Форма 2: разбор

1:

> На лекции мы обсуждали, как сделать собственный компаратор для сортировки.

> def compare(a, b): return a - b

> Такой компаратор сравнивает два элемента в массиве, а значит, сортировка с таким компаратором не может работать быстрее, чем O(n log n) Представьте себе, что вы пишете сортировку подсчетом и хотите подготовить для нее собственный "компаратор". Кратко опишите, что такой компаратор должен принимать, что возвращать, и на каком этапе сортировки он будет вызываться. Если есть возможность, приведите пример того, как компаратор будет реализован в случае, если мы сортируем студентов (класс Student) по номеру группы (поле group).

```
count_sorted(arr, key=lambda student: student.group)
```

Функция из параметра `key` принимает элемент массива, а возвращает какое-то число от 0 до C - максимального значения для сортировки. Тогда внутри сортировки можно добавлять элемент в соответствующую группу через вызов `key(elem)`

```
# тут задаем параметр по умолчанию
def count_sorted(arr, key=lambda x: x)
    cnt = [0] * C
    for elem in arr:
        cnt[key(elem)] += 1
    # ...
```

Много кто не понял вопрос и предложил новую реализацию сортировки подсчетом, которая явно берет группу студента. Мы такое оценим в половину баллов.

Проблема такого подхода в том, что ваша сортировка будет пригодна только для одного типа данных. Если вместо студентов нужны будут школьники, придется писать новую. А весь смысл компараторов в том, чтобы сортировка имела одну реализацию, и мы ее "тюнинговали" компаратором для конкретных данных.

Вопрос, собственно, предлагал про это подумать и адаптировать подход для сортировки подсчетом.

2:

> Предположим, вы решили вопрос 1 и научились сортировать студентов методом подсчета. А как сделать такую сортировку стабильной?

В прошлом вопросе есть еще один нерешенный вопрос - если сортировка подсчетом сортируются числа, то как сортировать произвольные объекты?

Ну, например, для каждого значения хранить список с элементами

```
# тут задаем параметр по умолчанию
def count_sorted(arr, key=lambda x: x)
    # ...
    for elem in arr:
        blocks[key(elem)].append(elem)
    # ...
    res = []
    for block in blocks:
        for elem in block:
            res.append(elem)
```

Тогда в каждой группе будут все объекты, соответствующие блоку. Теперь можно склеить все массивы в один ответ.

Кроме того, если сохранить порядок элементов из массивов в cnt при склеивании, более ранний элемент в рамках группы окажется в ответе раньше, чем более поздний. Это и есть условие стабильности.

3:

> В сортировке слиянием для оценки асимптотики мы использовали идею, что если T(n) = 2T(n/2) + O(n), то T(n) = O(n log n).

> А если бы мы делили массив на 3 части, рекурсивно сортировали бы каждую, а затем делали слияние трех массивов? (такое слияние, кстати, можно сделать за O(n), проверьте, что вы понимаете, почему). Какое бы тогда время работы получилось? Сравните эту асимптотику с нижней оценкой на сложность алгоритма сортировки O(n log n). Подсказка-напоминалка: O(c * n log n) = O(n log n)

Для сортировки с разбиением на 3 блока мы получим асимптотику $T(n) = 3T(\frac{n}{3}) + O(n)$. По аналогии с сортировкой подсчета, можно разбить сортировку на слои и увидеть, что слоев $\log_3(n)$, а на каждом слое суммарно все работает за линейное время. Тогда сложность получается $O(n \log_3(n))$.

Почему же мы не делаем такую сортировку, раз $\log_3(n) < \log_2(n)$? Потому что $\frac{\log_3(n)}{\log_2(n)} = \log_2(3)$, а это какая-то константа. То есть разница может быть, но небольшая. Кроме этого, слияние трех массивов аналогичным образом требует $O(n)$, но константа больше, потому что надо сравнивать три элемента из сливаемых массивов, чтобы добавить минимум в итоговый массив.