1. Chapter 3, Ex. 3.12
    a. Prove Graph-Search works
        i. The graph separation property shown in Figure 3.9 shows that for each node that the agent can start in, the agent must travel through a frontier node before reaching an unexplored node.
        ii. In the initial state, the initial node is the frontier
        iii. With the way that graph search works, once it explores the initial node, it will expand the frontier to all nodes that are connected to it, making those frontier nodes.
        iv. Thus, no matter what node the graph search decides to explore next, it must be a frontier node – making it impossible for the graph search agent to reach an unexplored node without first passing through a frontier state of some sort.
        v. Thus, graph search satisfies the separation property described in Figure 3.9
    b. Name an algorithm that doesn't work
        i. An algorithm that starts in an initial node but doesn't expand the frontier to all of the initial nodes' connected nodes will not satisfy the separation property
        ii. For example:
            1. a ← **B** → C (where '**B'** is an explored node, 'C' is frontier node, and 'a' is an unexplored node)
            2. Thus, it is possible for the agent to move into an unexplored state without first passing into a frontier node.
            3. Thus, this agent would not satisfy the separation property described in Figure 3.9.
2. Chapter 3, Ex. 3.16
    a. Formulation of task as a search problem
        i. Initial State: Any piece
        ii. Possible Actions: Connect a piece to another
        iii. Transition Model: Connect an unconnected piece to a compatible piece in the track
        iv. Goal Test: All pieces are in the track, the track does not overlap on itself, and there are no openings in the track
        v. Path Cost Function: The number of pieces used, all pieces weighed the same
    b. Identify a suitable uninformed search algorithm
        i. I will rule out iterative and breadth first (and uniform-cost) searches because of the size of the state space
        ii. No backtracking search because it is not complete in this situation (the track will loop on itself)
        iii. No depth-limited or iterative deepening DFS because the state space is still finite
        iv. That leaves regular DFS or bidirectional DFS
            1. However, a bidirectional search is only complete if BFS is used in both searches, which we would not use for the reasons stated above
        v. Thus, I think DFS is the best suited algorithm

c. Explain why removing any one of the "fork" pieces makes the problem unsolvable
    i. If a fork piece is present, the only way to recombine the split tracks would be to have another fork piece. Thus, the number of fork pieces must be even. If a fork piece is removed it would make the number of fork pieces be odd, which would make the problem unsolvable as there would have to be an open-ended track.
d. Give upper bound on total size of the state space defined by your formulation
    i. The maximum amount of open pegs is 3 for any given track (e.g. when we use a "fork" track piece in the initial state)
    ii. There are a total of (12 + 2*16 + 2*2 + 2*2*2) = 56 combination of remaining pieces
    iii. Multiply 56*3 = 168 different combinations for the three pegs
    iv. So the maximum number of combinations for the whole state space is:
        1. (Total combination of all pieces)/(all different possible permutations)
        2. 168^32 / (12! * 16! * 2! * 2!)

3. Chapter 3, Ex. 3.26
    a. Branching factor b?
        i. 4 - there are 4 edges for each node
    b. Distinct states at depth k (k > 0)?
        i. 4k (1 node there are 4 states, 2 there are 8, as can be seen in the figure)
    c. Maximum number of nodes expanded by breadth-first tree search?
        i. BF tree search goes level by level so the maximum number of nodes would be 4^{x+y} (branching factor to the power of the depth)
    d. Maximum number of nodes expanded by BF graph search?
        i. BF graph search doesn't visit the same state twice so it should be the same as the number of distinct states so 4(x+y)
    e. Admissible heuristic?
        i. It is, as the heuristic provided represents the minimum amount of moves it takes to get from the origin state to the goal state since the problem only allows movement in the X and Y directions.
    f. Nodes expanded by A* using h?
        i. As h represents the minimum moves from the origin to the goal and the path cost between all nodes are the same, A* will open X+Y nodes (assuming the origin is at (0,0)
    g. Does h remain admissible if some links are removed?
        i. Yes – removing links will increase the path length and thus make h an underestimate (but still valid) heuristic
    h. Does h remain admissible if some links are added?
        i. No – adding links might decrease the path length and thus make h an overestimate (and thus invalid) heuristic

4. Chapter 4, Ex. 4.3

```
-- Randomized Initial State --
1 6 3 4 2 5 1
26
-- Found Either Local or Global Maximum --
1 2 3 4 6 5 1
Dist: 18
```

a.

```
-- Creating Randomized Initial Population --
Population Size: 10
-- Generational Settings --
Tournament Selection Number:            8
Number of Mutations per Generation:     2
Number of Pops Kept with Mutation:      4
Number of Generations:                  10
1 3 2 6 4 5 1
24
---
1 3 2 6 4 5 1
24
---
1 2 3 6 4 5 1
19
---
1 2 3 6 4 5 1
19
---
1 2 3 6 4 5 1
19
---
1 2 3 6 4 5 1
19
---
1 2 3 6 4 5 1
19
---
1 2 3 4 6 5 1
18
---
1 2 3 4 6 5 1
18
---
1 2 3 4 6 5 1
18
---

Optimal Path:
1 2 3 4 6 5 1
Total Distance: 18
```

b.

5. Chapter 4, Ex. 4.5
    a. We implement something similar to memoization in which every time OR-SEARCH finds a solution it records said solution in such a way that it can be found when returning to the corresponding state (e.g. solution label stored inside state, solution stored in hashmap with the state as the key, etc.). Then, whenever we return to the state we can return the label we found instead of having to run through everything again.