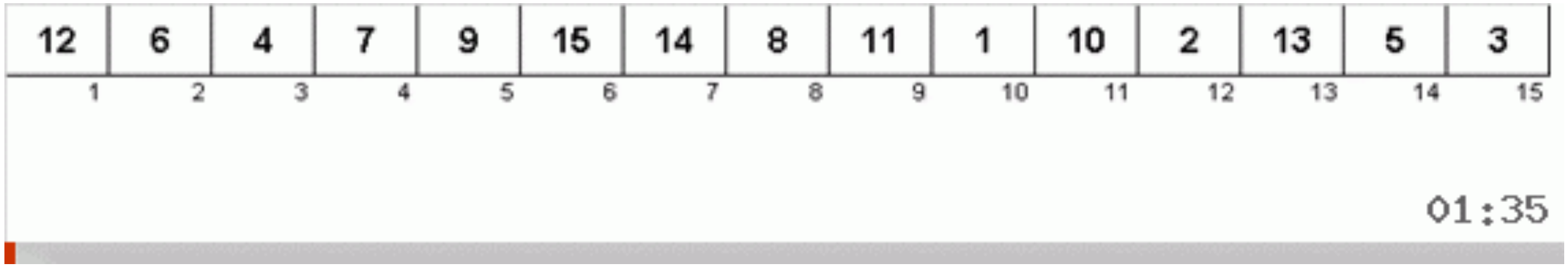
# Сортировка выбором (Selection sort)

- Асимптотическая сложность - $O(n^2)$
- Время работы в худшем и среднем случае - $O(n^2)$
- Дополнительная память - $O(1)$

# Устойчивость

- Устойчива или нет?

