In a previous challenge you implemented the *Insertion Sort* algorithm. It is a simple sorting algorithm that works well with small or mostly sorted data. However, it takes a long time to sort large unsorted data. To see why, we will analyze its running time.

Running Time of Algorithms
The running time of an algorithm for a specific input depends on the number of operations executed. The greater the number of operations, the longer the running time of an algorithm. We usually want to know how many operations an algorithm will execute in proportion to the size of its input, which we will call N.

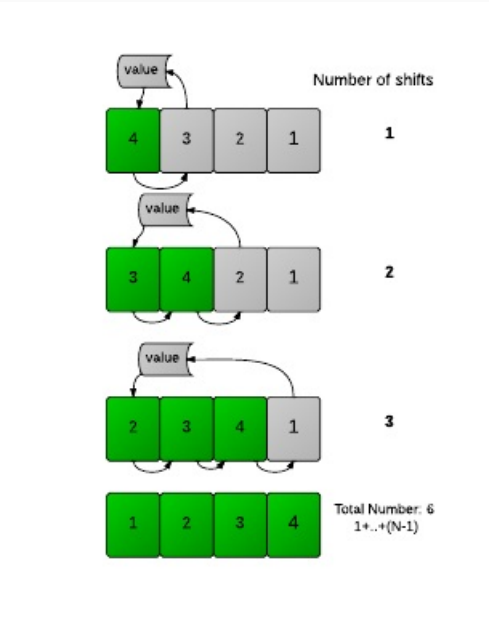
What is the ratio of the running time of Insertion Sort to the size of the input? To answer this question, we need to examine the algorithm.

Analysis of Insertion Sort
For each element V in an array of N numbers, Insertion Sort compares the number to those to its left until it reaches a lower value element or the start. At that point it shifts everything to the right up one and inserts V into the array.

How long does all that shifting take?

In the best case, where the array was already sorted, no element will need to be moved, so the algorithm will just run through the array once and return the sorted array. The running time would be directly proportional to the size of the input, so we can say it will take N time.

However, we usually focus on the worst-case running time (computer scientists are pretty pessimistic). The worst case for Insertion Sort occurs when the array is in reverse order. To insert each number, the algorithm will have to shift over that number to the beginning of the array. Sorting the entire array of N numbers will therefore take $1 + 2 + \ldots + (N - 1)$ operations, which is $N(N - 1)/2$ (almost $N^2/2$). Computer scientists just round that up (pick the dominant term) to N^2 and say that Insertion Sort is an "N^2 time" algorithm.



What this means
The running time of the algorithm against an array of N elements is N^2. For $2N$ elements, it will be $4N^2$. Insertion Sort can work well for small inputs or if you know the data is likely to be nearly sorted, like check numbers as they are received by a bank. The running time becomes unreasonable for larger inputs.

Challenge
Can you modify your previous Insertion Sort implementation to keep track of the number of shifts it makes while sorting? The only thing you should print is the number of shifts made by the algorithm to completely sort the array. A shift occurs when an element's position changes in the array. Do not shift an element if it is not necessary.

Function Description

Complete the *runningTime* function in the editor below. It should return an integer representing the number of shifts it will take to sort the given array.

runningTime has the following parameter(s):

- *arr*: an array of integers

Input Format

The first line contains the integer n, the number of elements to be sorted.
The next line contains n integers of $arr[arr[0] \ldots arr[n-1]]$.

Constraints

$1 \leq n \leq 1001$
$-10000 \leq a[i] \leq 10000, \text{ where } i \in \{0..(n-1)\}$

Output Format

Output the number of shifts it takes to sort the array.

Sample Input

```
5
2 1 3 1 2
```

Sample Output

```
4
```

Explanation

```
Iteration   Array      Shifts
0           2 1 3 1 2
1           1 2 3 1 2     1
2           1 2 3 1 2     0
3           1 1 2 3 2     2
4           1 1 2 2 3     1

Total                     4
```