Running Time of Algorithms

Problem Statement

In the previous challenges, you created an insertion sort algorithm. Insertion Sort 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.

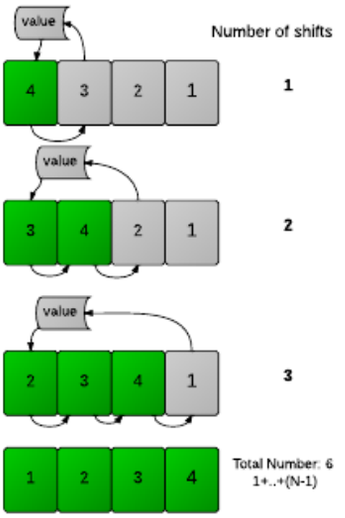
Running Time of Algorithms
The running time of an algorithm for a specific input depends on the number of operations executed. More the number of operations, the longer the running time of an algorithm. We usually want to know how many operations an algorithm will execute in proportion to the size of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

For example, the first two input files for InsertionSort were small, so it was able to sort them quickly. But the next two files were larger and sorting them took more time. What is the ratio of the running time of InsertionSort to the size of the Input? To answer this question, we need to examine the InsertionSort algorithm.

Analysis of Insertion Sort
For each element *V* in an array of *N* numbers, InsertionSort shifts everything to the right until it can insert *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+...+(N-1) operations, which is almost $N^2/2$. Computer scientists just round that up to N^2 and say that insertion sort is a "N^2 time" algorithm.



What this means
As the size of an input (N) increases, Insertion Sort's running time will increase by the square of N. So Insertion Sort can work well for small inputs, but becomes unreasonably slow for larger inputs. For large

inputs, people use sorting algorithms that have better running times, which we will examine later.

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 insertion sort 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.)

Input Format
The first line contains N, the number of elements to be sorted. The next line contains N integers a[1],a[2]...,a[N].

Output Format
Output the number of shifts it takes.

Constraints
1<=N<=1001
-10000 <= x <= 10000 , x ∈ a

Sample Input
5
2 1 3 1 2

Sample Output
4

Explanation
The first '1' is shifted once. The '3' stays where it is. The next '1' gets shifted twice. The final '2' gets shifted once. Hence, the total number of shifts is 4.

Task
For this problem, you can copy your code from the InsertionSort problem, and modify it to keep track of the number of shifts, instead of printing the array.