Exercises 2

3.5 Our algorithm works as follows. We first create a new empty adjacency list for GR. We traverse the original adjacency list. For each vertex v, we traverse its adjacency list. For each vertex u in the adjacency list, we add vertex v to the adjacency list of u in the new adjacency list.

Runtime: Since we traverse each edge once (we traverse each item of the adjacency list once) and it is O(1) time to add a new item to the adjacency list, we have that this algorithm is linear time.

```
function reverseGraph(adjList) {
    init newAdjList(size = numVertices in adjList)

    for v in adjList {
        for u in adjList[v] {
            newAdjList[u].prepend(v)
        }
    }
    Return newAdjList
}
```

3.9 Our algorithm works as follows. We first create an array degree, where degree[v] is the degree of vertex v. We iterate through each vertex v, counting the number of vertices in each adjacency list and updating degree[v] to this value.

Then we iterate through each vertex again. For each vertex v, we traverse its adjacency list. For each vertex u in the adjacency list, we add degree[u] to twodegree[v].

Runtime: It takes linear time to create the degree array, since we iterate through each edge once (we iterate through every element in the adjacency list once). It takes linear time to create the twodegree array, since we iterate through each edge once (we iterate through every element in the adjacency list once). Hence the algorithm runs in linear time.

Correctness: We have that our algorithm computes twodegree[v] as the sum of all the degree[u] of its neighbours, which is the definition of twodegree. Hence the algorithm is correct.

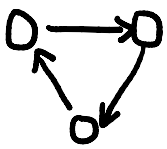
```
function computeTwoDegree(adjList) {
    degree = int[numVertices in adjList]
    twodegree = int[numVertices in adjList]

    for v in adjList {
        for u in adjList[v] {
            degree[v] += 1
        }
    }
    for v in adjList {
        for u in adjList[v] {
```

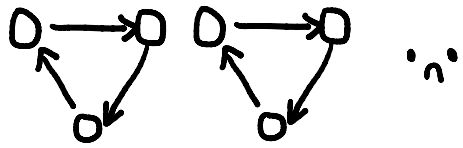
```
                twodegree[v] += degree[u]
            }
        }
        return twodegree
    }
```

3.13 a) We observe the DFS tree for G. Removing a leaf from the DFS tree and its edges leaves the rest of the tree connected, by definition of a tree. Hence we have found a vertex whose removal leaves G connected

b)



c)



3.16 We assume G has no cycles, otherwise there is no way to complete all the courses. We modify the topological sort algorithm to solve this problem. In each iteration, we take all courses with vertex of indegree 0 (have no prerequisites) and remove them from the graph G, updating the remaining indegrees. The number of iterations we do is the minimum number of semesters.

```
function numSemesters(adjList) {
    result = 0
    indegrees = int[numVertices in adjList]
    for v in adjList {
        for u in adjList[v] {
            indegree[v] += 1
        }
    }

    queue = []
    for v in adjList {
        if (indegrees[v] == 0) queue.push(v)
    }

    while (queue) {
        result += 1
        nextLevel = []
        for v in queue {
            for u in adjList[v] {
                indegrees[u] -= 1
                if (indegrees[u] == 0) nextLevel.push(u)
            }
        }
        queue = nextLevel
    }
    return result
```

}

Runtime: Determining the initial indegree of each node takes linear time, since we iterate through each edge once. Iterating through the initial number of nodes that are indegree 0 takes linear time, since we iterate through each vertex once. Then, we visit each node once and check each of its edges once. Hence our algorithm runs in linear time.

Correctness: We use a topological sorting algorithm, which we know will schedule courses in order of prerequisite. Since in each iteration we only schedule courses with 0 indegree, we know they have no uncompleted prerequisites. Hence the algorithm produces a correct scheduling.