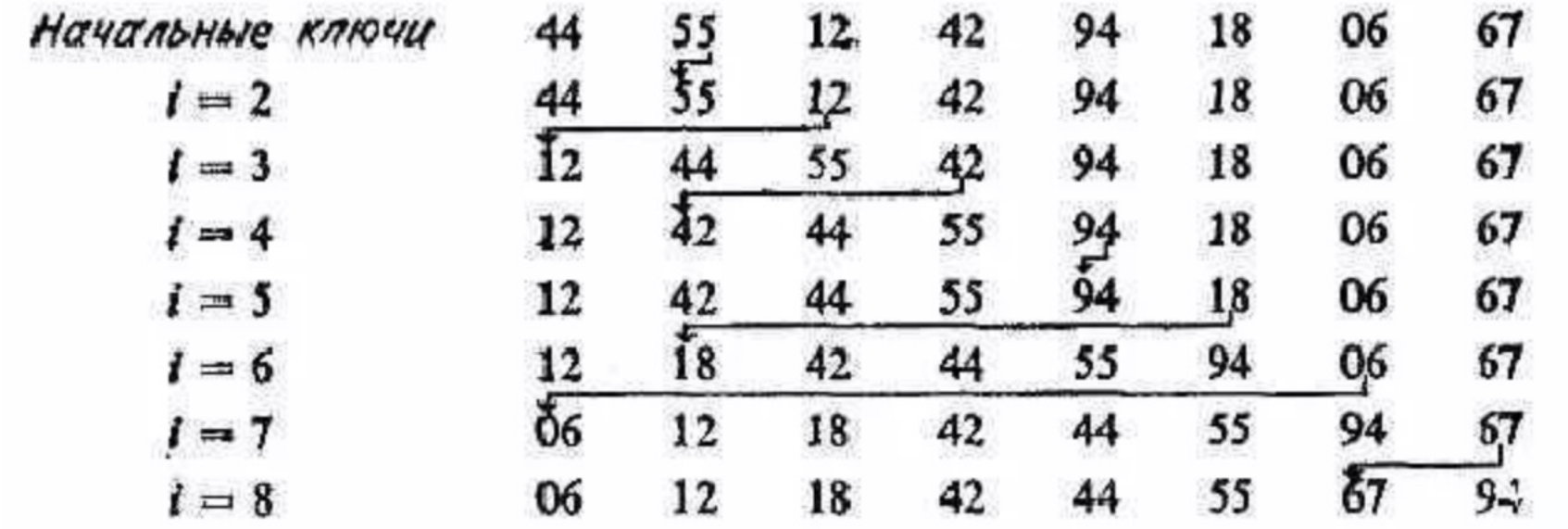
# Пример




https://habr.com/ru/post/422085/

# Сортировка вставками (Insertion sort)

- Сортировка вставками — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов



# Сортировка вставками (Insertion sort)

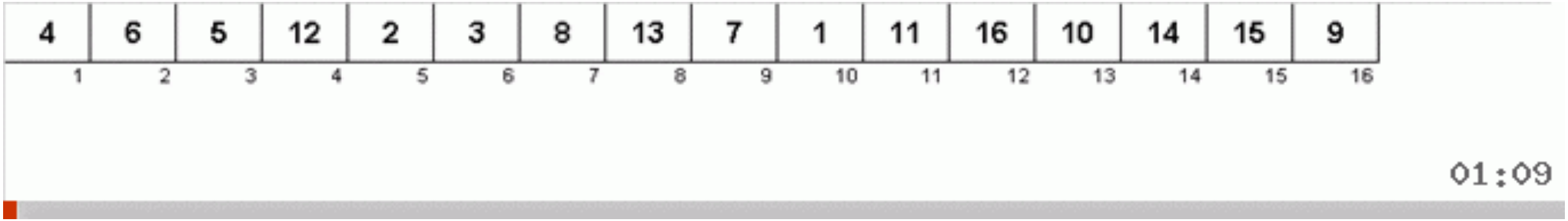
- Время работы в худшем и среднем случае - $O(n^2)$
- Время работы в лучшем случае - $O(n)$
- Дополнительная память - $O(1)$
- Устойчивость?

# Сортировка вставками (Insertion sort)

- Время работы в худшем и среднем случае - $O(n^2)$
- Время работы в лучшем случае - $O(n)$
- Дополнительная память - $O(1)$
- Устойчивость?

```
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        current_value = arr[i]
        j = i - 1
        while j >= 0:
            if current_value < arr[j]:
                arr[j+1] = arr[j]
                arr[j] = current_value
                j = j - 1
            else:
                break
    return arr
```

# Еще пример




https://habr.com/ru/post/415935/

# Сортировка пузырьком (bubble sort)

- Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются N-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции как пузырёк в воде, отсюда и название алгоритма).

# Сортировка пузырьком (bubble sort)

# Сортировка пузырьком

- Асимптотическая сложность - $O(n^2)$
- Время работы в худшем и среднем случае - $O(n^2)$
- Дополнительная память - $O(1)$

# Сортировка пузырьком

- Асимптотическая сложность - $O(n^2)$
- Время работы в худшем и среднем случае - $O(n^2)$
- Дополнительная память - $O(1)$

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

# Пример

| *i* | *j* | | *1* | *2* | *3* | *4* | *5* | *6* |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | 4 | 5 | 9 | 1 | 3 | 6 |
| | 2 | | 4 | 5 | 9 | 1 | 3 | 6 |
| | 3 | обмен | 4 | 5 | 9 | 1 | 3 | 6 |
| | 4 | обмен | 4 | 5 | 1 | 9 | 3 | 6 |
| | 5 | обмен | 4 | 5 | 1 | 3 | 9 | 6 |
| 2 | 1 | | 4 | 5 | 1 | 3 | 6 | **9** |
| | 2 | обмен | 4 | 5 | 1 | 3 | 6 | **9** |
| | 3 | обмен | 4 | 1 | 5 | 3 | 6 | **9** |
| | 4 | | 4 | 1 | 3 | 5 | 6 | **9** |
| 3 | 1 | обмен | 4 | 1 | 3 | 5 | **6** | **9** |
| | 2 | обмен | 1 | 4 | 3 | 5 | **6** | **9** |
| | 3 | | 1 | 3 | 4 | 5 | **6** | **9** |
| 4 | 1 | | 1 | 3 | 4 | **5** | **6** | **9** |
| | 2 | | 1 | 3 | 4 | **5** | **6** | **9** |
| 5 | 1 | | 1 | 3 | **4** | **5** | **6** | **9** |

# Сортировка подсчетом (Counting sort)

- Алгоритм сортировки, в котором используется диапазон чисел сортируемого массива (списка) для подсчёта совпадающих элементов
- Применение сортировки подсчётом целесообразно лишь тогда, когда сортируемые числа имеют (или их можно отобразить в) диапазон возможных значений, который достаточно мал по сравнению с сортируемым множеством
- Есть устойчивый и неустойчивый вариант сортировки

