TENNIS CLUB

Ideas for solution

It's quite easy to recognize that this task is about reconstructing a graph from it's degree sequence:

> `http://mathworld.wolfram.com/DegreeSequence.html`
> `http://mathworld.wolfram.com/GraphicSequence.html`

The important part from the above links is the following result:

> Hakimi (1962) and Havel (1955) showed that if a degree sequence is graphic, then there exists a graph G such that the node v of the highest degree is adjacent to the d next highest degree vertices of G, where d is the maximum degree of G.

Applying the above result recursively to the graph $G - v$, we can conclude that the problem is solvable using the following greedy algorithm:

> order the vertices in non-increasing order of degrees
> let v be the first vertex from the ordered list (i.e. the one with maximal degree)
> let d be the degree of v
> if $d = 0$ then stop: we're done
> for $i = 1$ to d do
> let v' be the $i + 1$'th vertex from the ordered list
> let d' be the degree of v'
> if $d' = 0$ then stop: we have a contradiction
> add edge (v, v') to the graph
> set degree of v' equal to $d' - 1$
> end for
> set degree of v equal to 0
> start over from beginning

When implemented naively, the above algorithm can take N^3 steps in the worst case: the list of vertices is sorted N times, with N^2 steps spent on each sorting. An obvious optimization is to use a better sorting algorithm, which yields $N^2 \log N$ steps in the worst case.

Another optimization is to take advantage of the fact that, after the first pass, the list of vertices is already mostly sorted. Leaving out the head element, which we can ignore during the next iterations, the rest of the list consists of two sorted sub-lists with known endpoints. Normally this kind of list could be re-sorted using a merge algorithm in linear time. In this case we need to do even less than a full merge, because we know that initially the list was sorted and the degree of each item in the first sub-list was reduced by one while the second sub-list did not change at all.

The above optimization does not give us better worst-case complexity if the indices have to be sorted on each output line, because for a complete or near-complete graph sorting the lists of indices would require about $N \log N$ operations for each vertex, consuming a total of $N^2 \log N$ operations. However, because the indices are from a very narrow range, we can sort them in linear time using the counter-sort, and thus truly reduce the worst-case complexity to N^2. Or, we could construct the graph into an adjacency matrix to avoid the counting step in the counter-sort.

Yet another optimization is to count the total sum of degrees when reading the data and use it's parity as a quick rejection test. This does not change the worst-case complexity, but could still be handy in a real-life application if the problem source did not eliminate such inputs by definition.

Test cases

A. Small positive test case, can be solved with exhaustive search, can be solved with greedy algorithm that sorts the players just once in the very beginning.

B. Small positive test case, can be solved with exhaustive search, fill fail a greedy algorithm that sorts the players just once in the very beginning.

C. Smallish random positive test case: 30 players, any number of games.

D. Medium random positive test case: 250 players, any number of games.

E. Large random positive test case: 1000 players, any number of games.

F. Large sparse positive test case: 1000 players, each playing 10–30 games.

G. Large dense positive test case: 1000 players, each playing 990 games.

H. Trivial negative test case: the total number of games is odd.

I. Medium non-trivial negative test case.

J. Large non-trivial negative test case.

Our intention is to tune the time limit so that the naive solution would exceed it in about 3 largest tests. It's probably not possible to estimate the constant factors precisely enough to distinguish between the $N^2 \log N$ and N^2 solutions.