АиСД_дз3

CovarianceMomentum

1term

1

Утверждается, что на любой перестановке, представляющей собой один цикл, сортировка выбором сделает максимальное число обменов. Покажем это по индукции:

База: для перестановок из двух элементов очевидно.

Переход: Так как в нашеё перестановке больше одного элемента, то первый элемент стоит *не* на своём месте. Значит, на этой итерации мы совершим обмен. Поэтому, если перестановка является одним большим циклом, то на этой итерации мы совершим обмен, а оставшаяся часть перестновки останется одним большим циклом, в котором, по индукционному предположению, сортировка выбором совершает максимальное число обменов.

Покажем, что на перестановках, не являющихся большими циклами, сортировка выбором совершает меньшее число обменов. От противного — пусть есть такая перестановка, в которой не только большой цикл, которая совершает столько же обменов. Значит, она должна совершить обмен на этой итерации и после него по предположению индукции должен остаться один большой цикл. Раз мы совершили обмен, то пройдём в элемент, который должен был стоять на месте первого и обойдём этот цикл до конца. В любом случае, этот цикл не может захватывать всю перестановку, иначе бы изначальная перестановка тоже была одним большим циклом. Значит, он захватывает не все элементы, но так как в этом цикле не меньше двух вершин, то после обмена получится не один только большой цикл, а ещё какой-то мусор. Доказано.

Теперь посчитаем количество таких перестановок, что они сотавляют один большой цикл. Посмотрим на первую позицию. На неё можно поставить любой элемент, кроме первого, то есть $(n-1)$. Посмотрим, куда указывает этот элемент. На это место нельзя ставить никакой из ранее поставленных элементов, равно как и элемент, указывающий на первую позицию, то есть всего $(n-2)$, затем $(n-3)$ и так далее, до последнего элемента, который будет указывать на первую позицию. Получаем $(n-1)!$ перестановок.

2

Покажем, что в любой перестановке, где 1-ый элемент стоит на последнем месте, сортировка пузырьком совершит все итерации. Заметим, что на каждой итерации 1 сдвигается только на одну позицию, так как не существует элементов меньше неё, а значит за одну итерацию её можно подвинуть только один раз. Таким образом показали, что если единица стоит на последней позиции, то пузырёк делает максимальное число итераций.

Теперь докажем, что другие перестановки завершаются за меньшее число итераций. Пусть 1 стоит на позиции x. Тогда через $x-1$ итерацию первый элемент встанет на своё место, а суффикс массива длиной $x-1$ будет отсортирован. В оставшейся и, возможно, неотсортированной части массива будет $n-x$ элементов, которые будут отсортированы за максимум $n-x-1$ итерацию пузырька. Таким образом, если только $x \neq 0$, пузырёк отсортирует массив за не больше чем $(x-1)+(n-x-1) = x-2$ итераций пузырька. Победа.

Также поймём, что интересующих нас перестановок всего $(n-1)!$, так как помимо единицы на последнем месте мы можем разместить элементы в любом порядке.

3

Вспомним, как мы строили перестановки из задачи 1:

1. Ставим элемент $x_1 \neq 1$ на первую позицию;
2. Переходим в позицию x_1;
3. Ставим элемент $x_2 \neq x_1 \neq 1$ на позицию x_1;
4. Переходим в позицию x_2;
5. ...
6. Переходим в позицию x_n;
7. Ставим в неё 1.

Теперь вспомним, как мы строили перестановки из задачи 2:

1. Ставим элемент $x_1 \neq 1$ на первую позицию;
2. Переходим в позицию 2;
3. Ставим элемент $x_2 \neq x_1 \neq 1$ на вторую позицию;
4. Переходим в позицию 3;
5. ...
6. Переходим в позицию n;
7. Ставим в неё 1.

Схожесть нетрудно заметить. Теперь приведем способ перейти из какой-то определённой перестановки первого вида в перестановку второго обратно.

$1 \to 2$
Попросту выпишем все x_j в ряд и получим перестановку второго рода.

$2 \to 1$
Пойдём по нашей перестановке, создавая перестановку первого рода по алгоритму, каждый раз выбирая очередной член нашей последовательности как x_j, получим перестановку второго рода.

Биективная перестановка вычисляется за $\mathcal{O}(n)$ так как для её вычисления надо всего лишь раз пройтись по оригинальной перестановке.

4

Предлагается следующий алгоритм:

1. Разбиваем оригинальный массив на блоки по k элементов;
2. Сортируем подмассив, образованный первым и вторым блоками;
3. Сортируем подмассив, образованный вторым и третьим блоками;
4. ...
5. Сортируем подмассив, образованный предпоследним и последним блоками.

Докажем, что это работает. Инвариантом будет то, что элементы в предыдущих блоках уже стоят на своих местах. Тогда если мы сортируем новую пару блоков, то все элементы, которые должны были стоять в первом блоке этой пары точно содержатся в паре, так как они не могут быть в предыдущих блоках, потому как те уже отсортированы. Значит, эти элементы встанут на свои места — мы отсортировали ещё один блок, отсортированный префикс увеличился, переходим дальше.

Имеется $\frac{n}{k}$ блоков, каждую пару из которых мы сортируем за $\mathcal{O}(k \log k)$. Таким образом, итоговая асимптотика равна $2 \cdot \frac{n}{k} \cdot \mathcal{O}(k \log k) = \mathcal{O}(\frac{n}{k} \cdot k \log k) = \mathcal{O}(n \log k)$.