We prove that our algorithm outputs an optimal scheduling. Suppose for the sake of contradiction that there exists a smaller number of semesters to schedule all courses than our algorithm. Let x be the number of semesters our algorithm outputs, and y < x be the optimal solution. Let Xi, where 1 <= i <= x be the set of courses our algorithm schedules in semester i, and Yi, where 1 <= i <= y be the set of courses the optimal algorithm schedules in semester i.

We use strong induction on y to show that for every semester, the size of Xi >= size of Yi.

Base case: For the first semester, X1 contains the courses with 0 indegree. The size of Y1 must be less than or equal to X1, since we cannot take any courses other than those with 0 indegree, since these are the only ones with no prerequisites.

Inductive step: Assume that for all i < k, the size of Yi >= size of Xi. Then for semester k, we have that our algorithm schedules Xk courses, the number of current courses with 0 indegree. Since our algorithm has scheduled greater than or equal to the total number of courses scheduled by the optimal algorithm, we have previously scheduled a greater than or equal number of prerequisite courses, so we must have that the size of Xk >= size of Yk. Hence we have proven the statement.

Since in every semester, our algorithm schedules greater than or equal number of courses as the optimal algorithm, our algorithm must use less than or equal number of semester to schedule all the courses, which is a contradiction. Hence our algorithm is optimal and correct.

3.18 We can T is a directed acyclic graph: T is acyclic by definition since it is a tree, and we have that all its edges are directed away from the root. Let Td be the DFS tree of T starting from the root. For any two nodes u and v, we have that start[u] < start[v] and finish[v] < finish[u] if and only if u is an ancestor of v, by the parenthesis property. Hence for this problem, we create two arrays start and finish, where start[v] is the start time and finish[v] is the finish time for vertex v in a DFS starting from the root. To create this, we run a DFS as described in class, keeping track of the start and finish times of each node. This takes linear time.

To look up whether u is an ancestor of v for any two nodes, we simply check whether start[u] < start[v] and finish[v] < finish[u], which takes constant time.

3.24 Our algorithm works as follows. We first run a topological sort on G. This will return a valid topological ordering, since G is acyclic. We then check iterate through this ordering to check if it is a path. If it is, we return true; otherwise return false.

Runtime: Topological sort takes linear time. Iterating through each vertex in the topological ordering takes linear time. Hence our algorithm runs in linear time.

Correctness: Topological sort returns an ordering where every edge goes forward. A path must be a topological ordering, otherwise we would not touch every vertex. If the topological ordering is not a path, then there does not exist such a path, this means there exists some branch on the graph not on the path.

3.23 Our algorithm works as follows. We first run a topological sort on G. This will return a valid topological ordering, since G is acyclic. We create an array numpaths, where numpaths[v] is the number of paths from v to t. We initialize every entry to 0, except numpaths[t] = 1.

We iterate through each vertex in the topological ordering in reverse order. For each vertex v, we iterate through its neighbours u, add the numpaths[u] to numpaths[v]. We then return numpaths[s].

```
function numPaths(G, s, t) {
    numpaths = int[numVertices in G] // 0 initializedt
    numpaths[t] = 1
    topoSort = topologicalSort(G)

    for v in topoSort.reverse() {
        for neighbour u of v {
            numpaths[v] += numpaths[u]
        }
    }
    return numpaths[s]
}
```

Runtime: Topological sort takes linear time. Iterating through each vertex and its edges takes linear time. Hence our algorithm takes linear time.

Correctness: For each vertex v, the number of paths from v to t is the sum of the number of paths from its neighbours to t. Computing these in reverse topological order ensures that for each vertex v, the number of paths of its neighbours are already computed before we reach v. Hence our algorithm is correct.

3.27 We prove this statement using induction on the number of vertices of odd degree. By the handshaking lemma, we have that the number of vertices with odd degree in any connected component of G must be even.

Base case: There is one pair of vertices with odd degree. They must be in the same connected component, so there is a path between them.

