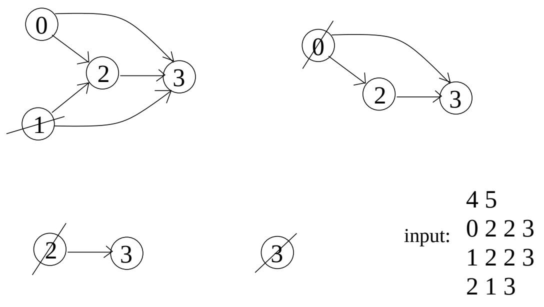

input:
```
4 5
0 2 2 3
1 2 2 3
2 1 3
```

Figure 1: Successively deleting sources from a directed graph. If acyclic, the order of deletion is a topological order.

16 Topological sort

A *directed cycle* in a digraph (V, E) is a sequence $x_0, \ldots, x_k$ such that for $0 \leq j < k$, $(x_j, x_{j+1}) \in E$. It is *nontrivial* if $k > 0$.

A *topological order* on a digraph is a sequencing $x_0, \ldots, x_{n-1}$ of the vertices such that no edge (x_i, x_j) is 'retrograde,' meaning that no if (x_i, x_j) is an edge then $i < j$.

A *source* is a vertex of indegree zero, i.e., v is *not* a source if and only if there exists an edge (u, v).

(16.1) Definition *G is acyclic if and only if it has no nontrivial directed cycles.*

(16.2) Theorem (i) *G admits a topological order if and only if it is acyclic.*

(ii) G is acyclic if and only if it is empty, or it contains a source u, and $G \backslash u$ is also acyclic. ▮

A topological sort of a digraph G can be constructed by repeatedly choosing some (any) source u, and replacing G by $G \backslash u$. This means removing u from the vertex set, and removing all outedges from u from the edges of G. Figure 1 shows sources being crossed out in a loose simulation of the process. The topological order is $1, 0, 2, 3$. Another topological order is $0, 1, 2, 3$. Topological order is not unique.

It is not necessary to do so much work. Build an array `indegree[]` which at every stage gives the indegree of vertices as more and more are deleted from the digraph.

This can be implemented efficiently. The following code fragment gives the main program. The full program is stored in the 'sample code' subdirectory.

```
int main()
{
  int n, m;

  DIGRAPH * digraph = read_digraph();

  print_digraph ( digraph );
```

```
int indeg[n];
int sources[ n ];
int s_count = 0;

set_indegrees ( indeg, digraph, sources, & s_count );

int topsorted[n];
int ts_count = 0;

while (  s_count > 0 )
{
  int next = sources[ s_count - 1 ];
  -- s_count;
      // take another source
  topsorted[ ts_count ] = next;
  ++  ts_count;
      // next in topological order
  int j;
  EDGE * e = digraph->an_edge[next];
      // iterate through edges out of ‘next.’
  for ( j=0; j<digraph->out_deg[next]; ++j )
  {
    int k = e->to;
    -- indeg[k];
      // edge e = (next,k): decrement indegree of k

    if ( indeg[k] == 0 )
    { sources[s_count] = k; ++ s_count; }
      // if k has zero indegree, add it
      // to the list of sources.

    e = e->next;
  }
}
      // At this stage, the deleted
      // graph has no sources. If
      // empty, we have a topological
      // sort.  If nonempty, the
      // graph contains a directed cycle.

if ( ts_count == digraph->n )
{ printf("topological order\n");
  int i;
  for(i=0; i<ts_count; ++i)
    printf(" %d", topsorted[i]);
    printf("\n");
  }
  else
  { printf("Not acyclic, no topological order\n"); }
}
```

Sample runs

```
% a.out < acyclic-1
4 5
0 2 2 3
1 2 2 3
2 1 3
3 0
topological order
 1 0 2 3
% a.out < cyclic-1
4 4
0 1 1
1 1 2
2 1 3
3 1 1
Not acyclic, no topological order
%
```

16.1 Runtime

One usually measures the runtime of a graph algorithm in terms of n and m, the number of vertices and edges. The best runtime would be linear time: $O(m+n)$.

The runtime in the while-loop in the main program is $O(m+n)$, as follows.

- In an iteration a source `next` is chosen and added to the 'topologically sorted' list of vertices: $O(1)$.
- The edges `(next,k)` out of `next` are inspected in turn and `indegree[k]` decremented to reflect the simulated removal of `'next'` from the graph.

  Also, `k` may be added to the list of sources.

  Write u for `next` and d_u for the out-degree of u. The cost of the iteration is $O(1+d_u)$.
- The overall cost is

$$O\left(\sum_u (1+d_u)\right), \quad \text{i.e.,}$$

$$O(n+m)$$

- The other bits and pieces (reading the digraph, setting up the `indegree` array) are also $O(m + n)$.
- Linear time.