# Сортировка подсчетом (Counting sort)

```
def counting_sort(alist, largest):
    c = [0]*(largest + 1)
    for i in range(len(alist)):
        c[alist[i]] = c[alist[i]] + 1

    # Find the last index for each element
    c[0] = c[0] - 1 # to decrement each element for zero-based indexing
    for i in range(1, largest + 1):
        c[i] = c[i] + c[i - 1]

    result = [None]*len(alist)

    # Though it is not required here,
    # it becomes necessary to reverse the list
    # when this function needs to be a stable sort
    for x in reversed(alist):
        result[c[x]] = x
        c[x] = c[x] - 1

    return result
```

# Достоинства и недостатки

- Асимптотическая сложность - $O(n + k)$
- Сортировка выполняется только для целых чисел
- Целесообразно применять, когда диапазон чисел небольшой по сравнению с размером сортируемого массива
- Требуется дополнительная память - $O(k)$ в случае неустойчивой сортировки и $O(n + k)$ в случае устойчивой

# Рекурсивная функция

- Базовый случай
- Шаг рекурсии

- Количество вызовов подпрограммы самой себя называется глубиной рекурсии или глубиной рекурсивных вызовов.

# Прямая и косвенная рекурсия

- В случае прямой рекурсии вызов функцией самой себя делается непосредственно в этой же функции.
- Косвенная рекурсия создаётся за счёт вызова данной функции из какой-либо другой функции, которая сама вызывалась из данной функции.

# Вычисление факториала

```
def fac(n):
    if n == 0:
        return 1
    return fac(n-1) * n


print(fac(5))
```

# Достоинства и недостатки рекурсии

+ Для составления функции используется постановка задачи, поэтому ряд задач легче запрограммировать в виде рекурсии
- Расходы памяти
- Медленная работа при большой глубине

# Быстрая сортировка

- Алгоритм сортировки, разработанный английским информатиком Тони Хоаром во время его работы в МГУ в 1960 году.
- Один из самых быстрых известных универсальных алгоритмов сортировки массивов.
- Алгоритм основан на принципе «разделяй и властвуй».

# Быстрая сортировка. Алгоритм

1) В исходном несортированном массиве некоторым образом выбирается разделительный элемент  x (барьерный элемент, опорный элемент, pivot).

2) Массив разбивается на две части. Элементы массива переставляются таким образом, чтобы

- в левой части массива оказались элементы <=x,
- в правой – элементы массива, большие или равные  >=x.

В итоге все элементы левой части меньше любого элементов правой части, за исключением элементов, равных барьерному (они могут быть как слева, так и справа).

3) Рекурсивно обрабатываются левый и правый подмассивы

# Быстрая сортировка. Алгоритм

```python
import random

def quick_sort(arr):
    n = len(arr)
    if n <= 1:
        return arr
    else:
        pivot = random.choice(arr)
        less = [x for x in arr if x < pivot]
        greater_or_equal = [x for x in arr if x >= pivot]
        return quick_sort(less) + quick_sort(greater_or_equal)
```

