Code Jam 2014 - Round 1A

Proper Shuffle

Problem

A *permutation* of size **N** is a sequence of **N** numbers, each between 0 and **N**-1, where each number appears exactly once. They may appear in any order.

There are many (**N** *factorial*, to be precise, but it doesn't matter in this problem) permutations of size **N**. Sometimes we just want to pick one at random, and of course we want to pick one at random *uniformly*: each permutation of size **N** should have the same probability of being chosen.

Here's the pseudocode for one of the possible algorithms to achieve that goal (we'll call it the *good* algorithm below):

```
for k in 0 .. N-1:
  a[k] = k
for k in 0 .. N-1:
  p = randint(k .. N-1)
  swap(a[k], a[p])
```

In the above code, `randint(a .. b)` returns a uniform random integer between **a** and **b**, inclusive.

Here's the same algorithm in words. We start with the *identity* permutation: all numbers from 0 to **N**-1 written in increasing order. Then, for each **k** between 0 and **N**-1, inclusive, we pick an independent uniform random integer $\mathbf{p_k}$ between **k** and **N**-1, inclusive, and swap the element at position **k** (0-based) in our permutation with the element at position $\mathbf{p_k}$.

Here's an example for **N**=4. We start with the identity permutation:

0 1 2 3

Now **k**=0, and we pick a random $\mathbf{p}_0$ between 0 and 3, inclusive. Let's say we picked 2. We swap the 0th and 2nd elements, and our permutation becomes:

2 1 0 3

Now **k**=1, and we pick a random $\mathbf{p}_1$ between 1 and 3, inclusive. Let's say we picked 2 again. We swap the 1st and 2nd elements, and our permutation becomes:

2 0 1 3

Now **k**=2, and we pick a random $\mathbf{p}_2$ between 2 and 3, inclusive. Let's say we picked 3. We swap the 2nd and 3rd elements, and our permutation becomes:

2 0 3 1

Now **k**=3, and we pick a random $\mathbf{p}_3$ between 3 and 3, inclusive. The only choice is 3. We swap the 3rd and 3rd elements, which means that the permutation doesn't change:

2 0 3 1

The process ends now, and this is our random permutation.

There are many other algorithms that produce a random permutation uniformly. However, there are also many algorithms to generate a random permutation that look very similar to this algorithm, but are not uniform — some permutations are more likely to be produced by those algorithms than others.

Here's one bad algorithm of this type. Take the *good* algorithm above, but at each step, instead of picking $\mathbf{p_k}$ randomly between **k** and **N**-1, inclusive, let's pick it randomly between 0 and **N**-1, inclusive. This is such a small change, but now some permutations are more likely to appear than others!

Here's the pseudocode for this algorithm (we'll call it the *bad* algorithm below):

```
for k in 0 .. N-1:
  a[k] = k
for k in 0 .. N-1:
  p = randint(0 .. N-1)
  swap(a[k], a[p])
```

In each test case, you will be given a permutation that was generated in the following way: first, we choose either the good or the bad algorithm described above, each with probability 50%. Then, we generate a permutation using the chosen algorithm. Can you guess which algorithm was chosen just by looking at the permutation?

Solving this problem

This problem is a bit unusual for Code Jam. You will be given **T** = 120 permutations of **N** = 1000 numbers each, and should print an answer for each permutation – this part is as usual. However, you don't need to get all of the answers correct! Your solution will be considered correct if your answers for at least **G** = 109 cases are correct. However, you must follow the output format, even for cases in which your answer doesn't turn out to be correct. The *only* thing that can be wrong on any case, yet still allow you to be judged correct, is swapping GOOD for BAD or vice versa; but you should still print either GOOD or BAD for each case.

It is guaranteed that the permutations given to you were generated according to the method above, and that they were generated independently of each other.

This problem involves randomness, and thus it might happen that even the best possible solution doesn't make 109 correct guesses for a certain input, as both the good and the bad algorithms can generate any permutation. Because of that, this problem doesn't have a Large input, and has just the Small input which you can try again if you think you got unlucky. Note that there is the usual 4-minute penalty for incorrect submissions if you later solve that input, even if the only reason you got it wrong was chance.

In our experience with this problem, that *did happen* (getting wrong answer just because of chance); so if you are confident that your solution should be working, but it failed, it might be a reasonable strategy to try again with the same solution which failed.

Good luck!

Input

The first line of the input gives the number of test cases, **T** (which will always be 120). Each test case contains two lines: the first line contains the single integer **N** (which will always be 1000), and the next line contains **N** space-separated integers - the permutation that was generated using one of the two algorithms.

Output

For each test case, output one line containing "Case #**x**: **y**", where **x** is the test case number (starting from 1) and **y** is either "GOOD" or "BAD" (without the quotes). You should output "GOOD" if you guess that the permutation was generated by the first algorithm described in the problem statement, and "BAD" if you guess that the permutation was generated by the second algorithm described in the problem statement.

Limits

Time limit: 60 seconds.
Memory limit: 1 GB.
T = 120
G = 109
N = 1000
Each number in the permutation will be between 0 and **N**-1 (inclusive), and each number from 0 to **N**-1 will appear exactly once in the permutation.

Sample

Sample Input

```
2
3
0 1 2
3
2 0 1
```

Sample Output

```
Case #1: BAD
Case #2: GOOD
```

Note

The sample input doesn't follow the limitations from the problem statement - the real input will be much bigger.