4) We use the k-d tree data structure for our implementation. A k-d tree is essentially a binary tree for storing data with multiple fields, which in this case would be the 2 or 3 dimensions of the prisms. We store the point with the median x value in the root, then put all points with x value < x_{root} in the left subtree, and all points with x value > x_{root} in the right subtree. For the next level, we sort by the y value, then the z value, and we rotate between the 3 for each level of the tree. We store all the points that would form the bounding box which is the maximum x, y, and z values contained in the list of points, so (x_{min}, z_{min}), ($x_{min,}$ z_{max}), (x_{max}, z_{min}), and ($x_{max,}$ z_{max}) for 2D, and the same points with y_{min} and y_{max} for 3D. We use the sort and sweep algorithm on all the dimensions simultaneously, so if another point of prism P's bounding box is checked after the first, it marks a potential collision with prism P and every prism added after it. Tree data structures have height O(logn), and the number of collision checks at each point cannot exceed O(n), which results in an O(nlogn) implementation.

7. a) We used the k-d tree data structure, implemented it from scratch, and implemented it into PotentialCollisions.

c) This was done by making an expansion of the 2D implementation of the GJK algorithm. The 2D implementation was done by checking in a line, then checking in a triangle. In 3D, we also have to check in a tetrahedron. So, instead of finishing the search at the triangle stage, we add another point to the simplex so that there is a tetrahedron to search. At the point where the search would end in 2D, we also have to check up and down, so the algorithm reflects this. We just searched one of the triangles at the point of reaching the tetrahedron, so we only have to test for the origin in the direction of one of the remaining sides of the tetrahedron, and if it is there, we go back to doing a triangle search. Otherwise, we know the origin is inside the tetrahedron and can return true.

d) The EPA implemented for 2D also works for 3D. The reason is that the logic of the algorithm applies for both 2D and 3D because of the way we are finding the penetration depth vector. In both versions, we are looking for the point on the polytope closest to the origin. Next, we see how much the vector moved since finding the closest point to the origin, and then finally we add a supporting point from the Minkowski difference to the polytope in the direction of the closest point to the origin. All of the above operations function the same in 2D and 3D. Therefore, all we had to do was make the supporting functions compatible for Vector3s instead of just Vector2s and the EPA works.

Link to demonstration:
https://www.youtube.com/watch?v=eUPz_UaYyOQ