Inductive step: Assume for every graph with n pairs of vertices of odd degree, there is pairing where there are edge-disjoint paths between each pair. Consider the graph with n+1 pairs of vertices of odd degree. We find two vertices of odd degree in the same connected component of G. They must exist by the handshaking lemma. Since they are in the same connected component, there is a path between them. We remove all edges on this path. Since we have removed one edge from each of the pair, they now have even degree. Since

we removed two edges from each middle node on the path, they still have the same edge parity. Hence we now have a graph with n pairs of odd vertices. By the inductive hypothesis, there exists a pairing with edge-disjoint paths between the vertices of odd degree. We can add our pair of edges and their path edges back to the graph. Hence we have created a pairing of the vertices of odd degree with edge-disjoint paths between them.

3.26 a) We assume G is connected.

<=
Suppose G has a Eulerian tour. For every vertex in the tour, the walk leaves the vertex right after entering, so each time we visit a vertex we use two edges. Since each edge is used only once, each vertex must have an even degree.

=>
Suppose all the vertices of G have even degree. We use induction on the number of edges in G to prove that G has a Eulerian tour.

Base case: G has 0 edges. Then G is a graph with one node, so G vacuously has a Eulerian tour.

Inductive step: Assume graphs with k < n edges have a Eulerian tour if all vertices have even degree. Observe a graph G with n edges, where all vertices have even degree. Since all vertices have even degree, G must contain a cycle. If the cycle is a Eulerian tour, we are done. Otherwise, we remove all the edges of the cycle from G to create G'. G' must have all vertices with even degree, since we removed two edges from each vertex in the cycle. G' now contains some number of connected components. By the inductive hypothesis, there is a Eulerian tour in each connected component of G'. We add the cycle back to G'. Hence G has a Eulerian tour as follows. We start with a vertex v0 on the cycle. If there is a connected component of G' containing v0, we traverse this component's Eulerian cycle and return to v0. We then continue along the cycle, until we reach another component of X' with a vertex on the cycle, and traverse this Eulerian tour. This process terminates when we return to vertex v0, and thus have found a Eulerian tour for G.

Since the Konigsberg bridges is a graph with vertices of odd degree, it has no Eulerian tour.

b) A graph has a Eulerian path if and only if the graph has at most two vertices of odd degree. Any vertex of odd degree must be at the start or end of the Eulerian path: any vertices in the middle have the path enter and exit the vertex, hence even degree.

c) A directed graph contains a Eulerian circuit if and only if every vertex with non-zero degree is part of a single strongly connected component, and every vertex has its indegree equal to its outdegree.

22-4 Solution 1: Our algorithm works as follows. We first compute all the strongly connected components of the graph. For each strongly connected component, we contract it into a

single vertex, where the label of the vertex is the minimum label vertex in the SCC. We now have a directed acyclic graph on the SCCs. We set each min(v) = v. We then run topological sort on the DAG. We traverse the vertices in the DAG in reverse topological order. For each vertex v, we set min(v) to the minimum of its current min and the min of min(u) for each outgoing edge vu. Since we traverse vertices in reverse topological order, all descendants will be visited before their ancestors.

Runtime: This takes linear time, since computing strongly connected components takes linear time, running topological sort takes linear time, and iterating through each vertex in the topological ordering takes linear time. Hence this algorithm takes linear time.

Solution 2: Our algorithm works as follows. We first compute Gr: G with all its edges inverted. We then sort the vertices in increasing order of label. Starting from the smallest label v, we run BFS or DFS, and updated every visited vertex u to min(u) = v. This visits every vertex that is reachable from v. We continue iterating through our sorted list of vertices, running BFS/DFS on unvisited ones.

Runtime: Inverting the graph takes linear time, and every vertex and edge is visited once. Hence this algorithm takes linear time.

22-3 a) <=

Suppose G has a Euler tour. For every vertex in the tour, the walk leaves the vertex right after entering, so each time we visit a vertex we use one indegree edge and one outdegree edge. Hence each vertex must have the same indegree as outdegree.

