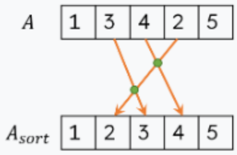
Рассмотрим механизм подсчета «степени упорядоченности» некоторого целочисленного массива **A = [a_1, a_2, a_3, ..., a_n]**, заполненного *уникальными* числами.

Элементы a_i и a_j массива **A** назовем *переставленными*, если $i < j$, но $a_i > a_j$. Например, в массиве **A = [1, 3, 4, 2, 5]** необходимо выполнить две перестановки, а именно **3 → 2** и **4 → 2** (см. на рисунке ниже), чтобы получить отсортированный массив **A = [1, 2, 3, 4, 5]**.



1. **(4 балла)** Разработайте DaC-алгоритм, сложность которого соответствует **O(n · log n)**, для подсчета степени упорядоченности массива путем вычисления количества необходимых перестановок. Описание алгоритма представьте в любом удобном формате. Опишите суть шагов **DIVIDE**, **CONQUER** и **COMBINE**, а также представьте рекуррентное соотношение для ***T(n)*** и обоснуйте соответствие требуемой сложности.
2. **(2 балла)** Элементы a_i и a_j массива **A** назовем *значительно переставленными*, если $i < j$, но $a_i > 2 \cdot a_j$. Какие изменения необходимо внести в алгоритм, разработанный на предыдущем шаге с тем, чтобы в качестве степени упорядоченности велся подсчет количества пар значительно переставленных элементов? Например, в массиве **A = [1, 3, 4, 2, 5]** нет значительно переставленных элементов, а в массиве **A = [5, 3, 2, 4, 1]** всего 4 пары значительно переставленных элементов: **5 → 1**, **5 → 2**, **4 → 1** и **3 → 1**. *Сложность измененного алгоритма должна остаться неизменной.*

Пункт 1

Назовём *беспорядком* пару элементов массива a_i и a_j, таких что $i < j$ и $a_i > a_j$,
т.е. *беспорядок* - это пара *переставленных* элементов массива.
Опишем алгоритм нахождения количества *беспорядков* во входном массиве A на интервале $[l; r]$ (то есть на подмассиве $A[l; r]$):
DIVIDE Рекурсивно найдём количество *беспорядков* на интервалах $[l; m]$ и $[m + 1; r]$, где $m = \left\lfloor \frac{l+r}{2} \right\rfloor$.
CONQUER Тогда для получения ответа осталось найти количество *беспорядков*, таких, что первый элемент пары имеет индекс от l до m включительно, а второй элемент пары имеет индекс от $m + 1$ до r включительно. Это верно, так как количество всех остальных возможных *беспорядоков*, оба индекса элементов которых принадлежат либо $[l; m]$, либо $[m + 1; r]$, было найдено рекурсивно.
COMBINE Тогда ответом будет сумма количества найденных *беспорядков* и количеств *беспорядков*, найденных рекурсивно.

В условии задачи не запрещено использовать дополнительную память.
Приведём пример алгоритма, решающего поставленную задачу и использующего $\mathcal{O}(n)$ дополнительной памяти.
Модифицируем сортировку слиянием (Merge sort) так, чтобы при слиянии подмассивов $[l; m]$ и $[m + 1; r]$ подсчитывалось количество *беспорядков* из шага **CONQUER**, а функция сортировки подмассива $A[l; r]$ возвращала количество *беспорядков* на интервале $[l; r]$.
Т.к. разрабатываемый DaC-алгоритм не должен изменять входной массив, то для работы функции сортировки потребуется создать копию массива, т.е. будет использовано $\mathcal{O}(n)$ дополнительной памяти. В таком случае, для удобства (и понятности написанного кода) будем использовать "неэффективную"реализацию сортировки слиянием, требующую $\mathcal{O}(n)$ дополнительной памяти. В таком случае, расход дополнительной памяти останется равным $\mathcal{O}(n)$.

Пусть на каком-то шаге сортировки необходимо слить (merge) подмассивы $A[l; m]$ и $A[m + 1; r]$, индексом l_1 обозначим передвигаемый индекс в подмассиве $A[l; m]$, а индексом l_2 обозначим передвигаемый индекс в подмассиве $A[m + 1; r]$.
Если в какой-то момент $A[l_1] > A[l_2]$, то и любой элемент в подмассиве $A[l_1 + 1; m]$ больше элемента $A[l_2]$. В таком случае ответ нужно увеличить на $m - l_1 + 1$. При этом мы учтём все возможные *беспорядоки*, т.к. индекс l_2 пройдёт по всем элементам правого подмассива, а левый индекс l_1 будет двигаться вправо, пока $A[l_1] \leq A[l_2]$.