5

Первым шагом алгоритма будет сортировка массива p. Тут есть два варианта — уложиться в указанную в задаче асимптотику $\mathcal{O}(n \log m + m)$, но скушать дополнительные $\mathcal{O}(n)$ памяти для того, чтобы использовать count sort (так как элементы массивы p не больше n), либо отсортировать его при помощи quick sort, но не кушать память. Мне нравится второй, но от этого принципиально ничего не зависит. Поэтому будем использовать первый.

Итак, мы имеем отсортированный за $\mathcal{O}(m)$ массив p. Предложим следующий алгоритм:

1. Рассмотрим средний элемент массива p, то есть $p[\frac{m}{2}]$. Найдём его порядковую статистику за $\mathcal{O}(n)$.
2. За все те же $\mathcal{O}(n)$ сделаем partition по этому элементу. Тогда $\forall i < \frac{m}{2}$ порядковая статистика $p[i]$ будет лежать в левой части массива после partition, а для $\forall j > \frac{m}{2}$ порядковая статистика $p[j]$ будет лежать в правой части. Рекурсивно вызовемся от левой части и нижней половины массива p и от правой части и верхней половины массива p.

Очевидно, что вышеописанный алгоритм находит все порядковые статистики. Теперь докажем его асимптотику. Сортировка массива p проходит отдельно за $\mathcal{O}(m)$. Теперь решим рекурренту. Так как я не умею решать её по-человечески, решим почти графически. Поймём, что глубина рекурсии будет равна $\log m$, так как мы уменьшаем массив p в два раза для каждого следующего шага, а на каждом её уровне алгоритм совершает $\mathcal{O}(n)$ действий — поиск средней порядковой статистики в каждом массиве и новый partition. Таким образом, эта часть алгоритма работает за $\mathcal{O}(n \log m)$, суммарная асимптотика $\mathcal{O}(n \log m + m)$.

6

Алгоритм для массива $[a;\ b]$ будет таков:

1. Выбрать пару разделительных элементов l и r в массиве;
2. Посчитать количество отрезков с суммой, не превышающей k, таких, что они содержат оба разделительных элемента;
3. Запуститься рекурсивно от массивов $[a;\ l]$ и $[r;\ b]$, добавить результат на них к общей сумме;
4. Вернуть ответ.

Для доказательства алгоритма поймём, что при наличии разделительных элементов все отрезки можно поделить на три типа:

- Захватывающие левый;
- Захватываюшие правый;
- Захватывающие оба.

Как несложно заметить, подсчёт первых двух типов происходит в рекурсивных вызовах, а третьего — на втором шаге. Покажем, как это делается за линейное время при помощи метода двух указателей:

```
        countInBothParts(div, partSize):
            int leftmost = div - 1, rightmost = div
            int sum = arr[div - 1] + arr[div]
            int retval = 0
            while sum <= k and leftmost >= div - partSize:
                leftmost -= 1
                sum += arr[leftmost]
            while sum <= k and rightmost < div + partSize:
                rightmost += 1
                sum += arr[rightmost]
            while leftmost < div:
                retval += rightmost - div + 1
                sum -= arr[leftmost]
                leftmost += 1
                while sum <= k and rightmost < div + partSize:
                    rightmost += 1
                    sum += arr[rightmost]
            return retval
```

Приведенный код для каждой возможной левой границы самую дальнюю возможную правую границу отрезка, затем добавляет все меньшие отрезки к ответу, а затем переходит к следующей левой границе, проходя все возможные из них.

Теперь найдём асимптотику этого алгоритма. Два указателя работают за $\mathcal{O}(n)$, и, брать за разделительные элементы центральные, то суммарная асимптотика будет $T(n) = 2T(\frac{n}{2}) + n = \mathcal{O}(n \log n)$.

Но это плохой алгоритм, который вообще можно было бы исполнить за линию. А ещё он работает только для неотрицательных элементов массива. Теперь решим нормально.

Заменим массив на массив префиксных сумм, это делается за $\mathcal{O}(n)$. Запустим на этом массиве merge sort со следующей модификацией: помимо возврата отсортированных половинок массива алгоритм будет возвращать также и количество подходящих нам отрезков, полностью лежащих внутри этих половинок. Покажем, как можно сделать это за $\mathcal{O}(n)$, если мы имеем две отсортированных половинки массива:

1. Рассмотрим самый большой элемент левого массива l_n;
2. До тех пор, пока $l_n - r_i \leq k$, где r_i — элемент правого массива под номером i, начиная с 1, увеличиваем i. Останавливаемся на последнем элементе, для которого это верно (или вообще на самом последнем, если для него тоже верно);
3. Заметим, что все отрезки, концам которых соответствуют суммы l_n и r_j, где $j \leq i$, подходят нам. Их будет в точности i штук, прибавим это значение к результату на этом отрезке;
4. Перейдём к следующему по величине элементу левого массива l_{n-1};
5. Будем уменьшать i, до тех пор пока неравенство $l_{n-1} - r_i \leq k$ снова не выполнится (возможно, оно сразу будет выполняться);
6. Опять же, нам подходят все отрезки, концам которых соответствуют суммы l_{n-1} и r_j, где $j \leq i$. Их будет i штук, прибавим это значение к результату на отрезке.
7. Повторим пункты 4-6 до тех пор, пока не дойдём до l_1.

При помощи вышеописанного применения метода двух указателей на массиве мы посчитаем количество отрезков, подходящих нам и проходящих через центр. Рекурсивные вызовы от обеих половинок же уже посчитали количество подходящих отрезков, не проходящих через центр. Победа.