UCF Local Contest — September 3, 2011

An (Almost) Perfect Match

filename: `music`

You've been using an awesome new service from Google that lets you stream your music from the cloud. However, there's one part of the service that you really don't like: it takes forever to upload all of your music to their servers. Being a smart computer scientist and avid computer programmer, you decide you are going to fix this problem for them. You realize that with all the people using Google Music, there are probably a lot of duplicate tracks (for instance, nearly everyone has their own copy of Poker Face by Lady Gaga in their locker). When a new user comes along and wants to upload their own copy of Poker Face, it would be much more efficient to recognize that this is a match to an existing track and give the new user access to an existing copy instead of making them upload their own copy.

Let's assume that you've already figured out a nice way to create a fingerprint of a track as a sequence of integers. Assuming F is a function that takes as input a track and returns the fingerprint of that track, it's trivially true that tracks x and y are the same if F(x) = F(y). Unfortunately, your life is not this easy! Often the same track encoded by two different users will have close, but not identical, fingerprints. Because you want your service to be robust and not too picky about slight differences in encoding formats, your matching algorithm should allow tracks that are *close enough* to still be considered a match.

In order to define *close enough*, we need to define some terms. Recall that the fingerprint of a track is a sequence of integers. A single integer in this sequence is called a block, and between 1 and 5 contiguous blocks are called a section. Formally, we define a new track x to match an existing track y if:

1. At most K sections of track y are missing from track x.
2. Each non-deleted block of track y is within a tolerance T of the matching block in track x.

In other words, the absolute value of the difference between each matched block is at most T. K and T are parameters that you set before running your matching algorithm against a set of tracks.

Consider the following example:

Existing track y: 6 12 11 6 7 14 25
New track x: 6 7 7 22

If K=3 and T=5, then x matches y by deleting two sections (12 11, 14) and the aligned blocks being within the required tolerance (6:6 6:7 7:7 25:22).
If K=1 and T=7, then x matches y by deleting one section (12 11 6) and the aligned blocks being within the required tolerance (6:6 7:7 14:7 25:22).
If K=1 and T=2, then x does not match y.

The Problem:

Write a program that reads the fingerprints of all existing tracks and the fingerprints of a set of new tracks to add and determines for each whether the new track matches any existing track.

The Input:

Input will begin with a positive integer C denoting the number of test cases to process. Following this will be C test cases. Each test case will begin with a line containing two non-negative integers $K \leq 20$ and $T \leq 255$ denoting the number of sections that may be deleted and the tolerance, respectively. Following this will be a line containing a single non-negative integer $E \leq 100$ denoting the number of existing tracks on the server. This will be followed by E lines, each containing the fingerprint of an existing track. Each fingerprint will begin with a non-negative integer $N \leq 100$ denoting the length of the fingerprint, and will be followed by N space-separated integers (each integer will be between 0 and 255 inclusive). This will be followed by a line containing a single non-negative integer $U \leq 100$ denoting the number of new tracks the user wants to upload. This will be followed by U lines, each containing the fingerprint of a new track to upload. Each fingerprint will be of the same format described above.

The Output:

For each test case, start with a line "Case #x:" where x is the test case number, starting with 1. Follow this with U lines of output, one for each new track the user wants to upload. For each track, output either the line "Track #i: Match found!" or "Track #i: Need to upload this track." where i is the track number starting with 1. Follow the output of each test case with a blank line.

Sample Input:

```
3
3 5
1
7 6 12 11 6 7 14 25
1
4 6 7 7 22
1 2
1
7 6 12 11 6 7 14 25
1
4 6 7 7 22
0 1
2
5 1 2 3 4 5
5 5 4 3 2 1
3
5 0 2 3 4 5
5 4 4 3 2 1
4 1 2 3 5
```

Sample Output:

```
Case #1:
Track #1: Match found!

Case #2:
Track #1: Need to upload this track.

Case #3:
Track #1: Match found!
Track #2: Match found!
Track #3: Need to upload this track.
```