Пример реализации алгоритма на языке программирования C++
(функция size_t countPermutations(const std::vector<int64_t>&))
Описанное выше обновление ответа происходит на строке с номером 36

```cpp
8   using std::vector;
9
10  size_t slowMergeSortWithCounting(vector<int64_t>& array, size_t l, size_t r) {
11      switch(r - l) {
12          case 0:
13              return 0;
14          case 1:
15              size_t current_count = array[l] > array[l + 1];
16              if (current_count) {
17                  std::swap(array[l], array[l + 1]);
18              }
19              return current_count;
20      }
21      size_t m = (l + r) / 2;
22      size_t left_count = slowMergeSortWithCounting(array, l, m);
23      size_t right_count = slowMergeSortWithCounting(array, m + 1, r);
24      size_t current_count = 0;
25      vector<int64_t> tmp(r - l + 1);
26      size_t l1 = l;
27      size_t l2 = m + 1;
28      size_t l3 = 0;
29      while (l1 <= m && l2 <= r) {
30          if (array[l1] <= array[l2]) {
31              tmp[l3] = array[l1];
32              l1++;
33          }
34          else {
35              tmp[l3] = array[l2];
36              current_count += m - l1 + 1;
37              l2++;
38          }
39          l3++;
40      }
41      while (l1 <= m) {
42          tmp[l3] = array[l1];
43          l1++;
44          l3++;
45      }
46      while (l2 <= r) {
47          tmp[l3] = array[l2];
48          l2++;
49          l3++;
50      }
51      memcpy(&array[l], &tmp[0], tmp.size() * sizeof(tmp[0]));
52      return left_count + right_count + current_count;
53  }
54
55  size_t countPermutations(const vector<int64_t>& array) {
56      if (array.empty()) {
57          return 0;
58      }
59
60      vector<int64_t> copy(array);
61      return slowMergeSortWithCounting(copy, 0, array.size() - 1);
62  }
```

Функция временной сложности $T_1(n)$ алгоритма slowMergeSortWithCounting описывается рекуррентным соотношением $T_1(n) = 2 \cdot T_1(\frac{n}{2}) + \Theta(n) \implies$ по master-теореме $T_1(n) = \Theta(n \log n)$
Функция временной сложности $T(n)$ искомого алгоритма countPermutations равна $T(n) = T_1(n) + \Theta(n) \implies T(n) = \Theta(n \log n)$.

Пункт 2

Модифицируем функцию slowMergeSortWithCounting так, чтобы количество
беспорядков шага **CONQUER** *current_count* увеличивлось только для
элементов a_i и a_j, таких что $a_i > 2 * a_j$.
Для этого вынесем подсчёт *current_count* в отдельный цикл, в котором
также будет два индекса: l_1 для левого подмассива и l_2 для правого подмассива.
В цикле по l_2 от $m+1$ до r индекс l_1 будет увеличиваться во внутреннем цикле,
пока $l_1 \leq m \land A[l_1] \leq 2 * A[l_2]$. После остановки внутреннего цикла обновляется 1
current_count на величину $m - l_1 + 1$. Случай, когда цикл завершился из-за $l_1 > m$
будет обработан автоматически, т.к. в таком случае $l_1 = m + 1 \implies m - l_1 + 1 = 0$
Таким образом на каждой итерации добавляется дополнительная работа $\Theta(n)$
(т.к. каждый из индексов l_1 и l_2 пройдут значения от l до m и от $m+1$ до r) $\implies$
У модифицированного алгоритма $T_2(n) = 2 \cdot T_2(\frac{n}{2}) + \Theta(n) + \Theta(n) = 2 \cdot T_2(\frac{n}{2}) + \Theta(n) \implies$
$\implies T_2(n) = \Theta(n \log n)$

Пример реализации алгоритма на языке программирования C++ (на следующей странице)

```cpp
using std::vector;

size_t slowMergeSortWithCounting(vector<int64_t>& array, size_t l, size_t r) {
    switch(r - l) {
        case 0:
            return 0;
        case 1:
            size_t current_count = array[l] > 2 * array[l + 1];
            if (array[l] > array[l + 1]) {
                std::swap(array[l], array[l + 1]);
            }
            return current_count;
    }
    size_t m = (l + r) / 2;
    size_t left_count = slowMergeSortWithCounting(array, l, m);
    size_t right_count = slowMergeSortWithCounting(array, m + 1, r);

    size_t current_count = 0;
    for (size_t l1 = l, l2 = m + 1; l1 <= m && l2 <= r;  l2++) {
        while (l1 <= m && array[l1] <= 2 * array[l2]) {
            l1++;
        }
        current_count += m - l1 + 1;
    }

    vector<int64_t> tmp(r - l + 1);
    size_t l1 = l;
    size_t l2 = m + 1;
    size_t l3 = 0;
    while (l1 <= m && l2 <= r) {
        if (array[l1] <= array[l2]) {
            tmp[l3] = array[l1];
            l1++;
        }
        else {
            tmp[l3] = array[l2];
            l2++;
        }
        l3++;
    }
    while (l1 <= m) {
        tmp[l3] = array[l1];
        l1++;
        l3++;
    }
    while (l2 <= r) {
        tmp[l3] = array[l2];
        l2++;
        l3++;
    }
    memcpy(&array[l], &tmp[0], tmp.size() * sizeof(tmp[0]));
    return left_count + right_count + current_count;
}

size_t countPermutations(const vector<int64_t>& array) {
    if (array.empty()) {
        return 0;
    }

    vector<int64_t> copy(array);
    return slowMergeSortWithCounting(copy, 0, array.size() - 1);
}
```