(3.9) With an adjacency list graph representation, we first compute the degree of
each node (the number of edges incident on each node) by traversing each adjacency
list. The result can be stored in an array called \( \text{onedefgrees}[u] \) for each vertex \( u \).
The complexity is \( O(|E|) \). We compute the sum of \( \text{onedefgree} \) starting from the sec-
ond node in the adjacency list for vertex \( u \) and assign the result to \( \text{twodegrees}[u] \).
The complexity is also \( O(|E|) \).

(3.8) We build a state transition graph, in which each node indicates a possible
distribution of water in the three containers. Each state is represented by a tuple
\((a, b, c)\) that indicates the 10-pint container has \( a \) pint water, the 7-pint container
has \( b \) point water and the 4-pint container has \( c \) pint water. We create an edge
from vertex \((a, b, c)\) to vertex \((x, y, z)\) if by pouring water one container into another
implements the state transition. With this graph, we want to find out whether there
is a path from vertex \((0, 7, 4)\) to \((x, 2, z)\) and \((x, y, 2)\). The path can be found by
depth first search.

One feasible path is \((0, 7, 4) \rightarrow (4, 7, 0) \rightarrow (10, 1, 0) \rightarrow (6, 1, 4) \rightarrow (6, 5, 0) \rightarrow
(2, 5, 4) \rightarrow (2, 7, 2)\) and the other one is \((0, 7, 4) \rightarrow (4, 7, 0) \rightarrow (10, 1, 0) \rightarrow (6, 1, 4) \rightarrow
(6, 5, 0) \rightarrow (2, 5, 4) \rightarrow (2, 7, 2) \rightarrow (9, 0, 2) \rightarrow (9, 2, 0)\).

(3.10) We denote the vertex set of \( G \) as \( V \) and its edge set as \( E \). The depth-first
search algorithm is as follows:

```python
function explore(G, v)
    create an empty stack S
    for all u in V
        visited(u) = false
    visited(v) = true
    previsit(v)
    push(S, v)
    while S is not empty
        p = pop(S)
        if all the neighbors of p are visited
            postvisit(p)
        else
            push(S, p)
            for each edge (p, q) in E
                if not visited(q)
```
visited(q) = true
previsit(q)
push(S, q)
endif
endfor
endif
endwhile

(3.11) Let $e = (u, v)$. We do a depth first search starting from $v$. If the graph has a circle containing $e$, $u$ must be reachable from $v$ and it must be a descendant of $v$ in the depth first search tree. We therefore just need to check whether the depth first explore starting from $v$ will hit $u$. This is a linear complexity algorithm.

(3.12) It is true. Since there is an undirected edge between $u$ and $v$, in the depth first search tree we have either $u$ as the ancestor of $v$ or the other way around depending on which node is visited first. In another view, the pre-post time brackets of $u$ and $v$ must be embedded to each other, due to the undirected edge between the two nodes.

(3.18) We just need to run depth first search on the tree (a directed graph) starting from the root node. It takes linear time to label the pre-vist and post-vist time for each node on the tree. If $u$ is an ancestor of $v$ on the tree (there is a path from $u$ to $v$), we must have $pre-time(u) < pre-time(v) < post-time(v) < post-time(u)$. It is also true the other way around. Checking the relations between a pair of post-and pre-times takes constant time.

(3.23) Linearize the DAG by following the reversed order of the post-visit times of vertices on the depth first search tree. This takes linear time. We scan the linearized vertices and find the number of continuous nodes $n_i, i = 1..m$ that do not have edges between them. These vertices with no edges between them can be permuted in different orders, while vertices linked by edges must be in specific order. The number of total paths is thus $\prod_{i=1}^{m} (n_i!)$.

(3.24) Sort the DAG using depth first search. If there is an edge between each successive vertices in the ordered list, there is a directed path that goes through each node of the DAG just once. It is apparently true the other way around.

(3.25) Let’s consider the generic directed graph. We find the strongly connected components. Recall that the strongly connected components form a meta-DAG. All the nodes in a strongly connected components have the same cost. We treat each strongly connected component as a single big node. We sort the DAG and then fill up the array cost from the sink nodes, which are initialized as their own costs.
Going up the linearized DAG, we compute the min value among the cost for node $u$ and all the cost of nodes that $u$ connects (these costs have been filled up in order) and assign the result to $\text{cost}[u]$. We repeat the procedure until we exhaust all the vertices in the DAG.