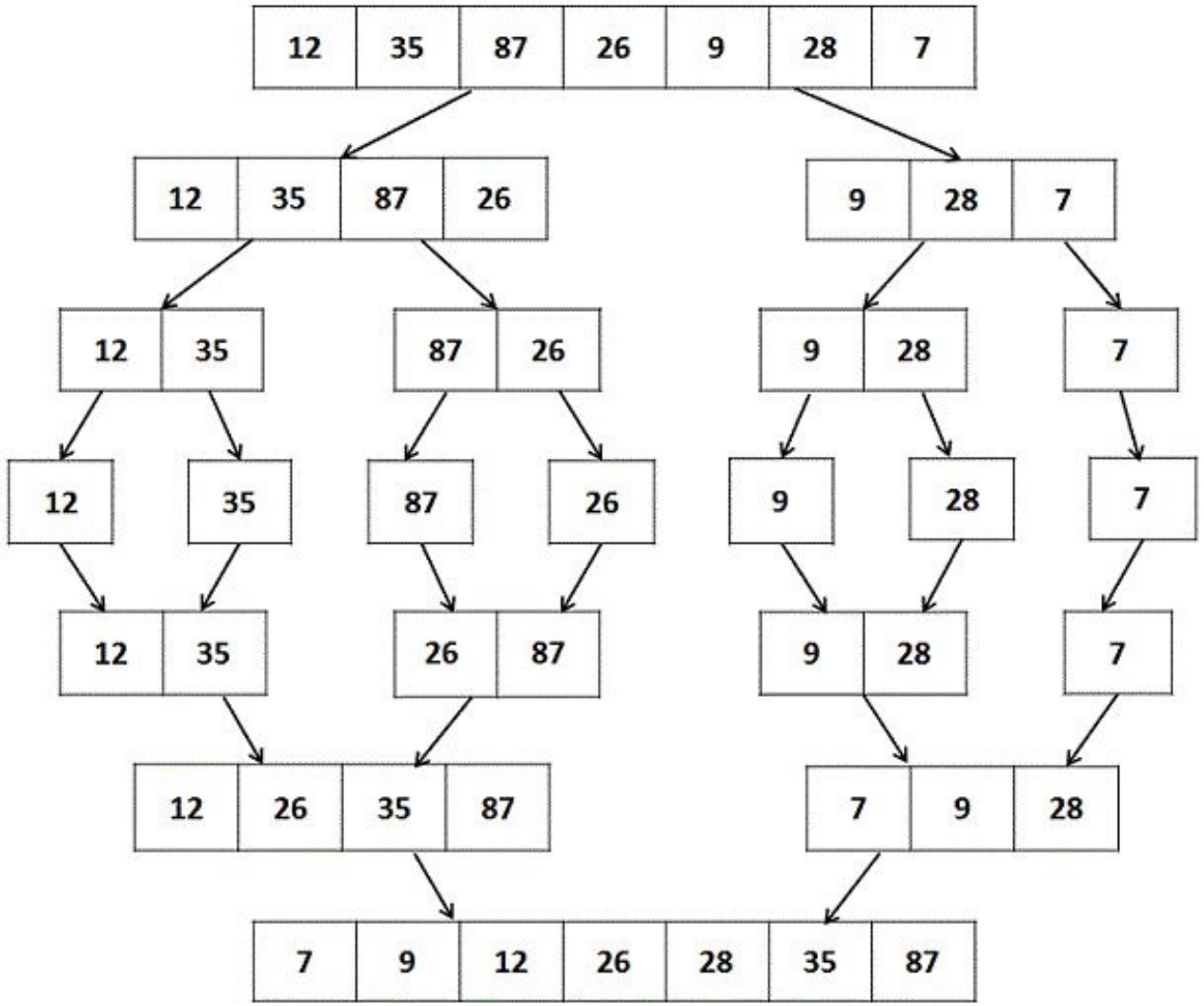
# Разделительный элемент

1. Первый (последний) элемент рассматриваемой части массива (разбиение Ломуто).
2. Второй (предпоследний) элемент рассматриваемой части массива.
3. Элемент, находящийся в середине рассматриваемой части массива.
4. Среднее арифметическое всех элементов рассматриваемой части массива.
5. Среднее арифметическое из трех элементов в начале, в конце и в середине рассматриваемой части массива.
6. Медиана трех элементов в начале, в конце и в середине рассматриваемой части массива.
7. Медиана подмассива.
8. Случайным образом.

# Достоинства и недостатки

- Прост в реализации
- Возможно распараллеливание
- Один из самых быстрых на практике
- Асимптотическая сложность - $O(n \log n)$ в среднем случае
- Деградация до квадратичной сложности в худшем случае
- Рекурсивная реализация может привести к переполнению стека
- Не устойчивая

# Сортировка слиянием



# Алгоритм

- Сортируемый массив разбивается на две части примерно одинакового размера;
- Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;
- Два упорядоченных массива половинного размера соединяются в один.

```
mergeSort(A, l, r)
 if (l < r) then
 m = l + (r - l) / 2;
 mergeSort(A, l, m)
 mergeSort(A, m+1, r);
 merge(A, l, m, r)
```

# Алгоритм

```python
def merge_sort(arr):
    n = len(arr)
    if n <= 1:
        return arr
    else:
        middle = int(len(arr) / 2)
        left = merge_sort(arr[:middle])
        right = merge_sort(arr[middle:])
        return merge(left, right)

def merge(left, right):
    result = []
    while len(left) > 0 and len(right) > 0:
        if left[0] <= right[0]:
            result.append(left[0])
            left = left[1:]
        else:
            result.append(right[0])
            right = right[1:]
    if len(left) > 0:
        result += left
    if len(right) > 0:
        result += right
    return result
```