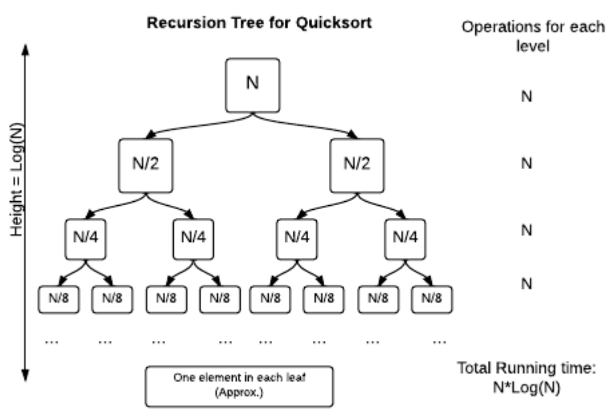
Running Time of Quicksort

by HackerRank

The running time of Quicksort will depend on how balanced the partitions are. If you are unlucky and select the greatest or the smallest element as the pivot, then each partition will separate only one element at a time, so the running time will be similar to Insertion Sort.

However, Quicksort will usually pick a pivot that is mid-range, and it will partition the array into two parts. Let's assume Partition is lucky and it always picks the median element as the pivot. What will be the running time in such a case?

Running Time of Recursive Methods
Quicksort is a recursive method, so we will have to use a technique to calculate the total running time of all the method calls. We can use a version of the "Recursion Tree Method" to estimate the running time for a given array of N elements.



Each time *partition* is called on a sub-array, each element in the sub-array needs to be compared to the pivot element. Since all the sub-arrays are passed to *partition*, there will be N total operations for each level of the tree.

How many levels will it take for the Quicksort to finish? Since we assume it always picks the middle element, the array will be split into two equal halves each time. So it will take $log(N)$ splits until we get single elements in the sub-arrays. Since there are $log(N)$ levels and each one involves N operations, the total running time for Quicksort will be $N \times log(N)$.

In real sorting, Quicksort won't always pick the exact middle element. But as long as it's regularly picking elements near the median value, it will have a running time better than Insertion Sort. To make sure that Quicksort works well on most inputs, the real-world implementations do not pick the same index as pivot each time. They use some other technique, e.g. picking a random element. There are other techniques, as well, that can be used to improve Quicksort. The Java Arrays class uses a modified version of Quicksort to sort primitives.

Notice that $O(N \times log(N))$ of Quicksort is much much faster than the $O(N^2)$ of Insertion Sort. For example, for an array of one million elements, $N^2 = 10^{12}$, while $N \times log(N)$ is approximately 2×10^7, a much more manageable number.

Challenge
In practice, how much faster is Quicksort (in-place) than Insertion Sort? Compare the running time of the two algorithms by counting how many swaps

or shifts each one takes to sort an array, and output the difference. You can modify your previous sorting code to keep track of the swaps. The number of swaps required by Quicksort to sort any given input have to be calculated. Keep in mind that the *last* element of a block is chosen as the pivot, and that the array is sorted in-place as demonstrated in the explanation below.

Any time a number is lower than the partition, it should be "swapped", even if it doesn't actually move to a different location. Also ensure that you count the swap when the pivot is moved into place. The count for Insertion Sort should be the same as the previous challenge, where you just count the number of "shifts".

Note
Please use Lomuto Partition for this challenge.

Input Format
There will be two lines of input:

- n - the size of the array
- ar - n numbers that make up the array

Output Format
Output one integer D, where $D =$ (insertion sort shifts) - (quicksort swaps)

Constraints
$1 \le n \le 1000$
$-1000 \le x \le 1000, x \in ar$

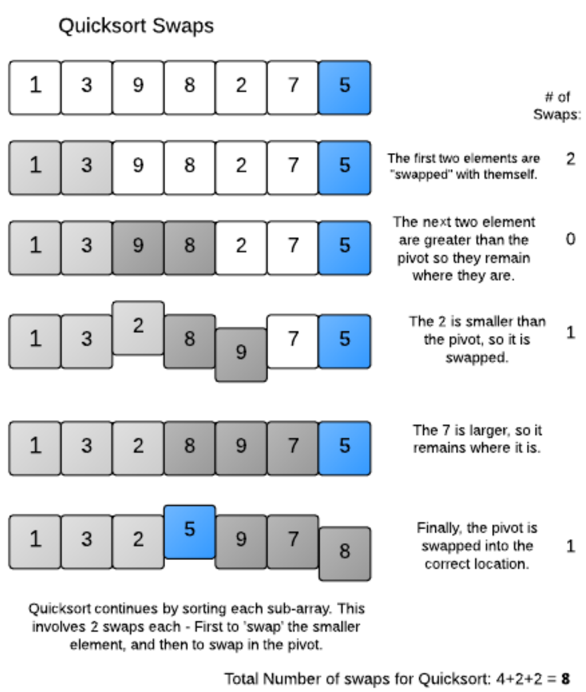
Sample Input

```
7
1 3 9 8 2 7 5
```

Sample Output

```
1
```

Explanation
Insertion Sort will take *9* "shifts" to sort the array. Quicksort will take *8* "swaps" to sort it, as shown in the diagram below. $9 - 8 = 1$ is the final output.



Submissions: 8940
Max Score: 35
Difficulty: Easy

Rate This Challenge:
Thanks!

More

Current Buffer (saved locally, editable)

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;


class Solution {
    static int swapInsertion = 0;
        static int swapQuick = 0;


        static int partition(int[] arr, int low, int high)
        {
            int pivot = arr[high];
            int temp;
            int i = (low - 1); // index of smaller element
            for (int j = low; j <= high - 1; j++)
            {
                // If current element is smaller than or
                // equal to pivot
                if (arr[j] <= pivot)
                {
                    i++;

                    // swap arr[i] and arr[j]
                    temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                    swapQuick++;
                }
            }

            // swap arr[i+1] and arr[high] (or pivot)
            temp = arr[i + 1];
            arr[i + 1] = arr[high];
            arr[high] = temp;
            swapQuick++;

            return i + 1;
        }


        static void quickSort(int[] array, int start, int end)
        {
            if (start < end)
            {
                int pivotIndex = partition(array, start, end);
                //printArray(array);
                quickSort(array, start, pivotIndex - 1);
                quickSort(array, pivotIndex + 1, end);
            }
        }

        static void insertionSort(int[] ar)
        {
            for (int i = 1; i < ar.Length; i++)
            {
                int indice = i;
```

```
                while (indice > 0 && ar[indice - 1] > ar[indice])
                {
                    int temp = ar[indice - 1];
                    ar[indice - 1] = ar[indice];
                    ar[indice] = temp;
                    swapInsertion++;
                    indice--;
                }
            }
        }


        static void Main(string[] args)
        {
            int n = int.Parse(Console.ReadLine());
            int[] arrQuick = Array.ConvertAll(Console.ReadLine().Split(' '), e => int.Parse(e));
            int[] arrInsertion = new int[n];
            Array.Copy(arrQuick, arrInsertion, n);

            quickSort(arrQuick, 0, arrQuick.Length - 1);
            insertionSort(arrInsertion);

            Console.WriteLine(swapInsertion - swapQuick);

            Console.ReadLine();
        }
}
```

Line: 6 Col: 1

Upload Code as File

Test against custom input

Run Code

Submit Code

Congrats, you solved this challenge!

- Test Case #0
- Test Case #1
- Test Case #2
- Test Case #3
- Test Case #4

Next Challenge