1. You are given an implementation of a search algorithm (astar, rrt, etc.) with api as below. Write a set of unit tests (in c++, eg. using gtest) which validate the functionality. If required, propose any extensions or changes to the structure.

2. Suppose that the Problem class is further extended with a MultiRobotProblem class. In this case, how would you change the api for SearchAlgorithm, and how would you modify your testing approach?

3. Explain the a-star algorithm (no math required), and propose how you may extend it for multiple robots.

```cpp
class Pose2 {
public:
   /**
     * @brief Construct a pose object.
     * @param[in] x The x-location.
     * @param[in] y The y-location.
     * @param[in] theta The angle in radians.
     */
    Pose2(float x, float y, float theta);

    /**
     * @brief Get the x-coordinate.
     */
    auto getX() const -> float;

    /**
     * @brief Get the y-coordinate.
     */
    auto getY() const -> float;

    /**
     * @brief Get the angle in radians.
     */
    auto getTheta() const -> float;

protected:
    float x;
    float y;
    float theta;

};

using IsObstructed = std::function<bool(float,float)>;
enum class LogLevel { DEBUG, INFO, ERROR, CRITICAL };
```

```
class Problem {
public:
    enum class ProblemType { PLAN, REFINE, REPLAN, UNKNOWN };

    Problem() = delete;
    Problem(ProblemType prob_type, LogLevel level = LogLevel::INFO);

    /**
     * @brief Check whether the problem is valid or not.
     * @param[in] is_obstructed A function “pointer” which returns whether a
location is obstructed.
     * @return True if the problem is a valid one, false otherwise.
     */
    virtual auto checkValidity(IsObstructed &is_obstructed) -> bool;

    ProblemType problem_type = ProblemType::UNKNOWN;
    LogLevel log_level       = LogLevel::INFO;
};

class SingleRobotProblem : public Problem {
public:
    SingleRobotProblem() = delete;
    SingleRobotProblem(
        Pose2 start_pose,
        Pose2 goal_pose,
        ProblemType prob_type,
        LogLevel level = LogLevel::INFO);

    /**
     * @brief Check whether the problem is valid or not.
     * @details This function validates that start and goal are not obstructed.
     * @param[in] is_obstructed A function “pointer” which returns whether a
     * location is obstructed.
     * @return True if the problem is a valid one, false otherwise.
     */
    bool checkValidity(IsObstructed &is_obstructed) const;

    Pose2 start;
    Pose2 goal;
};

class SearchAlgorithm {
public:
    /**
     * @brief Reset the search using a new problem.
```

```
     * @param[in] problem The problem to reset with.
     */
    void reset(const Problem &problem);

    /**
     * @brief Update the search using a new problem.
     * @details This differs from reset() in that the current solution
     * and search space are not reset.
     * @param[in] problem The problem to update with.
     */
    void update(const Problem &problem);

    /**
     * @brief Check if the search algorithm has found a solution.
     * @return True if found, false otherwise.
     */
    auto isSolutionFound() const -> bool;

    /**
     * @brief Get the resulting obstruction-free plan from the search algorithm.
     */
    auto getPlan() const -> std::vector<Pose2>;

    /**
     * @brief Run the algorithm until successful.
     */
    void plan();

    /**
     * @brief Replan the algorithm, by re-using the current problem and solution.
     * @details This may be called in place of plan when a new obstruction
     * is encountered along the resulting path.
     */
    void replan();

    /**
     * @brief Refine the current solution to get a better result than
     * the current one.
     */
    void refine();

protected:
    void runOneIteration();
    void runUntilComplete();
};
```