 Answers to review questions from Chapter 9

1. What is the principal characteristic of a backtracking algorithm?

**A backtracking algorithm is one that solves a problem consisting of a sequence of choice points from which it is necessary to explore each path. The solution uses recursion to keep track of each choice point to make sure that all options have been explored.**

2. Using your own words, state the right‑hand rule for escaping from a maze. Would a left‑hand rule work equally well?

**When you apply the right‑hand rule, you put your right hand on a wall and make sure that you never remove your hand from that wall, following the path around intersections as necessary. The corresponding left‑hand rule would work equally well. Both strategies can fail if the maze includes loops that surround either the starting position or the goal.**

3. What is the insight that makes it possible to solve a maze by recursive backtracking?

**If you take one step in each possible direction and then mark the square at which you started, the original maze is solvable if and only if one of the new mazes is solvable. Since each of these new mazes has one additional marked square, it represents a simpler instance of the problem than the original maze.**

4. What are the simple cases that apply in the recursive implementation of **solveMaze**?

**If you are outside the maze, you’ve succeeded.**

**If you are at a marked square that you’ve previously visited, you’ve failed.**

5. Why is important to mark squares as you proceed through the maze? What would happen in the **solveMaze** function if you never marked any squares?

**If you failed to mark any squares, there would be nothing to prevent you from circling endlessly around a loop within the maze.**

6. What is the purpose of the **unmarkSquare** call at the end of the **for** loop in the **solveMaze** implementation? Is this statement essential to the algorithm?

**Unmarking squares as you backtrack means that the correct path is marked when you reach the exit. If you leave out this step, the algorithm will still solve the maze, but some squares are likely to be marked even if they are not on the exit path.**

7. What is the purpose of the Boolean result returned by **solveMaze**?

**The result indicates whether the maze is solvable. In the recursive subdivision of the problem, many derived mazes will be unsolvable even if the entire maze is solvable.**

8. In your own words, explain how the backtracking process actually takes place in the recursive implementation of **solveMaze**.

**At each square in the maze, the algorithm first checks to see whether the square is outside the maze or marked, both of which represent simple cases in the algorithm. If neither of these cases apply, the code then marks the current square and tries to solve the maze after moving one step in each open direction. If that direction leads to a solution, the maze is solvable; if not, the code moves on to the next direction. If all directions have been attempted, the code reports failure, which is then passed back to a higher level of the recursive process.**

9. In the simple Nim game, the human player plays first and begins with a pile of 13 coins. Is this a good or a bad position? Why?

**Thirteen coins is a bad position because there are no good moves. No matter how many you take, the computer can take some number of coins that leaves a pile with nine coins, which is a bad position.**

10. Write a simple C++ expression based on the value of **nCoins** that has the value **true** if the position is good for the current player and **false** otherwise.

nCoins % 4 != 1

11. What is the minimax algorithm? What does its name signify?

**The *minimax algorithm* is a general strategy for two‑player games in which you choose the move that leaves your opponent with the worst possible best move (the minimum of the maximum).**

12. Why is it useful to develop an abstract implementation of the minimax algorithm that does not depend on the details of a particular game?

**Creating an abstract implementation means that you can use the same strategy code for a variety of games and don’t need to recode that strategy for each one.**

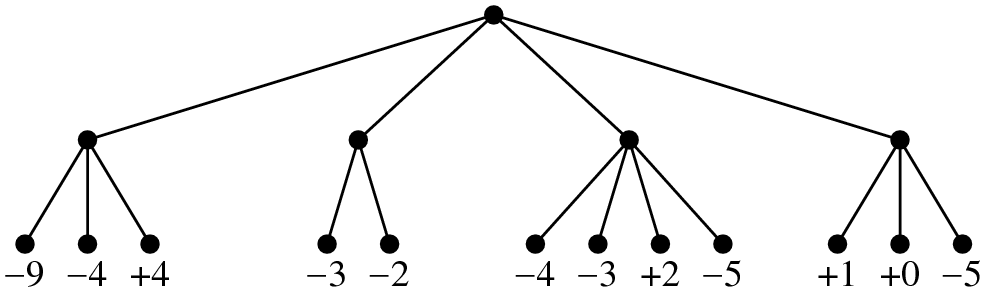
13. What is the role of the **depth** argument in the functions **findBestMove** and **evaluatePosition**?

**The depth argument is used to limit the extent of the search to a maximum number of moves. This argument is necessary if solving the game all the way to the end would be so slow as to make the game unplayable.**

14. Explain the role of the **evaluateStaticPosition** function in the minimax implementation.

**Once you have explored all game paths to the maximum depth, the evaluateStaticPosition function allows you to analyze the final position to determine how good it is for the player to move.**

15. Suppose you are in a position in which the analysis for the next two moves shows the following rated outcomes from your original player’s point‑of‑view:



If you adopt the minimax strategy, what is the best move to make in this position? What is the rating of that move from your perspective?

