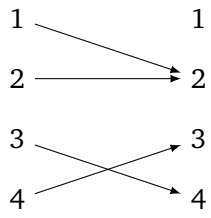
CS180 Homework 2 Solution

1. Let f be a function that maps a set A of integers $\{1 \dots n\}$ to itself. Assume f is represented by an array $f[1..n]$ such that $f[i]$ holds the value of $f(i)$ (which is an integer between 1 to n). We call f a one-to-one function if, for every element j, there is at most one element i that is mapped to j. An example of function f can be represented by the following diagram.



(An example of f that maps $\{1,2,3,4\}$ to iteself, f is not one-to-one)

Given such a function f in the array representation $f[1..n]$, design an algorithm that find a subset $S \subseteq \{1 \dots n\}$ with the maximum number of elements, such that

(a) the function f maps every element of S to another element of S (i.e., f maps S into itself)

(b) no two elements of S are mapped to the same element (i.e., f is one-to-one when restricted to S)

(c) your algorithm should run in $O(n)$ time

We can think about the problem inductively. If there is only one element in the set , then the subset is just itself since the function must be a one-to-one function. Assume we know how to solve the problem for sets of $n-1$. To reduct a problem with set of n, we claim that any element i such that no other element mapped to it cannot belong to S. Otherwise, the function restricted to S will map from $|S|$ elements to $|S|-1$ elements, and so the function will not be one-to-one. Hence, if there exists such an element that no element maps to it, we can remove them and solve the subproblem by our induction hypothesis. We design an interactive algorithm using the previous idea: repeatedly find an element that has no element maps to, and remove the element.

```
ONE-TO-ONE(f)
  S ← {1...n}
  for i ← 1 to n
      counter[i] ← 0     \\initialized a counter to count how many elements map to i
  for i ← 1 to n
      increment counter[f[i]]
  for i ← 1 to n
      if counter[i] = 0 then add i in Queue
  while Queue is not empty    \\repeatedly remove the empty element and update
      remove k from the top of the Queue
      S ← S − {k}
      decrement counter[f[k]]
      if counter[f[k]] = 0
          add f[k] in Queue
  return S
```

2. A node v in a rooted tree is an *ancestor* of node u if the path from the root to u passes through v. The *lowest common ancestor* of two nodes u and v in a rooted tree T is the node w that is an ancestor of both u and v and that has the greatest depth in T. Given two nodes in an undirected tree, design an algorithm that computes the lowest common ancestor in $O(h)$ time, where h is height of the tree. You are allowed $O(|E|)$ preprocessing time before the two nodes are given to you.

   The idea is we can construct the path from u and v to the root and compare them level by level from bottom to top.

   Assume the tree is represented in a standard data structure that every node have serval pointers that point to its children

```
PREPROCESSING(T, u, v)
  for every node v in the tree
      for each child c
          add a parent pointer in c that points to v
  save the paths from the root to u and v using the parent pointer in array u[1..h] and v[1..h]
  reverse u[1..h] and v[1..h]
```

   The preprocessing takes $O(|E|)$ time since for each node except root we add a pointer and there are $|E|+1$ nodes. The path recovering takes $O(h)$ time. Then we just need to compare the nodes in two paths from the level $h-1$ to1

```
ANNCESTOR(T)
  for i ← h − 1 to 1
      if u[i] = v[i]
          return u[i]
```

3. Given a directed acyclic graph (DAG) $G = (V, E)$, design an algorithm that finds the longest directed path in G in $O(|E|)$ time. (Hint: use the technique in finding the diameter of the tree from the lecture. You are required to use only what we have learned so far.)

   The idea of the algorithm is to remove all the sources and repeat. The total number of rounds will be the length of the longest path in the DAG. For each vertex we can store the round number when it is removed. To recover the longest path, we can start from the vertex with the maximum round and go to its neighbour against the edges with a round number reduced by one.

```
CALROUND(G)
  for each vertex v in G
      if v is a source
          add v in Queue
          v.round ← 0
  while Queue is not empty
      remove v from the top of the Queue
      for each outgoing edges v → w
          remove v → w
          if w has no incoming edges
              add w in Queue
              w.round ← v.round + 1
```

```
RECOVERPATH(G)
  R ← the maximum round recorded
  v ← a node with round R
  for i ← R to 1
      go to v's neighbour w against the direction such that w.round = i − 1
      v ← w
  return the reversed path
```

To prove the correctness of the algorithm, we show that the maximum round number is indeed the length of the longest path.

Theorem 1. *The maximum round number is the length of the longest path in G.*

Proof. The longest path in G is at least the maximum round number since the algorithm we use will return a path which the length is maximum round number. On the other hand, assume a path P is the longest path in G with length $|P|$. the number of round executed in the algorithm will be at least as $|P|$ since we can prove that each round at most one vertex in P will be removed inductively (since each round at most one edge will be removed in the longest path). □

In the algorithm, we processed all vertices and remove all edges and the path construction is a linear scan. So the time complexity of the algorithm is $O(|V| + |E|)$.

4. Given an undirected connected graph $G = (V, E)$ such that all the vertices have even degrees, design an *iterative* algorithm to find the Eulerian path (i.e., a closed path such that every edge in the graph appears in the path exactly once). Your algorithm should run in $O(|E|)$ time.

   We know that an Eulerian path (tour) can be decomposed by some simple cycle in G. Moreover, we know that starting from a vertex and arbitrarily traverse edges will eventually return to the starting vertex, which is a simple cycle. Hence, we can decompose the graph by these simple cycles and then connected these cycles to form a Eulerian tour. Here is the idea of the algorithm.

   Let us start at a vertex u and, via arbitrary traversal of edges, create a cycle and get back to vertex u, and if there are still edges leaving u that we have not taken, we can continue the cycle. Eventually, we get back to vertex u and there are no untaken edges leaving u. If we have visited every edge in the graph G, we are done. Otherwise, since G is connected, there must be some unvisited edge leaving a vertex, say v, on the cycle. We can traverse a new cycle starting at v, visiting only previously unvisited edges, and we can splice this cycle into the cycle we already know. That is, if the original cycle is $(u, ..., v, w, ..., u)$, and the new cycle is $(v, x, ..., v)$, then we can create the cycle $(u, ..., v, x, ..., v, w, ..., u)$. We continue this process until we have visited every edge.

```
EULERTOUR(G)
  T ← {}    //an empty list
  L ← (v)   //v is an arbitrary vertex
  while L is not empty
      remove u from L
      C ← get a simple cycle of u
      remove C from G and add a non-0 degree vertex from C into L
      if T = {}
          T ← C
      else
          splice C into T from u
  return T
```

To see the algorithm takes $O(|E|)$ time, observe that because we remove each edge as it is visited, no edge is visited more than once. Since each edge is visited at some time, the number of vertices we processed in L is at most $|E|$. Hence, the total running time is $O(|E|)$.