Условие.

Уже давно прошли те времена, когда Петя Булочкин писал сортировку вставками и интересовался вопросом, сколько сравнений выполнит его алгоритм, сортируя заданную перестановку натуральных чисел от 1 до n. Теперь Петя уже не тот глупенький мальчик, которым он был два года назад. Теперь он знает, что число этих сравнений будет равно числу инверсий в перестановке. К слову сказать, инверсией в перестановке $p_1, p_2, \ldots, p_n$ называется пара (i, j), для которой $i < j$ и $p_i > p_j$. Но, как уже говорилось, вопрос об инверсиях в перестановке сейчас мало волнует Петю. Теперь он придумал новую фишку, которая не даёт ему покоя. Петя ввел новый математический термин. Он назвал мегаинверсией в перестановке $p_1, p_2, \ldots, p_n$ тройку (i, j, k), для которой $i < j < k$ и $p_i > p_j > p_k$. Сейчас Петя ходит и ломает голову, придумывая алгоритм для быстрого подсчёта числа мегаинверсий в перестановке. Так как Петя очень долго уже думает над этой задачей, и никаких дельных мыслей в голову ему пока не пришло, то ему кажется, что для решения этой задачи вообще не существует быстрого алгоритма. Докажите Пете, что он неправ.

Необходимо написать программу, которая вводит число n и перестановку натуральных чисел от 1 до n, находит число мегаинверсий в перестановке, выводит результат.

Формат входных данных

Первая строка содержит целое число n ($1 \le n \le 300\,000$). Следующие n строк описывают перестановку: i-я из этих строк содержит ровно одно целое число p_i ($1 \le p_i \le n$; все p_i попарно различны).

Формат выходных данных

Выведите число мегаинверсий в перестановке $p_1, p_2, \ldots, p_n$.

Пример

input.txt	output.txt
4 4 3 2 1	4

Решение.

Мегаинверсия - это такая тройка, в которой индексы перестановки идут по возрастанию, а сами элементы по убыванию ($i < j < k$ и $p_i > p_j > p_k$). Я заметила, что если для определенного элемента посчитать сначала количество элементов, больших, чем рассматриваемый, и идущих перед ним, а потом - количество элементов, которые меньше, но следуют после, то, перемножив полученные значения, получим количество троек, для которых данный элемент стоит в середине.

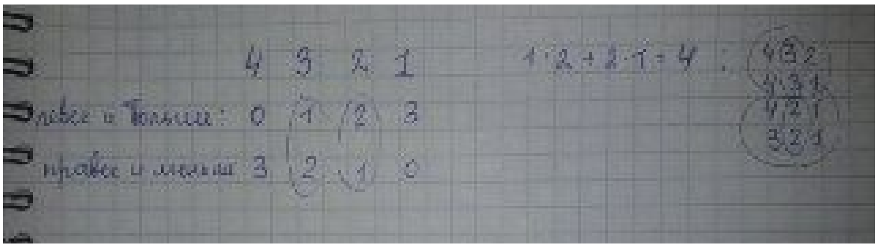

Рис.1. Алгоритм на примере.

Для реализации алгоритма я использовала такую структуру данных, как дерево Фенвика. (Узнала о ней, думая над общей задачей о сумме, которую, к сожалению, так и не удалось правильно решить, что досадно). Дерево Фенвика позволяет выполнять операции запроса модификации (в моем случае “инкремент”) и суммы за $O(\log n)$. Пусть имеется некоторый массив $A = [a_1, a_2, \dots, a_{n-1}]$, тогда для T − дерева Фенвика − i-ый элемент будет представлять собой частичную сумму элементов этого массива, то есть $T_i = \sum_{k=F(i)}^{i} a_k$, где $F(i)$ − некоторая функция, определенная в зависимости от требуемой операции. Так для суммы этой функцией является $F(i) = i \,\&\, (i+1)$,а для инкремента - $F(i) = i \mid (i+1)$. (О том, почему функции задаются именно так, лучше расскажет университет ИТМО https://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%A4%D0%B5%D0%BD%D0%B2%D0%B8%D0%BA%D0%B0).

На изображении ниже можно увидеть, что я реализовала дерево Фенвика на динамическом массиве. Насчет функции “*upperSize*” могу сказать, что я вначале не была уверена в размере, который понадобится, поэтому приблизительно высчитала, что должно сработать умножение на 2 (и, конечно, обычное умножение заменила на побитовый сдвиг, который дает некоторый выигрыш во времени!).

```cpp
int upperSize(int n) {
    int upper = 1;
    while (upper < n)
        upper <<= 1;
    return upper;
}
struct Struct {
    vector<int> vec;
    Struct(int n) : vec(upperSize(n)+1, 0) {}
    int sum(int r) {
        int res = 0;
        while (r >= 0) {
            res += vec[r];
            r = (r & (r + 1)) - 1;
        }
        return res;
    }
    void incElement(int i) {
        while (i < vec.size()) {
            vec[i]++;
            i = (i | (i + 1));
        }
    }
};
```

Рис.2. Структура, реализующая дерево Фенвика.

Создаем целых четыре объекта структуры "*Struct*". После инициализации массива "*a*" в цикле *for* начинаем заполнять "*inversions*" с помощью метода "*incElement*", "*inversions[0]*" полагаем равным нулю. Таким образом, мы увеличиваем элемент "*inversions[a[i]]*" на единицу на каждой итерации. Однако, чтобы вычислить количество бОльших элементов, но при этом стоящих перед *i*-ым, необходимо посчитать сумму тех элементов "*inversions*" , что стоят после "*a[i]*". А это есть не что иное, как разность всех элементов и суммы элементов до текущего.

Так как на каждой итерации элемент "*inversions[a[i]]*" увеличивается на единицу, то на *i*-ой итерации их сумма равна *i*. Получаем формулу *"bigger[i] = i - (inversions[a[1]] + inversions[a[2]] + … + inversions[a[i]])".*

Для вычисления меньших и стоящих правее элементов используем ту же тактику, но в этот раз цикл *for* запускаем с конца.

Сумма произведений соответствующих элементов "*bigger*" и "*smaller*" есть количество мегаинверсий в перестановке. (Умножила на "*1LL"* на случай, если количество цифр перешагнет допустимое для *int.*

```
int main() {
    int n;
    fin >> n;
    long long  res = 0;
    Struct a(n);
    Struct bigger(n);
    Struct inversions(n);
    Struct smaller(n);
    for (int i = 1; i <= n; ++i) {
        fin >> a.vec[i];
    }
    for (int i = 1; i <= n; i++) {
        inversions.incElement(a.vec[i]);
        bigger.vec[i] = i -inversions.sum(a.vec[i]);
    }
    for (int i = n; i >= 1; i--) {
        smaller.incElement(a.vec[i]);
        res += 1LL * bigger.vec[i] * smaller.sum(a.vec[i] - 1);
    }
    fout << res;
    return 0;
}
```

Рис.3. Реализация алгоритма подсчета числа мегаинверсий.

Асимптотика.

Как было описано ранее, дерево Фенвика выполняет операции за $O(log\ n)$, а так как операции вызываются для каждого из n элементов, то в худшем случае время работы алгоритма - $O(n \cdot log\ n)$.

P.S. Надеюсь, что написанное мной будет более понятно, нежели сказанное.
P.S.S. Хорошего дня. Не болейте:)