Fall 2021 CS: 4310 Homework 2

For a positive integer m, an *m-permutation* is simply a permutation of the numbers 1, 2, ..., m. So, for example, a (3, 1, 4, 2) is a 4-permutation. For a pair of m-permutations π and π', the *inversion distance* between π and π', denoted $ID(\pi, \pi')$ is the number of pairs of elements in $\{1, 2, \ldots, m\}$ that appear in the opposite order in π and π'. For example, consider two 4-permutations $\pi = (3, 1, 2, 4)$ and $\pi' = (1, 3, 4, 2)$. Of the 6 possible pairs of numbers $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$, and $\{3, 4\}$, the pairs $\{1, 3\}$ and $\{2, 4\}$ appear in opposite order in π and π'. Therefore, $ID(\pi, \pi') = 2$.

You are given as input, positive integers m, N, and D, followed by N distinct m-permutations. You can assume that the input has format illustrated by the following example. Here $m = 4$, $N = 10$, and $D = 3$.

```
4 10 3
3 1 2 4
1 3 4 2
4 3 2 1
4 3 1 2
2 3 1 4
3 4 2 1
1 3 2 4
1 2 4 3
2 1 4 3
1 2 3 4
```

Write a program that outputs, for each permutation π in the input, the number of permutations π' in the input for which $ID(\pi, \pi') \leq D$. Your output should have 10 lines and the first 2 lines of your output should look like:

```
3 1 2 4: x
1 3 4 2: y
```

with `x` replaced by the actual number of 4-permutations among the 10 given 4-permutations that are within inversion distance of 3 from permutation (3, 1, 2, 4) and similarly `y` replaced by the actual number of 4-permutations among the 10 given 4-permutations that are within inversion distance of 3 from permutation (1, 3, 4, 2).

The core of your implementation should be 3 functions, let us call them `InversionDistancev1`, `InversionDistancev2`, and `InversionDistancev3`, which implement 3 different algorithms for computing the inversion distance between a pair of m-permutations. Each of these functions has the same inputs and outputs. Specifically, the functions take as input two m-permutations `Pi1`, `Pi2`, and a positive integer `D`. The functions return -1 if the inversion distance between `Pi1`, `Pi2` is (strictly) greater than `D`. Otherwise, the functions return the inversion distance between `Pi1` and `Pi2`.

Now I will briefly describe the algorithms you will implement for these 3 functions. The function `InversionDistancev1` will simply implement the algorithm in which you consider each pair of integers $\{i, j\}$, $1 \leq i < j \leq m$, and check if this pair appears in opposite order in `Pi1` and `Pi2`. This algorithm should run in $O(m^2)$ time. The function `InversionDistancev2` will implement a divide-and-conquer algorithm, running in $O(m \log m)$ time, based on MergeSort. To find out exactly how this algorithm works watch Tim Roughgarden's videos on Section 3.2, Part 1 and Part 2. The function `InversionDistancev3` will implement a small improvement to `InversionDistancev2`. Specifically, if it turns out that when we call `InversionDistancev3` on the left half and we get a `-1` back, it means that we already know that the inversion distance between `Pi1` and `Pi2` is too large and there is no reason to do anything else.

Additional instructions: (i) You can use Python or Java for your implementation. (ii) Your code should be extremely well-documented and easy-to-read. (iii) It is critical that the asymptotic running times of your implementations are exactly as required. (iv) Further details on exactly what you should turn in will be provided in 3-4 days.