You are to write a Java program (Eclipse project) that will accept a directed graph description in adjacency list
form and either (a) output the vertices, starting at vertex 0, of a cycle that traverses each edge exactly once, or (b)
announce that the graph admits no such cycle. The input file has the form on the left below and corresponds to the
graph on the right.

The first entry on each line is a vertex number. The remaining entries are the neighbors of that vertex. The first
line of the example indicates that (0,1) and (0,2) are edges in the graph. This particular graph, which is available
as g1.txt, does admit a cycle of the type described, namely

The required output is just the text, the cycle, not the picture. To accommodate larger graphs, you should format
the output cycle, if you find one, with 20 vertices per line. In the example, you can check that each edge is traversed
exactly once and the cycle begins and ends with zero.

Your algorithm should be \( \Theta(V + E) \), which is the time required to read the input. Your Java program can use the
ArrayList class as though it were a linked list in terms of performance. That is, you can add a node anywhere in the
list in \( \Theta(1) \) time, \textbf{provided} you do not have to search for the insertion point. Assuming that you know the insertion
point, that operation is simply swapping a few pointers, so we will assume that the ArrayList class avails itself of
that efficiency. Otherwise, you could use a linked list, but that requires implementing all of the pointer swaps.

Please submit the following files in an email prior to the assignment deadline.

1. A copy of the output (.txt file) corresponding to the input on g2.txt.
2. A copy of your Eclipse project as a .zip file
3. A writeup (\LaTeX-prepared .pdf file) that describes how you achieve your goal and why the complexity is
   \( \Theta(V + E) \), linear in the graph size.
Solution:

We first establish the graph as an adjacency list. I used an inner Vertex class, in which each edge list is an ArrayList.

```java
private static class Vertex {
    public int in_degree = 0;
    public int out_degree = 0;
    public ArrayList<Integer> edges = new ArrayList<Integer>();
}
```

The cycle that we are seeking is called an Euler Tour. A little thought convinces us that an Euler Tour cannot exist unless each vertex \( v \) satisfies \( v.\text{out-degree} = v.\text{in-degree} \). To traverse each edge exactly once is to enter and leave by separate edges on each visit to a vertex. We discussed simple linear time algorithms for annotating vertices with their in-degrees and out-degrees. If the graph contains a vertex with unbalanced in and out-degrees, we announce that no Euler Tour is possible.

Otherwise, we start at vertex 0 and follow edges. After traversing an edge and recording it in our tour, we destroy the edge. We continue until we find ourselves at a vertex with no exit edge. Note that in-degree and out-degree equality at each vertex means that the evolving path cannot branch into a different strongly connected component. If it did, it would leave the initial component with an unbalanced incoming edge, which would have to come from outside the component. That scenario would produce a cycle among the strongly connected components — a contradiction. So, we must eventually return to vertex 0, where we started the tour. All other vertices have matched in and out edges. If we can come in, then we can leave. So, we have a cycle that begins and ends with vertex 0, but there still may be some untraversed edges.

The sketch shows a cycle that leaves vertex 0 and returns twice before exhausting all of its edges. But, there can still be another cycle that launches from some other vertex. But, our tour is destroying edges as it uses them. So, we keep a follower pointer to the node where we initiated our first cycle. When it exhausts all edges at that vertex, we simply follow the existing tour until we find a vertex with intact edges. We initiate a new cycle at this point and splice it into the existing one.
In the sketch, we might start via 1, 2, 3, 4, 5, 1, 6, 7, 8, 1. At this point we have exhausted all edges at vertex 1. We move our “follower” along 1, 2, 3, 4 encountering exhausted vertices until we reach 4. We then splice 9, 10, 11, 12, 4 after the existing 4 entry to obtain 1, 2, 3, 4, 9, 10, 11, 12, 4, 5, 1, 6, 7, 8, 1.

Using an ArrayList to maintain the Euler Tour is particularly convenient. When you splice new vertices into an existing ArrayList, it shifts entries on the far side of the splice to the right, which is precisely the behavior that we need. However, adding splices forces internal movement of potentially large segments, which will destroy our desired linear performance. A similar drawback holds for the ArrayList that implements the adjacency list for each vertex. Recall that we are destroying edges as we traverse them, which entails a deletion from the ArrayList.

Each of the splicing and deleting activities can be done in $\Theta(1)$ time per operation, if we were to use a linked list instead. I believe that the Java ArrayList class is designed so as to perform splices and deletions in amortized $\Theta(1)$ time. Nevertheless, we consider the following an acceptable solution, given that we could implement the model with linked list if absolutely necessary.

We assume that the input graph is available as a vector of Vertex objects, and that the vertices have been annotated with in-degrees and out-degrees. There preliminaries are linear operations that will not detract from the linear running time of our solution. The ArrayList tour contains the vertices along a tour starting at vertex zero.

```java
ArrayList<Integer> tour = new ArrayList<Integer>();
tour.add(0);
int src = 0;
while (true) {
    int k = tour.get(src);
    int insert = src + 1;
    while (graph[k].edges.size() > 0) {
        k = graph[k].edges.remove(0);
        tour.add(insert, k);
        insert++;
    }
    src = src + 1;
    if (src >= tour.size())
        break;
}
```

Here is the Euler tour for graph g2.txt.

```
0 1 4 6 7 8 5 6 9 5 7 9 8 6 4 3 1 2 3 0 2 4 0 10 40 60 61 64 66 67 68 65 66 69 65 67 69 68 66 64 63 61 62 63 60 62 64
60 70 71 74 76 77 78 75 76 79 75 77 79 78 76 74 73 71 72 73 70 72 74 70 80 81 84 86 87 88 85 86 89 85 87 89 88 86
84 83 81 82 83 80 82 84 80 50 51 54 56 57 58 55 56 59 57 59 58 56 54 53 51 52 53 50 54 53 50 52 54 50 60 90 91 94 96 97
98 95 96 99 95 97 99 98 96 94 93 91 92 93 90 92 94 90 50 70 90 80 60 40 30 10 11 14 16 17 18 15 16 19 15 17 19 18
38 35 36 39 35 37 39 38 36 34 33 31 32 33 30 32 34 30 0 20 40 41 44 46 47 48 45 46 49 45 47 49 48 46 44 43 41 42 43
40 42 44 40 0
```