2015 Fall Computer Science I Program #5: Risk^{1000000}

Please Consult WebCourses for the due date/time

In the board game Risk, a group of armies will attack from one country to another. To simulate the battle, both teams roll dice. The number of dice vary from attack to attack. To "score" the battle, you match the highest roll of the attacker with the highest roll of the defender. If the attacker's roll is greater than the defender's roll, then the defender loses an army. Otherwise, the attacker loses an army. The comparisons continue, between the attacker's second highest roll and the defender's second highest roll.

Recently, a new game, Risk^{1000000}, similar to Risk, has been released that allows for megabattles of up to 1,000,000 armies versus 1,000,000 armies. Also, rather than simulating a battle with a die roll, an army's strength can be any number from 1 to a billion (instead of 1 through 6 for a die roll). One final difference in Risk^{1000000} is that it gives the defenders an extra advantage. Instead of lining up the highest attacking die with the highest defending die and so forth, the attacker must position his armies for each battle and show this information to the defender. The defender can then choose any ordering of her armies.

To see the difference, let's consider an example from regular Risk and a couple examples from Risk^{1000000}. In the original game Risk, if an attacker with two armies rolls a 6 and 3, while a defender rolls a 5 and a 2, we must match the two maximum rolls (6 versus 5) and the two minimum rolls (3 verses 2), which results in the defender losing two armies.

If we were to have the same situation in Risk^{1000000}, the defender would see that the attacker has 6 for its first army and 3 for his second army. The defender can then strategically place the 2 for her first army and the 5 for her second army, resulting in the loss of one defending army and one attacking army.

Consider a slightly larger situation where the attacker had the following values for its armies showing:

18 9 21 5 3 27 15

Imagine that the defender's army values are:

15 8 6 2 25 19 20

If the defender kept this original ordering, she'd only win two battles (25 versus 3 and 20 versus 15).

An optimal rearrangement of armies for the defender is as follows:

19 15 25 6 8 2 20 versus
18 9 21 5 3 27 15

where the defender wins all but one battle, incurring the loss of just one army (in the 27 versues 2 battle) while the attacker would lose six armies.

The Problem

Write a program that, given a list of the values of both the attackers armies and defenders armies, determines the maximum number of attacking armies that can be defeated by the defender, if she plays optimally.

The Input (read from standard input)

The first line of input will contain a single positive integer, *c* (*c* ≤ *100)*, representing the number of input cases. The first line of each input case will contain a single integer, *n* (*n* ≤ 10^6), representing the number of armies battling from each side. The next *n* lines of the input case will contain a single integer each, representing the value of one of the attacking armies. The following *n* lines of the input case will contain a single integer each, representing the value of one of the defending armies. All of the values on these *2n* lines will be positive integers less than or equal to one billion.

The Output (to standard out)

For each input case, output a single integer, on a line by itself, representing the maximum number of attacking armies that can be defeated by the defender.

Sample Input

```
2
2
3
6
5
2
3
2
3
12
9
3
4
```

Sample Output

```
1
2
```

Implementation Detail - Sorting (must implement either Merge Sort or Quick Sort)

The goal of this problem is to practice coding either Merge Sort or Quick Sort. Significant credit will be given only if your solution contains an implementation of one of these sorts. ***Thus, if you use C's built in sorting function or solve the problem without sorting values, you will lose significant credit, even if your solution is correct.***

Please write your own implementation of the sort you choose instead of copying the one on the course web page or copying one from online. The whole point of this assignment is to give you the critical experience of implementing an efficient sorting algorithm on your own, as every beginning computer science student before you has done.

While there may be other ways to solve this problem, the grading criteria will include points for implementing one of the two aforementioned sorts, sorting both sets of input with them, and then processing that sorted input in either O(n) or O(nlgn) time, using a greedy strategy that always obtains the optimal answer. (Note that while the optimal answer is unique, optimal orderings are not; there may be multiple orderings for a case that achieve the optimal outcome for the defender.)

Deliverables

You must submit a single file, *riskpowmillion.c*, over WebCourses. Please use stdin, stdout.