```
/* FILE: long_walk.cpp   last change: 8-Sept-2012   Romeo Rizzi
 * a solver for problem long_walk in 28-09-2012 exam in Algorithms
 */
/* BASIC FACTS: a digraph is a directed graph. A node with no exiting arcs is ca
lled a sink. In a digraph, no directed circuit can pass through a sink. Converse
ly, if a directed graph has no sink then it must necessarily contain a directed
circuit since once can be found as follows: place a pebble into a node and then
move it from nodo to node following the arcs (from the tail node to the head nod
e) until it gets back to an already visited node.
 ALGORITHM: until the digraph has some sinks, we keep removing them (no directed
 circuit can pass through a sink) meanwhile labelling each of them with the leng
th of a longest walk starting from it (whence entirely made of removed nodes). I
ndeed, the subdigraph induced by the removed nodes is a DAG (=directed acyclic g
raph), whence these longest paths can be computed by dynamic programming.
 IMPLEMENTATION: to get a linear time algorithm, when a node becomes a sink it i
s put in the stack "LIFOsink" where it stays until it gets removed from the digr
aph. (A node becomes a sink when its out_degree drops to 0).
The original digraph is stored in star representation implemented by means of 2
vectors for the out-neighborhoods plus 2 vectors for the in-neighborhoods. The b
oolean array "removed" is used to spot out removed nodes
                                              .
*/

#define NDEBUG   // NDEBUG definita nella versione che consegno
#include <cassert>
#ifndef NDEBUG
#  include <iostream>  // uso di cin e cout non consentito in versione finale
#endif
#include <fstream>

using namespace std;

const int MAX_N =  100000; int n; // numero nodi.
const int MAX_M = 1000000; int m; // numero archi.
const int TAIL=0, HEAD=1;
int out_deg[MAX_N +1], in_deg[MAX_N +1]; // i nodi (e gli archi) sono numerati d
a 1 ad n
int arc[MAX_M +1][2]; // a temporary buffer to receive the input. L’arco j-esimo
 e’ (arcs[j][TAIL], arcs[j][HEAD])
int first_out_nei[MAX_N +2], out_nei[MAX_M]; // 2 vectors to represent the out-n
eighborhoods of each node
int first_in_nei[MAX_N +2], in_nei[MAX_M]; // the very same for the in-neighs
int LIFOsink[MAX_N], LIFOpos = 0; // just an handy bag where to store nodes whic
h have become sinks
bool removed[MAX_N +1];
int max_from[MAX_N +1], next[MAX_N +1], max_so_far, max_start; // implement the
dynamic programming to find the max lenght path; their value is defined on top o
f the already removed nodes. When v is a removed node then:
// max_from[v] is the maximum length of a path starting at v;
// next[v] stores the second node of a maximum length path starting at v.
// Meanwhile, max_so_far is the running maximum of the max_from[v] values over t
he nodes v removed so far and max_start stores the removed node achieving this m
aximum.

/* void displayVect( int v[], int from, int to) {
  for(int i = from; i <= to; i++)
    cout << v[i] << " ";
  cout << endl;
  } */

int main() {
  ifstream fin("input.txt");  assert( fin );
  fin >> n >> m;
  for(int i=0; i<=n; i++)
     next[i] = max_from[i] = out_deg[i] = in_deg[i] = 0;
  for(int j = 1; j <= m; j++) {
    fin >> arc[j][TAIL] >> arc[j][HEAD];
    out_deg[ arc[j][TAIL] ]++; in_deg[ arc[j][HEAD] ]++;
  }
```



```
  fin.close();

  first_out_nei[1] = first_in_nei[1] = 0;
  for(int i = 1; i <= n; i++) {
    first_out_nei[i+1] = first_out_nei[i] + out_deg[i];
    first_in_nei[i+1] = first_in_nei[i] + in_deg[i];
  }
  int cur_out_nei[MAX_N +1], cur_in_nei[MAX_N +1]; // temporary arrays auxiliary
 to the compilation of out_nei and in_nei
  for(int i = 1; i <= n; i++) {
    cur_out_nei[i] = first_out_nei[i];
    cur_in_nei[i]  = first_in_nei[i];
  }
  for(int j = 1; j <= m; j++) {
    out_nei[ cur_out_nei[ arc[j][TAIL] ]++ ] = arc[j][HEAD];
    in_nei[ cur_in_nei[ arc[j][HEAD] ]++ ] = arc[j][TAIL];
  } // displayVect( out_deg, 1, n); displayVect( first_out_nei, 1, n+1); display
Vect( out_neighbour, 0, m-1);

  for(int i=1; i<=n; i++) {
    removed[i] = false;
    if( out_deg[i] == 0 )  LIFOsink[ LIFOpos++ ] = i;
  }
  int n_removed = 0;

  while( LIFOpos ) {  // hearth of the algorithm
    int v = LIFOsink[ --LIFOpos ];
    removed[v] = true; n_removed++;

    for(int i=first_out_nei[v]; i<first_out_nei[v+1]; i++) // begin: dyn prog
      if( max_from[ out_nei[i] ] >= max_from[v] ) {
        max_from[v] = max_from[ out_nei[i] ] +1;
        next[v] = out_nei[i];
        if( max_from[v] > max_so_far )  {
          max_so_far = max_from[v]; max_start = v;
        }
      } // end: dynamic programming

    for(int i=first_in_nei[v]; i<first_in_nei[v+1]; i++) {// begin: update graph
      out_deg[ in_nei[i] ] --;
      if( out_deg[ in_nei[i] ] == 0 )
      LIFOsink[ LIFOpos++ ] = in_nei[i];
    } // end: updating the graph (node v has been removed)
  }

  ofstream fout("output.txt");  assert( fout );
  if( n_removed == n ) { // it was a DAG: we are ready to output a longest path
    fout << max_so_far << endl;  // max length of a path in the removed nodes
    int v = max_start;  // max_start = the first node of a maximum length path
    while( v ) {  // we print the nodes one by one
      fout << v << " ";
      v = next[v];   // next[] was properly set up during the dyn programming
    }
  }
  else {  // we are left with some nodes but no sink -> there must be a cycle
    fout << -1 << endl; // there must be a cycle, we now search for it ...
    bool visited[MAX_N +1]; // with the algo described in "BASIC FACTS"
    for(int i = 1; i <= n; i++)  visited[i] = false; // no node visited yet
    int v = 1; while( removed[v] )  v++;  // place the pebble in any unremoved v
    while( !visited[v]) {
      visited[v] = true;  // cout << endl << endl << v << " ";
      for(int i=first_out_nei[v]; i<first_out_nei[v+1]; i++)
        if( !removed[ out_nei[i] ] ) {
          next[v] = out_nei[i];
          //cout << "   next[" << v << "]=" << next[v] <<", i = " << i
          //     << ", first_out_nei[v+1] = " << first_out_nei[v+1];
        }
      v = next[v];
    }
```

```
    int u = v; //we got back to an already visited node v, now go round again...

    do { // and print the nodes one by one
      fout << u << " "; // cout << u << " ";
      u = next[u];
    } while( u != v );
  }
  fout.close();
  return 0;
}
```