=>

Suppose G has indegree(v) = outdegree(v) for each vertex v. Then for every vertex v, there is a path starting from v that comes back to v (since for every outdegree edge there is an indegree edge). Then G must contain a cycle. If the cycle is a Eulerian tour, we are done. Otherwise, we remove all the edges of the cycle from G to create G'. G' must have all vertices with indegree = outdegree, since we removed one indegree edge and one outdegree edge from each vertex in the cycle. G' now contains some number of connected components. By the inductive hypothesis, there is a Euler tour in each connected component of G'. We add the cycle back to G'. Hence G has a Eulerian tour as follows. We start with a vertex v0 on the cycle. If there is a connected component of G' containing v0, we traverse this component's Eulerian cycle and return to v0. We then continue along the cycle, until we reach another component of X' with a vertex on the cycle, and traverse this Eulerian tour. This process terminates when we return to vertex v0, and thus have found a Eulerian tour for G.

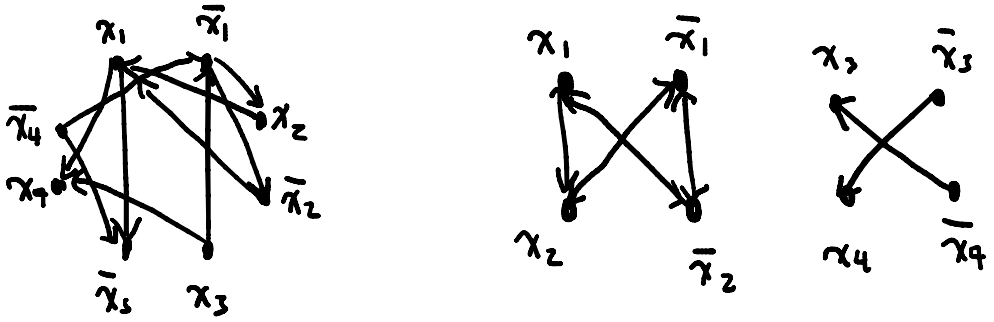
b) We first check whether indegree(v) = outdegree(v) for each vertex. This takes linear time since we check each edge once. If this is false, the graph contains no Euler tour.

Otherwise, we choose some vertex v as a starting point. We push v to a stack. While the stack is nonempty, we observe the top vertex on the stack, u. If u has an unvisited outgoing edge uw, we push w to the stack and mark uw as visited. Otherwise if u has no unvisited outgoing edges, we pop it off the stack. Once the stack is empty, we have that the order in which we popped vertices off the stack is a Euler tour.

3.28 a) x1 = true, x2 = true, x3 = false, x4 = true
There's probably more but I don't feel like finding all of them and neither should you :P

b) (x1 v x2) ^ (!x1 v x2) ^ (x1 v !x2) ^ (!x1 v !x2) ^ (x3 v x4)

c)



d) Suppose GI has a strongly connected component containing both x and !x for some variable x. This means we have a path from x to !x, and a path from !x to x. This means we have some implication !x => x and x => !x, or x <=> !x, which is a contradiction. Hence this means I has no satisfying assignment.

e) Suppose none of GI's strongly connected components contain both a literal and its negation. We repeatedly pick a sink strongly connected component of GI, assign true to all literals and false to their negations, and delete all of them. For any implication !a => b, the implication is always satisfied if b is true. Hence setting the literals to true ensures all implications are satisfied. For each edge !a => b, there exists an edge !b => a elsewhere. This means these are source edges from a source SCC, so setting the negations to false removes both. Since no variable has x and !x in the same SCC, we end up with an empty graph at the end.

f) We have an algorithm as follows. We first convert the expression into a directed graph. We split the graph into its strongly connected components using DFS in linear time. Then, we check if each strongly connected component contains both x and !x for all its variables: this takes linear time since we iterate all variables. If so, we return false: not satisfiable. Otherwise, we use the algorithm described in part e), since we iterate each vertex once, this takes linear time.